



Дж. Рамбо, М. Блаха

UML 2.0

**Объектно-ориентированное
моделирование и разработка
2-е издание**

-  Объектно-ориентированные концепции
-  Анализ и проектирование объектно-ориентированной модели
-  Реализация проектных решений
-  Технологии разработки программного обеспечения

J. R. Rumbaugh, M. R. Blaha

Object-Oriented Modeling and Design with UML

2nd Edition





БИБЛИОТЕКА ПРОГРАММИСТА

Дж. Рамбо, М. Блаха

UML 2.0

**Объектно-ориентированное
моделирование и разработка**

2-е издание



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск

2007

Дж. Рамбо, М. Блах

UML 2.0. Объектно-ориентированное моделирование и разработка

2-е издание

Заведующий редакцией
Ведущий редактор
Научный редактор
Литературные редакторы
Иллюстрации
Художник
Корректор
Верстка

*А. Кривцов
А. Крузенигер
Ф. Новиков
Е. Бурнашова, В. Шрага
Г. Андреева, Л. Родионова, С. Романов
Л. Адуевская
В. Листова
Р. Гришанов*

ББК 32.973-018.1
УДК 004.43

Дж. Рамбо, М. Блах

P21 UML 2.0. Объектно-ориентированное моделирование и разработка.
2-е изд. — СПб.: Питер, 2007. — 544 с.: ил.

ISBN 5-469-00814-2

Перед вами поистине революционная книга, посвященная базовым принципам объектно-ориентированного мышления. Своей универсальностью она выгодно отличается от множества книг, описывающих отличительные черты какого-нибудь одного языка программирования.

Новое издание этого бестселлера обновлено в соответствии со стандартом UML 2.0. Авторы четко и ясно объясняют суть важнейших концепций объектно-ориентированного программирования, представляют способы реализации этих идей при разработке ПО с использованием языков C++ и Java, а также реляционных баз данных. В книге есть задания и множество советов, что делает ее очень практичной.

© 2005, 1991 by Pearson Education, Inc.

© Перевод на русский язык, ООО «Питер Пресс», 2007

© Издание на русском языке, оформление, ООО «Питер Пресс», 2007

Права на издание получены по соглашению с Pearson Education Inc.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 5-469-00814-2

ISBN 0130159204 (англ.)

ООО «Питер Пресс», 198206, Санкт-Петербург, Петергофское шоссе, д. 73, лит. А29.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Подписано в печать 10.08.06. Формат 70×100/16. Усл. п. л. 43,86. Тираж 2000. Заказ 699

Отпечатано с готовых диапозитивов в ОАО «Техническая книга»

190005, Санкт-Петербург, Измайловский пр., 29

Краткое содержание

От издательства	16

Глава 1. Введение	17

Часть I. Концепции моделирования

Глава 2. Моделирование как методика проектирования	34

Глава 3. Моделирование классов	42

Глава 4. Углубленное моделирование классов	86

Глава 5. Моделирование состояний	116

Глава 6. Углубленное моделирование состояний	138

Глава 7. Моделирование взаимодействий	161

Глава 8. Дополнительные вопросы моделирования взаимодействий	180

Глава 9. Обзор концепций	197

Часть II. Анализ и проектирование

Глава 10. Обзор процесса разработки	202

Глава 11. Концептуализация системы	209

Глава 12. Анализ предметной области	218

Глава 13. Анализ приложения	259

Глава 14. Проектирование системы	286

Глава 15. Проектирование классов	323

Глава 16. Резюме процесса разработки	356

Часть III. Реализация

Глава 17. Моделирование реализации	360

Глава 18. Объектно-ориентированные языки	372

Глава 19. Базы данных	409

Глава 20. Стиль программирования	440

Часть IV. Разработка программного обеспечения

Глава 21. Итерационная разработка	456

Глава 22. Управление моделированием	465

Глава 23. Унаследованные системы	480

Приложение А. Система графических обозначений UML	490

Приложение Б. Краткий словарь	491

Ответы к избранным упражнениям	508

Алфавитный указатель	538

Содержание

От издательства	16
Глава 1. Введение	17
1.1. Что такое объектная ориентированность?	17
1.2. Объектно-ориентированная разработка	20
1.2.1. Моделирование концепций, а не реализации	20
1.2.2. Объектно-ориентированная методология	21
1.2.3. Три модели	23
1.3. Объектно-ориентированные концепции	24
1.3.1. Абстракция	24
1.3.2. Инкапсуляция	24
1.3.3. Объединение данных и поведения	24
1.3.4. Совместное использование	25
1.3.5. Выделение сущности объекта	25
1.3.6. Когда целое больше суммы частей	26
1.4. Доводы в пользу объектной ориентированности	26
1.5. История объектно-ориентированного моделирования	27
1.6. Структура книги	28
Библиографические заметки	29
Ссылки	30
Упражнения	30
 Часть I. Концепции моделирования	
Глава 2. Моделирование как методика проектирования	34
2.1. Моделирование	34
2.2. Абстрагирование	35
2.3. Три модели	36
2.3.1. Модель классов	36
2.3.2. Модель состояний	37
2.3.3. Модель взаимодействия	37
2.3.4. Отношения моделей	37
2.4. Резюме	38
Библиографические замечания	38
Упражнения	39
 Глава 3. Моделирование классов	42
3.1. Концепции объекта и класса	42
3.1.1. Объекты	42
3.1.2. Классы	43
3.1.3. Диаграммы классов	44
3.1.4. Значения и атрибуты	45
3.1.5. Операции и методы	46
3.1.6. Резюме системы обозначений классов	48

3.2. Концепции связи и ассоциации	49
3.2.1. Связи и ассоциации	49
3.2.2. Кратность	51
3.2.3. Имена полюсов ассоциации	53
3.2.4. Упорядочение	55
3.2.5. Мультимножества и последовательности	55
3.2.6. Классы ассоциаций	56
3.2.7. Квалифицированные ассоциации	58
3.3. Обобщение и наследование	59
3.3.1. Определения	59
3.3.2. Использование обобщения	62
3.3.3. Подмена составляющих	63
3.4. Пример модели классов	63
3.5. Навигация моделей классов	66
3.5.1. Прослеживание моделей с помощью конструкций OCL	67
3.5.2. Построение выражений OCL	68
3.5.3. Примеры выражений OCL	69
3.6. Практические советы	71
3.7. Резюме по моделям классов	73
Библиографические заметки	74
Ссылки	75
Упражнения	76

Глава 4. Углубленное моделирование классов 86

4.1. Расширенные концепции классов и объектов	86
4.1.1. Перечисление	86
4.1.2. Кратность	87
4.1.3. Область действия	88
4.1.4. Видимость	89
4.2. Полюса ассоциаций	90
4.3. N-арные ассоциации	91
4.4. Агрегация	93
4.4.1. Агрегация и ассоциация	93
4.4.2. Агрегация и композиция	94
4.4.3. Распространение операций	95
4.5. Абстрактные классы	95
4.6. Множественное наследование	97
4.6.1. Виды множественного наследования	97
4.6.2. Множественная классификация	99
4.6.3. Обходные маневры	99
4.7. Метаданные	101
4.8. Воплощение	102
4.9. Ограничения	103
4.9.1. Ограничения на объекты	103
4.9.2. Ограничения на наборы обобщений	104
4.9.3. Ограничения на связи	104
4.9.4. Использование ограничений	105
4.10. Производные данные	105
4.11. Пакеты	106
4.12. Практические рекомендации	107
4.13. Резюме	108
Библиографические замечания	110
Ссылки	110
Упражнения	110

Глава 5. Моделирование состояний	116
5.1. События	116
5.1.1. Событие сигнала	117
5.1.2. События изменения	118
5.1.3. События времени	118
5.2. Состояния	119
5.3. Переходы и условия	121
5.4. Диаграммы состояний	122
5.4.1. Пример диаграммы состояний	122
5.4.2. Одноразовые диаграммы состояний	124
5.4.3. Основные обозначения для диаграмм состояний	125
5.5. Поведение на диаграммах состояний	126
5.5.1. Действия и деятельность	126
5.5.2. Текущая деятельность	126
5.5.3. Деятельность при входе и при выходе	127
5.5.4. Переход по завершении	128
5.5.5. Отправка сигналов	129
5.5.6. Пример диаграммы состояний с деятельностью	129
5.6. Практические рекомендации	129
5.7. Резюме	131
Библиографические замечания	132
Ссылки	133
Упражнения	133
Глава 6. Углубленное моделирование состояний	138
6.1. Вложенные диаграммы состояний	138
6.1.1. Задачи с одноуровневыми диаграммами состояний	138
6.1.2. Разложение состояний	138
6.2. Вложенные состояния	139
6.3. Обобщение сигналов	142
6.4. Параллелизм	143
6.4.1. Параллелизм в агрегации	143
6.4.2. Параллелизм в объекте	144
6.4.3. Синхронизация параллельной деятельности	145
6.5. Пример модели состояний	147
6.6. Модель состояний и модель классов	152
6.7. Практические рекомендации	153
6.8. Резюме	154
Библиографические замечания	155
Ссылки	156
Упражнения	156
Глава 7. Моделирование взаимодействий	161
7.1. Модели вариантов использования	162
7.1.1. Действующие лица	162
7.1.2. Варианты использования	162
7.1.3. Диаграммы вариантов использования	165
7.1.4. Руководство к вариантам использования	166
7.2. Модели последовательности	167
7.2.1. Сценарии	167
7.2.2. Диаграммы последовательности	168
7.2.3. Руководство к диаграммам последовательности	171

7.3. Модели деятельности	171
7.3.1. Деятельность	172
7.3.2. Ветвление	173
7.3.3. Инициализация и завершение	174
7.3.4. Параллельная деятельность	174
7.3.5. Выполняемые диаграммы деятельности	175
7.3.6. Руководство к моделям деятельности	175
7.4. Резюме	176
Библиографические замечания	176
Ссылки	177
Упражнения	177
Глава 8. Дополнительные вопросы моделирования взаимодействий	180
8.1. Отношения вариантов использования	180
8.1.1. Отношение включения	180
8.1.2. Отношение расширения	181
8.1.3. Обобщение	182
8.1.4. Комбинации отношений вариантов использования	183
8.1.5. Руководство по применению отношений к вариантам использования	185
8.2. Процедурные модели последовательности	185
8.2.1. Диаграммы последовательности с пассивными объектами	186
8.2.2. Диаграммы последовательности с временными объектами	187
8.2.3. Руководство к процедурным моделям последовательности	188
8.3. Специальные конструкции для моделей деятельности	189
8.3.1. Отправка и получение сигналов	189
8.3.2. Плавательные дорожки	189
8.3.3. Потоки объектов	190
8.4. Резюме	191
Ссылки	192
Упражнения	192
Глава 9. Обзор концепций	197
9.1. Модель классов	197
9.2. Модель состояний	197
9.3. Модель взаимодействия	198
9.4. Отношения между моделями	198
9.4.1. Обобщение	199
9.4.2. Агрегация	200

Часть II. Анализ и проектирование

Глава 10. Обзор процесса разработки	202
10.1. Этапы разработки	202
10.1.1. Концептуализация системы	203
10.1.2. Анализ	203
10.1.3. Проектирование системы	204
10.1.4. Проектирование классов	205
10.1.5. Реализация	205
10.1.6. Тестирование	205
10.1.7. Обучение	206
10.1.8. Развертывание	206
10.1.9. Поддержка	206

10.2. Жизненный цикл разработки	206
10.2.1. Водопадная разработка	207
10.2.2. Итерационная разработка	207
10.3. Резюме	207
Библиографические заметки	208
Упражнения	208
Глава 11. Концептуализация системы	209
11.1. Изобретение концепции системы	209
11.2. Проработка концепции	210
11.3. Подготовка задачи к постановке	213
11.4. Резюме	215
Упражнения	216
Глава 12. Анализ предметной области	218
12.1. Обзор этапа анализа	218
12.2. Модель классов предметной области	220
12.2.1. Выделение классов	221
12.2.2. Удаление лишних классов	222
12.2.3. Подготовка словаря данных	224
12.2.4. Выделение ассоциаций	225
12.2.5. Удаление лишних ассоциаций	225
12.2.6. Выделение атрибутов	231
12.2.7. Удаление лишних атрибутов	232
12.2.8. Реструктурирование при помощи наследования	235
12.2.9. Проверка маршрутов	236
12.2.10. Итерационная разработка модели классов	237
12.2.11. Смещение уровня абстрагирования	240
12.2.12. Группировка классов в пакеты	241
12.3. Модель состояний предметной области	242
12.3.1. Выявление классов с разными состояниями	242
12.3.2. Выделение состояний	243
12.3.3. Выделение событий	243
12.3.4. Построение диаграмм состояний	244
12.3.5. Проверка диаграмм состояния	244
12.4. Модель взаимодействия предметной области	245
12.5. Итерационный анализ	245
12.5.1. Уточнение аналитической модели	246
12.5.2. Корректировка требований	246
12.5.3. Анализ и проектирование	247
12.6. Резюме	247
Библиографические замечания	248
Ссылки	248
Упражнения	248
Глава 13. Анализ приложения	259
13.1. Модель взаимодействия приложения	259
13.1.1. Определение границы системы	260
13.1.2. Идентификация действующих лиц	260
13.1.3. Идентификация вариантов использования	261
13.1.4. Идентификация начальных и конечных событий	262
13.1.5. Подготовка типовых сценариев	263
13.1.6. Нетипичные сценарии и исключительные ситуации	265

13.1.7. Выделение внешних событий	265
13.1.8. Подготовка диаграмм деятельности для сложных вариантов использования	267
13.1.9. Структурирование действующих лиц и вариантов использования	267
13.1.10. Проверка по модели классов предметной области	269
13.2. Модель классов приложения	269
13.2.1. Определение интерфейсов пользователя	269
13.2.2. Определение пограничных классов	270
13.2.3. Определение управляющих объектов	271
13.2.4. Проверка по модели взаимодействия	271
13.3. Модель состояний приложения	272
13.3.1. Выделение классов приложения	273
13.3.2. Поиск событий	273
13.3.3. Построение диаграмм состояний	273
13.3.4. Проверка по другим диаграммам состояний	275
13.3.5. Проверка по модели классов	277
13.3.6. Проверка по модели взаимодействия	278
13.4. Добавление операций	278
13.4.1. Операции из модели классов	278
13.4.2. Операции из вариантов использования	278
13.4.3. Операции «по списку»	279
13.4.4. Упрощение операций	279
13.5. Резюме	279
Библиографические замечания	281
Литература	281
Упражнения	282

Глава 14. Проектирование системы 286

14.1. Обзор проектирования систем	286
14.2. Оценка производительности	287
14.3. Планирование повторного использования	288
14.3.1. Библиотеки	288
14.3.2. Каркасы	290
14.3.3. Образцы	290
14.4. Разбиение системы на подсистемы	291
14.4.1. Уровни	292
14.4.2. Разделы	293
14.4.3. Сочетание уровней и разделов	293
14.5. Выделение параллелизма	294
14.5.1. Выделение неотъемлемой параллельности	294
14.5.2. Определение параллельных задач	295
14.6. Распределение подсистем	295
14.6.1. Оценка требований к аппаратным ресурсам	296
14.6.2. Выбор между аппаратным и программным обеспечением	296
14.6.3. Распределение задач по процессорам	297
14.6.4. Определение физической связности	298
14.7. Управление хранилищами данных	299
14.8. Распределение глобальных ресурсов	300
14.9. Выбор стратегии управления программным обеспечением	301
14.9.1. Процедурное управление	302
14.9.2. Событийное управление	302
14.9.3. Параллельное управление	303
14.9.4. Внутреннее управление	303
14.9.5. Другие парадигмы	304

14.10. Учет граничных условий	304
14.11. Установка приоритетов	305
14.12. Распространенные архитектурные стили	306
14.12.1. Пакетное преобразование	306
14.12.2. Непрерывное преобразование	307
14.12.3. Интерактивный интерфейс	308
14.12.4. Динамическое моделирование	309
14.12.5. Системы реального времени	310
14.12.6. Администратор транзакций	311
14.13. Архитектура сети банкоматов	311
14.14. Резюме	312
Библиографические замечания	314
Литература	315
Упражнения	315
Глава 15. Проектирование классов	323
15.1. Обзор этапа проектирования классов	323
15.2. Наведение мостов	324
15.3. Реализация вариантов использования	326
15.4. Проектирование алгоритмов	328
15.4.1. Выбор алгоритмов	328
15.4.2. Выбор структур данных	330
15.4.3. Определение внутренних классов и операций	330
15.4.4. Назначение операций классам	331
15.5. Рекурсия вниз	333
15.5.1. Уровни функциональности	334
15.5.2. Уровни механизмов	334
15.6. Реорганизация	335
15.7. Оптимизация проекта	336
15.7.1. Добавление избыточных ассоциаций для повышения эффективности доступа	336
15.7.2. Изменение порядка выполнения для повышения эффективности	338
15.7.3. Сохранение промежуточных результатов	339
15.8. Воплощение поведения	340
15.9. Корректировка иерархии наследования	341
15.9.1. Реорганизация классов и операций	341
15.9.2. Абстрагирование общего поведения	342
15.9.3. Делегирование	344
15.10. Организация проекта модели классов	345
15.10.1. Соккрытие информации	345
15.10.2. Согласованность сущностей	346
15.10.3. Коррекция определений пакетов	347
15.12. Резюме	349
Библиографические замечания	350
Литература	350
Упражнения	351
Глава 16. Резюме процесса разработки	356
16.1. Концептуализация системы	357
16.2. Анализ	357
16.2.1. Анализ предметной области	357
16.2.2. Анализ приложения	357
16.3. Проектирование	358
16.3.1. Проектирование системы	358
16.3.2. Проектирование классов	358

Часть III. Реализация

Глава 17. Моделирование реализации	360
17.1. Обзор реализации	360
17.2. Уточнение классов	361
17.3. Уточнение обобщений	362
17.4. Реализация ассоциаций	363
17.4.1. Анализ прослеживания ассоциаций	365
17.4.2. Односторонние ассоциации	365
17.4.3. Двусторонние ассоциации	365
17.4.4. Сложные ассоциации	367
17.5. Тестирование	367
17.5.1. Модульное тестирование	368
17.5.2. Системное тестирование	369
17.6. Резюме	369
Библиографические замечания	370
Литература	370
Упражнения	370
Глава 18. Объектно-ориентированные языки	372
18.1. Введение	372
18.1.1. Введение в C++	372
18.1.2. Введение в Java	374
18.1.3. Сравнение C++ и Java	376
18.2. Сокращенная модель банкомата	377
18.3. Реализация структуры	378
18.3.1. Типы данных	378
18.3.2. Классы	381
18.3.3. Управление доступом	381
18.3.4. Обобщение	384
18.3.5. Ассоциации	388
18.4. Реализация функциональности	391
18.4.1. Создание объекта	392
18.4.2. Существование объекта	395
18.4.3. Уничтожение объекта	396
18.4.4. Создание связи	397
18.4.5. Удаление связи	399
18.4.6. Производные атрибуты	400
18.5. Практические рекомендации	401
18.6. Резюме	402
Библиографические замечания	403
Литература	403
Упражнения	404
Глава 19. Базы данных	409
19.1. Введение	409
19.1.1. Концепции баз данных	409
19.1.2. Концепции реляционных баз данных	410
19.1.3. Нормальные формы	412
19.1.4. Выбор СУБД	413
19.2. Сокращенная модель банкомата	413
19.3. Реализация структуры — основы	414
19.3.1. Классы	414
19.3.2. Ассоциации	416

19.3.3. Обобщения	418
19.3.4. Индивидуальность	419
19.3.5. Основные правила реализации РСУБД	420
19.4. Реализация структуры — дополнительные вопросы	421
19.4.1. Внешние ключи	421
19.4.2. Проверка ограничений	423
19.4.3. Индексы	423
19.4.4. Представления	423
19.4.5. Дополнительные правила реализации моделей UML в РСУБД	424
19.5. Реализация структуры из примера с банкоматом	424
19.6. Реализация функциональности	428
19.6.1. Связь языка программирования с базой данных	428
19.6.2. Преобразование данных	430
19.6.3. Инкапсуляция и оптимизация запросов	431
19.6.4. Использование кода SQL	432
19.7. Объектно-ориентированные базы данных	432
19.8. Практические рекомендации	433
19.9. Резюме	434
Библиографические замечания	435
Литература	435
Упражнения	436
Глава 20. Стиль программирования	440
20.1. Объектно-ориентированный стиль	440
20.2. Повторное использование	441
20.2.1. Виды повторного использования	441
20.2.2. Правила хорошего стиля для повторного использования	441
20.2.3. Использование наследования	443
20.3. Возможность расширения	445
20.4. Устойчивость	446
20.5. Программирование крупных систем	448
20.6. Резюме	451
Библиографические замечания	452
Литература	452
Упражнения	453
 Часть IV. Разработка программного обеспечения	
Глава 21. Итерационная разработка	456
21.1. Обзор итерационной разработки	456
21.2. Итерационная и водопадная модели	456
21.3. Итерационная разработка и быстрое прототипирование	457
21.4. Масштаб итераций	458
21.5. Выполнение итерации	459
21.6. Планирование следующей итерации	460
21.7. Моделирование и итерационная разработка	461
21.8. Идентификация рисков	462
21.9. Резюме	463
Библиографические замечания	464
Литература	464

Глава 22. Управление моделированием	465
22.1. Обзор управления моделированием	465
22.2. Виды моделей	465
22.3. Ловушки моделирования	466
22.4. Сеансы моделирования	468
22.4.1. Скрытое моделирование	469
22.4.2. Циклическое моделирование	469
22.4.3. Моделирование на месте	470
22.5. Организация персонала	472
22.6. Методики изучения	473
22.7. Методики обучения	474
22.8. Средства	475
22.8.1. Средства моделирования	475
22.8.2. Средства управления конфигурациями	476
22.8.3. Генераторы кода	476
22.8.4. Средства интерпретации моделей	477
22.8.5. Репозиторий	477
22.9. Оценка затрат на моделирование	477
22.10. Резюме	478
Библиографические замечания	478
Литература	479
Глава 23. Унаследованные системы	480
23.1. Инженерный анализ	480
23.1.1. Инженерный анализ и разработка	481
23.1.2. Входные данные для инженерного анализа	481
23.1.3. Выходные данные инженерного анализа	482
23.2. Построение модели классов	482
23.2.1. Восстановление реализации	483
23.2.2. Восстановление проекта	483
23.2.3. Восстановление анализа	483
23.3. Построение модели взаимодействия	484
23.4. Построение модели состояний	484
23.5. Рекомендации по проведению инженерного анализа	485
23.6. Обертка	486
23.7. Обслуживание	487
23.8. Резюме	487
Библиографические замечания	488
Литература	488
Приложение А. Система графических обозначений UML	490
Приложение Б. Краткий словарь	491
Ответы к избранным упражнениям	508
Алфавитный указатель	538

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на веб-сайте издательства: <http://www.piter.com>.

Введение

Объектно-ориентированное моделирование и проектирование — это подход к решению задач с использованием моделей, основанных на понятиях реального мира. Фундаментальным элементом является объект, объединяющий структуру данных с поведением. Объектно-ориентированные модели полезны для понимания задач, взаимодействия с экспертами по работе с приложениями, моделирования предприятий, подготовки документации и проектирования программ и баз данных. Эта книга предлагает объектно-ориентированную систему обозначений и объектно-ориентированный процесс разработки программного обеспечения, которые могут использоваться на всех этапах, начиная с анализа и заканчивая проектированием и реализацией.

1.1. Что такое объектная ориентированность?

Объектная ориентированность в простейшем смысле означает представление программного обеспечения в виде дискретных объектов, содержащих в себе структуры данных и поведение. До появления объектно-ориентированного подхода структуры данных и поведение были связаны между собой очень слабо. Характеристики объектно-ориентированного подхода до сих пор являются предметом обсуждения, но, как правило, речь идет об индивидуальности, классификации, наследовании и полиморфизме.

Индивидуальность (identity) означает, что данные делятся на дискретные сущности, хорошо отличимые друг от друга. Эти сущности называются *объектами* (objects). Примерами объектов являются *первый абзац этой главы*, *мой компьютер* и *белый ферзь в шахматах*. На рис. 1.1 приведены другие примеры объектов. Объекты могут быть конкретными, как, например, *файл* в составе файловой системы, или концептуальным, как *политика планирования* в многопроцессорной

операционной системе. Каждый объект обладает своей собственной внутренней индивидуальностью. Другими словами, два объекта различимы даже в том случае, если значения всех их атрибутов (таких как имя или размер) одинаковы.

В реальном мире объекты просто существуют. В языках программирования каждый объект обладает уникальным абстрактным идентификатором (handle), посредством которого осуществляются ссылки на данный объект. Абстрактный идентификатор может быть реализован по-разному в разных языках, например, как адрес ячейки памяти, индекс массива или произвольное число. Ссылки на объекты должны быть однотипными и не зависящими от содержимого объектов, что позволяет создавать смешанные совокупности объектов. Например, каталог файловой системы содержит как файлы, так и подкаталоги.



Рис. 1.1. Объекты — фундамент объектно-ориентированных технологий

Классификация (classification) означает, что объекты с одинаковыми структурами данных (атрибутами) и поведением (операциями) группируются в классы. *Абзац*, *Монитор* и *Шахматная Фигура* — это примеры классов. **Класс** (class) — это абстракция, описывающая свойства, важные для конкретного приложения, и игнорирующая все остальные. Любой выбор классов произволен и зависит от приложения.

Каждый класс описывает множество индивидуальных объектов, которое может быть бесконечным. Каждый объект из этого множества называется *экземпляром* класса (instance). Объект имеет свои собственные значения атрибутов, но названия атрибутов и операции являются общими для всех экземпляров класса. На рис. 1.2 изображены два класса и их экземпляры. Объект всегда содержит неявную ссылку на свой класс, он «знает, чем является».

Наследование (inheritance) — это наличие у разных классов, образующих иерархию, общих атрибутов и операций (*составляющих*). **Суперкласс** задает наиболее общую информацию, которую затем уточняют и улучшают его *подклассы*. Каждый подкласс соединяет в себе, то есть наследует, все черты его суперкласса, к которым добавляет собственные уникальные черты. Подклассам не обязательно воспроизводить все черты суперкласса. Например, классы *ScrollingWindow* (ОкноСПрокруткой)

и *FixedWindow* (ФиксированноеОкно) являются подклассами класса *Window* (Окно). Оба подкласса наследуют черты класса *Window* (например, наличие видимой области на экране). *ScrollingWindow* добавляет полосу прокрутки и смещение. Возможность выделять общие черты нескольких классов в суперкласс значительно сокращает количество повторений в проектах и программах и является одним из основных достоинств объектно-ориентированной технологии.

Полиморфизм (polymorphism) означает, что одна и та же операция может подразумевать разное поведение в разных классах. Например, в шахматах операция ход для пешки и для ферзя характеризуется разным поведением. *Операция* (operation) — это процедура или трансформация, которую объект выполняет сам или которая осуществляется с данным объектом. Примерами операций являются *выравниваниеВправо*, *отображение* и *ход*. Реализация операции в конкретном классе называется *методом* (method). Поскольку объектно-ориентированная операция является полиморфной, в разных классах объектов она может быть реализована разными методами.

В реальном мире операция является абстрагированием похожего поведения у объектов одного рода. Каждый объект «сам знает», как выполнить свои собственные операции. В объектно-ориентированных языках программирования выбор подходящего метода для реализации операции осуществляется автоматически, исходя из имени операции и класса объекта, к которому она относится. Клиенту операции нет необходимости знать о том, сколько еще методов могут реализовывать данную полиморфическую операцию. Разработчики могут добавлять новые классы, не меняя существующий код, при условии, что они предоставляют методы для каждой возможной операции.

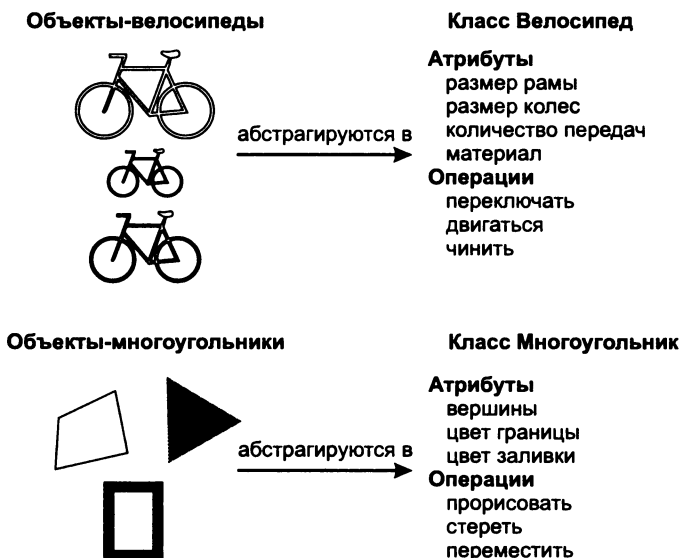


Рис. 1.2. Объекты и классы: каждый класс описывает множество индивидуальных объектов

1.2. Объектно-ориентированная разработка

Эта книга посвящена объектно-ориентированной разработке программного обеспечения как способу мышления, основанному на абстракциях, существующих как в реальном мире, так и в программах. В этом контексте *разработка* (development) обозначает жизненный цикл программного обеспечения: анализ, проектирование и реализацию. Целью объектно-ориентированной разработки является идентификация и упорядочение концепций приложения, а не окончательная реализация на языке программирования. Брукс заключает, что наиболее сложной частью разработки программного обеспечения является работа с его *сущностью* (essence), а не *акцидентией* (accidents) отображения этой сущности в конкретный язык программирования [Brooks-95].

Наша книга не содержит непосредственного описания проблем, связанных с интеграцией, обслуживанием и усовершенствованием программного обеспечения. Однако ясный проект в четкой системе обозначений упрощает все этапы жизненного цикла программного обеспечения. Объектно-ориентированные концепции и система обозначений, используемые для выражения проекта, также оказываются полезными при написании документации.

1.2.1. Моделирование концепций, а не реализации

В прошлом объектно-ориентированное сообщество занималось, главным образом, языками программирования. В литературе исследовались вопросы реализации, а не анализа и проектирования. Объектно-ориентированные языки программирования были полезны тем, что они обладали гибкостью, нехарактерной для традиционных языков программирования. Однако для технологий разработки программного обеспечения это был шаг назад, поскольку все внимание уделялось механизмам реализации, а не лежащим в основе мыслительным процессам.

Гораздо большего выигрыша можно достичь, если сосредоточиться на концептуальных вопросах переднего плана, а не на деталях реализации. Недостатки проекта, всплывающие в процессе реализации, стоят дороже, чем те, которые обнаруживаются раньше. Слишком ранний переход к реализации ограничивает возможные варианты представления проекта, а потому часто приводит к снижению качества продукта. Объектно-ориентированный подход к разработке поощряет разработчиков работать и мыслить в терминах приложения на протяжении всего жизненного цикла программного продукта. Эффективное решение проблем, связанных со структурами данных и функциями, может быть осуществлено только после идентификации, упорядочения и постижения внутренних концепций приложения.

Объектно-ориентированная разработка — это концептуальный процесс, независимый от языка программирования, по крайней мере, до последних этапов. Фактически это образ мышления, а не методика программирования. Главное преимущество объектной ориентированности состоит в том, что она помогает тем, кто

пишет спецификации, разработчикам и заказчикам ясно выражать абстрактные концепции и обсуждать их друг с другом. Таким образом облегчается составление спецификаций, анализ, документирование и определение интерфейсов и, конечно же, программирование.

1.2.2. Объектно-ориентированная методология

В этом разделе мы опишем процесс объектно-ориентированной разработки и системы графических обозначений для объектно-ориентированных концепций. Процесс состоит из построения модели приложения и последующей ее детализации. Одна и та же система обозначений используется на всем протяжении процесса разработки, начиная с анализа и заканчивая проектированием и реализацией. Информация, добавленная на одном из этапов, не будет утеряна или даже преобразована при переходе к следующему этапу. Описываемая методология включает следующие этапы.

1. **Концептуализация системы.** Разработка программного обеспечения начинается с бизнес-аналитиков или пользователей, которые придумывают приложение и формулируют первичные требования к нему.
2. **Анализ.** Аналитик тщательно исследует и безжалостно переформулирует требования, конструируя модели, исходя из концепций системы. Аналитик должен работать с заказчиком, чтобы добиться понимания задачи, потому что формулировки редко оказываются полными или корректными. Аналитическая модель — это сжатая и точная абстракция того, что именно должна делать система (а не то, каким образом это будет сделано). Аналитическая модель не должна содержать никаких решений относительно реализации. Например, класс Window в аналитической модели системы управления окнами для рабочей станции должен быть описан в терминах видимых атрибутов и операций.

Аналитическая модель состоит из двух частей: *модели предметной области* (domain model) — описания объектов реального мира, отражаемых системой, и *модели приложения* (application model) — описания видимых пользователю частей самого приложения. Например, для приложения биржевого маклера объектами предметной области могут быть акции, облигации, торги и комиссия. Объекты модели приложения могут управлять выполнением торгов и отображать результаты. Хорошая модель должна быть доступной для понимания и критики со стороны экспертов, не являющихся программистами.

3. **Проектирование системы.** Команда разработчиков продумывает стратегию решения задачи на высшем уровне, определяя *архитектуру системы* (system architecture). На этом этапе определяются политики, которые послужат основой для принятия решений на следующих этапах проектирования. Проектировщик системы должен выбрать параметры системы, по которым будет проводиться оптимизация, предложить стратегический

подход к задаче, провести предварительное распределение ресурсов. Например, проектировщик может решить, что любые изменения изображения на экране рабочей станции должны быть быстрыми и плавными, даже при перемещении и закрытии окон. На основании этого решения он может выбрать подходящий протокол обмена и стратегию буферизации памяти.

4. **Проектирование классов.** Проектировщик классов уточняет аналитическую модель в соответствии со стратегией проектирования системы. Он прорабатывает объекты предметной области и объекты модели приложения, используя одинаковые объектно-ориентированные концепции и обозначения, несмотря на то, что эти объекты лежат в разных концептуальных плоскостях. Цель проектирования классов состоит в том, чтобы определить, какие структуры данных и алгоритмы требуются для реализации каждого класса. Например, проектировщик классов должен выбрать структуры данных и алгоритмы для всех операций класса *Window*.
5. **Реализация.** Ответственные за реализацию занимаются переводом классов и отношений, образовавшихся на предыдущем этапе, на конкретный язык программирования, воплощением их в базе данных или в аппаратном обеспечении. Никаких усложнений на этом этапе быть не должно, потому что все ответственные решения уже были приняты на предыдущих этапах. В процессе реализации необходимо использовать технологии разработки программного обеспечения, чтобы соответствие кода проекту было очевидным, а система оставалась гибкой и расширяемой. В нашем примере группа реализации должна написать код класса *Window* на каком-либо языке программирования, используя вызовы функций или методов графической подсистемы рабочей станции.

Объектно-ориентированные концепции действуют на протяжении всего жизненного цикла программного обеспечения, на этапах анализа, проектирования и реализации. Одни и те же классы будут переходить от одного этапа к другому без всяких изменений в нотации, хотя на последних этапах они существенно обростут деталями. В нашем примере модели класса *Window*, созданные на этапе анализа и на этапе реализации, являются корректными, но они служат разным целям, а потом отражают разные уровни абстракции. Концепции индивидуальности, классификации, полиморфизма и наследования действуют на протяжении всего процесса разработки.

Обратите внимание, что мы не предполагаем обязательного использования водопадного процесса разработки (составление требований, анализ, проектирование, реализация). Для каждой конкретной части системы этапы проектирования должны выполняться в указанном выше порядке, но это не значит, что части системы обязательно должны создаваться последовательно. Мы поддерживаем итерационный процесс разработки: составляющая часть системы разрабатывается в несколько этапов, после чего к ней добавляются новые возможности.

Некоторые классы не входят в аналитическую модель. Они появляются позднее, на этапах проектирования или реализации. Например, структуры данных, подобные *деревьям*, *хэш-таблицам* и *связным спискам*, редко появляются в реальном мире и обычно невидимы пользователям. Проектировщики добавляют их в систему для того, чтобы обеспечить поддержку выбранных алгоритмов. Объекты структур данных существуют внутри компьютера и не являются непосредственно наблюдаемыми.

Мы не рассматриваем тестирование как отдельный этап процесса. Тестирование очень важно, но оно должно быть частью системы контроля качества, которая применяется на протяжении всего жизненного цикла. Разработчики должны сравнивать аналитические модели с реальностью. Они должны проверять проектировочные модели на наличие ошибок различных видов, а не только тестировать корректность реализации. Выделение всех операций по контролю качества в отдельный этап стоит дороже и оказывается менее эффективно.

1.2.3. Три модели

Для описания системы с различных точек зрения мы используем три типа моделей. Модель классов описывает объекты, входящие в состав системы, и отношения между ними. Модель состояний описывает историю жизни объектов. Модель взаимодействий описывает взаимодействия между объектами. Каждая модель применяется на всех этапах проектирования и постепенно обрастает деталями. Полное описание системы требует наличия всех трех моделей.

Модель классов описывает статическую структуру объектов системы и их отношения. Эта модель определяет контекст разработки программы, то есть предметную область. Модель классов изображается на диаграммах классов. Диаграмма классов — это граф, вершинами которого являются классы, а ребрами — их отношения.

Модель состояний описывает изменяющиеся со временем аспекты объектов. Эта модель реализуется посредством диаграмм состояний. Диаграмма состояний — это граф, вершинами которого являются состояния, а ребрами — переходы между состояниями, инициируемые событиями.

Модель взаимодействия описывает кооперацию объектов системы для достижения лучших результатов. Построение модели начинается с вариантов использования, которые затем уточняются на диаграммах последовательности и диаграммах деятельности. Вариант использования описывает функциональность системы, то есть то, что система делает для пользователей. Диаграмма последовательности изображает взаимодействие объектов и временную последовательность этого взаимодействия. Диаграмма деятельности уточняет важные этапы обработки.

Три описанные модели являются связанными между собой составляющими полного описания системы. Центральной является модель классов, поскольку сначала нужно определить, *что* именно изменяется или трансформируется, а затем уже описывать *когда* и *как* это происходит.

1.3. Объектно-ориентированные концепции

В объектно-ориентированной технологии широко используются несколько базовых концепций. Они не ограничиваются рамками объектно-ориентированных систем, но объектная ориентированность означает, прежде всего, поддержку этих концепций.

1.3.1. Абстракция

Абстракция (abstraction) означает сосредоточение на важнейших аспектах приложения и игнорирование всех остальных. Сначала принимается решение о том, что представляет собой объект и что он делает, а уже затем подбирается способ его реализации. Использование абстракций позволяет сохранить свободу принятия решений как можно дольше благодаря тому, что детали не фиксируются раньше времени. Большинство современных языков программирования позволяют абстрагировать данные, но наследование и полиморфизм значительно расширяют возможности концепции абстрагирования. Умение создавать абстракции, вероятно, является самым важным качеством для объектно-ориентированного разработчика.

1.3.2. Инкапсуляция

Инкапсуляция (encapsulation), или, иначе говоря, сокрытие информации, состоит в отделении внешних аспектов объекта, доступных другим объектам, от деталей внутренней реализации, которые от других объектов скрываются. Инкапсуляция исключает возникновение взаимозависимостей участков программы, из-за которых небольшие изменения приводят к значительным непредвиденным последствиям. Реализация объекта может быть изменена безо всяких последствий для использующих его приложений. Изменение реализации может быть предпринято для повышения производительности, устранения ошибки, консолидации кода или для подготовки к переносу программы на другие системы. Инкапсуляция используется не только в объектно-ориентированном подходе, однако возможность объединения структур данных и поведения в одной сущности делает инкапсуляцию более ясной и более мощной, по сравнению с ее реализацией в языках типа Fortran, Cobol и C.

1.3.3. Объединение данных и поведения

При вызове операции не нужно беспокоиться о том, сколько реализаций этой операции существует в системе. Полиморфизм операторов перекладывает ответственность за выбор подходящей реализации с вызывающего кода на иерархию классов. Рассмотрим в качестве примера обычный (не объектно-ориентированный) код, отображающий содержимое окна. Такой код должен учитывать тип каждой фигуры (многоугольник, окружность, текст) и вызывать соответствующие типу процедуры. Объектно-ориентированный код будет вызывать для каждой

фигуры метод *draw* (прорисовать). Каждый объект выберет подходящую процедуру согласно своему классу. Это облегчает поддержку программы, потому что добавление нового класса не требует изменения вызывающего кода. В объектно-ориентированной системе иерархия структур данных соответствует иерархии наследования операций (рис. 1.3).

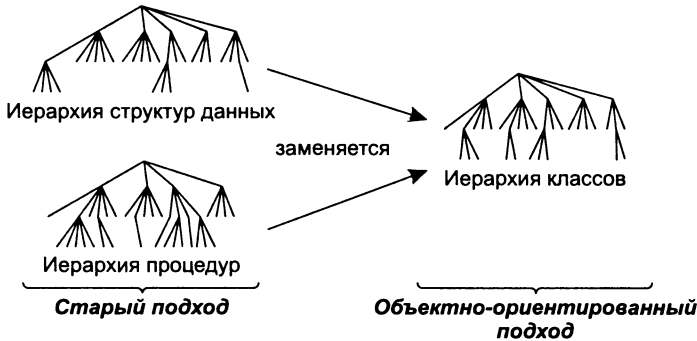


Рис. 1.3. Единая иерархия операций и структур данных

1.3.4. Совместное использование

Объектно-ориентированные технологии способствуют совместному использованию сущностей на самых разных уровнях. Наследование структур данных вместе с поведением дает возможность подклассам совместно использовать общий код. Совместное использование при наследовании является одним из главных преимуществ объектно-ориентированных языков. Однако концептуальная ясность, проистекающая из осознания того, что разные операции на деле представляют собой одно и то же, оказывается важнее, чем экономия кода. Благодаря этому сокращается количество различных ситуаций, которые вам требуется понять и проанализировать.

Объектно-ориентированная разработка позволяет не только совместно работать с общей информацией внутри приложения, но и повторно использовать проекты и код в последующих программах. Объектно-ориентированные средства (абстрагирование, наследование, инкапсуляция) помогают строить библиотеки повторно используемых компонентов. К сожалению, важность этого фактора была переоценена. Повторное использование не осуществляется само по себе. Проектировщики должны планировать на будущее, видеть дальше границ текущего приложения и вкладывать дополнительные усилия в получение более универсальных конструкций.

1.3.5. Выделение сущности объекта

Объектно-ориентированная технология выделяет то, чем объект *является*, а не то, как он *используется*. Использование объекта зависит от особенностей приложения и часто изменяется в процессе разработки. По мере уточнения требований, черты объекта остаются более стабильными, чем детали его использования,

поэтому системы, основанные на объектной структуре, в конечном счете оказываются более стабильными. При объектно-ориентированной разработке большее внимание уделяется структурам данных и меньшее — процедурам, нежели в методологиях, связанных с функциональным разбиением. В этом отношении объектно-ориентированная разработка подобна технологиям информационного моделирования, используемым при проектировании баз данных (за исключением добавленной в объектно-ориентированном подходе концепции поведения, зависящего от класса).

1.3.6. Когда целое больше суммы частей

Все объектно-ориентированные языки характеризуются поддержкой концепций индивидуальности, классификации, полиморфизма и наследования. Каждая из этих концепций может использоваться и сама по себе, однако вместе они образуют нечто большее. Преимуществ объектно-ориентированного подхода оказывается больше, чем может показаться с первого взгляда. Выделение основных свойств объекта заставляет разработчика более внимательно относиться к тому, чем объект является и что он делает. В результате система оказывается более ясной, универсальной и устойчивой, чем в том случае, если бы основное значение придавалось данным и операциям.

1.4. Доводы в пользу объектной ориентированности

Впервые мы применили объектно-ориентированный подход для создания приложений для внутренних потребностей исследовательского центра General Electric. Мы использовали объектно-ориентированные методики для разработки компиляторов, графики, пользовательских интерфейсов, баз данных, объектно-ориентированного языка, систем автоматизированного проектирования, симуляторов, метамodelей, систем управления и других приложений. Мы использовали объектно-ориентированные модели для документирования программ с плохой структурой, которые трудно было понять просто так. Мы реализовывали свои идеи на разных языках, как объектно-ориентированных, так и обычных, а также воплощали их в базах данных. Мы успешно обучали объектно-ориентированному подходу других программистов и использовали его для общения с экспертами по приложениям.

С середины 90-х годов XX века мы расширили опыт применения объектно-ориентированных технологий и распространили его в компаниях, разбросанных по всему миру. Когда мы писали первое издание этой книги, объектная ориентированность и объектно-ориентированное моделирование были в новинку. Отсутствовал опыт их применения в крупномасштабных проектах. Сейчас объектно-ориентированную технологию уже нельзя считать модным увлечением или умозрительной фикцией. Это часть информатики в целом и основное направление в технологиях разработки программного обеспечения.

На ежегодных конференциях OOPSLA (Object-Oriented Programming Systems, Languages, and Applications), ECOOP (European Conference on Object-Oriented Programming) и TOOLS (Technology of Object-Oriented Languages and Systems) распространяются новые объектно-ориентированные идеи и обсуждаются результаты их применения для создания приложений. В трудах конференций описано множество приложений, выигравших от применения объектно-ориентированного подхода. Статьи по объектно-ориентированным системам выходили в крупных журналах, таких как IEEE Computer и Communications of the ACM.

1.5. История объектно-ориентированного моделирования

Наша работа в исследовательском центре General Electric подвела нас к созданию методики объектного моделирования (Object Modeling Technique, OMT), которая была описана в первом издании этой книги в 1991 году. Эта методика оказалась удачной, но то же самое можно было сказать и о нескольких других подходах. Популярность объектно-ориентированного моделирования привела к новой проблеме: возникло множество альтернативных систем обозначений. Они выражали похожие идеи разными символами, что затрудняло взаимодействие между разработчиками.

Вследствие этого, сообщество программистов вплотную занялось объединением разных систем обозначений. В 1994 году Джим Рамбо (Jum Rumbaugh) присоединился к фирме Rational (которая ныне принадлежит IBM) и вместе с Грейди Бучем (Grady Booch) начал работу над объединением систем обозначений OMT и Буча. В 1995 году к Rational присоединился и Ивар Якобсон (Ivar Jacobson), который принес с собой концепцию Objectory.

В 1996 году группа управления объектами (Object Management Group — OMG) объявила конкурс на лучший стандарт обозначений для объектно-ориентированного моделирования. В этом конкурсе приняло участие несколько компаний. В результате их предложения были объединены в конечную систему. Окончательной доработкой стандарта занималась фирма Rational. В этом процессе активно участвовали Буч, Рамбо и Якобсон. В ноябре 1997 года группа OMG анонимно приняла получившийся в результате унифицированный язык моделирования (Unified Modeling Language — UML) в качестве стандарта. Компании, принимавшие участие в конкурсе, передали права на UML группе OMG, которая стала владельцем торговой марки и спецификаций UML. Эта группа управляет дальнейшим совершенствованием языка UML.

Система обозначений UML оказалась настолько удачной, что вытеснила практически все другие системы. Большинство авторов других методов приняли систему UML по собственной воле или из-за рыночного давления. Создание UML завершило войны систем обозначений, и теперь эта система действительно является общепринятой. Мы использовали UML в своей книге, потому что сейчас этот язык стал стандартом.

В 2001 году члены OMG начали работу над новой версией UML, добавляя в нее недостающие элементы и устраняя недостатки, выявленные в UML1. Наша книга основана на версии UML 2.0, которая была принята в 2004 году. Официальная спецификация UML имеется на веб-сайте OMG по адресу www.omg.org.

1.6. Структура книги

Книга, которую вы держите в руках, состоит из четырех основных частей: концепция моделирования, анализ и проектирование, реализация и технология разработки программного обеспечения. Краткий словарь терминов и решения к некоторым упражнениям приведены в приложении. В конце книги показана используемая в книге система обозначений.

В части I описываются объектно-ориентированные концепции и система графических обозначений для них. Глава 2 рассказывает о моделировании и о трех типах моделей (классов, состояний и взаимодействия). Главы 3 и 4 описывают модель классов, которая задает структуру данных системы. Эти главы являются основными в первой части. Понимание модели классов совершенно необходимо для успешной объектно-ориентированной разработки. Главы 5 и 6 рассказывают о модели состояний, которая отражает аспекты управления системой. Главы 7 и 8 описывают модель взаимодействия, которая отражает взаимодействие разных объектов системы. В главе 9 рассматриваются все три модели и взаимодействие между ними. Концепции, рассматриваемые в первой части, распространяются на весь цикл разработки программного продукта. Они применимы на этапах анализа, проектирования и реализации. Во всей книге используется система обозначений, описанная в первой части.

Часть II рассказывает, как подготовить объектно-ориентированную модель и использовать ее для анализа и проектирования системы. Глава 10 дает обзор процессов анализа и проектирования. В главе 11 обсуждается концепция системы и изобретение приложения. Главы 12 и 13 посвящены анализу, описанию и пониманию системы. Анализ начинается с постановки задачи заказчиком. Аналитик объединяет сведения, полученные от заказчика, и знания о приложении, после чего строит модель предметной области и модель приложения. В главе 14 рассматриваются вопросы проектирования системы, которое заключается, главным образом, в разбиении системы на подсистемы и принятии решений о политиках верхнего уровня. Глава 15 рассказывает о проектировании классов, которое, по сути, представляет собой соединение аналитической модели с проектными решениями. Проектные решения — это спецификации алгоритмов, определение функциональности объектов и оптимизация. Глава 16 подытоживает сведения о процессе.

Часть III посвящена реализации. Глава 17 рассматривает вопросы реализации, не связанные с языком программирования. В главах 18 и 19 речь пойдет о C++, Java и базах данных. Глава 20 содержит указания по улучшению читаемости и упрощению повторного использования и поддержки, чего можно достичь благодаря хорошему стилю объектно-ориентированного программирования.

Часть IV рассказывает о технологиях разработки программного обеспечения. Во второй части этапы рассматривались в линейной последовательности (как и должно быть в книге), но мы не согласны с водопадной моделью разработки. В главе 21 рассматривается итерационная модель разработки, в которой этапы процесса многократно повторяются до тех пор, пока не будет построена вся система. В главе 22 мы даем советы по управлению моделями. Проще всего понять и применить объектно-ориентированную разработку к новой системе, но большинство проектов начинаются не с нуля. Глава 23 рассказывает о том, как работать с существующими системами.

Почти во всех главах имеются упражнения. Ответы к некоторым из них вы найдете в конце книги. Мы советуем вам решать эти упражнения по мере чтения книги, даже если вы давно уже не студент. Упражнения помогают лучше понять многие тонкие моменты. Вы приобретете практический опыт работы с объектно-ориентированными технологиями, который послужит вам первой ступенью на пути к созданию приложений.

Таблица 1.1. Ключевые понятия главы

абстракция	инкапсуляция	объектная ориентированность
анализ	индивидуальность	полиморфизм
проектирование классов	реализация	модель состояний
модель классов	наследование	проектирование системы
классификация	модель взаимодействия	

Библиографические заметки

Хороший обзор объектно-ориентированных технологий приводится в статье [Taylor-98]. Полезным источником информации является книга [Meurer-97], несмотря на то, что она посвящена, главным образом, объектно-ориентированным языкам. В книге [Love-93] приведены примеры промышленных проектов, в которых использовалась объектно-ориентированная технология.

Мы пытаемся научить читателя объектно-ориентированным концепциям и образу мышления. Эта книга не должна использоваться в качестве справочника по UML. Если вам нужен именно справочник, обратитесь к [Rumbaugh-05]. В учебниках выделяются самые важные концепции и могут быть опущены несущественные детали. Мы не пытались описать все. Сосредоточившись на ключевых концепциях, вы изучите предмет гораздо быстрее.

Аналогичным образом мы обошлись и с процессом разработки. Описанный в этой книге подход к разработке относительно прост и рассчитан на небольшие и средние по размеру проекты. Однако этот подход включает в себя важнейшие моменты Унифицированного процесса (см. [Jacobson-99]).

Язык UML основан на концепции классификатора, более универсальной, чем понятие класса. Классификатор позволяет абстрагировать самые различные виды модельных сущностей. В этой книге мы используем слово *класс* вместо слова

классификатор, потому что разработчикам моделей предстоит работать главным образом с классами.

Ссылки

[Brooks-95] Frederick P. Brooks, Jr. *The Mythical Man-Month, Anniversary Edition*. Boston: Addison-Wesley, 1995.

[Jacobson-99] Ivar Jacobson, Grady Booch, James Rumbaugh. *The Unified Software Development Process*. Boston: Addison-Wesley, 1999.

[Love-93] Tom Love. *Object Lessons: Lessons Learned in Object-Oriented Development Practices*. New York: SIGS Books, 1993.

[Meyer-97] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. Hertfordshire, England: Prentice Hall International, 1997.

[Rumbaugh-05] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Boston: Addison-Wesley, 2005.

[Taylor-98] David A. Taylor. *Object Technology: A Manager's Guide, Second Edition*. Boston: Addison-Wesley, 1998.

Упражнения

Число в скобках указывает уровень сложности упражнения, от (1) — самый легкий, до (10) — самый сложный.

- 1.1 (3) С какими основными проблемами вам приходилось сталкиваться в своих проектах по разработке программного обеспечения? Оцените, какую часть времени вы затратили на анализ, проектирование, кодирование и тестирование, отладку, устранение ошибок вместе взятые. Каким образом вы оцениваете возможные затраты времени на будущие проекты?
- 1.2 (3) Рассмотрите какую-нибудь созданную вами ранее систему. Кратко опишите ее. С какими препятствиями вам пришлось столкнуться в ходе проектирования? Какой технологией разработки программного обеспечения вы пользовались? По какой причине вы выбрали именно ее? Довольны ли вы существующим вариантом системы? Насколько сложно добавлять в нее новые функции? Легко ли ее сопровождать?
- 1.3 (3) Опишите недавний крупный программный проект, который не уложился в сроки, превысил бюджет или не выполнил все поставленные требования. Какие факторы были причиной неудачи? Каким образом можно было избежать провала?
- 1.4 (3) Выскажите критику в адрес программной или аппаратной системы, в которой имеется особенно раздражающий вас недостаток. Например, в некоторых моделях автомобилей приходится снимать бампер, чтобы заменить тормозные огни. Опишите систему, ее недостаток, причины, по которым он не был устранен. Подумайте, каким образом его можно было бы устранить, приложив больше усилий на этапе проектирования.

1.5 (5) Все объекты обладают индивидуальностью, а потому отличаются друг от друга. Однако придумать схему для разделения объектов, принадлежащих к большой совокупности, может быть достаточно сложно. Кроме того, схема может зависеть от цели, с которой проводится разделение. Опишите возможную схему для каждой из перечисленных ниже совокупностей объектов:

- 1) все люди в мире с целью отправки почты;
- 2) все люди в мире с целью проведения криминальных расследований;
- 3) все клиенты, имеющие сейфы в конкретном банке;
- 4) все телефоны в мире с целью выполнения исходящих звонков;
- 5) все клиенты телефонной компании с целью рассылки счетов;
- 6) все адреса электронной почты в мире;
- 7) все сотрудники компании с целью разграничения доступа для обеспечения безопасности.

1.6 (4) Подготовьте список классов, которые, по вашему мнению, должны обрабатываться каждой из перечисленных ниже систем:

- 1) программа компьютерной верстки газеты;
- 2) программа вычисления и хранения результата при игре в боулинг;
- 3) система голосовой почты с параметрами доставки, пересылкой сообщений и групповыми списками;
- 4) контроллер видеомэгнитофона;
- 5) система ввода заказов интернет-магазина.

1.7 (6) Для каждого из перечисленных ниже классов выберите применимые к объектам этого класса операции. Одна операция может быть помещена в несколько классов. Обсудите поведение каждой из операций.

Классы:

- *variable-length array* (массив переменной длины) — упорядоченная совокупность объектов, индексом в которой является целое число. Размер совокупности может меняться во время выполнения программы;
- *symbol table* (таблица символов) — таблица, отображающая ключевые слова в дескрипторы;
- *set* (множество) — неупорядоченная совокупность объектов, в которой отсутствуют дубликаты.

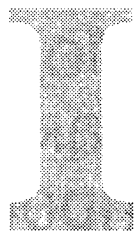
Операции:

- *append* (добавление в конец) — добавление объекта к концу совокупности;
- *copy* (копирование) — создание копии совокупности;
- *count* (подсчет) — подсчет количества элементов в совокупности;
- *delete* (удаление) — удаление элемента из совокупности;

- *index* (индекс) — получение объекта, находящегося в совокупности на заданном месте;
- *intersect* (пересечение) — определение элементов, принадлежащих двум совокупностям одновременно;
- *insert* (вставка) — помещение объекта в заданное место совокупности;
- *update* (обновление) — добавление элемента в совокупность с перезаписыванием существующего элемента.

1.8 (4) Обсудите общие характеристики классов в каждом из перечисленных ниже списков. Вы можете также расширить каждый из этих списков по своему желанию:

- сканирующий электронный микроскоп, очки, телескоп, прицел для бомбометания, бинокль;
- труба, обратный клапан, кран, фильтр, манометр;
- велосипед, парусник, автомобиль, грузовик, самолет, планер, мотоцикл, лошадь;
- гвоздь, винт, болт, заклепка;
- навес, погреб, сарай, гараж, амбар, дом, небоскреб.



Концепции моделирования

В первой части описываются концепции и система обозначений, используемые в объектно-ориентированном моделировании. И концепции, и система обозначений применяются на этапах анализа, проектирования и реализации.

В главе 2 обсуждаются общие вопросы моделирования и рассматриваются три вида объектно-ориентированных моделей: модели классов, состояний и взаимодействия.

В главе 3 рассматривается модель классов, описывающая статическую структуру системы. Модель классов является контекстом по отношению к двум другим видам моделей. Глава 4 посвящена более сложным концепциям моделирования классов. При первом чтении ее можно пропустить.

Глава 5 описывает модель состояний, отражающую изменяющиеся со временем аспекты системы, а также поведение управления. Глава 6 посвящена более сложным концепциям. Ее тоже можно пропустить при первом чтении.

В главе 7 обсуждается модель взаимодействия и завершается рассмотрение всех трех моделей. Модель взаимодействия описывает кооперацию объектов для достижения нужных результатов. Глава 8 посвящена сложным вопросам моделирования взаимодействия. Ее тоже можно пропустить при первом чтении.

В главе 9 подытоживаются полученные сведения о всех трех типах моделей и их взаимоотношениях.

Прочитав первую часть целиком, вы поймете объектно-ориентированные концепции и их обозначения в языке UML. Вы будете готовы применить эти концепции к разработке программного обеспечения, чему посвящены оставшиеся три части книги.

Моделирование как методика проектирования



Модель (model) — это абстракция, которая создается с целью постижения чего-либо перед тем, как оно будет создано. Поскольку модель не содержит несущественных деталей, работать с ней оказывается проще, чем с моделируемой сущностью. Абстрагирование — это одно из важнейших человеческих умений, которое дает нам возможность работать со сложными вещами. Инженеры, художники и ремесленники тысячелетиями занимались созданием моделей, на которых они опробовали свои планы перед тем, как воплощать их в жизнь. Проектирование аппаратных и программных систем не является исключением. Чтобы создать сложную систему, разработчик должен абстрагировать разные ее представления, построить модели, используя четкую систему обозначений, проверить, что модели удовлетворяют требованиям, предъявленным к системе, и постепенно добавлять детали, превращая модели в реализацию.

2.1. Моделирование

Конструкторы строят множество моделей разных типов, прежде чем перейти к созданию самих продуктов. В качестве примеров можно привести архитектурные модели, предназначенные для показа заказчикам, масштабные модели самолетов для тестирования в аэродинамических трубах, карандашные эскизы масляных полотен, светокопии деталей машины, раскадровки рекламных роликов и планы книг.

Различные модели могут быть объединены в несколько групп по своему назначению.

- 1. Модели для тестирования физического объекта перед его созданием.** Средневековые каменщики не были знакомы с достижениями современной физики, поэтому они строили масштабные модели готических соборов, чтобы проверить действующие в них силы. Инженеры тестируют масштабные модели самолетов, автомобилей и кораблей в аэродинамических

трубах и бассейнах, чтобы усовершенствовать их динамические характеристики. Современные вычислительные возможности позволяют моделировать многие физические структуры в компьютере, не создавая реальных моделей. Такое моделирование не только оказывается дешевле, но и дает дополнительные данные, которые сложно или невозможно измерить в физической модели. Физические и компьютерные модели стоят дешевле реальной системы и при этом позволяют выявить и устранить недостатки на ранних этапах проектирования.

2. **Взаимодействие с заказчиками.** Архитекторы и дизайнеры создают модели, которые затем показывают своим заказчикам. Это демонстрационные модели, имитирующие (частично или полностью) внешнее поведение системы.
3. **Визуализация.** Раскадровка фильмов, телешоу и рекламных роликов позволяет сценаристам увидеть воплощение их идей. Они могут исправить неудачные переходы, устранить незавершенные линии и избыточные сегменты перед тем, как начать проработку более детального сценария. Эскизы дают художникам возможность набросать свои замыслы вчерне и внести необходимые изменения перед тем, как воплощать их в масле или камне.
4. **Уменьшение сложности.** Основное назначение моделирования, включающее все предыдущие пункты, состоит в том, что оно позволяет работать с системами, которые являются слишком сложными для непосредственного изучения. Человеческий мозг может обрабатывать лишь ограниченный объем информации в единицу времени. Модели уменьшают сложность реальных систем, выделяя ограниченный набор важнейших свойств.

2.2. Абстрагирование

Абстрагирование — это выборочное изучение некоторых аспектов проблемы. Цель абстрагирования состоит в том, чтобы изолировать аспекты, важные для некоторой цели, и избавиться от всех остальных. Абстрагирование всегда должно иметь цель, поскольку именно она определяет, что является важным, а что нет. Одна и та же сущность может иметь множество разных абстракций, отличающихся друг от друга назначением.

Все абстракции являются неполными и неточными. Реальность — это бесшовная ткань. Любое описание реальности не является полным. Все слова человеческих языков являются абстракциями — неполными описаниями реального мира. Однако это не лишает их полезности. Назначение абстракции состоит в том, чтобы ограничить множество возможностей и сделать его доступным для понимания. Поэтому при построении моделей следует стремиться не к абсолютной истине, а к адекватности некоторой цели. Не существует единственной «правильной» модели ситуации, существуют только адекватные и неадекватные модели.

Хорошая модель описывает важнейшие аспекты проблемы и опускает все прочие. Большинство компьютерных языков плохо подходят для моделирования алгоритмов, потому что они требуют указания деталей реализации, несущественных

для алгоритмов. Модель, содержащая избыточные детали, ограничивает ваш выбор при проектировании и отвлекает ваше внимание от важнейших аспектов.

2.3. Три модели

Нам кажется полезным моделировать систему с трех различных точек зрения, связанных между собой. Каждая модель описывает важные аспекты системы, но для более полного описания требуются все три модели. Модель классов представляет статические, структурные аспекты системы, связанные с данными. Модель состояний представляет временные, поведенческие, управленческие аспекты системы. Модель взаимодействия представляет кооперацию отдельных объектов, другими словами, все аспекты системы, связанные с взаимодействиями. Типичная процедура в программе обладает всеми тремя аспектами: она использует структуры данных (модель классов), упорядочивает операции во времени (модель состояний) и передает данные и управление между объектами (модель взаимодействия). Каждая модель содержит ссылки на сущности из других моделей. Например, модель классов связывает операции с классами, тогда как модели состояний и взаимодействия конкретизируют операции.

Модели трех типов дают различные представления системы. Модели не являются абсолютно независимыми, потому что система представляет собой нечто большее, нежели объединение независимых частей, однако каждая модель в значительной степени может быть проанализирована и понята сама по себе. Связи между моделями ограничены и выражены явным образом. Разумеется, всегда можно создать такой плохой проект, в котором три модели окажутся настолько переплетены друг с другом, что их невозможно будет разделить. Хороший проект разделяет разные аспекты системы и ограничивает связь между ними.

В процессе разработки все три модели развиваются постепенно. Сначала аналитики конструируют модель приложения, не задумываясь о последующей реализации. Затем проектировщики добавляют в эту модель конструкции, необходимые для решения поставленных задач. Группа реализации кодирует конструкции приложения. Модель определяется не только видом (модель классов, состояний, взаимодействия), но и этапом разработки (аналитическая модель, проектная модель, модель реализации).

2.3.1. Модель классов

Модель классов (class model) описывает структуру объектов системы: их индивидуальность, отношения с другими объектами, атрибуты и операции. Модель классов создает контекст для моделей состояний и взаимодействия. Изменения и взаимодействия не имеют смысла, если отсутствует изменяющийся объект или взаимодействующие объекты. Объекты (objects) — это блоки, на которые мы разбиваем наш мир, молекулы нашей модели.

Цель конструирования модели классов состоит в том, чтобы охватить те реальные концепции, которые важны для нашего приложения. При моделировании инженерной задачи модель классов должна содержать термины, знакомые инженерам.

При моделировании бизнес-задачи должны использоваться термины из бизнеса. Модель пользовательского интерфейса должна быть выражена в терминах приложения. Аналитическая модель не должна содержать компьютерных конструкций, если только моделируемое приложение не является чисто компьютерным, как, например, компилятор или операционная система. Проектная модель описывает возможности решения задачи, а потому может содержать компьютерные конструкции.

Модель классов изображается на диаграммах классов. Обобщение позволяет классам использовать общую структуру и поведение, а связи между классами осуществляются при помощи ассоциаций. Классы определяют значения атрибутов для каждого объекта и операции, которые выполняются самими объектами или с их участием.

2.3.2. Модель состояний

Модель состояний (state model) описывает аспекты объектов, связанные с течением времени и с последовательностью операций, то есть события, связанные с изменениями, состояния, определяющие контекст событий, и упорядочение событий и состояний. Модель состояний охватывает вопросы управления — аспект системы, описывающий порядок осуществляемых операций без учета их фактического значения, участников и реализации.

Модель состояний изображается на диаграммах состояний. Каждая диаграмма состояний показывает порядок состояний и событий, возможный в рамках данной системы для одного класса объектов. Диаграммы состояний ссылаются на другие модели. Действия и события на диаграмме состояний становятся операциями объектов модели классов. Ссылки между диаграммами состояний становятся взаимодействиями в модели взаимодействия.

2.3.3. Модель взаимодействия

Модель взаимодействия (interaction model) описывает взаимодействие между объектами, то есть кооперацию объектов для обеспечения необходимого поведения системы как целого. Модели состояний и взаимодействия описывают разные аспекты поведения, и для полного описания поведения необходимы они обе.

Модель взаимодействия изображается при помощи вариантов использования на диаграммах последовательности и деятельности. Варианты использования описывают основные варианты взаимодействия системы с внешними актерами. Диаграммы последовательности показывают временную последовательность взаимодействия объектов вместе с самими объектами. Диаграммы деятельности показывают поток управления между последовательными этапами вычислений.

2.3.4. Отношения моделей

Каждая модель описывает свои аспекты системы, но при этом она ссылается на другие модели. Модель классов описывает структуры данных, которыми оперируют модели состояний и взаимодействия. Операции в модели классов связаны

с событиями и действиями. Модель состояний описывает структуру управления объектов. Она показывает решения, зависящие от значений объектов, и действия, изменяющие значения объектов и состояния. Модель взаимодействия подчеркивает обмен между объектами и дает единый обзор операций в рамках системы.

Иногда возникают неоднозначности, касающиеся того, к какой модели должна быть отнесена некоторая информация. Это естественно, поскольку любая абстракция является лишь грубым срезом реальности. Что-нибудь неизбежно окажется на границе этого среза. Некоторые свойства системы могут быть плохо отражены в трех моделях. Это тоже нормально, потому что никакая абстракция не может быть совершенной. Цель состоит в том, чтобы упростить описание системы и избежать перегрузки модели большим количеством конструкций, из-за чего она стала бы бесполезной. Если какие-то детали плохо отражаются в моделях, для них можно использовать естественные языки или систему обозначений, предназначенную специально для данного приложения.

2.4. Резюме

Модели являются абстракциями и создаются для того, чтобы лучше понять задачу, перед тем как приступить к ее решению. Любая абстракция является подмножеством реальности, выбранным с некоторой целью.

Мы рекомендуем использовать модели трех типов. Модель классов описывает статическую структуру в терминах классов и отношений между ними. Модель состояний описывает управляющую структуру системы в терминах событий и состояний. Модель взаимодействия описывает кооперацию отдельных объектов для достижения необходимого поведения системы как целого. Значение модели определенного вида зависит от задачи.

Таблица 2.1. Ключевые понятия главы

абстракция	моделирование
модель классов	отношение между моделями
модель взаимодействия	модель состояний

Библиографические замечания

В первом издании этой книги тоже предлагалось использовать три модели (объектную, динамическую и функциональную), но они были организованы иначе, нежели во втором издании.

Модель объектов из первого издания совпадает по смыслу с моделью классов, о которой мы рассказываем теперь. Мы изменили название модели, чтобы подчеркнуть, что ее элементами являются дескрипторы (классы и отношения между ними), а не экземпляры (объекты и связи). Описание модели классов в этой книге включает вопросы моделирования ограничений, которые не рассматривались в первом издании.

Динамическая модель из первого издания совпадает по смыслу с моделью состояний в этом издании. Мы изменили название, чтобы избежать путаницы с другими представлениями динамического поведения. UML определяет модели различных видов, которые в той или иной мере перекрываются друг с другом. В этой книге мы рассказываем только о наиболее важных из них.

Функциональную модель мы исключили из второго издания. Разумеется, создаваемое программное обеспечение обладает некоторой функциональностью, но мы редко отражаем ее на диаграммах потоков данных, как предлагалось в первом издании. Диаграммы потоков данных были включены в первое издание для обеспечения непрерывного перехода от структурного анализа и проектирования, принятого в прошлом, к объектно-ориентированному. Функциональная модель оказалась не такой полезной, как мы думали, поэтому мы решили отказаться от нее.

Вместо нее во втором издании рассматривается модель взаимодействия. Диаграммы состояний полностью описывают динамическое поведение, но часто это описание оказывается довольно сложным для понимания. Каждая диаграмма состояний относится к одному классу. Когда у многих классов диаграммы состояний получаются достаточно сложными, система в целом может стать недоступной для понимания. Модель взаимодействия описывает кооперацию и помогает разработчику получить более полное понимание, чем это возможно с одними диаграммами состояний.

Упражнения

- 2.1. (1) Среди характеристик автомобильной шины можно перечислить ее размер, материал, внутреннее устройство (со смещенными слоями, металлокордная и т. п.), рисунок протектора, стоимость, срок службы и вес. Какие факторы являются решающими при выборе шин для вашего автомобиля? Какие могут оказаться важными для того, кто тестирует компьютеризованную систему защиты от заносов для машин? А какие характеристики важны тому, кто делает своему ребенку качели из старой шины?
- 2.2. (2) Представьте, что у вас в ванной забился сток и вы решили прочистить его, засунув в сливное отверстие проволоку. У вас в кладовке есть проволока разных типов, как с изоляцией, так и без нее. Какие характеристики проволоки вы сочтете наиболее важными? Поясните свой ответ.
 - 1) Защищенность от электрических помех.
 - 2) Цвет изоляции.
 - 3) Устойчивость изоляции к соленой воде.
 - 4) Огнеупорность изоляции.
 - 5) Стоимость.
 - 6) Жесткость.
 - 7) Легкость снятия изоляции.
 - 8) Вес.

- 9) Наличие.
 - 10) Прочность.
 - 11) Стойкость к высоким температурам.
 - 12) Стойкость при растяжении.
- 2.3. (3) Провода используются в самых разных ситуациях. Для каждого из перечисленных ниже случаев укажите важные характеристики провода и объясните, почему важны именно они.
- 1) Выбирается провод для трансатлантического кабеля.
 - 2) Выбирается провод для создания красочного художественного произведения.
 - 3) Проектируется электрическая система самолета.
 - 4) Выбирается провод для того, чтобы подвесить на дерево кормушку для птиц.
 - 5) Проектируется фортепиано.
 - 6) Подбирается нить для лампы накаливания.
- 2.4. (3) Если бы вы проектировали протокол для передачи файлов с одного компьютера на другой по телефонным линиям, какие детали вы сочли бы важными? Объясните, почему.
- 1) Электрические помехи в линиях связи.
 - 2) Скорость последовательной передачи данных.
 - 3) Наличие базы данных.
 - 4) Наличие хорошего полноэкранный редактора.
 - 5) Буферизация и управление потоком (например, протокол XON/XOFF, управляющий потоком данных).
 - 6) Количество дорожек и секторов на жестком диске.
 - 7) Интерпретация символов (особая обработка управляющих символов).
 - 8) Организация файлов (например, линейный поток байтов или наличие записей).
 - 9) Математический сопроцессор.
- 2.5. (2) При анализе и проектировании электрических двигателей используются несколько моделей. Электрическая модель включает сведения о напряжениях, токах, электромагнитных полях, индуктивностях и сопротивлениях. Механическая модель учитывает жесткость, плотность, движение, силы и крутящие моменты. Тепловая модель учитывает поглощение и передачу тепла. Гидроаэродинамическая модель описывает поток охлаждающего воздуха. Какие модели могут дать ответ на поставленные ниже вопросы? Обсудите.
- 1) Какая мощность требуется для работы двигателя? Какая часть этой мощности рассеивается в виде тепла?
 - 2) Сколько весит двигатель?

- 3) Как сильно нагревается двигатель?
 - 4) Какой уровень вибрации он создает?
 - 5) Сколько времени прослужат подшипники двигателя?
- 2.6. (3) Какие модели (классов, состояний, взаимодействия) описывают перечисленные ниже аспекты компьютерной шахматной программы. На экране отображается доска и фигуры. Человек управляет фигурами при помощи курсора, который, в свою очередь, управляется мышью. Разумеется, в некоторых случаях может быть применима не одна модель, а несколько. Поясните свои ответы.
- 1) Пользовательский интерфейс, отображающий ходы компьютера и позволяющий человеку делать свои ходы.
 - 2) Представление конфигурации, образованной фигурами на доске.
 - 3) Анализ последовательности возможных разрешенных ходов.
 - 4) Проверка хода, сделанного человеком.

Моделирование классов

Модель классов описывает статическую структуру системы: объекты и отношения между ними, атрибуты и операции для каждого класса объектов. Модель классов — самая важная из трех основных моделей. Мы считаем, что в основе системы должны быть объекты, а не требуемая функциональность, потому что объектно-ориентированная система лучше соответствует реальному миру и оказывается более жизнеспособной при возможных изменениях. Модели классов являются интуитивным графическим представлением системы и потому особенно полезны для общения с заказчиками.

В главе 3 обсуждаются основные концепции моделирования, которые будут использоваться во всей книге. Мы даем определение каждой концепции, указываем соответствующую систему обозначений UML и приводим примеры. К важнейшим концепциям, рассмотренным ниже, относятся объект, класс, связь, ассоциация, обобщение и наследование. Материал этой главы совершенно необходим для понимания книги в целом.

3.1. Концепции объекта и класса

3.1.1. Объекты

Цель моделирования классов состоит в описании объектов. В качестве примеров объектов можно привести следующие: *Джо Смит*, *компания Симплекс*, *процесс № 7648* и *активное окно*.

Объект (object) — это концепция, абстракция или сущность, обладающая индивидуальностью и имеющая смысл в рамках приложения. Объекты часто бывают именами собственными или конкретными ссылками, которые используются в описании задач или при общении с пользователями. Некоторые объекты существуют или существовали в реальном мире (например, *Альберт Эйнштейн* или *компания General Electric*), тогда как другие являются сугубо концептуальными

сущностями (например, *тестовый прогон № 1234* или *формула корней квадратного уравнения*). Объекты третьего типа (*бинарное дерево 634* и *массив, связанный с переменной a*) добавляются в модель в процессе реализации и не имеют никакого отношения к физической реальности. Выбор объектов зависит от природы задачи и от предпочтений разработчика. Корректное решение в данном случае не является единственным.

Все объекты обладают индивидуальностью и потому отличимы друг от друга. Два яблока одинакового цвета, формы и текстуры все равно являются разными яблоками. Вы можете съесть сначала одно из них, а потом второе. Похожие друг на друга близнецы тоже являются независимыми индивидуальностями. Индивидуальность означает, что объекты отличаются друг от друга внутренне, а не по внешним свойствам.

3.1.2. Классы

Объект является экземпляром класса. Класс (class) описывает группу объектов с одинаковыми свойствами (атрибутами), одинаковым поведением (операциями), типами отношений и семантикой. В качестве примеров классов можно привести следующие: *человек*, *компания*, *процесс* и *окно*. Каждый человек имеет имя и дату рождения, а также может где-либо работать. Каждый процесс имеет владельца, приоритет и список необходимых ресурсов. Классы часто бывают именами нарицательными и именными группами, которые используются в описании задач или при общении с пользователями.

Объекты одного класса имеют одинаковые атрибуты и формы поведения. Большинство объектов отличаются друг от друга значениями своих атрибутов и отношениями с другими объектами. Однако возможно существование разных объектов с одинаковыми значениями атрибутов, находящихся в одинаковых отношениях со всеми остальными объектами. Выбор классов зависит от природы и области применения приложения и зачастую является субъективным.

Объекты класса имеют общее семантическое значение, помимо обязательных общих атрибутов и поведения. Например, и амбар, и лошадь могут характеризоваться стоимостью и возрастом. С финансовой точки зрения они могли бы относиться к одному классу. Однако если разработчик учтет, что человек может красть амбар и кормить лошадь, то эти две сущности будут отнесены к разным классам. Интерпретация семантики зависит от назначения конкретного приложения и является делом субъективным.

Каждый объект «знает» свой собственный класс. Большинство объектно-ориентированных языков программирования позволяют определять класс объекта во время выполнения программы. Класс объекта — это его неявное свойство.

Если предметом моделирования являются объекты, то почему вообще заходит речь о классах? Все дело в необходимости абстрагирования. Группируя объекты в классы, мы производим абстрагирование в рамках задачи. Именно благодаря абстракциям моделирование оказывается столь мощным инструментом, позволяющим проводить обобщения от нескольких конкретных случаев к множеству подобных альтернатив. Общие определения (такие как название класса и названия атрибутов) хранятся отдельно для каждого класса, а не для каждого экземпляра.

Операции могут быть написаны один раз для целого класса, благодаря чему все объекты класса получают возможность повторно использовать написанный код. Например, все эллипсы имеют общие процедуры прорисовки, вычисления площади и проверки на наличие пересечения с некоторой прямой. У многоугольников будет свой набор процедур. Даже в особых случаях (например, для окружностей и квадратов) могут использоваться универсальные процедуры, хотя специальные алгоритмы могут быть и эффективнее.

3.1.3. Диаграммы классов

В начале этой главы мы обсудили некоторые основные концепции моделирования, а именно *объект* и *класс*. Мы рассказывали о них словами и приводили примеры. Такой подход не дает достаточной четкости и непригоден для описания сложных приложений. Нам нужно средство описания моделей, которое было бы согласованным, точным и простым в использовании. Модели структуры бывают двух типов: диаграммы классов и диаграммы объектов.

Диаграммы классов позволяют описать модель классов и их отношений (а значит, и возможные объекты) при помощи графической системы обозначений. Диаграммы классов полезны как для абстрактного моделирования, так и для проектирования конкретных программ. Они точны, просты для понимания и хорошо зарекомендовали себя на практике. На протяжении всей книги мы будем пользоваться диаграммами классов для отражения структуры приложений.

Периодически мы будем использовать и диаграммы объектов. На таких диаграммах изображаются отдельные объекты и отношения между ними. Диаграммы объектов полезны для документирования тестовых ситуаций и обсуждения примеров. Диаграмма классов описывает бесконечное множество диаграмм объектов.

На рис. 3.1 показан класс (слева) и его экземпляры (справа). Объекты *JoeSmith* (ДжоСмит), *MarySharp* (МэриШарп) и безымянная личность являются экземплярами класса *Person* (Человек). В языке UML для обозначения объекта используется прямоугольник, внутри которого ставится имя объекта, двоеточие и имя класса, к которому относится этот объект. И имя объекта, и имя класса подчеркиваются. Мы используем дополнительное соглашение, которое состоит в том, что имена объектов и классов выделяются полужирным шрифтом.

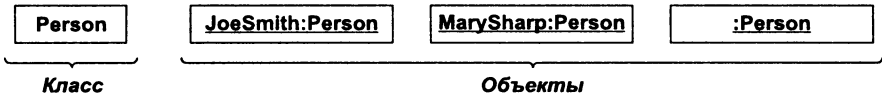


Рис. 3.1. Объекты и классы — предмет модели классов

Для обозначения класса в UML тоже используется прямоугольник. Мы указываем имя класса полужирным шрифтом, располагая его посередине прямоугольника. Имя класса мы всегда будем начинать с заглавной буквы. Эти соглашения используются для ссылок на объекты, классы и другие конструкции. Могут быть приняты альтернативные соглашения, например, о допустимости пробелов или знаков подчеркивания в названиях классов или объектов (*Joe Smith*,

Joe_Smith). Соглашение об использовании заглавных букв между строчных популярно в литературе, посвященной объектно-ориентированным технологиям, но не является требованием языка UML.

3.1.4. Значения и атрибуты

Значение (value) — это элемент данных. Значения можно определить, изучив примеры, приведенные в документации по поставленной задаче. Атрибут (attribute) — это именованное свойство класса, описывающее значение, которое может иметь каждый объект класса. Атрибуты — это прилагательные. Они получают абстрагированием типичных значений. Можно провести следующую аналогию: объект относится к классу так, как значение относится к атрибуту. В моделях классов преобладают структурные конструкции (то есть классы и отношения между ними, о которых речь пойдет дальше). Атрибуты имеют не столь важное значение и служат для уточнения характеристик классов и отношений.

Атрибутами объектов класса *Person* (Человек) являются *name* (имя), *birthdate* (датарождения) и *weight* (вес). Атрибутами объектов класса *Car* (Машина) являются *color* (цвет), *modelYear* (годвыпуска) и *weight* (вес). У каждого конкретного объекта атрибут принимает свое конкретное значение. Например, у объекта *JoeSmith* атрибут *birthdate* может иметь значение «21 октября 1983». Другими словами, Джо Смит родился 23 октября 1983 года. У разных объектов один и тот же атрибут может иметь как разные, так и одинаковые значения. Имя атрибута уникально в рамках класса (но не обязательно уникально во множестве всех классов). Поэтому у классов *Person* и *Car* может быть атрибут с одним и тем же названием — *weight*.

Не следует путать значения с объектами. Атрибут должен описывать значения, а не объекты. В отличие от объектов значения не обладают индивидуальностью. Например, все экземпляры числа 17 неразличимы между собой. Точно так же неразличимы экземпляры строкового значения «Канада». С другой стороны, страна, которая называется Канада, является объектом, у которого атрибут «название» имеет значение «Канада».

На рис. 3.2 показана система обозначений атрибутов. Класс *Person* (Человек) имеет атрибуты *name* (имя) и *birthdate* (датарождения). *Name* — это строка, а *birthdate* — дата. У одного из объектов класса *Person* атрибут *name* имеет значение «Джо Смит», а *birthdate* имеет значение «21 октября 1983». У другого объекта того же класса атрибут *name* имеет значение «Мэри Шарп», а атрибут *birthdate* имеет значение «16 марта 1950».

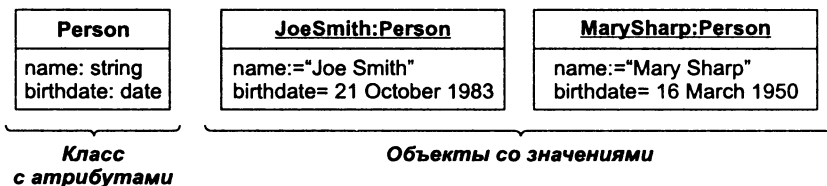


Рис. 3.2. Атрибуты обеспечивают детализацию классов

Согласно системе обозначений UML, атрибуты указываются во втором (сверху) отделе прямоугольника, обозначающего класс. После каждого атрибута могут быть указаны необязательные сведения о нем (например, тип и значение по умолчанию). Перед значением по умолчанию ставится знак равенства. Мы указываем название атрибута обычным шрифтом (не полужирным) и первую букву названия не делаем заглавной.

Во втором отделе прямоугольника объекта могут быть указаны значения атрибутов. В этом случае после названия атрибута ставится знак равенства, после которого записывается значение этого атрибута. Значения атрибутов мы выравниваем по левому краю и не выделяем их полужирным шрифтом.

Иногда среда реализации может требовать наличия у объекта уникального идентификатора. Идентификаторы всегда неявным образом присутствуют в модели классов. Их не следует указывать явно (рис. 3.3). Большинство объектно-ориентированных языков генерируют идентификаторы для ссылок на объекты автоматически. Так же легко определить идентификаторы и для баз данных. Идентификаторы являются артефактом вычислительной системы и не имеют внутреннего смысла.

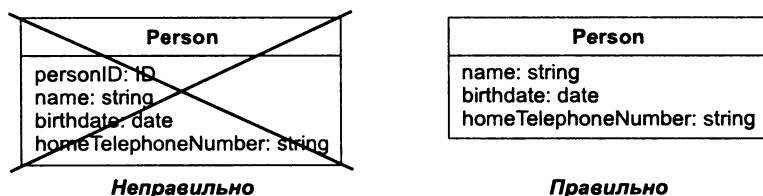


Рис. 3.3. Не следует указывать идентификаторы объектов явным образом

Не следует путать внутренние идентификаторы с атрибутами. Внутренние идентификаторы вводятся исключительно для удобства реализации и не имеют никакого смысла в контексте приложения. Такие атрибуты, как индивидуальный номер налогоплательщика, серийный номер и телефонный номер, не являются внутренними идентификаторами, поскольку они имеют значение в реальном мире. Их можно назвать полноправными атрибутами.

3.1.5. Операции и методы

Операция — это функция или процедура, которая может быть применена к объектам класса. Операциями класса *Company* (Компания) могут быть *hire* (нанять), *fire* (уволить) и *payDividend* (выплатить Дивиденды). Операциями класса *Window* (Окно) могут быть *open* (открыть), *close* (закрыть), *hide* (скрыть) и *redisplay* (перерисовать). Все объекты одного класса имеют общий список операций.

Каждая операция в качестве неявного аргумента принимает свой целевой объект. Поведение операции зависит от класса целевого объекта. Объект всегда знает свой собственный класс, а потому он всегда знает и правильную реализацию операции.

Одна и та же операция может быть применена к разным классам. Такая операция называется полиморфной: в разных классах она может принимать разные

формы. Методом (method) называется реализация операции в конкретном классе. Например, класс *File* (Файл) может иметь операцию *print* (печать). Печать ASCII-файлов, двоичных файлов и цифровых изображений может осуществляться разными методами. Все эти методы с логической точки зрения выполняют одно и то же действие: печать файла. Поэтому они и являются реализациями одной и той же операции *print*. При этом каждый метод может быть реализован своим собственным кодом.

У операции могут быть и другие аргументы, кроме целевого объекта. Эти аргументы могут быть как значениями, так и другими объектами. Выбор метода зависит только от класса целевого объекта, но не от классов аргументов, которые являются объектами, сколько бы их ни было. В некоторых объектно-ориентированных языках, в частности в CLOS, выбор метода может определяться произвольным числом аргументов, но такая универсальность значительно усложняет семантику модели.

Если операция реализована несколькими методами в разных классах, очень важно, чтобы у всех методов была одна и та же сигнатура (signature) — количество и типы аргументов, а также тип возвращаемого значения. Например, у операции *print* не может быть аргумента *fileName* (имяФайла) в одном методе и *filePointer* (указательФайла) в другом методе. Поведение всех методов операции должно быть согласованно. Лучше всего избегать использования одинаковых названий для операций, отличающихся друг от друга с семантической точки зрения, даже если они применяются к разным множествам классов. Например, неправильно было бы использовать название «инверсия» для операций инвертирования матрицы и переворота геометрической фигуры. В большом проекте может потребоваться использование пространств имен для предотвращения конфликтов, но лучше всего заранее принимать меры для предотвращения возможных проблем.

На рис. 3.4 изображен класс *Person* (Человек) с атрибутами *name* (имя) и *birthdate* (датарождения) и операциями *changeJob* (сменитьРаботу) и *changeAddress* (сменитьАдрес). Атрибуты и операции называются составляющими класса. У класса *File* (Файл) есть операция *print* (печать). У класса *GeometricObject* (ГеометрическийОбъект) есть операции *move* (перемещение), *select* (выделение) и *rotate* (поворот). Операция *move* имеет один аргумент *delta* (смещение) типа *Vector* (Вектор), операция *select* имеет аргумент *p* типа *Point* (Точка) и возвращает значение типа *Boolean* (логическое значение). Операция *rotate* имеет аргумент *angle* (угол), который относится к типу чисел с плавающей точкой и имеет значение по умолчанию 0.0.

Person	File	GeometricObject
name	fileName	color
birthdate	sizeInBytes	position
changeJob	lastUpdate	move (delta : Vector)
changeAddress	print	select (p : Point): Boolean
		rotate (in angle : float = 0.0)

Рис. 3.4. Операции

Система обозначений UML предписывает перечислять операции в третьем отделе прямоугольника класса. Название операции мы указываем обычным шрифтом и выравниваем по левому краю. Первую букву названия мы не делаем заглавной. После названия операции могут быть указаны необязательные дополнительные сведения, такие как список аргументов и тип возвращаемого результата. Список аргументов указывается в круглых скобках. Аргументы отделяются друг от друга запятыми. Перед типом возвращаемого результата ставится двоеточие. Пустой список аргументов в круглых скобках явно показывает, что данная функция не принимает никаких аргументов. Если же списка просто нет, никаких выводов делать нельзя. Мы не указываем операции для отдельных объектов, поскольку у всех объектов одного класса операции одинаковые.

3.1.6. Резюме системы обозначений классов

Полная система обозначений классов показана на рис. 3.5. Класс обозначается прямоугольником, который может иметь до трех отделов. Отделы нумеруются сверху вниз. В первом отделе указывается имя класса, во втором — список атрибутов, в третьем — список операций. После названия атрибута может быть указан его тип и значение по умолчанию. После названия операции может быть указан список аргументов и тип возвращаемого значения.

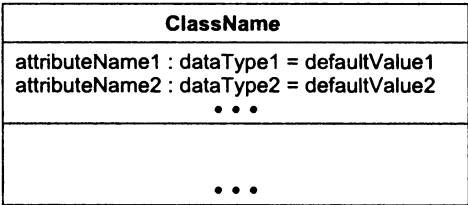


Рис. 3.5. Система обозначений для класса

Каждый аргумент может быть охарактеризован направлением, названием, типом и значением по умолчанию (рис. 3.6). По направлению аргумент может быть входным (in), выходным (out) или изменяемым (inout). Перед типом аргумента ставится двоеточие. Перед значением по умолчанию ставится знак равенства. Это значение используется в том случае, если при вызове операции значение аргумента не было указано явно.

direction argumentName : type=defaultValue

Рис. 3.6. Система обозначений аргумента операции

Отделы со списком атрибутов и операций являются необязательными элементами системы обозначений. Отсутствие отдела атрибутов говорит о том, что в данном представлении атрибуты не указаны. Отсутствие отдела операций также говорит о том, что в данном представлении не указаны операции. Напротив, наличие пустого отдела говорит о том, что атрибуты или операции отсутствуют.

3.2. Концепции связи и ассоциации

Связи и ассоциации позволяют устанавливать отношения между объектами и классами.

3.2.1. Связи и ассоциации

Связь (link) — это физическое или концептуальное соединение между объектами. Например, *Джо Смит работает на компанию Симплекс*. В большинстве случаев связь соединяет ровно два объекта, но бывают связи, соединяющие большее количество объектов. В этой главе мы будем рассказывать только о бинарных ассоциациях, а в главе 4 займемся n-арными. С математической точки зрения связь является *кортежем* (tuple), то есть списком объектов. Связь — это экземпляр ассоциации.

Ассоциация (association) — это описание группы связей, обладающих общей структурой и общей семантикой. Например, *человек может работать на* какую-либо *компанию*. Связи, являющиеся экземплярами некоторой ассоциации, соединяют объекты тех классов, которые соединены между собой этой ассоциацией. Ассоциация описывает множество потенциальных связей точно так же, как класс описывает множество потенциальных объектов. Связи и ассоциации обычно присутствуют в постановке задачи в виде глаголов.

На рис. 3.7 приведен пример модели финансового приложения. Брокерские конторы должны иметь возможность выполнять следующие действия: записывать принадлежность различных акций, отслеживать дивиденды, предупреждать клиентов об изменениях на рынке и вычислять предписываемую маржу. В верхней части рисунка показана диаграмма классов, а в нижней части — диаграмма объектов.

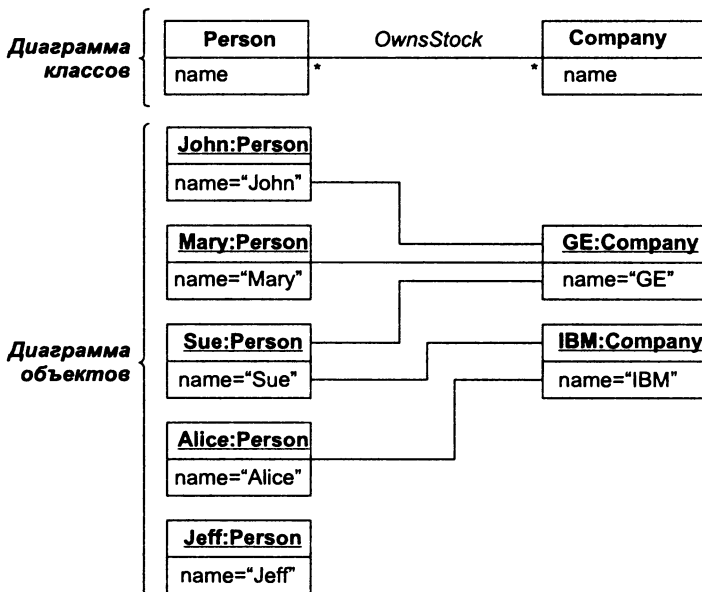


Рис. 3.7. Ассоциация и связи

Диаграмма классов показывает, что *человек* может быть владельцем акций неопределенного количества *компаний* (от нуля и более). С другой стороны, *владельцами акций* одной *компании* могут быть несколько *человек*. На диаграмме объектов показаны примеры. *Джон*, *Мэри* и *Сью* являются владельцами акций *General Electric*. *Сью* и *Элис* являются владельцами акций компании *IBM*. *Джефф* не купил никаких акций, а потому и связи для него нет. Звездочка на нашем рисунке — это символ кратности. Кратность (multiplicity) определяет количество экземпляров одного класса, которые могут быть связаны с одним экземпляром другого класса. Эта концепция обсуждается в следующем разделе.

Система обозначений UML предписывает изображать связь как линию между двумя объектами. Линия может состоять из нескольких прямолинейных сегментов. Если у связи есть имя, оно подчеркивается. В нашем примере Джон является владельцем акций *General Electric*. Ассоциация соединяет между собой классы и тоже обозначается линией (которая тоже может иметь несколько прямолинейных сегментов). В нашем примере люди являются владельцами акций компаний. По нашему соглашению названия связей и ассоциаций выделяются курсивом, а сегменты линий привязываются к прямоугольной сетке. Удобно по возможности расставлять классы так, чтобы ассоциация читалась слева направо.

Название ассоциации указывать не обязательно, если в модели не возникает двусмысленности. Неоднозначность появляется в тех случаях, когда между одними и теми же классами существует несколько ассоциаций. (Пример: *человек работает на компанию* и *человек является владельцем акций компании*). В этом случае необходимо использовать имена ассоциаций или имена полюсов ассоциаций (см. раздел 3.2.3).

Ассоциации по сути своей являются двусторонними. Название бинарной ассоциации обычно читается в конкретном направлении, но сама ассоциация может быть прослежена в любом направлении. Например, ассоциация *работает на* соединяет *человека* и *компанию*. Обратная ассоциация могла бы называться *оплачивает услуги*, и она соединяла бы *компанию* и *человека*. В реальности оба направления ассоциации имеют одинаково важное значение и относятся к одной и той же ассоциации. Только названия ассоциаций задают им определенные направления.

Разработчики часто реализуют ассоциации в виде ссылок из одного объекта на другой. Ссылка (reference) — это атрибут объекта, ссылающийся на другой объект. Например, структура данных класса *Person* (Человек) может содержать атрибут *employer* (работодатель), ссылающийся на объект класса *Company* (Компания), а объект класса *Company* может содержать атрибут *employees* (сотрудники), ссылающийся на множество объектов класса *Person*. Реализация ассоциаций в виде ссылок вполне приемлема, но моделировать ассоциации таким образом не следует.

Связь — это отношение между объектами. Моделирование связи в виде ссылки скрывает тот факт, что связь не является частью одного из объектов, а зависит от них обоих. Компания не является частью человека, а человек — частью компании. Более того, использование пары ссылок (например, ссылку из *Person* на *Company* и из *Company* на *Person*) скрывает тот факт, что прямая и обратная ссылки зависят друг от друга. Поэтому все соединения между классами следует моделировать как ассоциации, даже при разработке проектов программ.

В объектно-ориентированной литературе подчеркивается важность инкапсуляции, которая состоит в скрытии деталей реализации внутри класса. Ассоциации особенно важны потому, что они нарушают инкапсуляцию. Ассоциации не могут быть скрыты внутри класса, поскольку они соединяют разные классы. Ассоциации должны считаться равноправными по отношению к классам, в противном случае в программах будут содержаться скрытые предположения и зависимости. Такие программы трудно расширять, а классы, из которых они состоят, сложно использовать повторно.

Хотя при моделировании ассоциации считаются двусторонними, не обязательно реализовывать их в обоих направлениях. Ассоциации могут быть реализованы как ссылки, особенно если вам предстоит проследживать их только в одном направлении. В главе 17 рассматриваются некоторые вопросы, связанные с реализацией ассоциаций.

3.2.2. Кратность

Кратность (multiplicity) — это количество экземпляров одного класса, которые могут быть связаны с одним экземпляром другого класса через одну ассоциацию. Кратность ограничивает количество связанных между собой объектов. В литературе чаще всего рассматриваются два значения кратности: 1 и «много», но в общем случае кратность может быть потенциально бесконечным подмножеством неотрицательных целых чисел. На диаграммах UML кратность указывается явно около конца линии, которой обозначается ассоциация. Значение кратности указывается в виде диапазона, например «1» (ровно один), «1..*» (один и более) или «3..5» (от трех до пяти включительно). Специальный символ «*» обозначает слово «много» (many) — ноль и более.

Рисунок 3.7 демонстрирует ассоциацию многие-к-многим (кратность «много» на обоих концах). *Человек может владеть акциями произвольного количества компаний. Владельцами акций конкретной компании может быть произвольное число людей.* В нашем конкретном случае Джон и Мэри владеют акциями GE, Элис владеет акциями IBM, Сью является держателем акций обеих компаний, а Джефф вообще никаких акций не покупал. Акции GE принадлежат троим, а IBM — двоим держателям.

На рис. 3.8 показана ассоциация один-к-одному и соответствующие ей связи. В каждой стране есть ровно одна столица. Столица относится только к одному государству. (Вообще говоря, в некоторых странах столиц может быть несколько, например в Нидерландах и Швейцарии. Если это важно, модель можно изменить, поменяв кратность или создав отдельную ассоциацию для каждого типа столиц.)

На рис. 3.9 приведен пример кратности ноль-или-один. Одно из окон рабочей станции может быть выделено для сообщений об ошибках. Однако можно и не выделять такого окна. (Слово *console* на диаграмме — это название полюса ассоциации. См. раздел 3.2.3.)

Не следует путать кратность с количеством элементов. *Кратность* (multiplicity) — это ограничение на размер совокупности, а *количество элементов* (cardinality) — это число элементов, которые фактически входят в совокупность. Следовательно, кратность ограничивает количество элементов.

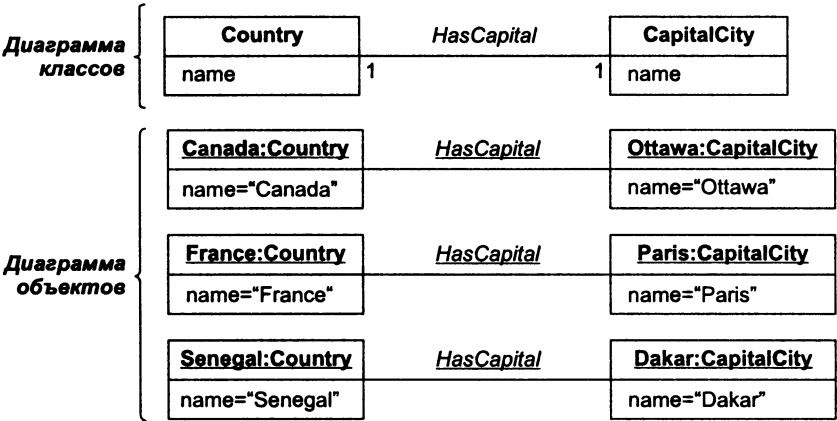


Рис. 3.8. Кратность определяет количество экземпляров одного класса, которые могут быть связаны с экземпляром другого класса

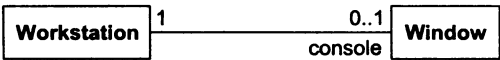


Рис. 3.9. Участие объекта в ассоциации не является обязательным

Кратность «много» указывает, что объект может быть связан с произвольным количеством объектов. Однако для каждой ассоциации между конкретной парой объектов может существовать только одна связь (за исключением мультимножеств и последовательностей — см. раздел 3.2.5). Если между двумя объектами должно быть две связи, необходимо создать две ассоциации (рис. 3.10 и 3.11).

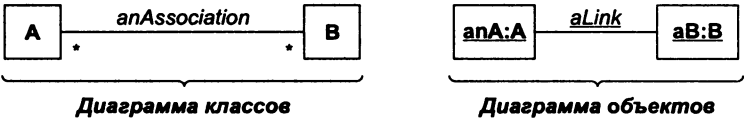


Рис. 3.10. Пара объектов может иметь только одну связь для данной ассоциации

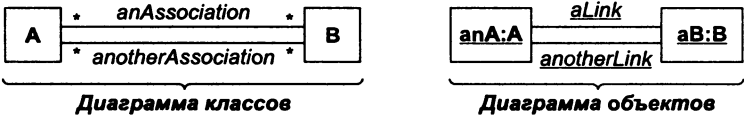


Рис. 3.11. Несколько связей между двумя объектами можно смоделировать несколькими ассоциациями

Кратность зависит от предположений и от определенных разработчиком границ задачи. Нечеткие требования часто затрудняют определение кратности. Не следует сильно беспокоиться о правильных значениях кратности на начальных этапах разработки системы. Сначала следует определить классы и ассоциации, а затем указывать кратность. Если кратность не указана на диаграмме, она считается неопределенной.

Кратность часто показывает скрытые допущения, на которых основана модель. Например, какую кратность имеет ассоциация *работает на* между *Человеком* и *Компанией*? Один-к-многим или многие-к-многим? Все зависит от контекста. Приложение для вычисления налогов, подлежащих уплате, должно допускать занятость одного человека в нескольких компаниях. С другой стороны, с точки зрения профсоюза занятость на других работах может не иметь значения. Диаграммы классов позволяют выявить эти скрытые предположения и проанализировать их.

Сильнее всего отличаются друг от друга значения «один» и «многo». Недооценивая кратность, вы можете ограничить гибкость приложения. Например, многие программы не позволяют указывать несколько телефонных номеров для одного человека. С другой стороны, переоценка кратности вызывает дополнительные расходы и требует указания дополнительной информации для того, чтобы различать членов множества. В настоящей иерархической организации «босс» должен иметь кратность «ноль или один», и не следует тратить ресурсы на несуществующую матричную систему управления.

3.2.3. Имена полюсов ассоциации

Кратность неявно подразумевает наличие *полюсов ассоциации* (association end). Например, ассоциация один-к-многим имеет два полюса, у одного из которых указана кратность «один», а у другого — «многo». Концепция полюса ассоциации — одна из важнейших в UML. Полюс ассоциации может иметь не только кратность, но и свое собственное имя. (В главе 4 рассматриваются дополнительные свойства полюсов ассоциации.)

Имена полюсов ассоциаций часто присутствуют в описаниях задач в виде существительных. Имя полюса указывается около конца ассоциации (рис. 3.12). На нашем рисунке *Person* (Человек) и *Company* (Компания) участвуют в ассоциации *WorksFor* (РаботаетНа). Человек по отношению к компании является сотрудником (employee), а компания по отношению к человеку — работодателем (employer). Использование имен полюсов ассоциаций не является обязательным, но чаще всего оказывается проще указывать имена полюсов вместо имен ассоциаций или, по крайней мере, вместе с ними.

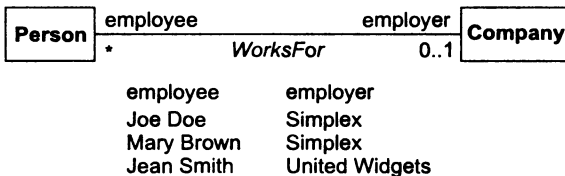


Рис. 3.12. Каждый полюс ассоциации может иметь имя

Имена полюсов ассоциаций особенно удобны для прослеживания ассоциаций, потому что каждый из них может рассматриваться как псевдоатрибут. Каждый полюс бинарной ассоциации ссылается на объект или множество объектов, связанных с исходным объектом. С точки зрения исходного объекта, прослеживание

ассоциации — это операция, которая возвращает связанные с ним объекты. Имя полюса ассоциации — это средство для прослеживания ассоциации без явного ее указания. В разделе 3.5 мы расскажем о прослеживании ассоциаций в моделях классов более подробно.

Имена полюсов ассоциаций обязательны для установления ассоциаций между двумя объектами одного и того же класса. Например, на рис. 3.13 имена *container* и *contents* позволяют различить два случая использования класса *Directory* (Каталог), который имеет ассоциацию с самим собой. Каталог может содержать вложенные каталоги и даже может содержаться сам в себе. Имена полюсов ассоциаций позволяют различать между собой разные ассоциации между одними и теми же классами. На рис. 3.13 каждый каталог имеет ровно одного пользователя, который является владельцем, и множество пользователей, которые имеют право работать с каталогом. Если между парой классов существует только одна ассоциация, имени классов может быть вполне достаточно, поэтому имена полюсов можно не указывать.

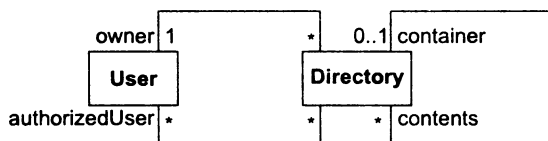


Рис. 3.13. Имена полюсов ассоциации

Имена полюсов позволяют унифицировать несколько ссылок на один и тот же класс. При построении диаграмм классов следует корректно использовать имена полюсов ассоциаций и не вводить отдельный класс для каждой ссылки (рис. 3.14). В некорректной модели *Человек с Ребенком* может быть представлен двумя экземплярами: один для родителя и один для ребенка. В корректной модели экземпляр *Человека* принимает участие в двух и более связях: дважды в качестве родителя и произвольное количество раз в качестве ребенка. В корректной модели нужно показать, что ребенок не обязательно должен иметь родителя, чтобы рекурсия могла быть прервана.

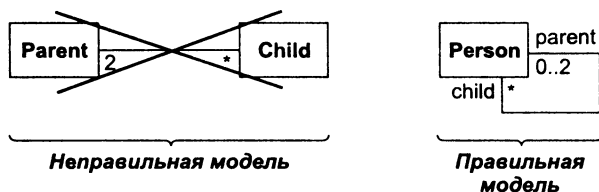


Рис. 3.14. Моделирование ссылок на один и тот же класс

Поскольку имена полюсов ассоциации позволяют отличать объекты друг от друга, все имена на дальнем полюсе ассоциации, прикрепленной к некоторому классу, должны быть уникальными. Хотя имя ставится около целевого объекта ассоциации, фактически оно является псевдоатрибутом исходного класса, а потому

должно быть уникально внутри него. По той же причине имя полюса ассоциации не должно совпадать с именем какого-либо атрибута исходного класса.

3.2.4. Упорядочение

Достаточно часто объекты у полюса ассоциации с обозначением кратности «много» не имеют никакого выраженного порядка. В этом случае их можно рассматривать как множество. На рис. 3.15 изображен экран рабочей станции с перекрывающимися окнами. Каждое окно присутствует на экране не более, чем в одном экземпляре. Окна имеют явный порядок, и в каждой конкретной точке экрана видимо только одно из них (верхнее). Упорядочение является внутренним свойством ассоциации. Упорядоченность множества объектов можно указать при помощи слова *{ordered}*, которое ставится около соответствующего полюса ассоциации.



Рис. 3.15. Упорядочение объектов у полюса ассоциации

3.2.5. Мультимножества и последовательности

Бинарная ассоциация обычно позволяет создать между парой объектов не более одной связи. Однако указав около полюса ассоциации слова *{bag}* или *{sequence}*, вы можете разрешить создание множества связей между двумя объектами. Мультимножество (*bag*) — это совокупность элементов, в которой допускается наличие дубликатов. Последовательность (*sequence*) — это упорядоченная совокупность элементов, в которой также допускается наличие дубликатов. *Маршрут* на рис. 3.16 — это последовательность аэропортов, причем один и тот же аэропорт можно посетить несколько раз. Обозначения *{bag}* и *{sequence}*, как и *{ordered}*, применимы только к бинарным ассоциациям.

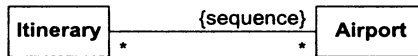


Рис. 3.16. Пример последовательности

В UML1 не допускалось создание множества связей между двумя объектами. Некоторые разработчики моделей неправильно понимали это требование в случае упорядоченных полюсов ассоциаций и создавали некорректные модели, допуская возможность существования нескольких связей между двумя объектами. В UML2 разработчик получает возможность ясно выразить свои мысли. Если около полюса указано слово *{bag}* или *{sequence}*, связей между двумя объектами может быть несколько. Если же этих указаний нет, связь может быть только одна.

Обратите внимание, что разница между *{ordered}* и *{sequence}* состоит в том, что первое обозначение запрещает наличие дубликатов, а второе — разрешает.

Последовательность — это упорядоченное мультимножество, тогда как упорядоченность подразумевается по отношению к обычному множеству.

3.2.6. Классы ассоциаций

Подобно тому, как объекты класса могут быть описаны при помощи атрибутов, связи ассоциации также могут быть описаны атрибутами. UML позволяет представлять информацию такого характера при помощи классов ассоциаций. Класс ассоциации — это ассоциация, которая одновременно является классом. Подобно связям ассоциации, экземпляры класса ассоциации обладают индивидуальностью, связанной с теми объектами, между которыми они проводятся. Подобно обычным классам, классы ассоциаций могут иметь атрибуты и операции и участвовать в ассоциациях. Классы ассоциаций присутствуют в формулировке задачи в виде наречий или получаются абстрагированием известных значений.

На рис. 3.17 *accessPermission* (разрешениеДоступа) является атрибутом *AccessibleBy* (Доступно). Данные в нижней части рисунка показывают возможные значения каждой связи. В UML класс ассоциации обозначается прямоугольником, прикрепленным к ассоциации пунктирной линией.

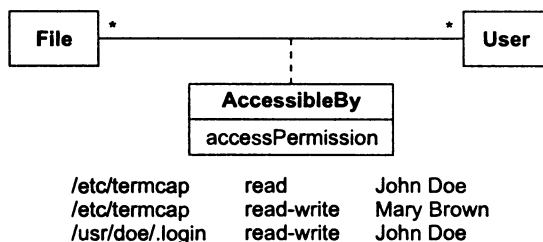


Рис. 3.17. Связи ассоциации могут обладать атрибутами

Основанием для введения классов ассоциаций послужила возможность создания ассоциаций типа многие-ко-многим. Атрибуты таких ассоциаций без всяких сомнений являются принадлежностью связей и не могут быть приписаны ни к одному из объектов-участников. На рис. 3.17 атрибут *accessPermission* относится к файлу и пользователю одновременно и не может быть прикреплен только к одному из них без потери информации.

На рис. 3.18 показаны атрибуты двух ассоциаций типа один-ко-многим. Каждый человек, работающий на компанию, получает зарплату и занимает некоторую должность. Босс оценивает работу каждого сотрудника. Атрибуты могут присутствовать и в ассоциациях типа один-к-одному.

На рис. 3.19 показано, каким образом можно упаковать атрибуты ассоциации типа один-к-одному или один-ко-многим в класс, противоположный полюсу с кратностью «один». Для ассоциаций типа многие-ко-многим это сделать невозможно. Как правило, упаковывать атрибуты ассоциации в файл все же не рекомендуется, поскольку кратность ассоциации может измениться. Оба варианта, изображенных на рис. 3.19, допустимы для ассоциации типа один-ко-многим, однако только первая форма останется корректной, если кратность ассоциации *WorksFor* изменится на многие-ко-многим.

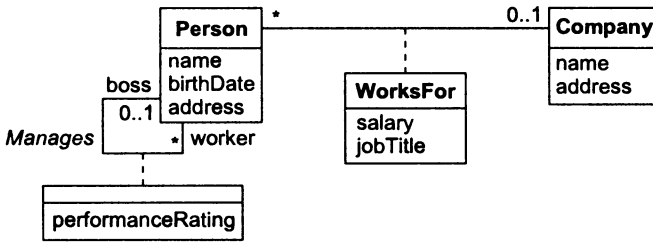


Рис. 3.18. Атрибуты в ассоциациях типа один-ко-многим

На рис. 3.20 показан класс ассоциации, участвующий в другой ассоциации. Пользователи могут проходить авторизацию на нескольких рабочих станциях. Каждая авторизация обладает приоритетом и правами доступа. Пользователь имеет домашний каталог на всех рабочих станциях, где он авторизован, но у нескольких рабочих станций и пользователей может быть один домашний каталог. Классы ассоциаций являются важным аспектом модели классов, поскольку они позволяют точно указать индивидуальности и маршруты навигации.

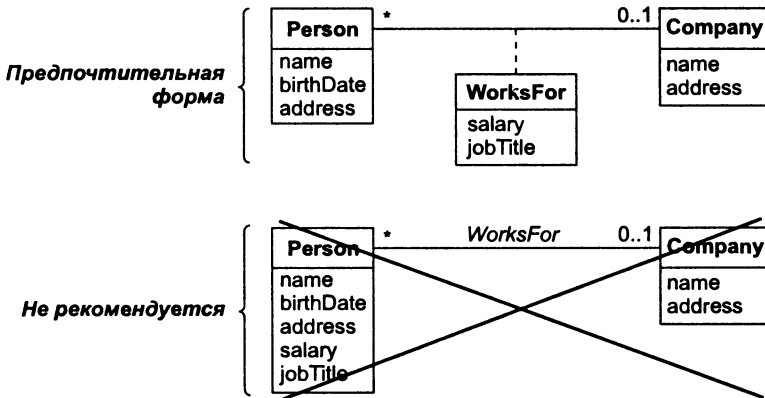


Рис. 3.19. Не следует упаковывать атрибуты ассоциации в класс

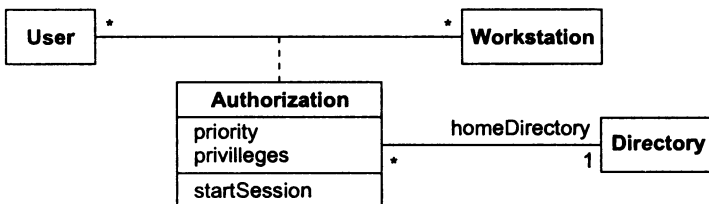


Рис. 3.20. Класс ассоциации позволяет четко указать индивидуальности и маршруты навигации

Не следует путать классы ассоциаций с ассоциациями, которые были выделены в отдельные классы. Разница демонстрируется на рис. 3.21. Класс ассоциации порождает один-единственный экземпляр для каждой пары экземпляров человека и компании. Напротив, экземпляров покупок (акций) между одним человеком

и одной компанией может быть сколько угодно. Каждая покупка обладает собственной индивидуальностью и собственными значениями количества, даты и стоимости.

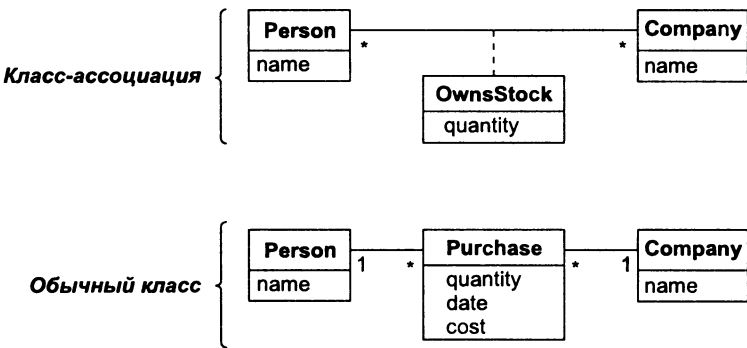


Рис. 3.21. Классы ассоциаций существенно отличаются от обычных классов

3.2.7. Квалифицированные ассоциации

Квалифицированной называется ассоциация, у которой имеется специальный атрибут (квалификатор), используемый для того, чтобы отличать друг от друга объекты, находящиеся на полюсе ассоциации с кратностью «много». Квалификаторы могут быть определены для ассоциаций типа один-ко-многим и многие-ко-многим. Квалификатор позволяет выбрать отдельный объект из множества целевых объектов, уменьшая таким образом эффективную кратность до значения «один». Квалифицированная ассоциация с целевой кратностью «один» или «не более одного» образует четкий маршрут для поиска целевого объекта по исходному.

На рис. 3.22 демонстрируется наиболее типичный пример использования квалификатора для ассоциаций с кратностью один-ко-многим. *Банк обслуживает множество счетов. Счет принадлежит одному-единственному банку.* В контексте банка уникальный счет определяется своим номером. *Банк* и *Счет* — это классы, а *НомерСчета* — квалификатор. Квалификация уменьшает эффективную кратность ассоциации до единицы.

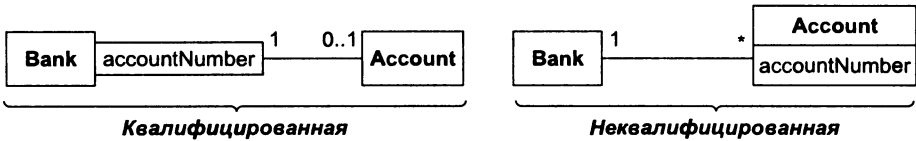


Рис. 3.22. Квалифицированная ассоциация

Обе модели (квалифицированная и неквалифицированная) вполне корректны, но модель с квалифицированной ассоциацией сообщает дополнительную информацию. В квалифицированной модели добавляется ограничение на кратность: сочетание банка и номера счета дает не более одного счета. Квалифицированная модель также передает значение номера счета для прослеживания модели, что

отражается в методах. Сначала следует найти банк, а затем указать номер счета, и в результате вы получаете нужный счет.

Для обозначения квалификатора используется небольшой прямоугольник, который пристыковывается к исходному классу около конца линии, обозначающей ассоциацию. Квалификатор может быть пристыкован к любой стороне прямоугольника исходного класса. Исходный класс вместе с квалификатором определяют целевой класс. На рис. 3.22 *Bank* и *accountNumber* вместе дают *Account*, а потому *accountNumber* указывается в прямоугольнике, пристыкованном к объекту *Bank*.

Другой пример применения квалификатора приведен на рис. 3.23. На бирже представлены многие компании. Однако каждая из них имеет свой собственный код ценных бумаг. *Компания* может *присутствовать* на разных *биржах* с разными *кодами*. (Мы предполагаем, что это утверждение истинно. Если бы на всех биржах система кодов была единой, мы бы сделали *tickerSymbol* атрибутом класса *Company*.)

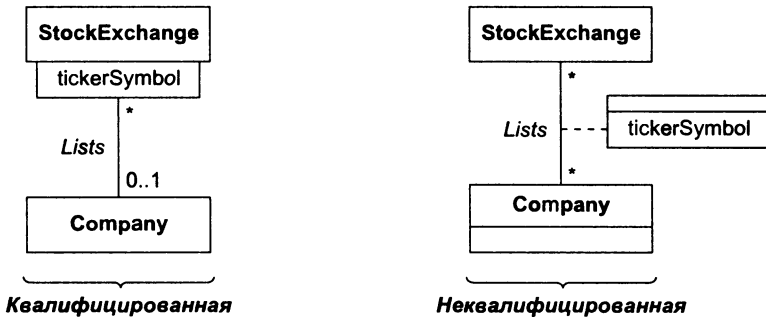


Рис. 3.23. Квалификатор облегчает прослеживание моделей классов

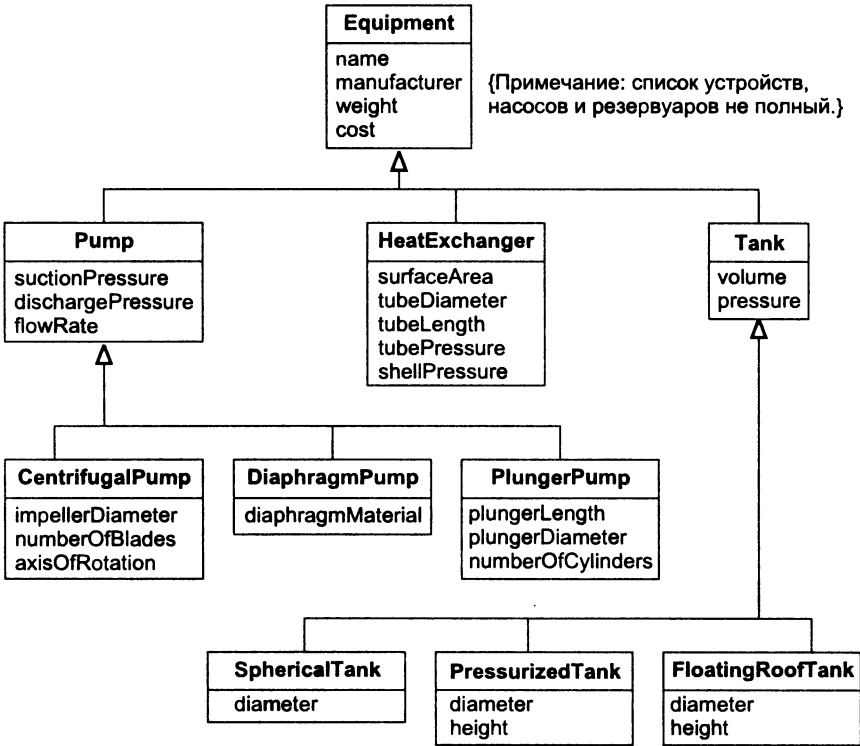
3.3. Обобщение и наследование

3.3.1. Определения

Обобщение (generalization) — это отношение между классом (суперклассом) и одной или несколькими его вариациями (подклассами). Обобщение объединяет классы по их общим свойствам, благодаря чему обеспечивается структурирование описания объектов. Суперкласс характеризуется общими атрибутами, операциями и ассоциациями. Подклассы добавляют к ним свои собственные атрибуты, операции и ассоциации. Говорят, что подкласс наследует составляющие суперкласса. Обобщение иногда называется отношением типа «является», поскольку каждый экземпляр подкласса одновременно *является* экземпляром суперкласса.

Простое обобщение упорядочивает классы в рамках некоторой иерархии. В этом случае каждый подкласс имеет одного непосредственного предка (его суперкласс). В главе 4 рассматривается более сложная форма обобщения, при которой подкласс может иметь несколько непосредственных суперклассов. Уровней обобщения может быть много.

Примеры обобщений приведены на рис. 3.24. Оборудование может быть насосом, теплообменником или резервуаром. Насосы бывают нескольких типов: центробежные, мембранные и плунжерные. Резервуары бывают сферические, с наддувом и с плавающей крышей. То, что символ обобщения резервуара изображен ниже символа обобщения насоса, никакого особого значения не имеет. В нижней части рисунка изображены несколько объектов. Каждый объект наследует составляющие от одного класса с каждого уровня иерархии обобщений. Поэтому объект P101 обладает составляющими оборудования, насоса и мембранного насоса. Объект E302 обладает составляющими оборудования и теплообменника.



P101:DiaphragmPump name = "P101" manufacturer = "Simplex" weight = 100 kg cost = \$5000 suctionPres = 1.1 atm dischargePres = 3.3 atm flowRate = 300 l/hr diaphragmMatl = Teflon	E302:HeatExchanger name = "E302" manufacturer = "Brown" weight = 5000 kg cost = \$20000 surfaceArea = 300 m ² tubeDiameter = 2 cm tubeLength = 6 m tubePressure = 15 atm shellPressure = 1.7 atm	T111:FloatingRoofTank name = "T111" manufacturer = "Simplex" weight = 10000 kg cost = \$50000 volume = 400000 liter pressure = 1.1 atm diameter = 8 m height = 9 m

Рис. 3.24. Многоуровневая иерархия наследования с экземплярами

Обобщение обозначается большой незакрашенной стрелкой. Стрелка указывает на суперкласс. Суперкласс можно соединять с каждым из его подклассов непосредственно, но мы предпочитаем группировать обобщения в дерево. Треугольник можно повернуть и расположить его с любой стороны от суперкласса, но по возможности следует изображать суперкласс сверху, а его подклассы — снизу. Комментарии UML, указывающие на наличие дополнительных подклассов, не приведенных на диаграмме, ставятся в фигурных скобках.

Обобщение транзитивно и действует через произвольное количество уровней иерархии. Термины *предок* (ancestor) и *потомок* (descendant) используются для описания классов, находящихся далеко друг от друга по уровням, но связанных отношением обобщения. Экземпляр подкласса одновременно является экземпляром всех его предков. Экземпляр обладает значениями всех атрибутов всех классов-предков. Экземпляр может вызывать любую операцию, указанную у любого из его предков. Подклассы не только наследуют все составляющие своих предков, но и добавляют к ним свои собственные составляющие. Например, *Pump* добавляет атрибуты *suctionPressure*, *dischargePressure* и *flowRate*, отсутствующие у оборудования других классов.

На рис. 3.25 изображены классы геометрических фигур. Этот пример лежит ближе к программированию и демонстрирует наследование операций. Операции *move*, *select*, *rotate* и *display* наследуются всеми подклассами. Операция *scale* применяется только к одномерным и двумерным фигурам. Операция *fill* применима только к двумерным фигурам.

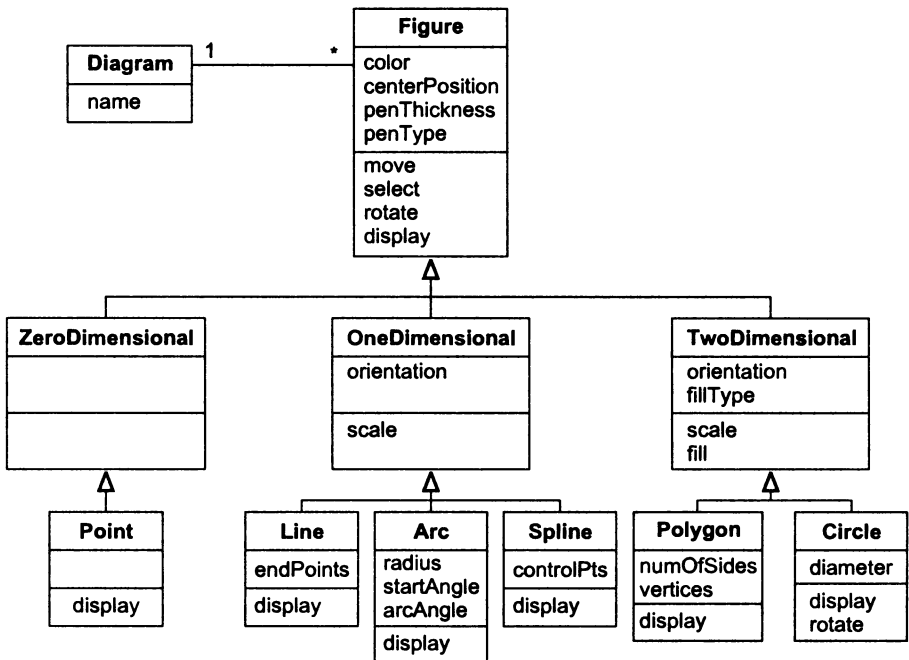


Рис. 3.25. Наследование на примере геометрических фигур

Слово, написанное на диаграмме рядом с линией, обозначающей обобщение, — это имя *набора обобщений* (dimensionality — размерность). Имя набора обобщений — это перечислимый атрибут, показывающий, какой аспект объекта абстрагируется конкретным обобщением. Каждый набор должен абстрагировать только один аспект. Например, аспектами обобщений для класса *ТранспортноеСредство* являются *источник энергии* (ветер, горючее, животное, сила тяжести) и *область передвижения* (земля, воздух, вода, безвоздушное пространство). Значения наборов обобщений находятся во взаимно-однозначном соответствии с подклассами обобщений. Название набора обобщений указывать не обязательно.

Не следует создавать слишком глубокую иерархию подклассов. Глубоко вложенные подклассы могут затруднять восприятие модели, точно так же, как глубоко вложенные участки кода в процедурных языках. Небольшое реструктурирование часто помогает уменьшить глубину иерархии наследования. На практике подходящая глубина вложенности определяется вкусом разработчика. Руководствуйтесь следующими соображениями: двух или трехуровневая иерархия наверняка приемлема, десять уровней — наверняка чересчур, а пять-шесть уровней может быть как приемлемо, так и неприемлемо — в зависимости от системы.

3.3.2. Использование обобщения

Обобщение служит трем основным целям. Первая цель — обеспечение поддержки полиморфизма. Операция может быть вызвана на уровне суперкласса, а компилятор объектно-ориентированного языка автоматически разрешит вызов в метод, соответствующий классу вызывающего объекта. Полиморфизм увеличивает гибкость программного обеспечения: вы добавляете новый подкласс и автоматически наследуете поведение суперкласса. Более того, новый подкласс не нарушает работу существующего кода. Сравните это с процедурным программированием, где добавление нового типа может вызвать значительные побочные эффекты.

Вторая цель обобщения состоит в структурировании описаний объектов. Используя обобщение, вы делаете концептуальное утверждение, образуя таксономию и упорядочивая объекты на основании их сходств и различий. Такой подход гораздо более основателен, нежели моделирование каждого класса в изоляции от всех остальных.

Третья цель состоит в обеспечении повторного использования кода: вы можете наследовать код в рамках одного приложения, а также из существующих приложений (например, из библиотек классов). Повторное использование повышает производительность по сравнению с многократным переписыванием кода «с нуля». Обобщение позволяет корректировать код для получения именно такого поведения, которое требуется в данном приложении. Повторное использование является веским доводом в пользу обобщений, однако его преимущества часто переоценивают (см. главу 14).

Термины *обобщение* (generalization), *конкретизация* (specialization) и *наследование* (inheritance) описывают одну и ту же концепцию. Обобщение и конкретизация — это отношения между классами, противоположные друг другу по смыслу. Обобщение подчеркивает, что суперкласс обобщает подклассы. Конкретизация

подчеркивает, что подклассы конкретизируют (уточняют) суперкласс. Наследование — это механизм совместного использования атрибутов, операций и ассоциаций объектами, классы которых находятся в отношениях обобщения (конкретизации). На практике перепутать эти термины довольно сложно.

3.3.3. Подмена составляющих

Подкласс может *подменять* или *перекрывать* (override) составляющую суперкласса, определяя составляющую с тем же именем внутри себя. Подменяющая составляющая (подкласса) уточняет и заменяет подмененную (суперкласса в подклассе). Подмена составляющих может потребоваться по нескольким причинам: для спецификации поведения, зависящего от подкласса, для уточнения спецификации составляющей, для повышения производительности. Например, на рис. 3.25 каждый подкласс-лист должен реализовать операцию *display*, несмотря на то, что она определена в классе *Figure*. Класс *Circle* повышает производительность, подменяя операцию *rotate* пустым блоком кода.

Подменять можно методы и значения по умолчанию для атрибутов. Никогда не следует подменять сигнатуру составляющей. Подмена должна сохранять тип атрибута, количество и тип аргументов операции, а также тип возвращаемого операцией значения. Уточнение типа атрибута или аргумента операции (то есть указание в качестве типа подтипа исходного типа) является формой ограничения и должно использоваться с осторожностью. Часто подмена используется для повышения производительности. В этом случае универсальный метод подменяется специальным, который использует имеющиеся дополнительные сведения, но не изменяет семантику операции (например, *Circle.rotate* на рис. 3.25).

Никогда не следует подменять составляющую, если в результате она окажется несогласованной с унаследованной от предка составляющей. Подкласс является частным случаем суперкласса и должен быть совместим с ним во всех отношениях. К сожалению, на практике объектно-ориентированные программисты часто заимствуют класс, похожий на тот, который им нужен, а затем изменяют его, модифицируя или игнорируя его составляющие, несмотря на то, что новый класс перестает быть частным случаем исходного класса. Такие действия могут приводить к концептуальным ошибкам и появлению скрытых предположений в программах.

3.4. Пример модели классов

На рис. 3.26 показана модель классов системы управления окнами для рабочей станции. Эта модель значительно упрощена (для реальной модели потребовалось бы несколько страниц), но она демонстрирует многие конструкции моделей классов и их совместное использование.

Класс *Window* определяет общие параметры окон всех типов, в частности прямоугольную границу, определяемую атрибутами *x1*, *y1*, *x2*, *y2*, и операции, позволяющие отображать и скрывать окно, а также помещать его на передний или на задний план в общем наборе окон.

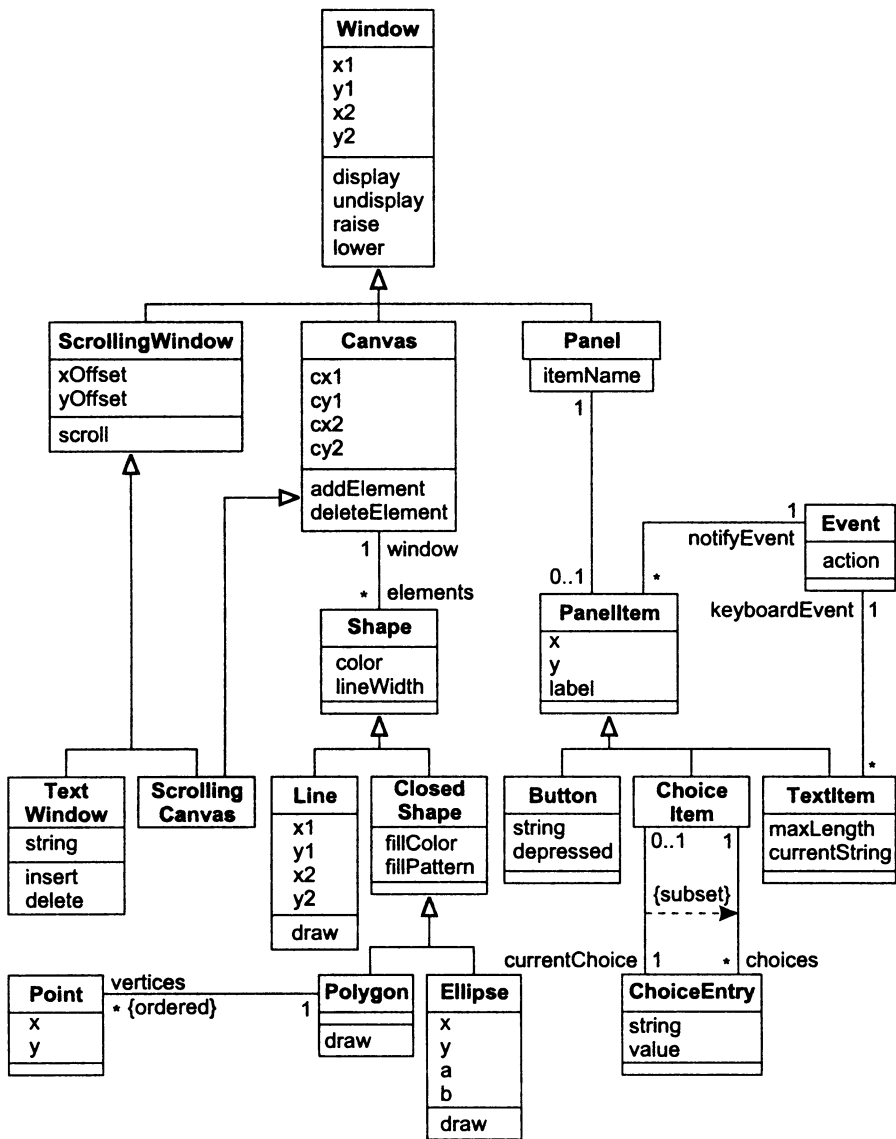


Рис. 3.26. Модель классов системы управления окнами

Холстом (canvas) называется область, в которой может отображаться графика. *Canvas* наследует границу от класса *Window* и добавляет размеры области холста, которые определяются атрибутами *cx1*, *cy1*, *cx2*, *cy2*. *Canvas* содержит множество элементов, на что указывает ассоциация с классом *Shape*. Все фигуры характеризуются цветом и шириной линии. Фигуры могут быть линиями, эллипсами или многоугольниками. Каждый класс фигур обладает своими собственными параметрами. Многоугольник (*Polygon*) определяется списком его вершин. Эллипсы

(*Ellipse*) и многоугольники являются замкнутыми фигурами, которые характеризуются цветом и шаблоном заливки. Линии (*Line*) являются одномерными, а потому не могут быть залиты. Холст поддерживает операции добавления и удаления элементов.

Текстовое окно *TextWindow* — это разновидность окна с полосами прокрутки *ScrollingWindow*, которое характеризуется двумя смещениями прокрутки *xOffset* и *yOffset* и поддерживает операцию *scroll*, изменяющую значения смещений. Текстовое окно содержит строку и поддерживает операции вставки и удаления символов. Холст с прокруткой (*ScrollingCanvas*) — особая разновидность холста, поддерживающая прокрутку. Это окно одновременно является холстом и окном с прокруткой. Таким образом, оно является собой пример множественного наследования, о котором речь пойдет в главе 4.

Панель (*Panel*) содержит множество объектов *PanelItem*, каждый из которых характеризуется уникальным в рамках одной панели именем *itemName*, о чем говорит наличие квалификатора у соответствующей ассоциации. Каждый элемент панели может принадлежать только одной панели. Такой элемент представляет собой предопределенный значок, посредством которого пользователь может взаимодействовать с системой. Элементы панели бывают трех видов: кнопки, переключатели и текстовые элементы. Кнопка (*Button*) характеризуется отображаемой на экране строкой, а также атрибутом *depressed*, который показывает, нажата она или нет. Переключатель (*Switch*) позволяет пользователю выбрать один из предопределенных вариантов, каждый из которых является объектом класса *ChoiceEntry*, характеризующимся отображаемой строкой и возвращаемым при его выборе значением. Между *ChoiceItem* и *ChoiceEntry* имеется две ассоциации: ассоциация один-ко-многим определяет множество доступных вариантов, а ассоциация один-к-одному определяет выбранный вариант. Выбранный вариант должен быть одним из доступных, поэтому одна ассоциация является подмножеством другой, на что указывает стрелка со словом *{subset}*. Это пример ограничения (см. главу 4).

Когда элемент панели выбирается пользователем, это порождает событие (*Event*), которое представляет собой объединение сигнала о происшествии с подлежащим выполнению действием. Все виды элементов панелей имеют ассоциации *notifyEvent*. Каждый элемент связан с одним событием, но одно и то же событие может порождаться несколькими элементами панели. Текстовые элементы порождают события другого типа, связанные с нажатием на клавишу клавиатуры, происходящим в тот момент, когда текстовый элемент является выбранным. Эти события указываются ассоциацией с именем полюса *keyboardEvent*. Текстовые элементы наследуют событие *notifyEvent* от суперкласса *PanelItem*. Событие *notifyEvent* порождается при выделении текстового элемента мышью.

Эта модель обладает множеством недостатков. Например, нам следовало бы определить тип *Rectangle*, который можно было бы использовать для определения границ окон и холстов, а не вводить два одинаковых набора из четырех атрибутов. Линия могла бы быть частным случаем полилинии (последовательности соединенных между собой линейных сегментов), и тогда полилиния и многоугольник были бы подклассами нового суперкласса, определяющего список точек. Здесь не хватает множества атрибутов, операций и классов, которые необходимы для реальной

оконной системы. Между окнами должны быть ассоциации, например, они могут перекрывать одно другое. Тем не менее эта простая модель даст вам некоторое ощущение того, что должна представлять собой модель классов. Мы можем критиковать ее, потому что она содержит в себе некоторые четкие утверждения. Такая модель могла бы послужить основой для создания более полной модели.

3.5. Навигация моделей классов

Мы показали, каким образом модель классов может выражать структуру приложения. Теперь мы покажем, что они могут выражать и поведение навигации по классам. Навигация достаточно важна, поскольку она позволяет вам проверить модель и обнаружить скрытые дефекты и недостатки, а затем устранить их. Навигация может осуществляться вручную (неформальная методика) или при помощи навигационных выражений (см. ниже).

Рассмотрим простую модель счетов для кредитных карт (рис. 3.27). Учреждение может открыть множество счетов для кредитных карт, каждый из которых будет идентифицироваться номером счета. Каждый счет характеризуется максимальным кредитом, текущим балансом и почтовым адресом. Счет может принадлежать одному или нескольким клиентам, проживающим по данному адресу. Учреждение периодически рассылает отчеты по счетам. В отчете указывается дата платежа, стоимость всех элементов кредита и минимальный платеж. Кроме того, перечисляются транзакции, которые были осуществлены за отчетный период: получение наличных, процентные платежи, покупки, штрафы и корректировки. Для каждой покупки указывается название магазина.

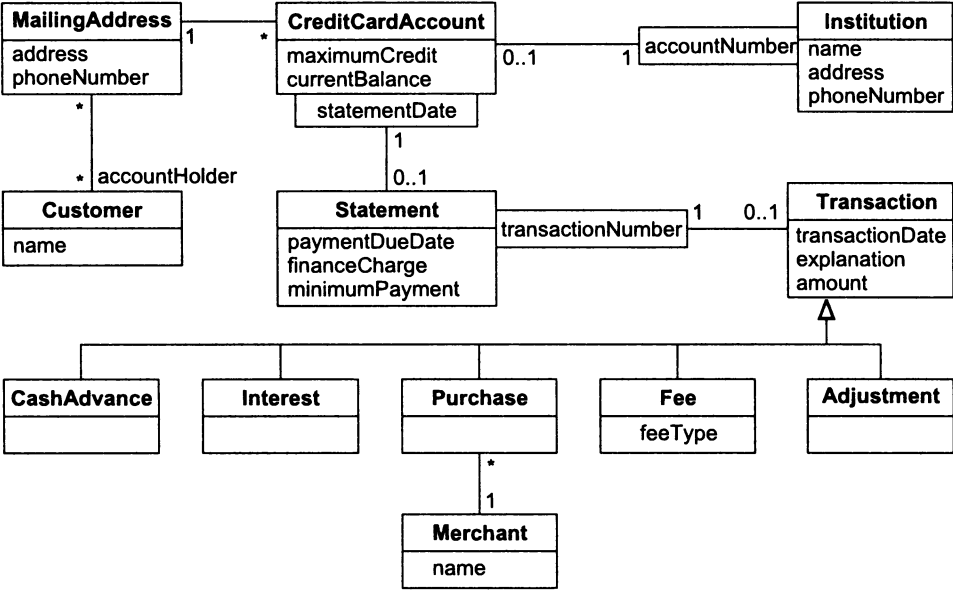


Рис. 3.27. Модель классов для управления счетами кредитных карт

Описанную модель можно протестировать, задавая различные вопросы.

- Какие транзакции были проведены по данному счету за определенный промежуток времени?
- Какой объем транзакций был обработан учреждением за последний год?
- Какие клиенты пользовались кредитными картами для оплаты покупок в конкретном магазине?
- Сколько счетов для кредитных карт имеет конкретный клиент в данный момент?
- Каков максимальный кредит для данного клиента по всем его счетам?

В состав UML входит специальный язык для выражения вопросов такого рода. Этот язык называется объектным языком ограничений (Object Constraint Language — OCL) [Warmer-99]. Мы опишем его в следующих двух разделах, а в разделе 3.5.3 переформулируем приведенные выше вопросы на OCL. Разумеется, мы не пытаемся дать полное описание этого языка, мы расскажем только о том, что нужно для прослеживания моделей.

3.5.1. Прослеживание моделей с помощью конструкций OCL

Язык OCL позволяет прослеживать конструкции в моделях классов.

- **Атрибуты.** OCL позволяет переходить от объекта к значениям его атрибутов. Используется следующий синтаксис: сначала указывается имя объекта, после которого ставится точка, а затем имя атрибута. Например, выражение *aCreditCardAccount.maximumCredit* означает значение атрибута *maximumCredit* объекта класса *CreditCardAccount*. Мы ставим перед именем класса артикль *a*, если хотим обозначить произвольный объект данного класса. Тем же способом можно обратиться к атрибуту каждого объекта в совокупности, получив совокупность значений атрибутов. Можно получить значение атрибута для связи или совокупность значений атрибутов для совокупности связей.
- **Операции.** Можно вызвать операцию для некоторого объекта или совокупности объектов. Используется следующий синтаксис: сначала указывается имя объекта или совокупности объектов, после которого ставится точка, а затем имя операции. За именем операции должны следовать круглые скобки, даже если у данной операции нет никаких аргументов. Это необходимо для того, чтобы иметь возможность отличать операции от атрибутов. Вы можете вызывать не только операции, определенные в вашей модели классов, но и предопределенные операции языка OCL.

В OCL имеются специальные операции, которые применяются к совокупности в целом (а не к отдельным объектам этой совокупности). Например, вы можете посчитать количество объектов совокупности или просуммировать совокупность численных значений. Для таких операций используется следующий синтаксис: сначала указывается имя совокупности объектов, затем ставится знак «->», после которого указывается имя операции.

- **Простые ассоциации.** Точка используется также для прослеживания ассоциации и перехода к целевому полюсу. Целевой полюс может быть задан его именем или именем класса, если это не создает неоднозначности. В нашем примере выражение *aCustomer.MailingAddress* возвращает множество адресов клиента (целевой полюс обладает кратностью «много»). С другой стороны, выражение *aCreditCardAccount.MailingAddress* возвратит один адрес (целевой полюс обладает кратностью «один»).
- **Квалифицированные ассоциации.** Квалификатор делает прослеживание ассоциаций более точным. Выражение *aCreditCardAccount.Statement[30 November 1999]* возвращает отчет по счету для кредитной карты, датированный 30 ноября 1999 года. Значение квалификатора указывается в квадратных скобках. Квалификатор можно игнорировать. При этом квалифицированная ассоциация будет прослеживаться так же, как и обычная. Таким образом, выражение *aCreditCardAccount.Statement* возвратит все отчеты для данного счета. (Когда квалификатор не используется, кратность принимает значение «много».)
- **Классы ассоциаций.** По связи, представляющей собой экземпляр класса ассоциации, можно найти участвующие в этой связи объекты. И наоборот: по участвующему в связи объекту можно найти связи, являющиеся экземплярами класса ассоциации.
- **Обобщения.** Прослеживание иерархии обобщений встроено в систему обозначений OCL неявным образом.
- **Фильтры.** Часто возникает необходимость отфильтровать объекты из некоторого множества. В OCL имеются фильтры нескольких типов. Наиболее широко используется операция *select*. Эта операция применяет некоторый предикат к каждому элементу совокупности и возвращает все элементы, удовлетворяющие данному предикату. Например, выражение *aStatement.Transaction->select(Amount>\$100)* возвращает все транзакции одного отчета, величина которых превышает 100 долларов.

3.5.2. Построение выражений OCL

Эффективность OCL обусловлена возможностью построения сложных выражений из простых конструкций. Например, выражение OCL может последовательно прослеживать несколько ассоциаций. В нем может быть несколько квалификаторов, фильтров, операторов и т. д.

Прослеживание одной ассоциации от объекта возвращает одно- или многоэлементное множество (или мультимножество, если ассоциация является мультимножеством или последовательностью). В общем случае прослеживание нескольких ассоциаций может дать в результате мультимножество (в зависимости от кратности этих ассоциаций), поэтому с выражениями OCL следует обращаться аккуратно. Множество — это совокупность уникальных элементов, а мультимножество допускает наличие дубликатов.

Пример на рис. 3.28 иллюстрирует тот факт, что выражение OCL может вернуть мультимножество. Предположим, что компания хочет отправить по

одному письму на каждый адрес, принадлежащий какому-либо держателю акций. Мы начинаем проследживать ассоциации с компании GE. Проследживание ассоциации *OwnsStock* дает нам множество из трех человек. Проследживая ассоциацию от этих троих к их адресам, мы получаем мультимножество, в котором адрес 456 State встречается дважды.

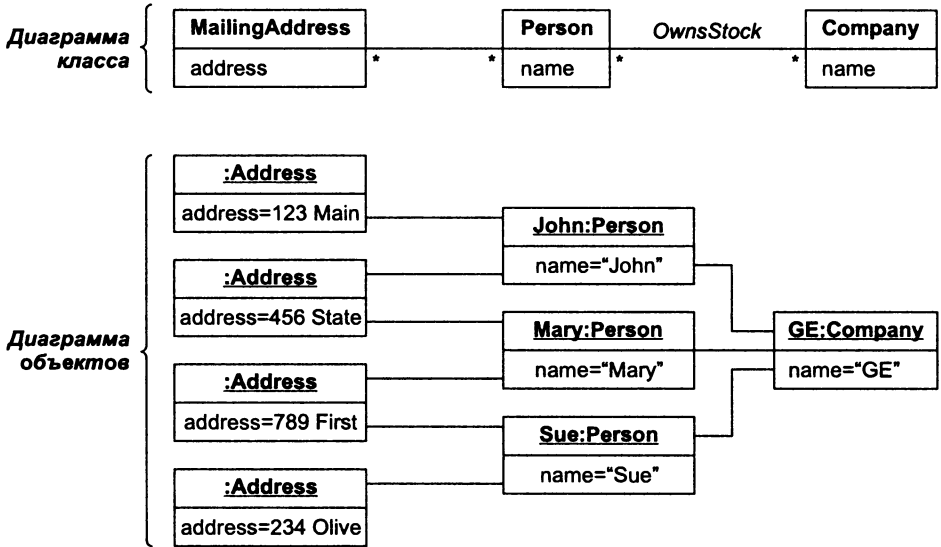


Рис. 3.28. Проследживание нескольких ассоциаций может вернуть мультимножество

В книге [Warmer-99] ничего не говорится о пустых значениях, поскольку там обсуждаются только вопросы, связанные со спецификацией ограничений на корректно реализованную систему. (*Null* — это особое значение, указывающее на то, что значение атрибута неизвестно или неприменимо.) В этой книге ничего не говорится об обработке исключений и ошибок времени выполнения.

В этой главе мы хотим обсудить навигацию по классам, а не спецификацию ограничений. Пустые значения не появляются в правильных ограничениях, но при навигации по моделям они встречаются достаточно часто. Например, у человека может не быть почтового адреса. Мы расширяем определение выражений OCL, допуская существование пустых значений. Проследживание может вернуть пустое значение, и выражение OCL возвратит вам то же значение, если объект окажется пустым.

3.5.3. Примеры выражений OCL

Воспользуемся языком OCL, чтобы ответить на поставленные выше вопросы для модели, описывающей счета для кредитных карт.

- Какие транзакции были проведены по данному счету за определенный промежуток времени?

```

aCreditCardAccount.Statement.Transaction->select(aStartDate <=
transactionDate and transactionDate <= anEndDate)
  
```

В данном выражении осуществляется переход от объекта класса *CreditCardAccount* к объекту *Statement*, а затем к *Transaction*, в результате чего возвращается множество транзакций. Мы получаем именно множество, а не мультимножество, потому что обе ассоциации принадлежат к типу один-ко-многим. Затем при помощи оператора OCL *select*, применяемого к множеству в целом, мы находим транзакции, осуществлявшиеся на заданном интервале, ограниченном датами *aStartDate* и *anEndDate*.

- Какой объем транзакций был обработан учреждением за последний год?

```
anInstitution.CreditCardAccount.Statement.Transaction->select(aStartDate
<= transactionDate and transactionDate <= anEndDate).amount->sum()
```

В выражении осуществляется переход от объекта *Institution* к *CreditCardAccount*, затем к *Statement*, а затем к *Transaction*. В результате получается множество, а не мультимножество, потому что все ассоциации имеют тип один-ко-многим. Оператор *select* выбирает транзакции, попадающие во временной интервал между *aStartDate* и *anEndDate*. (Мы решили сделать запрос более общим, нежели просто «за последний год».) Затем мы определяем сумму каждой из транзакций и суммируем все полученные величины при помощи оператора OCL *sum*, действующего на множество в целом.

- Какие клиенты пользовались кредитными картами для оплаты покупок в конкретном магазине?

```
aMerchantPurchase->select(aStartDate <= transactionDate and
transactionDate <= anEndDate).Statement.CreditCardAccount.
MailingAddress.Customer->asSet()
```

В этом выражении мы переходим от объекта *Merchant* к *Purchase*. Оператор *select* выбирает транзакции в рамках временного интервала, ограниченного *aStartDate* и *anEndDate*. Возможность перехода по обобщению от *Purchase* к *Transaction* неявно поддерживается OCL. Для каждой из этих транзакций мы переходим к объекту *Statement*, затем к *CreditCardAccount*, затем к *MailingAddress* и, наконец, к *Customer*. Ассоциация, связывающая *MailingAddress* с *Customer*, имеет тип многие-ко-многим, поэтому в результате перехода получается мультимножество. Оператор *asSet* преобразует мультимножество клиентов к обычному множеству (удаляет повторяющиеся элементы), что и дает нужный нам ответ.

- Сколько счетов для кредитных карт имеет конкретный клиент в данный момент?

```
aCustomer.MailingAddress.CreditCardAccount->size()
```

Для объекта класса *Customer* мы находим множество объектов *MailingAddress*. Затем по множеству адресов мы находим множество объектов *CreditCardAccount*. В результате этого перехода получается множество, а не мультимножество, потому что каждому счету соответствует ровно один адрес. К множеству объектов *CreditCardAccount* мы применяем оператор OCL *size*, возвращающий число элементов множества.

- Каков максимальный кредит для данного клиента по всем его счетам?

```
aCustomer.MailingAddress.CreditCardAccount.maximumCredit->sum()
```

Это выражение осуществляет переход от объекта *Customer* к объекту *MailingAddress*, затем к *CreditCardAccount*, в результате чего получается множество объектов *CreditCardAccount*. Для каждого из этих объектов определяется значение максимального кредита *maximumCredit*, после чего полученные значения суммируются при помощи оператора OCL *sum*.

Обратите внимание, что наши вопросы проверяют модель и позволяют обнаружить скрытые недостатки, которые затем можно будет устранить. Например, запрос количества счетов для кредитных карт подразумевает, что нам может понадобиться каким-то образом отличать действующие счета от недействующих.

Не забывайте о том, что OCL изначально разрабатывался как язык ограничений (см. главу 4). Однако, как вы уже видели, этот язык удобен и для прослеживания моделей.

3.6. Практические советы

В этом разделе мы даем вам несколько советов, касающихся построения моделей классов. Эти советы мы вывели из собственного опыта по разработке приложений. Многие идеи уже упоминались в этой главе.

- **Область задачи.** Не следует начинать моделирование классов с выписывания на листе списка классов и ассоциаций и построения иерархии наследования. Сначала необходимо понять задачу, требующую решения. Содержимое модели определяется решением задачи. Вам придется решать, какие объекты следует отразить в модели, а какие — нет. В модели отражаются только те аспекты, которые имеют непосредственное отношение к решению задачи (раздел 3.1.1).
- **Простота.** Старайтесь создавать простые модели. Простая модель оказывается более доступной для понимания и требует меньше усилий со стороны разработчиков. Постарайтесь использовать минимальное количество классов, которые должны быть четко определены. Избегайте избыточности. С подозрением относитесь к классам, которые сложно определить. Возможно, их лучше пересмотреть, а модель — реструктурировать.
- **Компоновка диаграммы.** Диаграмма должна быть построена таким образом, чтобы отражать симметрию решения. Чаще всего задача обладает сверхструктурой, которая не отражается самой системой обозначений. Постарайтесь располагать важные классы так, чтобы они выделялись на диаграмме. Старайтесь не допускать пересечения линий.
- **Имена.** Внимательно относитесь к выбору имен: они очень важны и несут в себе значительный скрытый смысл. Имена должны быть описательными, четкими и однозначными. В именах не должен преобладать какой-либо аспект объекта. Выбор правильных имен — одна из наиболее сложных задач,

стоящих перед разработчиком модели. В качестве имен классов следует использовать существительные в единственном числе.

- **Ссылки.** Не скрывайте ссылки на объекты в качестве атрибутов других объектов. Такие ссылки следует представлять ассоциациями. В этом случае вы будете отражать истинную суть задачи, а не подход, который будет применен при реализации модели (раздел 3.2.1).
- **Кратность.** Проверяйте полюса ассоциаций с кратностью «один». Часто наличие объекта на одном из полюсов оказывается необязательным и лучше будет указать кратность 0..1. В других случаях правильным оказывается значение «много» (раздел 3.2.2).
- **Имена полюсов ассоциаций.** Аккуратно относитесь к неоднократному использованию одного и того же класса. Имена полюсов ассоциаций позволяют сделать ссылки на один класс единообразными (раздел 3.2.3).
- **Мультимножества и последовательности.** Обычная бинарная ассоциация позволяет создать не более одной связи между парой объектов. Однако вы можете разрешить установку нескольких связей между двумя объектами, обозначив полюс ассоциации словом *{bag}* или *{sequence}* (раздел 3.2.5).
- **Атрибуты ассоциаций.** На этапе анализа не допускайте помещения атрибутов ассоциации в один из участвующих в ней классов. Модель должна непосредственно описывать объекты и связи между ними. На этапах проектирования и разработки вы всегда сможете разместить информацию там, где потребуется, чтобы сделать программу более эффективной (раздел 3.2.6).
- **Квалифицированные ассоциации.** Проверяйте полюса ассоциаций с кратностью «много». Квалификатор часто может повысить точность ассоциации и подчеркнуть важные направления навигации (раздел 3.2.7).
- **Уровни обобщения.** Избегайте глубокой вложенности иерархии обобщений (раздел 3.3.1).
- **Подмена составляющих.** Вы можете подменять методы и значения по умолчанию для атрибутов. Однако не допускайте нарушения сигнатуры или семантики исходной унаследованной составляющей при ее подмене (раздел 3.3.3).
- **Рецензии.** Попросите кого-нибудь дать рецензию на вашу модель. Будьте готовы к тому, что она потребует пересмотра. При работе с моделями классов часто приходится подбирать более четкие имена, улучшать абстракции, исправлять ошибки, добавлять сведения и добиваться более точной передачи ограничений. Почти все наши модели прошли несколько редакций.
- **Документация.** Всегда документируйте свои модели. На диаграмме отражается структура модели, но не могут быть отражены основания для ее построения. Объяснение, изложенное в письменном виде, направит читателя в нужную сторону и разъяснит ему тонкие моменты, определившие выбор конкретных конструкций.

3.7. Резюме по моделям классов

Модели классов описывают статические структуры данных, представленные объектами, и отношения между этими объектами. Содержимое модели определяется волей проектировщика и потребностями конкретного приложения. Объект — это концепция, абстракция или нечто, обладающее индивидуальностью, имеющее при этом смысл в рамках приложения. Класс описывает группу объектов с одинаковыми атрибутами, поведением, видами отношений и семантикой. Атрибут — это именованное свойство класса, описывающее значение, хранимое каждым объектом данного класса. Операция — это функция или процедура, которая может быть применена к объекту класса или вызвана им.

Связь — это физическое или концептуальное соединение между объектами. Связь является экземпляром ассоциации. Ассоциация — это описание группы связей с одинаковой структурой и семантикой. Ассоциация описывает множество потенциальных связей точно так же, как класс описывает множество возможных объектов. Ассоциация — это логическая конструкция, одной из возможных реализаций которой является ссылка. Существуют и другие способы реализации ассоциаций.

Полюс ассоциации может иметь имя и кратность. Кратность определяет количество экземпляров одного класса, которые могут быть связаны с одним экземпляром другого класса данной ассоциацией. Класс ассоциации — это ассоциация, которая одновременно является классом. Класс ассоциации может иметь атрибуты и операции и может участвовать в других ассоциациях. Квалифицированная ассоциация характеризуется тем, что объекты, находящиеся у полюса с кратностью «много», частично или полностью отделяются друг от друга при помощи атрибута, называемого квалификатором. Квалификатор позволяет выбрать один из целевых объектов, в результате чего эффективная кратность ассоциации снижается (чаще всего со значения «много» до значения «один»). Часто квалификаторами служат имена.

Обобщение — это отношение между классом (суперклассом) и его разновидностями (подклассами). Обобщение упорядочивает классы по подобию и различиям, благодаря чему структурируется описание объектов. Подкласс наследует атрибуты, операции и ассоциации суперкласса. Благодаря наследованию подкласс может повторно использовать составляющие суперкласса, а при необходимости — подменять их. Подкласс может также добавлять новые свойства.

Обобщение важно не только для концептуального моделирования, но и для реализации. На первом этапе обобщение позволяет разработчику упорядочить классы. На этапе реализации наследование способствует полиморфизму и повторному использованию кода. Наследование образует иерархию с произвольным числом уровней, в которой каждый уровень будет отражать один аспект объекта. Объект объединяет атрибуты, операции и ассоциации с каждого уровня иерархии обобщений.

Модели классов полезны не только для определения структур данных. Проследивание моделей классов позволяет вам выразить некоторые виды поведения. Кроме того, проследивание позволяет проверить модель классов и устранить

скрытые ошибки и недостатки, которые затем можно будет исправить. В состав UML входит специальный язык, который может использоваться для прослеживания моделей: язык объектных ограничений — OCL.

Различные конструкции модели классов позволяют точно описать сложную систему. Это демонстрирует наш пример с системой управления окнами рабочей станции. Как только у вас будет модель, пусть даже упрощенная, вы уже сможете проверять на ней требования к приложению, критиковать и улучшать ее.

Таблица 3.1. Ключевые понятия главы

ассоциация	потомок	метод	квалифицированная ассоциация
атрибут	обобщение	объект	сигнатура
диаграмма классов	наследование	перекрытие	суперкласс
класс	индивидуальность	операция	подкласс
класс ассоциации	направление	кратность	квалификатор
модель классов	значение по умолчанию	упорядочивание	значение
мультимножество	имя набора обобщений	диаграмма объектов	конкретизация
полюс ассоциации	составляющая	навигация	последовательность
предок	полиморфизм	связь	

Библиографические заметки

Описанный в этой книге подход к моделированию классов опирается на систему обозначений OMT, предложенную в работе [Loomis-87]. Сейчас вместо нее используется UML [Booch-99][Rumbaugh-05][UML]. Модель классов UML соответствует системе обозначений OMT, описанной в [Loomis-87]. Книга [Blaha-98] описывает систему обозначений UML для моделирования классов и уделяет особое внимание конструкциям, необходимым для работы с базами данных.

Система обозначений моделей классов является одним из множества подходов, выросших из модели сущностей и отношений (entity-relationship — ER) [Chen-76]. Эту модель пытались улучшить все последователи. Методика ER была успешно применена к моделированию баз данных и в результате на ее расширения возник большой спрос. Кроме того, моделирование ER относится только к базам данных, а не к программированию. Этих расширений было создано слишком много, чтобы мы могли обсуждать их в нашей книге.

Достоинством внимания аспектом системы обозначений OMT и последовавшей за ней системой UML является выделение роли ассоциаций. Ассоциации, как и наследование, важны для концептуального моделирования и реализации. Идея ассоциаций впервые прозвучала в [Rumbaugh-87]. Термин «отношение» в книге [Rumbaugh-87] совпадает по смыслу с термином ассоциация, используемым в этой книге.

В системах обозначений, разработанных для моделирования данных, таких как ER и IDEF1X, бинарная ассоциация может порождать не более одной связи для каждой пары объектов. UML1 следует этому соглашению и также ограничивает бинарную ассоциацию одной связью для одной пары объектов. UML2 является исключением: при помощи ключевых слов *{bag}* и *{sequence}* можно обозначить ассоциацию, которая допускает создание нескольких связей между одной парой объектов.

Концепция индивидуальности объекта и ее важность для языков программирования и систем баз данных были впервые рассмотрены в [Khoshafian-86].

В качестве справочника по языку OCL, входящему в состав UML, можно указать [Warmer-99]. В этой главе мы используем OCL для прослеживания моделей классов.

В книге [Rayside-00] объектно-ориентированные концепции сравниваются с философией. Автор подчеркивает важность кратких имен и ясности мышления.

В качестве дополнительной литературы по UML можно указать работы [Chonoles-03], [Fowler-00] и [Larman-02]. Мы благодарим Майкла Хонолза за пример на рис. 3.10 и 3.11, иллюстрирующий, что каждая ассоциация может порождать лишь одну связь между двумя объектами (за исключением случаев мультимножества и последовательности).

Ссылки

[Blahe-98] Michael Blaha and William Premerlani. Object-Oriented Modeling and Design for Database Applications. Upper Saddle River, NJ: Prentice Hall, 1998.

[Booch-99] Grady Booch, James Rumbaugh, and Ivar Jacobson. The Unified Modeling Language User Guide. Boston: Addison-Wesley, 1999.

[Chen-76] P.P.S. Chen. The Entity-Relationship model—toward a unified view of data. ACM Transactions on Database Systems 1, 1 (March 1976), 9-36.

[Chonoles-03] Michael Jesse Chonoles and James A. Schardt. UML2 for Dummies. New York: Wiley, 2003.

[Fowler-00] Martin Fowler. UML Distilled, Second Edition. Boston: Addison-Wesley, 2000.

[Khoshafian-86] S.N. Khoshafian and G.P. Copeland. Object identity. OOPSLA'86 as ACM SIGPLAN 21, 11 (November 1986), 406-416.

[Larman-02] Craig Larman. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. Upper Saddle River, NJ: Prentice Hall, 2002.

[Loomis-87] Mary E.S. Loomis, Ashwin V. Shah, and James E. Rumbaugh. An object modeling technique for conceptual design. European Conference on Object-Oriented Programming, Paris, France, June 15-17, 1987, published as Lecture Notes in Computer Science, 276, Springer-Verlag, 192-202.

[Rayside-00] Derek Rayside and Gerard Campbell. An Aristotelian understanding of object-oriented programming. OOPSLA'00 as ACM SIGPLAN 35, 10 (October 2000), 337-353.

[Rumbaugh-87] James E. Rumbaugh. Relations as semantic constructs in an object-oriented language. OOPSLA'87 as ACM SIGPLAN 22, 12 (December 1987), 466-481.

[Rumbaugh-05] James Rumbaugh, Ivar Jacobson, Grady Booch. The Unified Modeling Language Reference Manual, Second Edition. Boston: Addison-Wesley, 2005.
[UML] www.uml.org
[Warmer-99] Jos Warmer and Anneke Kleppe. The Object Constraint Language. Boston: Addison-Wesley, 1999.

Упражнения

3.1. (3) Превратите диаграмму объектов с рис. У3.1 в диаграмму классов.

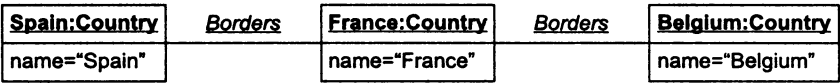


Рис. У3.1. Диаграмма объектов

3.2. (5) Превратите диаграмму объектов с рис. У3.2 в диаграмму классов. Объясните указанные вами значения кратности. Каждая точка характеризуется двумя координатами: x , y . Какое минимальное количество точек позволяет построить многоугольник? Имеет ли значение, может ли точка принадлежать сразу двум многоугольникам или нет? Ответ должен учитывать факт упорядоченности точек.

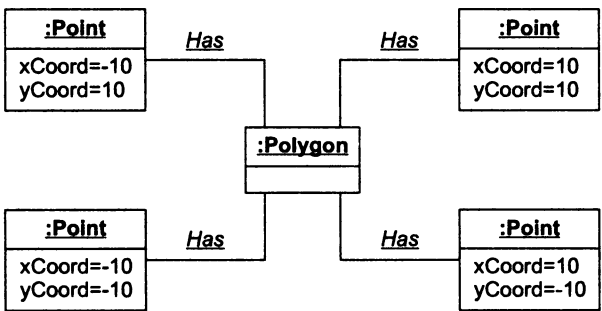


Рис. У3.2. Диаграмма объектов для многоугольника, являющегося квадратом

- 3.3. (5) По построенной вами диаграмме классов из упражнения 3.2 подготовьте диаграмму объектов для двух треугольников с одной общей стороной. Учитывайте одно из следующих условий:
- 1) Точка принадлежит ровно одному многоугольнику.
 - 2) Точка принадлежит одному или нескольким многоугольникам.
- 3.4. (5) Превратите диаграмму объектов с рис. У3.3 в диаграмму классов. Каким образом ваша диаграмма отражает факт упорядоченности точек. Предположите, что точка может принадлежать не более чем одному многоугольнику.
- 3.5. (2) Подготовьте письменное описание диаграмм классов из упражнений 3.2 и 3.4.

3.6. (6) Подготовьте диаграмму классов по диаграмме объектов с рис. УЗ.4.

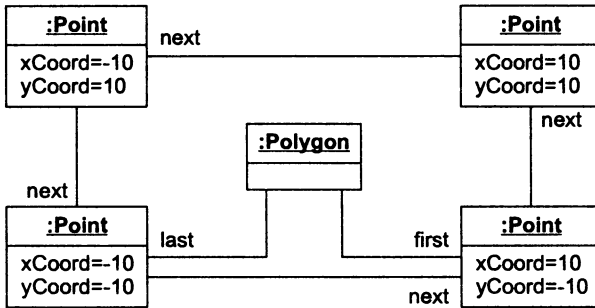


Рис. УЗ.3. Диаграмма объектов для многоугольника, являющегося квадратом

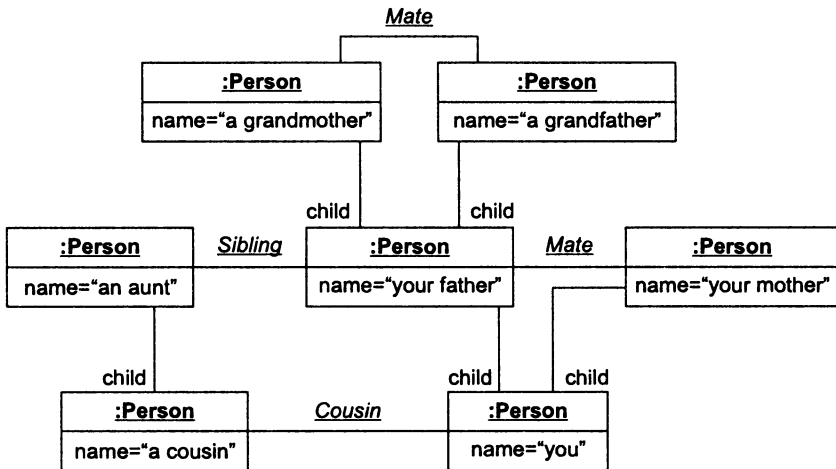


Рис. УЗ.4. Диаграмма объектов, на которой изображена часть вашего генеалогического древа

- 3.7. (5) Подготовьте диаграмму классов по диаграмме объектов с рис. УЗ.5. В данном документе имеется 4 страницы. На первой странице отображается красная точка и желтый квадрат. На второй странице нарисована линия и эллипс. На последних двух страницах изображены дуга, окружность и прямоугольник. При построении диаграммы следует использовать по крайней мере одно обобщение.
- 3.8. (4) На рис. УЗ.6 изображена неполная диаграмма классов системы подачи воздуха. Кратность не указана. Укажите кратность. Объясните, каким образом значения кратности могут зависеть от вашего восприятия мира.
- 3.9. (3) Укажите недостающие имена ассоциаций на рис. УЗ.6.
- 3.10. (3) Укажите имена полюсов ассоциаций на рис. УЗ.6. Имена должны иметь смысл и отличаться от имен классов. Вы должны добавить по меньшей мере шесть имен.

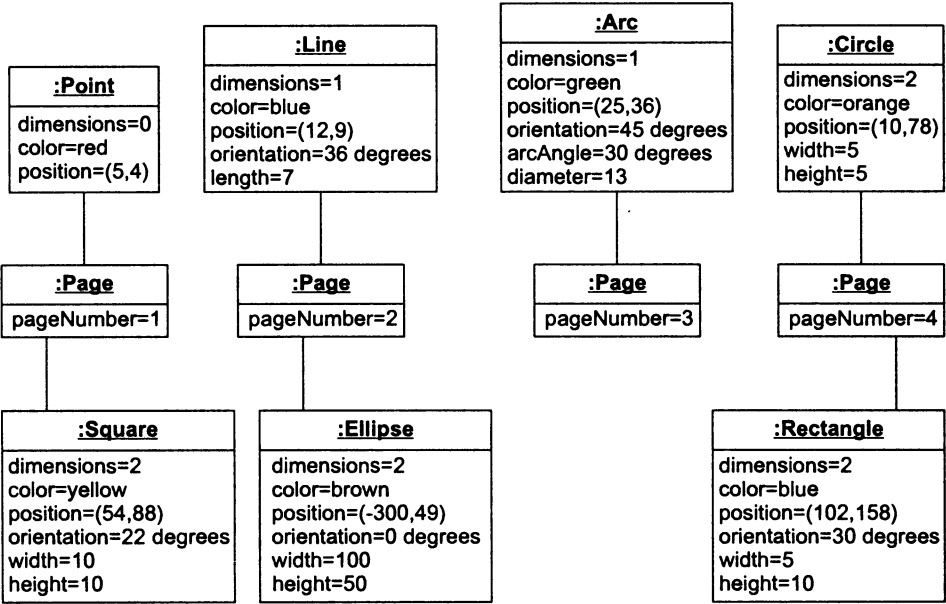


Рис. УЗ.5. Диаграмма объектов для документа с геометрическими фигурами

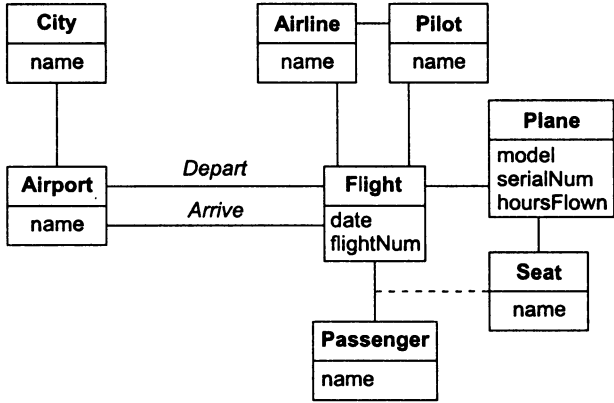


Рис. УЗ.6. Неполная диаграмма классов системы подачи воздуха

- 3.11. (2) Добавьте на диаграмму классов рис. УЗ.6 следующие операции: *heat* (нагреть), *hire* (нанять), *fire* (уволить), *refuel* (заправить), *reserve* (зарезервировать), *clean* (очистить), *de-ice* (убрать лед), *take off* (взлететь), *land* (приземлиться), *repair* (чинить), *cancel* (отменять), *delay* (откладывать). Операция может быть добавлена в несколько классов.
- 3.12. (6) Подготовьте диаграмму объектов для вашей воображаемой поездки в Лондон на прошлых выходных. На этой диаграмме должен присутствовать по крайней мере один экземпляр каждого класса. К счастью, вам удалось взять билет на прямой рейс. С вами был друг, но он решил остаться и до сих пор

живет в Лондоне. Пилотом на обоих рейсах был капитан Джонсон. У вас были разные места, но вы заметили, что самолет был тот же самый (у него была вмятина в хвостовой части). Незвестные значения укажите знаком «?».

- 3.13. Подготовьте диаграмму классов для каждой группы классов. Добавьте на каждую диаграмму не менее 10 отношений (ассоциаций и обобщений). При необходимости используйте имена ассоциаций и полюсов. Используйте также квалифицированные ассоциации и указывайте кратность. Показывать атрибуты и операции не нужно. При подготовке диаграмм вы можете добавлять дополнительные классы. Не забудьте прокомментировать диаграммы.
- 1) (6) школа, детская площадка, директор, школьный совет, класс, книга, студент, учитель, кафетерий, комната отдыха, компьютер, парта, стул, линейка, дверь, качели
 - 2) (4) автомобиль, двигатель, колесо, тормоз, тормозной огонь, дверь, аккумулятор, глушитель, выводящая труба глушителя
 - 3) (4) замок, ров, подъемный мост, башня, призрак, лестница, подземелье, этаж, коридор, комната, окно, камень, лорд, леди, кухарка
 - 4) (8) выражение, константа, переменная, функция, список аргументов, оператор отношения, член, множитель, арифметический оператор, оператор, компьютерная программа
 - 5) (6) файловая система, файл, ASCII-файл, двоичный файл, файл-каталог, диск, привод, дорожка, сектор
 - 6) (4) газовая печь, вентилятор, двигатель вентилятора, комнатный терморегулятор, терморегулятор печи, увлажнитель, датчик влажности, регулятор подачи газа, регулятор вентилятора, клапан для теплого воздуха
 - 7) (7) шахматная фигура, горизонталь, вертикаль, поле, доска, ход, дерево ходов
 - 8) (4) сточная труба, холодильник, морозильник, стол, лампа, выключатель, окно, дымовая сигнализация, охранная сигнализация, шкаф, хлеб, сыр, лед, дверь, кухня
- 3.14. (4) Добавьте по крайней мере 10 атрибутов и по крайней мере 5 методов на каждую из диаграмм классов, созданных в процессе выполнения предыдущего упражнения.
- 3.15. (6) На рис. У3.7 изображена часть диаграммы классов компьютерной программы, предназначенной для игры в несколько карточных игр. Колода (*deck*), карты на руках (*hand*), прикуп (*draw pile*) и снос (*discard pile*) — множества карт. Исходное количество карт на руках зависит от типа игры. Каждая карта характеризуется мастью (*suit*) и старшинством (*rank*). Добавьте на диаграмму следующие операции: *display* (открыть), *shuffle* (перетасовать), *deal* (сдать), *initialize* (инициализировать), *sort* (сортировать), *topOfPile* (верх колоды), *bottomOfPile* (низ колоды), *insert* (вставить), *draw* (брать)

и *discard* (сносить). Некоторые операции могут входить не в один класс, а в несколько. Для каждого класса, в котором есть операции, укажите их аргументы и поведение для экземпляра класса.

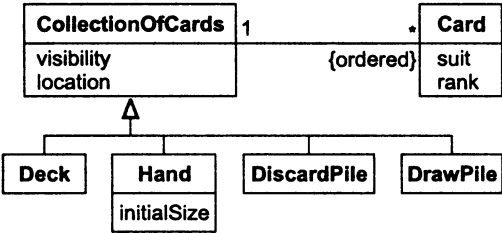


Рис. УЗ.7. Часть диаграммы классов программы для игры в карты

3.16. (5) На рис. УЗ.8 приведена часть диаграммы классов компьютерной системы верстки газет. Система предназначена для работы с газетными страницами, на которых могут располагаться, помимо всего прочего, колонки текста. Пользователь может изменять ширину и длину колонки, перемещать ее по странице или переносить с одной страницы на другую. В нашем примере одна колонка может располагаться только на одной странице.

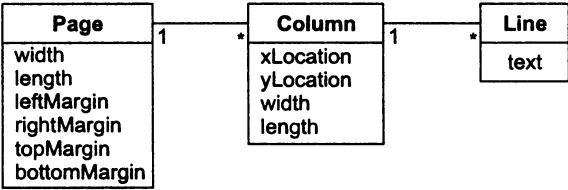


Рис. УЗ.8. Часть диаграммы классов системы компьютерной верстки

Измените диаграмму классов таким образом, чтобы одна колонка могла размещаться одновременно на нескольких страницах. Если пользователь изменяет текст на одной странице, изменения должны автоматически проявляться и на других страницах. Координаты *x* и *y* должны стать атрибутами ассоциации.

3.17. (6) На рис. УЗ.9 изображена диаграмма классов, которая может быть использована для разработки системы, упрощающей планирование расписания и подсчет очков для соревнований, в которых выступление спортсменов оценивается судьями (например, гимнастика, прыжки в воду и фигурное катание). Соревнования проводятся по нескольким дисциплинам, и каждый спортсмен может выступать в нескольких из них.

По каждой дисциплине судейство осуществляется несколькими судьями, которые субъективно оценивают выступления спортсменов по данной конкретной дисциплине. Судья оценивает выступление каждого спортсмена. В некоторых случаях судья может осуществлять судейство по нескольким дисциплинам.

Спортсмены имеют несколько попыток. Попытка оценивается судейским комитетом по данной дисциплине. Затем подсчитывается суммарный результат спортсмена по всем попыткам.

Вам необходимо указать на диаграмме классов значения кратности.

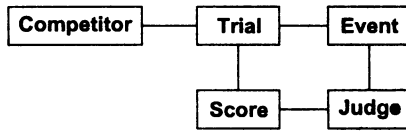


Рис. УЗ.9. Часть диаграммы классов системы учета очков

- 3.18. (3) Добавьте на рис. УЗ.9 следующие атрибуты: адрес, возраст, дата, коэффициент сложности, баллы, имя. Некоторые атрибуты могут быть добавлены в несколько классов.
- 3.19. (3) Добавьте на рис. УЗ.9 ассоциацию, которая сделала бы возможным определение списка дисциплин, в которых планирует принимать участие конкретный спортсмен, до того, как он в них выступит.
- 3.20. (6) Создайте модель классов для описания неориентированных графов. Неориентированный граф состоит из множества вершин и множества дуг. Дуга соединяет между собой пару вершин. Модель должна описывать только структуру графа (соединения между вершинами) и не должна включать сведения о размещении вершин и дуг. Пример неориентированного графа показан на рис. УЗ.10.

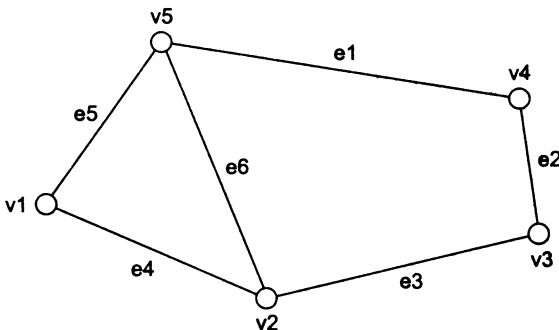


Рис. УЗ.10. Неориентированный граф

- 3.21. (4) Нарисуйте диаграмму объектов для рис. УЗ.10. (Замечание для преподавателя: может быть, студентам придется дать наш ответ для упражнения 3.20.)
- 3.22. (5) Расширьте диаграмму классов, получившуюся в упражнении 3.20, добавив в нее сведения о расположении вершин графа и толщине и цвете дуг. Добавьте также названия вершин и дуг. (Замечание для преподавателя: может быть, студентам придется дать наш ответ для упражнения 3.20.)

3.23. (7) Создайте модель классов для описания ориентированных графов. Ориентированный граф отличается от неориентированного тем, что его дуги имеют направления. На рис. УЗ.11 показан пример ориентированного графа.

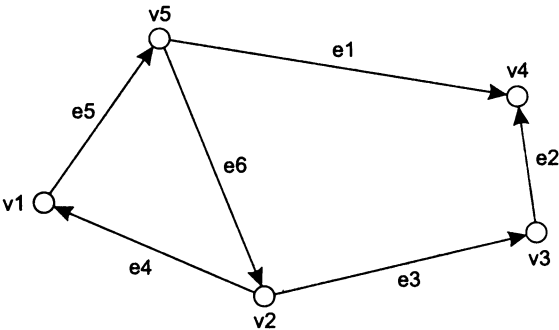


Рис. УЗ.11. Ориентированный граф

3.24. (4) Нарисуйте диаграмму объектов для рис. УЗ.11. (Замечание для преподавателя: может быть, студентам придется дать наш ответ для упражнения 3.23.)

3.25. (7) Атрибуты некоторых классов на рис. УЗ.12 на самом деле представляют собой ссылки на другие классы. Их следовало бы заменить ассоциациями. Работодателями человека могут выступать до трех компаний. У каждого человека есть свой идентификатор. Идентификатор также присваивается автомобилю. Автомобили могут принадлежать людям, компаниям и банкам. Идентификатор владельца ссылается на идентификатор человека, компании или банка, которому принадлежит машина. Для покупки машины можно взять ссуду.

Упаковывать ссылки на объекты внутри других объектов — это неправильный подход к конструированию модели. Создайте диаграмму классов без идентификаторов, в которой использовались бы ассоциации и обобщения. Постарайтесь указать на ней возможные значения кратности. Возможно, вам придется добавить несколько своих собственных классов.

Person	Car	CarLoan	Company	Bank
name birthdate employer1ID employer2ID employer3ID personID address	ownerID vehicleID ownerType model year	vehicleID customerType customerID accountNumber bankID interestRate currentBalance	name companyID	name bankID

Рис. УЗ.12. Классы с атрибутами-ссылками

3.26. (4) Когда один и тот же объект должен идентифицироваться несколькими независимыми системами, это создает определенные проблемы. Предположим, что министерство автомобильной промышленности, страховая

компания, банк и полиция хотят идентифицировать конкретный автомобиль. Обсудите достоинства и недостатки следующих методов идентификации:

- 1) идентификация по владельцу;
- 2) идентификация по атрибутам (производитель, модель, год);
- 3) идентификация по серийному номеру, который присваивается автомобилю производителем;
- 4) идентификация по внутренним номерам, используемым в каждой из независимых систем.

3.27. (7) Разработайте модель классов, которая могла бы быть использована для устранения неполадок в 4-тактном двигателе газонокосилки. Представьте эту модель на трех диаграммах, по одной для каждого из следующих трех абзацев.

В четырехтактном двигателе движущей силой является сгорание смеси воздуха с бензином, находящейся под давлением поршня. Поршень прикреплен к коленчатому валу шатуном. При вращении коленчатого вала поршень перемещается внутри цилиндра вверх и вниз. Когда поршень опускается, открывается впускной клапан и поршень затягивает рабочую смесь в цилиндр. В самой нижней точке хода поршня впускной клапан закрывается. Двигаясь вверх, поршень сжимает и нагревает смесь. Манжета прижимается к стенкам цилиндра, обеспечивая уплотнение, необходимое для сжатия смеси, и смазывая стенки цилиндра маслом. В верхней точке свеча зажигания дает электрическую искру, которая детонирует смесь. Расширяющиеся газы создают движущую силу, толкая поршень вниз. В нижней точке открывается выпускной клапан. При следующем подъеме поршня выхлопные газы выталкиваются из цилиндра.

Топливо смешивается с воздухом в карбюраторе. Пыль и грязь, которые могли бы вызвать преждевременный износ двигателя, задерживаются воздушным фильтром. Оптимальное соотношение топлива и воздуха регулируется винтами. Дроссельная заслонка управляет количеством смеси, подаваемым в цилиндр. Дроссельная заслонка соединена с педалью газа в салоне машины, а также с регулятором хода, который представляет собой механическое устройство, стабилизирующее скорость вращения двигателя при меняющейся механической нагрузке. Впускной и выхлопной клапаны удерживаются в закрытом состоянии пружинами, а в нужный момент открываются распределительным (кулачковым) валом, который соединен с коленчатым валом через систему шестерен.

Электрическая искра создается магнитом, катушкой, конденсатором и нормально-закрытым реле, которое называется прерыватель. Первичная обмотка низкого напряжения соединена с прерывателем. Вторичная обмотка с высоким напряжением соединена со свечой зажигания. Магнит устанавливается на маховике. Вращаясь около катушки, он наводит ток в замкнутой первичной обмотке. Прерыватель открывается в нужный момент под

действием кулачка, укрепленного на коленчатом вале. Ток в первичной обмотке перестает течь, из-за чего во вторичной обмотке возникает высоковольтный импульс.

- 3.28. (6) Подготовьте диаграмму классов для задачи об обедающих философах. Условие задачи таково: пять философов сидят за круглым столом, на котором лежат пять вилок. Каждый философ может дотянуться до двух вилок (по одной с каждой стороны). Каждая вилка может быть взята одним из двух философов. Вилка может либо лежать на столе, либо находиться в руке у философа. Философ может есть только двумя вилками.
- 3.29. (7) Ханойские башни — задача, широко используемая для обучения программированию с использованием рекурсии. Задача состоит в том, чтобы перенести все диски с одного из трех стержней на другой, используя промежуточный третий стержень. Все диски разного размера. Со стержня можно снять верхний диск и надеть его на другой стержень при условии, что диск большего размера не будет лежать на диске меньшего размера. Алгоритм определения последовательности ходов зависит от структуры диаграммы классов, которая кладется в основу программы. Подготовьте диаграммы классов по каждому из приведенных ниже описаний. На диаграммах следует изобразить классы и ассоциации. Атрибуты и операции можно не обозначать.
- 1) Башня состоит из трех стержней. На каждом стержне надето несколько дисков в определенном порядке.
 - 2) Башня состоит из трех стержней. Диски на стержнях объединены в подмножества, называемые стопками. Стопка — это упорядоченное множество дисков. Каждый диск принадлежит ровно одной стопке. На одном стержне может быть несколько стопок, расположенных в определенном порядке.
 - 3) Башня состоит из трех стержней. Диски на стержнях объединены в подмножества, называемые стопками, как в п. 2. На одном стержне может находиться несколько стопок. Стопка обладает рекурсивной структурой. Стопка состоит из одного диска, находящегося в самом ее низу, и не более чем одной вложенной стопки (в зависимости от высоты данной стопки).
 - 4) Аналогично п. 3, но с тем отличием, что на стержне находится ровно одна стопка, а все остальные являются вложенными по отношению к ней.
- 3.30. (8) Рекурсивный алгоритм для получения последовательности ходов решения задачи, описанной в предыдущем упражнении, работает со стопками дисков. Чтобы переместить на другой стержень стопку высоты N , где $N > 1$, сначала нужно переместить стопку высоты $N-1$ на свободный стержень при помощи рекурсивного вызова. Затем нижний диск перекладывается на нужный стержень. Затем стопка переносится с временного стержня на конечный. Рекурсия заведомо завершается, поскольку перемещение стопки высоты 1 является тривиальной операцией. Какая из подготовленных в предыдущем упражнении диаграмм классов лучше всего подходит

для реализации данного алгоритма? Объясните, почему. Добавьте на диаграмму атрибуты и операции. Укажите аргументы операций. Опишите, что каждая из операций должна делать с тем классом, в котором она определена.

- 3.31. (6) Напишите выражение OCL для вычисления множества названий авиакомпаний, услугами которых воспользовался клиент в некотором году (рис. У3.6). Предположите, что имеется функция *getYear(date)*, определяющая год по заданной дате. (Замечание для преподавателя: наш ответ к упражнению 3.10 следует дать студентам в качестве отправной точки для решения этого задания.)
- 3.32. (6) Напишите выражение OCL для отбора беспосадочных рейсов из города *aCity1* в город *aCity2* (рис. У3.6). (Замечание для преподавателя: наш ответ к упражнению 3.10 следует дать студентам в качестве отправной точки для решения этого задания.)
- 3.33. (6) Напишите выражение OCL для определения суммарного результата участника по оценкам одного конкретного судьи. (Замечание для преподавателя: наш ответ к упражнению 3.18 следует дать студентам в качестве отправной точки для решения этого задания.)
- 3.34. (6) Сравните модели классов, изображенные на рис. У3.13. В левой модели *Subscription* представляет собой класс ассоциации, в правой — обычный класс. Человек может быть подписан на несколько журналов. На один журнал может быть подписано множество человек. Для каждой подписки необходимо отслеживать дату и размер каждого платежа, а также текущий срок окончания подписки.

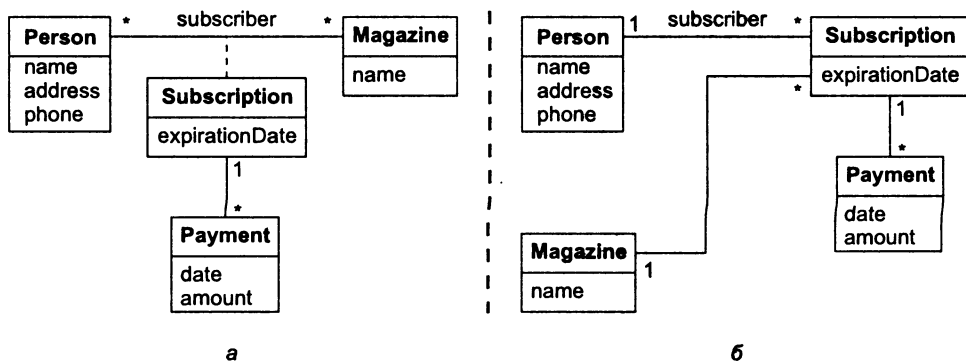


Рис. У3.13. Диаграммы классов для подписки на журналы

Углубленное моделирование классов



Глава 4 продолжает обсуждение концепций моделирования классов и переходит к более сложным темам. Здесь рассматриваются тонкости, требующиеся для создания сложных моделей. При первом чтении эту главу можно пропустить.

4.1. Расширенные концепции классов и объектов

4.1.1. Перечисление

Тип данных — это описание возможных значений. К типам данных относятся числа, строки и перечисления. Перечисление (enumeration) — это тип данных, имеющий конечное множество значений. Например, атрибут *accessPermission* (разрешениеДоступа) на рис. 3.17 представляет собой перечисление с возможными значениями *read* (чтение) и *read-write* (чтение и запись). На рис. 3.25 также присутствуют перечисления, которые вынесены на рис. 4.1. *Figure.penType* (Фигура.типКисти) — перечисление, включающее возможные значения *solid* (сплошная), *dashed* (штриховая) и *dotted* (пунктирная). *TwoDimensional.fillType* (Двумерная.типЗаливки) — перечисление, включающее возможные значения *solid* (сплошная), *grey* (полутон), *none* (нет), *horizontal lines* (горизонтальные линии) и *vertical lines* (вертикальные линии).



Рис. 4.1. Примеры перечислений

При конструировании моделей перечисления следует обязательно выделять, потому что они встречаются достаточно часто и важны для пользователей. Пере-

числения важны и для реализации. Все возможные значения перечисления можно указать в списке для выбора. Необходимо также ограничить возможные данные разрешенными значениями.

Не следует использовать обобщение для указания значений перечислимого атрибута. Перечисление — это просто список значений, а обобщение предназначено для упорядочивания описания объектов. Обобщение следует использовать только там, где по крайней мере один из подклассов обладает атрибутами, операциями или ассоциациями, неприменимыми к суперклассу. Как показано на рис. 4.2, не следует строить обобщение класса *Card*, потому что в большинстве игр карты разных мастей ведут себя одинаково.

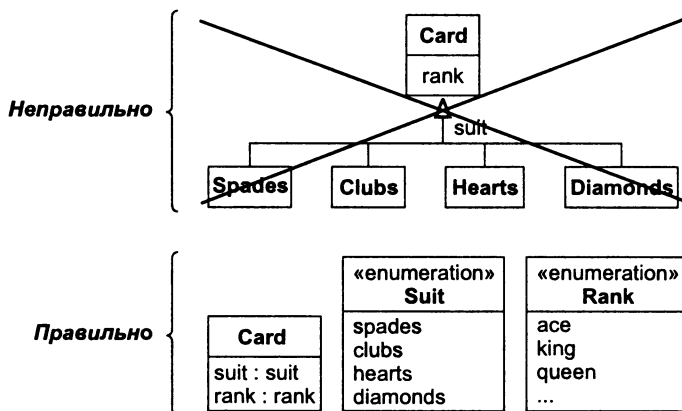


Рис. 4.2. Не используйте обобщение для моделирования перечислений

В языке UML перечисление является типом данных. Для объявления перечисления необходимо указать ключевое слово *enumeration* в угловых кавычках («») над именем перечисления в верхнем разделе прямоугольника. Во втором разделе перечисляются значения перечисления (рис. 4.2).

4.1.2. Кратность

Кратность — это ограничение на количество возможных значений в наборе. В главе 3 было описано применение кратности к ассоциациям. Кратность применима и к атрибутам.

Указание кратности для атрибутов часто оказывается полезным, особенно в приложениях, связанных с базами данных. Кратность атрибута определяет количество возможных значений в каждом экземпляре атрибута. Наиболее типичные значения кратности: [1] — обязательное единственное значение, [0..1] — необязательное единственное значение, [*] — произвольное количество значений. Кратность определяет, является ли атрибут обязательным (в терминах баз данных — может ли атрибут иметь пустое значение). Она также указывает, является ли атрибут единственным значением, или он может представлять собой совокупность значений. Если кратность не указана, предполагается, что она равна [1]

(обязательное единственное значение). На рис. 4.3 человек должен иметь одно имя, один или несколько адресов, одну дату рождения и *может* иметь один или несколько телефонных номеров.

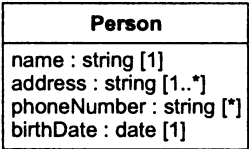


Рис. 4.3. Задание кратности для атрибутов

4.1.3. Область действия

В главе 3 описывались составляющие индивидуальных объектов. Действительно, обычно составляющими обладают именно объекты, но в некоторых случаях приходится определять составляющие класса как целого. *Область действия* (scope) определяет, к чему относится данная составляющая: к объекту или к классу. Если областью действия составляющей является класс, ее имя подчеркивается. Такая составляющая называется статической. По нашему соглашению атрибуты и операции, областью действия которых является класс, указываются в верхней части соответствующих разделов (первыми).

Атрибут, областью действия которого является класс, может хранить так называемый экстенсн класс (множество всех его объектов). Это довольно часто применяется в базах данных. Других статических атрибутов следует избегать, поскольку их использование ухудшает качество модели. Лучше всего моделировать группы явным образом и приписывать им нужные атрибуты. Например, в верхней части рис. 4.4 изображена простая модель голосовой почты. Каждое сообщение характеризуется ящиком владельца, датой записи, временем записи, приоритетом, содержанием сообщения и флагом, указывающим, было ли оно уже доставлено адресату. Источником сообщения может быть почтовый ящик или звонок извне. Каждый почтовый ящик характеризуется номером телефона, паролем и записанным приветствием. В классе *PhoneMessage* мы можем хранить максимальную длительность сообщения и максимальный срок его хранения. В классе *PhoneMailbox* можно хранить ограничение на количество сообщений.

Эта модель обладает определенными недостатками, поскольку все три ограничения задаются для системы в целом. В модели, изображенной на нижней части рисунка, эти ограничения могут быть определены по-разному для разных типов пользователей, что в результате дает более гибкую и расширяемую систему голосовой почты.

Статические операции, в отличие от атрибутов, вполне допустимы. Чаще всего операции, областью действия которых являются классы, применяются для создания новых экземпляров класса. Иногда оказывается удобным определять статические операции для получения суммарных данных. Однако следует с осторожностью относиться к статическим операциям в распределенных приложениях.

4.1.4. Видимость

Видимость (visibility) характеризует способность метода ссылаться на составляющую другого класса и может принимать значения *public* (открытая), *protected* (защищенная), *private* (закрытая) и *package* (пакетная). Конкретная интерпретация этих значений зависит от языка программирования (подробнее см. главу 18). Любой метод может свободно обращаться к открытым составляющим (public). Методы класса-владельца составляющей и всех его потомков могут обращаться к защищенным составляющим (protected). (Защищенные составляющие в Java одновременно являются доступными в рамках пакета.) К закрытым составляющим (private) могут обращаться только методы класса-владельца. К составляющим, доступным в рамках пакета (package), могут обращаться все методы того же пакета.

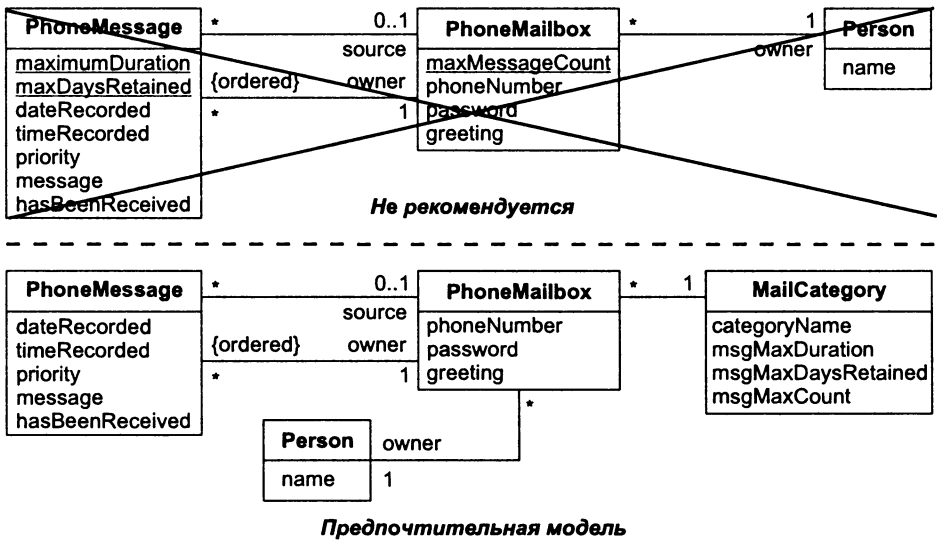


Рис. 4.4. Область действия атрибута

В языке UML видимость обозначается при помощи префикса. Символ + ставится перед открытыми составляющими. Символ # ставится перед защищенными составляющими. Символ – ставится перед закрытыми составляющими. Символ ~ ставится перед составляющими, доступными в рамках пакета. Отсутствие префикса не позволяет делать никаких заключений о видимости.

При выборе значения видимости для какой-либо составляющей следует учитывать следующие соображения.

- **Понятность модели.** Чтобы получить представление о возможностях класса, необходимо понять все его открытые составляющие. Закрытые, защищенные и пакетные составляющие можно игнорировать: они вводятся лишь для удобства реализации.
- **Расширяемость.** Открытые методы класса могут использоваться множеством других классов, поэтому изменение их сигнатуры (количества и типа

аргументов, типа возвращаемого значения) может разрушить согласованность всей модели. Поскольку защищенные, закрытые и пакетные методы заведомо используются меньшим числом классов, их изменения обходятся дешевле.

- **Контекст.** Закрытые, защищенные и пакетные методы могут зависеть от предусловий или информации о состоянии, произведенной другими методами того же класса. Вне своего контекста закрытые методы могут возвращать неверные результаты или привести к отказу объекта в целом.

4.2. Полюса ассоциаций

Полюсом ассоциации называется ее конец. Бинарная ассоциация имеет два полюса, тернарная (см. раздел 4.3) — три и т. д. В главе 3 мы обсудили следующие свойства полюса ассоциации.

- **Имя полюса.** Полюс ассоциации может иметь имя. Имя позволяет различать разные ссылки на один и тот же класс и упрощает навигацию. Часто удается подобрать осмысленные имена полюсов ассоциаций, которые оказываются полезными в своем контексте.
- **Кратность.** Для каждого полюса ассоциации можно указать кратность. Чаще всего используются следующие значения кратности: «1» (ровно один), «0..1» (не более одного) и «*» («много», то есть нуль и более).
- **Упорядочение.** Объекты у полюса ассоциации с кратностью «много» обычно образуют обычное множество. Однако иногда объекты этого множества могут быть упорядочены.
- **Мультимножества и последовательности.** Объекты у полюса ассоциации с кратностью «много» могут образовывать мультимножество или последовательность.
- **Квалификаторы.** Квалификатор (атрибут полюса ассоциации) позволяет различать объекты, находящиеся у полюса ассоциации с кратностью «много».
- **Агрегация.** Полюс ассоциации может быть агрегацией или составляющей частью (см. раздел 4.4). Агрегацией может быть только бинарная ассоциация. Один из полюсов ассоциации должен быть агрегацией, а другой — составляющей частью.
- **Изменяемость.** Это свойство характеризует возможность обновления полюса ассоциации. Возможные значения: *changeable* (может быть изменен) и *readonly* (может быть только проинициализирован).
- **Возможность навигации.** С концептуальной точки зрения ассоциация может прослеживаться в любом направлении. Однако реализация может фактически поддерживать навигацию только в одном направлении. В UML возможность навигации в одном направлении обозначается стрелкой у полюса ассоциации, прикрепленной к целевому классу. Стрелки можно поставить у одного или двух полюсов ассоциации или их можно не ставить вообще.

- **Видимость.** Полюса ассоциации могут иметь видимость, точно так же, как атрибуты и операции (см. раздел 4.1.4). Возможные значения: *public* (открытая), *protected* (защищенная), *private* (закрытая), *package* (пакетная).

4.3. N-арные ассоциации

В главе 3 мы рассказывали только о бинарных ассоциациях. Однако иногда вам могут встретиться и *n-арные ассоциации* (ассоциации между тремя и более классами). Мы рекомендуем вам избегать использования n-арных ассоциаций в моделях: большинство из них могут быть разбиты на бинарные ассоциации с квалификаторами и атрибутами. На рис. 4.5 показана ассоциация (*Человек совершает покупку акции на бирже*), которая с первого взгляда может показаться n-арной, но на самом деле может быть переформулирована в виде двух бинарных ассоциаций.

Не атомарная n-арная ассоциация «A person... in a company» ...

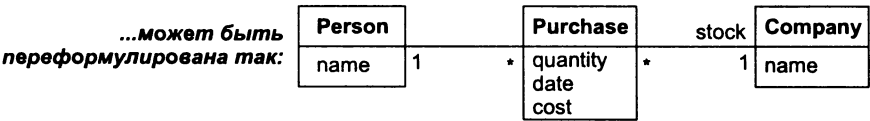


Рис. 4.5. Разбиение тернарной ассоциации на две бинарные

На рис. 4.6 показана настоящая тернарная ассоциация, не сводимая к бинарным: программисты используют в проектах языки программирования. Эта ассоциация является атомарной и не может быть разделена на бинарные без потери информации. Программист может знать язык и участвовать в проекте, но не использовать этот язык в данном проекте.

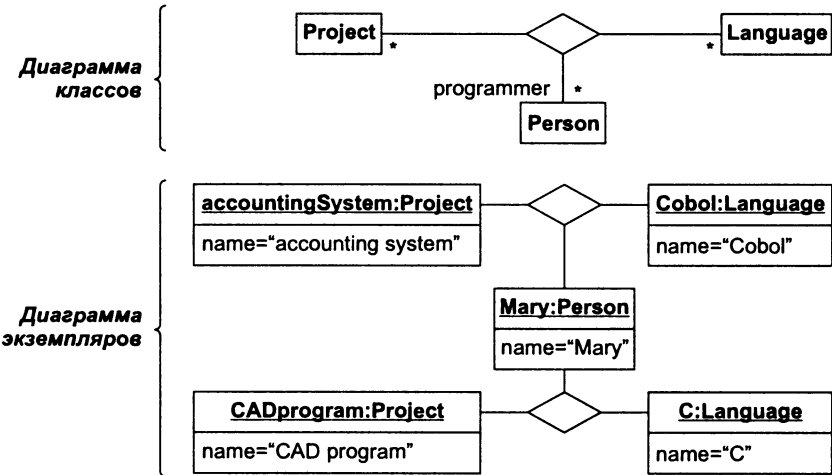


Рис. 4.6. n-арные ассоциации

В UML для обозначения n-арных ассоциаций используется значок ромба, соединяемый с классами при помощи линий. Имя ассоциации может быть указано около ромба и выделено курсивом.

Полюса n-арной ассоциации могут иметь имена, точно так же, как и полюса бинарной ассоциации (рис. 4.6). Указание имен полюсов является обязательным, если класс участвует в n-арной ассоциации несколько раз. Проследить n-арную ассоциацию от одного полюса к другому невозможно, в отличие от бинарной ассоциации, поэтому имена полюсов не являются псевдоатрибутами участвующих в ассоциации классов. Язык OCL [Warmer-99] не имеет обозначений для прослеживания n-арных ассоциаций.

На рис. 4.7 показана еще одна тернарная ассоциация: профессор читает курс лекций в течение семестра. Курс лекций характеризуется номером комнаты и произвольным количеством учебников.

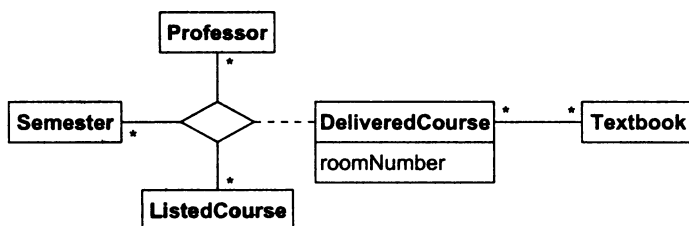


Рис. 4.7. n-арные ассоциации тоже могут быть классами

В обычных языках программирования нет средств для описания n-арных ассоциаций. Поэтому если вы собираетесь программировать свою модель, вам придется сделать n-арные ассоциации классами (*DeliveredClass* на рис. 4.8). Помните, что делая n-арную ассоциацию классом, вы изменяете смысл модели. N-арная ассоциация требует, что для каждой комбинации классов должно существовать не более одной связи. Например, для каждой комбинации *Professor*, *Semester* и *ListedCourse* на рис. 4.7 существует один класс *DeliveredCourse*. Ассоциация, превращенная в класс, допускает любое количество связей: для каждой комбинации *Professor*, *Semester* и *ListedCourse* на рис. 4.8 классов *DeliveredCourse* может быть несколько. При реализации рис. 4.8 пришлось бы писать отдельный код для обеспечения уникальности *DeliveredCourse*.

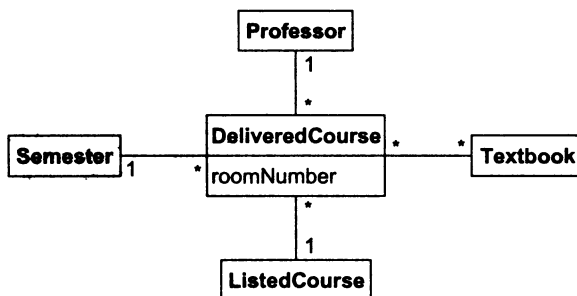


Рис. 4.8. Превращение n-арной ассоциации в класс

4.4. Агрегация

Агрегация (aggregation) — это частный случай ассоциации, описывающий объекты, состоящие из частей. Составляющие являются частями агрегации. Агрегат с семантической точки зрения представляет собой расширенный объект, обрабатываемый многими операциями как единое целое, хотя физически он состоит из нескольких объектов.

Мы определяем агрегацию как отношение между классом-агрегатом и одним составляющим классом. Агрегат из составляющих частей разных типов должен моделироваться как несколько агрегаций. Например, *газонокосилка состоит из ножа, двигателя, нескольких колес и корпуса*. Здесь газонокосилка является агрегатом, а остальные детали — составляющими частями. Газонокосилка и нож образуют первую агрегацию, газонокосилка и двигатель — вторую и т. д. Каждая пара классов определяется как отдельная агрегация, чтобы мы могли указывать кратность каждой составляющей части в агрегате. Это определение подчеркивает, что агрегация является частным случаем бинарной ассоциации.

Наиболее важным свойством агрегации является транзитивность: если A является частью B , а B является частью C , то A является частью C . Кроме того, агрегация антисимметрична: если A является частью B , то B не является частью A . Многие операции с агрегациями подразумевают транзитивное замыкание¹ и действуют как на непосредственные, так и на косвенные (вложенные) части.

4.4.1. Агрегация и ассоциация

Агрегация — это частный случай ассоциации. Агрегация характеризуется дополнительным семантическим подтекстом. Если два объекта связаны отношением часть-целое, их следует моделировать при помощи агрегации. Если же объекты обычно рассматриваются как независимые, хотя между ними и может возникать связь, это будет ассоциация. Свой выбор вы можете проверить при помощи следующих контрольных вопросов:

- Можно ли связать объекты словом «часть»?
- Применимы ли операции над целым к его частям?
- Распространяются ли некоторые атрибуты целого на его части?
- Обладает ли ассоциация внутренней антисимметричностью (один класс подчинен другому)?

К агрегациям относятся спецификации материалов, детальные чертежи, а также разложения на составляющие. Агрегация изображается как ассоциация с небольшим ромбом, который ставится около полюса, являющегося агрегатом. На рис. 4.9 показана газонокосилка, состоящая из одного ножа, одного двигателя, нескольких

¹ Транзитивное замыкание (transitive closure) — это термин из теории графов. Если E — дуга, а N — вершина и S — множество всех пар вершин, соединенных ребром, то S^+ (транзитивное замыкание S) — множество всех пар вершин, соединенных последовательностью дуг непосредственно или косвенно. Таким образом, в S^+ входят все вершины, соединенные непосредственно, соединенные двумя дугами, тремя дугами и т. д. — *Примеч. авт.*

колес и одного корпуса. Процесс производства является гибким, в нем часто комбинируются стандартные детали, поэтому ножи, двигатели, колеса и корпуса в различных комбинациях образуют разные модели газонокосилок.

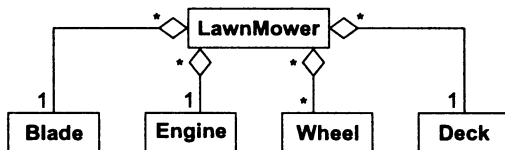


Рис. 4.9. Агрегация

Решение об использовании агрегации является субъективным и может быть достаточно произвольным. Часто бывает не вполне очевидно, должна ли некоторая ассоциация моделироваться как агрегация. Обычно эта неопределенность относится именно к модели. Моделирование вообще требует опыта в принятии решений. В нем нет строго определенных правил. По нашему опыту, при аккуратных рассуждениях и последовательном принятии решений нечеткие различия между агрегациями и ассоциациями на практике не доставляют особых сложностей.

4.4.2. Агрегация и композиция

В UML определено две формы отношения «часть-целое»: общая (агрегация) и частная (композиция).

Композиция (composition) — это частный случай агрегации, характеризующийся двумя дополнительными ограничениями. Составляющая часть может принадлежать не более чем одному агрегату. Более того, составляющая часть, приписанная к некоторому агрегату, автоматически получает срок жизни, совпадающий со сроком жизни агрегата. Таким образом, композиция подразумевает, что части *принадлежат* целому. Это удобно для программирования: удаление объекта-агрегата автоматически вызывает удаление всех его составляющих, если он образует их композицию. Для обозначения композиции используется небольшой закрашенный ромбик, который ставится рядом с классом-агрегатом (для агрегации, не являющейся композицией, используется незакрашенный ромбик).

На рис. 4.10 изображена компания, состоящая из отделений, которые, в свою очередь, состоят из отделов. Компания косвенно является композицией отделов. Однако компания не является композицией сотрудников, поскольку компания и человек являются независимыми объектами одинакового уровня.

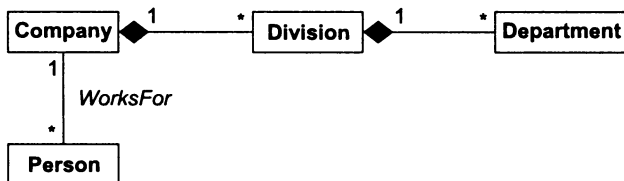


Рис. 4.10. Композиция

4.4.3. Распространение операций

Распространение (propagation или triggering) операций — автоматическое применение операции к сети объектов, вызываемое применением этой операции к некоторому начальному объекту [Rumbaugh-88]¹. Например, перемещение агрегата перемещает его составляющие, таким образом, операция перемещения распространяется на части целого. Распространение операций на составляющие является одним из явных признаков агрегации.

На рис. 4.11 показан пример распространения. У человека может быть множество документов. Каждый документ состоит из абзацев, которые, в свою очередь, состоят из символов. Операция копирования распространяется с документа на его абзацы и символы. Копирование абзаца приводит к копированию всех его символов. В обратном направлении операция не распространяется: абзац можно скопировать, не копируя весь документ. Аналогичным образом, копирование документа копирует и связь с владельцем, но не приводит к порождению нового экземпляра владельца документа.

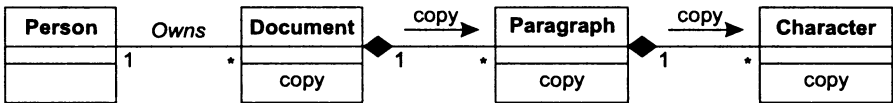


Рис. 4.11. Распространение операций по агрегациям и композициям

В других подходах чаще всего приходится делать выбор «все или ничего»: либо копировать всю сеть, либо копировать только начальный объект. Концепция распространения операций является ясным и мощным средством описания области действия поведения. Операция начинается с некоторого начального объекта и переходит с объекта на объект по связям в соответствии с правилами распространения. Распространение возможно и для других операций: сохранение/восстановление, уничтожение, печать, блокирование, отображение.

В моделях классов распространение обозначается небольшой стрелкой, которая указывает направление. Кроме того, около ассоциации, по которой происходит распространение, ставится имя операции. Эта система обозначений привязывает распространение к ассоциации (агрегации), направлению и поведению. Однако заметьте, что данная система обозначений не является частью UML.

4.5. Абстрактные классы

Абстрактным (abstract) называется класс, не имеющий непосредственных экземпляров. Непосредственные экземпляры могут быть у потомков класса. Конкретным называется класс, который может порождать экземпляры. Конкретный класс может иметь абстрактные подклассы (но у них, в свою очередь, обязательно должны быть конкретные потомки). Листьями дерева наследования могут быть только конкретные классы.

¹ Термин «ассоциация» в этой книге совпадает по смыслу с термином «отношение», который использовался Рамбо [Rumbaugh-88].

Все виды деятельности на рис. 4.12 являются конкретными классами. *Butcher* (Мясник), *Baker* (Пекарь) и *CandlestickMaker* (ПроизводительПодсвечников) — конкретные классы, поскольку у них есть непосредственные экземпляры. *Worker* (Работник) — это тоже конкретный класс, потому что некоторые виды деятельности не имеют собственных названий.



Рис. 4.12. Конкретные классы

Примером абстрактного класса является класс *Employee* (Служащий) на рис. 4.13. Все служащие должны иметь либо полную, либо частичную занятость. *FullTimeEmployee* (СлужащийНаПолнойЗанятости) и *PartTimeEmployee* (СлужащийНаЧастичнойЗанятости) — конкретные классы, поскольку они могут иметь непосредственные экземпляры. В системе обозначений UML имя абстрактного класса выделяется курсивом. Под именем абстрактного класса или после него можно указать ключевое слово *{abstract}*.

Абстрактные классы могут использоваться для определения методов, которые будут наследоваться подклассами. Абстрактный класс может определить только сигнатуру операции без реализующего ее метода. Такая операция называется абстрактной. (Вспомните, что операция указывает форму функции или процедуры. Реализацией ее является метод.) Абстрактная операция определяет сигнатуру операции, реализацию которой должен предоставить каждый конкретный подкласс. Конкретный класс не может содержать абстрактные операции, потому что иначе его экземпляры имели бы неопределенные методы.

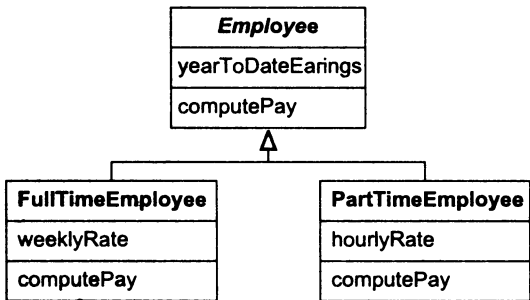


Рис. 4.13. Абстрактный класс и абстрактная операция

На рис. 4.13 показана абстрактная операция. Она выделяется курсивом или ключевым словом *{abstract}*. В нашем примере *ComputePay* (ВычислитьЗарплату) является абстрактной операцией класса *Employee* (Сотрудник). В классе определена только сигнатура операции, но не ее реализация. Каждый конкретный подкласс должен предоставить метод, реализующий данную операцию.

С точки зрения стиля моделирования, лучше всего стараться избегать конкретных суперклассов. В этом случае деление классов на абстрактные и конкретные становится очевидным: все суперклассы являются абстрактными, а все листья дерева наследования — конкретными. Более того, это позволит вам избежать неудачных ситуаций, в которых конкретный суперкласс должен не только указывать сигнатуру операции для своих потомков, но и приводить ее реализацию. Избежать конкретности суперкласса можно, добавив в модель подкласс *Other* (Прочие), как показано на рис. 4.14.

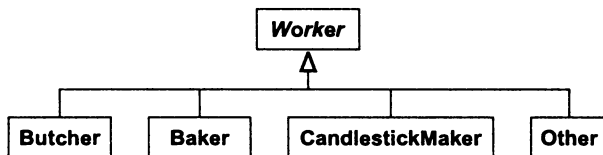


Рис. 4.14. Как сделать конкретный суперкласс абстрактным

4.6. Множественное наследование

Множественное наследование позволяет классу иметь несколько суперклассов и наследовать составляющие от всех предков. Это позволяет смешивать информацию из нескольких источников. Множественное наследование является более сложной формой обобщения по сравнению с единичным наследованием, ограничивающим иерархию классов до дерева. Преимущество множественного наследования в увеличении возможностей спецификации классов и их повторного использования. Недостаток — усложнение концепций и реализации.

Термин множественное наследование иногда используется для обозначения концептуального отношения между классами или механизма языка, реализующего это отношение. Мы всегда стараемся делать различие между обобщением (концептуальным отношением) и наследованием (механизмом языка), однако термин «множественное наследование» используется чаще, чем «множественное обобщение».

4.6.1. Виды множественного наследования

Чаще всего множественное наследование используется для наследования от множества несовместных классов. Каждый подкласс является потомком одного класса из каждого множества. На рис. 4.15 *FullTimeIndividualContributor* является одновременно *FullTimeEmployee* и *IndividualContributor* и сочетает в себе их составляющие. Классы *FullTimeEmployee* и *PartTimeEmployee* являются несовместными, то есть каждый сотрудник принадлежит ровно одному из них. Классы *Manager* и *IndividualContributor* тоже являются несовместными. Мы могли бы определить три дополнительных класса: *FullTimeManager*, *PartTimeIndividualContributor* и *PartTimeManager*. Допустимые комбинации зависят от приложения.

Каждое обобщение должно производиться по одному аспекту. Множественное наследование можно использовать в том случае, если класс можно уточнить по нескольким независимым аспектам. На рис. 4.15 класс *Employee* уточняется по занятости и участию в управлении. Поэтому модель содержит два набора обобщений.

Подкласс наследует составляющие от каждого суперкласса только один раз, даже если к нему ведут несколько путей по графу обобщений. Например, на рис. 4.15 *FullTimeIndividualContributor* наследует составляющие *Employee* по двум путям: через *employmentStatus* и *managerialStatus*. Однако каждый *FullTimeIndividualContributor* будет иметь только одну копию составляющих *Employee*.

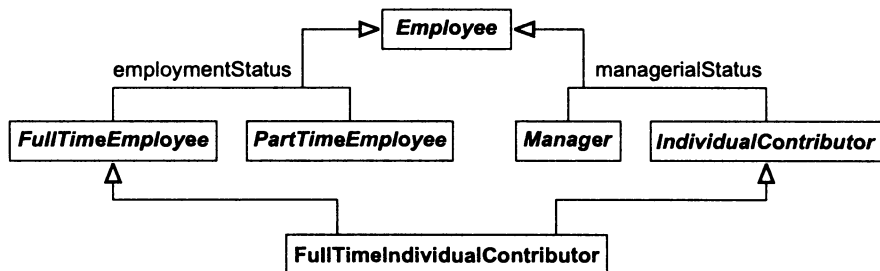


Рис. 4.15. Множественное наследование от несовместных классов

Конфликты между параллельными определениями создают двусмысленности, которые должны разрешаться на уровне реализации. На практике следует избегать подобных двусмысленностей в моделях или разрешать их явным образом, даже если в используемом языке программирования имеются правила разрешения конфликтов. Предположим, к примеру, что в классах *FullTimeEmployee* и *IndividualContributor* определен атрибут *name*. При этом *FullTimeEmployee.name* может подразумевать полное имя сотрудника, а *IndividualContributor.name* — его титул или звание. Очевидного способа разрешения такого конфликта не существует. Лучше всего переименовать атрибуты, например, так: *FullTimeEmployee.personName* и *IndividualContributor.title*.

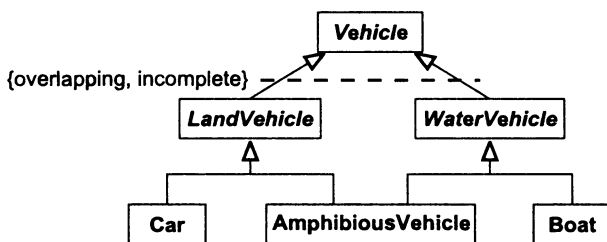


Рис. 4.16. Множественное наследование от перекрывающихся классов

Множественное наследование может иметь место с перекрывающимися классами. На рис. 4.16 *AmphibiousVehicle* (Амфибия) является одновременно *LandVehicle* (НаземноеТранспортноеСредство) и *WaterVehicle* (НадводноеТранспортноеСредство). Классы *LandVehicle* и *WaterVehicle* перекрываются, поскольку некоторые транспортные средства могут перемещаться как по земле, так и по воде. Для обозначения перекрывающихся наборов обобщений в UML используются ограничения (см. раздел 4.9). Для обозначения используется пунктирная линия, которая проводится поперек перекрывающихся обобщений. Около линии ставятся ключевые слова в фигурных скобках. В нашем примере ключевое слово *overlapping*

означает, что транспортное средство может принадлежать одновременно к нескольким подклассам. Ключевое слово *incomplete* означает, что на диаграмме не перечислены все возможные подклассы класса *Vehicle*.

4.6.2. Множественная классификация

Экземпляр класса неявным образом является экземпляром всех предков этого класса. Например, преподаватель может одновременно принадлежать к профессорско-преподавательскому составу и к учащимся. Но может ли профессор Гарвардского университета слушать лекции в Массачусеттском технологическом институте? Такая комбинация не описывается никаким классом (если бы он существовал, это было бы крайне искусственное построение). Это пример множественного наследования, при котором один экземпляр относится к двум перекрывающимся классам.

Язык UML допускает множественную классификацию, но большинство объектно-ориентированных языков поддерживают ее довольно плохо. При использовании традиционных языков лучше всего работать с *Person* как с объектом, состоящим из множества объектов *UniversityMember* (Член Университета) (рис. 4.17). Такой обходной маневр заменяет наследование делегированием (о котором речь пойдет в следующем разделе). Правда, при этом теряется индивидуальность отдельных ролей, однако альтернативные подходы потребовали бы радикальных изменений во многих языках программирования [McAllester-86].

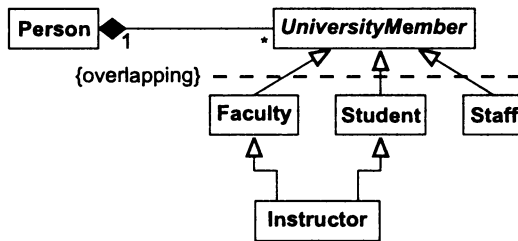


Рис. 4.17. Множественная классификация не поддерживается большинством языков

4.6.3. Обходные маневры

Проблемы с поддержкой множественного наследования должны были бы решаться на этапе реализации, но чаще всего самым простым решением оказывается реструктурирование модели на ранних этапах проектирования. В этом разделе мы рассмотрим некоторые приемы реструктурирования. Два подхода используют концепцию делегирования (delegation) — механизма реализации, при помощи которого объект передает операцию другому объекту для ее выполнения. Более подробное описание делегирования продолжается в главе 15.

- **Делегирование с использованием композиции частей.** Суперкласс с несколькими независимыми обобщениями можно переформулировать в виде композиции, в которой обобщение будет заменяться составляющими частями. Этот подход аналогичен описанному в предыдущем разделе для множественной классификации. Один объект с уникальным идентификатором

заменяется группой связанных объектов, образующих композицию (составной объект). Наследование операций не является автоматическим. Составной объект должен «перехватывать» операции и направлять их соответствующим частям.

Например, на рис. 4.18 *EmployeeEmployment* (ЗанятостьСлужащего) становится суперклассом *FullTimeEmployee* (СлужащийНаПолнойЗанятости) и *PartTimeEmployee* (СлужащийНаНеполнойЗанятости). Класс *EmployeeManagement* (УчастиеСлужащегоВУправлении) становится суперклассом *Manager* (Управляющий) и *IndividualContributor* (Сотрудник). Тогда *Employee* (Служащий) может быть представлен в виде композиции *EmployeeEmployment* и *EmployeeManagement*. Операция, вызванная у объекта *Employee*, должна быть перенаправлена одной из его частей (*EmployeeEmployment* или *EmployeeManagement*).

В этом подходе вам не придется создавать различные комбинации (типа *FullTimeIndividualContributor*) в виде отдельных классов. Допустимы любые комбинации подклассов различных обобщений.

- **Наследование наиболее важных классов с делегированием остальных.** В диаграмме классов на рис. 4.19 сохранены индивидуальность и наследование для наиболее важных обобщений. Остальные обобщения низведены до композиции, а их операции делегированы, как в предыдущем варианте.

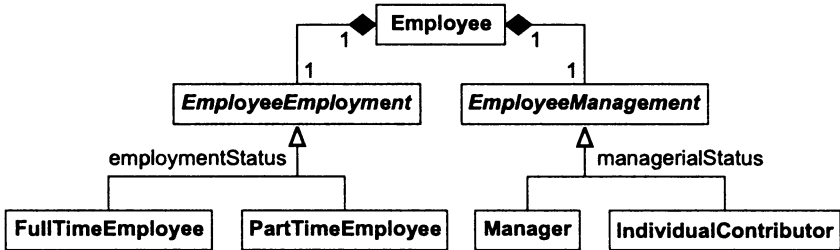


Рис. 4.18. Делегирование

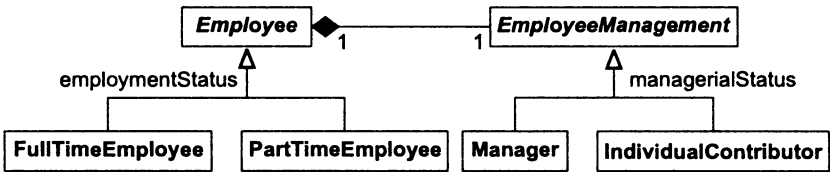


Рис. 4.19. Наследование и делегирование

- **Вложенное обобщение.** Сначала система факторизуется по первому обобщению, затем по второму. При этом перебираются все возможные комбинации. Например, на рис. 4.20 мы добавили по два подкласса для управляющих и сотрудников в классы *FullTimeEmployee* и *PartTimeEmployee*. При этом сохраняется наследование, но зато удваивается количество объявлений (и кода) и не сохраняется дух объектно-ориентированного программирования.

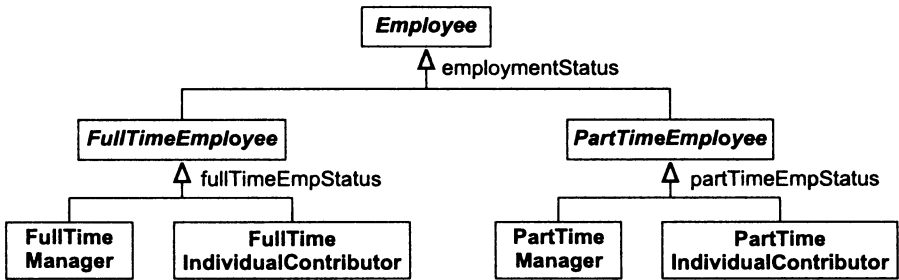


Рис. 4.20. Замена множественного наследования на вложенные обобщения

Любой обходной маневр пригоден для облегчения реализации, но все они ухудшают логическую структуру и затрудняют поддержку программ. При выборе подходящего обходного маневра следует руководствоваться несколькими принципами.

- **Суперклассы одинаковой важности.** Если подкласс имеет несколько суперклассов и все они одинаково важны, лучше всего воспользоваться делегированием (см. рис. 4.18) и сохранить симметрию модели.
- **Доминирующий суперкласс.** Если один суперкласс очевидно является наиболее важным, а остальные — менее важны, нужно сохранить наследование от этого суперкласса (см. рис. 4.19 или 4.20).
- **Несколько подклассов.** Если количество комбинаций невелико, можно воспользоваться вложенным обобщением (см. рис. 4.20). В противном случае от него следует отказаться.
- **Упорядочение наборов обобщений.** Если вы использовали вложенное обобщение (см. рис. 4.20), сначала следует обобщать по наиболее важному критерию, затем — по остальным.
- **Большие объемы кода.** Следует избегать вложенных обобщений (см. рис. 4.20), если для их реализации придется дублировать большие объемы кода.
- **Индивидуальность.** Учитывайте важность сохранения индивидуальности. Это обеспечивается только вложенным обобщением (см. рис. 4.20).

4.7. Метаданные

Метаданные (metadata) — это данные о данных. Хорошим примером метаданных является определение класса. Модели по определению являются метаданными, потому что они описывают моделируемые объекты (а не являются ими). Метаданные используются во многих применяемых на практике приложениях: это каталоги деталей, светокопии и словари. Столь же широко используются метаданные в реализациях компьютерных языков.

На рис. 4.21 приведен пример, позволяющий сравнить концепции данных и метаданных. Модель автомобиля характеризуется названием, годом, базовой ценой и производителем. В качестве примеров моделей автомобилей можно привести

Ford Mustang 1969 года и Volkswagen Rabbit 1975 года. Автомобиль как физическая сущность характеризуется серийным номером, цветом, комплектацией и владельцем. В качестве примера можно привести голубой Ford с серийным номером 1FABP и красный Volkswagen с серийным номером 7E81F, принадлежащие Джону До. Модель автомобиля описывает множество автомобилей и содержит общие для них всех данные. Модель автомобиля является метаданными по отношению к реальному автомобилю (который в нашем примере представляет собственно данные).

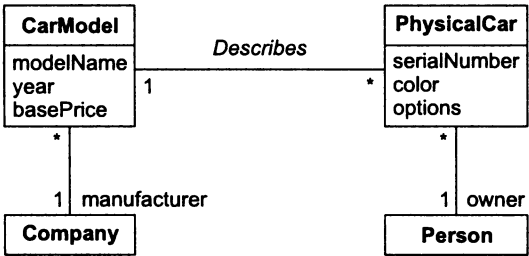


Рис. 4.21. Метаданные и данные

Вы можете рассматривать классы как объекты, но на самом деле они являются метаобъектами, а не реальными объектами. Объекты дескрипторов классов обладают составляющими, а те, в свою очередь, характеризуются собственными классами, которые называются метаклассами. Если же рассматривать все сущности как объекты, реализация оказывается более единообразной, и функциональные возможности для решения сложных проблем при таком подходе возрастают. В разных языках доступность метаданных оказывается различной. В некоторых языках, например Lisp и Smalltalk, метаданные могут считываться и изменяться программами во время их выполнения. В других языках, например C++ и Java, метаданные обрабатываются во время компиляции, а во время выполнения они не доступны (по крайней мере, явным образом).

4.8. Воплощение

Воплощение (reification) — это превращение в объект чего-либо не являющегося таковым. Воплощение полезно для работы с метаприложениями, потому что оно позволяет сместить уровень абстрагирования. Иногда оно полезно для превращения атрибутов, методов, ограничений и управляющей информации в объекты, чтобы иметь возможность описывать их и манипулировать ими, как данными.

В качестве примера воплощения рассмотрим администратора базы данных. Разработчик может написать код для каждого приложения, чтобы оно могло читать файлы и записывать их. Однако для многих приложений лучше воплотить идею службы данных в виде администратора базы данных. Администратор обладает абстрактной функциональностью, которая дает универсальное решение, позволяющее множеству пользователей надежно и быстро обращаться к данным.

В качестве второго примера рассмотрим диаграммы состояний и переходов (о них речь пойдет в следующих двух главах). Диаграмма состояний и переходов

может использоваться для задания схемы управления и последующей реализации этой схемы при написании соответствующего кода. Альтернативный подход заключается в подготовке метамодели, в которой модель состояний и переходов будет храниться как данные. Универсальный интерпретатор будет считывать содержимое метамодели и выполнять его, реализуя нужную последовательность управления.

На рис. 4.22 атрибут *substanceName* был сделан классом для того, чтобы описать отношение многие-ко-многим между *SubstanceName* и *Substance*. Химическое вещество может быть известно под несколькими названиями. Например, пропилен можно назвать пропиленом или C_3H_6 . И наоборот: под одним и тем же названием может скрываться несколько химических веществ. Различные смеси этиленгликоля с присадками имеют общее название «антифриз».

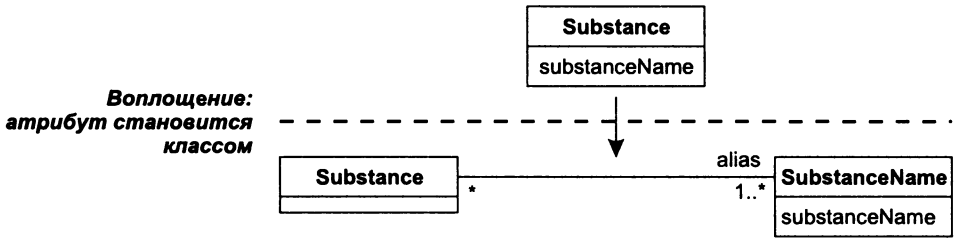


Рис. 4.22. Воплощение

4.9. Ограничения

Ограничение (constraint) — это логическое условие, накладываемое на элементы модели, такие как объекты, классы, атрибуты, связи, ассоциации и наборы обобщений. Ограничение действует на значения, которые могут принимать элементы. Ограничения можно записывать на естественном или на формальном языке (например, OCL [Warmer-99]).

4.9.1. Ограничения на объекты

На рис. 4.23 приведены несколько примеров ограничений, наложенных на объекты. Зарплата сотрудника не может превышать зарплату босса (ограничение наложено одновременно на два объекта). Ни у какого окна соотношение сторон (длины к ширине) не может быть меньше 0.8 или больше 1.5 (ограничение на атрибуты одного объекта). Приоритет задачи нельзя повысить (ограничение на один и тот же объект в разные моменты времени). В моделях классов ограничения могут быть достаточно простыми.

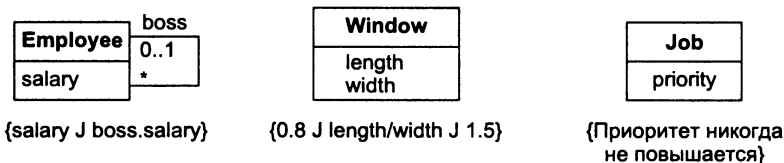


Рис. 4.23. Ограничения на объекты

4.9.2. Ограничения на наборы обобщений

Модели классов содержат множество ограничений в самой своей структуре. Например, семантика обобщения подразумевает некоторые структурные ограничения. При единичном наследовании подклассы являются взаимоисключающими. Более того, каждый экземпляр абстрактного суперкласса является экземпляром ровно одного его подкласса. Каждый экземпляр конкретного суперкласса является экземпляром не более, чем одного из его подклассов.

На рис. 4.16 и 4.17 ограничения используются в качестве вспомогательных средств для описания множественного наследования. UML разрешает использовать с наборами обобщений следующие ключевые слова:

- **disjoint (несовместные)**. Подклассы взаимно исключают друг друга. Любой объект принадлежит ровно одному из них.
- **overlapping (перекрывающиеся)**. Подклассы могут иметь общие объекты. Объект может принадлежать сразу нескольким подклассам.
- **complete (полное)**. В обобщении перечислены все возможные подклассы.
- **incomplete (неполное)**. Некоторые подклассы могли быть пропущены.

4.9.3. Ограничения на связи

Кратность — это ограничение на количество элементов множества. Кратность ассоциации ограничивает количество объектов, которые могут быть связаны с одним конкретным объектом. Кратность атрибута определяет количество значений, которые может принимать конкретный экземпляр атрибута.

Квалификаторы также являются ограничениями для ассоциации. Атрибут квалификатора не просто описывает связи ассоциации. Он полезен для различения множества объектов у противоположного полюса.

Класс ассоциации подразумевает некоторое ограничение. Он является классом во всех отношениях: может иметь атрибуты и операции, участвовать в ассоциациях и обобщениях. Ограничение заключается в том, что его индивидуальность возникает из экземпляров связываемых им классов.

Обычная ассоциация не предполагает никакого порядка объектов на полюсе с кратностью «много». Ограничение *{ordered}* указывает, что элементы у этого полюса имеют определенный порядок, который должен сохраняться.

На рис. 4.24 показано явное ограничение, не являющееся частью структуры модели. Председатель комитета должен быть одним из его членов — ассоциация *ChairOf* является подмножеством ассоциации *MemberOf*.

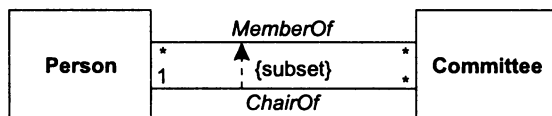


Рис. 4.24. Отношение подмножества накладывает ограничение на ассоциации

4.9.4. Использование ограничений

Мы рекомендуем выражать ограничения в явном виде, то есть объявлять их в модели. Объявление дает вам возможность выразить суть ограничения без всяких предположений о возможной реализации. Обычно ограничения приходится преобразовывать в форму процедур перед тем, как реализовывать их на языке программирования, но это преобразование чаще всего бывает достаточно простым.

Ограничения позволяют оценить качество модели классов: в «хорошей» модели значительная часть ограничений заложена в саму структуру. Часто требуется несколько итераций проектирования, чтобы структура получилась правильной с точки зрения ограничений. На практике чаще всего не получается заложить *все* ограничения в структуру, но нужно стараться добиться этого по крайней мере для наиболее важных из них.

В UML имеется две альтернативные системы обозначения ограничений. Ограничение можно отделить фигурными скобками или поместить его в прямоугольник комментария с подогнутым краем (так называемое «собачье ухо» — рис. 4.26). В любом случае нужно стараться размещать ограничения около тех элементов, к которым они относятся. Для соединения элементов можно использовать пунктирные линии. Пунктирная линия может связывать элемент, на который наложено ограничение, с тем элементом, от которого это ограничение зависит.

4.10. Производные данные

Производный элемент (derived element) — это функция одного или нескольких элементов, которые, в свою очередь, тоже могут быть производными. Производный элемент является избыточным, поскольку он полностью определяется другими элементами. Дерево вывода заканчивается базовыми элементами. Производными могут быть классы, ассоциации и атрибуты. Перед именем производного элемента ставится косая черта. Необходимо также указать ограничение, определяющее порядок вычисления элемента.

На рис. 4.25 показан пример производного элемента.*Возраст можно рассчитать по дате рождения и текущей дате.

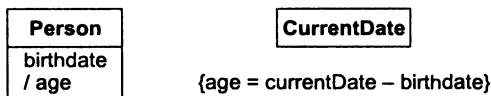


Рис. 4.25. Производный атрибут

На рис. 4.26 станок состоит из нескольких агрегатов, которые, в свою очередь, состоят из деталей. Агрегат характеризуется смещением относительно координат станка. Каждая деталь смещена относительно координат агрегата. Мы можем определить систему координат для каждой детали, которая будет определяться координатами станка, смещением агрегата и смещением детали. Эта система координат будет представлена в модели производным классом *Offset*, связанным с каждой деталью производной ассоциацией *NetOffset*.

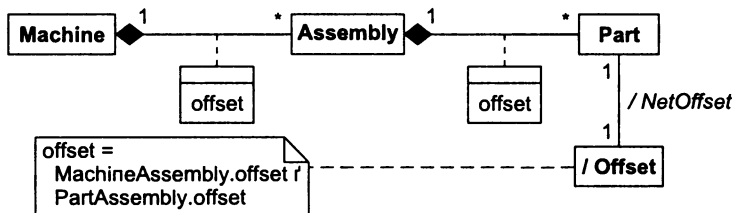


Рис. 4.26. Производный объект и производная ассоциация

Полезно отделять операции с побочными эффектами от операций, возвращающих некоторое значение, но не создающих побочных эффектов. Последние называются *запросами* (query). Запросы без аргументов (за исключением целевого объекта) могут рассматриваться как производные атрибуты. Например, ширину прямоугольника можно вычислить по координатам его сторон. Во многих случаях объект обладает множеством атрибутов со связанными между собой значениями, из которых лишь некоторые могут быть определены независимо. Модель классов должна описывать отличие независимых базовых атрибутов (base attributes) от зависимых производных атрибутов (derived attributes). Выбор базовых атрибутов произволен, но его необходимо сделать, чтобы не переопределять состояние объекта.

Некоторые разработчики включают в модель много производных элементов. Обычно это не приносит никакой пользы и перегружает модель. Производные элементы стоит включать только в том случае, если они отражают важные для приложения концепции или значительно упрощают реализацию. Поддерживать согласованность производных элементов с базовыми может быть довольно затруднительно, поэтому использовать их для реализации нужно с осторожностью.

4.11. Пакеты

Модель классов можно уместить на одной странице, если вы решаете задачу небольшого или среднего размера. Большую модель бывает трудно охватить целиком. Мы рекомендуем разбивать большие модели на части, чтобы другим людям было проще понять их.

Пакет — это группа элементов (классов, ассоциаций, обобщений и вложенных пакетов), объединенных общей темой. Пакеты разбивают модель на части, что упрощает ее понимание и поддержку. Для крупных приложений может потребоваться несколько уровней пакетов. Пакеты образуют дерево, уровень абстрактности которого увеличивается в направлении корня — приложения, которое одновременно является пакетом верхнего уровня. Пакет обозначается прямоугольником с закладкой (рис. 4.27). Закладка подразумевает наличие вложенного содержимого и должна вызывать ассоциации с папкой.



Рис. 4.27. Обозначение пакета

Тема пакета может быть достаточно произвольной. Темой может быть основной класс, основные отношения, важнейшие аспекты функциональности или симметрия. Во многих бизнес-системах имеются пакеты, которые называются *Customer (Клиент)* или *Part (Деталь)*. *Клиент* и *Деталь* — основные классы, важные для бизнеса, поэтому они и присутствуют во множестве приложений. В инженерном приложении мы разделили модель классов на пакеты по преобладающему отношению. Модель была нацелена на описание оборудования, а атрибуты и отношения сильно отличаются у разных типов оборудования. Модель классов компилятора можно было бы разделить на пакеты лексического анализа, разбора, семантического анализа, генерации кода и оптимизации. После формирования пакетов можно воспользоваться соображениями симметрии для формирования дополнительных пакетов.

Ниже мы приводим рекомендации, которыми стоит воспользоваться при проектировании пакетов.

- **Аккуратно определяйте область каждого пакета.** Точные границы пакета зависят только от вас. Определение области пакета требует планирования и организации, подобно многим другим аспектам моделирования. Следите за тем, чтобы имена классов и ассоциаций в каждом пакете были уникальными и по возможности согласовывались друг с другом в разных пакетах.
- **Определяйте каждый класс только в одном пакете.** В этом пакете должны быть указаны название класса, его атрибуты и операции. В остальных пакетах, ссылающихся на этот класс, достаточно значка класса — прямоугольника с его именем. Такое соглашение облегчает чтение моделей классов, потому что сразу видно, к какому пакету относится какой-либо класс. Читатель не отвлекается на определения, которые могут быть несогласованными (если вы забудете исходное определение класса). Кроме того, оно упрощает параллельную разработку пакетов.
- **Делайте пакеты удобными для совместного использования.** Ассоциации и обобщения должны уместиться внутри одного пакета, но классы могут присутствовать в разных пакетах и тем самым связывать их. В среднем не более 20—30 % классов должны присутствовать в нескольких пакетах.

4.12. Практические рекомендации

В этой главе мы приводим новые рекомендации, в дополнение к тем, что были даны в главе 3.

- **Перечисления.** При конструировании модели нужно объявлять перечисления и множество их значений, поскольку встречаются они довольно часто и важны для пользователей. Не создавайте ненужных обобщений для атрибутов, которые являются перечислениями. Класс следует специализировать только в том случае, если его подклассы обладают собственными атрибутами, операциями или ассоциациями (раздел 4.1.1).

- **Статические атрибуты.** Статический атрибут допустимо использовать для хранения экстенста класса (множества его объектов). Других статических атрибутов следует избегать, так как они ухудшают модели. Вы можете улучшить модель, создав явные модели групп и приписав атрибуты им (раздел 4.1.3).
- **N-арные ассоциации.** Старайтесь их избегать. Большинство N-арных ассоциаций могут быть разбиты на бинарные (раздел 4.3).
- **Конкретные суперклассы.** С точки зрения стиля, их тоже следует избегать. Удобно делать все суперклассы абстрактными, а все подклассы-листья — конкретными. Избежать конкретизации суперкласса можно, добавив подкласс *Other* (Прочие) (раздел 4.5).
- **Множественное наследование.** Стоит использовать только в том случае, если оно действительно необходимо для модели (раздел 4.6).
- **Ограничения.** Вы можете попытаться реструктурировать модель классов, чтобы охватить дополнительные ограничения ее структурой (раздел 4.9).
- **Производные элементы.** Производность элемента нужно обозначать всегда, а пользоваться производными элементами следует по возможности реже (раздел 4.10).
- **Крупные модели.** Для организации крупных моделей лучше всего использовать пакеты. Это позволит читателю воспринимать модель по частям, а не целиком за один раз (раздел 4.11).
- **Определение классов.** Каждый класс следует определять в отдельном пакете, где и указывать его составляющие. В пакетах, ссылающихся на этот класс, достаточно значка класса — прямоугольника с его именем. Это упрощает чтение моделей и облегчает параллельную разработку (раздел 4.11).

4.13. Резюме

В этой главе мы затронули несколько тем и раскрыли тонкости моделирования классов. Для простых моделей описанные здесь концепции не понадобятся, но для сложных приложений они могут оказаться необходимыми. Помните, что содержимое модели должно определяться потребностями приложения. Усложненные концепции нужно использовать только тогда, когда они помогают создать приложение, делая модель более ясной, совершенствуя ее структуру или позволяя выразить сложную идею.

Тип данных является описанием значений. Каждый атрибут должен получить свой тип, иначе модель не может быть реализована. Перечисление — это особый тип данных с конечным набором возможных значений. Перечисляемые значения часто преобладают в пользовательских интерфейсах.

Кратность — это ограничение на количество элементов множества. Она применима к атрибутам и к ассоциациям. Кратность ассоциации ограничивает количество объектов, которые могут быть связаны с конкретным объектом. Кратность атрибута

ограничивает количество значений, которые могут быть присвоены конкретному экземпляру атрибута.

N-арных ассоциаций лучше избегать, разбивая их на бинарные (чаще всего это возможно). Большинство языков программирования потребуют превращения N-арных ассоциаций в классы.

Агрегация — это частный случай ассоциации, в которой объект-агрегат состоит из составляющих частей. Агрегация характеризуется транзитивностью и антисимметрией. Эти свойства отличают ее от ассоциации. Операции над агрегатом часто распространяются на составляющие части.

Композиция — это частный случай агрегации с двумя дополнительными ограничениями. Часть композиции может принадлежать только одному агрегату. Как только она была отнесена к какому-либо агрегату, ее время жизни становится равным времени жизни целого агрегата. Композиция подразумевает, что часть принадлежит целому.

Абстрактный класс не может иметь непосредственных экземпляров. Конкретный класс может иметь непосредственные экземпляры. Абстрактные классы позволяют определить методы, которые будут использоваться несколькими подклассами. Кроме того, в них можно определить сигнатуру операции, а реализацию предоставить в подклассах.

Множественное наследование позволяет подклассу наследовать составляющие нескольких суперклассов. Каждое обобщение должно проводиться по одному аспекту. Подклассы следует разбить на несколько обобщений, если они специализируют суперкласс по нескольким аспектам. Подкласс может наследовать составляющие от классов из нескольких обобщений и даже из перекрывающихся обобщений, но он не может быть потомком двух классов одного обобщения, если оно является несовместным.

Явные ограничения на классы, ассоциации и атрибуты повышают точность модели. Примерами встроенных ограничений являются обобщения и кратность. В модели могут присутствовать производные элементы, но они не несут в себе никакой информации.

Пакет — это группа классов, ассоциаций, обобщений и вложенных пакетов, объединенных общей темой. Пакеты позволяют разбить крупную модель на части, чтобы облегчить ее восприятие и поддержку.

Таблица 4.1. Ключевые понятия главы

абстрактный класс	конкретный класс	обобщение	пакет
абстрактная операция	ограничение	метаданные	распространение
агрегация	делегирование	множественное наследование	воплощение
полюс ассоциации	производный элемент	кратность (атрибута)	область действия
композиция	перечисление	n-арная ассоциация	видимость

Библиографические замечания

В книге [Rumbaugh-05] подробно рассказывается о многих тонкостях UML, которым не нашлось места в этой главе. Книга [Warmer-99] является официальным описанием языка объектных ограничений OCL.

В предыдущем издании этой книги для указания кратности N-арных ассоциаций использовались потенциальные ключи (candidate keys). В этом контексте потенциальный ключ является минимальным набором полюсов ассоциации, уникально идентифицирующим связь. В настоящем издании мы не рассказывали об этой концепции, потому что она редко используется в программировании. Кроме того, концепция области (scope) заменила термины атрибут класса (class attribute) и операция класса (class operation) из предыдущего издания.

Ссылки

[McAllester-86] David McAllester, Ramin Zabih. Boolean classes. OOPSLA'87 as SIGPLAN 22, 12 (December 1987), 417-424.

[Rumbaugh-88] James E. Rumbaugh. Controlling propagation of operations using attributes on relations. OOPSLA'88 as ACM SIGPLAN 23, 11 (November 1988), 285-296.

[Rumbaugh-05] James Rumbaugh, Ivar Jacobson, Grady Booch. The Unified Modeling Language Reference Manual, Second Edition. Boston: Addison-Wesley, 2005.

[Warmer-99] Jos Warmer and Anneke Kleppe. The Object Constraint Language. Boston: Addison-Wesley, 1999.

Упражнения

- 4.1. (3) На диаграмме классов (рис. У4.1) представлена часть структуры автомобиля. Усовершенствуйте ее, заменив некоторые ассоциации на агрегации.
- 4.2. (4) На рис. У4.2 представлена частично законченная диаграмма классов интерактивного редактора диаграмм. Лист — это совокупность линий и прямоугольников. Линия — это последовательность прямолинейных сегментов, соединяющих два прямоугольника. Каждый сегмент определяется двумя конечными точками. Точка может принадлежать вертикальному и горизонтальному сегменту одной и той же линии. Выделение — это совокупность линий и прямоугольников, выделенная пользователем (для последующего редактирования). Буфер — это совокупность линий и прямоугольников, вырезанная или скопированная с листа.

В таком виде диаграмма не выражает ограничения, которое заключается в том, что линия или прямоугольник могут принадлежать ровно одному буферу, или выделению, или листу. Пересмотрите диаграмму классов и воспользуйтесь обобщениями, чтобы выразить это ограничение, создав суперкласс для классов *Buffer*, *Selection* и *Sheet*. Обсудите преимущества пересмотренной модели.

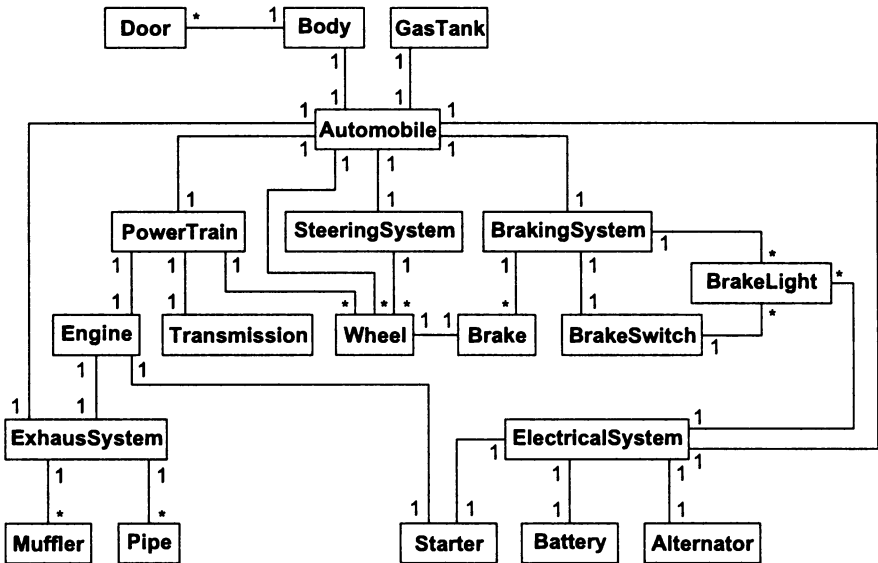


Рис. У4.1. Часть структуры автомобиля

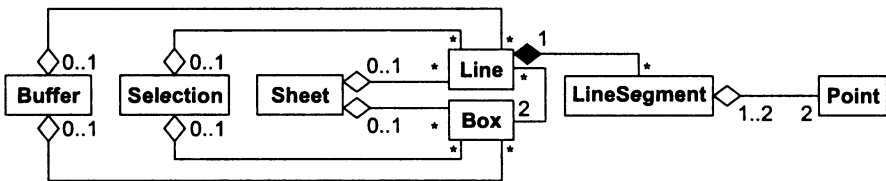


Рис. У4.2. Часть диаграммы классов простого редактора диаграмм

4.3. (3) Охарактеризуйте перечисленные ниже отношения как обобщения, агрегации или ассоциации. В списке могут быть и n-арные ассоциации, поэтому не пытайтесь исходить из предположения, что отношение, включающее несколько классов, обязательно является обобщением. Ответ поясните.

- 1) У страны есть столица.
- 2) Обедаящий философ пользуется вилок.
- 3) Файл — это обычный файл или файл каталога.
- 4) Файлы содержат записи.
- 5) Многоугольник состоит из упорядоченного множества точек.
- 6) Объект рисунка может быть текстом, геометрическим объектом или группой.
- 7) Человек использует компьютерный язык для выполнения проекта.
- 8) Модемы и клавиатуры являются устройствами ввода-вывода.
- 9) Классы могут иметь несколько атрибутов.
- 10) Человек играет за команду в определенном году.

11) Маршрут соединяет два города.

12) Студент слушает курс лекций профессора.

4.4. (7) Подготовьте диаграмму классов для редактора графических документов, поддерживающего группировку объектов. Пусть документ состоит из нескольких листов. На каждом листе могут располагаться объекты рисунка, включая текст, геометрические объекты и группы. Группа — это множество объектов рисунка, в которое могут входить другие группы. Группа должна содержать по меньшей мере два объекта. Объект может быть непосредственным членом только одной группы. К геометрическим объектам относятся окружности, эллипсы, прямоугольники, отрезки и квадраты.

4.5. (7) В этом упражнении мы рассмотрим частичную таксономию электрических двигателей. С аналитической точки зрения электродвигатели можно разделить на двигатели постоянного тока и двигатели переменного тока. Некоторые двигатели работают на постоянном токе, некоторые на переменном, а остальные — на любом. В качестве примеров электродвигателей можно привести большие синхронные электродвигатели, небольшие индукционные электродвигатели, универсальные электродвигатели и двигатели с постоянными магнитами. В домашнем хозяйстве чаще всего используются индукционные и универсальные электродвигатели.

Электродвигатель переменного тока может быть синхронным или индукционным. Универсальные двигатели используются для достижения высоких скоростей, например, в измельчителях и пылесосах. Они могут работать как на постоянном, так и на переменном токе. Двигатели с постоянными магнитами часто применяются в игрушках и работают только на постоянном токе.

Изобразите описанные категории двигателей и их отношения на диаграмме классов. При необходимости используйте множественное наследование.

4.6. (7) Переделайте диаграмму классов, полученную в предыдущем упражнении, исключив из нее множественное наследование.

4.7. (8) Подготовьте метамодель, поддерживающую ограниченный набор концепций UML: класс, атрибут, ассоциация, полюс ассоциации, кратность, имя класса и имя атрибута. При построении метамодели пользуйтесь только этими конструкциями.

4.8. (9) Подготовьте диаграмму объектов метамодели, полученной в предыдущем упражнении. Рассматривайте метамодель как диаграмму классов, которая может быть представлена экземплярами классов метамодели.

4.9. (5) Переделайте ответ к упражнению 4.7 с использованием обобщений, но так, чтобы атрибут мог принадлежать либо классу, либо ассоциации, но не обоим одновременно.

4.10. (7) На рис. У4.3 приведена часть метамодели, описывающей обобщение. Обобщение связано с несколькими ролями, которые классы могут играть в отношении обобщения. Типов ролей всего два: подкласс и суперкласс. Поддерживает ли эта модель множественное наследование? Поясните свой ответ.

- 4.11. (8) Опишите, каким образом можно найти суперкласс обобщения при помощи метамодели с рис. У4.3. Переделайте метамодель таким образом, чтобы упростить этот запрос. Опишите, каким образом следует искать суперкласс обобщения в вашей метамодели. Проверьте, что она поддерживает множественное наследование. Напишите запросы OCL для поиска суперкласса по обобщению для метамодели с рис. У4.3 и для вашей метамодели.

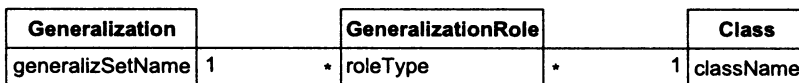


Рис. У4.3. Метамодель обобщения

- 4.12. (7) Насколько хорошо метамодель с рис. У4.3 выражает ограничение, состоящее в том, что у каждого обобщения должен быть только один суперкласс? Переделайте ее с точки зрения этого ограничения.
- 4.13. (7) На рис. У4.3 изображена метамодель, описывающая модели классов, подобные изображенной на рис. У4.4. Подготовьте диаграмму объектов, используя классы метамодели для описания модели с рис. У4.4.

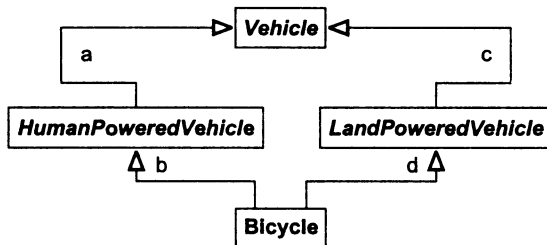


Рис. У4.4. Диаграмма классов с множественным наследованием

- 4.14. (6) Подготовьте часть диаграммы классов для системы контроля книжек в библиотеке, в которой штраф за просроченную книгу отображался бы в виде выводимого атрибута.
- 4.15. (10) Подготовьте метамодель представления компьютерных языков Бакуса—Наура (Backus-Naur, BNF). Такая модель используется в компиляторах компиляторов (например, UNIX-программе YACC). Компилятор компиляторов обрабатывает эти представления в графической форме в качестве входных данных, а на выходе дает компилятор нужного языка.

На рис. У4.5 приведен пример синтаксической диаграммы Бакуса—Наура, удовлетворяющей требованиям компилятора компиляторов. Прямоугольники обозначают нетерминальные символы, а окружности или прямоугольники со скругленными углами — терминальные. Отдельные символы заключаются в окружности, а последовательности из нескольких символов — в прямоугольники со скругленными углами. Стрелки указывают направление потока по диаграмме. В точке разветвления нескольких направленных путей можно выбрать любой из них. Название описываемого нетерминального символа ставится в начале его представления.

- 4.16. (7) Подготовьте простую модель классов, достаточную для представления рецептов. За основу возьмите рецепт с рис. У4.6. Это упражнение — на использование воплощения. В одном смысле задачи рецепта могут быть операциями. В другом — они могут быть данными в модели классов.
- 4.17. (9) Расширьте модель классов для описания рецептов, включив в нее поддержку замены ингредиентов. Например, некоторые рецепты лазаньи допускают замену прессованного творога на итальянский творог.
- 4.18. (8) Североамериканская ассоциация администраторов ценных бумаг (North American Securities Administrators Association — NASAA, www.nasaa.org) пытается защитить инвесторов и научить их торговле ценными бумагами. NASAA рекомендует инвесторам делать заметки во время разговоров с брокерами, используя форму, показанную на рис. У4.7 (если за один звонок брокер дает несколько рекомендаций, следует использовать несколько форм). Предположим, вы хотите автоматизировать эту форму, написав соответствующую программу. Подготовьте модель классов для формы NASAA.
- 4.19. (9) Подготовьте модель классов для слов из словаря. Модель должна включать: варианты написания слова, антонимы, словарь, тип (существительное, глагол, прилагательное, наречие), этимологию, варианты переноса, значения, комментарии, приоритет по частоте использования, произношение и синонимы.

В качестве примеров мы приведем несколько статей из словаря Webster's New World Dictionary:

- been (bin; also, chiefly Brit., ben &, esp. if unstressed, ben), pp. of be.
- kum quat (kum'kwot), n. [Chin. chin-chu, golden orange], 1. a small, orange-colored, oval fruit, with a sour pulp and a sweet rind, used in preserves, 2. the tree that it grows on. Also cp. cumquat.
- lac y (las'i), adj. [-IER, -IEST], 1. of lace. 2. like lace; having a delicate open pattern. — lac'I ly, adv. — laciness, n.
- Span ish (span'ish), adj. of Spain, its people, their language, etc. n. 1. the Romance language of Spain and Spanish America. 2. the Spanish people.

На рис. У4.8 приведена часть ответа к задаче: диаграмма с классами и отношениями. Добавьте в нужных местах атрибуты и упорядоченность ассоциаций.

Ассоциации *RelatedWord*, *Synonym* и *Antonim* определены не совсем правильно и обладают некоторыми недостатками. Прокомментируйте их.

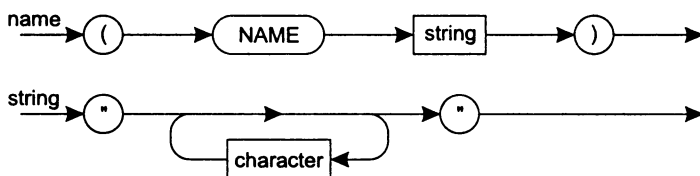


Рис. У4.5. Часть синтаксической диаграммы Бакуса—Наура

Lasagna

2.5 tbsp. salad oil for browning	2 tsp. oregano
1 cup minced onion	.5 box lasagna noodles
1 clove garlic	1 lb. ricotta cheese
1 lb. ground beef	1 cup grated mozzarella cheese
2 tsp. salt	.5 cup parmesan cheese
3.5 cups whole tomatoes (large can)	2 cans tomato paste

Cook onion, clove garlic, ground beef, 1 tsp. salt in salad oil until meat is browned. Add tomatoes, tomato paste, 1 tsp salt, oregano, and simmer, covered, 1 hour until thick. Cook noodles 15 min in water until tender. Drain and blanch. Butter 12x8 inch pan and place in layers of noodles, sauce, mozzarella, ricotta cheese and parmesan. Bake at 350 degrees for 45 to 60 min.

Рис. У4.6. Пример рецепта

Date	Time
☐ Call made ☐ Call received ☐ Meeting	Location
Name of Broker	Phone
Broker's Firm	Phone
Broker's CRD No.	☐ Obtained CRD Report
Investment Recommendation	I asked to receive written information about the investment before making a decision.
☐ Buy ☐ Sell	☐ Yes ☐ No
Name of Security	I will get:
	☐ a prospectus
	☐ an offering memorandum
	☐ most recent Annual Report
	☐ most recent quarterly or interim reports
	☐ research reports
	☐ other information
Reasons for recommendation	Proposed Trade
	Number of shares/units
	Price per shares \$
	Total cost \$
	commission
How does this meet investment objectives?	My instructions
	☐ Do nothing ☐ Buy ☐ Sell
What are the risks?	Number
	Price
	\$
Notes	
Notes made by:	

Рис. У4.7. Форма NASAA

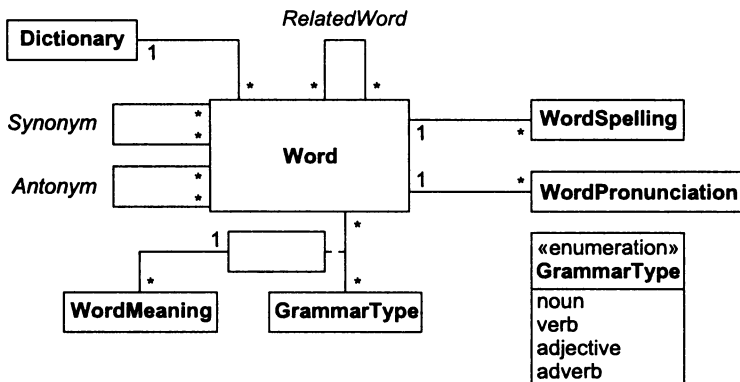
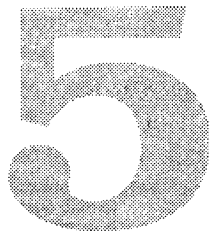


Рис. У4.8. Неполная модель словаря

Моделирование состояний



Чтобы хорошо понять систему, лучше всего сначала изучить ее статическую структуру, то есть структуру объектов и отношений между ними в фиксированный момент времени (модель классов). После этого можно уделить внимание изменениям объектов и их отношений с течением времени (модель состояний). Модель состояний описывает последовательности операций, происходящих в системе в ответ на внешние воздействия (в противоположность содержанию, предмету и реализации операций, описываемым моделью классов).

Модель состояний состоит из нескольких диаграмм состояний, по одной на каждый класс, поведение которого во времени важно для приложения. Диаграмма состояний — это стандартная концепция из информатики (графическое представление конечного автомата), связывающая события и состояния. События представляют внешние воздействия, а состояния — значения объектов.

Обязательно изучите материал этой главы, перед тем, как читать дальше.

5.1. События

Событие (event) — это происшествие, случившееся в определенный момент времени, например *нажатие пользователем левой кнопки мыши* или *вылет рейса 123 из Чикаго*. Часто события соответствуют глаголам в прошедшем времени (питание было включено, будильник был установлен) или выполнению некоторого условия (опустошился лоток для бумаги, температура опустилась ниже точки замерзания) в описании задачи. По определению событие происходит мгновенно, по крайней мере, во временном масштабе приложения. Разумеется, ничего мгновенного в реальном мире не бывает. Событие — это происшествие, которое рассматривается как атомарное и скоротечное. Неявным атрибутом события является момент его осуществления. Продолжительные изменения, осуществляющиеся в течение некоторого промежутка времени, хорошо описываются с помощью концепции состояния.

Одно событие может логически предшествовать другому или следовать за ним. События могут быть и несвязанными друг с другом. Рейс 123 должен вылететь из Чикаго прежде, чем он сможет прибыть в Сан-Франциско. Эти два события находятся в причинно-следственной связи. Однако рейс 123 может вылететь как перед вылетом рейса 456 из Рима, так и после него. Эти два события не связаны друг с другом. Несвязанные события называются *параллельными* (concurrent). Они никак не влияют друг на друга. Если временная задержка при передаче информации между двумя точками превышает временной интервал между событиями, эти события обязаны быть параллельными, поскольку они никак не могут повлиять друг на друга. Даже если физически события осуществляются не слишком далеко друг от друга, мы все равно будем считать их параллельными в том случае, если они не связаны между собой. При моделировании системы не следует задавать относительный порядок параллельных событий, потому что на практике они могут происходить в любом порядке.

К событиям относятся не только нормальные происшествия, но и ошибочные ситуации. В качестве примеров типичных ошибочных ситуаций можно привести *заклинивание двигателя, отмену транзакции и тайм-аут*. Ошибочная ситуация ничем не отличается от любого другого события. Только в нашей интерпретации она становится «ошибкой».

Термин «событие» часто используется в нескольких смыслах. В некоторых случаях он обозначает экземпляр, а в других случаях — класс. На практике эта двусмысленность обычно не создает проблем, так как точный смысл слова следует из контекста. При необходимости вы можете употреблять точные термины: «осуществление события» или «происшествие» (экземпляр события) и «тип события» (класс).

События бывают разных видов. Чаще всего встречаются события сигналов, события изменения и события времени.

5.1.1. Событие сигнала

Сигнал (signal) — это явная односторонняя передача информации от одного объекта другому. Сигнал отличается от вызова подпрограммы, который может возвращать значение. Объект, передающий сигнал другому объекту, может рассчитывать на получение ответа, но этот ответ будет отдельным сигналом, и его отправка (или задержка) будет целиком зависеть от второго объекта.

Событие сигнала (signal event) — это событие получения или отправки сигнала. Обычно более важным считается получение сигнала, потому что оно влияет на объект-получатель. Обратите внимание на разницу между сигналом и событием сигнала: сигнал — это сообщение между объектами, а событие сигнала — это происшествие.

Каждая передача сигнала является уникальным происшествием, но мы группируем их в классы сигналов и даем каждому классу имя, подчеркивая общую структуру и поведение. Например, *вылет рейса 123 Юнайтед Эйрлайнс из Чикаго 10 января 1991 года* — это экземпляр класса сигналов *ВылетРейса*. Некоторые сигналы являются обычными происшествиями, но большинство из них характеризуются атрибутами, в которых хранятся передаваемые этими сигналами значения.

Например, на рис. 5.1 класс *FlightDeparture* (*ВылетРейса*) обладает атрибутами *airline* (авиакомпания), *flightNumber* (номерРейса), *city* (город) и *date* (дата). В UML сигнал обозначается ключевым словом «*signal*» в угловых кавычках («»), которое ставится над именем класса сигнала в верхнем разделе прямоугольника. Во втором разделе указываются атрибуты сигнала.

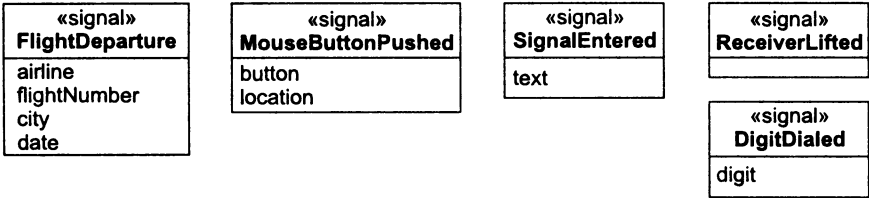


Рис. 5.1. Классы сигналов и их атрибуты

5.1.2. События изменения

Событие изменения (change event) — это событие, вызванное выполнением логического выражения. Суть события состоит в том, что некоторое выражение постоянно проверяется, и как только его значение изменяется с «ложно» на «истинно», осуществляется событие изменения. Разумеется, в реализации никакой непрерывной проверки осуществляться не будет, но она должна быть достаточно частой, чтобы в масштабе приложения казаться непрерывной.

В UML событие изменения обозначается ключевым словом *when*, за которым следует логическое выражение в круглых скобках. На рис. 5.2 приведено несколько примеров событий изменения.

- when (температура в комнате) < установка нагрева
- when (температура в комнате) > установка охлаждения
- when (заряд батареи < нижнее ограничение)
- when (давление в шинах < минимальное давление)

Рис. 5.2. События изменения

5.1.3. События времени

Событие времени (time event) — это событие, вызванное достижением момента абсолютного времени или истечением временного интервала. В UML момент абсолютного времени обозначается ключевым словом *when*, за которым следует временное выражение в круглых скобках. Временной интервал обозначается ключевым словом *after*, за которым следует выражение, результатом вычисления которого является временной интервал (рис. 5.3).

- when (дата = 1 января 2000 г.)
- after (10 секунд)

Рис. 5.3. События времени

5.2. Состояния

Состояние (state) — это абстракция значений и связей объекта. Множества значений и связей группируются в состояние в соответствии с массовым поведением объектов. Например, состояние банка может быть либо «платежеспособен», либо «банкрот», в зависимости от того, что больше: активы или обязательства. Состояния часто соответствуют отглагольным формам или деепричастиям (*Ожидает, Дозванивается*) или выполнению некоторого условия (*Включен, НижеТочкиЗамерзания*).

На рис. 5.4 показана система обозначений состояния в UML: прямоугольник со скругленными углами, в котором ставится необязательное название состояния. Мы выделяем название состояния полужирным шрифтом, центрируем его и пишем с заглавной буквы.

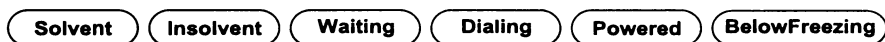


Рис. 5.4. Состояния

Определяя состояния, мы не учитываем атрибуты, не оказывающие влияния на поведение объекта, и объединяем вместе в одно состояние все комбинации значений и связей, характеризующиеся одинаковыми откликами на события. Разумеется, каждый атрибут должен влиять на поведение, иначе он не будет иметь никакого значения, однако достаточно часто некоторые атрибуты не влияют на последовательность управления. Их можно рассматривать просто как значения параметров состояния. Вспомните, что суть моделирования состоит в том, чтобы сосредоточиться на качествах, важных для решения задачи, и отбросить все неважные. Три модели UML (классов, состояний и взаимодействия) дают разные представления системы. В разных представлениях важными оказываются разные атрибуты и значения. Например, набранные на телефоне цифры (за исключением первых 0 или 8) не влияют на управление телефонной линией, поэтому их можно обобщить в состоянии *НаборНомера* и считать телефонный номер параметром. В некоторых случаях все возможные значения атрибута оказываются важными, но это бывает только в тех случаях, когда их количество невелико.

Объекты класса обладают конечным числом возможных состояний. В конкретный момент времени каждый объект может находиться ровно в одном состоянии. Объекты могут проходить через одно или несколько состояний в течение времени своего существования. В конкретный момент времени разные объекты класса могут охватывать широкий спектр состояний.

Состояние описывает отклик объекта на получаемые события. В конкретном состоянии игнорируются любые события, за исключением тех, поведение при получении которых описано явным образом. Отклик на событие может быть вызовом поведения или изменением состояния. Например, если в состоянии *Гудок* нажать кнопку с цифрой на телефоне, зуммер сбрасывается, а телефонная линия переходит в состояние *НаборНомера*. Если же в состоянии *Гудок* повесить трубку, линия отключается и переходит в состояние *Свободно*.

Между событиями и состояниями существует некоторая симметрия, которую мы показываем на рис. 5.5. События — это точки на линии времени, а состояния —

интервалы. Состояние соответствует интервалу между двумя точками, обозначающими два полученных объектом события. Например, между снятием трубки и нажатием первой цифры телефонная линия находится в состоянии *Зуммер*. Состояние объекта зависит от предыдущих событий, которые в большинстве случаев перекрываются последующими событиями. Например, события, произошедшие до того, как трубка была повешена, не влияют на будущее поведение. Состояние *Свободно* «забывает» о событиях, полученных до сигнала *повесить трубку*.



Рис. 5.5. События и состояния

И события и состояния зависят от уровня абстрагирования. Например, коммивояжер, планирующий свой маршрут, будет рассматривать каждый его сегмент как отдельное событие. Информатор в аэропорту будет объявлять о вылетах и прибытиях. Авиадиспетчерская служба разделит каждый перелет на множество географических отрезков.

Состояние можно характеризовать несколькими способами. Рисунок 5.6 демонстрирует это для состояния *Звонок будильника* для часов. Состояние обладает именем и описанием на естественном языке. Последовательность событий, которая приводит к этому состоянию, состоит из установки будильника, произвольных действий, не приводящих к его сбрасыванию, и наступления заданного момента времени. Условие состояния выражается в терминах параметров, таких как текущее и целевое время. Звонок прекращается после 20 секунд. Таблица событий и откликов показывает реакцию на события текущее время и нажатие кнопки. В этой таблице указывается не только действие, но и следующее состояние. Разные описания состояния могут перекрываться.

Состояние: <i>AlarmRinging</i>		
Описание: alarm on watch is ringing to indicate target time		
Событие, приводящее к данному состоянию:		
<i>setAlarm (targetTime)</i>		
any sequence not including <i>clearAlarm</i>		
<i>when (currentTime = targetTime)</i>		
Условие, характеризующее данное состояние:		
alarm = on, alarm set to <i>targetTime</i> , <i>targetTime</i> J <i>currentTime</i> J		
<i>targetTime</i> + 20 seconds, and no button has been pushed since <i>targetTime</i>		
События, возможные в данном состоянии:		
Событие	Отклик	Следующее состояние
<i>when (currentTime = targetTime + 20)</i>	<i>resetAlarm</i>	<i>normal</i>
<i>buttonPushed</i> (any button)	<i>resetAlarm</i>	<i>normal</i>

Рис. 5.6. Разные способы описать состояние

Могут ли связи обладать состояниями? Постольку, поскольку они могут считаться объектами. С практической точки зрения обычно оказывается достаточно задать состояния только для объектов.

5.3. Переходы и условия

Переход (transition) — это мгновенная смена одного состояния другим. Например, когда вы отвечаете на входящий звонок, телефонная линия переходит из состояния *Звонок* в состояние *Разговор*. Говорят, что переход *запускается* (fire) при смене исходного состояния целевым. Исходное и целевое состояния обычно отличаются друг от друга, но они могут и совпадать. Переход запускается, когда происходит связанное с ним событие (если только необязательное сторожевое условие не приводит к игнорированию события). Выбор целевого состояния зависит как от исходного состояния, так и от полученного события. Событие может вызвать переходы во множестве объектов. С концептуальной точки зрения эти переходы происходят одновременно.

Сторожевое условие (guard condition) — это логическое выражение, которое должно быть истинным, чтобы переход мог запуститься. Например, сигнал светофора на перекрестке может переключиться только в том случае, если на дороге имеются ожидающие этого машины. Переход со сторожевым условием запускается в тот момент, когда осуществляется соответствующее событие, но только если в этот же момент выполнено его сторожевое условие. Например, «когда будешь выходить утром из дома (событие), если на улице будет ниже нуля (условие) — надень перчатки (целевое состояние)». Сторожевое условие проверяется только один раз, в тот момент, когда осуществляется событие, и если условие выполняется — происходит переход. Обратите внимание, что сторожевое условие концептуально отличается от события: условие проверяется только один раз, тогда как наличие события, по сути, проверяется непрерывно.

На рис. 5.7 показаны переходы со сторожевыми условиями для светофоров на перекрестке. Одна пара фотоэлементов контролирует полосы в направлении север-юг, из которых возможен поворот налево. Другая пара контролирует полосы в направлении запад-восток, из которых тоже возможен поворот налево. Если на одной из пар полос отсутствуют машины, управляющая логика светофора пропускает часть цикла, разрешающую левый поворот.

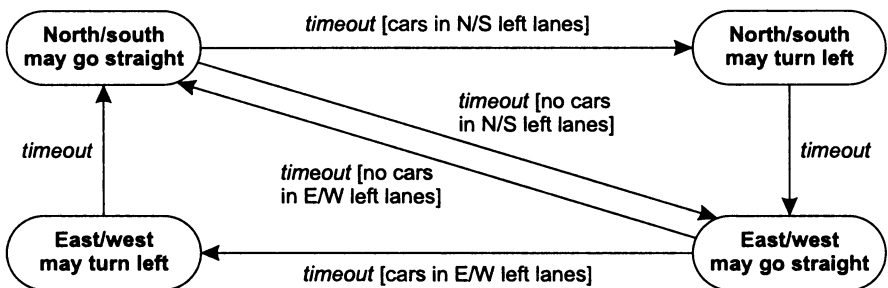


Рис. 5.7. Переходы со сторожевыми условиями

В UML для обозначения перехода используется линия, соединяющая исходное состояние с целевым. На одном из концов линии, указывающем на целевое состояние, ставится стрелка. Линия может состоять из нескольких сегментов. Событие может быть указано в качестве метки перехода. После события в квадратных скобках можно указать необязательное сторожевое условие. Мы обычно привязываем сегменты линий к прямоугольной сетке. Название события мы выделяем курсивом, а сторожевое условие записываем в квадратных скобках.

5.4. Диаграммы состояний

Диаграмма состояний — это граф, узлами которого являются состояния, а направленными дугами — переходы между состояниями. Диаграмма состояний описывает последовательности состояний, вызываемые последовательностями событий. Названия состояний должны быть уникальными в рамках диаграммы. Все объекты класса следуют диаграмме состояний, описывающей общее для них поведение. Диаграмма состояний может быть реализована непосредственной интерпретацией или преобразованием семантики в эквивалентный программный код.

Модель состояний состоит из множества диаграмм состояний, по одной на каждый класс, поведение которого с течением времени важно для приложения. Диаграммы состояний должны быть согласованы по интерфейсам (событиям и сторожевым условиям). Отдельные диаграммы взаимодействуют друг с другом посредством передачи событий, а также косвенно, через сторожевые условия. Некоторые события и сторожевые условия присутствуют только на одной диаграмме, тогда как другие — на нескольких. В этой главе мы рассматриваем только одиночные диаграммы состояний, а в главе 6 обсуждаются модели состояний, состоящие из множества взаимодействующих диаграмм.

Класс, имеющий несколько состояний, характеризуется важным поведением во времени. Если же класс обладает одним состоянием, его поведение во времени можно игнорировать. Диаграммы состояний с одним состоянием можно описать в простой форме без всякой графики, а именно в виде таблицы воздействий и откликов, в которой будут приводиться события и сторожевые условия, а также вызываемое ими поведение.

5.4.1. Пример диаграммы состояний

На рис. 5.8 показана диаграмма состояний для телефонной линии. Данная диаграмма относится именно к телефонной линии, а не к звонящему или вызываемому абоненту. На диаграмме приведены последовательности, описывающие нормальные звонки, а также некоторые ненормальные последовательности, например тайм-аут при наборе номера или перегрузка линий. Для обозначения диаграммы состояний в UML используется прямоугольник. Название диаграммы указывается в пятиугольном теге в левом верхнем углу. Внутри прямоугольника изображаются состояния и переходы, образующие диаграмму.

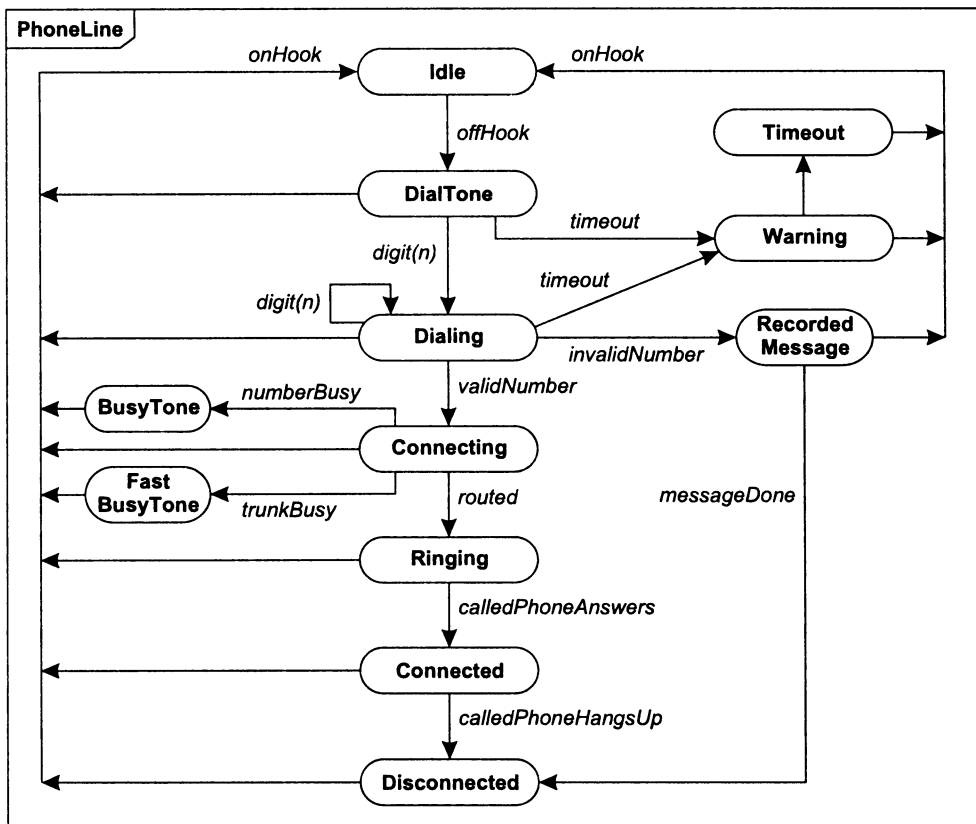


Рис. 5.8. Диаграмма состояний телефонной линии

Перед началом вызова телефонная линия находится в отключенном состоянии (состоянии ожидания). Когда трубка снимается с рычага, она издает сигнал ответа станции (гудок). После этого можно набирать цифры. При вводе корректного номера система пытается выполнить соединение и направить его к нужному адресату. Соединение может оказаться невозможным, если абонент или его узел заняты. Если же соединение оказывается успешным, телефон вызываемого абонента начинает звонить. Если абонент отвечает на звонок, оба абонента могут осуществить разговор. Когда один из абонентов вешает трубку, линия разъединяется и снова возвращается в состояние ожидания.

Обратите внимание, что получение сигнала *onHook* (наРычаг) вызывает переход в состояние *Idle* (Ожидание) из любого другого состояния. В главе 6 мы продемонстрируем более универсальную систему обозначений, представляющую события, применимые к группам состояний, единым переходом.

Состояния не определяют все значения объекта полностью. Например, состояние *Dialing* (НаборНомера) включает все последовательности неполных телефонных номеров. Не обязательно рассматривать отдельные номера как разные

состояния, потому что они характеризуются одинаковым поведением. Реальный набранный номер следует хранить как атрибут.

Если из состояния выходит несколько переходов, первое осуществившееся событие запускает соответствующий ему переход. Если происходит событие, которому не сопоставлен ни один переход, оно просто игнорируется. Если событию соответствует несколько переходов, выбран будет только один из них, и притом случайным образом.

5.4.2. Одноразовые диаграммы состояний

Диаграммы состояний могут описывать непрерывные циклы или одноразовые жизненные циклы. Диаграмма состояний телефонной линии является непрерывным циклом. Описывая обычные режимы использования телефона, мы не интересуемся тем, каким образом цикл был запущен. (Если бы мы описывали установку новой линии, начальное состояние было бы для нас важно.)

Одноразовые диаграммы состояний описывают объекты с конечным сроком существования. Такие диаграммы имеют начальное и конечное состояния. Сразу после создания объект оказывается в начальном состоянии. Вход в конечное состояние означает уничтожение объекта. На рис. 5.9 показан упрощенный жизненный цикл игры в шахматы с начальным состоянием по умолчанию (сплошной кружок) и конечным состоянием по умолчанию («бычий глаз»).

Начальное и конечное состояние можно обозначать точками входа и выхода. На рис. 5.10 точка входа *Start* соединена с первым ходом белых, и игра заканчивается одним из трех возможных состояний. Точки входа (пустые кружки) и выхода (кружки с символом *X*) ставятся на периметре диаграммы состояний и могут иметь имена.

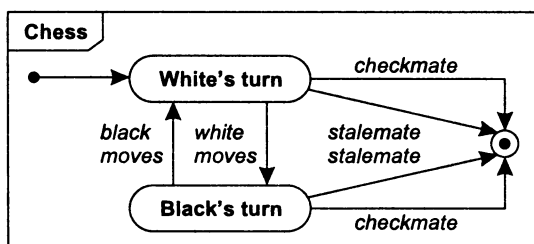


Рис. 5.9. Диаграмма состояний игры в шахматы

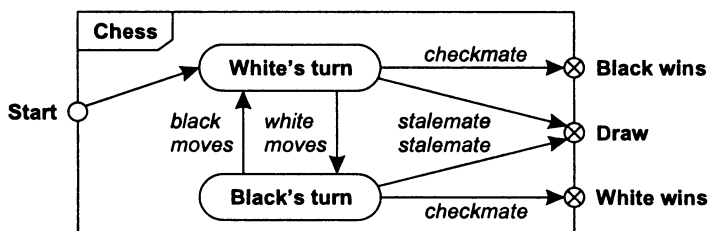


Рис. 5.10. Диаграмма состояний с точками входа и выхода

5.4.3. Основные обозначения для диаграмм состояний

Рисунок 5.11 демонстрирует основные обозначения, используемые на диаграммах состояний UML.

- **Состояние (state)**. Обозначается прямоугольником со скругленными углами, в котором может быть указано имя состояния. Для начальных и конечных состояний имеются специальные обозначения (сплошной кружок и «бычий глаз» или перечеркнутый кружок соответственно).
- **Переход (transition)**. Изображается линией, соединяющей исходное состояние с целевым. Стрелка ставится около конца линии, указывающего на целевое состояние. Линия может состоять из нескольких прямолинейных сегментов.
- **Событие (event)**. Событие сигнала изображается меткой на переходе. После названия события в круглых скобках можно указать атрибуты. Событие изменения обозначается ключевым словом *when*, после которого в круглых скобках указывается логическое выражение. Событие времени также обозначается ключевым словом *when*, после которого в круглых скобках указывается временное выражение, или ключевым словом *after*, после которого в круглых скобках указывается интервал времени).
- **Диаграмма состояний (state diagram)**. Заключается в прямоугольную рамку. Название диаграммы указывается в небольшом пятиугольном теге в верхнем левом углу рамки.
- **Сторожевое условие (guard condition)**. Может быть указано в квадратных скобках после события.
- **Действия (effects, см. следующий раздел)**. Могут прикрепляться к переходу или состоянию. Указываются после символа косой черты (/). Действия, если их несколько, отделяются друг от друга запятыми и выполняются параллельно. (Если нужно выполнять действия последовательно, их можно разделить промежуточными состояниями.)

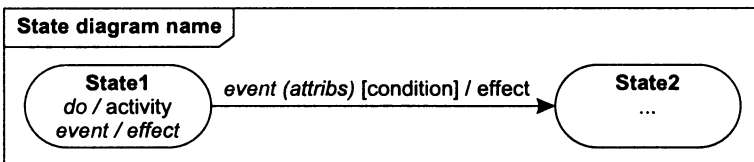


Рис. 5.11. Основные обозначения для диаграмм состояний

Мы выделяем название состояния полужирным шрифтом и делаем первую букву заглавной. Названия событий мы пишем курсивом, а первую букву заглавной не делаем. Сторожевые условия и действия не выделяются шрифтом и тоже пишутся со строчной буквы. Сегменты линий переходов мы стараемся привязывать к прямоугольной сетке.

5.5. Поведение на диаграммах состояний

Диаграммы состояний были бы не слишком полезны, если бы они описывали только события. Полное описание объекта должно указывать, что именно делает объект в ответ на события.

5.5.1. Действия и деятельность

Действие (effect) — это ссылка на поведение, выполняемое в ответ на произошедшее событие. *Деятельность* (activity) — это фактическое поведение, которое может вызываться любым количеством действий. Например, деятельность *disconnectPhoneLine* (разъединитьЛинию) может выполняться в ответ на событие *onHook* (наРычаг) на рис. 5.8. Деятельность может выполняться при переходе, при входе в состояние или при выходе из него, а также при наступлении какого-либо иного события в состоянии.

Деятельность может описывать внутренние управляющие операции, например установку атрибутов или порождение других событий. Эта деятельность не имеет аналогов в реальном мире и предназначена для структурирования управления при реализации. Например, программа может увеличивать внутренний счетчик на единицу каждый раз при осуществлении какого-либо события.

Деятельность обозначается косой чертой (/), после которой ставится название или описание деятельности. Деятельность указывается после вызывающего ее события. Ключевое слово *do* используется для обозначения текущей деятельности (см. далее) и не может использоваться в качестве имени события. На рис. 5.12 показана диаграмма состояний для всплывающего меню рабочей станции. При нажатии правой кнопки мыши меню отображается на экране. Когда пользователь отпускает эту кнопку, меню исчезает. Пока меню отображается на экране, в нем подсвечивается один элемент, над которым в данный момент находится указатель мыши.

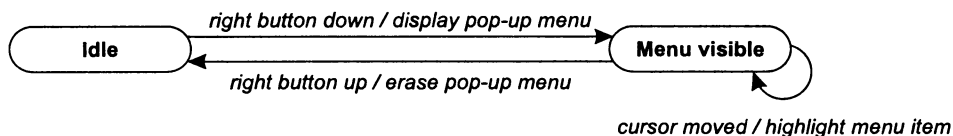


Рис. 5.12. Деятельность для всплывающего меню

5.5.2. Текущая деятельность

Текущей (do activity) называется деятельность, занимающая некоторый промежуток времени. По определению такая деятельность может выполняться только в некотором состоянии и не может прикрепляться к переходу. Например, индикатор аварии у ксерокса может мигать в состоянии *Затор бумаги* (рис. 5.13). Текущая деятельность включает непрерывные операции, такие как отображение картинки на телеэкране, а также последовательные операции, завершающиеся по прошествии некоторого промежутка времени (например, закрытие клапана).

Paper jam
do / flash warning light

Рис. 5.13. Текущая деятельность для ксерокса

Для обозначения текущей деятельности используется ключевое слово *do* и символ косой черты. Текущая деятельность может быть прервана событием, полученным в процессе выполнения этой деятельности. Это событие может вызвать переход из состояния, в котором осуществляется текущая деятельность, но может и не вызвать такого перехода. Например, робот, перемещающий деталь, может столкнуться с сопротивлением, что приведет к остановке его движения.

5.5.3. Деятельность при входе и при выходе

Деятельность может быть прикреплена не только к переходу, но и ко входу в состояние или к выходу из него. Никаких отличий в возможностях двух систем обозначений нет, однако часто при всех переходах в одно и то же состояние выполняется одинаковая деятельность, которую, в таком случае, удобнее привязать к самому событию.

На рис. 5.14 показана схема управления открыванием гаражной двери. Пользователь порождает событие *depress* (нажатие) нажатием кнопки, посредством которой он закрывает и открывает дверь. Каждое событие изменяет направление движения двери на противоположное, однако для обеспечения безопасности дверь должна быть открыта полностью, прежде чем ее можно будет снова закрыть. Система управления включает деятельность *motor up* (двигатель вверх) и *motor down* (двигатель вниз) для двигателя. Двигатель порождает события *door open* (дверь открыта) и *door closed* (дверь закрыта), когда перемещение двери завершается. Оба перехода в состояние *Opening* (Открывание) приводят к открытию двери.

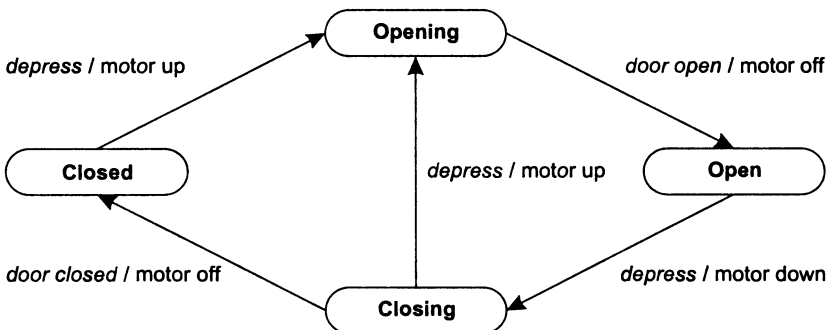


Рис. 5.14. Деятельность при переходах

На рис. 5.15 показана та же самая модель, в которой деятельность прикреплена к входу в состояние. Деятельность при входе изображается внутри значка состояния с ключевым словом *entry* и символом /. Такая деятельность выполняется при входе в состояние по любому из входящих переходов. Деятельность

при входе эквивалентна прикреплению той же деятельности к каждому из входящих переходов. Если входящий переход имеет свою собственную деятельность, она выполняется в первую очередь.

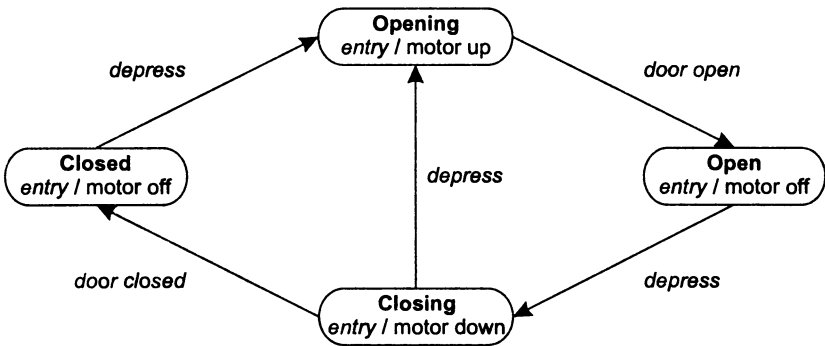


Рис. 5.15. Деятельность при входе

Деятельность при выходе используется реже, чем при входе, но иногда тоже оказывается полезной. Деятельность при выходе указывается внутри символа состояния после ключевого слова *exit* и символа */*. Эта деятельность выполняется в первую очередь при выходе из состояния по любому из исходящих переходов.

Если к состоянию привязано несколько видов деятельности, они выполняются в следующем порядке: деятельность при входящем переходе, деятельность при входе, текущая деятельность, деятельность при выходе, деятельность при исходящем переходе. События, вызывающие исходящие из состояния переходы, могут прерывать текущую деятельность. Если текущая деятельность прерывается, деятельность при выходе все равно должна быть выполнена до начала перехода.

В принципе, в состоянии может произойти любое событие. *Entry* и *exit* — частные случаи возможных событий. Как показано на рис. 5.16, существует отличие между событием в состоянии и переходом в себя: деятельность при входе и при выходе выполняется только в том случае, если происходит переход в себя.

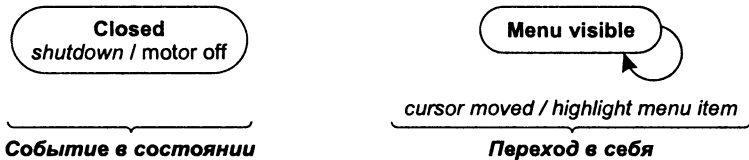


Рис. 5.16. Событие в состоянии и переход в себя

5.5.4. Переход по завершении

Часто единственным назначением состояния является последовательное выполнение некоторой деятельности. Как только деятельность завершается, запускается переход в следующее состояние. Стрелка без названия события обозначает автоматический переход, который запускается, как только завершается деятельность, свя-

званная с исходным состоянием. Такой переход называется переходом по завершении, потому что он переключается завершением деятельности в исходном состоянии.

Сторожевое условие проверяется только один раз, в тот момент, когда осуществляется событие. Если переход имеет переход по завершении, а его сторожевое условие не выполнено, состояние остается активным и может привести к «застреванию» управления: событие завершения деятельности не может произойти второй раз. Если состояние имеет переходы по завершении, сторожевые условия должны охватывать весь спектр возможных событий. Специальное условие *else* позволяет задать ситуацию, когда все остальные условия оказываются ложными. Не используйте сторожевые условия на переходы по завершении для ожидания изменения значения. Лучше ожидать события изменения.

5.5.5. Отправка сигналов

Объект может выполнять деятельность, заключающуюся в отправке сигнала другому объекту. Система объектов взаимодействует, обмениваясь сигналами.

Деятельность *send target.S(attributes)* передает сигнал *S* с заданными атрибутами целевому объекту (или группе объектов) *target*. Например, телефонная линия передает коммутатору сигнал *connect(phone number)* (соединить(номер телефона)), когда пользователь набирает на своем аппарате полный номер телефона. Сигнал может быть адресован множеству объектов или одному объекту. В первом случае все объекты получают по копии сигнала одновременно, и каждый из них обрабатывает свою копию независимо. Если же сигнал всегда адресуется одному и тому же объекту, на диаграмме его можно не указывать (но при реализации это сделать все равно придется).

Если объект может получать сигналы от нескольких объектов, порядок получения параллельных сигналов может повлиять на итоговое состояние. Это называется *ситуацией гонок* (race condition). Например, в модели на рис. 5.15 дверь может как остаться открытой, так и закрыться, если дверь будет нажата почти в тот же момент, когда дверь полностью откроется. Ситуация гонок не обязательно бывает ошибкой проектирования, но в параллельных системах часто возникают нежелательные ситуации гонок, которых следует избегать аккуратным проектированием. Требование одновременного получения двух сигналов никогда не может быть выполнено в реальном мире, потому что небольшие отклонения скорости передачи неотъемлемо присущи распределенным системам.

5.5.6. Пример диаграммы состояний с деятельностью

На рис. 5.17 изображена диаграмма состояний с рис. 5.8, на которую была добавлена деятельность.

5.6. Практические рекомендации

Содержимое любой модели всегда зависит от потребностей приложения. Все перечисленные ниже советы уже упоминались в тексте главы, но мы решили повторить их здесь еще раз для вашего удобства.

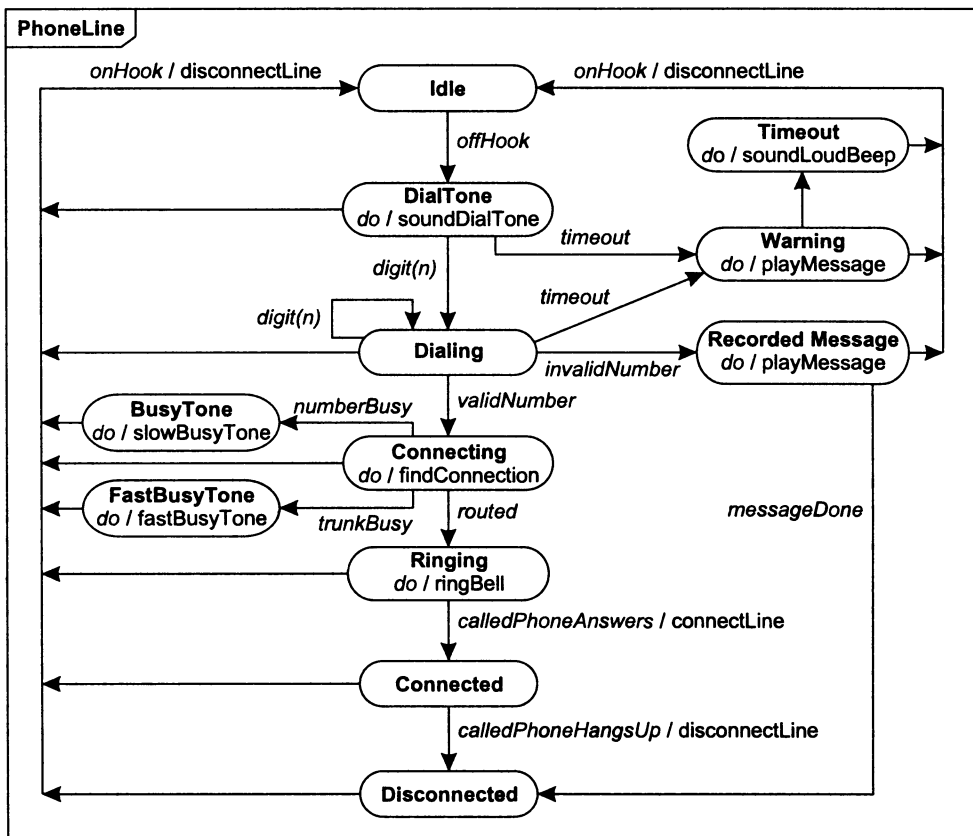


Рис. 5.17. Диаграмма состояний с указанием деятельности

- **Абстрагируйте значения до состояний.** При определении состояния учитывайте только важные атрибуты. На диаграммах состояний не обязательно использовать все атрибуты, показанные в модели классов (раздел 5.2).
- **Параметры.** Параметризируйте события вспомогательными данными, которые не влияют на поток управления (раздел 5.2).
- **Уровень детализации событий и состояний.** Выбирая уровень детализации событий и состояний в модели, учитывайте в первую очередь потребности приложения (раздел 5.2).
- **Когда использовать диаграммы состояний?** Диаграммы следует строить только для классов, поведение которых изменяется с течением времени. Так можно сказать о классе, который по-разному реагирует на разные события или обладает несколькими состояниями. Не для всех классов требуются диаграммы состояний (раздел 5.4).
- **Деятельность при входе и при выходе.** Если все входящие переходы приводят к выполнению одной и той же деятельности, используйте деятель-

ность при входе в состояние, вместо того чтобы указывать одну и ту же деятельность для каждого из входящих переходов. То же касается деятельности при выходе (раздел 5.5.3).

- **Сторожевые условия.** Аккуратно формулируйте сторожевые условия, чтобы объект не застрял в каком-нибудь состоянии навсегда (раздел 5.5.4).
- **Ситуации гонок.** Опасайтесь нежелательных ситуаций гонок на диаграммах состояний. Ситуации гонок возможны в том случае, когда состояние может получать сигналы (события) от нескольких объектов (раздел 5.5.5).

5.7. Резюме

Событие и состояние — две основополагающие концепции моделирования состояний. Событие — это происшествие, связанное с конкретным моментом времени. Состояние — это абстракция значений и связей объекта. События соответствуют точкам на временной оси, а состояния — промежуткам между ними. Объект может реагировать на некоторые события, находясь в некоторых состояниях. В конкретном состоянии игнорируются все события, за исключением тех, поведение для которых прописано явно. Одно и то же событие в разных состояниях может вызывать разные действия.

Существует несколько видов событий: событие сигнала, событие изменения и событие времени. Событие сигнала состоит в отправке или получении информации, передаваемой между объектами. Событие изменения вызывается выполнением логического выражения. Временное событие вызывается достижением момента абсолютного времени или истечением интервала относительного времени.

Переход — это мгновенная смена одного состояния на другое. Переход вызывается осуществлением события. Игнорированием событий управляют необязательные сторожевые условия. Сторожевое условие — это логическое выражение, которое должно быть выполнено, чтобы переход мог быть осуществлен.

Действие — это ссылка на поведение, которое выполняется объектами в ответ на получение события. Деятельность — это фактическое поведение, которое может вызываться различными действиями. Деятельность может выполняться при переходе или при осуществлении события внутри состояния. Текущая деятельность — это прерываемая деятельность, которая продолжается в течение некоторого времени. Поэтому текущая деятельность может осуществляться только в некотором состоянии и не может быть прикреплена к переходу.

Диаграмма состояний — это граф, узлами которого являются состояния, а направленными дугами — переходы между состояниями. Диаграмма состояний описывает возможные состояния, разрешенные переходы между состояниями, события, вызывающие запуск переходов, и выполняемую в ответ на события деятельность. Диаграмма состояний описывает поведение, общее для всех объектов некоторого класса. Каждый объект имеет свои собственные значения и связи, а потому обладает своим собственным состоянием (положением на диаграмме состояний). Модель состояний состоит из множества диаграмм состояний, по одной диаграмме на каждый класс, поведение которого во времени важно для

приложения. Диаграммы состояний согласовываются друг с другом по интерфейсам (событиям и сторожевым условиям).

Таблица 5.1. Ключевые понятия главы

деятельность	текущая деятельность	ситуация гонок	модель состояний
событие изменения	действие	сигнал	событие времени
переход по завершении	событие	событие сигнала	переход
параллелизм	запустить фактически	состояние	
управление	сторожевое условие	диаграмма состояний	

Библиографические замечания

Подробное сравнение различных методов спецификации программного обеспечения, а в том числе и динамического поведения систем приводится в [Wieringa-98].

Конечные автоматы — это базовая концепция информатики, они описываются в любом учебнике по теории автоматов, например в [Hopcroft-01]. Часто они описываются как распознаватели или генераторы формальных языков. Обычные конечные автоматы ограничены в возможностях выражения сложных концепций. Они были расширены добавлением локальных переменных и рекурсии и получили название «расширенная сеть переходов» или «рекурсивная сеть переходов» [Woods-70]. Эти расширения позволили выразить при помощи конечных автоматов более широкий диапазон формальных языков, но не решили проблему комбинаторного взрыва, мешающую их использованию в практических целях (см. главу 6).

Традиционные конечные автоматы рассматривались и с точки зрения синхронизации. Сети Петри [Reisig-92] формализуют параллельность и синхронизацию в системах с распределенной деятельностью без привязки к глобальному времени. Они оказались удачными в качестве абстрактных моделей, но для описания больших систем их уровень оказался недостаточно высоким.

Необходимость описания интерактивных пользовательских интерфейсов привела к созданию нескольких методик спецификации управления. Эта работа направлена на поиск систем обозначения, которые ясно выражали бы мощные средства взаимодействия и при этом были бы простыми в реализации. Сравнение нескольких методик приводится в [Green-86].

В первом издании этой книги мы проводили разделение между действиями (мгновенным поведением) и деятельностью (длительным поведением). UML2 переопределяет оба термина, и нам тоже пришлось изменить изложение. UML 2 определяет деятельность как спецификацию выполняемого поведения, а действие — как предопределенную примитивную деятельность. По сути дела, новое определение деятельности включает в себя старые определения действия и деятельности.

Ссылки

[Green-86] Mark Green. A survey of three dialogue models. *ACM Transactions on Graphics* 5, 3 (July 1986), 244–275.

[Hopcroft-01] J. E. Hopcroft, Rejeev Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation.*, Second Edition, Boston: Addison-Wesley, 2001.

[Reisig-92] Wolfgang Reisig. *A Primer in Petri Net Design*. New York: Springer-Verlag, 1992.

[Wieringa-98] Roel Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys* 30, 4 (December 1998), 459–527.

[Woods-70] W. A. Woods. Transition network grammars for natural language analysis. *Communications of ACM* 13, 10 (October 1970), 591–606.

Упражнения

5.1. (6) Раздвижная лестница снабжена веревкой, шкивом и защелкой для подъема, опускания и фиксирования. Когда защелка зафиксирована, можно смело лезть по лестнице вверх. Чтобы отцепить защелку, нужно слегка поднять раздвижной сегмент веревкой. После этого его можно свободно поднимать или опускать. Защелка щелкает при прохождении по ступеням лестницы. Защелка может быть зафиксирована при подъеме лестницы изменением направления сразу после того, как она прошла очередную ступень. Нарисуйте диаграмму состояний раздвижной лестницы.

5.2. (4) Простейшие цифровые часы состоят из дисплея и двух кнопок А и В. Часы могут работать в двух режимах: отображения и настройки. В режиме отображения часы показывают часы и минуты, между которыми мигает символ двоеточия.

Режим настройки состоит из двух подрежимов: настройка часов и настройка минут. Кнопка А позволяет выбрать режим. Каждый раз при ее нажатии происходит переход к очередному режиму в последовательности: отображение, установка часов, установка минут, отображение и т. д. Кнопка В позволяет увеличивать значение часов или минут на единицу при каждом нажатии в одном из режимов установки. Чтобы кнопка смогла породить новое событие, ее необходимо отпустить. Нарисуйте диаграмму состояний часов.

5.3. (4) На рис. У5.1 изображена неполная и упрощенная диаграмма состояний телефонного автоответчика. Автоответчик детектирует входящий звонок по первому же сигналу и отвечает заранее записанным сообщением. Когда сообщение завершается, автоответчик записывает сообщение звонящего. Когда звонящий вешает трубку, автоответчик тоже вешает трубку и отключается до следующего звонка. Добавьте на диаграмму следующие надписи: *call detected* (обнаружен звонок), *answer call* (ответ на звонок), *play announcement* (воспроизвести приветствие), *record message* (записать послание), *caller hangs up* (звонящий вешает трубку), *announcement complete* (приветствие закончилось).

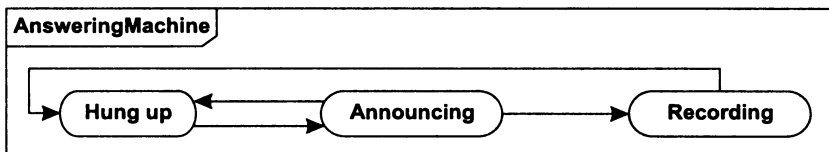


Рис. У5.1. Диаграмма состояний автоответчика

- 5.4. (7) Автоответчик из предыдущего примера срабатывал по первому же звонку. Измените диаграмму состояний таким образом, чтобы он срабатывал по пятому звонку. Если кто-нибудь подойдет к телефону до пятого звонка, автоответчик не должен делать ничего. Следите за тем, чтобы не перепутать пять вызовов, когда человек снимает трубку с первого звонка, и пять звонков от одного вызова.
- 5.5. (3) В персональном компьютере контроллер диска обычно передает поток байтов с дисководом в буфер в памяти с помощью ведущего узла (центрального процессора или контроллера прямого доступа к памяти DMA). На рис. У5.2 показана частичная упрощенная диаграмма состояний управления передачей данных.

Контроллер передает ведущему узлу сигнал о каждом новом доступном байте. Данные должны быть считаны и сохранены для того, чтобы контроллер мог перейти к следующему байту. Когда контроллер обнаруживает, что данные были считаны, он сообщает об отсутствии данных до тех пор, пока не подготовит следующий байт. Если байт не будет считан до того, как контроллер подготовит следующий, контроллер выдает сигнал потери данных до тех пор, пока не получит сигнал сброса. Добавьте на диаграмму надписи: *reset* (сброс), *indicate data is not available* (индикация отсутствия данных), *indicate data available* (индикация наличия данных), *data read by host* (данные считаны ведущим узлом), *new data ready* (готовы новые данные), *indicate data lost* (индикация потери данных).

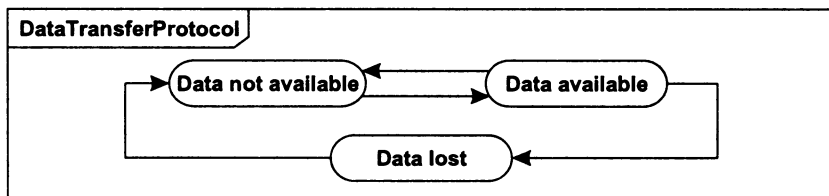


Рис. У5.2. Неполная диаграмма состояний протокола передачи данных

- 5.6. (5) На рис. У5.3 изображена неполная диаграмма состояний системы управления двигателем, которая часто применяется в бытовой технике. Независимая система управления всего устройства определяет, когда двигатель должен быть включен, и непрерывно подает сигнал ВКЛ на управляющий вход двигателя.

Когда на вход подается сигнал ВКЛ, система управления двигателя должна запустить двигатель и поддерживать его работу. Двигатель запускается по-

дачей напряжения на пусковую и рабочую обмотки. Датчик, называемый стартовым реле, определяет момент запуска двигателя, после чего отключает пусковую обмотку. Напряжение остается только на рабочей обмотке. Когда сигнал ВКЛ пропадает, обе обмотки отключаются.

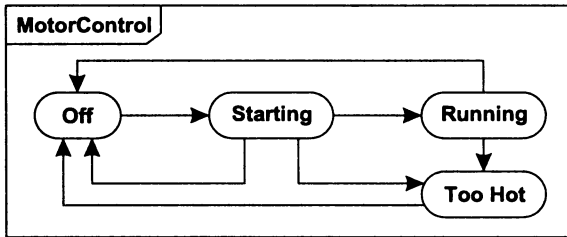


Рис. У5.3. Неполная диаграмма состояний системы управления двигателем

Электродвигатели, применяемые в бытовой технике, могут перегреваться из-за чрезмерной нагрузки или невозможности запуска. Для защиты от перегрева в систему управления двигателем часто добавляется датчик превышения температуры. Если двигатель нагревается слишком сильно, система управления снимает напряжение с обеих обмоток и игнорирует сигнал ВКЛ до тех пор, пока двигатель не остынет и не будет нажата клавиша сброса.

Добавьте на диаграмму следующие элементы. Деятельность: подать напряжение на пусковую обмотку, подать напряжение на рабочую обмотку. События: двигатель перегрелся, подан сигнал ВКЛ, снят сигнал ВКЛ, двигатель работает, сброс. Условие: двигатель не перегрет.

- 5.7. (6) Система управления в упражнении 5.6 имела один непрерывно активный вход. В другой широко используемой системе двигатель управляется двумя кнопками: *пуск* и *стоп*. Для запуска двигателя пользователь нажимает кнопку *пуск*. Двигатель продолжает вращаться и после ее отпускания. Для остановки двигателя пользователь нажимает кнопку *стоп*. Кнопка *стоп* имеет приоритет перед кнопкой *пуск*, поэтому при нажатии обеих кнопок двигатель не вращается.

Если обе кнопки нажать и отпустить одновременно, результат будет зависеть от того, в каком порядке кнопки будут отпущены. Если кнопку *стоп* отпустить первой, двигатель запускается. В противном случае он не запускается. Измените диаграмму состояний из упражнения 5.6, добавив на нее кнопки *пуск* и *стоп*.

- 5.8. (5) Подготовьте диаграмму состояний для выделения и перетаскивания объектов при помощи редактора диаграмм из упражнения 4.2.

Курсор управляется двухкнопочной мышью. При нажатии левой кнопки в тот момент, когда курсор находится над объектом (прямоугольником или линией), объект выделяется (при этом выделение снимается с любого ранее выделенного объекта). Если левая кнопка нажимается в тот момент, когда курсор не находится над объектом, выделение снимается со всех ранее выделенных объектов. Перемещение мыши с нажатой левой кнопкой приводит к перетаскиванию выделенного объекта.

- 5.9. (6) Расширьте диаграмму состояний из упражнения 5.8. Если пользователь щелкает левой кнопкой мыши на объекте при нажатой клавише **SHIFT**, объект добавляется к множеству выделенных объектов. Перемещение мыши при нажатой левой кнопке приводит к перетаскиванию всех выделенных объектов.
- 5.10. (5) На рис. У5.4 приведена диаграмма состояний копировального аппарата. В начальном состоянии копировальный аппарат выключен. Включение питания переводит аппарат в основное состояние: одна копия, автоматическая настройка контраста, нормальный размер. В процессе прогрева аппарат мигает индикатором готовности. Когда самопроверка автомата завершается, индикатор готовности перестает мигать и начинает гореть непрерывно. После этого автомат считается готовым к работе.

Оператор может изменить любой параметр, пока автомат находится в режиме готовности к работе. Оператор может увеличить или уменьшить количество копий, их размер, переключаться между автоматической и ручной настройкой контраста, а также изменять контраст в том случае, если выбран ручной режим его настройки. После установки нужных значений параметров оператор может нажать кнопку ПУСК, после чего аппарат начнет копирование. Обычно копирование продолжается до тех пор, пока не будет сделано необходимое количество копий. Исключительные ситуации связаны с застреванием бумаги или ее отсутствием. Когда бумага застревает в аппарате, оператор должен удалить затор, и аппарат сможет продолжить копирование. Добавление новой бумаги позволяет аппарату продолжить работу после остановки из-за отсутствия бумаги.

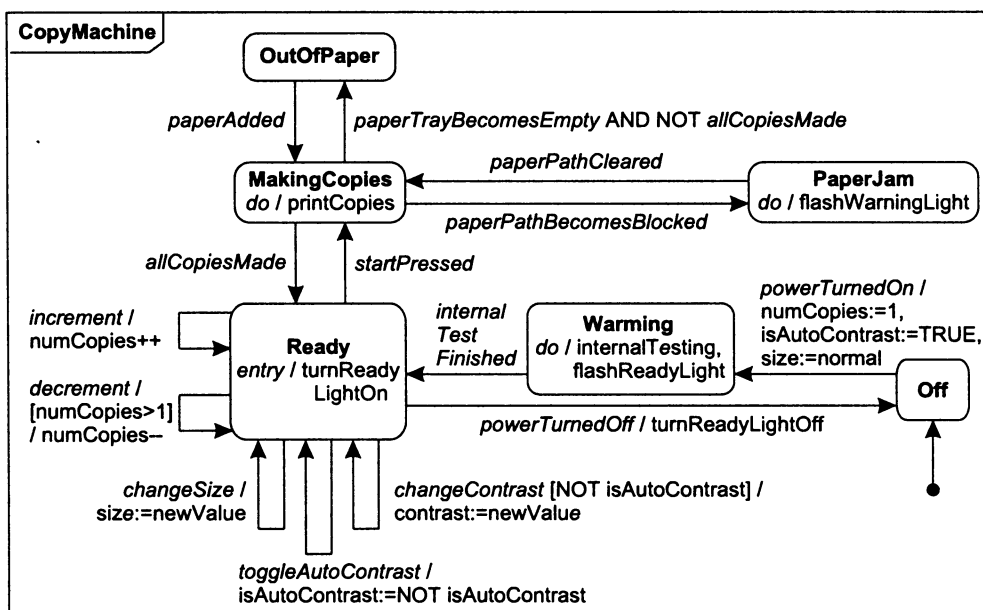


Рис. У5.4. Диаграмма состояний копировального аппарата

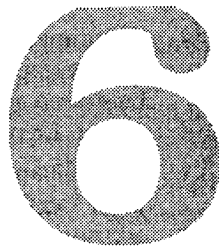
Расширьте диаграмму, добавив на нее следующие моменты. Копировальный аппарат работает не совсем так, как положено: после удаления затора оператор должен выключить аппарат и снова включить его, чтобы он заработал. Если аппарат отключается без удаления затора, затор остается.

- 5.11. (7) Обшаривая старый замок, вы с другом нашли книжный шкаф, за которым, судя по всему, открывается вход в секретный коридор. Пока вы осматривали шкаф, ваш друг вынул свечку из подсвечника и обнаружил, что этот подсвечник управлял входом. Шкаф повернулся на половину оборота, подтолкнув вас и отделив вас от друга. Друг вставил свечу обратно. Шкаф повернулся на полный оборот, но вы остались за ним.

Друг вынул свечу. Шкаф опять начал делать полный поворот, но вы не дали ему закончить оборот, оперевшись в него плечом. Друг передал вам свечу, и вместе вам удалось провернуть шкаф назад на пол-оборота. В результате друг остался за шкафом, а вы перед ним. Вы вставили свечу в подсвечник. Шкаф начал поворачиваться, вы вынули свечу и шкаф остановился на четверти оборота. Вы вместе вошли в коридор и отправились на поиски приключений.

Подготовьте диаграмму состояний системы управления книжным шкафом, которая описывала бы приведенный выше пример. Что нужно было сделать с самого начала, чтобы войти в коридор без всяких проблем?

Углубленное моделирование состояний



Обычных диаграмм состояний достаточно для описания простых систем, но для сложных задач нужны средства помощнее. Для моделирования сложных систем можно использовать вложенные диаграммы состояний, вложенные состояния, обобщение сигналов и параллелизм.

Эта глава содержит сложный материал, который при первом чтении можно пропустить.

6.1. Вложенные диаграммы состояний

6.1.1. Задачи с одноуровневыми диаграммами состояний

Диаграммы состояний часто критикуются, потому что они якобы непрактичны для больших задач. Это верно для плоских, бесструктурных диаграмм состояний. Рассмотрим объект, имеющий n логических атрибутов, влияющих на управление. Представление такого объекта на одной плоской диаграмме потребовало бы 2^n состояний. При разбиении состояния на n независимых диаграмм состояний потребовалось бы всего 2^n состояний.

Другой пример — диаграмма с рис. 6.1, на которой n^2 переходов соединяют все состояния попарно. Если переформулировать эту модель с использованием структуры, количество переходов можно снизить вплоть до n . Сложные системы обычно содержат много избыточной информации, которую можно исключить при помощи структурирования.

6.1.2. Разложение состояний

Одним из способов организации модели является применение диаграммы высокого уровня, отдельные состояния которой раскрываются на поддиаграммах. Принцип аналогичен подстановке макросов в языках программирования.

На рис. 6.2 показана такая диаграмма для торгового автомата. Изначально автомат находится в бездействии. Когда человек кидает монету в автомат, тот добавляет ее цену к общему балансу. Положив несколько монет, человек может выбрать товар. Если данный товар кончился или денег для него недостаточно, автомат ждет, пока человек выберет что-нибудь другое. В противном случае автомат выдает товар и возвращает остаток с баланса.

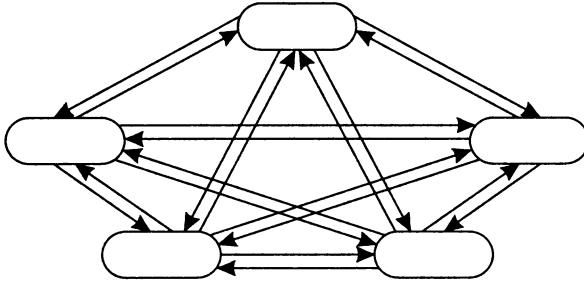


Рис. 6.1. Комбинаторный взрыв переходов на плоских диаграммах состояний

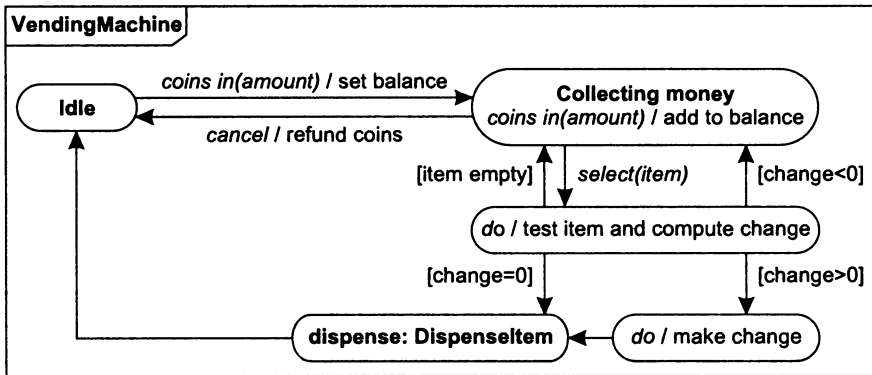


Рис. 6.2. Диаграмма состояний торгового автомата

На рис. 6.3 раскрывается состояние *Dispense* (Выдача). Вложенная диаграмма состояний называется *submachine* (вложенный конечный автомат). Вложенный автомат может вызываться как часть другого конечного автомата. Для вызова вложенного автомата на диаграмме указывается название состояния, после которого ставится двоеточие и название вложенного автомата. С концептуальной точки зрения вложенная диаграмма заменяет состояние внешней диаграммы. Фактически, вложенный конечный автомат — это нечто вроде подпрограммы для диаграммы состояний.

6.2. Вложенные состояния

Состояние можно структурировать, а не только заменять его вложенной диаграммой состояний. Вложение состояний может использоваться для выделения их общих черт и совместного использования поведения. (В соответствии с UML2

мы не применяем к состояниям термин *обобщение*. Подробнее см. раздел *Библиографические замечания* в этой главе.)

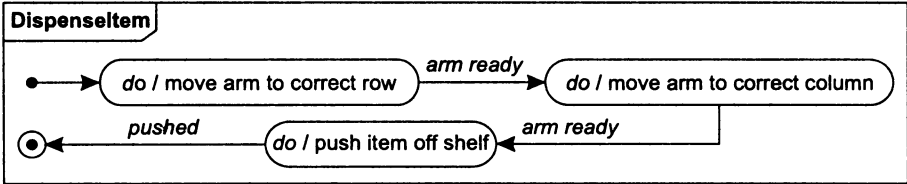


Рис. 6.3. Вложенный конечный автомат торгового автомата

На рис. 6.4 представлена упрощенная модель телефонной линии из главы 5. Один переход из *Active* (Активна) в *Idle* (Отключена) заменяет переходы из всех состояний в состояние *Idle*. Все состояния из главы 5, за исключением *Idle*, теперь становятся вложенными состояниями состояния *Active*. Осуществление события *onHook* (наРычаг) в любом из вложенных состояний вызывает переход в *Idle*.

Название *составного* или *композиционного состояния* (composite state) относится ко внешнему контуру, который охватывает все вложенные состояния. Таким образом, *Active* оказывается композиционным состоянием со вложенными состояниями *DialTone* (Гудок), *Timeout* (Тайм-аут), *Dialing* (НаборНомера) и т. д. Состояния могут образовывать иерархию произвольной глубины вложенности. Каждое вложенное состояние получает исходящие переходы от содержащего его композиционного состояния. Входящие переходы обязательно должны быть направлены в конкретное вложенное состояние, в противном случае возникает неоднозначность.

На рис. 6.5 показана диаграмма состояний автоматической коробки передач. Рукоятка коробки может находиться в трех положениях: назад, нейтраль и вперед. Если рукоятка находится в положении вперед, сама коробка передач может включать первую, вторую или третью скорость. Состояния *First* (Первая), *Second* (Вторая) и *Third* (Третья) являются вложенными по отношению к состоянию *Forward* (Вперед).

Каждое из вложенных состояний имеет общие с композиционным состоянием исходящие переходы. На какой бы передаче вы ни переключились на нейтраль, переход будет выполнен. Переход из состояния *Forward* (Вперед) в состояние *Neutral* (Нейтраль) подразумевает целых три перехода, по одному с каждой из передач на нейтраль. При переключении с *Neutral* на *Forward* происходит переход в состояние *Forward*. Однако само состояние *Forward* является абстрактным, тогда как управление должно обязательно находиться в конкретном состоянии, поэтому реальный переход происходит в одно из вложенных состояний, а именно в состояние *First*, потому что оно является начальным состоянием по умолчанию. Это показывает переход без названия из закрашенного кружка внутри контура состояния *Forward*.

По событию *Stop* все три вложенных состояния переходят в состояние *First*, что показывает переход с контура *Forward* в состояние *First*. Это означает, что остановка машины приводит к переключению на первую передачу, на какой бы передаче мы перед этим ни ехали.

Система обозначений позволяет описать более сложные ситуации, например явный переход из вложенного состояния в состояние, внешнее по отношению

к композитному состоянию, или явный переход из вложенного состояния на контур внешнего состояния. В подобных случаях все состояния должны находиться на одной диаграмме. В простых случаях, когда взаимодействие ограничивается запуском и завершением, вложенные состояния можно изображать на отдельных диаграммах и ссылаться на них, как на вложенные автоматы (см. пример на рис. 6.2).

В простых задачах вложенные состояния можно реализовать, сделав диаграммы состояния плоскими. Можно также сделать каждое состояние классом, однако при этом важно следить за тем, чтобы не потерять индивидуальность объектов. Операция *becomes* в языке Smalltalk позволяет объекту изменять класс, не теряя при этом индивидуальность, что облегчает превращение состояния в класс. Однако затраты на выполнение операции *becomes* могут стать серьезной проблемой при частой смене состояний. Превращение операции в класс оказывается невыгодным в C++, если не использовать сложные методы, подобные описанным в [Coplien-92]. Java в этом отношении так же неудобна, как и C++.

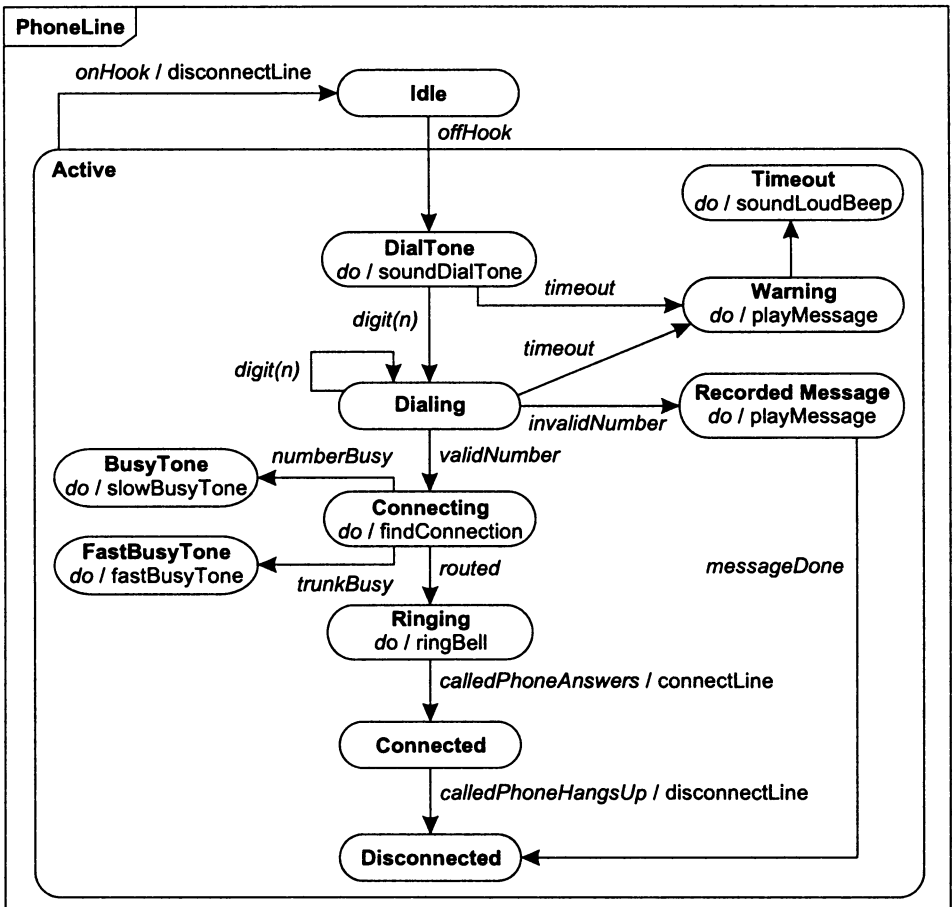


Рис. 6.4. Диаграмма состояний телефонной линии с вложенными состояниями

Деятельность при входе и при выходе особенно полезна на диаграммах с вложенными состояниями, потому что с ее помощью можно выражать состояние (в некоторых случаях — целую вложенную диаграмму) в терминах соответствующей деятельности при входе и при выходе, не заботясь о том, что происходило и будет происходить, когда состояние не активно. Переход во вложенное состояние или из него может приводить к выполнению последовательности деятельности при входе и при выходе, если переход пересекает несколько уровней вложенности. Деятельность при входе осуществляется от внешних состояний к внутренним, а деятельность при выходе — наоборот. Это позволяет реализовать поведение, подобное поведению вызовов вложенных подпрограмм.

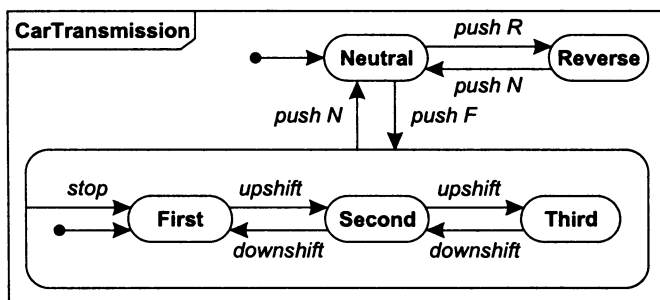


Рис. 6.5. Диаграмма состояний автоматической коробки передач

6.3. Обобщение сигналов

Сигналы могут образовывать иерархию обобщений, наследуя от предков свои атрибуты. На рис. 6.6 изображена часть дерева входных сигналов рабочей станции. Сигналы *MouseButton* (КнопкаМыши) и *KeyboardCharacter* (КлавишаКлавиатуры) являются разновидностями пользовательского ввода. Оба сигнала наследуют атрибут *device* (устройство) от сигнала *UserInput* (ПользовательскийВвод) — корня дерева. Сигналы *MouseButtonDown* (КнопкаМышиВниз) и *MouseButtonUp* (КнопкаМышиВверх) наследуют атрибут *location* (координаты) от сигнала *MouseButton* (КнопкаМыши). Класс *KeyboardCharacter* (КлавишаКлавиатуры) может быть разделен на *Control* (Управляющая) и *Graphic* (Графическая). В конечном итоге любой сигнал можно считать листом дерева обобщений сигналов. На диаграмме состояний сигнал может переключать переходы, в определении которых указаны сигналы, являющиеся его предками. Например, нажатие клавиши *a* вызовет переход, переключаемый сигналом *AlphaNumeric* (АлфавитноЦифровая), а также переход, переключаемый сигналом *KeyboardCharacter* (КлавишаКлавиатуры). Мы рекомендуем делать все сигналы-предки абстрактными, как и при обобщении классов.

Иерархия сигналов позволяет использовать в модели разные уровни абстракции. Например, некоторые состояния могут одинаково обрабатывать все входящие символы, тогда как другие могут обрабатывать управляющие символы иначе, чем печатные, а третьи могут интерпретировать каждый нажатый символ по-своему.

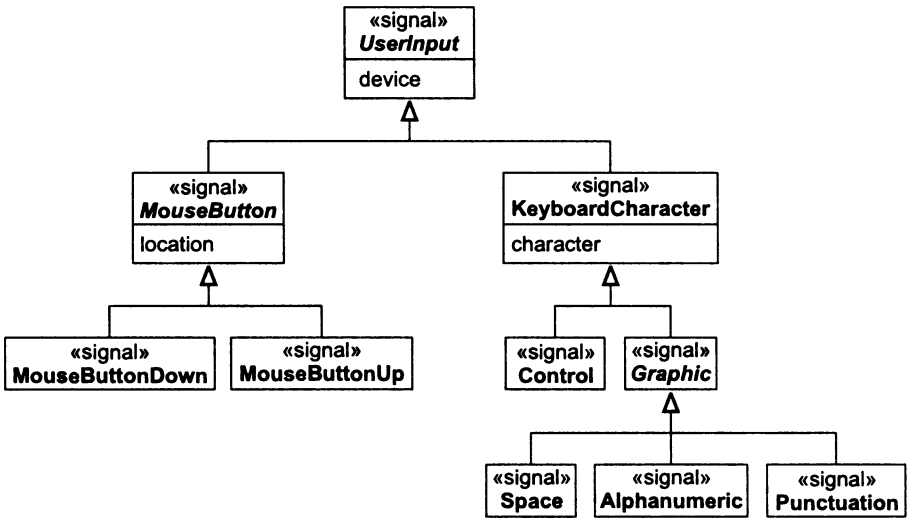


Рис. 6.6. Часть иерархической схемы сигналов, поступающих с клавиатуры

6.4. Параллелизм

Модель состояний неявным образом поддерживает параллелизм разных объектов. Вообще говоря, объекты являются автономными сущностями, которые могут действовать и изменяться независимо друг от друга. Однако полная независимость объектов на практике не выполняется. На них могут быть наложены общие ограничения, вызывающие некоторую корреляцию между изменением состояний объектов.

6.4.1. Параллелизм в агрегации

Диаграмма состояний агрегата является совокупностью диаграмм состояний (по одной на каждую составляющую часть агрегата). Состояние агрегата соответствует комбинации состояний всех его частей. Агрегация является отношением И. Состояние агрегата — это одно состояние с первой диаграммы И одно состояние со второй диаграммы И одно состояние с третьей диаграммы и т. д. В наиболее интересных моделях состояния частей взаимодействуют друг с другом. Переходы одного объекта могут зависеть от того, в каком состоянии находится другой объект. Это обеспечивает взаимодействие между диаграммами состояний при сохранении их модульной структуры.

На рис. 6.7 показано состояние объекта *Car* (Автомобиль), которое представляет собой агрегацию состояний частей: *Ignition* (Зажигание), *Transmission* (Передача), *Accelerator* (Газ), *Brake* (Тормоз) и других, не указанных на диаграмме объектов. Состояние автомобиля включает одно состояние от каждой из частей. Каждая часть совершает переходы из состояния в состояние параллельно с другими частями. Диаграммы состояний частей являются почти независимыми,

но не совсем: автомобиль не заведется, если коробку передач не поставить на нейтраль. Это показывает сторожевое условие *Transmission in Neutral* (Коробка передач на нейтрالي) на переходе от *Ignition-Off* (Зажигание-Выключено) к *Ignition-Starting* (Зажигание-Стартер).

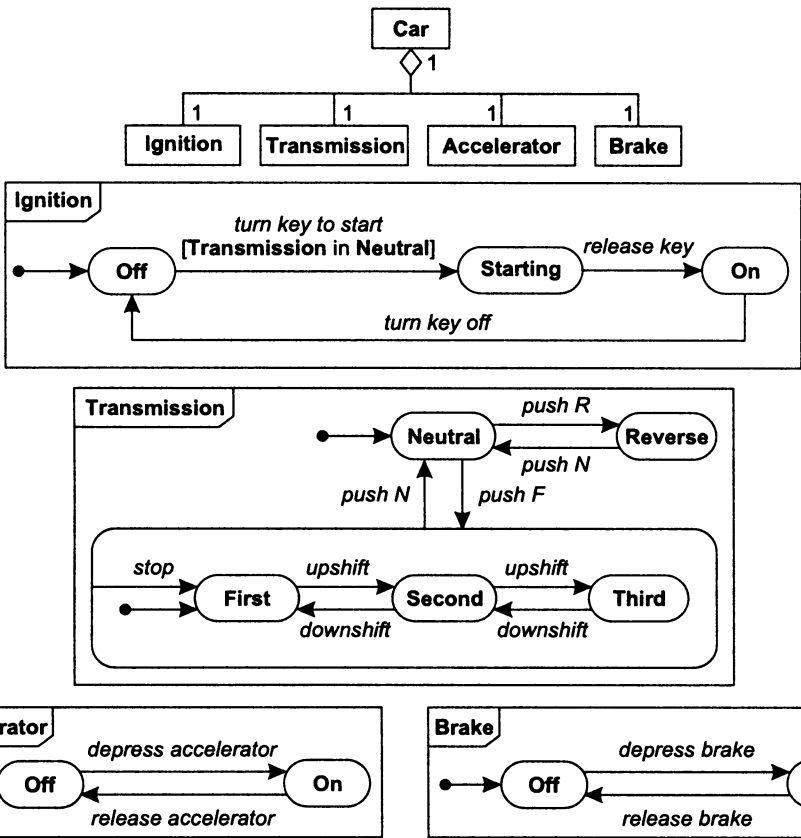


Рис. 6.7. Агрегация и диаграммы параллельных состояний

6.4.2. Параллелизм в объекте

Некоторые объекты можно разбить на подмножества атрибутов или связей, каждое из которых характеризуется своей собственной диаграммой. Состояние объекта включает одно состояние с каждой из поддиаграмм. Эти диаграммы не обязательно должны быть независимыми: одно и то же событие может вызывать переходы на нескольких диаграммах. Параллельность внутри объекта обозначается в UML разбиением композитного состояния на области при помощи пунктирных линий. Название композитного состояния должно быть указано в отдельной закладке, которая не должна быть похожа на одну из областей.

На рис. 6.8 показана диаграмма состояний одного роббера при игре в бридж. Когда сторона выигрывает гейм, она становится «уязвима». Первая сторона,

выигрывающая два гейма, выигрывает и роббер. Состояние роббера включает в себя одно состояние с каждой из вложенных диаграмм. При входе в композитное состояние *Playing rubber* (Играется роббер) обе области находятся в начальных состояниях *Not vulnerable* (Неуязвимы). Каждая область независимо от другой может перейти в состояние *Vulnerable* (Уязвимы), когда соответствующая сторона выигрывает гейм. Когда сторона выигрывает второй гейм, произойдет переход в состояние *Wins rubber* (Выиграли роббер). Этот переход завершает выполнение обеих параллельных областей, потому что они являются частями одного и того же композитного состояния *Playing rubber* и активными могут быть только тогда, когда активно это композитное состояние.

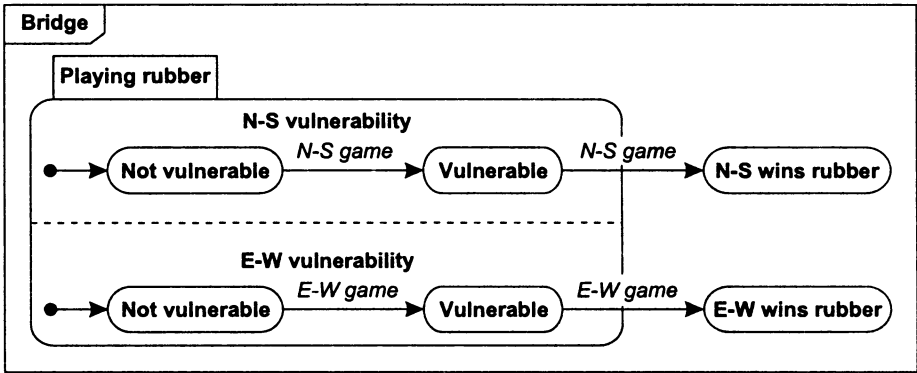


Рис. 6.8. Диаграмма игры в бридж с параллельными состояниями

Большинство языков программирования не имеют внутренней поддержки параллелизма. Для реализации этой концепции вы можете использовать библиотеки, примитивы операционной системы или СУБД. В процессе анализа все объекты должны рассматриваться как действующие параллельно. В процессе проектирования следует выбрать оптимальный вариант. Многие реализации не требуют параллельности, им оказывается вполне достаточно одного потока управления.

6.4.3. Синхронизация параллельной деятельности

Иногда один объект должен участвовать в двух и более видах деятельности параллельно. Объект не синхронизирует внутренние этапы этих видов деятельности, но должен завершить оба вида деятельности перед тем, как он перейдет в следующее состояние. Например, банкомат должен выдать и наличные, и карту пользователя после завершения транзакции. Банкомат не должен возвращаться в состояние бездействия до тех пор, пока пользователь не заберет карту и деньги, причем пользователь может взять их в произвольном порядке или даже одновременно. Порядок не имеет значения, важно только завершение обоих видов деятельности. Это пример разделения управления на параллельные потоки с последующим слиянием.

На рис. 6.9 показана диаграмма состояний для параллельной деятельности. Количество состояний, которые могут быть активными одновременно, зависит от

конкретных событий и в нашем примере может меняться от одного до двух. В языке UML параллельные виды деятельности в рамках одного состояния обозначаются разделением этого состояния на области при помощи пунктирных линий. Каждая область — это вложенная диаграмма, описывающая один из параллельных видов деятельности в рамках композитной деятельности. Композитная деятельность включает ровно одно состояние из каждой поддиаграммы.

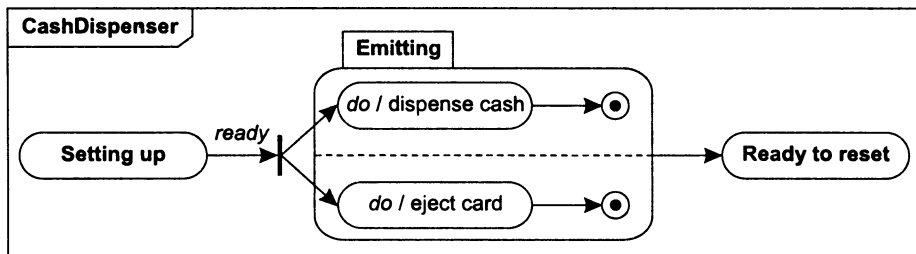


Рис. 6.9. Синхронизация управления

Развилка перехода означает разделение управления на параллельные части. Небольшая жирная полоса с одной входящей стрелкой и двумя исходящими используется для обозначения развилки. Около входящей стрелки указывается событие и необязательное сторожевое условие. Около исходящих стрелок пометок быть не должно. Каждая исходящая стрелка должна быть направлена на состояние из своей собственной поддиаграммы. В нашем примере переход по событию *ready* (готово) разделяется на две параллельные части, по одной на каждую из параллельных поддиаграмм. Когда этот переход запускается, два параллельных подсостояния становятся активными и выполняются независимо друг от друга.

Любой переход в состояние с параллельными областями активирует каждую из них. Если для какого-либо перехода не указано целевое состояние в одной из параллельных областей, эта область начинает выполняться с начального состояния по умолчанию. Поэтому в нашем примере развилку можно было и не рисовать: достаточно было нарисовать переход в состояние *Emitting* (Выдача), и параллельные области начали бы выполняться с начальных состояний.

Слияние управления обозначается в UML явным образом при помощи перехода с одной исходящей и несколькими входящими стрелками, соединенными с небольшой жирной полосой (на рис. 6.9 ее нет). Переключающее событие и необязательное сторожевое условие ставятся около этой полосы. Обратите внимание, что переход запускается одним событием (а не несколькими, по числу стрелок). Если некоторые области композитного состояния не участвуют в слиянии, они автоматически завершаются, как только запускается переход со слиянием. Поэтому переход из одного из параллельных подсостояний в состояние за пределами композитного вызывает завершение всех остальных параллельных состояний. Можно рассматривать это как вырожденное слияние с единственным потоком управления.

Переход по завершении из внешнего композитного состояния в другое состояние неявным образом подразумевает слияние параллельных потоков управления (см. рис. 6.9). Переход по завершении запускается, когда заканчивается деятельность исходного состояния. Композитное состояние с параллельными областями

считается закончившим деятельность, когда заканчивают свою деятельность все его области, то есть когда каждая из них достигает своего конечного состояния. Все подсостояния должны завершиться прежде, чем переход по завершении будет запущен. В нашем примере завершение обоих видов деятельности приводит к тому, что оба подсостояния достигают своих конечных состояний, в результате чего запускается переход со слиянием, и активным становится состояние *Ready to reset* (Готовность к перезапуску). Если бы мы нарисовали отдельный переход из каждого подсостояния в конечное состояние, это имело бы другой смысл: любой переход приводил бы к завершению выполнения второго подсостояния до его реального окончания. Запуск перехода со слиянием приводит к выполнению деятельности при выходе для всех областей, как для явных, так и для неявных слияний.

6.5. Пример модели состояний

Рассмотрим модель состояний реального устройства — программируемого термостата Сирса «Уикендер». На этом примере мы продемонстрируем совместную работу модельных конструкций. Чтобы создать свою модель, мы прочитали инструкцию по эксплуатации и поэкспериментировали с реальным устройством. Устройство управляет обогревателем и кондиционером в соответствии с зависящими от времени атрибутами, которые владелец вводит с клавиатуры.

В процессе работы термостат поддерживает текущую температуру равной целевому значению, управляя нагревателем и кондиционером. Целевая температура берется из таблицы значений в начале каждого периода, описываемого программой. Таблица задает целевую температуру и начальное время для каждого из восьми периодов, четыре из которых относятся к будним дням и четыре — к выходным. Пользователь всегда может вручную ввести значение, перекрывающее целевую температуру из таблицы.

Пользователь программирует термостат при помощи десяти кнопок и трех выключателей. Параметры отображаются на алфавитно-цифровом дисплее. Каждая кнопка при каждом нажатии порождает событие. Мы приписываем свое событие вводу каждой из кнопок:

- *TEMP UP* (Температура вверх) — увеличивает целевую температуру или температуру в программе.
- *TEMP DOWN* (Температура вниз) — понижает целевую температуру или температуру в программе.
- *TIME FWD* (Время вперед) — увеличивает показания часов или время в программе.
- *TIME BACK* (Время назад) — уменьшает показания часов или время в программе.
- *SET CLOCK* (Установка часов) — установка текущего времени суток.
- *SET DAY* (Установка дня) — установка текущего дня недели.
- *RUN PRGM* (Выполнить программу) — выход из настройки и начало выполнения программы.

- *VIEW PRGM* (Просмотреть программу) — вход в режим программирования для отображения и изменения восьми пар значений времени и температуры.
- *HOLD TEMP* (Поддерживать температуру) — поддерживать текущую температуру независимо от программы.
- *F-C BUTTON* (Фаренгейт/Цельсий) — переключение между градусами Фаренгейта и Цельсия.

Каждый выключатель позволяет выбрать одно из двух или трех значений параметра. В модели мы представляем каждый выключатель в виде независимой параллельной поддиаграммы, по одному состоянию на каждое положение выключателя. Хотя мы присваиваем названия событий изменению состояния, для устройства имеет значение именно состояние выключателя (а не его изменение). Итак, устройство снабжено следующими выключателями:

- *NIGHT LIGHT* (Подсветка) — управляет подсветкой дисплея. Значения: *light off* (подсветки нет), *light on* (подсветка есть).
- *SEASON* (Время года) — выбирает устройство, которым управляет термостат. Значения: *heat* (обогреватель), *cool* (кондиционер), *off* (устройства нет).
- *FAN* (Вентилятор) — управляет работой вентилятора. Значения: *fan on* (вентилятор работает непрерывно), *fan auto* (вентилятор работает только одновременно с обогревателем или кондиционером).

Термостат управляет реле обогревателя, кондиционера и вентилятора. В модели мы представляем это деятельностью *run furnace* (включить обогреватель), *run air conditioner* (включить кондиционер) и *run fan* (включить вентилятор).

Термостат снабжен датчиком температуры воздуха, показания которого он непрерывно контролирует. В модели мы описываем это внешним параметром *temp* (температура). Кроме того, термостат снабжен часами, показания которых он непрерывно считывает и отображает на дисплее. Часы мы тоже моделируем внешним параметром *time*, поскольку модель состояний часов нас в этой задаче не интересует. При построении модели состояний важно учитывать только те состояния, которые влияют на поток управления, а всю прочую информацию описывать переменными и параметрами. Внутренняя переменная состояния *target temp* (целевая температура) обозначает температуру, которую термостат пытается поддерживать. Некоторые виды деятельности считывают значение этой переменной, а другие устанавливают его. Переменная состояния обеспечивает взаимодействие между разными частями модели состояний.

На рис. 6.10 показана диаграмма состояний высшего уровня для программируемого термостата. Она содержит семь параллельных поддиаграмм (областей). Пользовательский интерфейс мы показываем отдельно (рис. 6.11). На этой же диаграмме находятся тривиальные поддиаграммы переключателей *SEASON* и *FAN*. Четыре другие поддиаграммы показывают выходные параметры термостата: состояния реле обогревателя, кондиционера, индикатора и вентилятора. Каждая из этих поддиаграмм содержит подсостояния *Off* (Выкл) и *On* (Вкл). Состояние каждой поддиаграммы полностью определяется входными параметрами и состоянием других поддиаграмм (например, переключателей *SEASON* и *FAN*). Состояние четырех поддиаграмм в правой части полностью выводимо из других частей модели и не содержит никакой дополнительной информации.

На рис. 6.11 показана поддиаграмма, описывающая пользовательский интерфейс. Она содержит три параллельные поддиаграммы: интерактивный дисплей, температурный режим, подсветка. Подсветка управляется выключателем, поэтому начальное значение для нее не имеет смысла. Значение подсветки может быть определено непосредственно. Режим отображения температуры управляется единственной кнопкой, позволяющей переключаться между градусами Фаренгейта и Цельсия. Здесь начальное состояние по умолчанию необходимо: когда устройство включается, оно отображает температуру в градусах Фаренгейта.

Поддиаграмма интерактивного дисплея представляет существенно больший интерес. Устройство может находиться либо в режиме работы, либо в режиме настройки. Состояние *Operate* (Работа) делится на три параллельных подсостояния. К одному из них относятся *Run* (Выполнение программы) и *Hold* (Поддержание температуры), другое управляет отображением целевой температуры, а третье управляет отображением времени и текущей температуры. Каждые две секунды дисплей переключается между режимом отображения текущего времени и режимом отображения текущей температуры. Целевая температура отображается постоянно и может корректироваться кнопками *TEMP UP* и *TEMP DOWN*, а также событием *set target* (установка целевой температуры), которое может быть порождено только в состоянии *Operate* (Работа). Обратите внимание, что параметр *target temp*, устанавливаемый на этой диаграмме, — это тот же самый параметр, который управляет выходными реле.

Каждую секунду, проведенную в состоянии *Run* (Выполнение программы), текущее время сравнивается с временами из программной таблицы. Если текущее время совпадает с одним из указанных в таблице значений, осуществляется переход к следующему пункту программы, и вход в состояние *Run* выполняется заново. Вход в это состояние выполняется и в том случае, если в каком-либо состоянии нажимается кнопка *RUN PRGM* (Выполнить программу), на что указывает переход, соединяющий контур с состоянием *Operate*, и переход по умолчанию в состояние *Run*. При входе в это состояние деятельность при входе устанавливает целевую температуру равной значению из программной таблицы.

Пока устройство находится в состоянии *Hold* (Поддержание температуры), автоматического изменения целевой температуры в соответствии с программой не происходит. Температура может быть изменена только пользователем при помощи кнопок *TEMP UP* и *TEMP DOWN*. Если интерфейс находится в одном из подсостояний настройки в течение 90 секунд и пользователь ничего не вводит, система переходит в состояние *Hold*. Вход в подсостояние *Hold* одновременно вызывает переход в начальные состояния двух других параллельных областей *Operate* (Работа). Состояние *Setup* (Настройка) было включено в модель только для того, чтобы сгруппировать три вложенных состояния настройки для задания перехода по 90-секундному тайм-ауту. Обратите внимание на небольшую странность в поведении устройства: кнопка *HOLD* в состоянии *Setup* (Настройка) не действует, но перейти в состояние *Hold* (Поддержание температуры) все-таки можно, подождав 90 секунд.

На рис. 6.12 показаны три поддиаграммы настройки. Нажатие кнопки *SET CLOCK* (Установка часов) вызывает переход во вложенное состояние *Set minutes* (Установка минут), потому что оно является начальным состоянием по умолчанию. Последующие нажатия кнопки *SET CLOCK* вызывают переключение между вложенными состояниями *Set hours* (Установка часов) и *Set minutes* (Установка минут). Кнопки *TIME FWD* (Время вперед) и *TIME BACK* (Время назад) изменяют

время в программной таблице. Нажатие *SET DAY* (Установка дня) вызывает переход во вложенное состояние *Set day*. На дисплее отображаются дни недели. Последующие нажатия *SET DAY* переключают дни недели.

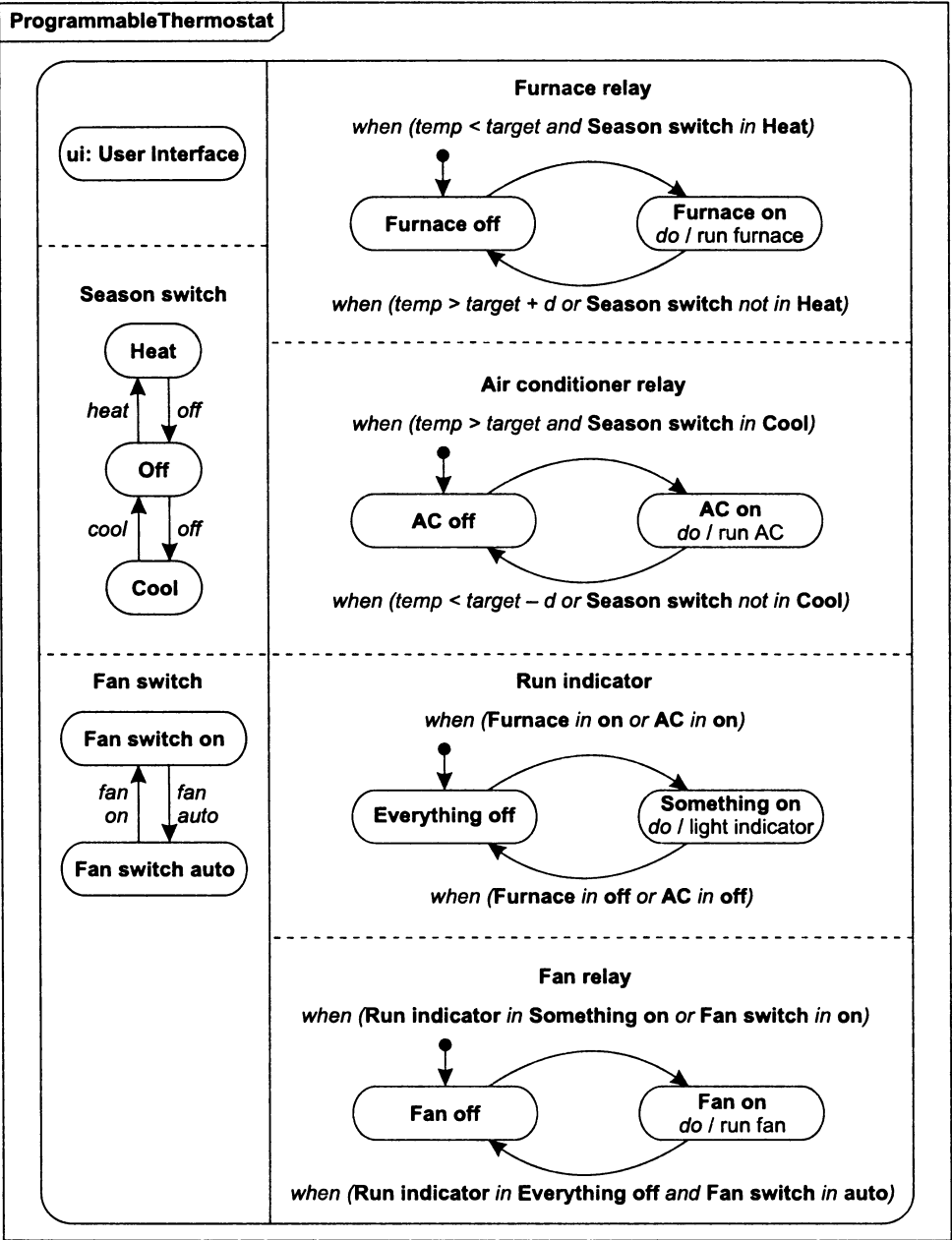


Рис. 6.10. Диаграмма состояний программируемого термостата

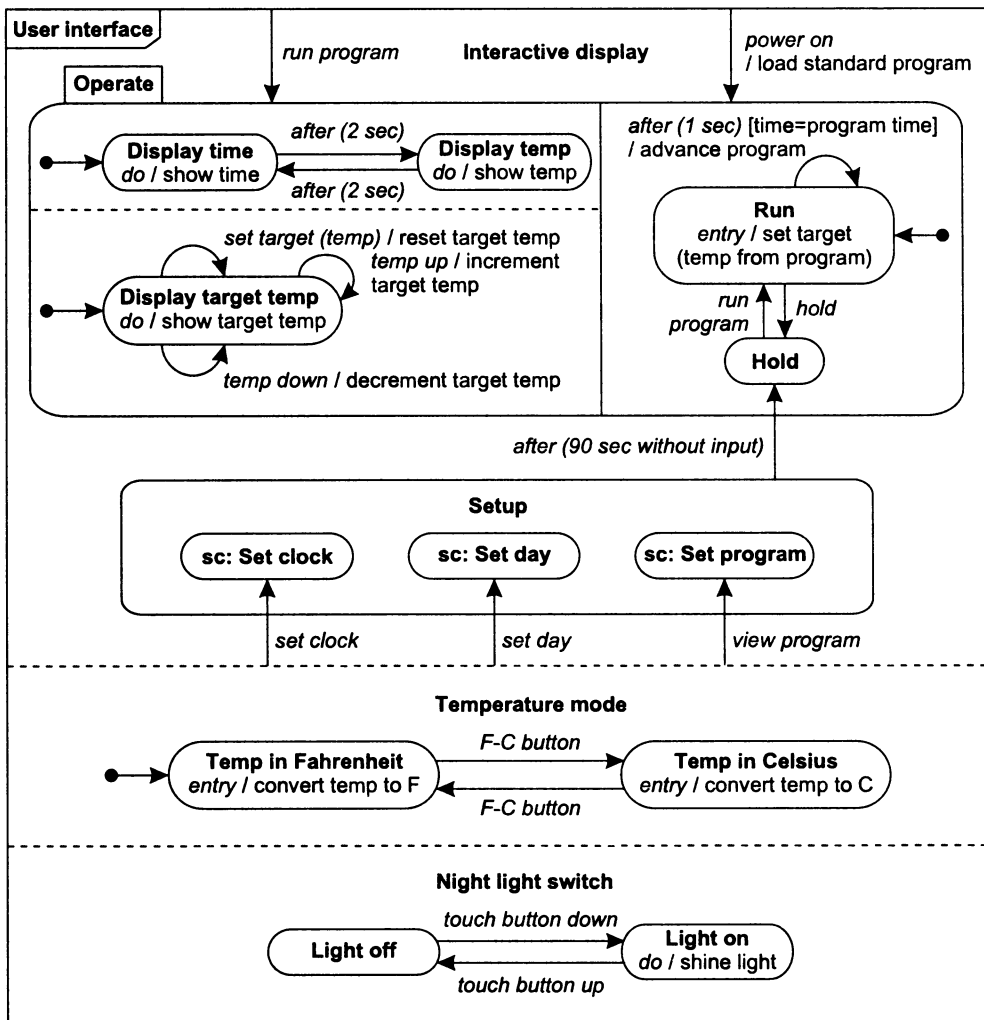


Рис. 6.11. Поддиаграмма пользовательского интерфейса термостата

Нажатие кнопки *VIEW PRGM* (Просмотреть программу) вызывает переход во вложенное состояние *Set program* (Установка программы), которое состоит из трех параллельных поддиаграмм, управляющих отображением времени, температуры и периода программы. Состояние *Set program* всегда начинается с первого периода программы. Последующие события *view program* вызывают переключение между периодами. Событие *view program* показано на всех трех поддиаграммах, потому что каждая из них переключается на отображение следующего значения. Обратите внимание, что события *time fwd* и *time back* изменяют время с шагом в 15 минут, в отличие от тех же событий в состоянии *Set clock*. Кроме того, переходы *temp up* и *temp down* снабжены сторожевыми условиями, потому что температура должна находиться в разрешенном диапазоне.

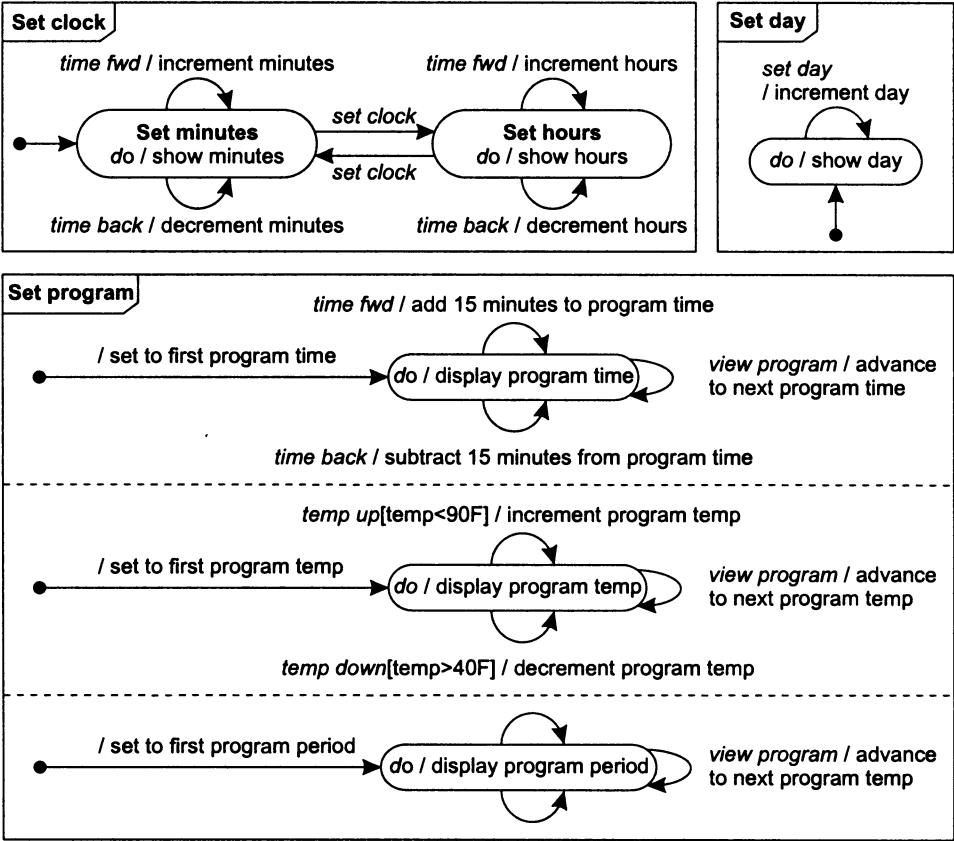


Рис. 6.12. Поддиаграммы режима настройки термостата

Ни у одного из вложенных состояний *Interactive display* (Интерактивный дисплей) нет явных исходящих переходов. Каждое вложенное состояние завершается неявным образом переходом в другое вложенное состояние с основного контура *Interactive display*.

6.6. Модель состояний и модель классов

Модель состояний указывает разрешенные последовательности изменений объектов из модели классов. Диаграмма состояний полностью или частично описывает поведение объектов одного класса. Состояния являются классами эквивалентности для значений и связей объекта.

Структура состояний связана со структурой классов. Вложенное состояние уточняет возможные значения и связи для своего объекта. И обобщение классов, и вложение состояний позволяют разбить множество возможных значений объекта. Один объект может находиться в разных состояниях в разные моменты времени, сохраняя свою индивидуальность, но не может изменять класс. Поэтому внутренние

различия объектов правильно отражать в модели в виде разных классов, тогда как временные различия представляются разными состояниями одного класса.

Композитное состояние — это агрегация нескольких параллельных состояний. В модели классов существует три источника параллелизма. Во-первых, это агрегация объектов: каждая часть агрегации имеет свое состояние, поэтому агрегат в целом находится в состоянии, которое представляется как комбинация состояний всех его частей. Вторым источником является агрегация в рамках объекта: частями объекта являются его значения и связи. Их группы определяют параллельные подсостояния композитного состояния объекта. Третьим источником является параллельное поведение объекта (пример приведен на рис. 6.9). Три источника параллельности обычно являются взаимозаменяемыми. Например, объект может содержать атрибут, указывающий на выполнение некоторой деятельности.

Модель состояний класса наследуется его подклассами. Подклассы наследуют состояния своего предка и переходы между ними. Подклассы могут иметь и свои собственные диаграммы состояний. Но каким образом взаимодействуют между собой диаграммы состояний подкласса и суперкласса? Если они описывают несовместные наборы атрибутов, то никаких проблем это не создает: подкласс находится в композитном состоянии, состоящем из параллельных диаграмм.

Если же на диаграмме состояний подкласса присутствуют атрибуты с диаграммы состояний суперкласса, в модели возможен конфликт. Диаграмма состояний подкласса должна быть уточнением диаграммы состояний суперкласса. Любое состояние из диаграммы состояний предка может получить вложенные состояния или может быть разбито на параллельные области, однако новые состояния или переходы не могут добавляться на диаграмму состояний предка непосредственно, поскольку диаграмма состояний предка должна быть проекцией диаграммы состояний потомка. Несмотря на то что уточнение унаследованных диаграмм состояний возможно, в большинстве случаев следует делать диаграмму состояний подкласса независимым ортогональным параллельным дополнением к диаграмме, унаследованной от суперкласса, определяя ее на несовместном наборе атрибутов (обычно на тех атрибутах, которые были введены в данном подклассе).

Иерархия сигналов не зависит от иерархии классов, которые обмениваются этими сигналами. Это утверждение всегда выполняется на практике, если и не в теории. Сигналы могут определяться на разных классах объектов. Сигналы более фундаментальны, чем состояния. Состояния определяются взаимодействием объектов и событий. Переходы часто могут быть реализованы как операции с объектами, причем имя операции может соответствовать названию сигнала. Сигналы обладают большей выразительностью, нежели операции, поскольку действие сигнала зависит не только от класса объекта, но и от его состояния.

6.7. Практические рекомендации

Все перечисленные ниже советы уже упоминались в тексте главы, но мы решили повторить их здесь еще раз для вашего удобства.

- **Структурированные диаграммы состояний.** Используйте структуру для упорядочения моделей с большим числом состояний (раздел 6.1).

- **Вложение состояний.** Используйте вложенные состояния в тех случаях, когда один переход применим ко множеству состояний (раздел 6.2).
- **Конкретные суперсигналы.** Как и в модели классов, конкретных суперсигналов лучше избегать. Тогда будет очевидно, какие сигналы являются абстрактными, а какие конкретными. Все суперсигналы будут абстрактными, а сигналы-листья — конкретными. Для абстрагирования суперсигнала достаточно ввести сигнал-потомок *Other* (Прочие) (раздел 6.3).
- **Параллелизм.** Чаще всего параллелизм связан с агрегацией объектов. На диаграмме состояний его можно не показывать явным образом. Для определения параллельных видов деятельности одного объекта используйте композитные состояния (раздел 6.4).
- **Согласованность диаграмм.** Проверяйте согласованность различных диаграмм состояний при обработке общих событий, чтобы модель состояний в целом была корректной (раздел 6.5).
- **Модель состояний и наследование классов.** Старайтесь делать диаграммы состояний подклассов не зависящими от диаграмм состояний их предков. Диаграммы состояний подклассов должны описывать изменение атрибутов, которые были добавлены в этих подклассах (раздел 6.6).

6.8. Резюме

Модель классов описывает объекты, значения и связи, которые могут существовать в системе. Значения и связи объекта задают его состояние. С течением времени объекты воздействуют друг на друга, в результате чего их состояния претерпевают последовательность изменений. Объекты реагируют на события — происшествия, связанные с конкретными моментами времени. Реакция на событие зависит от состояния, в котором находится объект, и может заключаться в изменении состояния или отправке сигнала отправителю первого сигнала или иному объекту.

Комбинацию событий, состояний и переходов между ними для определенного класса можно абстрагировать и представить на диаграмме состояний. Диаграмма состояний — это граф состояний и переходов, подобный диаграмме классов, которая является графом классов и отношений между ними. Модель состояний состоит из множества диаграмм состояний (по одной на каждый класс, динамическое поведение которого важно для модели в целом) и показывает возможное поведение системы в целом. Каждый объект следует диаграмме состояний своего класса независимо от других объектов. Диаграммы состояний разных классов взаимодействуют друг с другом посредством общих событий.

Состояния и события могут детализироваться по мере разработки модели. Вложенные события имеют общие переходы с композитными состояниями, к которым они относятся. Сигналы могут быть упорядочены в иерархии наследования. Сигналы-потомки переключают те же переходы, что и сигналы-предки.

Неотъемлемым свойством объектов является параллелизм. Каждый объект обладает своим собственным состоянием. На диаграммах состояний параллелизм изображается как агрегация параллельных состояний, каждое из которых выполняется независимо от других. Параллельные объекты взаимодействуют, обмениваясь событиями и проверяя состояние других объектов, в том числе и состояний. Помимо простых переходов существуют также разделения и слияния потока управления.

Деятельность при выходе и при входе относится ко всем входящим и исходящим переходам. Их использование позволяет применять замкнутые диаграммы состояний в различных контекстах. Внутренняя деятельность соответствует переходам, не выходящим из состояния.

Подкласс наследует диаграммы состояний от своих предков и может определять свои собственные диаграммы. Унаследованная диаграмма состояний может быть уточнена разбиением состояний на вложенные состояния или параллельные поддиаграммы.

Реалистичная модель программируемого термостата занимает три страницы и демонстрирует тонкости поведения, не показанные в руководстве по эксплуатации и не проявляющиеся в каждодневной работе.

Таблица 6.1. Ключевые понятия главы

составное состояние	вложенные диаграммы состояний	модель состояний
параллелизм	область	синхронизация
управление	обобщение сигналов	вложенный конечный автомат
вложенное состояние	агрегация состояний	

Библиографические замечания

В этой главе мы в значительной мере следуем трудам Дэвида Хэрела (David Harel), который формализовал свои концепции в системе обозначений, названной им диаграммой состояний (state chart) [Harel-87]. Подход Хэрела на сегодняшний день является наиболее успешной попыткой структурировать конечные диаграммы состояний и избежать комбинаторного взрыва, который всегда создавал проблемы для разработчиков. Хэрел описывает систему обозначений для диаграмм состояний, основанную на контурах, как частный случай общей системы обозначений для диаграмм, которую он называет higraph [Harel-88].

В первом издании этой книги рассматривалась концепция обобщения состояний, однако в соответствии с UML2 мы выкинули ее из второго издания. Мета-модель UML2 позволяет применять обобщения только к классификаторам, а состояние таковым не является. Обобщение классов имеет много общего с вложением состояний, но строго говоря обобщения состояний как такового в UML2 нет.

В UML2 имеется много тонкостей, касающихся моделирования состояний. Более подробно вы можете прочитать об этом в книге [Rumbaugh-05].

Благодарим Майкла Берндтссона за предложенное им упражнение 6.12.

Ссылки

[Coplien-92] James O. Coplien. *Advanced C++: Programming Styles and Idioms*. Boston: Addison-Wesley, 1992.

[Harel-87] David Harel. *Statecharts: a visual formalism for complex systems*. *Science of Computer Programming* 8 (1987), 231–274.

[Harel-88] David Harel. On visual formalisms. *Communications of ACM* 31, 5 (May 1988), 514–530.

[Rumbaugh-05] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Boston: Addison-Wesley, 2005.

Упражнения

- 6.1. (3) Управление направлением движения первых игрушечных электропоездов осуществлялось отключением питания поезда. Нарисуйте диаграммы состояний для прожектора и колес поезда в соответствии с перечисленными ниже последовательностями событий.
 - Питание выключено, поезд не движется.
 - Питание включается, поезд едет вперед, прожектор светится.
 - Питание выключается, поезд останавливается, прожектор гаснет.
 - Питание включается, прожектор загорается, поезд не едет.
 - Питание выключается, прожектор гаснет.
 - Питание включается, поезд едет назад, прожектор светится.
 - Питание выключается, поезд останавливается, прожектор гаснет.
 - Питание включается, прожектор загорается, поезд не едет.
 - Питание выключается, прожектор гаснет.
 - Питание выключается, поезд едет вперед, прожектор светится.
- 6.2. (6) Измените диаграмму состояний из упражнения 5.2, добавив ускоренную установку времени длительным нажатием кнопки В. Если кнопка В нажимается и держится нажатой в течение 5 секунд в режиме установки времени, часы или минуты (в зависимости от подрежима) увеличиваются на единицу каждые полсекунды. (Замечание преподавателю: вы можете предоставить студентам наш ответ к упражнению 5.2 в качестве отправной точки для этого упражнения.)
- 6.4. (5) Переделайте диаграмму состояний из упражнения 5.6 с учетом схожести стартового и рабочего режимов. При исчезновении сигнала ON из любого из этих состояний происходит переход в выключенное состояние. (Замечание преподавателю: вы можете предоставить студентам наш ответ к упражнению 5.6 в качестве отправной точки для этого упражнения.)
- 6.5. (6) Трехфазный индукционный двигатель может вращаться либо по часовой стрелке, либо против часовой стрелки, в зависимости от подключения

фаз. Если для устройства требуется, чтобы двигатель мог вращаться в обе стороны, его можно подключить через два реле (свое реле для каждого направления). Мощные двигатели часто подключаются через трансформаторы, уменьшающие влияние двигателя на сеть питания. Трансформатор отключается от цепи через некоторое время после того, как двигатель разгонится до нужной скорости. Для этой цели используется третье — обходное — реле. Имеется три мгновенных управляющих входа: «вперед», «назад» и «выключить». Когда двигатель выключен, сигнал «вперед» или «назад» запускает двигатель в нужном направлении. Если двигатель уже вращается в направлении «вперед», сигнал «назад» игнорируется (и наоборот). Сигнал «выключить» отключает двигатель независимо от его состояния.

На рис. 6.1 приведена диаграмма состояний возможной реализации системы управления двигателем. Разбейте одну диаграмму состояний на две параллельные диаграммы, одна из которых будет управлять направлением двигателя, а другая — его запуском.

- 6.5. (3) Система управления из предыдущего упражнения не предусматривает защиты от перегрева. Измените диаграмму состояний с рис. Уб.1 таким образом, чтобы в случае перегрева двигатель отключался. Измените параллельные диаграммы состояний, полученные в упражнении 6.4 так, чтобы в случае перегрева двигатель отключался.

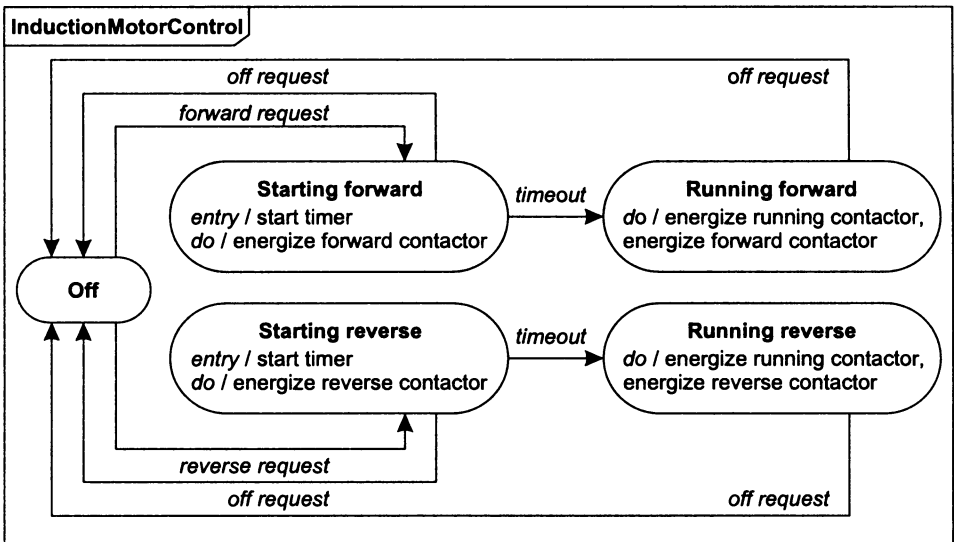


Рис. Уб.1. Диаграмма состояний системы управления индукционного двигателя

- 6.6. (2) Создайте иерархию обобщений для следующих сигналов: выбор, ввод символа, выбор линии, выбор окружности, выбор прямоугольника, выбор текста, входной сигнал.

- 6.7. (7) Газовая система отопления с принудительным нагнетанием горячего воздуха поддерживает температуру и влажность в комнатах в зимних условиях. Система снабжена распределенным интерфейсом. Условия в разных комнатах могут контролироваться независимо друг от друга. Тепло от нагревателя распределяется по комнатам в соответствии с текущим и заданным значениями температуры для каждой комнаты. Если по крайней мере одна комната нуждается в подаче теплого воздуха, нагреватель включается. Когда температура горелки поднимается достаточно высоко, включается вентилятор, который подает горячий воздух через воздуховоды. Когда температура нагревателя превышает верхнее ограничение, нагреватель отключается, а вентилятор продолжает работать. Заглушки в комнатах позволяют системе доставлять тепло только в те комнаты, где оно действительно нужно, в объеме, пропорциональном разности температур. Когда обогрев перестает требоваться, горелка отключается, но вентилятор продолжает работать до тех пор, пока она не остынет.

Влажность также поддерживается исходя из заданной влажности, текущего значения и температуры наружного воздуха. Влажность устанавливается пользователем для всего дома сразу. Измерения влажности проводятся на холодном воздухе, который возвращается к вентилятору. Когда система определяет, что влажность упала слишком низко, включается увлажнитель, встроенный в горелку. Увлажнитель, распыляющий воду в поток нагретого воздуха, включается только вместе с горелкой.

Разбейте модель состояний системы на несколько параллельных диаграмм состояний. Опишите функционирование каждой из них, не вдаваясь в подробности, касающиеся состояний или деятельности.

- 6.8. На рис. У6.2 изображена часть диаграммы состояний видеоманитофона. Видеоманитфон снабжен несколькими кнопками: *select* (выбор), *on/off* (вкл/выкл), *set* (установка) — для установки часов и таймеров автоматического запуска и остановки, *auto* (авто) — включает автоматическую запись, *vcr* (обход видео), *timed* (запись сегмента, кратного 15 мин). Часть событий на рис. У6.2 соответствуют нажатиям соответствующих клавиш. Некоторые кнопки являются переключателями. Например, нажатие *vcr* осуществляет переключение между режимами *VCR* и *TV*. Некоторые кнопки, служащие для ручного управления видеоманитофоном, на рис. У6.2 не показаны: это *play*, *record*, *fast forward*, *rewind*, *pause*, *eject*. Эти кнопки включаются только в состоянии *Manual*. Выполните следующие действия:

- 1) (2) Напишите список событий и деятельности с короткими определениями.
- 2) (7) Напишите руководство пользователя по работе с видеоманитофоном.
- 3) (7) Добавьте на диаграмму состояния, описывающие функционирование второго таймера для второго канала.
- 4) (7) Обсудите, каким образом можно избежать дублирования в этой модели. Например, установка часа может осуществляться в различных контекстах, но одинаковым образом.

VideoCassetteRecorder

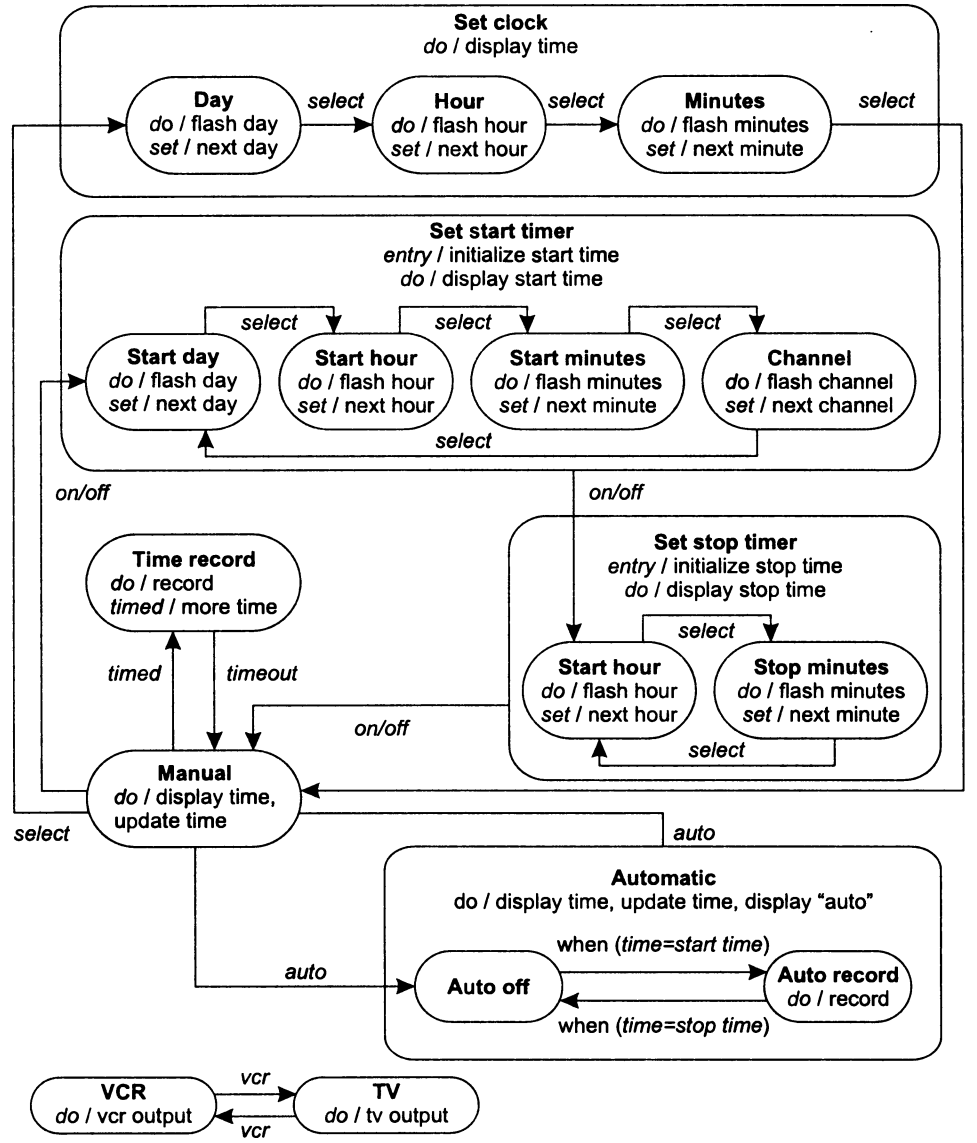


Рис. У6.2. Часть диаграммы состояний видеомэгнитофона

6.9. (6) Диаграмма У5.4 обладает одним существенным недостатком. Питание может быть выключено в любой момент, и машина должна осуществить переход в состояние выключено. Мы могли бы добавить к каждому состоянию переход в выключенное состояние, но при этом мы перегрузили бы диаграмму. Исправьте проблему при помощи вложенных состояний.

6.10. (6) На рис. У6.3 изображена диаграмма классов для двух игроков в настольный теннис. Создайте модель состояний, соответствующую этой модели классов.

Правила игры выглядят так. В начале игры два игрока разыгрывают подачу. Для этого они перебрасываются мячиком через сетку несколько раз. Победитель подает первым и делает это пять раз. После этого пять раз подает второй игрок, затем снова первый. Подачи сменяются до тех пор, пока один из игроков не станет победителем игры.

Чтобы выиграть матч, нужно либо достичь счета 11-0, либо набрать 21 очко с преимуществом по крайней мере в 2 очка. Если счет становится 20-20, игроки меняются подачами до тех пор, пока один из них не оторвется на 2 очка. Он и станет победителем.

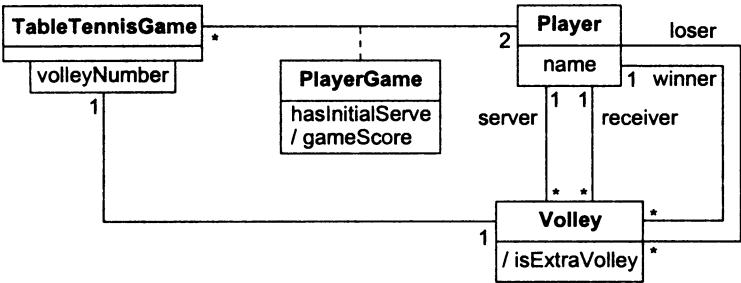


Рис. У6.3. Модель классов для игры в настольный теннис

6.11. (10) Иногда оказывается полезным использовать воплощение, то есть делать объектом нечто, изначально не являвшееся таковым. Воплощение полезно для метаприложений, потому что оно позволяет сдвинуть уровень абстрагирования. В некоторых случаях бывает полезно делать атрибуты, методы, ограничения и управляющую информацию объектами, чтобы иметь возможность описывать их и манипулировать ими как данными.

Создайте модель классов, воплощающую и поддерживающую следующие концепции моделирования состояний: событие, состояние, переход, условие, деятельность, событие сигнала, событие изменения, атрибут сигнала.

6.12. (7) Удалите вложенность состояний с модели с рис. 6.5. Иными словами, вам следует создать плоскую диаграмму состояний, обладающую той же семантикой.

Моделирование взаимодействий

Модель взаимодействия — третья составляющая моделирования систем. Модель классов описывает объекты системы и отношения между ними, модель состояний описывает жизненные циклы объектов, а модель взаимодействия описывает взаимодействие объектов между собой для получения нужных результатов.

Модель взаимодействия — это глобальный взгляд на поведение множества объектов, тогда как модель состояний — это редуцированное представление индивидуального поведения объектов. Для полного описания поведения необходимы обе модели. Они дополняют друг друга, позволяя рассматривать поведение с двух разных точек зрения.

Взаимодействия можно моделировать на разных уровнях абстрагирования. На самом высоком уровне взаимодействие системы со внешними действующими лицами описывается вариантами использования. Каждый вариант использования описывает элемент функциональности, предоставляемой системой ее пользователям. Варианты использования полезны для представления в модели неформальных требований.

Диаграммы последовательности более детализированы, они показывают сообщения, которыми обмениваются объекты с течением времени. К сообщениям относятся асинхронные сигналы и вызовы процедур. Диаграммы последовательности полезны для демонстрации последовательностей поведения, видимых пользователям системы.

Наконец, диаграммы деятельности содержат всю информацию и показывают поток управления между этапами вычислений. Диаграммы деятельности могут показывать не только потоки управления, но и потоки данных. С их помощью документируются этапы диаграмм последовательности, необходимые для реализации операции или бизнес-процесса.

7.1. Модели вариантов использования

7.1.1. Действующие лица

Действующее лицо (actor) — это непосредственный внешний пользователь системы. Это объект или множество объектов, непосредственно взаимодействующих с системой, но не являющихся ее частью. Каждое действующее лицо является обобщением группы объектов, ведущих себя определенным образом по отношению к системе. Например, клиент и ремонтник являются разными действующими лицами по отношению к торговому автомату. Действующими лицами по отношению к системе бюро путешествий могут быть путешественники, агенты и авиакомпании. По отношению к компьютерной базе данных ими могут быть пользователи и администраторы. Действующими лицами могут быть люди, устройства и другие системы — все, что взаимодействует с интересующей нас системой непосредственно.

Объект может быть связан с несколькими действующими лицами, если его поведение обладает разными гранями. Например, объекты Мэри, Фрэнк и Пол могут быть клиентами торгового автомата. Пол может одновременно являться и ремонтником, обслуживающим этот автомат.

Действующее лицо должно иметь одну четко определенную цель. Объекты и классы обычно сочетают в себе несколько назначений (целей). Действующее же лицо представляет лишь одну грань объекта в его взаимодействии с системой. Одно и то же лицо может представлять объекты разных классов, ведущие себя одинаково по отношению к системе. Например, люди, так или иначе взаимодействующие с торговым автоматом, могут быть разделены на две основные категории: клиенты и ремонтники. Каждое действующее лицо обладает согласованным набором возможностей тех объектов, которые оно представляет.

Моделирование действующих лиц помогает определить границы системы, то есть идентифицировать объекты, находящиеся внутри системы, и объекты, лежащие на ее границе. Действующее лицо непосредственно взаимодействует с системой. Объекты, участвующие только в косвенных взаимодействиях с системой, не являются действующими лицами и потому не должны включаться в модель системы. Любое косвенное взаимодействие должно осуществляться посредством действующих лиц. Например, диспетчер мастеров по ремонту торговых автоматов не является действующим лицом по отношению к этим автоматам. Непосредственно с автоматами работают только мастера. Если нужно построить модель косвенного взаимодействия внешних объектов, нужно строить модель среды как большей системы, включающей в себя исходную систему. Например, можно построить модель ремонтной службы, включающую в себя (в качестве действующих лиц) диспетчеров, мастеров и торговые автоматы, но эта модель уже не будет моделью торговых автоматов.

7.1.2. Варианты использования

Различные взаимодействия действующих лиц с системой группируются в варианты использования. Вариант использования (use case) — это связный элемент функциональности, предоставляемый системой при взаимодействии с действующим

щими лицами. Например, лицо *Клиент* может *купить напиток* в торговом автомате. Клиент кидает монеты в автомат, выбирает продукт и забирает свой напиток. *Ремонтник* может *провести техническое обслуживание* автомата. На рис. 7.1 приведены несколько вариантов использования для торгового автомата.

В каждом варианте использования участвуют одно или несколько действующих лиц и система. В варианте использования *купить напиток* участвует лицо *Клиент*, а в варианте использования *провести плановый ремонт* участвует лицо *Ремонтник*. В телефонной системе вариант использования *позвонить* включает два действующих лица: *Звонящего* и *Отвечающего*. Совсем не обязательно, чтобы все эти лица были людьми. Вариант использования *заключить сделку* на сетевой бирже включает лицо *Клиент* и лицо *Фондовая биржа*. Система биржевого маклера должна взаимодействовать с обоими действующими лицами, чтобы заключить сделку.

Вариант использования подразумевает обмен последовательностью сообщений между системой и действующими лицами. Например, в варианте использования *купить напиток* *Клиент* сначала вставляет монету, а торговый автомат отображает текущий аванс. Эта процедура может быть повторена несколько раз. Затем клиент нажимает кнопку выбора товара, а автомат выдает заказанный товар и сдачу, если таковая имеется.

- **Купить напиток.** Торговый автомат выдает напиток после того, как клиент выбирает нужный вариант и платит за него.
- **Провести плановый ремонт.** Ремонтник выполняет плановое техобслуживание автомата, необходимое для обеспечения его безотказной работы.
- **Провести техническое обслуживание.** Ремонтник выполняет незапланированное обслуживание автомата при выходе его из строя.
- **Загрузить продукты.** Обслуживающий персонал загружает продукты в торговый автомат для пополнения запасов продаваемых напитков.

Рис. 7.1. Варианты использования торгового автомата

Некоторые варианты использования характеризуются фиксированной последовательностью сообщений. Однако чаще последовательность сообщений может варьироваться в определенных пределах. Например, клиент может кинуть в автомат некоторое количество монет (вариант использования *купить напиток*). В зависимости от количества и номинала монет, а также от выбранного товара автомат может либо вернуть сдачу, либо нет. Вариации последовательностей можно показать, продемонстрировав несколько примеров различающихся между собой последовательностей поведения. Обычно сначала определяется основная последовательность, а затем — необязательные последовательности, повторения и другие вариации.

Частью варианта использования являются и сбойные ситуации. Например, если клиент выбирает напиток, который уже кончился, торговый автомат отображает соответствующее сообщение. Кроме того, клиент может отменить транзакцию покупки. Например, он может нажать кнопку возврата монет в любой

момент до того, как его выбор будет принят. В этом случае автомат возвращает монеты клиента, и последовательность поведения для варианта использования завершается. С точки зрения пользователя некоторые виды поведения могут считаться ошибками. Однако проектировщик должен учитывать все возможные последовательности. С точки зрения системы ошибки пользователя или ресурсов являются разновидностями поведения, которые устойчивая система должна быть способна обработать.

Вариант использования объединяет все поведение, имеющее отношение к элементу функциональности системы: нормальное поведение, вариации нормального поведения, исключительные ситуации, сбойные ситуации и отмены запросов. На рис. 7.2 представлено подробное описание варианта использования Покупка напитка. Объединение нормального и аномального поведения в одном варианте использования помогает убедиться, что все последовательности взаимодействия рассматриваются вместе.

Вариант использования: Покупка напитка.

Краткое описание: Торговый автомат выдает напиток после того, как клиент выбирает нужный вариант и платит за него.

Действующие лица: Клиент.

Исходные условия: Автомат ожидает опускания монеты.

Описание: Автомат изначально находится в состоянии ожидания и выводит на дисплей сообщение «Опустите монеты». Клиент опускает монеты в щель автомата. Автомат выводит на дисплей принятую от клиента сумму и включает подсветку кнопок с названиями напитка. Автомат выдает соответствующий напиток и выдает сдачу, если напиток стоит меньше, чем заплатил клиент.

Исключения:

Отмена: Если клиент нажмет кнопку отмены до того, как произведет выбор напитка, автомат вернет клиенту деньги и перейдет в состояние ожидания.

Напиток отсутствует: Если клиент выбирает напиток, который в данный момент отсутствует в автомате, выводится сообщение: «Напиток отсутствует». Автомат готов к приему монет и выбору напитка клиентом.

Недостаточно денег: Если клиент выбирает напиток, который стоит больше, чем он заплатил, выводится сообщение: «Необходимо доплатить *nn.nn* руб. для покупки этого напитка», где *nn.nn* — недостающая сумма. Автомат продолжает принимать монеты и готов к выбору напитка клиентом.

Нет сдачи: Если клиент вставил достаточное количество монет для покупки напитка, но в автомате нет денег, чтобы корректно выдать сдачу, выводит сообщение «Невозможно выдать сдачу». Автомат продолжает принимать монеты и готов к выбору напитка клиентом.

Постусловия: Автомат готов к приему монет.

Рис. 7.2. Описание варианта использования

В полной модели варианты использования образуют разбиение функциональности системы. Все варианты должны находиться на сравнимом уровне абстракции. Например, варианты использования *позвонить по телефону* и *записать сообщение голосовой почты* не подходят на одинаковых уровнях абстракции, чего нельзя сказать о варианте использования *установить громкость внешнего динамика на высокую*. Лучше было бы назвать этот вариант *установить громкость динамика* (выбор уровня громкости будет частью варианта использования) или даже *установить параметры телефона* (в этом варианте можно установить громкость, отобразить параметры клавиатуры, установить часы и т. д.).

7.1.3. Диаграммы вариантов использования

Любая система обладает своим множеством вариантов использования и множеством действующих лиц. Каждый вариант использования описывает элемент предоставляемой системой функциональности. Множество вариантов использования описывает всю функциональность системы на некотором уровне абстракции. Каждое действующее лицо представляет собой один вид объектов, для которых система может выполнять некоторое поведение. Множество действующих лиц описывает полное множество объектов, которые могут быть обслужены системой. Объекты аккумулируют поведение всех систем, с которыми они взаимодействуют в качестве действующих лиц.

Язык UML предусматривает систему графических обозначений для вариантов использования (рис. 7.3). Варианты использования системы заключаются в прямоугольник, снаружи которого изображаются действующие лица. Название системы может быть указано около одного из краев прямоугольника. Вариант использования обозначается эллипсом, внутри которого указывается его название. Значок «человечек» обозначает действующее лицо. Его имя ставится рядом со значком или под ним. Действующие лица соединяются с вариантами использования сплошными линиями.

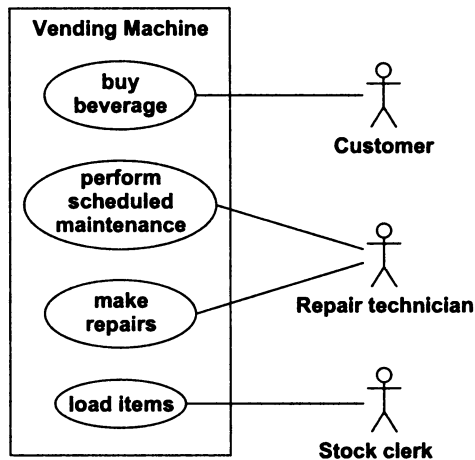


Рис. 7.3. Варианты использования для торгового автомата

На нашем рисунке лицо (*Repair technician*) участвует в двух вариантах использования, а остальные лица — каждый в своем. В одном варианте использования могут участвовать несколько действующих лиц (в нашем примере этого нет).

7.1.4. Руководство к вариантам использования

Варианты использования идентифицируют функциональность системы и упорядочивают ее с точки зрения пользователей. Традиционные списки требований могут описывать функциональность, не вполне ясную для пользователей, а также пропускать вспомогательную функциональность (например, инициализацию и завершение). Варианты использования описывают только полные транзакции, а потому создают меньше шансов что-нибудь пропустить. Традиционные списки требований все равно могут использоваться для описания глобальных ограничений и другой нелокализованной функциональности, например средней наработки на отказ или общей пропускной способности. Однако большую часть взаимодействий с пользователями следует описывать именно при помощи вариантов использования. Основное назначение системы почти всегда отражается в вариантах использования, а списки требований задают дополнительные ограничения на реализацию. Вот некоторые рекомендации, касающиеся конструирования моделей вариантов использования.

- **Прежде всего следует определить границы системы.** Невозможно идентифицировать варианты использования и действующие лица, если граница системы определена нечетко.
- **Убедитесь в сосредоточенности действующих лиц.** Каждое лицо должно иметь одну явно выраженную цель. Если объект из реального мира сочетает в себе несколько целей, их следует представлять разными действующими лицами. Например, владелец персонального компьютера может устанавливать программное обеспечение, настраивать базу данных и отправлять электронную почту. Эти функции существенно отличаются друг от друга по влиянию на компьютерную систему и по возможностям ее повреждения. Они могут быть разделены между тремя действующими лицами: администратором системы, администратором базы данных и обычным пользователем. Помните, что действующее лицо определяется только по отношению к системе, а не как некоторая самостоятельная сущность.
- **Каждый вариант использования должен предоставлять пользователям значения.** Вариант использования должен описывать полную транзакцию, предоставляющую пользователям некоторые значения и обладающую не слишком узким определением. Например, *набрать телефонный номер* — не самый подходящий вариант использования для телефонной системы. Он не является полной транзакцией и не предоставляет пользователю значений. Это всего лишь часть варианта использования *позвонить*. Последний включает в себя вызов, разговор и завершение вызова. Работая с полными вариантами использования, мы сосредоточиваем внимание на назначении

функциональности системы и не принимаем никаких преждевременных решений о реализации. Подробности добавляются позже. Часто требуемая функциональность может быть реализована несколькими способами.

- **Связывайте варианты использования с действующими лицами.** Каждый вариант использования должен иметь по крайней мере одно действующее лицо. Каждое действующее лицо должно принимать участие по крайней мере в одном варианте использования. Вариант использования может иметь несколько действующих лиц, действующее лицо может участвовать в нескольких вариантах использования.
- **Помните о том, что варианты использования не являются формальным описанием системы.** Не стоит добиваться формальности описания вариантов использования. Они были придуманы для того, чтобы идентифицировать и упорядочивать функциональность системы с точки зрения пользователя. Варианты использования на первом этапе вполне могут быть определены недостаточно четко. Подробности добавляются позже, когда варианты использования расширяются и реализуются.
- **Варианты использования можно структурировать.** Для многих приложений вполне достаточно отдельных вариантов использования. В больших системах можно конструировать варианты использования из отдельных фрагментов, используя отношения между ними (см. главу 8).

7.2. Модели последовательности

Модель последовательности представляет собой углубленное рассмотрение ситуаций, описываемых вариантами использования. Такие модели бывают двух типов: сценарии и диаграммы последовательности. Последние обладают более четкой структурой.

7.2.1. Сценарии

Сценарий (scenario) — это последовательность событий, осуществляющихся в процессе одного конкретного выполнения системы, например в соответствии с каким-либо вариантом использования. Сценарий может иметь различную область охвата: он может описывать все события системы или только те, которые влияют на конкретные объекты или порождаются ими. Сценарий можно рассматривать как журнал выполнения фактической системы или как мысленный эксперимент, в котором участвует предлагаемая система.

Сценарий может быть представлен в виде текста (рис. 7.4). В нашем примере Джон Доу входит в сетевую брокерскую систему, размещает заказ на приобретение акций General Electric, после чего отключается. Через некоторое время после выполнения заказа фондовая биржа сообщает результаты сделки брокерской системе. Джон Доу увидит результаты выполнения своего заказа при следующем входе в систему, но наш сценарий это не описывает.

Джон Доу входит в систему.
 Система устанавливает безопасные соединения.
 Система выводит информацию по портфолио.
 Джон Доу вводит заказ на покупку 100 акций GE по рыночной цене.
 Система проверяет наличие необходимых средств для совершения сделки.
 Система выводит запрос подтверждения и предполагаемую стоимость.
 Джон Доу подтверждает сделку.
 Система размещает заказ на фондовой бирже.
 Система выводит номер транзакции.
 Джон Доу выходит из системы.
 Система устанавливает незащищенное соединение.
 Система выводит сообщение о завершении сеанса.
 Фондовая биржа сообщает о результатах торгов.

Рис. 7.4. Сценарий для он-лайновой брокерской сделки

Наш пример отражает взаимодействие на самом высоком уровне. Такой этап, как *Джон Доу входит в Систему*, может потребовать обмена несколькими сообщениями между *Джоном Доу* и *Системой*. Однако суть этапа сводится к тому, что Джон Доу запрашивает систему о входе и предоставляет необходимые идентификационные данные. Детали реализации можно показать отдельно. На ранних стадиях разработки вы должны описывать сценарии в терминах высоких уровней. На более поздних стадиях можно показывать отдельные сообщения. Определение порядка обмена сообщений относится к проектированию.

Сценарий содержит информацию о сообщениях, которыми обмениваются объекты, а также о деятельности, выполняемой объектами. Каждое сообщение передает информацию от одного объекта другому. Например, *Джон Доу входит в Систему* передает сообщение от Джона Доу брокерской системе. Написание сценария начинается с идентификации объектов, обменивающихся сообщениями. После этого определяются отправитель и получатель каждого сообщения, а также последовательность сообщений. Наконец, можно добавить в сценарий деятельность, соответствующую внутренним вычислениям. Это делается уже на этапе кодирования.

7.2.2. Диаграммы последовательности

Текстовые сценарии удобно писать, но они не позволяют наглядно показать отправителя и получателя каждого сообщения, особенно если количество объектов превышает два. Диаграмма последовательности показывает участников взаимодействия и последовательность сообщений, которыми они обмениваются. Она описывает взаимодействие системы с действующими лицами в процессе полного или частичного выполнения варианта использования.

На рис. 7.5 показана диаграмма последовательности, соответствующая описанному выше сценарию сделки на бирже. Каждое действующее лицо, как и система, обозначается вертикальной линией, которая называется линией жизни, а каждое сообщение — горизонтальной стрелкой, направленной от отправителя

к получателю. Время направлено сверху вниз, однако масштаб на диаграмме не соблюдается. Диаграмма показывает только последовательность сообщений, а не точный их порядок во времени. В системах реального времени можно накладывать ограничения на последовательности событий, однако это требует дополнительных обозначений. Обратите внимание, что на диаграммах последовательности могут изображаться и параллельные сигналы: *брокерская система* отправляет сообщения *клиенту* и *фондовой бирже* параллельно. Кроме того, сигналы между участниками взаимодействия не обязаны чередоваться: *брокерская система* отправляет *защищенное сообщение*, за которым следует *отображение портфеля ценных бумаг*.

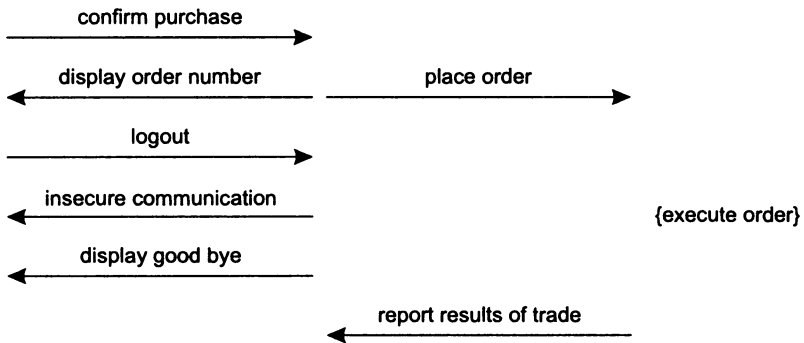


Рис. 7.5. Диаграмма последовательности для он-лайнowej брокерской сделки

Описание поведения каждого варианта использования требует по крайней мере одной диаграммы последовательности. Каждая диаграмма последовательности показывает конкретную последовательность поведения варианта использования. Лучше всего изображать на диаграмме определенную часть варианта использования и не стремиться к максимальной общности. На диаграмме последовательности можно показывать условия, но обычно модель получается яснее, если для каждого существенного потока управления строится своя диаграмма последовательности.

Диаграммы последовательности могут описывать крупномасштабные взаимодействия, такие как сеанс работы с сетевой брокерской системой, но достаточно часто взаимодействия такого рода содержат независимые задачи, которые могут комбинироваться разными способами. Вместо того чтобы повторять одно и то же для разных комбинаций, вы можете нарисовать отдельную диаграмму последовательности для каждой задачи. На рис. 7.6 и 7.7 показан заказ на приобретение пакета акций и запрос на котировку пакета. Эти задачи, вместе со многими другими, являются составляющими сеанса работы с сетевой брокерской системой как целого.

Обязательно нужно строить диаграмму последовательности для каждой исключительной ситуации, возможной для данного варианта использования. В качестве примера на рис. 7.8 изображена ситуация, в которой у клиента оказалось

недостаточно средств для размещения заказа. В этом случае клиент отменяет заказ. Однако он мог бы изменить количество приобретаемых акций, и тогда заказ был бы принят.

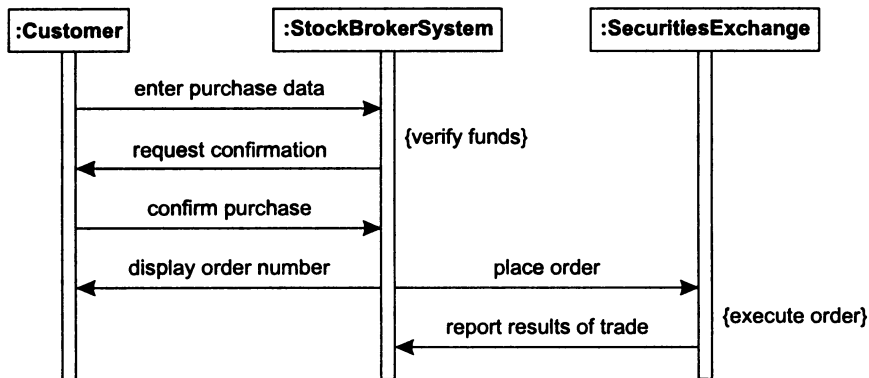


Рис. 7.6. Диаграмма последовательности совершения покупки акций

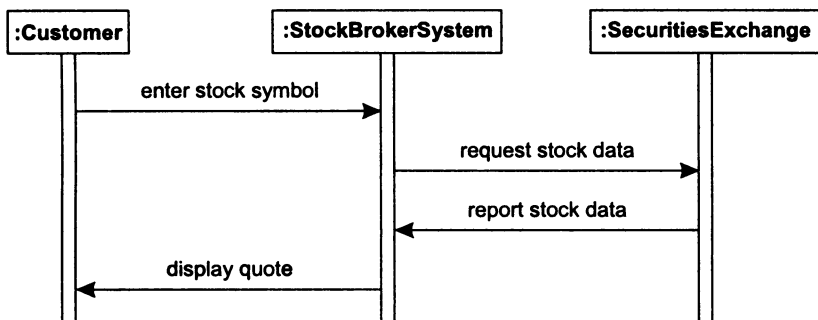


Рис. 7.7. Диаграмма последовательности запроса котировок

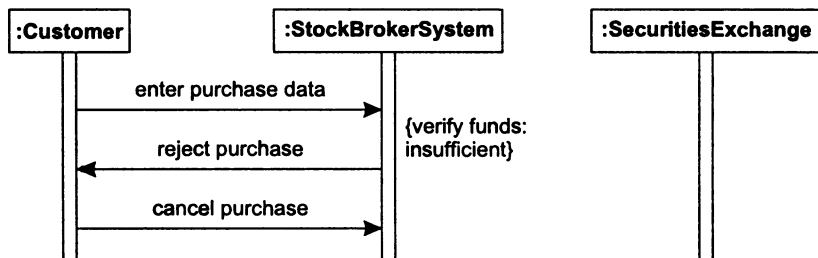


Рис. 7.8. Диаграмма последовательности неудачной покупки

В большинстве систем количество возможных сценариев бесконечно, поэтому показать их все на диаграммах последовательности невозможно. Однако нужно стараться проработать все варианты использования и описать основные виды поведения при помощи диаграмм последовательности. Например, в биржевой системе покупки, продажи и запросы котировок могут чередоваться между собой

произвольным образом. Необязательно показывать все варианты деятельности, достаточно рассмотреть один из основных вариантов.

7.2.3. Руководство к диаграммам последовательности

Модель последовательности прорабатывает и детализирует неформальное описание системы, заданное вариантами использования. Модели последовательности бывают двух видов. Сценарии представляют собой текстовое описание последовательности событий. Диаграммы последовательности тоже описывают последовательности событий, но в графической форме, что позволяет наглядно показать участников взаимодействия. При построении моделей последовательности можно руководствоваться следующими соображениями:

- **Для каждого варианта использования нужно написать по крайней мере один сценарий.** Этапами сценария должны быть логические команды, а не отдельные щелчки мыши и нажатия на клавиши. Точный синтаксис ввода может быть описан позднее, на этапе реализации. Начинайте с простейших взаимодействий: никаких повторов, одна основная деятельность, наиболее типичные значения всех параметров. Если основных вариантов несколько и они существенно отличаются друг от друга, напишите сценарий для каждого из них.
- **Абстрагируйте сценарии при помощи диаграмм последовательности.** Диаграмма последовательности наглядно показывает участие каждого действующего лица. Очень важно выделить вклад каждого из них, чтобы впоследствии организовать поведение объектов.
- **Разбивайте сложные взаимодействия на более простые.** Разделяйте крупные последовательности на составляющие задачи, для которых строите диаграммы последовательности.
- **Стройте диаграмму последовательности для каждой исключительной или сбойной ситуации.** Покажите реакцию системы на нестандартное воздействие.

7.3. Модели деятельности

Диаграмма деятельности показывает последовательность этапов, образующих сложный процесс, например вычислительный алгоритм или технологический процесс. Диаграмма деятельности показывает поток управления, подобно диаграмме последовательности, но сосредоточивает внимание на операциях, а не на объектах. Диаграммы деятельности особенно полезны на ранних этапах проектирования алгоритмов и технологических процессов.

На рис. 7.9 показана диаграмма деятельности обработки заказа на покупку акций, принятого сетевой брокерской системой. Прямоугольники со скругленными углами — это виды деятельности, порядок которых обозначается стрелками. Ромбик обозначает точку принятия решения, а сплошная полоса показывает разделение или слияние параллельных потоков управления.

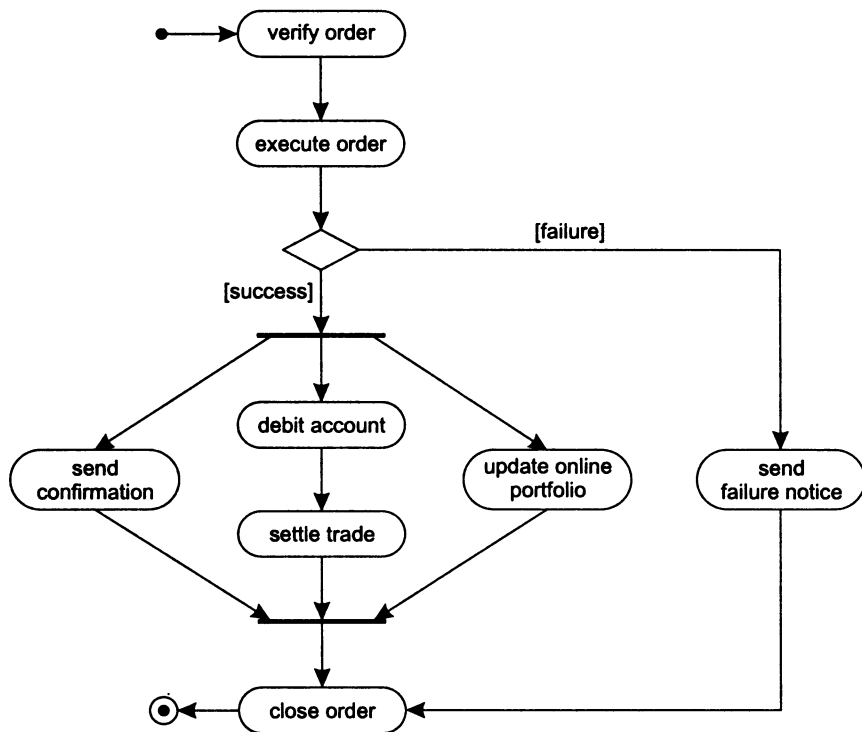


Рис. 7.9. Диаграмма деятельности брокерской системы при обработке заказа

Сетевая брокерская система начинает с проверки заказа. Затем заказ реализуется на фондовой бирже. Если выполнение заказа проходит успешно, система выполняет три действия одновременно: отправляет клиенту подтверждение, обновляет сетевой портфель ценных бумаг с учетом результатов сделки и заканчивает сделку со второй стороной, снимая средства со счета клиента и перечисляя наличные или ценные бумаги. Когда все три параллельных потока завершаются, система объединяет их в один поток и закрывает заказ.

Если же выполнение заказа оказывается неудачным, система отправляет сообщение об этом клиенту и тоже закрывает заказ.

Диаграмма деятельности похожа на обычную блок-схему, потому что тоже показывает поток управления от этапа к этапу. Однако в отличие от блок-схемы, диаграмма деятельности может показывать одновременно параллельную и последовательную деятельность. Это отличие важно для распределенных систем. Диаграммы деятельности часто используются для моделирования организаций, потому что последние состоят из множества объектов (людей и организационных единиц), одновременно выполняющих множество операций.

7.3.1. Деятельность

Элементами диаграммы деятельности являются операции, а именно виды деятельности из модели состояний. Назначение диаграммы деятельности состоит

в том, чтобы показать этапы сложного процесса и упорядочивающие ограничения, на них наложенные.

Некоторые виды деятельности выполняются до тех пор, пока не будут прерваны каким-либо внешним событием, однако в большинстве случаев деятельность имеет собственное логическое завершение. Завершение деятельности является событием завершения, которое обычно означает возможность начала выполнения следующей деятельности. Стрелка без надписи, соединяющая две деятельности на диаграмме, означает, что вторая деятельность может начаться только после того, как закончится первая.

Деятельность может быть разложена на более мелкие составляющие. На рис. 7.10 мы показываем внутреннюю структуру деятельности *execute order* (выполнить заказ) с рис. 7.9. Важно, чтобы виды деятельности на одной диаграмме относились к одному и тому же уровню детализации. Например, на рис. 7.9 деятельности *execute order* (выполнить заказ) и *settle trade* (завершить сделку) относятся к одному уровню детализации: они оба отражают операцию высокого уровня, не уточняя ее механизмы. Если бы один из этих этапов был бы разбит на более мелкие шаги, другие этапы также следовало бы разбить на мелкие шаги, чтобы все они остались на одинаковых уровнях. Сохранить уровень детализации можно, если показывать составляющие высокоуровневой деятельности на отдельных диаграммах.

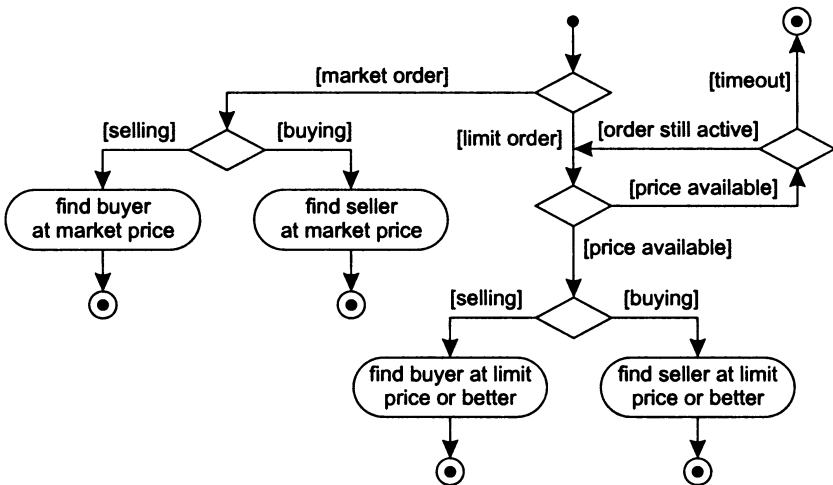


Рис. 7.10. Диаграмма деятельности для действия *execute order*

7.3.2. Ветвление

Если какая-либо деятельность имеет несколько последующих элементов, около каждой стрелки можно указать условие в квадратных скобках, например *[failure]* ([отказ]). Условия проверяются после завершения предшествующей деятельности. Если одно условие оказывается выполненным, стрелка, около которой оно стоит, указывает на деятельность, подлежащую выполнению. Если не удовлетворено ни

одно из условий, модель можно считать плохо согласованной, потому что система зависнет, если только не получит прерывание на более высоком уровне. Во избежание подобных ситуаций следует использовать условие *[else]* ([иначе]). Если выполнено несколько условий, запущена будет только одна деятельность, причем невозможно определить, какая именно. В некоторых случаях подобная неопределенность закладывается осознанно, но часто она указывает на ошибку разработчика, который должен был учесть возможность перекрытия условий.

Для удобства в качестве обозначения множественного ветвления используется ромб с расходящимися от него стрелками, однако смысл его тот же, что и у стрелок, расходящихся от предшествующего состояния. На рис. 7.9 такой ромб имеет одну входящую стрелку и две исходящие, каждая из которых снабжена условием. При фактическом выполнении системы выбирается только одна из этих стрелок.

Если несколько стрелок сходятся у некоторой деятельности, альтернативные пути выполнения сливаются. Здесь снова можно использовать ромб со сходящимися к нему стрелками.

7.3.3. Инициализация и завершение

Сплошной кружок с исходящей стрелкой обозначает начало выполнения диаграммы деятельности. Когда диаграмма деятельности активируется, управление передается на этот кружок, а затем переходит по исходящей из него стрелке к первым этапам деятельности. Завершение диаграммы обозначается символом «бычий глаз» (сплошной кружок с кольцом вокруг него). Этот символ не может иметь исходящих стрелок. Когда управление достигает «бычьего глаза», деятельность завершается и выполнение диаграммы заканчивается.

7.3.4. Параллельная деятельность

В отличие от традиционных блок-схем организации и компьютерные системы могут выполнять несколько видов деятельности одновременно. Скорость выполнения деятельности может меняться с течением времени. Например, за одной деятельностью может следовать другая (последовательное управление), после чего может выполняться несколько параллельных видов деятельности (разделение управления), а затем они снова объединятся в одну деятельность (слияние управления). Развилка и слияние обозначаются толстой линией с одной или несколькими входящими стрелками и одной или несколькими исходящими стрелками. Для синхронизации необходимо, чтобы управление было передано в точку слияния по всем стрелкам. На развилке управление передается всем параллельным видам деятельности одновременно.

На рис. 7.9 показан пример разделения и слияния управления. После выполнения заказа нужно выполнить несколько задач, причем порядок их выполнения может быть произвольным. Брокерская система должна отправить подтверждение клиенту, снять деньги с его счета и обновить содержимое сетевого портфеля ценных бумаг для данного клиента. Когда три параллельные задачи завершаются и сделка заканчивается, происходит слияние управления и выполняется операция закрытия заказа.

7.3.5. Выполняемые диаграммы деятельности

Диаграммы деятельности не только полезны для определения этапов сложного процесса. Они могут быть использованы для демонстрации управления выполнением. Около выполняемого символа деятельности можно поместить *маркер деятельности* (activity token). Когда деятельность завершается, маркер помещается около исходящей стрелки. В простейшем случае на следующем шаге маркер оказывается около следующего символа деятельности.

Если исходящих стрелок несколько и они снабжены условиями, последующая деятельность определяется проверкой этих условий. Маркер может перейти только к одному из последующих видов деятельности, даже если одновременно выполняются условия у нескольких стрелок. Если ни одно условие не выполняется, диаграмма деятельности является плохо согласованной.

Параллельное выполнение соответствует появлению нескольких маркеров. Если за выполняемой деятельностью следует разделение управления, завершение деятельности приводит к увеличению количества маркеров: у каждого из параллельно выполняемых видов деятельности помещается свой маркер. Слияние управления приводит к уменьшению количества маркеров. Вся предшествующая деятельность должна быть завершена до выполнения слияния.

7.3.6. Руководство к моделям деятельности

Диаграммы деятельности уточняют ход вычислений. Таким образом, они представляют собой описание этапов, необходимых для реализации операции или бизнес-процесса. Кроме того, диаграммы деятельности помогают разработчикам понимать сложные вычисления благодаря графическому изображению хода выполнения с промежуточными этапами. В этом разделе мы приводим некоторые рекомендации, касающиеся моделей деятельности.

- **Избегайте использования диаграмм деятельности не по их прямому назначению.** Диаграммы деятельности предназначены для уточнения вариантов использования и моделей последовательности. С их помощью разработчики изучают алгоритмы и технологические процессы. Диаграммы деятельности представляют собой вспомогательное средство в рамках объектно-ориентированного подхода и не должны использоваться для разработки программного обеспечения по блок-схемам.
- **Диаграммы должны быть одноуровневыми.** На одной диаграмме должны находиться виды деятельности, относящиеся к одному уровню детализации. Подробное содержание какой-либо деятельности следует выносить на отдельную диаграмму.
- **Будьте аккуратны с ветвлениями и условиями.** Если на диаграмме указаны условия, по крайней мере одно из них должно в любом случае быть выполнено. Для этого можно воспользоваться условием *else* (иначе). В недетерминированных моделях одновременно могут быть выполнены

несколько условий, однако чаще всего такая ситуация указывает на наличие ошибки.

- **Будьте аккуратны с параллельным выполнением.** Параллельность означает допустимость произвольного порядка завершения всех видов деятельности. Слияние возможно только после полного их завершения.
- **Рассмотрите возможность создания выполняемых диаграмм деятельности.** Выполняемая диаграмма поможет разработчикам лучше понять систему. Иногда она может быть полезна и для конечных пользователей.

7.4. Резюме

Модель взаимодействия дает единое представление поведения: взаимодействия объектов и обмена сообщениями. На высшем уровне модели функциональность системы разбивается на дискретные элементы, имеющие смысл для внешних действующих лиц, при помощи вариантов использования. Поведение вариантов использования можно уточнять при помощи сценариев и диаграмм последовательности. Диаграммы последовательности показывают взаимодействующие объекты и сообщения, которыми они обмениваются. Диаграммы деятельности позволяют отобразить ход вычислений.

Модели классов, состояний и взаимодействия основаны на одних и тех же концепциях (данные, последовательность, операции), но каждая модель сосредоточена на своем аспекте этих концепций. Для полного понимания задачи необходимы все три модели, хотя относительная важность каждой из них зависит от приложения. Три модели объединяются в процессе реализации методов, для чего требуются данные (целевой объект, аргументы, переменные), управление (упорядочивающие конструкции) и взаимодействия (сообщения, вызовы, последовательности).

Таблица 7.1. Ключевые понятия главы

деятельность	параллелизм	сценарий	вариант использования
диаграмма деятельности	модель взаимодействия	диаграмма последовательности	диаграмма вариантов использования
маркер деятельности	линия жизни	граница системы	
действующее лицо	сообщение	поток	

Библиографические замечания

Концепция вариантов использования была впервые предложена Якобсоном [Jacobson-92]. В первом издании этой книги рассказывалось о сценариях и диаграммах трассировки событий. Последние эквивалентны простым диаграммам последовательности.

Ссылки

[Jacobson-92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. Object-Oriented Software Engineering: A Use Case Driven Approach. Wokingham, England: Addison-Wesley, 1992.

Упражнения

7.1. В этой задаче рассматривается книжный магазин, например торговый пассаж.

- 1) (2) Перечислите три действующих лица, задействованных в схеме системы проверки на выходе. Объясните важность каждого из них.
- 2) (2) Один из вариантов использования называется покупка товаров. Рассмотрите другой вариант использования с точки зрения покупателя на аналогичном уровне абстрагирования. Опишите назначение каждого варианта использования одним предложением.
- 3) (4) Подготовьте диаграмму вариантов использования для системы проверки на выходе книжного магазина.
- 4) (3) Подготовьте типичный сценарий для каждого варианта использования. Помните, что сценарий — это пример, в котором не обязательно задействуется вся функциональность варианта использования.
- 5) (3) Подготовьте сценарий исключительной ситуации для каждого варианта использования.
- 6) (5) Подготовьте диаграмму последовательности для каждого сценария из пункта 4.

7.2. В этой задаче рассматривается компьютерная система электронной почты.

- 1) (2) Перечислите три действующих лица. Объясните важность каждого из них.
- 2) (2) Одним из вариантов использования является получение электронной почты. Перечислите еще четыре варианта использования, имеющие тот же уровень абстрагирования. Опишите назначение каждого из них одним предложением.
- 3) (4) Подготовьте диаграмму вариантов использования компьютерной системы электронной почты.
- 4) (3) Подготовьте типичный сценарий для каждого варианта использования. Помните, что сценарий — это пример, в котором не обязательно задействуется вся функциональность варианта использования.
- 5) (3) Подготовьте сценарий исключительной ситуации для каждого варианта использования.
- 6) (5) Подготовьте диаграмму последовательности для каждого сценария из пункта 4.

- 7.3. В этой задаче рассматривается интернет-система бронирования авиабилетов. Чтобы выполнить задание, вы можете изучить работу реальных веб-сайтов разных авиакомпаний.
- 1) (2) Перечислите два действующих лица. Объясните важность каждого из них.
 - 2) (2) Одним из вариантов использования является бронирование билета на рейс. Перечислите четыре варианта использования, имеющие тот же уровень абстрагирования. Опишите назначение каждого из них одним предложением.
 - 3) (4) Подготовьте диаграмму вариантов использования интернет-системы бронирования авиабилетов.
- 7.4. В этой задаче рассматривается программная система, обеспечивающая выдачу книг в библиотеке.
- 1) (2) Перечислите четыре действующих лица. Объясните важность каждого из них.
 - 2) (2) Одним из вариантов использования является взятие книги из библиотеки. Перечислите три варианта использования, имеющие тот же уровень абстрагирования. Опишите назначение каждого из них одним предложением.
 - 3) (4) Подготовьте диаграмму вариантов использования программной системы, обеспечивающей выдачу книг в библиотеке.
- 7.5. (3) Укажите не менее 10 вариантов использования Проводника (Windows Explorer). Достаточно просто написать их названия и описать назначение каждого в одном или двух предложениях.
- 7.6. (3) Напишите сценарии для следующих ситуаций.
- 1) Нужно переправить через реку волка, козу и капусту. За один раз можно перевезти только один объект. Если козу оставить с капустой, она ее съест. Если оставить волка с козой, он ее тоже съест. Подготовьте два сценария, в одном из которых что-то съедают, а в другом все успешно переправляется через реку.
 - 2) Приготовьтесь к поездке на автомобиле с автоматической коробкой передач. Не забудьте про ремень безопасности и стояночный тормоз.
 - 3) Подъем на лифте на верхний этаж.
 - 4) Управление системой автоматического регулирования скорости автомобиля. Сценарий должен включать описание ситуации, когда на дороге присутствует медленно движущееся транспортное средство, из-за чего систему приходится временно отключить, а затем включить снова.
- 7.7. (4) Смеситель для ванной имеет две ручки и рычаг, управляющий потоком воды. С его помощью воду можно направить в разбрызгивающую насадку или через кран в ванную. Когда вы включаете воду, она начинает течь через кран в ванную. Движением рычага вы можете направить ее в душ. Чтобы направить воду в ванную, нужно снова нажать рычаг. Когда вы закрываете

воду, рычаг автоматически переходит в такое положение, чтобы при следующем включении воды она текла через кран. Напишите сценарий, описывающий принятие душа с прерыванием телефонным звонком.

- 7.8. (4) Подготовьте диаграмму деятельности, описывающую вычисление счета в ресторане. За каждое блюдо следует взять определенную плату. Кроме того, с общей суммы взимается налог, а также 18 % надбавка за обслуживание групп более 6 человек. Для групп меньшей численности следует оставить поле для записи чаевых. Следует также учесть предъявленные клиентом дисконтные карты.
- 7.9. (4) Подготовьте диаграмму деятельности для предоставления премий частым клиентам авиакомпаний. В недавнем прошлом TWA начисляла за каждый полет не менее 750 миль премиальных. Держатели золотых и красных карт за каждый полет получали не менее 1000 миль. Держатели золотых карт получали 25 % премию за каждый полет, а держатели красных — 50 % премию.
- 7.10. (5) Подготовьте диаграмму деятельности, описывающую в подробностях процесс входа в систему электронной почты. Обратите внимание на то, что ввод пароля и имени пользователя может осуществляться в произвольном порядке.

Дополнительные вопросы моделирования взаимодействий

8

Модель взаимодействий поддерживает дополнительные средства, которые могут оказаться полезными в некоторых ситуациях. При первом чтении эту главу можно пропустить.

8.1. Отношения вариантов использования

Независимых вариантов использования может быть достаточно для описания простых приложений. Однако крупные приложения требуют структурирования вариантов использования. Составные варианты использования можно конструировать из более примитивных при помощи отношений *include* (включение), *extend* (расширение) и *generalization* (обобщение).

8.1.1. Отношение включения

Отношение включения (*include*) позволяет включить последовательность поведения одного варианта использования в другой вариант использования. Включаемый вариант использования подобен подпрограмме — он описывает поведение, которое, в противном случае, пришлось бы повторять. Часто включаемый фрагмент является осмысленной единицей поведения с точки зрения действующих лиц, однако это условие не является обязательным. Включаемый вариант использования может быть, а может и не быть полезен сам по себе.

Для обозначения отношения включения в UML используется пунктирная стрелка, направленная от исходного (включающего) варианта использования к целевому (включаемому) варианту использования. Над стрелкой ставится ключевое слово «*include*». На рис. 8.1 показан пример использования включаемого варианта использования для сетевой брокерской системы. Для установки защищенного сеанса взаимодействия необходимо проверить пароль пользователя. Кроме того, система проверяет пароль для каждой фондовой сделки. Варианты

использования *secure session* (защищенный сеанс) и *make trade* (заключить сделку) включают использование *validate password* (проверить пароль).

Вариант использования можно включить и в текстовое описание. В этом случае используется синтаксис *include название-варианта-использования*. Включаемый вариант использования можно вставить в конкретное место внутри последовательности поведения включающего варианта использования, подобно тому, как подпрограмму можно вызвать из нужного места основной программы.

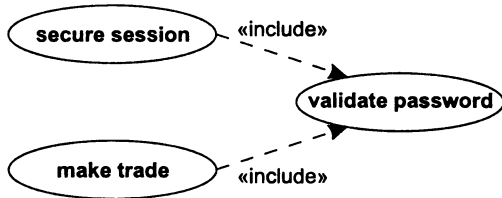


Рис. 8.1. Включение варианта использования

Не стоит использовать отношение включения для структурирования тонкостей поведения. Назначение вариантов использования заключается в том, чтобы идентифицировать функциональность системы и общий поток управления между действующими лицами и системой. Деление варианта использования на части имеет смысл в том случае, если эти части представляют собой значимые элементы поведения.

8.1.2. Отношение расширения

Отношение расширения (*extend*) добавляет к варианту использования дополнительное поведение. Оно аналогично отношению включения, рассматриваемому с противоположной точки зрения. В случае расширения один вариант использования (расширяющий) добавляет себя к другому варианту использования (базовому), тогда как при включении один вариант использования явным образом включает другой. Расширение описывает типичную ситуацию: изначально определяются некоторые возможности системы, к которым затем добавляются новые функции. И отношение расширения, и отношение включения добавляют поведение к базовому варианту использования.

В качестве примера рассмотрим брокерскую систему, поддерживающую базовый вариант использования *купить ценные бумаги*, позволяющий клиенту приобрести акции за средства, имеющиеся на его счете. Расширяющий вариант использования *купить с маржой* добавляет возможность осуществления займа для приобретения ценных бумаг в том случае, если на счете не имеется достаточного количества средств. Клиент все еще может приобретать ценные бумаги за имеющиеся средства, но в том случае, если их не хватает, система предлагает ему продолжить транзакцию (если он согласен заключить сделку с маржой). Дополнительное поведение вставляется в точке сравнения стоимости приобретаемых ценных бумаг с объемом средств на счете клиента.

На рис. 8.2 показан базовый вариант использования *trade stocks* (купить ценные бумаги). Отношение расширения обозначается в UML пунктирной стрелкой,

направленной от расширяющего варианта использования к базовому варианту использования. Над стрелкой ставится ключевое слово «*extend*». Базовый вариант использования позволяет выполнять простые покупки и продажи акций по рыночной цене. Брокерская система добавляет три возможности: сделка с маржой, продажа отсутствующих в наличии ценных бумаг, установка ограничения на цену транзакции. Вариант использования *trade options* (заключить сделку с опционами) также расширяется вариантом, позволяющим установить ограничение по цене транзакции.

Отношение расширения связывает расширяющий вариант использования с базовым. Расширяющий вариант использования часто представляет собой фрагмент, который не может присутствовать в модели как отдельная последовательность поведения. Однако базовый вариант использования должен быть вполне корректным и самостоятельным, допускающим применение без всяких расширений. Отношение расширения может указывать точку вставки дополнений внутри последовательности поведения базового варианта использования. Эта точка может соответствовать одному элементу последовательности или диапазону этих элементов. Последовательность поведения расширяющего варианта использования осуществляется в указанной точке последовательности поведения базового варианта. В большинстве случаев отношение расширения снабжается некоторым прикрепленным к нему условием. Расширяющее поведение выполняется только в том случае, если это условие оказывается истинным в тот момент, когда выполнение доходит до точки вставки расширения.

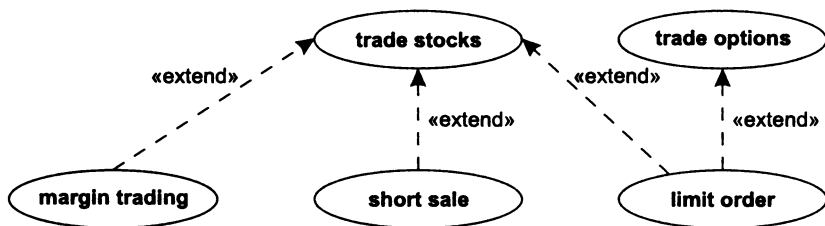


Рис. 8.2. Расширение вариантов использования

8.1.3. Обобщение

Отношение *обобщения* позволяет описывать вариации базового варианта использования, как и обобщение для классов. Вариант использования, являющийся предком, описывает общую последовательность поведения. Его потомки конкретизируют поведение предка, добавляя новые элементы в его последовательность или уточняя существующие. Обобщение вариантов использования в UML обозначается стрелкой, хвост которой находится у потомка, а треугольная стрелка указывает на предка. Та же система обозначений применяется и для классов.

В качестве примера мы снова рассмотрим сетевую брокерскую систему (рис. 8.3), которая может конкретизировать общий вариант использования *make trade* (заключить сделку) потомками *trade bonds* (заключить сделку с облигациями), *trade stocks* (заключить сделку с акциями) и *trade options* (заключить сделку с опционами). Вариант использования, являющийся предком, содержит элементы

поведения, выполняемые для любой сделки (например, ввод пароля для сделки). Каждый из потомков содержит дополнительные элементы, характерные для конкретного типа сделки, например ввод даты истечения опциона.

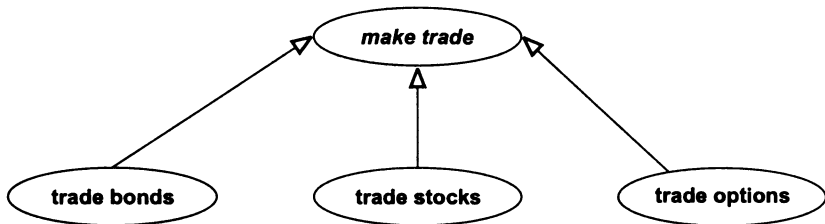


Рис. 8.3. Обобщение вариантов использования

Вариант-предок может быть как абстрактным, так и конкретным. Абстрактный вариант использования не может использоваться непосредственно. Мы не рекомендуем включать в модель конкретные варианты использования, являющиеся предками других вариантов использования. Благодаря этому модель становится более симметричной, а вариант-предок не перегружается обработкой особых ситуаций. Варианты использования обладают полиморфизмом: потомок варианта использования может свободно подставляться вместо него самого, например, включаться вместо него в третий вариант использования. Почти во всех отношениях обобщение вариантов использования ведет себя так же, как и обобщение классов.

Но существует и исключение. Подкласс добавляет собственные атрибуты к атрибутам суперкласса, но порядок этих атрибутов значения не имеет. Вариант-потомок также добавляет элементы поведения, однако они должны присутствовать в правильных местах внутри последовательности элементов поведения его предка. Можно провести аналогию с переопределением метода, унаследованного подклассом, с добавлением новых операторов в различных местах метода предка. Простейший подход заключается в простом перечислении всей последовательности поведения варианта-потомка, включая элементы, унаследованные от его предка. Более универсальный подход состоит в том, чтобы указать символьные обозначения различных точек внутри варианта-предка, после чего, при помощи этих обозначений, разместить внутри него дополнения, внесенные потомком. В общем случае потомок может изменять подпоследовательности поведения предка сразу в нескольких точках.

Классы поддерживают концепцию множественного наследования, однако для вариантов использования это создало бы слишком большие сложности. На практике потребность во множественном наследовании для вариантов использования компенсируется наличием отношений расширения и включения.

8.1.4. Комбинации отношений вариантов использования

На одной диаграмме могут сочетаться различные отношения вариантов использования. На рис. 8.4 мы приводим пример диаграммы вариантов использования для сетевой брокерской системы. Вариант использования *secure session* (защищенный сеанс) включает поведение вариантов *validate password* (проверить

пароль), *make trade* (заключить сделку) и *manage account* (управление счетом). Вариант *make trade* (заключить сделку) является абстрактным предком с конкретными потомками: *trade bonds* (заключить сделку с облигациями), *trade stocks* (заключить сделку с акциями) и *trade options* (заключить сделку с опционами). Вариант использования *make trade* включает вариант *validate password* (проверить пароль). Пароль проверяется один раз для сеанса в целом и для каждой сделки в отдельности.

Вариант использования *margin trading* (заключить сделку с маржой) расширяет варианты *trade bonds* (заключить сделку с облигациями) и *trade stocks* (заключить сделку с акциями), потому что клиент может купить ценные бумаги обоих видов с маржой (но не опционы). Вариант использования *limit order* (ограничить цену) расширяет абстрактный вариант использования *make trade* (заключить сделку), потому что ограничение по цене применимо ко всем видам ценных бумаг. Мы предполагаем, что продать отсутствующие в наличии (*short trade*) ценные бумаги можно только в том случае, если эти ценные бумаги являются акциями, но не облигациями и не опционами.

Обратите внимание, что действующее лицо *Customer* (Клиент) соединено только с вариантом использования *secure session* (защищенный сеанс). Все остальные варианты использования вызываются косвенно, посредством отношений включения, специализации и расширения. Действующее лицо *Securities exchange* (Биржа ценных бумаг) соединяется с вариантом использования *make trade* (заключить сделку). Это лицо не инициирует вариант использования, но участвует в его выполнении.

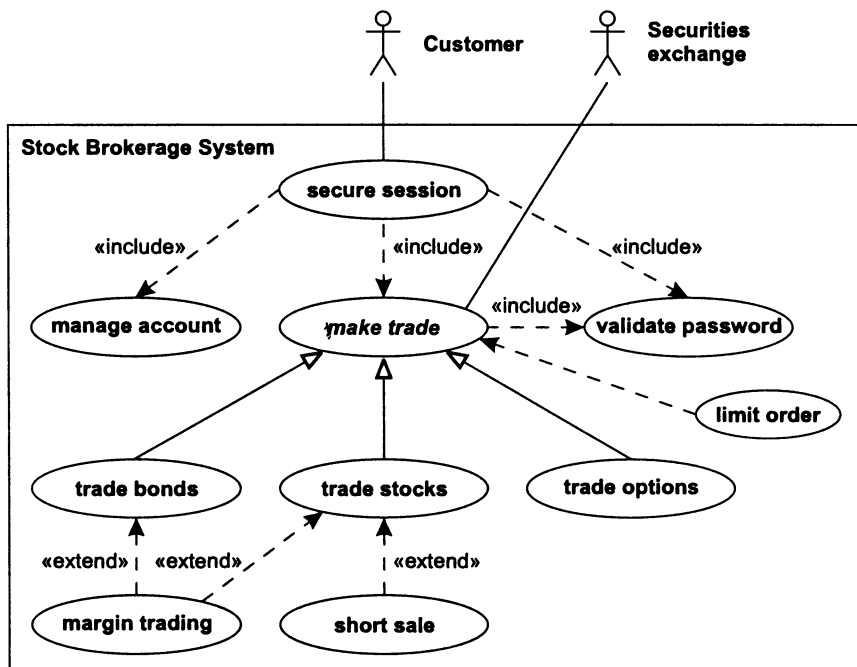


Рис. 8.4. Отношения вариантов использования

8.1.5. Руководство по применению отношений к вариантам использования

Не следует доводить до абсурда отношения между вариантами использования и переходить к программированию. Варианты использования предназначены для формализации и прояснения требований. Требования могут быть реализованы множеством способов, и разработчику не стоит переходить к выбору подхода, пока он не понял суть задачи. В этом разделе мы приводим рекомендации по применению отношений между вариантами использования.

- **Обобщение вариантов использования.** Если вариант использования встречается в модели в нескольких разновидностях, следует выделить общее поведение в абстрактный вариант использования, а затем конкретизировать каждую из разновидностей. Не стоит применять обобщение только для того, чтобы описать общий фрагмент поведения. Для этого предназначено отношение включения.
- **Включение вариантов использования.** Если вариант использования включает хорошо определенный фрагмент поведения, который может оказаться полезным и в других ситуациях, можно выделить этот фрагмент в отдельный вариант использования и включить его в исходный вариант использования. В большинстве случаев включаемый вариант должен быть осмысленным, но не самодостаточным. Например, проверка пароля имеет смысл для пользователей, но цель ее становится ясной только в контексте более широкого варианта использования.
- **Расширение вариантов использования.** Если вы можете определить осмысленный вариант использования с необязательными чертами, выделите базовое поведение в вариант использования и добавляйте к нему эти черты посредством отношения расширения. Это позволяет тестировать и отлаживать систему без расширений, которые могут быть добавлены впоследствии. Применяйте расширение вариантов использования в тех случаях, когда система может быть поставлена в различной конфигурации: либо с дополнительными функциями, либо без них.
- **Расширение и включение.** Оба эти отношения позволяют разбивать поведение на мелкие фрагменты. Отношение включения подразумевает, что включаемое поведение является обязательной частью системы (даже в том случае, если оно выполняется не каждый раз), тогда как отношение расширения подразумевает, что система будет иметь смысл и без добавленного поведения (даже если в ваши намерения не входит поставлять ее в такой конфигурации).

8.2. Процедурные модели последовательности

В главе 7 мы рассматривали диаграммы последовательности с независимыми объектами, которые одновременно находились в активном состоянии. Объект оставался активным после отправки сообщения и мог реагировать на другие сообщения, не

ожидаю ответа на свое. Такой подход применим к высокоуровневым моделям. Однако большинство реализаций являются процедурными. Они ограничивают количество объектов, которые могут выполняться одновременно. UML позволяет показывать процедурные вызовы на диаграммах последовательности при помощи специальных обозначений.

8.2.1. Диаграммы последовательности с пассивными объектами

Наличие процедур в коде предполагает, что не все объекты находятся в состоянии активности одновременно. Большинство объектов пассивны и не имеют собственных потоков управления. Пассивный объект не активируется до тех пор, пока он не будет вызван. Как только выполнение операции завершается и управление возвращается вызывающему, пассивный объект снова перестает быть активным.

На рис. 8.5 приведен пример диаграммы последовательности вычисления комиссии за транзакцию на фондовой бирже. Объект транзакции получает запрос на вычисление своей комиссии. Он получает уровень обслуживания клиента из его таблицы, затем определяет комиссию по этому уровню при помощи таблицы ставок, а затем возвращает значение комиссии вызывающему.

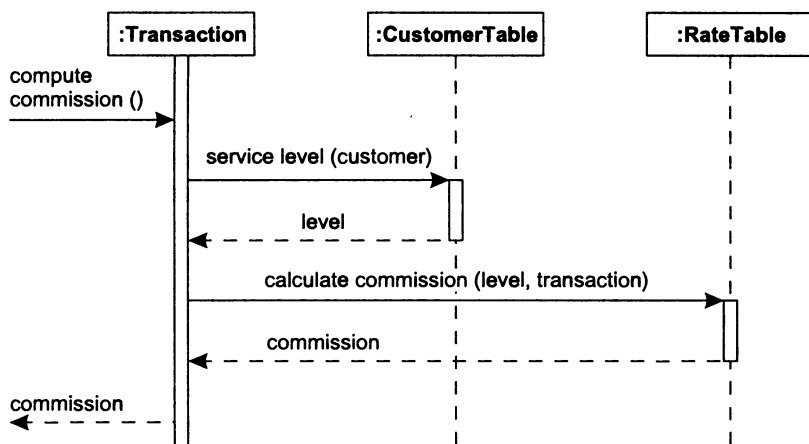


Рис. 8.5. Диаграммы последовательности с пассивными объектами

Период времени выполнения объекта обозначается в UML тонким прямоугольником. Это называется активацией или *фокусом управления* (focus of control). Активация показывает период времени, в течение которого осуществляется выполнение вызванного метода (даже если при этом вызываются другие операции). Период, в течение которого объект существует, но не является активным, обозначается пунктирной линией. Весь период существования объекта называется *линией жизни* (lifeline), которая показывает *время жизни* объекта (lifetime).

8.2.2. Диаграммы последовательности с временными объектами

Расширения системы обозначений, применяемые для описания процедурных вызовов, показаны на рис. 8.6. Объект *objectA* является активным и инициирует операцию. Поскольку он активен, его прямоугольник перекрывает весь временной период, отраженный на диаграмме. Объект *objectB* является пассивным, он существует весь период времени, отраженный на диаграмме, но проводит в активном состоянии только часть этого времени. Его существование обозначается пунктирной линией (линией жизни), перекрывающей всю диаграмму. Когда объект *objectB* обрабатывает вызов, его линия жизни расширяется и становится прямоугольником. Часть этого времени объект *objectB* выполняет рекурсивную операцию, на что указывает двойной прямоугольник активации между вызовом операции *operationE* объектом *objectC* и возвращением значения. Объект *objectC* создается и уничтожается в течение периода времени, изображенного на диаграмме, поэтому его линия жизни не перекрывает диаграмму целиком.

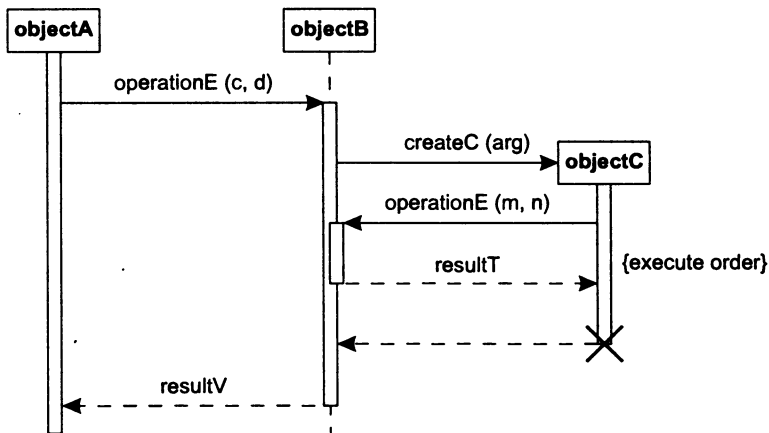


Рис. 8.6. Диаграмма последовательности с временным объектом

Вызов обозначается стрелкой, которая проводится от вызывающего активного объекта к объекту, который становится активным после его вызова. Хвост стрелки находится у края прямоугольника вызывающего активного объекта. Острый конец стрелки указывает на верхний край прямоугольника объекта, который становится активным, потому что именно вызов активирует его. Жирная стрелка обозначает передачу вызова (в противоположность тонкой стрелке, обозначающей сигнал — см. главу 7).

Возврат обозначается в UML пунктирной стрелкой, соединяющей нижний край прямоугольника вызванного (активированного) объекта с боковым краем вызвавшего объекта. Не каждый возврат из вызова сопровождается возвращаемым значением (например, возврат из *objectC* в *objectB*). Таким образом, активация

обозначается стрелкой вызова, указывающей на ее начало, и стрелкой возврата, исходящей из ее конца. Активированный объект может вызывать другие объекты. Стрелки возврата из вызова можно не ставить на диаграмме, потому что они всегда исходят из нижнего края прямоугольника, однако для ясности лучше их ставить.

Если объект не существует на момент начала диаграммы последовательности, он должен создаваться внутри нее. Создание обозначается в UML значком объекта, который ставится у конца стрелки вызова, создающего объект. Например, вызов *createC* создает объект *objectC*. Новый объект может и не получить управление после своего создания. В нашем примере *objectC* получает управление, на что показывает прямоугольник активации, начинающийся сразу после прямоугольника объекта.

Большой значок *X* обозначает конец существования объекта, уничтожаемого в период, отображенный на диаграмме последовательности. Этот значок ставится около конца стрелки вызова, уничтожающего объект. Если объект уничтожает себя сам и передает управление другому объекту, *X* ставится около начала исходящей стрелки. Линия жизни объекта не должна выходить за символы, обозначающие создание и уничтожение этого объекта.

Вызов второго метода того же объекта (включая рекурсивный вызов) обозначается в UML стрелкой, идущей от прямоугольника активности к верхнему краю другого прямоугольника, который накладывается на первый. В нашем примере второй вызов *operationE* объекта *objectB* является рекурсивным (он вложен в первый вызов *operationE*). Второй прямоугольник немного сдвигается в сторону относительно первого. Количество накладываемых прямоугольников показывает количество активаций одного и того же объекта.

На диаграмме последовательности можно показывать и условные переходы, но система обозначений для них слишком сложна, чтобы включать ее в нашу книгу. Подробнее см. [Rumbaugh-05].

8.2.3. Руководство к процедурным моделям последовательности

В этом разделе мы приводим дополнительные рекомендации по разработке процедурных моделей последовательности, не вошедшие в главу 7.

- **Активные и пассивные объекты.** Нужно понимать разницу между активными и пассивными объектами. Большинство объектов являются пассивными и не имеют собственного потока управления. Активные объекты, по определению, всегда активны и обладают собственным фокусом управления.
- **Дополнительные средства.** Дополнительные средства показывают возможную реализацию на диаграммах последовательности. Используйте их с осторожностью. Детали реализации стоит показывать только для особенно сложных или наиболее важных диаграмм последовательности.

8.3. Специальные конструкции для моделей деятельности

Диаграммы деятельности имеют свою собственную систему обозначений, удобную для больших и сложных приложений.

8.3.1. Отправка и получение сигналов

Рассмотрим рабочую станцию, которую включает пользователь. Она выполняет процедуру загрузки, после чего предлагает пользователю войти в систему. После ввода имени и пароля рабочая станция запрашивает подтверждение данных пользователя по сети. После подтверждения рабочая станция завершает процедуру загрузки. Соответствующая диаграмма деятельности показана на рис. 8.7.

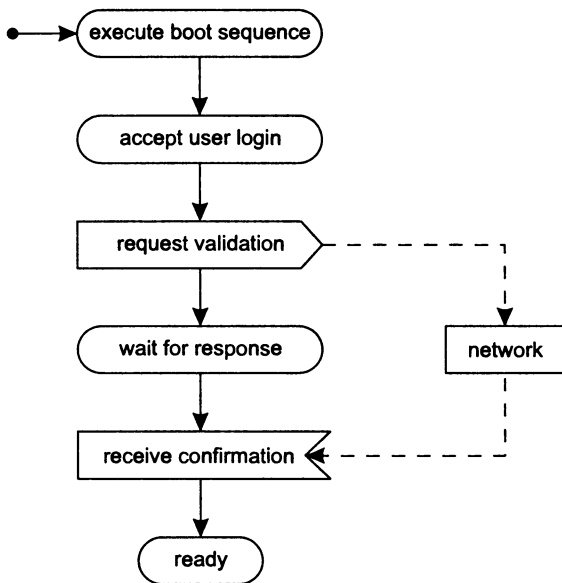


Рис. 8.7. Диаграмма деятельности с сигналами

Отправка сигнала обозначается в UML выпуклым пятиугольником. Когда завершается предшествующая деятельность, происходит отправка сигнала, после чего начинается последующая деятельность. Получение сигнала обозначается вогнутым пятиугольником. Эта конструкция означает ожидание получения сигнала, после чего начинается выполнение последующей деятельности.

8.3.2. Плавающие дорожки

В бизнес-моделях часто бывает полезно представлять, какое именно организационное подразделение отвечает за ту или иную деятельность. В качестве примеров подразделений можно привести отделы продаж, финансов, маркетинга

и снабжения. Когда проектирование системы завершается, деятельность приписывается конкретному человеку, но на высоком уровне достаточно распределить обязанности между организационными подразделениями.

Разделение такого рода можно показать на диаграмме деятельности при помощи столбцов и строк. Столбцы называются «плавательными дорожками». Помещение деятельности внутрь плавательной дорожки означает, что она выполняется человеком или людьми, входящими в состав организационного подразделения. Линии, пересекающие границы плавательных дорожек, обозначают взаимодействия между разными организационными подразделениями, к которым следует относиться с большим вниманием, нежели к взаимодействиям внутри подразделений. Взаимный порядок плавательных дорожек не несет смысловой нагрузки, но в некоторых ситуациях тем не менее может иметь значение.

На рис. 8.8 показан пример диаграммы деятельности по обслуживанию самолета. Дежурные по рейсу должны убрать мусор, наземная команда должна заправить баки топливом, команда снабжения должна погрузить в самолет еду и напитки. Только после этого самолет будет готов к следующему рейсу.

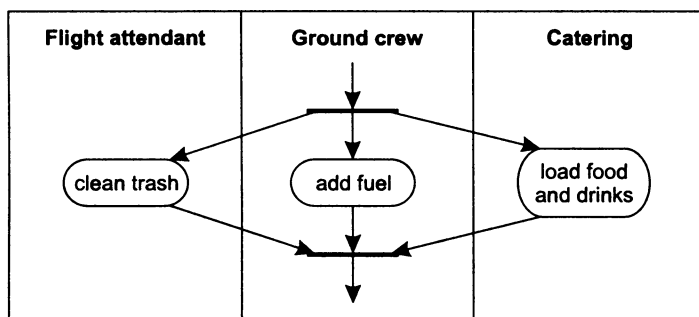


Рис. 8.8. Диаграмма деятельности с плавательными дорожками

8.3. Потоки объектов

Иногда бывает полезно показать отношения между операцией и объектами, являющимися ее аргументами или результатами. На диаграмме деятельности можно показывать объекты, поступающие на входы отдельных видов деятельности и выходящие из них. Входящая или исходящая стрелка подразумевает направление потока управления, поэтому не обязательно изображать поток управления отдельно в том случае, если на диаграмме показан поток объектов.

Часто один и тот же объект проходит через несколько состояний в процессе выполнения диаграммы деятельности. Один и тот же объект может быть входным по отношению к некоторым видам деятельности и выходным по отношению к другим. При детальном рассмотрении обычно оказывается, что деятельность использует или производит объект, находящийся в строго определенном состоянии. Значение объекта в конкретном состоянии обозначается в UML именем состояния, которое указывается в квадратных скобках после имени объекта. Если вместе с объектами на диаграмме показаны названия их состояний, эта диаграмма будет показывать одновременно поток управления и изменение состояний

объектов при воздействии на них различных видов деятельности. На рис. 8.9 самолет проходит через несколько состояний по мере того, как он проходит через ворота, взлетает, а затем снова садится.

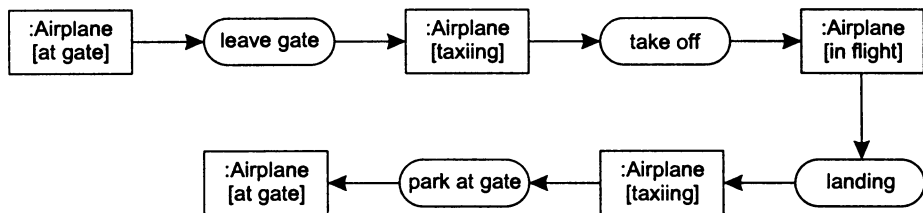


Рис. 8.9. Диаграмма деятельности с потоками объектов

Диаграмма деятельности с потоками объектов и указанием состояний обладает большинством преимуществ диаграммы потоков данных и в то же время лишена большинства ее недостатков. В частности, она объединяет потоки данных и управления, тогда как диаграммы потоков данных часто разделяют их.

8.4. Резюме

Независимые варианты использования пригодны только для простых приложений. В крупных приложениях варианты использования нужно структурировать при помощи отношений включения, расширения и обобщения. Отношение включения добавляет один вариант использования внутрь последовательности поведения другого варианта использования, подобно вызову подпрограммы. Отношение расширения добавляет поведение к базовому варианту использования. Обобщение позволяет показать разновидности общего варианта использования, подобно тому, как это делает обобщение классов. Не доводите эти отношения до абсурда. Помните, что варианты использования предназначены для неформального описания системы. Их отношения должны использоваться только для структурирования крупных элементов поведения.

Модели последовательности полезны не только для раскрытия взаимодействий, обеспечивающих варианты использования, но и для указания деталей реализации. Не все объекты модели последовательности должны быть активными и существовать на всем протяжении расчетов. Некоторые объекты являются пассивными и не имеют собственного потока управления. Другие объекты являются временными, они существуют только часть периода, который занимает операция, их использующая.

Существуют дополнительные обозначения для моделей деятельности, применимые в крупных и сложных приложениях. Вы можете показать на диаграмме отправку и получение событий, взаимодействующих с другими объектами, не относящимися к предмету данной диаграммы. Диаграммы деятельности можно делить на плавательные дорожки, показывающие, кто именно отвечает за ту или иную деятельность. Кроме того, вы можете показывать смену состояний объекта и чередование состояний и видов деятельности.

Таблица 8.1. Ключевые понятия главы

активация	пассивный объект	
диаграмма деятельности	диаграмма последовательности	расширение варианта использования
фокус управления	плавательная дорожка	обобщение варианта использования
модель взаимодействия	временный объект	включение варианта использования
линия жизни	вариант использования	

Ссылки

[Rumbaugh-05] James Rumbaugh, Ivar Jacobson, Grady Booch. The Unified Modeling Language Reference Manual, Second Edition. Boston: Addison-Wesley, 2005.

Упражнения

- 8.1. Рассмотрим покупку бензина на автозаправочной станции.
- 1) (4) Подготовьте диаграмму вариантов использования. Обычно клиент платит за бензин наличными. Добавьте отношения расширения, описывающие дополнительное поведение, возникающее, когда клиент платит кредитной картой снаружи или внутри АЗС. Добавьте отношение включения, отражающее необязательную оплату мойки машины.

2) (2) Перечислите действующие лица и докажите их важность.

3) (2) Опишите назначение каждого варианта использования одним предложением.
- 8.2. (5) Вы взаимодействуете с сетевым бюро путешествий и участвуете в перечисленных ниже вариантах использования. Подготовьте диаграмму вариантов использования с отношениями обобщения и включения.
- Купить билеты на самолет. Забронировать билеты и предоставить информацию об оплате и адресе.

◦ Предоставить информацию об оплате. Предоставить сведения о кредитной карте для оплаты выставленных счетов.

◦ Предоставить адрес. Предоставить почтовый и домашний адреса.

◦ Взять машину напрокат. Зарезервировать машину и предоставить информацию об оплате и адресе.

◦ Забронировать номер. Забронировать номер в отеле и предоставить информацию об оплате и адресе.

◦ Сделать покупку. Приобрести путешествие и предоставить информацию об оплате и адресе.

8.3. (7) Рассмотрим сетевую программу, обслуживающую постоянных клиентов авиакомпаний. Ниже перечислены некоторые возможные варианты использования. Подготовьте диаграмму вариантов использования и изобразите на ней соответствующие отношения. Для каждого обобщения вы можете добавить абстрактного предка.

- Посмотреть количество очков. Посмотреть текущее количество очков на счету клиента.
- Подать заявление о недостающих очках. Запросить очки за деятельность, которая не была оценена.
- Изменить адрес. Указать новый почтовый адрес.
- Изменить имя пользователя. Изменить имя пользователя данного счета.
- Изменить пароль. Установить новый пароль.
- Забронировать бесплатный полет. Использовать набранные очки для бронирования бесплатного полета.
- Забронировать бесплатный номер в отеле. Использовать набранные очки для бронирования бесплатного номера в отеле.
- Запросить о выдаче кредитной карты постоянного клиента. Запросить кредитную карту, которая позволяет использовать набранные очки при совершении покупок.
- Проверить цены и маршруты. Найти возможные маршруты и их цены для перелета, подлежащего оплате.
- Проверить возможность бесплатного полета. Проверить возможность бесплатного полета заданным рейсом.

8.4. (8) Рассмотрим программное обеспечение, управляющее коллекцией музыкальных треков. Ниже перечислены некоторые варианты использования. Подготовьте диаграмму вариантов использования и укажите необходимые отношения вариантов использования. Вы можете добавлять абстрактных предков для создаваемых вами обобщений.

- Воспроизвести трек. Добавить трек в конец очереди воспроизведения.
- Воспроизвести библиотеку. Добавить треки из библиотеки в конец очереди воспроизведения.
- Перемешать в случайном порядке. Перемешать в случайном порядке записи в очереди воспроизведения.
- Удалить трек. Удалить трек из музыкальной библиотеки.
- Уничтожить трек. Удалить трек из всех музыкальных библиотек и удалить соответствующий файл.
- Добавить трек. Добавить музыкальный файл в музыкальную библиотеку.
- Создать музыкальную библиотеку. Создать новую пустую музыкальную библиотеку.
- Удалить музыкальную библиотеку.

- Уничтожить музыкальную библиотеку. Уничтожить все треки в этой библиотеке, после чего удалить саму библиотеку.
 - Скопировать диск. Записать в файлы музыку с аудио компакт-диска (CDDA).
 - Создать диск. Прожечь аудио компакт-диск с выбранными треками.
 - Просмотреть треки по названию. Отобразить список треков в музыкальной библиотеке, отсортированный по названию.
 - Просмотреть треки по исполнителю. Отобразить список треков в музыкальной библиотеке, отсортированный по исполнителю.
 - Просмотреть треки по альбому. Отобразить список треков в музыкальной библиотеке, отсортированный по альбому.
 - Просмотреть треки по жанру. Отобразить список треков в музыкальной библиотеке, отсортированный по жанру.
 - Начать воспроизведение. Начать воспроизведение треков из очереди воспроизведения. Если перед этим воспроизведение было остановлено, начать воспроизведение с последнего воспроизводившегося трека, в противном случае начать с начала очереди.
 - Остановить воспроизведение. Прекратить воспроизведение музыки.
- 8.5. (8) Рассмотрим простую систему составления платежных ведомостей. Подготовьте диаграмму вариантов использования и укажите отношения между некоторыми вариантами. Для каждого обобщения вы можете добавить абстрактного предка.
- Добавить удержание. Добавить новый тип удержания для сотрудника и включать его в последующие ведомости.
 - Убрать удержание. Убрать удержание из списка удержаний сотрудника.
 - Просуммировать доходы. Просуммировать все доходы по ведомости.
 - Просуммировать удержания. Просуммировать все удержания по ведомости.
 - Вычислить суммарный чистый заработок. Посчитать разницу между суммарными доходами и суммарными удержаниями по ведомости.
 - Вычислить все пожертвования. Просуммировать все пожертвования на благотворительность по ведомости.
 - Вычислить налоги. Просуммировать все налоги, уплаченные по ведомости.
 - Вычислить отчисления в пенсионный фонд. Просуммировать все отчисления в пенсионные фонды по ведомости.
 - Вычислить прочие удержания. Просуммировать все прочие удержания по ведомости, за исключением благотворительности, налогов и взносов в пенсионный фонд.
 - Изменить имя сотрудника. Изменить записанное имя сотрудника.
 - Изменить адрес сотрудника. Изменить записанный адрес сотрудника.

- Вычислить тарифную ставку оплаты. Вычислить тарифную ставку оплаты сотрудника в данной ведомости.
- Вычислить оплату за сверхурочные. Вычислить всю оплату за сверхурочные для данного сотрудника по ведомости.
- Вычислить прочие выплаты. Просуммировать все прочие доходы сотрудника (помимо тарифной оплаты и сверхурочных) по данной ведомости.
- Изменить способ оплаты. Изменить метод оплаты ведомости: наличные, прямое зачисление в депозит по ведомости, выдача чека.

8.6. (4) Рассмотрим брокерскую программу, записывающую все транзакции для данного портфеля. Транзакции могут описывать, к примеру, покупку и продажу акций, получение дивидендов, а также сложные ситуации, подобные разделению акций.

Текущее содержимое портфеля можно определить, просмотрев журнал транзакций. Портфель характеризуется начальным содержимым, а все последующие изменения этого содержимого описываются только записями в журнале транзакций. Для определения текущего содержимого к начальному содержимому применяются все записи из журнала по текущую дату.

Сконструируйте процедурную диаграмму последовательности, показывающую порядок вычисления содержимого портфеля на конкретное число. На диаграмме должно присутствовать не более четырех потоков сообщений.

8.7. (5) Рассчитайте стоимость портфеля на конкретное число. Для этого сначала определите содержимое портфеля на эту дату (см. предыдущее упражнение), после чего умножьте количество акций каждого типа на их стоимость на соответствующее число и просуммируйте.

8.8. (7) В этом упражнении нужно снова рассчитать стоимость портфеля на конкретное число. Однако в этом случае портфель может содержать не только акции, но и другие портфели. Для простоты предположите, что глубина вложенности портфелей не может превышать трех.

Например, портфель суммарный может содержать портфели пенсионных фондов и облагаемый налогом счет. Эти портфели в нашем примере содержат только акции.

8.9. (6) Дама принимает решение модернизировать свой компьютер и купить DVD-проигрыватель. Она начинает со звонка в отдел продаж производителя компьютера, откуда ее перенаправляют в службу поддержки. Она звонит в службу поддержки, где ее переводят в режим ожидания на время разговора с инженерами. Наконец, служба поддержки сообщает ей о возможных вариантах установки DVD-проигрывателя. Дама выбирает модель проигрывателя и заказывает доставку почтой. Она получает проигрыватель, успешно устанавливает его в компьютер, после чего отправляет по почте оплаченный счет.

Нарисуйте диаграмму деятельности для этого процесса. Распределите ответственность при помощи плавательных дорожек. Покажите взаимодействия между подразделениями.

- 8.10. (6) Компания производит новый продукт и должна координировать несколько отделов. Продукт начинает свое существование с маркетинговой идеи, которая передается в проектный отдел. Проектный отдел моделирует функции продукта и подготавливает проект. Отдел производства изучает проект и корректирует его, приводя в соответствие с имеющимся оборудованием. Проектный отдел принимает изменения, после чего проект изучает служба поддержки: хороший проект должен подразумевать удобство ремонта. Проектный отдел принимает предложения службы поддержки и проверяет, что после корректировок проект все еще удовлетворяет требованиям, предъявленным к целевой функциональности.

Нарисуйте диаграмму деятельности для этого процесса. Покажите зоны ответственности при помощи плавательных дорожек. Покажите изменения состояния проекта по мере выполнения деятельности.

Обзор концепций

Нам кажется полезным моделировать системы с трех связанных между собой точек зрения: модель классов описывает объекты системы и отношения между ними; модель состояний описывает историю существования объектов, модель взаимодействия описывает взаимодействия между ними. Полное описание возможно только при использовании всех трех моделей, однако относительный вклад каждой зависит от конкретных задач. Каждая модель применяется на всех этапах разработки и постепенно обрывает деталями.

9.1. Модель классов

Модель классов описывает статическую структуру объектов системы: их индивидуальность, отношения с другими объектами, их атрибуты и операции. Модель классов является каркасом, в рамках которого определяются модели состояний и взаимодействия. Изменения и взаимодействия не имеют смысла, если отсутствует то, что может изменяться или взаимодействовать. Объекты — это элементы нашего мира, молекулы наших моделей.

Наиболее важными концепциями модели классов являются классы, ассоциации и обобщения. Класс описывает группу подобных объектов. Ассоциация описывает группу подобных связей между объектами. Обобщение структурирует описание объектов, упорядочивая классы по их сходствам и различиям. Атрибуты и операции являются вторичными и служат для уточнения фундаментальной структуры, образуемой классами, ассоциациями и обобщениями.

9.2. Модель состояний

Модель состояний описывает аспекты объекта, изменяющиеся с течением времени: события, отмечающие изменения, и состояния, определяющие контекст событий.

События соответствуют внешним воздействиям, а состояния обозначают значения объектов. С течением времени объекты воздействуют друг на друга, в результате чего их состояния претерпевают последовательные изменения. Модель состояний состоит из множества диаграмм состояний, по одной для каждого класса с нетривиальным поведением. Диаграммы состояний должны быть согласованы по интерфейсам (событиям и сторожевым условиям). Каждая диаграмма состояний показывает события и последовательности состояний, разрешенные для одного класса объектов.

Диаграмма состояний указывает возможные состояния, переходы между состояниями, воздействия, вызывающие эти переходы, операции, выполняемые в ответ на воздействия. Такая диаграмма описывает коллективное поведение объектов класса. Поскольку каждый объект имеет свои собственные значения и связи, то каждый объект имеет и собственное состояние или положение на диаграмме состояний.

9.3. Модель взаимодействия

Модель взаимодействия описывает кооперацию объектов для достижения результатов. Это объединенное представление поведения множества объектов, тогда как модель состояний рассматривает поведение каждого объекта индивидуально. Для полного описания поведения необходимы обе модели. Они дополняют друг друга, поскольку рассматривают поведение с двух различных точек.

Взаимодействия можно моделировать на различных уровнях абстракции. На высшем уровне взаимодействие системы с внешними действующими лицами описывается при помощи вариантов использования. Варианты использования соответствуют элементам функциональности и помогают описать неформализованные требования. Диаграммы последовательности содержат больше подробностей и показывают взаимодействующие объекты и временную последовательность их взаимодействия. Диаграммы деятельности содержат больше всего мелких деталей и показывают поток управления между этапами вычислений.

9.4. Отношения между моделями

Модели классов, состояний и взаимодействия оперируют одними и теми же концепциями: данными, последовательностями и операциями, но каждая модель фокусируется на своем аспекте и ничего не говорит о прочих аспектах. Все три модели необходимы для полного описания задачи. Все три модели объединяются при реализации методов, для которой требуются данные (целевой объект, аргументы, переменные), управление (упорядочивающие конструкции) и взаимодействия (сообщения и вызовы).

Каждая модель описывает один аспект системы, но при этом она содержит ссылки на другие модели. Модель классов описывает структуру данных, с которой работают модели состояний и взаимодействия. Операции в модели классов соответствуют событиям, условиям и видам деятельности. Модель состояний

описывает управляющую структуру объектов. На ней показываются решения, зависящие от значений объектов. Решения вызывают изменения значений объектов и последующих состояний. Модель взаимодействия фокусируется на обмене между объектами и дает единое представление о работе системы в целом.

Отношения обобщения и агрегации действуют, в том числе, и между моделями, поэтому мы рассматриваем их применение в этой главе.

9.4.1. Обобщение

Обобщение присутствует во всех трех моделях. Обобщение — это отношение «или», которое показывает различные вариации общей ситуации. В UML 2.0 наследование применяется к классификаторам, а классы, сигналы и варианты использования являются классификаторами.

- **Обобщение классов.** Обобщение позволяет упорядочить классы по их сходствам и различиям. Подкласс наследует атрибуты, операции, ассоциации и диаграммы состояний от своих суперклассов. Подклассы могут осуществлять повторное использование свойств, унаследованных от суперклассов, или переопределять их. Подклассы также могут добавлять новые свойства. Подкласс наследует диаграммы состояний своих предков, которые должны выполняться параллельно с его собственными диаграммами состояний. Подкласс наследует и состояния предков, и их переходы. Во избежание конфликтов диаграммы состояний подклассов должны быть ортогональными дополнениями к диаграммам состояний суперклассов.
Модель классов поддерживает множественное наследование: класс может быть потомком сразу нескольких суперклассов. Для простоты множественное наследование для сигналов и вариантов использования обычно не разрешается.
- **Обобщение сигналов.** Сигналы также можно упорядочить в иерархию с наследованием атрибутов. Каждый реальный сигнал может считаться листом этого дерева. Сигнал переключает переходы, рассчитанные на любой из его предков.
- **Обобщение вариантов использования.** Обобщение применимо и к вариантам использования. Вариант-предок описывает общую последовательность поведения. Потомки конкретизируют предка, добавляя новые этапы поведения или детализируя имеющиеся. В одном отношении обобщение вариантов использования оказывается сложнее обобщения классов. Подкласс может добавлять атрибуты к атрибутам, унаследованным от предка, однако их порядок не имеет значения. Вариант-потомок может добавлять этапы поведения, но они должны располагаться в конкретных местах между этапами поведения предка.

При наследовании классификатор-предок может быть как абстрактным, так и конкретным. Однако мы рекомендуем вам делать предков только абстрактными. В этом случае будет очевидно, какие классификаторы должны быть абстрактными, а какие конкретными (только листья). Классификаторы, кроме того, обладают свойством полиморфизма: потомок всегда может быть подставлен вместо предка.

В первом издании этой книги было описано наследование состояний, поскольку оно было разрешено, однако в UML2 оно было запрещено, так как состояние не является классификатором. Можно провести некоторые параллели между обобщением классификаторов и вложением состояний, однако, строго говоря, в UML2 обобщение к состояниям неприменимо.

9.4.2. Агрегация

Агрегация — это отношение «и», которое разбивает агрегат на ортогональные части, взаимодействие которых некоторым образом ограничивается.

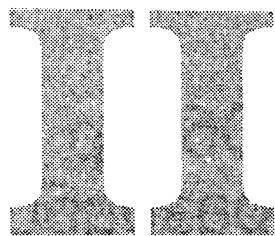
- **Агрегация объектов.** Агрегация — это особая форма ассоциации, которая обладает дополнительными свойствами: транзитивностью и антисимметрией. UML определяет две формы агрегации объектов: общая форма называется собственно агрегацией (составляющая часть может использоваться повторно и может существовать отдельно от агрегата) и композицией (составляющая часть может принадлежать не более чем одному агрегату и имеет одинаковое с ним время жизни).

Диаграмма состояний агрегата является совокупностью диаграмм состояний его частей. Состояние агрегата определяется комбинацией состояний всех его частей. Состояние агрегата — это одно состояние с первой диаграммы, И одно состояние со второй диаграммы, И одно состояние с третьей диаграммы, И так далее. В более сложных случаях состояния частей могут взаимодействовать между собой.

- **Агрегация состояний.** Некоторые состояния можно разбивать на более мелкие, каждое из которых выполняется независимо и имеет собственную поддиаграмму. Состояние объекта включает в себя одно состояние с каждой из диаграмм.

Эта глава заканчивает рассмотрение концепций и обозначений, применяемых в объектно-ориентированном моделировании.

Анализ и проектирование



В первой части нашей книги были описаны концепции и обозначения, относящиеся к моделям классов, состояний и взаимодействия. Во второй и третьей частях мы сосредоточим свое внимание на процессе формирования моделей. В первой части речь шла о том, из чего модель состоит, а во второй и третьей части о том, каким образом можно сформулировать модель. Наш подход к процессу проектирования не зависит от языка реализации и одинаково хорошо применим к объектно-ориентированным языкам, не объектно-ориентированным языкам и базам данных.

В главе 10 мы приводим обзор процесса построения моделей и подчеркиваем, что разработка обычно представляет собой итерационную деятельность, а не жесткую последовательность шагов.

Глава 11 посвящена первой стадии разработки — концептуализации системы, — на которой автор придумывает приложение и продает его идею организации.

Имея концепцию приложения, вы можете прорабатывать и уточнять ее в процессе формирования моделей. Об этом рассказывают главы 12 и 13. Сначала строится модель области применения, описывающая предметы реального мира, несущие семантику приложения. Затем строится модель приложения, описывающая компьютерные аспекты приложения, видимые пользователям.

Аналитические модели дадут вам полное понимание приложения. На следующем этапе придется перейти к решению практических вопросов реализации моделей. Глава 14 посвящена проектированию системы, то есть выработке высокоуровневой стратегии построения решения. Глава 15 посвящена проектированию классов, в процессе которого определяются особенности классов, ассоциаций и операций.

Глава 16 завершает вторую часть книги, подводя итоги этапам анализа и проектирования процесса разработки.

Изучив вторую часть, вы узнаете основы построения объектно-ориентированных моделей. Вы не станете экспертом, но зложите неплохой фундамент для приобретения ценного опыта по разработке программного обеспечения. Вы будете готовы к изучению вопросов реализации и технологий разработки программного обеспечения, которые рассматриваются в последних двух частях.

Обзор процесса разработки

10

Процесс разработки программного обеспечения (software development process) является основой организованного производства программного обеспечения при помощи совокупности заранее определенных методик и систем обозначений. Описанный в этой книге процесс начинается с формулировки задачи, затем продолжается этапами анализа, проектирования и реализации. Этапы процесса описываются линейно, но на практике процесс редко оказывается последовательным.

10.1. Этапы разработки

Разработка программного обеспечения представляет собой последовательность четко определенных этапов, на каждом из которых решается определенная задача. Каждый этап характеризуется входными и выходными данными.

- **Концептуализация системы.** Придумывается приложение и формулируются пробные требования.
- **Анализ.** Углубленное понимание требований достигается при помощи построения моделей. Цель анализа состоит в указании того, что должно быть сделано (а не того, как это должно быть достигнуто). Нужно понять задачу перед тем, как вы попытаетесь ее решить.
- **Проектирование системы.** Предлагается высокоуровневая стратегия решения задачи создания приложения (архитектура системы), определяющая основы для последующего проектирования классов.
- **Проектирование классов.** Модели реального мира, полученные на этапах анализа, расширяются и корректируются таким образом, чтобы их можно было реализовать на компьютере. Определяются алгоритмы для реализации операций.

- **Реализация.** Проект преобразуется в программный код и структуры баз данных.
- **Тестирование.** Проверяется, что приложение пригодно для практического использования и действительно удовлетворяет поставленным требованиям.
- **Обучение.** Пользователям помогают освоиться с приложением.
- **Развертывание.** Приложение развертывается там, где планируется его применение. Осуществляется переход с унаследованных приложений.
- **Поддержка.** Обеспечивается долгосрочная жизнеспособность приложения.

На самом деле, процесс разработки непрерывен. Модели постоянно уточняются и оптимизируются, даже когда вы переключаетесь с анализа на проектирование, а затем на реализацию. Одни и те же концепции и обозначения используются на всех этапах. Разница в том, что вначале большее внимание уделяется потребностям заказчика, а в конце — компьютерным ресурсам.

Объектно-ориентированный подход переносит основные усилия по разработке программного обеспечения на этапы анализа и проектирования. Иногда кажется неправильным тратить много времени на анализ и проектирование системы, однако эти дополнительные затраты с лихвой окупаются быстрой и простой реализацией. Поскольку конечная система оказывается более ясной и легче приспособляемой, ее проще поддерживать и изменять в соответствии с изменяющейся ситуацией.

Первые три этапа процесса разработки рассматриваются во второй части нашей книги. Третья часть целиком посвящена реализации. Мы подчеркиваем важность разработки и лишь отчасти затрагиваем вопросы тестирования, обучения, развертывания и поддержки. Эти этапы тоже важны, но они не относятся к предмету нашей книги.

10.1.1. Концептуализация системы

Концептуализация системы (system conception) означает рождение приложения. В самом начале его существования кто-то придумывает идею приложения, составляет бизнес-план и продает свою идею какой-либо организации. Человек, который этим занимается, должен представлять как потребности потенциальных клиентов, так и технологические возможности.

10.1.2. Анализ

Анализ (analysis) — это создание моделей. Аналитики формализуют и исследуют требования, конструируя свои модели. Они описывают то, что должно быть сделано, а не то, как это следует сделать. Анализ — это непростая задача. Разработчики должны полностью осознать сложность стоящей перед ними задачи, перед тем как заняться дополнительными сложностями, которые неизбежно возникнут на этапе реализации. Надежные модели — обязательное условие для создания расширяемого, эффективного, надежного и корректного приложения. Никакие исправления на этапах реализации не смогут исправить внутреннюю несогласованность приложения и скомпенсировать недостаток предусмотрительности.

В процессе анализа разработчики используют все доступные источники информации (документы, интервью, аналогичные и связанные приложения) и устраняют возникающие неопределенности. Часто бизнес-эксперты оказываются не в состоянии сформулировать точные требования, и их уточнением приходится заниматься разработчикам. Моделирование ускоряет сходимость представлений разработчиков и бизнес-экспертов, потому что гораздо быстрее работать с различными вариантами моделей, чем с различными реализациями кода. Модели подчеркивают недостатки и несогласованные моменты, которые можно затем исправить. По мере того как разработчики прорабатывают и уточняют модель, она становится все более согласованной.

Этап анализа делится на две стадии. Сначала осуществляется анализ предметной области, а затем — анализ приложения. Анализ предметной области применяется к объектам реального мира, семантику которых охватывает приложение. Например, рейс самолета — это объект реального мира, который должна отражать система бронирования билетов. Объекты предметной области существуют независимо от приложений и имеют смысл для бизнес-экспертов. Эти объекты выявляются во время анализа предметной области или исходя из априорных соображений. Объекты предметной области в модели несут информацию об объектах реального мира и обычно являются пассивными. Анализ предметной области выделяет понятия и отношения, а функциональность при этом неявным образом содержится в модели классов. Конструирование модели предметной области сводится, главным образом, к принятию решений о том, какую информацию следует отразить в модели и как ее нужно представить.

За анализом предметной области следует анализ приложения, который относится к компьютерным аспектам приложения, видимым пользователям. Например, меню бронирования билета является частью системы бронирования авиабилетов. Объекты приложения существуют не в предметной области. Они имеют смысл только в контексте приложения. Однако эти объекты не являются чем-то внутренним по отношению к проекту, потому что их видят пользователи. Объекты приложения должны полностью удовлетворять пользователей. Модель приложения не предписывает конкретной реализации системы. Она описывает внешний вид приложения, которое воспринимается как черный ящик. Классы приложения нельзя определить на этапе анализа, но их часто можно взять из предыдущих приложений. В противном случае объекты приложения приходится придумывать на этапе анализа, как часть интерфейса с другими системами и с пользователями.

10.1.3. Проектирование системы

В процессе *проектирования системы* (system design) разработчик принимает стратегические решения с наиболее широким спектром последствий. Он должен сформулировать архитектуру системы и выбрать глобальные стратегии и политики, определяющие последующие, более детализированные этапы проектирования. Архитектура — это высокоуровневый план, или стратегия решения задачи по созданию приложения. Выбор архитектуры определяется требованиями и предшест-

вующим опытом разработчика. По возможности, архитектура должна включать исполняемый скелет, который можно было бы тестировать. Проектировщик должен представлять, каким образом новая система будет взаимодействовать с существующими системами. Архитектура также должна принимать во внимание возможность последующих изменений системы.

В простых задачах подготовка архитектуры следует после анализа. Однако в крупномасштабных сложных проектах разработка архитектуры и анализ должны чередоваться. Архитектура помогает установить область применимости модели. Моделирование, в свою очередь, раскрывает важные стратегические вопросы, требующие решения. Конструкция модели и архитектура системы связаны друг с другом, а потому они должны строиться параллельно.

10.1.4. Проектирование классов

На этапе *проектирования классов* разработчик расширяет и оптимизирует аналитические модели. Происходит смещение точки приложения усилий с концепций приложения к компьютерным концепциям. Разработчики выбирают алгоритмы реализации основных функций системы, однако они должны воздерживаться от выбора конкретных языков программирования.

10.1.5. Реализация

Реализация (implementation) — это этап написания реального кода. Разработчики реализуют элементы проекта на языках программирования, создавая программный код и структуру базы данных. Достаточно часто код может частично генерироваться автоматически из проектной модели.

10.1.6. Тестирование

После реализации система оказывается завершенной, но перед сдачей ее в эксплуатацию она должна пройти обязательный этап *тестирования*. Идеи, вложенные в исходный проект, претерпели изменения в процессе формирования моделей. Тестеры возвращаются к исходным требованиям бизнес-экспертов и проверяют, что система обладает нужной функциональностью. Тестирование позволяет также раскрыть случайные ошибки («баги»), которые появились в системе в процессе ее создания. Если приложение должно работать на разном оборудовании или под разными операционными системами (на разных платформах), оно должно быть проверено на всех этих платформах.

Разработчики должны проверять программу на нескольких уровнях. Модульные тесты применяются к небольшим частям кода, например к методам или классам. Модульное тестирование позволяет выявить локальные проблемы и часто требует встраивания в код дополнительных средств, применяемых только для тестирования. Системные тесты применяются к крупной подсистеме приложения. В отличие от модульных тестов системные позволяют выявить значительные несоответствия спецификации. Оба вида тестов являются обязательными. Тестирование не следует откладывать до окончания полного кодирования

всего приложения. Его следует планировать с самого начала. Многие тесты могут выполняться в процессе реализации.

10.1.7. Обучение

Организация должна обучать пользователей, чтобы они могли полностью использовать все преимущества нового приложения. Обучение ускоряет усвоение пользователями новых навыков. Отдельная команда должна готовить пользовательскую документацию параллельно с разработкой программного обеспечения. Отдел контроля качества может сравнивать программное обеспечение с пользовательской документацией. Это позволяет гарантировать, что программа соответствует исходным требованиям.

10.1.8. Развертывание

Конечная система должна работать «в поле», на разных платформах в различных конфигурациях. При *развертывании* системы (deployment) на предприятии пользователя можно ожидать самых необычных эффектов от взаимодействия с другими системами. Разработчики должны оптимизировать систему под разные нагрузки и написать сценарии и процедуры установки. Некоторые клиенты потребуют настройки системы под себя. Персонал должен также выполнить локализацию продукта на разные языки с учетом пользовательской *локали* (locale). В результате получается пригодный к использованию *выпуск продукта* (release).

10.1.9. Поддержка

После завершения разработки и развертывания системы команда переходит к этапу *поддержки*. Поддержка подразумевает несколько видов деятельности. В процессе использования постепенно выявляются ошибки, содержащиеся в программном обеспечении. Эти ошибки необходимо исправлять. Успешное приложение вызывает запросы на усовершенствование, а долгоживущее приложение когда-нибудь придется реструктурировать.

Модели облегчают поддержку и передачу дел при смене персонала. Модель отображает бизнес-требования к приложению, воплощенные в реальный код, пользовательский интерфейс и структуру базы данных.

10.2. Жизненный цикл разработки

Объектно-ориентированный подход к разработке программного обеспечения поддерживает несколько вариантов жизненных циклов. Вы можете использовать водопадный подход к этапам анализа, проектирования и реализации, чередуя их в строгой последовательности для всей системы. Однако мы рекомендуем в большинстве случаев использовать итерационную стратегию разработки. В этом разделе мы коротко расскажем об отличиях этих двух стратегий, а более подробные сведения приведем в главе 21.

10.2.1. Водопадная разработка

Водопадный подход требует, чтобы разработчики выполняли этапы разработки в строгой линейной последовательности без всяких возвратов. Сначала описываются требования, затем конструируется аналитическая модель, затем выполняется проектирование системы и классов, после чего осуществляется реализация, тестирование и развертывание. Каждый этап должен полностью завершиться до начала следующего этапа.

Водопадный подход пригоден для определенных приложений с предсказуемыми результатами этапов анализа и проектирования, но такие приложения на практике встречаются редко. Слишком многие организации пытаются следовать водопадному подходу даже в том случае, когда требования к программе изменчивы. В результате получается знакомая многим ситуация: разработчики жалуются на изменение требований, а бизнес-эксперты жалуются на отсутствие гибкости при разработке программного обеспечения. Кроме того, водопадный подход не позволяет получить рабочую систему до того, как будут завершены все его этапы. Это затрудняет оценку выполнения проекта и коррекцию проектов, отклонившихся от первоначальных планов.

10.2.2. Итерационная разработка

Итерационный подход к разработке дает большую гибкость. Сначала разрабатывается ядро системы. Анализ, проектирование, реализация и развертывание дают рабочий код. Затем расширяется область применения системы, добавляются свойства и поведение к имеющимся объектам, создаются новые классы. Система проходит несколько таких итераций до того, как получается готовый продукт.

Каждая итерация подразумевает полное завершение этапов анализа, проектирования, реализации и тестирования. Однако в отличие от водопадного подхода, при итерационном подходе допускается перекрытие различных этапов и не требуется разработка системы целиком за один раз. Некоторые части системы могут быть закончены раньше, тогда как другие, не такие важные, доделываются потом. В конце каждой итерации получается исполняемая система, которая может быть интегрирована с другими системами и протестирована. Это дает возможность точно оценивать прогресс и корректировать планы с учетом информации, полученной на первых итерациях. Если возникают проблемы, всегда можно вернуться к предыдущим этапам работы.

Итерационная разработка хорошо подходит для большинства приложений, потому что она позволяет быстро реагировать на изменения и уменьшает риск провала. Менеджеры и клиенты почти сразу получают интересующие их сведения о прогрессе разработки.

10.3. Резюме

Процесс разработки программного обеспечения является основой организованного производства программного обеспечения. Существует определенная последовательность этапов разработки, которая применима к любой части системы.

Например, параллельные команды разработчиков могут разрабатывать дизайн базы данных, ключевые алгоритмы и пользовательский интерфейс. Итерационный подход характеризуется гибкостью и быстрым откликом на изменение требований. Сначала создается ядро системы, которое затем расширяется до тех пор, пока не получится готовое приложение.

Таблица 10.1. Ключевые понятия главы

анализ	анализ предметной области	концептуализация системы
анализ приложения	реализация	проектирование системы
архитектура	итерационная модель разработки	тестирование
проектирование классов	жизненный цикл	обучение
развертывание	поддержка	водопадная модель разработки

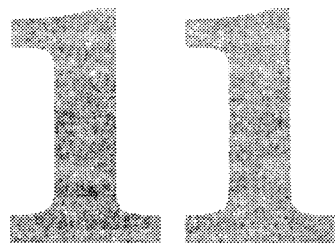
Библиографические заметки

В первом издании этой книги этап проектирования классов назывался этапом проектирования объектов.

Упражнения

1. (2) Никогда не хватает времени на то, чтобы сделать работу правильно с первого раза, но всегда находится время на то, чтобы ее переделать. Обсудите, каким образом подход, предлагаемый в этой главе, позволяет преодолеть особенности человеческого поведения. Какие ошибки вы совершите, если сразу броситесь кодировать новую программу? Сравните усилия, затрачиваемые на предотвращение ошибок, с усилиями, необходимыми для их выявления и устранения.
2. (4) В этой книге рассказывается о том, каким образом объектно-ориентированные технологии применяются для создания программ и баз данных. Обсудите, каким образом их можно было бы применить в других областях, например для проектирования языков, представления знаний и проектирования оборудования.

Концептуализация системы



Концептуализация системы — это зарождение приложения. Все начинается с того, что кто-то один, разбирающийся в потребностях бизнеса и в технологии, формулирует идею приложения. После этого разработчики исследуют идею, пытаются понять потребности потенциальных клиентов и предложить возможные варианты решений. На этапе концептуализации системы не нужно углубляться в детали, нужно понять картину в целом: какой потребности отвечает предлагаемая система? Можно ли ее разработать за приемлемую стоимость? Покроет ли спрос на конечный продукт затраты на его производство?

В этой главе мы рассмотрим конкретный пример: разработку банкомата (АТМ). Этот пример будет изучаться со всех сторон в последующих главах до конца книги.

11.1. Изобретение концепции системы

В большинстве случаев идеи, на которых основываются новые системы, являются продолжением уже существующих идей. Например, в отделе кадров имеется база данных сотрудников, в которую служащий вносит все изменения. Очевидное расширение системы состоит в том, чтобы предоставить сотрудникам возможность самостоятельно просматривать содержимое своих записей и вносить необходимые изменения. Для реализации этой идеи придется решить множество вопросов (защищенность, надежность, конфиденциальность и т. д.), но сама идея является достаточно очевидным продолжением существующей концепции.

Иногда новшество бывает радикальным. Например, сетевой аукцион автоматизирует древнюю идею аукциона — конкуренции покупателей за продукты, но первые системы для проведения сетевых аукционов были абсолютно новыми программами. Концепция стала осуществимой только после одновременного появления нескольких технологий: Интернета, персональных компьютеров, а также надежных серверов. Большое количество покупателей и низкая стоимость в расчете

на единицу товара, достигнутая благодаря автоматизации, изменили природу аукционов: на сетевом аукционе можно продавать даже дешевые товары и все равно извлекать прибыль. Кроме того, сетевые системы сделали аукцион параллельным и распределенным.

Существует несколько способов поиска концепций новых систем.

- **Новая функциональность.** Можно добавить функциональность в существующую систему.
- **Модернизация.** Снятие ограничений или универсализация работы системы.
- **Упрощение.** Предоставление обычным людям возможности заниматься тем, чем раньше занимались только специалисты.
- **Автоматизация.** Автоматизация ручных процессов.
- **Интеграция.** Объединение функциональности различных систем.
- **Аналогии.** Поиск аналогий в других предметных областях и исследование их на наличие полезных идей.
- **Глобализация.** Путешествия в другие страны и изучение их культуры и деловой практики.

11.2. Проработка концепции

Большинство систем начинают свое существование в виде нечетких идей, требующих дальнейшего уточнения. Хорошая концепция системы должна давать ответы на следующие вопросы.

- **Для кого предназначено приложение?** Нужно ясно представлять себе, какие организации и частные лица будут заинтересованы в новой системе. Заинтересованные лица бывают двух основных типов: это спонсоры и конечные пользователи.

Важность спонсоров не подлежит сомнению, потому что они оплачивают новую систему. Они рассчитывают, что проект будет выполняться в соответствии с планом и уложится в бюджет. Вам придется договориться с ними о том, каким образом будет оцениваться успешность выполнения проекта. Вы должны четко представлять себе, в каком состоянии система будет считаться готовой и удовлетворяющей их требованиям.

Пользователи тоже являются заинтересованными лицами, но в другом смысле. Именно пользователь в конечном итоге определяет успех новой системы тем, что его производительность или эффективность повышается (или понижается). Пользователи могут помочь вам, если они будут готовы к сотрудничеству, и не поленятся снабжать вас критическими комментариями. Они могут повысить качество системы, сообщая вам о том, чего в ней недостает, а что может быть усовершенствовано. Вообще говоря, пользователь не будет заниматься новой программой, если он не почувствует заинтересованности в этом (личной или деловой). Нужно постараться заинтересовать своих пользователей, чтобы и они внесли свой вклад в систему.

Если это не удастся, вам стоит задуматься о том, насколько необходим ваш проект и не нужно ли его пересмотреть.

- **Какую задачу приложение будет решать?** Нужно четко ограничить масштабы усилий и установить область их применения. Нужно определить, какими функциями будет обладать новая система, а какими не будет. Вы должны удовлетворить пользователей из разных организаций, обладающих разными точками зрения и политическими мотивациями. Вам нужно не только решить, каких функций будет достаточно, но и получить согласие влиятельных лиц.
- **Где будет использоваться система?** На раннем этапе бывает полезно получить общее представление о том, где может использоваться новая система. Нужно выяснить, будет ли она критической для задач организации, или станет экспериментом, или же это будет просто новая возможность, которую можно будет внедрить, не нарушая последовательность прочих работ. Вы должны получить хотя бы приблизительное представление о том, как новая система будет дополнять уже существующие. Важно знать, будет ли она использоваться на локальных компьютерах или в распределенной сетевой среде. Для коммерческих продуктов обязательно предварительно охарактеризовать контингент заказчиков.
- **Когда будет нужна система?** Для нового приложения важны два аспекта, связанных со временем. Во-первых, это время на осуществление, то есть время, за которое система может быть разработана с учетом ограничений по стоимости и с использованием доступных ресурсов. Во-вторых, это требуемое время, за которое система должна быть разработана, чтобы удовлетворить потребностям бизнеса. Нужно убедиться в том, что оценка времени, полученная с учетом технологических возможностей, соответствует временным потребностям бизнеса. В противном случае нужно начать диалог между технологами и бизнес-экспертами, чтобы найти приемлемое решение.
- **Почему нужна эта система?** Возможно, вам придется приготовить экономическое обоснование системы в том случае, если этого уже не сделал кто-нибудь другой. Экономическое обоснование включает информацию о стоимости, ощутимой выгоде, нематериальной выгоде, рисках и альтернативных вариантах. Вы должны ясно представлять мотивацию разработки новой системы. Экономическое обоснование даст вам представление о том, чего ожидают заинтересованные лица, позволит приблизительно указать область применения системы и даже предоставит некоторую информацию для будущих моделей. В случае коммерческого продукта необходимо оценить количество единиц продукции, которые могут быть проданы, и определить приемлемую цену. Доход от продаж должен покрывать затраты и давать некоторую прибыль.
- **Как она будет работать?** Нужно устроить мозговой штурм задачи технической осуществимости. Если речь идет о большой системе, следует рассмотреть достоинства различных архитектур. На этом этапе нужно не выбрать решение, а выработать у себя уверенность в том, что задача может быть решена. Возможно, потребуется прототипирование и проведение экспериментов.

Пример с банкоматом. На рис. 11.1 приведена наша оригинальная концепция системы банкомата (Automated Teller Machine — АТМ). Чтобы проработать эту концепцию, мы ответим на свои собственные вопросы.

Разработка программного обеспечения, позволяющего клиентам получать доступ к банковской компьютерной системе и выполнять транзакции без участия банковских служащих.

Рис. 11.1. Концепция банкомата

- Для кого предназначено приложение? Банкоматы производятся несколькими компаниями. Поэтому только производитель автомата или крупная компания может компенсировать расходы и трудозатраты на создание программного обеспечения банкомата.

Производителю пришлось бы конкурировать за клиентов на сформировавшемся рынке. Крупный производитель, конечно, сможет войти на такой рынок, но ему может оказаться выгоднее стать партнером существующего поставщика или купить его целиком. Небольшому производителю потребуется придать своему продукту какую-либо отличительную черту, чтобы выделиться из толпы других производителей и привлечь к себе внимание.

Маловероятно, что финансовому предприятию удастся компенсировать затраты на разработку программного обеспечения банкомата для своего собственного использования, потому что это будет дороже, чем купить готовый продукт. Если компании нужен продукт с какими-либо особыми функциями, она могла бы стать партнером производителя. Или же такая компания могла бы создать отдельную организацию, которая создаст программное обеспечение, продаст его спонсору, после чего начнет распространять его на рынке.

В нашем примере мы предполагаем, что наша фирма — это производитель банкомата, который сам разрабатывает для него программное обеспечение. Будем считать, что мы создаем обыкновенный продукт, потому что сложности предметной области банкоматов лежат за рамками нашей книги.

- Какие задачи приложение будет решать? Программное обеспечение банкомата должно работать как на банк, так и на клиента. С точки зрения банка, программное обеспечение повышает уровень автоматизации и сокращает объем ручной канцелярской работы. С точки зрения клиента, банкоматы должны быть распространены повсеместно и обязаны быть круглосуточно доступны. С их помощью клиент должен иметь возможность выполнять повседневные транзакции, где и когда ему удобно. Программное обеспечение банкомата должно быть простым в использовании и удобным, чтобы клиентам было приятнее работать с ним, чем с банковскими кассирами. Система должна быть надежной и защищенной, так как она будет работать с деньгами.
- Где будет использоваться система? Программное обеспечение банкоматов стало насущной необходимостью для финансовых учреждений. Клиенты считают естественным, что у всех банков есть свои банкоматы. Банкоматы установлены во множестве магазинов и в других местах по всему миру.

- Когда будет нужна система? Любой проект по разработке программного обеспечения является деловым предложением. Инвестиция в разработку в конечном итоге приводит к получению доходов. С экономической точки зрения лучше всего сокращать инвестиции, увеличивать доходы и по возможности приближать их получение. Достижению этой цели во многом и служат объектно-ориентированные технологии, применяемые в процессе вдумчивого моделирования.
- Почему нужна эта система? У производителя оборудования может найтись множество причин для создания программного продукта. Если другие компании зарабатывают деньги на аналогичных продуктах, это является достаточно веским основанием для вступления на рынок. Новый продукт может позволить производителю обойти конкурентов и дать ему возможность установить цену выше номинала. Предприятия обычно стараются производить труднодоступные технологии самостоятельно, если они им действительно нужны. Единственной мотивацией разработки программного обеспечения для банкомата у авторов этой книги была необходимость продемонстрировать применение объектно-ориентированных технологий на хорошем примере.
- Как она будет работать? Мы планируем реализовать трехуровневую архитектуру, отделив пользовательский интерфейс от программной логики, а логику — от базы данных. В реальности архитектура будет многоуровневой (с неопределенным числом уровней), потому что промежуточных программных уровней, взаимодействующих друг с другом, может быть сколько угодно. Архитектура будет обсуждаться в главе 14.

11.3. Подготовка задачи к постановке

После того как нечеткая идея кристаллизуется благодаря ответам на перечисленные выше вопросы, вы будете готовы записать требования к системе, которые будут описывать цели и общий подход к созданию системы.

Разработчик должен хорошо представлять разницу между требованиями, проектом и реализацией. Требования описывают, каким образом система должна себя вести с точки зрения пользователя. Система считается черным ящиком. Все, что нас интересует — это ее внешнее поведение. Например, в числе требований к автомобилю может быть такое: когда вы давите на педаль газа, автомобиль начинает ехать быстрее, а когда нажимаете на тормоз — машина замедляет движение. Проект — это принятые решения о том, каким образом должно быть предоставлено поведение, указанное в требованиях. Например, проект описывает способ прокладки внутренних соединений, управление двигателем автомобиля, разновидность тормозных колодок. Реализация — это окончательное воплощение в программном коде.

Часто заказчики путают требования с проектными решениями. Обычно это приводит к плохим последствиям. Отделяя требования от проектных решений, вы сохраняете свободу маневра. Большинство систем могут быть спроектированы

множеством способов, поэтому принятие решения следует отложить до тех пор, пока вы не разберетесь в задаче полностью.

В документе, описывающем концепцию системы, может быть приведен пример реализации. Назначение примера состоит в том, чтобы показать, каким образом система может быть реализована с использованием существующих технологий по приемлемой цене. Это просто «доказательство существования». Однако нужно ясно понимать, что в окончательной системе реализация идей может быть выполнена совсем другим способом. Пример реализации предлагается исключительно для демонстрации возможностей.

Например, в конце 60-х годов XX века был впервые предложен проект «Аполлон» (высадка человека на Луну). В первом его варианте предлагалось вывести ракету на орбиту Земли, затем запустить посадочный аппарат прямо на поверхность Луны. В окончательном варианте программы (который был успешно осуществлен) ракета была выведена на орбиту Луны, а затем с ракеты на поверхность Луны опустился посадочный модуль. Первый вариант был предложен не зря: он дал людям уверенность в том, что проект действительно может быть осуществлен.

Как показано на рис. 11.2, постановка (или формулировка) задачи должна содержать утверждения о том, что следует сделать (а не о том, как это должно быть реализовано). Это должен быть список требований, а не вариант реализации системы. Не нужно описывать внутренности системы, так как это снизит гибкость процесса разработки. В качестве допустимых требований можно привести список эксплуатационных характеристик и протоколы взаимодействия с внешними системами. Также допустимо перечисление стандартов технологии разработки программного обеспечения: модульное устройство, проектирование с расчетом на возможность тестирования и на дальнейшее расширение.

Постановка задачи	Проектирование	Реализация
■ Предметная область ■ Что требуется ■ Прикладной контекст ■ Допущения ■ Требования к производительности	■ Общий подход ■ Алгоритмы ■ Структуры данных ■ Архитектура ■ Оптимизация ■ Планируемая производительность	■ Платформы ■ Требования к оборудованию ■ Программные библиотеки ■ Стандарты интерфейса

Рис. 11.2. Разновидности требований

Постановка задачи может быть более или менее подробной. Требования к обычному продукту, например к программе для составления ведомостей или для рассылки счетов, могут содержать довольно много подробностей. Требования к исследовательскому проекту в новой области могут иметь меньшее количество деталей, однако исследование должно, по крайней мере, иметь некоторую цель, которая должна быть ясно выражена.

Чаще всего постановка задачи оказывается неоднозначной, неполной и даже внутренне противоречивой. Некоторые требования впоследствии оказываются просто неправильными. Другие требования могут быть ясно сформулированы, но окажут плохое влияние на поведение системы или приведут к неприемлемому повышению стоимости реализации. Некоторые требования сработают не так, как

рассчитывал инициатор создания системы. Постановка задачи — это отправная точка пути к ее пониманию, а не непреложная истина. Цель последующего анализа, описанного в главе 12, состоит в том, чтобы полностью понять задачу и ее подтекст. Нет никаких оснований предполагать, что постановка задачи, подготовленная без полного анализа, окажется правильной.

Пример с банкоматом. На рис. 11.3 показана постановка задачи для сети банкоматов.

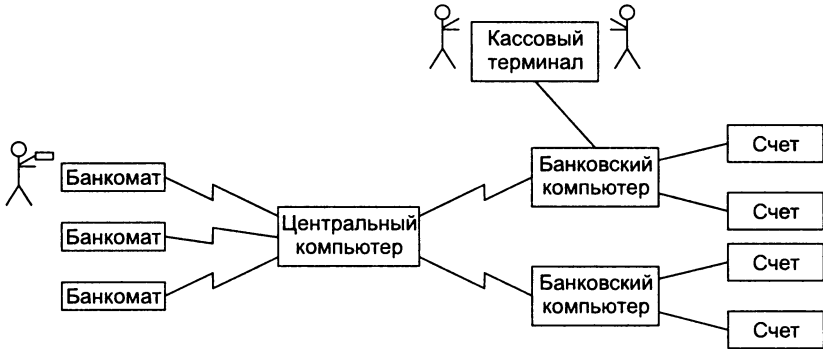


Рис. 11.3. Сеть банкоматов

Разработать программное обеспечение компьютеризированной банковской сети, включающей кассиров и банкоматы. Банкоматы могут совместно использоваться группой банков. Каждый банк предоставляет свой компьютер для учета своих счетов и обработки транзакций с ними. Кассиры также принадлежат отдельным банкам и взаимодействуют непосредственно с банковскими компьютерами. Кассиры вводят номера счетов и данные транзакций вручную.

Банкоматы взаимодействуют с центральным компьютером, который осуществляет транзакции с соответствующими банками. Банкомат принимает от клиента его карту, взаимодействует с клиентом, соединяется с центральной системой для выполнения транзакции, выдает наличные и печатает чеки. Система требует ведения записей и обеспечения безопасности. Система должна корректно обрабатывать одновременное обращение к одному и тому же счету.

Банки предоставляют собственное обеспечение для своих компьютеров. Вы должны разработать программное обеспечение для банкоматов и для компьютерной сети. Стоимость системы будет разделена между банками в соответствии с количеством клиентов-владельцев карт.

11.4. Резюме

Первый этап проекта — рождение идеи приложения. Можно придумать новую систему или усовершенствовать существующую. Перед тем как вкладывать средства и время в разработку, нужно оценить возможность создания системы, затраты и риски, связанные с ее разработкой, потребность в системе и отношение выигрыша

к затратам. Нужно учитывать точки зрения всех заинтересованных лиц и идти на компромиссы, необходимые для повышения вероятности успеха проекта, не только технического, но и коммерческого. При этом исходная идея обычно корректируется. Когда концептуализация системы заканчивается, можно приступать к постановке задачи, которая послужит отправной точкой для анализа. Постановка задачи не обязательно должна быть полной, она может измениться в процессе разработки. Сформулированная на бумаге задача помогает сосредоточить внимание на проекте.

Упражнения

- 11.1. (3) Рассмотрим новую противоблокировочную тормозную систему (ABS) для автомобиля. Придумайте развернутые ответы на перечисленные ниже вопросы.
 - Для кого предназначено приложение? Кто входит в число заинтересованных лиц? Оцените количество потенциальных покупателей в своей стране.
 - Укажите три свойства, которыми должно обладать приложение, и три других свойства, которыми оно обладать не должно.
 - Укажите три системы, с которыми должно работать приложение.
 - Укажите два наиболее серьезных риска.
- 11.2. (3) Повторите упражнение 11.1 для программного обеспечения книжного интернет-магазина.
- 11.3. (3) Повторите упражнение 11.1 для программного обеспечения, предназначенного для проведения модернизации кухни.
- 11.4. (3) Повторите упражнение 11.1 для системы проведения сетевого аукциона.
- 11.5. (4) Подготовьте постановку задачи (аналогично тому, как мы сделали это для банкомата) для каждой из перечисленных ниже систем. Вы можете ограничить область применения системы, однако будьте конкретными и не принимайте решений о реализации. Каждая спецификация должна состоять из 75–150 слов.
 - 1) Программа для игры в бридж.
 - 2) Разменный автомат.
 - 3) Система автоматического контроля скорости для автомобиля.
 - 4) Электронная пишущая машинка.
 - 5) Программа проверки орфографии.
 - 6) Автоответчик для телефона.
- 11.6. (3) Сделайте приведенные ниже требования более точными, изменив их формулировку. Выкиньте все решения, относящиеся к проектированию и реализации, выдвинутые в качестве требований.
 - 7) Система передачи данных с одного компьютера на другой по телефонной линии. Система должна обеспечивать надежную передачу данных по зашумленным линиям. Если принимающий компьютер не успевает обрабатывать данные или линия временно не позволяет выполнять

передачу, данные не должны быть утеряны. Данные следует передавать пакетами по протоколу с ведущим и ведомым, в котором получатель подтверждает (или отрицательно подтверждает, то есть сообщает об ошибке) все принятые пакеты.

- 8) Система автоматизации производства деталей со сложной механической обработкой. Детали проектируются в трехмерном редакторе, который входит в состав системы. Система должна давать на выходе ленту, которая может использоваться станками с ЧПУ для реального производства деталей.
- 9) Настольная издательская система, работающая по принципу WYSIWYG. Система должна уметь работать с текстом и с графикой. Графические элементы включают линии, квадраты, прямоугольники, многоугольники, окружности и эллипсы. Внутри системы окружность представляется как частный случай эллипса, а квадрат — как частный случай прямоугольника. Система должна поддерживать интерактивное редактирование документов в графическом режиме.
- 10) Система генерирования бреда. На вход поступает некоторый документ. На выходе должен появляться случайный текст, имитирующий входной документ благодаря воспроизведению частот появления комбинаций букв входного текста. Пользователь указывает порядок имитации и размер выходного текста. Имитация порядка N означает, что каждая последовательность из N выходных символов присутствует во входном тексте примерно с той же частотой. По мере повышения порядка стиль выходного текста становится все ближе к стилю входного текста.
Система должна генерировать выходной текст следующим образом. Сначала случайно выбирается некоторая позиция входного текста. Затем входной текст сканируется вперед до тех пор, пока не будет найдена последовательность символов, в точности соответствующая последним $N-1$ выходным символам. Если при этом достигается конец входного документа, сканирование необходимо продолжить с его начала. После обнаружения соответствия первая буква, следующая за найденной последовательностью, копируется в выходной документ. Процедура повторяется до тех пор, пока не будет сгенерирован необходимый объем бреда.
- 11) Система распространения электронной почты в компьютерной сети. Каждый пользователь системы должен иметь возможность отправлять почту с любого компьютера и получать почту по одной, принадлежащей ему, учетной записи. Система должна предоставлять средства для ответа на письма и для их пересылки, а также для сохранения сообщений в файл и для их распечатки. Кроме того, пользователи должны иметь возможность отправлять письма сразу нескольким другим пользователям при помощи списков рассылки. Каждый компьютер сети должен уметь хранить сообщения, предназначенные для временно отключенных от сети компьютеров.

Анализ предметной области

12

Целью анализа предметной области — следующего этапа разработки — является выработка точной, четкой, доступной для понимания и, наконец, корректной модели реального мира. Прежде чем строить что-то сложное, строитель должен понять требования, которые будут предъявлены к его продукту. Требования можно выражать словами, но слова часто оказываются неточными и двусмысленными. На этапе анализа разработчики занимаются строительством моделей и пытаются достичь глубокого понимания требований.

Чтобы построить модель предметной области, нужно провести интервью с экспертами в этой области, изучить составленные требования и тщательно исследовать связанные артефакты. Вам придется проанализировать скрытый смысл требований и переформулировать их в более строгой форме. Очень важно в первую очередь выделить наиболее значительные черты, а мелкие детали откладывать на потом. Успешная аналитическая модель показывает, что должно быть сделано, но не ограничивает возможности реализации.

В этой главе вы научитесь применять объектно-ориентированные концепции для конструирования модели предметной области. Эта модель служит несколькими целям: она уточняет требования, лежит в основе соглашения между заинтересованными лицами и разработчиками и становится отправной точкой для проектирования и реализации.

12.1. Обзор этапа анализа

Анализ начинается с постановки задачи, которая была выполнена на этапе концептуализации системы (рис. 12.1). Формулировка может быть неполной или неформальной. Анализ делает ее более точной и выявляет неоднозначности и несогласованности. Формулировку задачи не следует воспринимать как нечто неизменное. Она должна служить основой для определения реальных требований.

Вы должны достичь понимания реальной системы, которую описывает формулировка задачи, и представить ее важнейшие черты в виде модели. Утверждения, сделанные на естественном языке, часто бывают двусмысленными, неполными и несогласованными. Аналитическая модель — это точное, четкое представление задачи, позволяющее отвечать на вопросы и строить решение. На этапе проектирования вы будете ссылаться именно на аналитическую модель, а не на исходную туманную формулировку задачи.

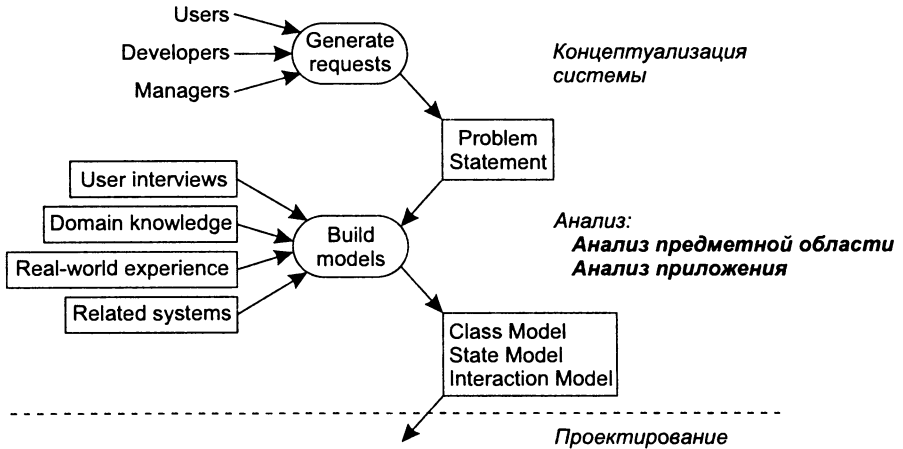


Рис. 12.1. Обзор этапа анализа

Еще важнее то, что процесс конструирования точной модели предметной области заставляет разработчика бороться с отсутствием взаимопонимания между ним и заинтересованными лицами на ранней стадии процесса разработки, когда последствия недопонимания легче устранить.

Аналитическая модель описывает три аспекта объектов: их статическую структуру (модель классов), взаимодействия между ними (модель взаимодействия) и жизненные циклы объектов (модель состояний). Важность каждого аспекта зависит от задачи. Модели классов, выведенные из сущностей реального мира, оказываются полезными почти для всех задач. В задачах, связанных с обработкой событий (например, пользовательские интерфейсы и управление процессами), будут важны аспекты состояний. В задачах, связанных со сложными вычислениями и различными видами пользователей, будут важны аспекты, отражаемые в модели взаимодействия.

Анализ — это не механический процесс, а, скорее, искусство. Выбор конкретного варианта представления зависит от суждений человека. Чаще всего в постановке задачи отсутствует важнейшая информация, которую приходится получать от заказчика или извлекать из знаний о предметной области, имеющихся у аналитика. Определенный произвол присутствует и при выборе уровня абстрагирования модели. Аналитику приходится взаимодействовать с заказчиком, устранять двусмысленности и неправильные представления. Аналитические модели делают возможным общение на языке однозначных символов.

Мы разделили этап анализа на две последовательные стадии. Первая стадия — анализ предметной области — описана в этой главе. Вторая стадия — анализ приложения — обсуждается в следующей главе. Модель приложения строится на основе модели предметной области путем добавления основных артефактов приложения, видимых пользователям. Пользователи должны одобрить решения, связанные с этими артефактами, поэтому они и включаются в модель приложения.

12.2. Модель классов предметной области

На первой стадии анализа требований выполняется конструирование модели предметной области. Модель предметной области отражает статическую структуру системы в реальном мире и делит ее на отдельные элементы, удобные для оперирования. Модель предметной области описывает реальные классы и их отношения друг с другом. Первой разрабатывается модель классов, потому что статическая структура обычно оказывается лучше определенной, меньше зависит от особенностей приложения и меньше изменяется в процессе эволюционирования решения задачи. Модель предметной области строится на основании информации, получаемой из постановки задачи, при изучении артефактов в аналогичных и связанных системах, из знаний экспертов в области приложения, а также из общих знаний о реальном мире. Вы должны учитывать всю доступную информацию и не полагаться на сведения, полученные из одного источника.

В первую очередь следует выделить классы и ассоциации, потому что они создают базовую структуру и подсказывают подход к решению задачи. Затем можно добавить атрибуты, с помощью которых описываются характеристики классов и ассоциаций. Затем классы следует объединить и организовать при помощи наследования. Попытки напрямую задать наследование без того, чтобы понять сначала структуру классов и их атрибутов, могут привести к необоснованному искажению структуры классов. Операции в модели предметной области обычно не отражаются. Основное ее назначение — это отражение информационного содержания предметной области.

Лучше всего записывать идеи на бумаге, даже если записи будут многословными и несогласованными. Главное — не пропустить важные детали. Начальный вариант аналитической модели почти наверняка будет содержать недочеты, которые нужно будет устранить на последующих итерациях. Модель не обязательно должна быть однородной по конструкции. Некоторые аспекты задачи придется анализировать на протяжении нескольких итераций, тогда как другие так и останутся набросками.

Конструирование модели классов предметной области выполняется в приведенной ниже последовательности.

1. Выделить классы (разделы 12.1.1—12.2.2).
2. Подготовить словарь данных (12.2.3).
3. Выделить ассоциации (разделы 12.2.4—12.2.5).
4. Выделить атрибуты объектов и связей (разделы 12.2.6—12.2.7).

5. Организовать и упростить классы при помощи наследования (раздел 12.2.8).
6. Проверить наличие маршрутов для наиболее вероятных запросов (раздел 12.2.9).
7. Перейти к следующей итерации и уточнить модель (раздел 12.2.10).
8. Пересмотреть уровень абстрагирования (раздел 12.2.11).
9. Сгруппировать классы в пакеты (раздел 12.2.12).

12.2.1. Выделение классов

Первый этап конструирования модели классов заключается в выделении классов, объединяющих объекты предметной области. К объектам относятся физические сущности, такие как дома, люди, машины, а также понятия, такие как траектории, расположение сидений и графики выплат. Все классы должны иметь смысл с точки зрения предметной области. Нужно избегать конструкций, относящихся к компьютерной реализации, таких как связные списки и подпрограммы. Не все классы явным образом присутствуют в описании задачи. Некоторые могут неявно определяться предметной областью или общими знаниями.

Начинать работу нужно с перечисления всех потенциальных классов из письменного описания задачи (рис. 12.2). Записывайте все названия, какие только придут вам в голову. Классы часто соответствуют существительным. Например, из предложения «система бронирования для продажи билетов на представления в различных театрах» можно выделить потенциальные классы *Бронирование*, *Система*, *Билет*, *Представление* и *Театр*. Однако не следует переписывать все подряд. Идея состоит в том, чтобы отразить в модели понятия. Не все существительные описывают понятия, и наоборот: понятия могут выражаться другими частями речи.

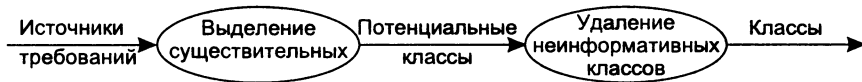


Рис. 12.2. Выделение классов

Не беспокойтесь о наследовании и классах высшего уровня. Постарайтесь выбрать конкретные классы таким образом, чтобы избежать подсознательного подавления деталей в попытке подогнать реальность под предполагаемую структуру. Например, если вы разрабатываете систему каталогизации и выдачи книг для библиотеки, выделите разные виды материалов: книги, журналы, газеты, записи, фильмы и т. д. Позднее вы сможете упорядочить их в более широкие категории с учетом сходств и отличий.

Пример с банкоматом. В результате выделения понятий из описания задачи о банкомате, которая приведена в главе 11, мы получили потенциальные классы, перечисленные на рис. 12.3. Дополнительные классы, отсутствующие в описании задачи, но выделенные из наших знаний о предметной области, приведены на рис. 12.4.



Рис. 12.3. Классы для модели банкомата, присутствующие в постановке задачи

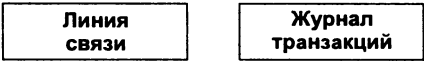


Рис. 12.4. Классы для модели банкомата, полученные из знаний о предметной области

12.2.2. Удаление лишних классов

Теперь нужно отбросить ненужные и некорректные классы, используя перечисленные ниже критерии. Рисунок 12.5 показывает, какие классы были удалены из модели банкомата.

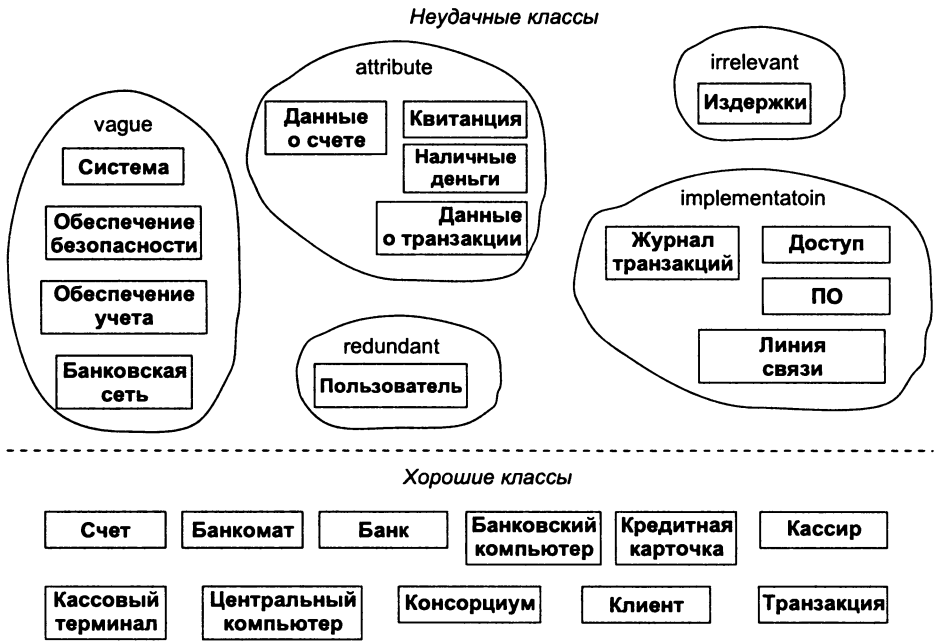


Рис. 12.5. Удаление лишних классов из модели банкомата

- **Избыточные классы.** Если два класса выражают одно и то же понятие, нужно оставить тот, название которого лучше всего описывает сущность понятия. Например, человека, покупающего билет на самолет, можно назвать

Клиент, однако более описательное название для класса будет *Пассажир*. С другой стороны, если задача имеет отношение к контрактам, заключаемым с charterными авиалиниями, класс *Клиент* также будет иметь смысл, поскольку контракт может быть заключен с несколькими пассажирами.

Пример с банкоматом. *Customer* (Клиент) и *User* (Пользователь) — избыточные классы; мы сохраняем класс *Customer*, как более соответствующий сути понятия.

Несущественные классы. Если класс имеет весьма слабое отношение к задаче, выбросьте его. В данном случае приходится принимать довольно субъективные решения, потому что важность класса зависит от контекста. Например, в системе бронирования театральных билетов профессии покупателей несущественны, но зато существенными могут быть профессии персонала театра.

Пример с банкоматом. *Cost* (Стоимость) лежит вне области классов, относящихся к программному обеспечению банкомата.

Нечеткие классы. Класс должен быть четко определен. Некоторые потенциальные классы могут иметь нечетко определенные границы или слишком широкую область охвата.

Пример с банкоматом. Класс *RecordkeepingProvision* (СредствоДляВеденияЗаписей) определен нечетко, а его задачи решаются классом *Transaction* (Транзакция). В других приложениях содержание этого класса может быть включено в другие классы, например *StockSales* (ПродажиАкций), *TelephoneCalls* (ТелефонныеЗвонки), *MachineFailures* (СбоиОборудования).

Атрибуты. Названия, характеризующие главным образом индивидуальные объекты, следует сделать атрибутами. Например, атрибутами обычно становятся такие потенциальные классы, как *name* (имя), *birthdate* (дата рождения) и *weight* (вес). Если некоторое свойство должно существовать независимо от носителя, его следует сделать классом, а не атрибутом. Например, номер офиса, в котором работает сотрудник, можно сделать классом в приложении, предназначенном для перераспределения офисов после реорганизации.

Пример с банкоматом. Элемент *AccountData* (ДанныеОСчете) определен нечетко, но в любом случае этот элемент, скорее всего, описывает характеристики счета, то есть является атрибутом. Банкомат выдает наличные и чеки, но, поскольку ничего другого он с ними не делает, эти сущности являются внешними по отношению к задаче, а потому правильнее будет сделать их атрибутами.

Операции. Если название описывает операцию, которая применяется к объектам, и она не рассматривается сама по себе, его следует исключить из списка классов. Например, телефонный звонок — это последовательность действий, в которых участвуют звонящий и телефонная сеть. Если мы занимаемся изготовлением телефонов, то звонок будет частью модели состояний, а не классом.

Операция, обладающая собственными чертами, должна быть представлена в модели классом. Например, в системе учета телефонных звонков *Call* (Звонок) будет важным классом с атрибутами *date* (дата), *time* (время), *origin* (исходный пункт), *destination* (пункт назначения).

- **Роли.** Название класса должно отражать его внутреннюю природу, а не роль, которую он играет в ассоциации. Например, *Owner* (Владелец) — плохое имя для класса из базы данных производителя автомобилей. Что если потом придется добавлять список водителей? Или людей, которые берут машины напрокат? Правильнее создать класс *Person* (Человек) или *Customer* (Клиент), который может играть разные роли, в том числе *owner* (владелец), *driver* (водитель), *lessee* (наниматель).

Иногда одна физическая сущность соответствует нескольким классам. Например, классы *Person* (Человек) и *Employee* (Сотрудник) в некоторых ситуациях могут быть отдельными классами, а в других — дублирующими друг друга. С точки зрения базы данных сотрудников предприятия эти два класса совпадают. С точки зрения федеральной налоговой базы один человек может быть занят на нескольких работах, поэтому класс *Person* необходимо отделить от класса *Employee*. Каждый экземпляр класса *Person* соответствует произвольному числу экземпляров класса *Employee*.

- **Конструкции, относящиеся к реализации.** Аналитическая модель не должна содержать конструкций, не принадлежащих к реальному миру. Они могут потребоваться на этапе проектирования, но не на этапе анализа. Такие сущности, как процессор, подпрограмма, процесс, алгоритм и прерывание, являются конструкциями, относящимися к реализации. Это верно для большинства приложений, однако для операционной системы перечисленные сущности могут быть и полноправными классами. Структуры данных, такие как связные списки, деревья, массивы и таблицы, почти всегда относятся к реализации, а не к реальному миру.

Пример с банкоматом. Некоторые потенциальные классы в действительности оказываются конструкциями, относящимися к реализации. *TransactionLog* (ЖурналТранзакций) — это просто множество транзакций. Его конкретное представление нужно выбирать на этапе проектирования. Коммуникационные связи можно представить в виде ассоциаций, а *CommunicationLine* (КоммуникационнаяЛиния) — просто физическая реализация такой связи.

- **Производные классы.** Как правило, классы, которые могут быть выведены из других классов, не следует включать в модель. Если производный класс особенно важен, его можно оставить, но поступать так можно только в исключительных случаях. Имена всех производных классов должны начинаться с символа косой черты (/).

12.2.3. Подготовка словаря данных

Сами по себе слова допускают слишком много интерпретаций, поэтому для всех элементов модели необходимо подготовить словарь данных. Для каждого класса

следует придумать описание размером в один абзац. Опишите область применения класса в рамках данной задачи, укажите все предположения и ограничения, касающиеся его использования. Словарь данных должен включать описание ассоциаций, атрибутов, операций и значений перечислимых типов. На рис. 12.6 показан пример словаря данных для классов нашей задачи с банкоматом.

12.2.4. Выделение ассоциаций

На следующем этапе вы должны выделить ассоциации между классами. Структурное отношение между двумя и более классами является ассоциацией. Если один класс ссылается на другой класс, это тоже ассоциация. Как мы уже писали в главе 3, для ссылок на классы не следует использовать атрибуты, их правильнее представлять в модели в виде ассоциаций. Например, в классе *Person* (Человек) не должно быть атрибута *employer* (работодатель). Правильно связать класс *Person* с классом *Company* (Компания) ассоциацией *WorksFor* (РаботаетНа). Ассоциации показывают отношения между классами на том же уровне абстракции, на котором находятся сами классы, тогда как атрибуты скрывают зависимости и их двустороннюю природу. Ассоциации могут быть реализованы множеством способов, выбор которых следует отложить до последующих этапов, чтобы сохранить относительную свободу во время проектирования.

Ассоциации часто соответствуют глаголам состояния или глагольным группам. К ним относятся характеристики физического размещения (*NextTo* — РядомС, *PartOf* — Часть, *ContainedIn* — СодержитсяВ), направленные действия (*Drives* — Управляет), передача информации (*TalksTo* — РазговариваетС), владение (*Has* — Имеет, *PartOf* — Часть) и выполнение некоторого условия (*WorksFor* — РаботаетНа, *MarriedTo* — ЖенатНа, *Manages* — Управляет). Сначала выделите все потенциальные ассоциации из описания задачи и запишите их на бумаге. Не пытайтесь слишком рано уточнять модель, но и не выписывайте все глаголы подряд. Суть в том, чтобы выделить все отношения, в какой бы форме они ни были выражены на естественном языке.

Пример с банкоматом. На рис. 12.7 показаны ассоциации. Большинство из них соответствуют глагольным формам из описания задачи. Некоторые ассоциации присутствовали в описании задачи неявным образом. Другие были введены в модель на основании знаний о реальном мире или предположений о сути задачи. Эти ассоциации нужно обсуждать с заказчиком, потому что в постановке задачи их не было.

12.2.5. Удаление лишних ассоциаций

Теперь нужно отбросить ненужные и некорректные ассоциации, руководствуясь перечисленными ниже критериями.

- **Ассоциации между классами, которые были удалены на предыдущих этапах.** Если один из классов, связываемых ассоциацией, был исключен, ассоциацию тоже нужно исключить или переформулировать в терминах других классов.

Счет — отдельный счет в банке, с которым производятся транзакции. Счета могут быть разных типов, например чеки и сбережения. Клиент может иметь несколько счетов.

Банкомат — терминал, позволяющий клиентам совершать транзакции, используя в качестве средства идентификации свои кредитные карты. Банкомат взаимодействует с клиентом, принимая от него данные, отправляя информацию о транзакции на центральный компьютер для ее проверки и обработки, а также выдавая клиенту наличные деньги. Предполагается, что банкомату не нужно работать вне сети.

Банк — финансовое учреждение, хранящее счета клиентов и выдающее кредитные карты для доступа к счетам с помощью банкоматов.

БанковскийКомпьютер — компьютер, принадлежащий банку и связанный в единую сеть с банкоматами и кассовыми терминалами банка. У банка может быть отдельный компьютер, работающий со счетами клиентов, но нас интересует только тот, который связан с сетью банкоматов.

КредитнаяКарта — карта, выданная клиенту банка и используемая для доступа к счету через банкомат. Каждая карта содержит код банка и имеет порядковый номер. Каждый банк имеет уникальный код внутри консорциума. Номер карты определяет счета, к которым открыт доступ. С помощью карты клиент может получать доступ не обязательно ко всем своим счетам. У карты может быть только один владелец, однако может существовать несколько копий карты, поэтому надо учитывать возможность одновременной работы с одной и той же картой с разных банкоматов.

Кассир — работник банка, уполномоченный вводить транзакции и принимать и выдавать наличные деньги и чеки клиентам. Все транзакции, наличные деньги и чеки, обрабатываемые кассирами, должны учитываться.

КассовыйТерминал — терминал, с помощью которого кассиры вводят транзакции. Кассиры выдают и принимают наличные деньги и чеки. Терминал печатает чеки. Кассовый терминал связывается с банковским компьютером для проверки и обработки транзакций.

ЦентральныйКомпьютер — компьютер, с которым работает консорциум для осуществления транзакций между банкоматами и банковскими компьютерами. Центральный компьютер сверяет коды банков, но не занимается напрямую обработкой транзакций.

Консорциум — сообщество банков, владеющих банкоматами и совершающих операции в сети банкоматов. Сеть банкоматов поддерживает транзакции только между банками, входящими в консорциум.

Клиент — владелец одного или нескольких счетов в банке. Клиента может представлять не только один человек, но и несколько человек или организация — для данной задачи это не имеет значения. Одного и того же человека, владеющего счетом в нескольких банках, следует рассматривать как нескольких клиентов.

Транзакция — единичный цельный запрос на операции со счетами одного клиента. Мы определили только, что банкоматы должны выдавать наличные деньги, но мы не должны запрещать возможность печати чеков и приема наличных денег или чеков. Можно также улучшить гибкость системы за счет обеспечения возможности обработки счетов разных клиентов, хотя изначально этого не было среди требований.

Пример с банкоматом. Мы можем исключить ассоциации *Banking network includes cashier stations and ATMs* (банковская сеть содержит кассиров и банкоматы), *ATM dispenses cash* (банкомат выдает наличные), *ATM prints receipts* (банкомат печатает чеки), *Banks provide software* (банки предоставляют программное обеспечение), *Cost apportioned to banks* (стоимость распределяется между банками), *System provides recordkeeping* (система обеспечивает ведение записей) и *System provides security* (система обеспечивает безопасность).

- **Несущественные, или относящиеся к реализации ассоциации.** Выбросьте все ассоциации, лежащие за пределами области задачи или описывающие конструкции, относящиеся к реализации.

Пример с банкоматом. Ассоциация *System handles concurrent access* (система обрабатывает параллельные обращения) описывает понятие, относящееся к реализации. Объекты реального мира изначально характеризуются параллелизмом. Требование, указанное в постановке задачи, относится именно к реализации алгоритма доступа.

Глагольные фразы

Банковская сеть включает в себя кассовые терминалы и банкоматы
 Консорциум совместно использует банкоматы
 Банк предоставляет банковский компьютер
 Банковский компьютер ведет учет счетов
 Банковский компьютер обрабатывает транзакции, совершаемые со счетами
 Банк владеет кассовым терминалом
 Кассовый терминал связан с банковским компьютером
 Кассир вводит транзакцию для данного счета
 Банкоматы связываются с центральным компьютером для проведения транзакции
 Центральный компьютер проверяет транзакцию, связываясь с банком
 Банкомат принимает кредитную карту
 Банкомат взаимодействует с пользователем
 Банкомат выдает наличные деньги
 Банкомат печатает квитанции
 Система поддерживает параллельный доступ
 Банки предоставляют программное обеспечение
 Издержки берут на себя банки

Имплицитные глагольные фразы

Консорциум состоит из банков
 Банк имеет свой счет
 Консорциум владеет центральным компьютером
 Система обеспечивает учет
 Система обеспечивает безопасность
 Клиенты имеют кредитные карты

Знания о предметной области

С помощью кредитной карты можно получить доступ к счетам
 В банке работают кассиры

Рис. 12.7. Ассоциации, выделенные из описания задачи о банкомате

Действия. Ассоциация должна описывать структурное свойство области приложения, а не кратковременное событие. В некоторых случаях требование выражается в виде действия, однако подразумевает некоторое структурное отношение. Такое утверждение нужно переформулировать.

Пример с банкоматом. Ассоциация *ATM accepts cash card* (банкомат принимает банковскую карту) описывает часть цикла взаимодействия банкомата с клиентом, а не постоянное отношение банкомата и карты. По той же причине можно исключить ассоциацию *ATM interacts with user* (банкомат взаимодействует с пользователем). *Central computer clears transaction with bank* (Центральный компьютер подтверждает транзакцию в банке) описывает действие, подразумевающее структурное отношение *Central computer communicates with bank* (Центральный компьютер взаимодействует с банком).

Тернарные ассоциации. Большинство *n*-арных ассоциаций можно выразить через бинарные, добавив соответствующие квалификаторы. Если термин в тернарной ассоциации является описательным и не имеет собственной индивидуальности, он является атрибутом бинарной ассоциации. Например, ассоциацию *company pays salary to person* (Компания платит сотруднику зарплату) можно переформулировать в виде бинарной ассоциации *company employs person* (Компания нанимает сотрудника), причем каждая связь *Company-Person* будет характеризоваться своим значением *salary* (зарплата).

Иногда в приложении действительно требуется тернарная ассоциация. Например, структуру *Professor teaches course in room* (Профессор читает курс лекций в аудитории) нельзя разбить на бинарные ассоциации без утраты информации. Однако учтите, что нам ни разу не попадались ассоциации с четырьмя и более классами за все время нашей работы.

Пример с банкоматом. Ассоциацию *Bank computer processes transaction against account* (Банковский компьютер обрабатывает транзакцию по счету) можно разбить на *Bank computer processes transaction* (Банковский компьютер обрабатывает транзакцию) и *Transaction concerns account* (Транзакция связана со счетом). Аналогичным образом нужно поступить с транзакцией *Cashier enters transaction for account* (Кассир вводит транзакцию по счету). Ассоциация *ATMs communicate with central computer about transaction* (Банкоматы взаимодействуют с центральным компьютером касательно транзакции) на самом деле представляет собой две бинарные ассоциации *ATMs communicate with central computer* (Банкоматы взаимодействуют с центральным компьютером) и *Transaction entered on ATM* (Транзакция начинается банкоматом).

Производные ассоциации. Отбросьте ассоциации, которые могут быть выражены через другие ассоциации. Например, *GrandparentOf* (Дедушка) можно выразить через пару ассоциаций *ParentOf* (Родитель). Выбрасывайте и ассоциации, выражаемые как ограничения на атрибуты. Например, *youngerThan* (моложе) выражает условие, касающееся дат рождения двоих человек, а не какую-то дополнительную информацию.

Классы, атрибуты и ассоциации модели классов должны отражать максимально независимую информацию. Наличие множества маршрутов между классами часто указывает на производные ассоциации, которые могут быть выражены через некоторые примитивные ассоциации. *Consortium shares ATMs* (Консорциум совместно владеет банкоматами) — это композиция ассоциаций *Consortium owns central computer* (Консорциум владеет центральным компьютером) и *Central computer communicates with ATMs* (Центральный компьютер взаимодействует с банкоматами).

Не все ассоциации, образующие множественные маршруты между классами, являются избыточными. Иногда существование ассоциации можно вывести из нескольких примитивных ассоциаций, а вот ее кратность таким путем определить нельзя. В подобной ситуации ассоциацию следует сохранить, при условии, что ограничение на кратность важно для модели. Например, на рис. 12.8 компания нанимает множество сотрудников и владеет множеством компьютеров. Каждому сотруднику предоставляется нуль и более компьютеров для персонального использования. Некоторые компьютеры предназначены для общего пользования и не приписаны ни к какому сотруднику. Кратность ассоциации *AssignedTo* (Приписан) не может быть выведена из кратности ассоциаций *Employs* (Нанимает) и *Owens* (Владеет).

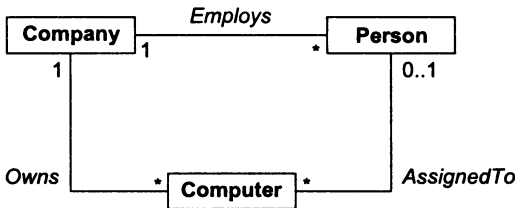


Рис. 12.8. Кажущаяся избыточность ассоциации

Хотя производные ассоциации не вносят дополнительной информации в модель, они могут быть полезны в реальном мире и при проектировании. Например, отношения родства типа «дядя», «теща» и «кузен» имеют собственные названия, потому что они описывают типичные отношения, которые считаются достаточно важными в нашем обществе. Если отношения такого рода особенно важны для модели, их можно оставить на диаграмме классов, указав перед именем символ косой черты (/), означающий зависимый статус и позволяющий отличать их от фундаментальных ассоциаций.

Дальше следует проанализировать семантику ассоциаций.

- **Неправильно названные ассоциации.** Название должно говорить не о том, как или почему возникла некоторая ситуация, а о том, в чем эта ситуация состоит. Названия важны для понимания модели в целом, а потому выбирать их следует очень осторожно.

Пример с банкоматом. *Bank computer maintains accounts* (Банковский компьютер обслуживает счета) — это утверждение, описывающее действие.

Название ассоциации правильное будет переформулировать как *Bank holds account* (Банк управляет счетом).

- **Названия полюсов ассоциаций.** Названия полюсов нужно указывать везде, где они имеют смысл. Например, в ассоциации *WorksFor* (РаботаетНа) класс *Company* (Компания) по отношению к классу *Person* (Человек) является работодателем (employer), а *Person* по отношению к *Company* представляет собой сотрудника (employee). Если пару классов связывает только одна ассоциация и ее смысл очевиден, названия полюсов можно не указывать. Например, смысл ассоциации *ATMs communicate with central computer* (Банкоматы взаимодействуют с центральным компьютером) ясен из названий классов. Ассоциация между двумя экземплярами одного и того же класса требует наличия имен полюсов, с помощью которых можно было бы отличать эти экземпляры. Например, в ассоциации *Person manages person* (Человек руководит человеком) полюса будут называться *boss* (начальник) и *worker* (подчиненный).
- **Квалифицированные ассоциации.** Обычно название идентифицирует объект в рамках некоторого контекста. Большинство названий не являются уникальными в масштабах всей системы (глобально). Контекст вместе с именем позволяют уникально идентифицировать объект. Например, название компании должно быть уникально в том штате, где она расположена, но может повторяться в других штатах (был момент, когда в каждом из штатов Огайо, Индиана, Калифорния и Нью-Джерси работала своя компания «Стандард Ойл»). Название компании квалифицирует ассоциацию *State charters company* (Штат разрешает создание компании); *State* (Штат) и название компании уникально идентифицируют *Company* (Компанию). Квалификатор позволяет отличать друг от друга объекты, находящиеся у полюса ассоциации с кратностью «много».

Пример с банкоматом. Квалификатор *bankCode* (кодБанка) позволяет отличать друг от друга банки, входящие в состав консорциума. Каждая карта должна иметь банковский код, чтобы транзакции можно было направлять в выдавший ее банк.

- **Кратность.** Этот параметр ассоциаций указывать нужно, но не старайтесь определить его точно на первом этапе моделирования. Кратность часто изменяется в процессе анализа. Проверяйте ассоциации с кратностью «один». Например, ассоциация *one Manager manages many employees* (один Менеджер руководит множеством сотрудников) не позволит создать матричную систему управления или описать сотрудника, отвечающего перед несколькими начальниками. Подумайте о необходимости введения квалификаторов для ассоциаций с кратностью «много», а также об упорядоченности объектов.
- **Недостающие ассоциации.** Добавьте все пропущенные ассоциации, которые вам удастся обнаружить.

Пример с банкоматом. Мы пропустили *Transaction entered on cashier station* (Транзакция вводится на кассе), *Customers have accounts* (Клиенты имеют счета) и *Transaction authorized by cash card* (Транзакция авторизуется картой).

- **Агрегация.** Агрегация важна для некоторых видов приложений, в частности для описания деталей механизмов и спецификаций материалов. Для других приложений важность агрегации не столь велика, и не всегда бывает понятно, следует ли ее использовать вместо обычной ассоциации. Не тратьте слишком много времени на определение типа ассоциации. Выберите то, что сразу покажется вам более правильным, и двигайтесь дальше.

Пример с банкоматом. Мы считаем *Bank* (Банк) частью *Consortium* (Консорциума) и обозначаем отношение между ними как агрегацию.

Пример с банкоматом. На рис. 12.9 приведена диаграмма классов с нанесенными на нее ассоциациями. Мы указали имена только для наиболее важных ассоциаций. Обратите внимание, что мы разделили *Transaction* на *RemoteTransaction* и *CashierTransaction*, чтобы создать разные ассоциации с их участием. На диаграмме указаны значения кратности. Некоторые решения были произвольными. Не стоит беспокоиться об этом: существует множество корректных моделей задачи. Мы разбираем процесс анализа с точностью до мелочей. Когда вы приобретете достаточный опыт, вы будете выполнять сразу несколько этапов в уме.

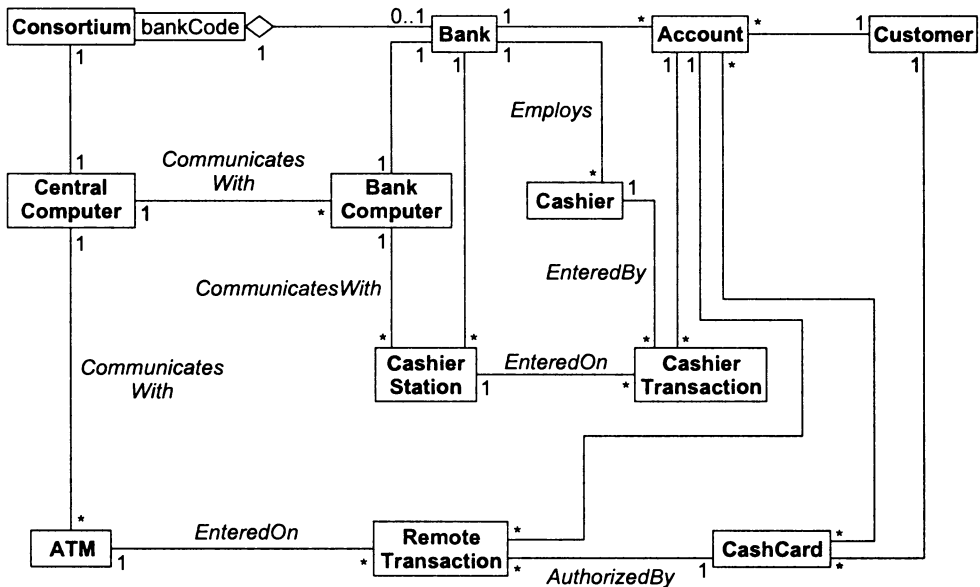


Рис. 12.9. Исходная диаграмма классов для банкомата

12.2.6. Выделение атрибутов

Теперь пришло время заняться выделением атрибутов. Атрибуты — это свойства объектов, такие как вес, скорость или цвет. Значения атрибутов не должны быть объектами; чтобы показать отношение между объектами, следует использовать ассоциации.

Атрибуты обычно присутствуют в описании задачи в виде существительных, участвующих в притяжательных оборотах, наподобие «цвет машины» или

«положение курсора». Прилагательные часто соответствуют конкретным значениям атрибутов-перечислений (например, «красный», «включен», «устарел»). В отличие от классов и ассоциаций атрибуты вряд ли будут полностью перечислены в описании задачи. Вам придется опираться на свое знание области задачи и реального мира, чтобы выделить их все. Атрибуты также присутствуют в артефактах родственных систем. К счастью, атрибуты редко влияют на базовую структуру задачи.

Не доводите процесс выделения атрибутов до абсурда. Рассматривайте только те из них, которые имеют непосредственное отношение к приложению. Сначала займитесь наиболее важными атрибутами, мелкие детали можно будет добавить в модель позже. В процессе анализа не тратьте время на детали реализации. Обязательно давайте всем атрибутам значащие имена.

Обычно производные атрибуты включать в модель на этом этапе не следует. Например, возраст можно определить по дате рождения и текущей дате (последняя является свойством окружения). Не выражайте производные атрибуты через операции (например, вычислить `Возраст`), пусть даже реализация их этим способом вполне допустима.

Ищите атрибуты и для ассоциаций. Эти атрибуты являются свойствами связей между объектами, а не свойствами индивидуальных объектов. Например, ассоциация «многие-ко-многим» между *Stockholder* (ДержательАкций) и *Company* (Компания) имеет атрибут *numberOfShares* (количествоАкций).

12.2.7. Удаление лишних атрибутов

Исключите ненужные и некорректные атрибуты, руководствуясь следующими критериями.

- **Объекты.** Если важной чертой элемента является независимое существование, то этот элемент — объект (а не атрибут). Например, босс — это класс, а зарплата — атрибут. Разница часто зависит от приложения. Например, в списке рассылки *город* может быть атрибутом, тогда как в Переписи населения *Город* будет классом с множеством атрибутов и отношений. Элемент, обладающий собственными чертами в рамках данного приложения, должен моделироваться как класс.
- **Квалификаторы.** Если значение атрибута зависит от конкретного контекста, его можно попробовать переформулировать в виде квалификатора. Например, *номерСотрудника* — это не уникальная характеристика человека, занятого на двух работах, это, скорее, квалификатор ассоциации «Компания нанимает сотрудника».
- **Имена.** Имена лучше моделировать как квалификаторы, а не как атрибуты. Ответьте на следующие вопросы: служит ли имя для выбора уникального объекта из множества? Может ли объект, принадлежащий множеству, иметь более одного имени? Если ответ положительный, имя служит квалификатором ассоциации. Если имя кажется уникальным, возможно, вы забыли добавить в модель класс, для которого оно служит квалификатором.

Например, название Отдела может быть уникально в рамках компании, но когда-нибудь вам придется работать с несколькими компаниями. Лучше сразу использовать соответствующую квалифицированную ассоциацию.

Имя является атрибутом в том случае, если его использование не зависит от контекста, в особенности если оно не уникально в некотором множестве. Имена людей, в отличие от названий компаний, могут повторяться, а потому являются атрибутами.

- **Идентификаторы.** Объектно-ориентированные языки используют понятие идентификатора объекта для обозначения однозначной ссылки на объект. Не следует включать в модель атрибут, единственным назначением которого является идентификация, потому что идентификаторы присутствуют в моделях классов неявным образом. Перечисляйте только те атрибуты, которые присутствуют в области приложения. Например, кодСчета — это настоящий атрибут, потому что банк назначает каждому счету свой код, а клиент видит этот код. Напротив, внутренний идентификатор Транзакции не является атрибутом, хотя на этапе реализации его использование может быть выгодно.
- **Атрибуты ассоциаций.** Если существование значения требует существования связи, соответствующее свойство является атрибутом ассоциации, а не одного из связанных ею классов. Атрибуты обычно достаточно ясно выделяются из ассоциаций «многие-ко-многим»: их нельзя прикрепить ни к одному из классов из-за их кратности. Например, атрибут датаВступления принадлежит ассоциации между *Человеком* и *Клубом*, потому что человек может принадлежать к нескольким клубам, а клуб может иметь несколько членов. Ассоциации «один-ко-многим» детализировать сложнее, потому что тут можно прикрепить атрибут к одному из классов без потери информации. Сопровождайтесь этому желанию, потому что если кратность изменится, модель станет некорректной. Те же проблемы возникают и с ассоциациями типа «один-к-одному».
- **Внутренние значения.** Если атрибут описывает внутреннее состояние объекта, невидимое снаружи, его следует исключить из аналитической модели.
- **Лишние детали.** Исключите незначительные атрибуты, не влияющие на большинство операций.
- **Нетипичные атрибуты.** Атрибут, полностью отличающийся от всех остальных и не связанный с ними, может указывать на то, что класс, к которому он относится, следует разделить на два разных класса. Класс должен быть простым и цельным. Смешивание разных классов — одна из основных причин появления ненадежных моделей. Нечетко определенные классы часто бывают результатом слишком раннего принятия решений, касающихся реализации.
- **Логические атрибуты.** Рассмотрите все логические атрибуты еще раз. Часто логический атрибут можно расширить и переформулировать в виде перечисления [Coad-95].

Пример с банкоматом. Применим эти критерии к нашему примеру, чтобы получить список атрибутов для каждого класса (рис. 12.10). Некоторые потенциальные атрибуты стали квалификаторами ассоциаций. Мы рассматриваем несколько аспектов модели.

- *bankCode* (кодБанка) и *cardCode* (кодКарты) хранятся на карте. Их формат зависит от реализации, но мы обязаны, по крайней мере, создать ассоциацию *Bank issues CashCard* (Банк выдает КредитнуюКарту). Код карты становится квалификатором этой ассоциации, а код банка — квалификатором банка по отношению к консорциуму.
- Состояние компьютеров не имеет отношения к нашей задаче. Включенное или выключенное состояние компьютера — это атрибут, являющийся частью реализации.
- Нам пришлось справиться с искушением исключить из модели *Consortium* (Консорциум), несмотря на то, что он присутствует в ней в единственном числе. Этот класс обеспечивает контекст для квалификатора *bankCode* (кодБанка) и может быть полезен для дальнейшего расширения.

Не забывайте о том, что наш пример с банкоматом — это всего лишь пример. В реальных приложениях после завершения построения модели количество атрибутов у каждого класса значительно больше, чем в нашем случае (см. рис. 12.10).

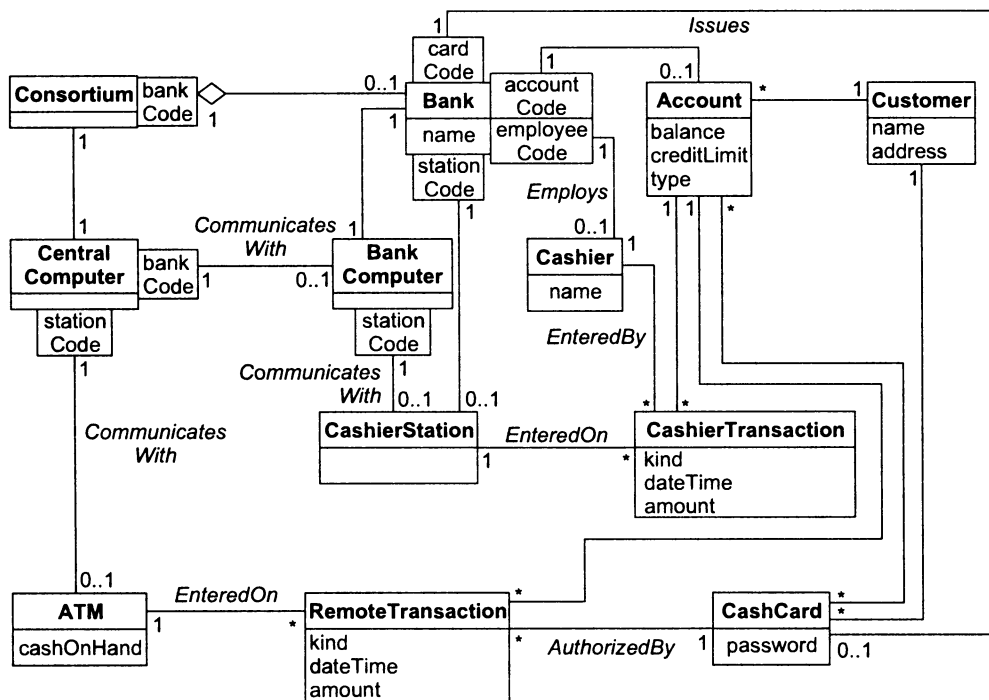


Рис. 12.10. Модель классов банкомата с атрибутами

12.2.8. Реструктурирование при помощи наследования

Следующий шаг: организация классов при помощи наследования путем выявления их общей структуры. Наследование может быть использовано двумя способами: как обобщение одинаковых аспектов существующих классов в супер-классах (снизу вверх) и как конкретизация существующих классов множеством подклассов (сверху вниз).

- **Обобщение снизу вверх.** Наследование можно проследить снизу вверх путем поиска классов с одинаковыми атрибутами, ассоциациями и операциями. Для каждого обобщения следует определить суперкласс, в котором будут содержаться общие черты. Для этого вам, возможно, придется несколько переопределить некоторые атрибуты или классы. Это вполне допустимо, однако не следует слишком упорствовать: возможно, вы неправильно выявили обобщение. Некоторые обобщения копируются из реального мира. Везде, где это возможно, следует использовать существующие понятия. Недостающие классы можно выписывать из соображений симметрии.

Пример с банкоматом. Классы *RemoteTransaction* (УдаленнаяТранзакция) и *CashierTransaction* (КассоваяТранзакция) подобны друг другу (за исключением инициализации) и могут быть обобщены классом *Transaction* (Транзакция). С другой стороны, с точки зрения нашего примера классы *CentralComputer* (ЦентральныйКомпьютер) и *BankComputer* (БанковскийКомпьютер) имеют довольно мало общего, поэтому обобщать их не следует.

- **Конкретизация сверху вниз.** Конкретизация обычно следует из описания области приложения. Ищите именные группы, состоящие из различных прилагательных с именем класса: *флуоресцентная* лампа, *лампа накаливания*; *фиксированное* меню, *раскрывающееся* меню, *выдвигающееся* меню. Избегайте чрезмерного уточнения. Если предлагаемая конкретизация несовместима с существующим классом, возможно, этот класс просто неправильно сформулирован.
- **Обобщение и перечисление.** Перечислимые подклассы области приложения чаще всего попадают под определение конкретизации. Обычно бывает достаточно отметить существование множества перечислимых частных случаев без явного их указания. Например, счет (в примере с банкоматом) может быть счетом до востребования или сберегательным счетом. В некоторых банковских приложениях такое деление может быть очень важным, однако на поведение банкомата оно не влияет, поэтому *type* (тип счета) можно сделать просто атрибутом класса *Account* (Счет).
- **Множественное наследование.** Вы можете использовать множественное наследование, но только тогда, когда это действительно необходимо, так как оно повышает концептуальную и техническую сложность модели.
- **Подобные ассоциации.** Если название ассоциации появляется в модели несколько раз, причем несет оно при этом одинаковый смысл, попробуйте обобщить ассоциируемые классы. Иногда у таких классов может не быть

ничего общего, за исключением ассоциации, но достаточно часто вы будете обнаруживать общие черты, пропущенные на предыдущих этапах.

Пример с банкоматом. *Transaction* (Транзакция) *вводится* как посредством *CashierStation* (Касса), так и на *ATM* (Банкомат). Класс *EntryStation* (УстройствоВвода) обобщает классы *CashierStation* и *ATM*.

- **Корректирование уровня наследования.** Атрибуты и ассоциации должны быть присвоены конкретным классам из иерархии. Каждый из них должен быть присвоен наиболее общему классу, к которому он применим. Для этого вам могут потребоваться некоторые корректировки. Из симметрии может следовать наличие дополнительных атрибутов, которые позволят более четко отличать подклассы друг от друга.

На рис. 12.11 показана модель классов банкомата после добавления наследования.

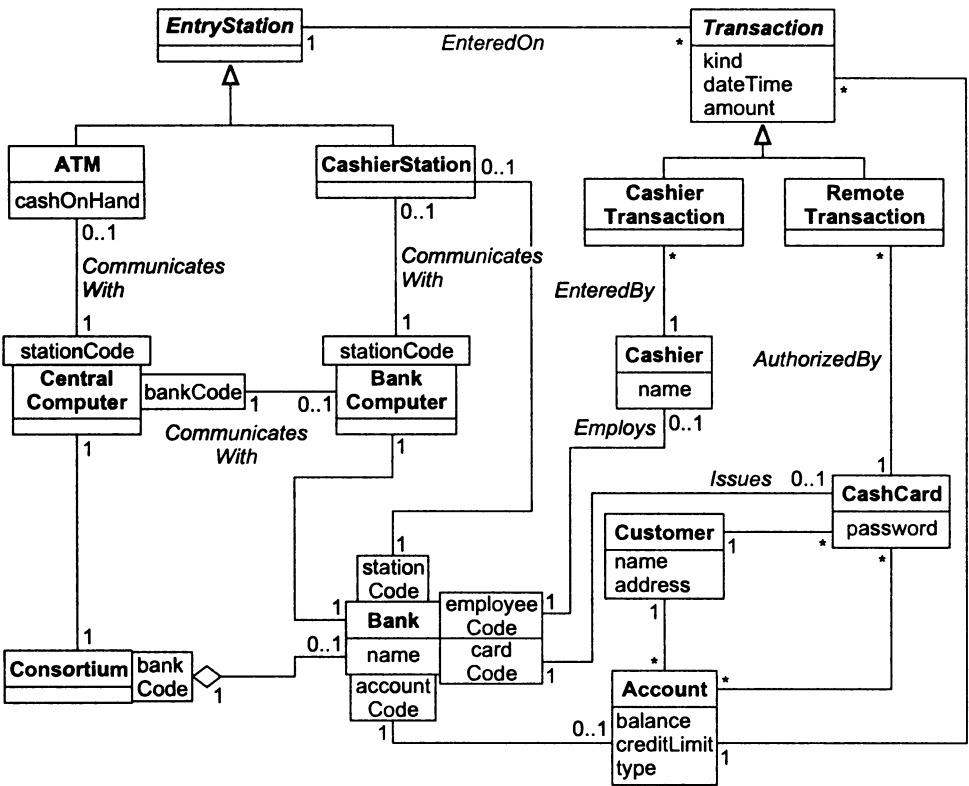


Рис. 12.11. Преобразованная модель классов банкомата

12.2.9. Проверка маршрутов

Проследите маршруты в модели классов и проверьте, что все они дают разумные результаты. Если в некотором месте предполагается уникальное значение, существует ли маршрут, дающий уникальный результат? Предусмотрен ли метод

выбора уникальных значений объектов, охарактеризованных кратностью «много»? Подумайте, какие вопросы вы считаете важными. Есть ли важные вопросы, на которые вы не можете ответить? Такие вопросы указывают на недостающую информацию. Если что-то простое из реального мира в модели оказалось сложным, вы могли что-то пропустить (однако обязательно проверьте, не свойственна ли та же сложность реальному миру).

В модели могут присутствовать классы, не связанные ни с какими другими. Обычно это происходит тогда, когда отношение между такими классами и остальной моделью носит распределенный (размытый) характер. Однако всегда проверяйте такие классы, потому что вы могли просто пропустить существующую ассоциацию.

Пример с банкоматом. Кредитная карта сама по себе не является уникальным идентификатором счета, поэтому пользователь должен иметь возможность каким-то образом выбрать свой счет. Если пользователь укажет тип счета (до востребования или накопительный), по каждой карте можно будет обращаться только к одному счету каждого типа. Это кажется разумным, и многие карты действительно так и работают, но такое решение ограничивает систему. Альтернатива — потребовать, чтобы клиенты помнили номера счетов. Если кредитная карта работает с одним счетом, переводы денег между счетами невозможны.

Мы предположили, что сеть банкоматов обслуживает один консорциум банков. Реальные банкоматы часто обслуживают перекрывающиеся сети банков и принимают не только банковские, но и кредитные карты. Нашу модель пришлось бы расширить, чтобы описать такую ситуацию. Мы будем предполагать, что клиент не возражает против определенных ограничений нашей системы.

12.2.10. Итерационная разработка модели классов

Модель классов редко оказывается правильной после первого же прохода. Весь процесс разработки программного обеспечения строится на итерациях. Разные части модели обычно находятся на разных стадиях разработки. Если вы нашли недостаток, вернитесь на предыдущую стадию (при необходимости) и устраните его. Некоторые уточнения могут быть сделаны только после разработки моделей состояний и взаимодействия.

Недостающие классы можно выявить по следующим признакам.

- **Асимметрия обобщений и ассоциаций.** Добавляйте новые классы по аналогии с имеющимися.
- **Разные атрибуты и операции в одном классе.** Разбейте класс таким образом, чтобы каждый из новых классов получился цельным.
- **Сложно провести ясное обобщение.** Один класс может играть две роли. Разделите его, и тогда каждый из новых классов сразу же займет свое место в иерархии.
- **Дублирование ассоциаций с одинаковым названием и смыслом.** Обобщите классы, выделив объединяющий их суперкласс.

- **Роль формирует семантику класса.** Возможно, ее следует выделить в собственный класс. Часто это означает преобразование ассоциации в класс. Например, человек может работать в нескольких компаниях на разных условиях. Тогда *Employee* — отдельный класс, обозначающий человека, работающего на конкретную компанию (в дополнение к классам *Person* и *Company*).

Ищите также недостающие ассоциации.

- **Отсутствуют маршруты для обращения к операциям.** Добавьте ассоциации, которые позволят вам отвечать на запросы.

Нужно позаботиться и об избыточных элементах модели.

- **У класса нет атрибутов, операций и ассоциаций.** Зачем он тогда нужен? Не создавайте подклассы просто для отражения перечисления. Если потенциальные подклассы во всем похожи друг на друга, используйте атрибуты.
- **Избыточная информация.** Удаляйте ассоциации, не несущие новых сведений, или помечайте их как производные.

Наконец, нужно скорректировать размещение атрибутов и ассоциаций.

- **Имена полюсов слишком широкие или слишком узкие по сравнению с классами, находящимися около этих полюсов.** Переместите ассоциацию вверх или вниз по иерархии классов.
- **Обращение к объекту осуществляется по значению одного из его атрибутов.** Рассмотрите возможность создания квалифицированной ассоциации.

На практике схема построения модели выглядит не такой жесткой, как та, что была описана нами. После приобретения необходимого опыта вы сможете объединять некоторые этапы в один. Например, вы можете выделить потенциальные классы, отбросить некорректные, даже не выписывая их, после чего добавить классы на диаграмму вместе с ассоциациями. Некоторые части модели можно прорабатывать более тщательно, чем другие. Порядок этапов проектирования при необходимости можно изменять. Однако в процессе изучения моделирования классов мы рекомендуем вам первые несколько раз в точности следовать описанной схеме.

Пример с банкоматом. Класс *CashCard* (БанковскаяКарта) на самом деле обладает двойственной сущностью: это одновременно модуль, при помощи которого осуществляется авторизация клиента для доступа к его счетам, и кусочек пластика, с которого банкомат считывает закодированные идентификаторы. В данном случае коды являются частью реального мира, а не просто компьютерными артефактами. Именно коды, а не сама кредитная карта передаются в центральный компьютер. Нам следует разделить банковскую карту на два класса: *CardAuthorization* (КартаАвторизации) — носитель права на доступ к счетам клиента, и *CashCard* (БанковскаяКарта) — кусочек пластика, на котором записан код банка и номер банковской карты, имеющий смысл для этого банка. Одной карте авторизации может соответствовать несколько банковских карт, каждая из которых должна по соображениям безопасности обладать своим последовательным номером. Код карты, написанный на ней самой, идентифицирует карту авторизации в данном

банке. Каждая карта авторизации идентифицирует один или несколько счетов, например один счет до востребования и один накопительный.

Класс транзакций определен не достаточно универсально, чтобы можно было выполнять трансфер денег между счетами, потому что он связан только с одним счетом. Вообще говоря, класс *Transaction* (Транзакция) должен состоять из одного или нескольких обновлений отдельных счетов. Обновление — это одно действие (снятие наличных, размещение наличных, запрос баланса), выполняемое с одним счетом. Все обновления, относящиеся к одной транзакции, должны быть выполнены атомарно, как одна операция. Если хотя бы одно обновление не будет выполнено успешно, вся транзакция отменяется.

Различие между классами *Bank* (Банк) и *BankComputer* (БанковскийКомпьютер), а также между *Consortium* (Консорциум) и *CentralComputer* (ЦентральныйКомпьютер) не влияет на аналитическую модель. Тот факт, что взаимодействие осуществляется компьютерами, на самом деле является артефактом реализации. Мы превращаем *BankComputer* в *Bank*, а *CentralComputer* в *Consortium*.

Класс *Customer* (Клиент) пока не появлялся в аналитической модели. Однако, если учесть возможные операции по открытию новых счетов, клиент может стать важным, поэтому пока мы оставим его в покое.

На рис. 12.12 показана исправленная диаграмма классов, которая стала проще и яснее.

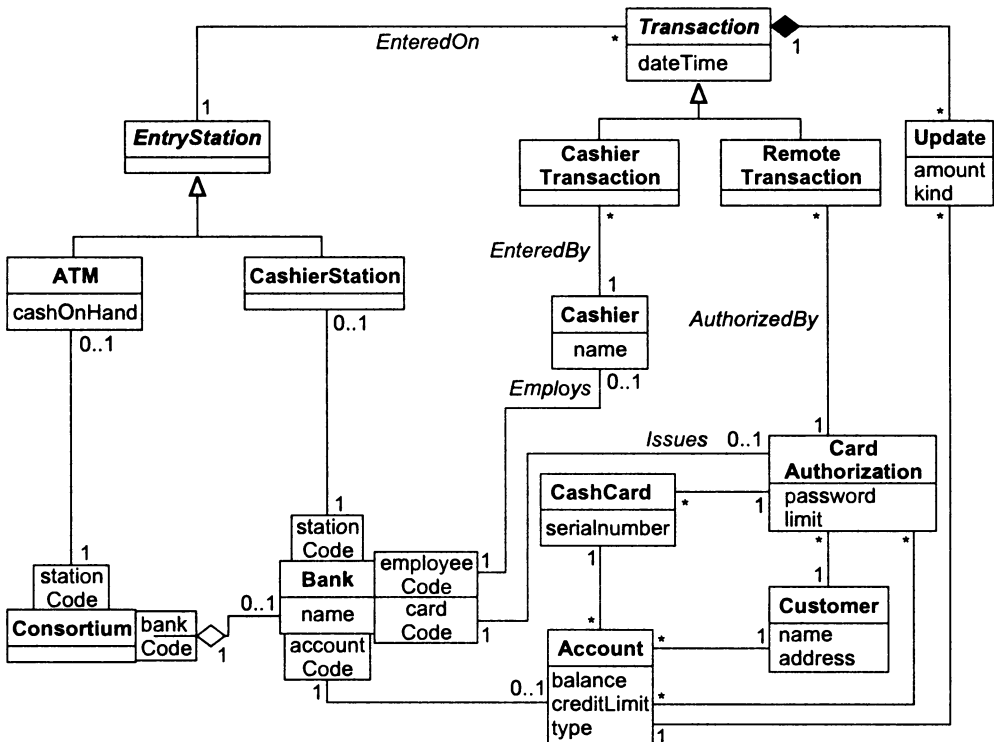


Рис. 12.12. Модель классов банкомата после очередных исправлений

12.2.11. Смещение уровня абстрагирования

Пока что мы воспринимали описание задачи буквально. Мы считали существительные и глаголы в описании задачи прямыми аналогами классов и ассоциаций. Такой подход удобен на начальном этапе анализа, но его не всегда оказывается достаточно. Иногда для решения задачи необходимо перейти на более высокий уровень абстрагирования. Этим следует заниматься в процессе построения модели, но мы добавили в описываемую последовательность моделирования отдельный этап, чтобы вы не пропустили процедуру абстрагирования.

Например, однажды мы столкнулись с приложением, разработчики которого создали отдельные классы *IndividualContributor* (Сотрудник), *Supervisor* (Контролер) и *Manager* (Управляющий). Сотрудники *отчитываются* перед контролерами, а те — перед управляющими. Эта модель корректна, но она обладает некоторыми недостатками. Три класса обладают слишком большим количеством общих черт; разница лишь в том, кто перед кем отчитывается. Например, они все характеризуются телефонными номерами и адресами. Общие черты можно было бы вынести в суперкласс, но от этого модель бы стала только сложнее. Еще одна проблема возникла, когда мы поговорили с разработчиками, и они сказали, что хотят добавить еще один класс, перед которым будут отчитываться управляющие.

На рис. 12.13 показаны исходная и улучшенная модели. Итоговая модель стала более абстрактной. Вместо жесткого кодирования иерархии управления в модели, мы можем ввести его более гибко, при помощи ассоциации между руководителем и подчиненным. *Человек*, атрибут *employeeType* которого имеет значение *individualContributor*, является подчиненным и отчитывается перед другим *человеком*, у которого *employeeType* = *supervisor*. Аналогичным образом, люди-контролеры отчитываются перед людьми-управляющими. В усовершенствованной модели подчиненный не обязательно имеет руководителя, потому что иерархия подчинения должна быть конечной. Добавление уровня руководства не изменяет структуру модели; изменяются только конкретные данные.

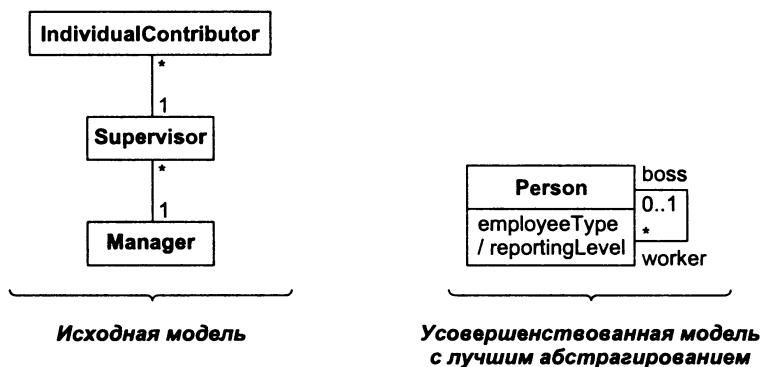


Рис. 12.13. Смещение уровня абстрагирования

Чтобы воспользоваться преимуществами абстрагирования, необходимо мыслить в категориях образцов. Разные образцы применяются на разных стадиях

разработки. Здесь нас интересуют аналитические образцы. Образец (pattern) является квинтэссенцией знаний экспертов и дает проверенное решение общей задачи. Например, правая часть рис. 12.13 представляет собой образец моделирования иерархии управления. Везде, где мы сталкиваемся с потребностью описания иерархии управления, мы вспоминаем об этом образце и добавляем его в нашу модель. Использование проверенных образцов дает уверенность в правильности подхода и значительно повышает продуктивность.

Пример с банкоматом. Некоторые абстракции мы уже включили в модель банкомата на предыдущих этапах. Мы провели различие между классами *CashCard* (БанковскаяКарта) и *CardAuthorization* (КартаАвторизации). Более того, мы включили концепцию транзакций вместо того, чтобы перечислять все возможные виды взаимодействий.

12.2.12. Группировка классов в пакеты

Последний этап моделирования — группировка классов в пакеты. *Пакет* (package) — это группа элементов (классов, ассоциаций, обобщений и вложенных пакетов), характеризующихся общей темой. Пакеты делают модель более удобной с точки зрения построения, печати и просмотра. Более того, помещая определенные классы и ассоциации в пакет, вы делаете семантическое утверждение. Классы одного пакета связаны друг с другом сильнее, чем классы в разных пакетах.

Нужно стараться ограничивать ассоциации рамками одного пакета. Некоторые классы можно повторять в разных пакетах. Чтобы распределить классы по пакетам, ищите точки сочленения (cut points) — классы, являющиеся единственным соединением двух частей модели, не имеющих между собой других связей. Такой класс образует мост между двумя пакетами. Например, в системе управления файлами класс *File* (Файл) является точкой сочленения между структурой каталогов и содержимым файла. Старайтесь выбирать пакеты так, чтобы сокращать количество пересечений на диаграммах классов. Приложив некоторые усилия, можно добиться того, что большинство диаграмм классов станут плоскими графами без пересекающихся линий.

По возможности следует использовать пакеты из предыдущих проектов, однако не стоит «втискивать» их в модель. Повторное использование осуществляется проще всего, если часть предметной области совпадает с предыдущей задачей. Если новая задача похожа на предыдущую, но отличается от нее, вы можете расширить исходную модель, чтобы описать обе задачи. Судите сами, что лучше в каждом конкретном случае: строить модель заново или использовать предыдущую.

Пример с банкоматом. Наша модель невелика и не требует деления на пакеты, но она может стать ядром для более детальной модели. Потенциальные пакеты могут быть такими:

- кассиры: кассир, устройство ввода, касса, банкомат;
- счета: счет, банковская карта, карта авторизации, клиент, транзакция, обновление, кассовая транзакция, удаленная транзакция;
- банки: консорциум, банк.

В каждом пакете детализация может осуществляться независимо. Пакет «счета» может включать разнообразные транзакции, сведения о клиентах, выплаты процентов, штрафы. Пакет «банки» может содержать сведения о филиалах, адреса банков и распределение затрат.

12.3. Модель состояний предметной области

Некоторые объекты предметной области за время своей жизни сменяют несколько качественно различных состояний. В этих состояниях они могут иметь разные ограничения на значения атрибутов, разные ассоциации или кратности, выполнять разные операции или иметь разное поведение и т. д. Часто бывает полезно построить диаграмму состояний для таких классов. Диаграмма состояний описывает различные состояния, в которых может находиться объект, свойства объекта и действующие на них ограничения, а также события, вызывающие переход объекта из одного состояния в другое.

Большинство классов предметной области не требуют использования диаграмм состояний. Для их описания достаточно списка операций. Модель состояний может помочь в понимании поведения тех классов, которые могут находиться в существенно разных состояниях.

Сначала нужно выявить классы, которые могут находиться в разных состояниях, и записать состояния для каждого класса. Затем необходимо определить события, вызывающие переход каждого объекта из одного состояния в другое. Зная состояния и события, вы можете построить диаграмму состояний для каждого из объектов. Наконец, вы должны проверить полученные диаграммы на полноту и корректность.

Модель состояний предметной области конструируется в несколько этапов:

- выявление классов, обладающих разными состояниями (раздел 12.3.1);
- выделение состояний (раздел 12.3.2);
- выделение событий (раздел 12.3.3);
- построение диаграмм состояний (раздел 12.3.4);
- проверка диаграмм состояний (раздел 12.3.5).

12.3.1. Выявление классов с разными состояниями

Изучите список классов предметной области, характеризующихся четко определенным жизненным циклом. Ищите классы, которые развиваются или имеют циклическое поведение. Идентифицируйте значимые состояния в жизненном цикле каждого из объектов. Например, научная статья для некоторого журнала последовательно переходит из состояния *Написание* в состояние *Рецензирование*, а затем либо в состояние *Принята* или *Отвергнута*. Состояния статьи могут сменяться циклично, например если рецензенты потребуют внесения изменений или дополнений, но основная последовательность состояний может быть охарактеризована как развитие. В качестве примера циклического поведения можно привести самолет, который обычно проходит состояния *Обслуживание*, *Загрузка*, *Полет*

и *Разгрузка*. Некоторые состояния могут присутствовать не в каждом цикле; могут существовать состояния и вне цикла, но основная последовательность в данном случае явно имеет циклический характер. Бывают классы и с хаотичным жизненным циклом, но большинство классов могут быть отнесены либо к развивающимся, либо к циклическим.

Пример с банкоматом. *Account* (Счет) — это важное понятие из области бизнеса. Поведение банкомата зависит от состояния счета. Жизненный цикл счета — смесь последовательного и циклического поведения. Другие классы предметной области банкомата не имеют значимых моделей состояния предметной области.

12.3.2. Выделение состояний

Перечислите состояния для каждого из классов. Охарактеризуйте объекты каждого класса: укажите значения атрибутов, которые может иметь объект, ассоциации, в которых он может участвовать, значения кратности узлов этих ассоциаций, определите атрибуты и ассоциации, которые имеют смысл только в определенных состояниях. Дайте каждому состоянию осмысленное название. Название не должно описывать, каким образом состояние было получено, оно должно обозначать само состояние.

Не тратьте время на мелкие различия между состояниями, особенно количественные (такие как «маленький», «средний», «большой»). Состояния должны выделяться на основании качественных отличий поведения, атрибутов или ассоциаций.

Не обязательно определять все состояния до перехода к выделению событий. Рассмотрев события и переходы между состояниями, вы сможете найти недостающие состояния.

Пример с банкоматом. Класс *Account* (Счет) может находиться в состояниях *Normal* (нормальное — обычный режим доступа), *Closed* (закрытый — клиент закрыл свой счет, но он еще не был исключен из записей банка), *Overdrawn* (с превышенным кредитным лимитом — клиент превысил кредитный лимит по своему счету) и *Suspended* (приостановлен — доступ к счету был заблокирован по каким-либо причинам).

12.3.3. Выделение событий

Получив предварительный список состояний, займитесь поиском событий, которые вызывают переходы между этими состояниями. Подумайте о внешних воздействиях, которые вызывают изменения состояний. Во многих ситуациях событие можно рассматривать как завершение текущей деятельности. Например, если статья находится в состоянии *Рецензирование*, переход из этого состояния осуществляется по завершении деятельности рецензента. Решение может быть положительным (переход в состояние *Принята*) или отрицательным (переход в состояние *Отвергнута*). Завершение деятельности может вызывать альтернативные переходы, которые могут добавляться в процессе совершенствования модели. Например, статья может перейти в состояние *Принята с обязательными изменениями*.

Другие события можно обнаружить, поразмыслив о том, каким образом объект может попасть в определенное состояние. Например, если вы снимаете трубку телефона, он переходит в состояние *Набор номера*. На многих аппаратах имеются специальные кнопки, нажатие которых вызывает специальные функции. Если вы нажмете кнопку *повтор*, телефон наберет последний номер и перейдет в состояние *Вызов*. Нажатие кнопки *программирование* приведет к переходу в состояние *Программирование*.

Внутри состояния могут происходить и события, не вызывающие переходов. Для модели состояний предметной области важны только те события, которые вызывают переходы. Информацию, содержащуюся в событии, следует представлять в форме списка его параметров.

Пример с банкоматом. Перечислим важнейшие события в нашей модели: *close account* (заккрытие счета), *withdraw excess funds* (превышение кредитного лимита), *repeated incorrect PIN* (повторный неправильный ввод PIN-кода), *suspected fraud* (возможная подделка) и *administrative action* (административные действия).

12.3.4. Построение диаграмм состояний

Распределите события по состояниям, к которым они относятся. Добавьте переходы, показывающие изменения состояний, вызванные осуществлением события в то время, когда объект находится в определенном состоянии. Если событие завершает состояние, из этого состояния обычно бывает только один переход в другое состояние. Если событие инициирует целевое состояние, вы должны рассмотреть, в каких состояниях это событие может происходить, и добавить на диаграмму переходы из этих состояний в целевое состояние. Если событие имеет разное действие на разные состояния, добавьте переходы для каждого из этих состояний.

После указания переходов проанализируйте смысл событий, не вызывающих переходы. Игнорируется ли событие? Тогда все в порядке. Не обозначает ли оно ошибку? Тогда добавьте переход в состояние обработки ошибки. Может быть, вы просто забыли учесть влияние этого состояния? Добавьте на диаграмму новый переход. Иногда вы будете даже обнаруживать новые события.

Обычно деятельность не слишком важна для построения диаграммы состояний классов предметной области. Если же объекты классов выполняют деятельность при переходах, добавьте эту деятельность на диаграмму.

Пример с банкоматом. На рис. 12.14 показана модель состояний предметной области для класса *Account* (Счет).

12.3.5. Проверка диаграмм состояния

Проверьте каждую модель состояний. Все ли состояния достижимы? Особенное внимание нужно уделить маршрутам по диаграмме. Если она описывает класс с развивающимся поведением, есть ли на диаграмме маршрут, соединяющий начальное состояние с конечным? Присутствуют ли на ней ожидаемые отклонения от основной последовательности? Если диаграмма описывает класс с циклическим поведением, есть ли на ней основная петля цикла? Есть ли тупиковые состояния, завершающие цикл?

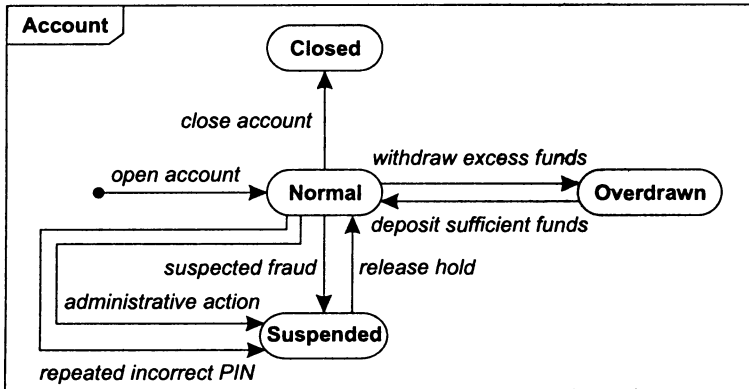


Рис. 12.14. Модель состояний предметной области

Ищите недостающие маршруты на основании своих знаний о предметной области. Иногда отсутствие маршрутов указывает на отсутствие нужных состояний. Готовая модель состояний должна точно описывать жизненный цикл класса.

Пример с банкоматом. Наша модель состояний для класса *Account* (Счет) выглядит чрезмерно упрощенной, но нас она вполне удовлетворяет. Для построения более совершенной модели потребовались бы значительные познания в банковском деле.

12.4. Модель взаимодействия предметной области

Модель взаимодействия редко бывает актуальна при анализе предметной области. При анализе требуется сосредоточиться на ключевых понятиях и глубоких структурных взаимоотношениях, а не на том, какими они кажутся пользователю. Однако модель взаимодействия является важным аспектом модели приложения, поэтому в следующей главе мы обязательно расскажем о ней.

12.5. Итерационный анализ

Большинство аналитических моделей не могут быть разработаны за один проход. Описание задачи часто содержит внутренние циклы, да и вообще большинство приложений не приемлет линейного подхода, потому что отдельные части задачи взаимодействуют между собой. Чтобы понять задачу целиком, вам придется проводить анализ методом итераций: сначала подготовить первое приближение к модели, а затем осуществлять итерации с углублением понимания. Между анализом и проектированием нет четкой границы, поэтому не следует увлекаться излишним уточнением аналитической модели. Проверьте конечную аналитическую модель вместе с заказчиком и экспертами в предметной области.

12.5.1. Уточнение аналитической модели

Аналитическая модель в целом может быть несогласованной. Отдельные ее составляющие могут быть плохо сбалансированы. Отредактируйте эти составляющие таким образом, чтобы модель стала более ясной и цельной. Постарайтесь уточнить формулировку классов, чтобы увеличить объем совместного использования и улучшить структуру модели. Добавьте детали, которые были пропущены при первом проходе.

Некоторые конструкции покажутся вам неуклюжими и неуместными. Внимательно изучите их; возможно, вы ошиблись в понимании понятий. Иногда по мере углубления понимания возникает необходимость значительного реструктурирования модели. Это проще выполнить на этапе анализа, чем на более поздних этапах, поэтому не избегайте внесения изменений только потому, что у вас уже есть готовая модель. Если множество конструкций кажутся похожими, но не вполне подходят друг к другу, это может означать, что вы пропустили более общее понятие. Ищите обобщения, основанные на неправильно выбранных аспектах (и исправляйте их).

Достаточно часто затруднения вызывают физические объекты, обладающие двумя логически различными значениями. Каждый аспект физического объекта должен быть представлен в модели отдельным объектом. Указанием на подобную проблему является то, что класс плохо встраивается в иерархическую структуру и имеет несколько наборов несвязанных атрибутов, ассоциаций и операций.

Ищите исключительные ситуации, частные случаи и отсутствие ожидаемой симметрии. Все это указывает на возможные проблемы. Подумайте о реструктурировании модели с целью лучшего описания ограничений структурой модели.

Не увлекайтесь кодированием произвольных бизнес-методик в своей модели. Программы должны помогать бизнесу, но не тормозить возможные изменения. Достаточно часто можно подобрать абстракции, которые будут повышать гибкость бизнеса без существенного усложнения модели.

Исключите классы и ассоциации, которые сначала казались полезными, но на данном этапе стали избыточными. Часто оказывается, что можно объединить пару классов аналитической модели, потому что различия между ними не влияют на остальную модель. Модели обычно растут по мере проведения анализа. Это важно, потому что объем разработки возрастает с увеличением модели. Ищите незначительные элементы, которые можно выбросить из модели, и абстракции, которые помогут ее упростить.

Хорошая модель кажется правильной и не имеет лишних деталей. Не беспокойтесь, если она не покажется вам идеальной. Даже хорошая модель обычно имеет несколько частей, структура которых может быть адекватной, но никогда не покажется абсолютно правильной.

12.5.2. Корректировка требований

После завершения анализа модель служит в качестве основы для формулирования требований и определяет область дальнейшей деятельности. Реальные требования, по большей части, войдут в состав модели. Кроме них могут быть выдвинуты также ограничения на производительность. Они должны быть установлены ясно

и четко, вместе с критериями оптимизации. Прочие решения обычно указывают метод решения, а потому должны быть отделены и поставлены под сомнение.

Окончательный вариант модели необходимо проверить вместе с заказчиком. В процессе анализа некоторые требования могли оказаться некорректными или непрактичными. Заказчик должен подтвердить корректировку этих требований. Аналитическую модель должны проверить и бизнес-эксперты, которые определяют ее соответствие реальному миру. Мы заметили, что аналитические модели — это эффективное средство взаимодействия с бизнес-экспертами, не являющимися экспертами в компьютерных областях.

Окончательно проверенная аналитическая модель служит основой для создания системной архитектуры, проектирования и реализации. Вы должны исправить исходную формулировку задачи в соответствии с теми сведениями, которые были получены на этапе анализа.

12.5.3. Анализ и проектирование

Цель анализа состоит в том, чтобы полностью определить проблему, не давая преимуществ какому-либо конкретному варианту реализации. На практике оказывается невозможно полностью устранить связь с реализацией. Этапы проектирования не имеют четких границ. Не существует и идеальной аналитической модели. Не стоит слепо руководствоваться приведенными выше правилами. Назначение правил — сохранить гибкость и сделать возможными последующие изменения, но вы должны помнить и о том, что цель моделирования заключается в выполнении задачи, а гибкость — лишь одно из средств.

Пример с банкоматом. На этом этапе мы не вносим больше никаких изменений в модель банкомата. В реальном приложении изменений наверняка будет больше, чем в примере из учебника, потому что у вас будут рецензенты, для которых это приложение будет жизненно важно.

12.6. Резюме

Модель предметной области содержит общие сведения о приложении: понятия и отношения, известные экспертам предметной области. Эта модель обычно включает в себя модели классов и состояний. Реже в ее состав входит модель взаимодействия. Назначение этапа анализа состоит в том, чтобы понять задачу для последующей разработки корректного проекта. Хорошая аналитическая модель описывает важнейшие черты задачи, но не вносит артефактов реализации, раньше времени ограничивающих проектные решения.

Модель классов предметной области отражает статическую структуру реального мира. Сначала нужно выделить классы, затем определить ассоциации между ними, отметить атрибуты (за исключением незначительных). Упорядочение и упрощение структуры классов может быть выполнено при помощи обобщения. Сгруппируйте родственные классы и ассоциации в пакеты. Дополните модель классов словарем данных — кратким текстовым описанием назначения и области применения каждого элемента.

Если класс предметной области может находиться в существенно различных состояниях, нарисуйте его диаграмму состояний. Для большинства классов этого не потребуется.

Методология никогда не бывает такой же линейной, как ее описание в учебнике. Любая сложная аналитическая модель строится итерациями на нескольких уровнях. Вы не должны разрабатывать все части модели в одинаковом темпе. Результат анализа заменяет исходную формулировку задачи и служит основанием для проектирования.

Библиографические замечания

Эбботт рассказывает о том, как использовать существительные и глаголы в описании задачи для ее решения [Abbott-83]. Еще одна хорошая книга [Coad-95] содержит примеры аналитических шаблонов.

Ссылки

[Abbott-83] Russell J. Abbott. Program Design by Informal English Descriptions. Communications of the ACM 26, 11 (November 1983), 882–894.

[Coad-95] Peter Coad, David North, and Mark Mayfield. Object Models: Strategies, Patterns, and Applications. Upper Saddle River, NJ: Yourdon Press, 1995.

Упражнения

- 12.1. (3) Для каждой из перечисленных ниже систем укажите относительную важность трех аспектов моделирования: 1) моделирование классов, 2) моделирование состояний, 3) моделирование взаимодействия. Поясните свои ответы. Например, для компилятора ответ может выглядеть как «3, 1, 2». Для компилятора наиболее важно моделирование взаимодействия, потому что это приложение занимается главным образом преобразованием данных.
- 1) Игрок в бридж.
 - 2) Разменный автомат.
 - 3) Система автоматического регулирования скорости автомобиля.
 - 4) Электронная пишущая машинка.
 - 5) Программа проверки орфографии.
 - 6) Телефонный автоответчик.
- 12.2. (7) Создайте диаграмму классов для каждой системы из упражнения 11.6. Заметьте, что требования сформулированы не полностью, а потому модель классов тоже будет неполной.

Упражнения 12.3–12.8 связаны между собой. Выполняйте их последовательно. Ниже мы приводим возможную спецификацию простого редактора диаграмм, который может использоваться в качестве ядра для множества различных приложений.

Редактор будет использоваться в интерактивном режиме для создания и редактирования рисунков. Рисунок состоит из нескольких страниц. Рисунки сохраняются в виде именованных ASCII-файлов и могут считываться из них. На страницах изображаются прямоугольники и связи между ними. Каждый прямоугольник может содержать необязательную строку текста. Текст может располагаться только внутри прямоугольников. Редактор должен автоматически изменять размер прямоугольника в соответствии с размером текста внутри него. Размер шрифта изменять нельзя. Любую пару прямоугольников на одной и той же странице можно соединить между собой ломаной, состоящей из вертикальных и горизонтальных отрезков. На рис. У12.1 показан пример простого рисунка, занимающего одну страницу.

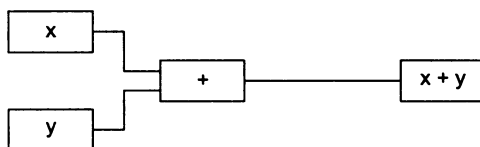


Рис. У12.1. Пример рисунка

Редактор будет управляться через систему меню (включая контекстные меню). Для выбора пунктов меню, объектов и связей будет использоваться трехкнопочная мышь. Ниже перечислены операции, которые должен поддерживать редактор: создание страницы, удаление страницы, переход к следующей странице, переход к предыдущей странице, создание прямоугольника, создание связей между прямоугольниками, ввод текста, группировка выделенных объектов, вырезание выделенных объектов в буфер, перемещение выделенных объектов, копирование выделенных объектов, вставка содержимого буфера, редактирование текста, сохранение рисунка, загрузка рисунка. Операции копирования, вырезания и вставки работают с буфером обмена. Копирование создает копию выделенных объектов в буфере. Вырезание перемещает выделенные объекты в буфер. Вставка копирует содержимое буфера на страницу. Каждая операция копирования и вставки затирает текущее содержимое буфера. Перемещение области просмотра и изменение масштаба не поддерживаются; страницы имеют фиксированный размер. При перемещении прямоугольников вместе с ними должен двигаться и содержащийся в них текст, а связи между прямоугольниками должны растягиваться и сокращаться.

12.3. (3) Ниже приведен список потенциальных классов. Подготовьте список классов, которые должны быть исключены из модели по причинам, указанным в этой главе. Объясните свой выбор. Если причин несколько, приводите основную. Потенциальные классы: символ, линия, координата x , координата y , связь, положение, длина, ширина, совокупность, выделение, меню, мышь, кнопка, компьютер, рисунок, файл рисунка, страница, всплывающее меню, точка, пункт меню, выделенный объект, выделенная линия, выделенный прямоугольник, выделенный текст, имя файла, прямоугольник, буфер, координата сегмента линии, соединение, текст, название, начало координат, масштабный множитель, угловая точка, конечная точка, графический объект.

12.4. (3) Подготовьте словарь данных для объектов, оставшихся от предыдущего списка.

- 12.5. (3) Ниже приведен список потенциальных ассоциаций и обобщений для редактора диаграмм. Подготовьте список ассоциаций и обобщений, которые нужно исключить или переименовать по любой из причин, указанных в этой главе. Объясните свой выбор. Если причин несколько, приводите основную. Потенциальные ассоциации и обобщения: прямоугольник содержит текст, прямоугольник обладает положением, связь логически соединяет два прямоугольника, прямоугольник перемещается, связь содержит точки, связь определяется последовательностью точек, выделение, или буфер, или страница представляют собой совокупности, строка символов характеризуется положением, прямоугольник содержит строку символов, строка символов содержит символы, линия обладает длиной, совокупность состоит из связей и прямоугольников, связь удаляется, линия перемещается, линия представляет собой графический объект, точка представляет собой графический объект.
- 12.6. На рис. У12.2 приведена неполная диаграмма классов редактора диаграмм. Покажите, каким образом ее можно использовать для каждого из следующих запросов. Для выражения своих запросов используйте комбинацию OCL (см. главу 3) и псевдокода.
- 1) (2) Найти все выделенные прямоугольники и связи.
 - 2) (4) По одному прямоугольнику найти все непосредственно с ним связанные.
 - 3) (8) По прямоугольнику найти все связанные с ним непосредственно или косвенно.
 - 4) (2) Имея прямоугольник и связь, определить, участвует ли прямоугольник в этой связи.
 - 5) (3) Имея прямоугольник и связь, найти другой прямоугольник, связанный с данным другим концом этой связи.
 - 6) (4) Найти все связи между двумя данными прямоугольниками.
 - 7) (6) Имея группу выделенных объектов, найти связи, являющиеся пограничными. Если не все прямоугольники входят в состав выделения, пограничные связи будут соединять выделенные прямоугольники с невыделенными. Такие связи требуют особой обработки при операциях вырезания или перемещения.
- 12.7. (6) На рис. У12.3 показана новая версия диаграммы классов, где класс *Connection* (Соединение) явным образом описывает соединение связи и прямоугольника. Перепишите запросы из предыдущего упражнения по этой модели.

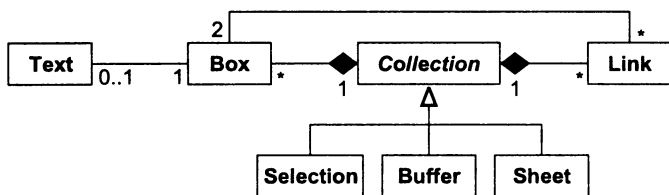


Рис. У12.2. Неполная диаграмма классов редактора диаграмм

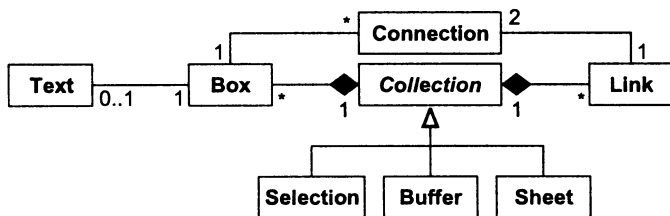


Рис. У12.3. Альтернативная диаграмма классов редактора диаграмм

12.8. (5) Для каких классов нужно строить диаграммы состояний? Опишите некоторые важные состояния и события.

Упражнения 12.9—12.13 связаны между собой. Выполняйте их последовательно. В этих упражнениях рассматривается компьютеризованная система учета очков, которую вы вызвались разработать для местной школы синхронного плавания. Команды собираются для участия в соревнованиях, на которых дети выступают в двух видах состязаний: соло и групповых. Соло исполняется индивидуально и заключается в выполнении элементов синхронного плавания, таких как плавание на спине с вертикально поднятой ногой. В групповом выступлении участвует вся команда. Очки начисляются как за индивидуальные выступления, так и за групповые, но ваша система должна работать только с индивидуальными.

На регистрации перед началом соревнований каждый участник должен указать свое имя, возраст, адрес и название команды, к которой он принадлежит. Для упрощения учета каждому участнику присваивается личный порядковый номер.

В процессе соревнований несколько человек выступают одновременно в разных секторах (углах бассейна). Их выступление оценивают судьи и секретари. Секретари быстро устают, поэтому они часто сменяются. Несколько судей и секретарей приписываются к каждому сектору. В течение сезона каждый судья и секретарь может обслуживать несколько секторов. Для обеспечения согласованности результатов каждое сольное выступление проводится в одном секторе с неизменным судейством. В процессе соревнований в одном секторе может быть проведено несколько сольных выступлений (упражнений).

Соревнующиеся спортсмены делятся на группы. Каждая группа начинает выступление в своем секторе. Когда спортсмен заканчивает выступление в одном секторе, он переходит в следующий сектор для следующего выступления. Когда все участники выполняют одно упражнение, сектор переключается на следующее предписанное ему упражнение.

Каждый соревнующийся получает одну попытку в одном выступлении. Перед выступлением оглашается номер участника. Иногда порядок нарушается или секретари путаются в участниках, и выступления в секторе останавливаются до тех пор, пока проблема не устраняется. Каждый судья дает свою оценку выступлению, поднимая карточку с числом. Секретарь записывает показанные числа и суммирует их, отбрасывая минимальное и максимальное значения. Средняя оценка умножается на коэффициент сложности данного упражнения.

Индивидуальные и командные призы вручаются во время закрытия соревнований на основании набранных очков. Участники делятся на несколько возрастных

категорий, в каждой категории вручается отдельный набор призов. Индивидуальные призы вручаются на основании сольных выступлений, а командные — по сумме результатов в сольных и командных выступлениях.

Система, которую вы должны разработать, будет использоваться для хранения всей информации, требуемой для планирования, регистрации и подсчета очков. В начале сезона всех участников внесут в систему, после чего будет подготовлен календарь соревнований. В частности, различные виды сольных выступлений будут распределены по соревнованиям. Перед соревнованиями система будет использоваться для регистрации участников. Во время соревнований в нее будут заноситься очки и с ее помощью будут определяться победители.

12.9. (3) Ниже приведен список потенциальных классов для системы начисления очков. Напишите, какие из них следует исключить по каким-либо причинам, упомянутым ранее в этой главе. Обоснуйте свое мнение. Если причин для удаления несколько, укажите наиболее важную из них. Список классов: адрес, возраст, возрастная категория, среднее количество очков, спина, карта, спортсмен, имя спортсмена, участник соревнований, вычисление среднего, заключение, соревнующийся, угол (бассейна), дата, коэффициент сложности, соревнование, сольное выступление, файл данных о членах команды, группа, индивидуум, индивидуальный приз, судья, лига, этап (эстафеты), список запланированных соревнований, соревнование, суммарный счет, номер, человек, бассейн, приз, регистрация, регистратор, судейские очки, групповые выступления, счет, секретарь, сезон, сектор, команда, командный приз, название команды, попытка, пытаться, синхронное плавание.

12.10. (3) Подготовьте словарь данных для классов, оставшихся в модели после выполнения предыдущего упражнения.

12.11. (4) Ниже приведен список потенциальных ассоциаций и обобщений для системы подсчета очков. Выберите ассоциации и обобщения, которые следует исключить или переименовать по причинам, упомянутым в настоящей главе. Объясните свой выбор. Если причин несколько, указывайте главную. Список ассоциаций и обобщений: сезон состоит из нескольких соревнований, участник регистрируется, участнику присваивается порядковый номер, номер объявляется, участники делятся на группы, соревнование состоит из нескольких упражнений, соревнование проводится в нескольких секторах, в каждом секторе выполняется несколько упражнений, каждому сектору приписываются несколько судей, упражнения бывают сольными и групповыми, зачитываются оценки, максимальная оценка отбрасывается, минимальная оценка отбрасывается, выступления проводятся, лига состоит из нескольких команд, команда состоит из нескольких спортсменов, попытка упражнения производится участником, попытка получает несколько оценок от судей, призы выдаются в соответствии с набранными очками.

12.12. На рис. У12.4 приведена неполная диаграмма классов для системы подсчета очков. Ассоциация между соревнованием и упражнением не является производной, потому что событие может быть отнесено к соревнованию до

того, как ему будет назначен конкретный сектор. Покажите, каким образом можно с помощью этой модели проследить следующие запросы. Для выражения запросов используйте комбинацию OCL (глава 3) и псевдокода.

- 1) (2) Найти всех членов данной команды.
- 2) (6) Найти, какие упражнения проводились в данном сезоне несколько раз.
- 3) (6) Найти суммарное количество очков, полученных участником за данное упражнение на данном соревновании.
- 4) (6) Найти среднее по команде и по всем упражнениям значение очков в данном сезоне.
- 5) (6) Найти среднее количество очков, набранных данным участником по всем упражнениям на данных соревнованиях.
- 6) (6) Найти среднее по команде количество очков по данному упражнению на данных соревнованиях.
- 7) (4) Найти множество всех участников каких-либо упражнений в данном сезоне.
- 8) (7) Найти множество всех участников, которые участвовали во всех соревнованиях в данном сезоне.
- 9) (6) Найти всех судей, которые судили данное упражнение в данном сезоне.
- 10) (6) Найти судью, который присудил наименьшее количество очков в данном соревновании.
- 11) (6) Найти судью, который присудил наименьшее количество очков за данное упражнение.
- 12) (7) Изменить диаграмму таким образом, чтобы можно было найти всех участников, зарегистрированных на какое-либо упражнение.

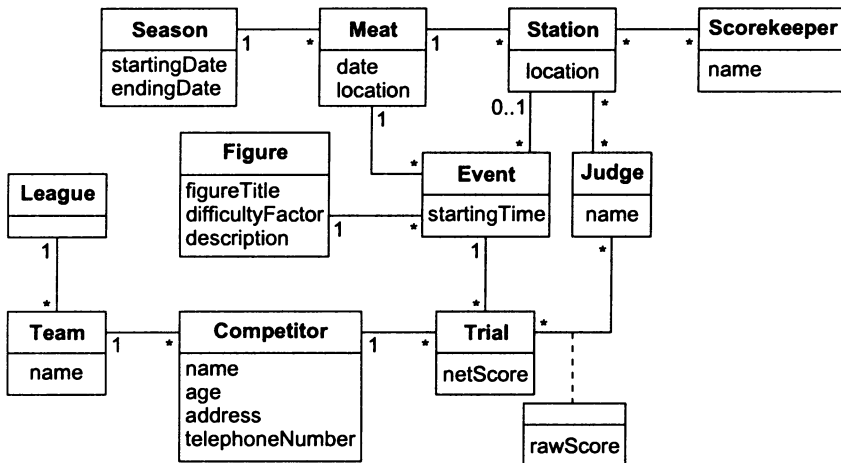


Рис. У12.4. Неполная диаграмма классов системы подсчета очков

12.13. (5) Для каких классов требуются диаграммы состояний? Опишите некоторые состояния и события.

12.14. (7) Отредактируйте диаграммы на рис. У12.5–У12.8 таким образом, чтобы исключить тернарные ассоциации. В некоторых случаях вам придется превращать ассоциации в классы.

На рис. У12.5 изображено отношение между классами *Doctor* (Доктор), *Patient* (Пациент) и *DateTime* (ДатаИВремя), с которым можно столкнуться в системах, используемых клиниками, в состав персонала которых входит несколько врачей. Комбинация *DateTime* и *Patient* уникальна, как и *DateTime* и *Doctor*.

На рис. У12.6 изображено отношение между классами *Student* (Студент), *Professor* (Профессор) и *University* (Университет), которое может использоваться для описания связей между студентами, слушающими лекции, и преподавателями, читающими лекции в разных университетах. Отношение состоит в том, что студент слушает один или несколько курсов у профессора в университете. Комбинация *Student* + *Professor* + *University* уникальна.

На рисунок У12.7 изображено отношение, выражающее порядок нумерации кресел на концерте. Комбинация *Concert* + *Seat* уникальна.

Рис. У12.8 выражает связность направленного графа. Каждая дуга направленного графа соединяет ровно две вершины в определенном порядке. Конкретную пару вершин может соединять несколько дуг. Атрибут *Edge* (Дуга) является уникальным для отношения.

В каждом случае вы должны постараться как можно точнее передать суть и сравнить достоинства и недостатки исходной и конечной моделей.

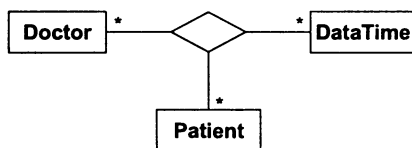


Рис. 12.5. Тернарная ассоциация Doctor, Patient и DateTime

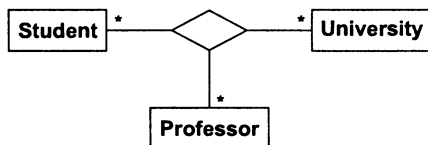


Рис. 12.6. Тернарная ассоциация Student, Professor и University

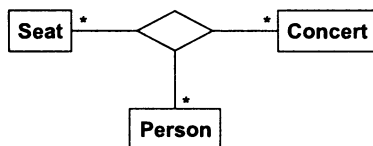


Рис. 12.7. Тернарная ассоциация Seat, Person и Concert

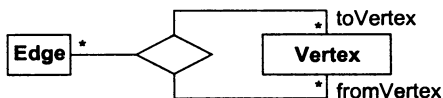


Рис. 12.8. Тернарная ассоциация для направленного графа

- 12.15. (9) На рис. У12.9 приведены требования к системе управления документами. Исходный вариант модели приведен на рис. У12.10. Обратите внимание на недостатки модели.

Develop software for managing professional records of papers, books, journals, notes, and computer files. The system must be able to record authors of published works in the appropriate order, name of work, date of publication, publisher, publisher city, an abstract, as well as a comment. The software must be able to group published works into various categories that are defined by the user to facilitate searching. The user must be able to assign a quality indicator of the perceived value of each work.

Only some of the papers in each issue of a journal may be of interest. It would also be helpful to be able to sections or even individual pages of a work.

Рис. У12.9. Требования к системе управления документами

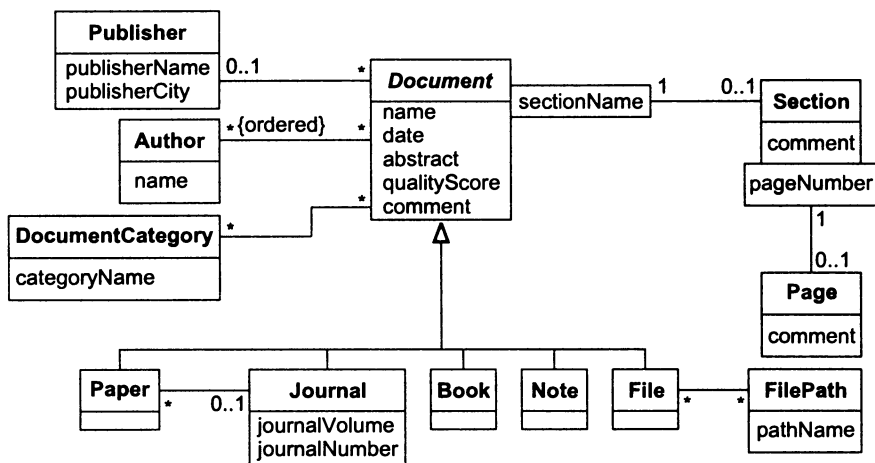


Рис. У12.10. Исходная модель системы управления документами

- Подклассы отличаются друг от друга не слишком сильно. набросок статьи — это документ или заметка? Каким образом мы должны работать с документом, который одновременно является файлом и лежит в папке? Каким образом следует представить сведения о слайдах для презентаций?
- Мы хотим иметь возможность обрабатывать стандартные комментарии (применимые ко множеству документов и выбираемые из возможных вариантов) и пользовательские комментарии (применяемые к одному документу и вводимые пользователем вручную).
- Мы должны иметь возможность комментировать пронумерованную страницу без наличия разделов.

Усовершенствуйте модель, сделав ее более абстрактной. (Совет: вы должны сформулировать универсальные классы, обозначающие размещение, свойства документа и комментарии. Структуру документа можно адекватно представить иерархией.)

Упражнения 12.16–12.19 связаны между собой. Выполняйте их последовательно. Ниже приведена предварительная спецификация программы-планировщика.

Программа-планировщик должна обеспечивать следующие функции: устраивать совещания, планировать встречи, планировать задачи, отслеживать выходные дни, праздники и отпуска.

Планировщик работает в сети, доступ к которой имеет множество пользователей. Каждый пользователь может иметь свое расписание. Расписание содержит множество записей. Большинство записей принадлежат только одному расписанию, однако запись о совещании может одновременно присутствовать в нескольких расписаниях.

Записи бывают четырех типов: совещания, встречи, задачи и праздники. Совещания и встречи занимают не более одного дня и характеризуются временем начала и временем окончания. Задачи и выходные могут тянуться несколько дней и характеризуются датами начала и окончания. Любая запись может циклически повторяться. Информация о повторе включает частоту повторения, дату начала и дату окончания.

12.16. (3) Ниже приведен список потенциальных классов. Исключите из него лишние классы по причинам, описанным в этой главе. Обоснуйте свой выбор. Если причин несколько, указывайте самую главную. Список классов: программа-планировщик, функция, совещание, встреча, задача, выходной, отпуск, планировщик, сеть, пользователь, расписание, запись, запись о совещании, день, время начала, время окончания, дата начала, дата окончания, информация о повторении.

12.17. (3) Подготовьте словарь данных для оставшихся после предыдущего упражнения классов.

12.18. (4) Ниже приведен список потенциальных ассоциаций и обобщений для программы-планировщика. Подготовьте список ассоциаций и обобщений, которые нужно исключить или переименовать по любой из причин, указанных в этой главе. Объясните свой выбор. Если причин несколько, приводите основную. Потенциальные ассоциации и обобщения:

- программа-планировщик поддерживает следующие функции;
- планировщик работает в сети, доступ к которой имеет множество пользователей;
- пользователь может иметь расписание;
- расписание содержит множество записей;
- записи принадлежат одному расписанию;
- запись о совещании может присутствовать в нескольких расписаниях;

- совещания и встречи занимают не более одного дня и характеризуются временем начала и временем окончания;
- задачи и выходные могут занимать несколько дней и характеризуются датами начала и окончания.

12.19. (5) Постройте модель классов программы-планировщика.

Упражнения 12.20—12.23 связаны между собой. Выполняйте их последовательно. Ниже приведены требования к совещаниям, расширяющие модель программы-планировщика из предыдущих упражнений.

Программа-планировщик способствует проведению совещаний. Когда пользователь (председатель) организует совещание, программа помещает запись о нем в расписание каждого из приглашенных сотрудников. При помощи планировщика председатель резервирует помещение для совещания, указывает участников и подбирает время, когда все они свободны. Председатель может указывать, является ли присутствие на совещании обязательным или желательным для каждого конкретного участника. Система отслеживает статус принятия приглашения для каждого из участников (согласился ли он участвовать в совещании).

Планировщик занимается уведомлением о совещании. В процессе организации совещания планировщик рассылает приглашения всем участникам, которые могут просматривать информацию о совещании. Каждый приглашенный может принять или отказаться, а также отменить принятие приглашения. Система обеспечивает уведомление участников в случае изменения даты или времени совещания, а также в случае его отмены.

12.20. (3) Ниже приведен список потенциальных классов. Подготовьте список классов, которые должны быть исключены из модели по причинам, указанным в этой главе. Объясните свой выбор. Если причин несколько, приводите основную. Потенциальные классы: программа-планировщик, совещание, пользователь, председатель, программа, запись о совещании, расписание, участник, планировщик, помещение, время, все, присутствие, статус, уведомление о совещании, приглашение, информация о совещании, приглашенный, уведомление.

12.21. (3) Подготовьте словарь данных для классов, оставшихся после выполнения предыдущего упражнения.

12.22. (4) Ниже приведен список потенциальных ассоциаций и обобщений. Подготовьте список ассоциаций и обобщений, которые нужно исключить или переименовать по любой из причин, указанных в этой главе. Объясните свой выбор. Если причин несколько, приводите основную. Потенциальные ассоциации и обобщения:

- программа-планировщик способствует проведению совещаний;
- пользователь (председатель) организует совещание;
- программа помещает запись о совещании в расписание каждого участника;
- с помощью планировщика председатель резервирует помещение, выбирает участников и подбирает время, когда все они свободны;

- председатель может указывать, является ли присутствие на совещании обязательным или желательным для каждого конкретного участника;
- система отслеживает статус принятия приглашения для каждого из участников;
- планировщик занимается уведомлением о совещании;
- планировщик рассылает приглашения всем участникам, которые могут просматривать информацию о совещании;
- система обеспечивает уведомление участников в случае изменения даты или времени совещания, а также в случае его отмены.

12.23. (7) Сконструируйте модель классов для расширения программы-планировщика. Ваш ответ должен решать задачу из упражнения 12.19. В нашей модели, описанной в ответе к упражнению 12.19, невозможно было определить, какому пользователю принадлежит некоторая запись. (Совет: ассоциации между председателем и участником из расширенных требований следует согласовать с ассоциацией между классами *Schedule* и *Entry* из требований упражнений 12.16–12.19).

В этой главе, завершающей рассказ об этапе анализа, рассмотрены вопросы, связанные с добавлением основных артефактов приложения в модель предметной области из предыдущей главы. Эти артефакты исследуются на этапе анализа, потому что они важны для приложения, видимы пользователям и должны быть одобрены ими. Вообще говоря, классы приложения не обязаны присутствовать в предметной области, но их можно получить из вариантов использования.

13.1. Модель взаимодействия приложения

Большинство моделей предметной области являются статическими. Операции в этих моделях не отражаются, потому что предметная область как целое обычно ничего не *выполняет*. Суть моделирования предметной области состоит в том, чтобы охватить внутренние концепции. Завершив разработку модели предметной области, мы переходим к рассмотрению взаимодействия.

Моделирование взаимодействия следует начинать с определения внешней границы системы. Затем следует выделить варианты использования и детально проработать их в сценариях и на диаграммах последовательности. Для сложных и неочевидных вариантов использования нужно построить диаграммы деятельности. После того как вы полностью разберетесь в вариантах использования, их можно упорядочить при помощи отношений. Наконец, необходимо провести проверку модели взаимодействия по модели классов предметной области: несогласованности между ними быть не должно.

Модель взаимодействия для приложения строится в несколько этапов.

1. Определить границу системы (раздел 13.1.1).
2. Выделить действующие лица (раздел 13.1.2).
3. Выделить варианты использования (раздел 13.1.3).

4. Выделить начальные и конечные события (раздел 13.1.4).
5. Подготовить типовые сценарии (раздел 13.1.5).
6. Добавить сценарии, описывающие вариации и исключительные ситуации (раздел 13.1.6).
7. Выделить внешние события (раздел 13.1.7).
8. Построить диаграммы деятельности для сложных вариантов использования (раздел 13.1.8).
9. Структурировать действующие лица и варианты использования (раздел 13.1.9).
10. Выполнить проверку по модели классов предметной области (раздел 13.1.10).

13.1.1. Определение границы системы

Для описания функциональности нужно точно знать область приложения, то есть границы системы. Вы должны решить, что будет входить в вашу систему, а что — нет (последнее даже более важно). Если граница системы очерчена корректно, во всех взаимодействиях вы сможете рассматривать систему как черный ящик — единый объект, внутренние особенности которого скрыты от внешнего взгляда и могут быть реализованы по-разному. В процессе анализа вы должны определить назначение системы и ее вид с точки зрения действующих лиц. В процессе проектирования вы сможете менять внутреннюю реализацию системы при условии сохранения внешнего поведения.

Обычно людей не следует рассматривать как часть системы, если только вы не занимаетесь моделированием организации (например, частной компании или правительственного учреждения). Люди — это действующие лица, которые должны взаимодействовать с системой, но их действия не управляются ею. Напротив, вы как разработчик системы должны учитывать, что люди могут ошибаться.

Пример с банкоматом. В исходной постановке задачи из главы 11 говорится, что вы должны «разработать программное обеспечение компьютеризированной банковской сети, включающей кассиров и банкоматы». На данном этапе важно обеспечить «бесшовное» согласование транзакций, проводимых на кассах и с помощью банкоматов. С точки зрения пользователя любой из этих методов должен давать один и тот же конечный результат. Однако коммерческая практика диктует, что приложение для банкоматов существует отдельно от приложения для кассиров, потому что первое работает с несколькими банками, тогда как второе используется внутри одного банка. Оба приложения основаны на общей модели предметной области, но каждое из них имеет свою собственную модель приложения. В этой главе мы будем заниматься только поведением банкомата, а о кассирах пока забудем.

13.1.2. Идентификация действующих лиц

После определения границ системы вы должны идентифицировать внешние объекты, непосредственно взаимодействующие с системой. Это и будут *действующие лица*. К их числу принадлежат люди, внешние устройства и другие программные системы. Самым важным качеством действующих лиц является то, что они

не контролируются приложением, а их поведение должно считаться непредсказуемым. Несмотря на то что может существовать какая-то ожидаемая последовательность действий действующего лица, приложение должно быть достаточно устойчивым, чтобы выдерживать нарушение этой последовательности.

В процессе поиска действующих лиц нас интересуют не индивидуальные сущности, а сущности с архетипичным поведением. Каждое действующее лицо представляет идеализированного пользователя, который задействует какое-либо подмножество функциональности системы. Изучите каждый внешний объект и выясните, не характеризуется ли он существенно различными видами поведения. Действующее лицо — это один вариант поведения по отношению к системе, а один внешний объект может соответствовать нескольким действующим лицам. С другой стороны, разные типы внешних объектов могут описываться одним действующим лицом.

Пример с банкоматом. Один человек может одновременно быть кассиром и клиентом одного и того же банка. Это интересное совпадение, которое, однако, чаще всего не является важным. По отношению к банку человек всегда выступает либо в одной, либо в другой роли. Для приложения банкомата действующими лицами будут *Customer* (Клиент), *Bank* (Банк) и *Consortium* (Консорциум).

13.1.3. Идентификация вариантов использования

Для каждого действующего лица необходимо перечислить разные способы использования системы. Каждый из этих способов называется *вариантом использования* (use case). Варианты использования разбивают функциональность системы на несколько дискретных единиц. Любое поведение системы должно принадлежать какому-либо варианту использования. Иногда могут возникнуть проблемы с отнесением какого-либо пограничного поведения к тому или иному варианту использования. При разбиении всегда следует помнить о том, что всегда существуют пограничные варианты. В этом случае решение можно принять произвольно.

Каждый вариант использования должен представлять собой некий сервис (услугу), предоставляемый системой, то есть нечто ценное для действующего лица. Все варианты использования следует рассматривать на одном уровне детализации. Например, если один из вариантов использования для банка называется *запросить ссуду*, то другой вариант не должен называться *снять наличные с депозитного счета через банкомат*. Второе название слишком детализированное по сравнению с первым. Лучше назвать этот вариант просто *снять деньги*. Постарайтесь сосредоточиться на основной цели варианта использования и не принимать решений по его реализации.

На этом этапе вы можете нарисовать предварительную диаграмму вариантов использования. На ней следует показать действующие лица и варианты использования, соединив их между собой. Обычно вариант использования можно соединить с действующим лицом, которое его инициирует, но, вообще говоря, в нем могут участвовать и другие действующие лица. Не беспокойтесь о том, что вы можете пропустить их. В процессе проработки вариантов использования это станет очевидно. Кроме того, для каждого варианта использования полезно сформулировать его краткое описание в одно-два предложения.

Пример с банкоматом. На рис. 13.1 показаны варианты использования, краткое описание которых приведено в списке ниже.

- *Initiate session* (Инициализация сеанса). АТМ определяет личность пользователя и предлагает ему список счетов и действий.
- *Query account* (Опрос счета). Система предоставляет общие сведения о счете, такие как текущий баланс, дату последней транзакции, дату отправки последнего отчета по почте.
- *Process transaction* (Обработка транзакции). Система банкомата выполняет действие, влияющее на баланс счета, такое как вклад, снятие и перевод. Банкомат гарантирует, что все завершённые транзакции в конечном итоге записываются в базу данных банка.
- *Transmit data* (Передача данных). Банкомат использует возможности консорциума для взаимодействия с компьютерами соответствующего банка.

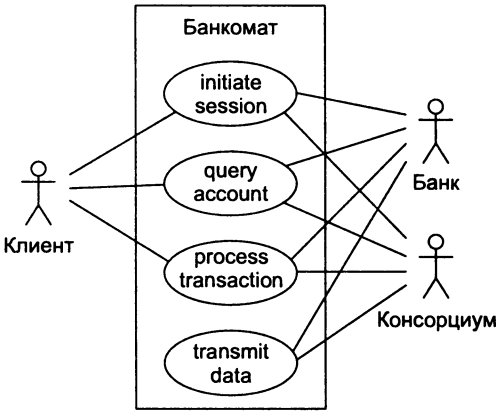


Рис. 13.1. Диаграмма вариантов использования для банкомата

13.1.4. Идентификация начальных и конечных событий

Варианты использования разбивают функциональность системы на дискретные части и показывают действующие лица, использующие каждую из этих частей. Однако поведение системы демонстрируется в них недостаточно четко. Для понимания поведения необходимо знать последовательности выполнения, соответствующие каждому из вариантов использования. Начинать их анализ следует с поиска событий, инициирующих каждый из вариантов использования. Определите, какое лицо инициирует вариант использования, и определите событие, которое оно для этого передает системе. Во многих случаях начальным событием является запрос некоторой услуги, предоставляемой системой. В других случаях начальным событием является происшествие, запускающее цепочку действий. Дайте этому событию осмысленное название, но пока не пытайтесь определить полный список его параметров.

Кроме того, следует определить конечное событие или группу событий, а также общие рамки событий, которые должны быть включены в каждый из вариантов использования. Например, вариант использования *запрос ссуды* может продолжаться до тех пор, пока просьба не будет подана, пока ссуда не будет выдана или отклонена или пока ссуда не будет погашена и закрыта. Любой из этих вариантов вполне приемлем. Разработчик модели должен определить границы варианта использования, установив его конечное событие.

Пример с банкоматом. Здесь мы приводим начальное и конечное событие для каждого варианта использования.

- *Initiate session* (Инициализация сеанса). Начальным событием является помещение клиентом банковской карты в банкомат. Конечных событий может быть два: либо система оставляет банковскую карту себе, либо возвращает ее обратно клиенту.
- *Query account* (Опрос счета). Начальное событие: клиент запрашивает данные о состоянии счета. Конечное событие: выдача необходимых сведений клиенту.
- *Process transaction* (Обработка транзакции). Начальное событие: клиент инициирует транзакцию. Конечных событий может быть два: завершение или откат транзакции.
- *Transmit data* (Передача данных). Начальным событием может быть запрос клиентом данных о состоянии счета. Кроме того, передача данных может быть инициирована после устранения неполадок в сети или перебоев с питанием. Конечное событие: успешная передача данных.

13.1.5. Подготовка типовых сценариев

Для каждого варианта использования нужно подготовить один или несколько типичных сценариев, чтобы почувствовать ожидаемое поведение системы. Эти сценарии описывают основные взаимодействия, форматы внешних отображаемых данных, а также обмен информацией. *Сценарием* (scenario) называется последовательность событий на множестве взаимодействующих объектов. Анализ следует осуществлять в терминах примеров взаимодействия, а не расписывать наиболее общие варианты. Таким образом, вы будете уверены, что не пропустите важные этапы и что общая последовательность взаимодействия получится корректной.

Для большинства задач логическая корректность зависит от взаимной последовательности взаимодействий, а не от конкретных временных промежутков между ними. (Системы реального времени предъявляют и временные требования к взаимодействиям, но мы в своей книге не рассматриваем такие системы.)

Иногда описание задачи полностью задает последовательность взаимодействий, но в большинстве случаев вам придется придумывать ее (или, по крайней мере, конкретизировать). Например, в постановке задачи о банкомате говорится о необходимости получения данных о транзакции от пользователя, но не указывается, какие конкретно параметры нужно у него спрашивать и в какой последовательности. Старайтесь не углубляться в подобные тонкости на этапе анализа.

Для большинства приложений порядок ввода выходных данных не имеет особой важности и может быть отложен до этапа проектирования.

Подготовьте сценарии для типовых ситуаций — взаимодействий без необычных входных параметров и ошибочных ситуаций. Событием является обмен информацией между объектом системы и внешним агентом (пользователем, датчиком или другой задачей). Параметром события является передаваемая информация. Например, параметром события «введен пароль» является сам введенный пароль. События без параметров тоже имеют значение и достаточно распространены. Само осуществление события является информацией. Для каждого события необходимо указать вызвавшее его лицо (систему, пользователя или иного внешнего агента) и параметры события.

Пример с банкоматом. На рис. 13.2 приведены типовые сценарии для каждого из вариантов использования.

<i>Инициация сеанса</i>	Банкомат просит пользователя вставить кредитную карту. Пользователь вставляет кредитную карту. Банкомат принимает карту и считывает ее серийный номер. Банкомат запрашивает пароль. Пользователь вводит «1234». Банкомат проверяет пароль, связываясь с консорциумом и банком. Банкомат выводит меню действий над счетами и команд. ... Пользователь выбирает команду завершения сеанса. Банкомат печатает квитанцию, возвращает карту и просит пользователя взять ее из банкомата. Пользователь берет квитанцию и карту. Банкомат просит пользователя вставить кредитную карту.
<i>Запрос счета</i>	Банкомат выводит меню счетов и команд. Пользователь выбирает запрос счета. Банкомат связывается с консорциумом и банком, которые предоставляют данные. Банкомат выводит данные о счете. Банкомат выводит меню счетов и команд.
<i>Обработка транзакции</i>	Банкомат выводит меню счетов и команд. Пользователь выбирает снятие денег со счета. Банкомат запрашивает снимаемую сумму. Пользователь вводит \$100. Банкомат проверяет сумму на превышение лимита выдачи наличных денег. Банкомат связывается с консорциумом и банком для проверки наличия достаточной суммы на счете. Банкомат выдает наличные деньги и просит пользователя забрать их. Пользователь берет наличные деньги. Банкомат выводит меню счетов и команд.
<i>Передача данных</i>	Банкомат запрашивает данные о счете, связываясь с консорциумом. Консорциум принимает запрос и направляет его в соответствующий банк. Банк принимает запрос и выдает требуемую информацию. Банк отправляет информацию в консорциум. Консорциум направляет информацию банкомату.

Рис. 13.2. Типовые сценарии для банкомата

13.1.6. Нетипичные сценарии и исключительные ситуации

После разработки типовых сценариев необходимо рассмотреть особые ситуации, такие как отсутствие введенных значений, ввод минимального и максимального значений, ввод одинаковых значений. Затем нужно рассмотреть ошибочные ситуации, включая ввод неправильных значений и отсутствие отклика. Для многих интерактивных приложений обработка ошибок является наиболее сложной частью процесса разработки. Необходимо предоставлять пользователю возможность отменить операцию или откатиться к четко определенной начальной точке каждого этапа. Наконец, нужно рассмотреть дополнительные взаимодействия, которые могут пересекаться с базовыми (такие как обращение к справочной системе или запрос сведений о состоянии).

Пример с банкоматом. Рассмотрим некоторые нетипичные сценарии и исключительные ситуации. Мы могли бы подготовить сценарии для каждого из них, но не хотим пока углубляться в детали (см. упражнения в конце этой главы).

- Банкомат не может прочесть карту.
- Срок действия карты истек.
- Тайм-аут при ожидании банкоматом ответа клиента.
- Сумма введена неверно.
- В банкомате кончились наличные или бумага.
- Линии связи не работают.
- Транзакция отклонена из-за подозрительной схемы использования карты.

Дополнительные сценарии описывают работу административных частей системы банкомата: авторизацию новых карт, добавление банков к консорциуму, получение журналов транзакций. Эти аспекты мы раскрывать не будем.

13.1.7. Выделение внешних событий

Проанализируйте все разработанные сценарии и выделите все внешние события: ввод данных, принятие решений, прерывания и взаимодействия с другими пользователями и внешними устройствами. Событие может вызывать действия целевого объекта. Этапы внутренних вычислений не являются событиями, за исключением расчетов, в ходе которых осуществляется взаимодействие с внешним миром. При помощи сценариев вы можете отыскать типовые события, но не забудьте и про нетипичные события и ошибочные ситуации.

Передача информации объекту является событием. Например, «введен пароль» — это сообщение, переданное от внешнего агента *User* (Пользователь) объекту приложения *АТМ* (*Банкомат*). Некоторые потоки информации присутствуют в модели неявным образом. Многие события характеризуются определенными параметрами.

Сгруппируйте под одинаковым названием события, оказывающие одинаковое влияние на поток управления, даже если значения их параметров отличаются. Например, «введен пароль» — это событие, параметром которого является

значение пароля. Выбор значения пароля не влияет на поток управления, поэтому события с разными паролями являются экземплярами одного и того же типа событий. Аналогичным образом, «выдача наличных» также является событием независимо от суммы (параметра). Экземпляры событий, значения которых влияют на поток управления, должны быть отнесены к разным типам событий. «Правильный счет», «неверный счет» и «неверный пароль» — разные события, которые не следует группировать под названием «состояние карты».

Вы должны следить за ситуациями, когда различия количественных значений достаточно существенны для того, чтобы события можно было считать разными. Например, нажатие любой цифровой клавиши на клавиатуре можно считать событием, не зависящим от конкретной цифры, тогда как нажатие клавиши «ввод» можно рассматривать отдельно, так как приложение будет обрабатывать его не так, как нажатие на цифровые клавиши. Различие событий зависит от приложения.

Подготовьте диаграмму последовательности для каждого сценария. Диаграмма последовательности показывает участников взаимодействия и последовательность сообщений, которыми они обмениваются. Каждому участнику выделяется свой столбец. Диаграмма показывает отправителя и получателя каждого сообщения. Если в объекте участвует несколько объектов одного и того же класса, им следует присвоить разные номера. Изучив один столбец таблицы, вы можете определить события, непосредственно влияющие на конкретный объект. После этого вы можете сгруппировать события, отправляемые и принимаемые каждым классом.

Пример с банкоматом. На рис. 13.3 показана диаграмма последовательности для сценария варианта использования *process transaction* (обработка транзакции). На рис. 13.4 сгруппированы события. Стрелки указывают, какой объект является отправителем, а какой получателем (для каждого сообщения). Параметры событий на рис. 13.4 не показаны.

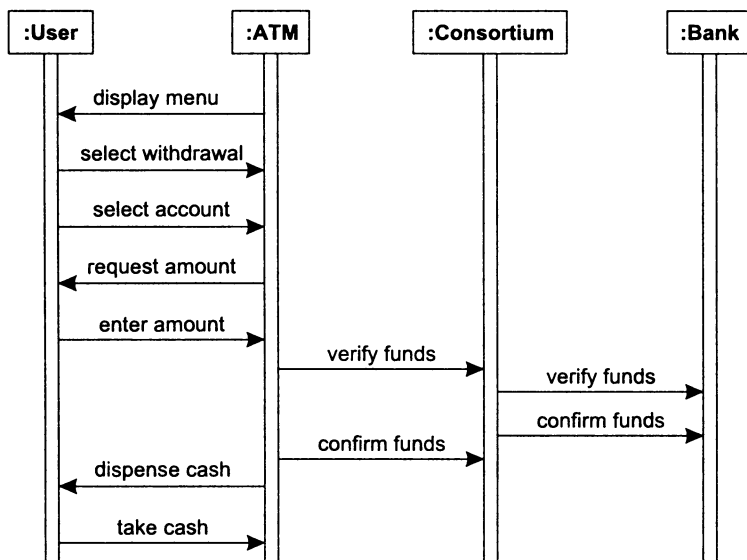


Рис. 13.3. Диаграмма последовательности для сценария «обработка транзакции»

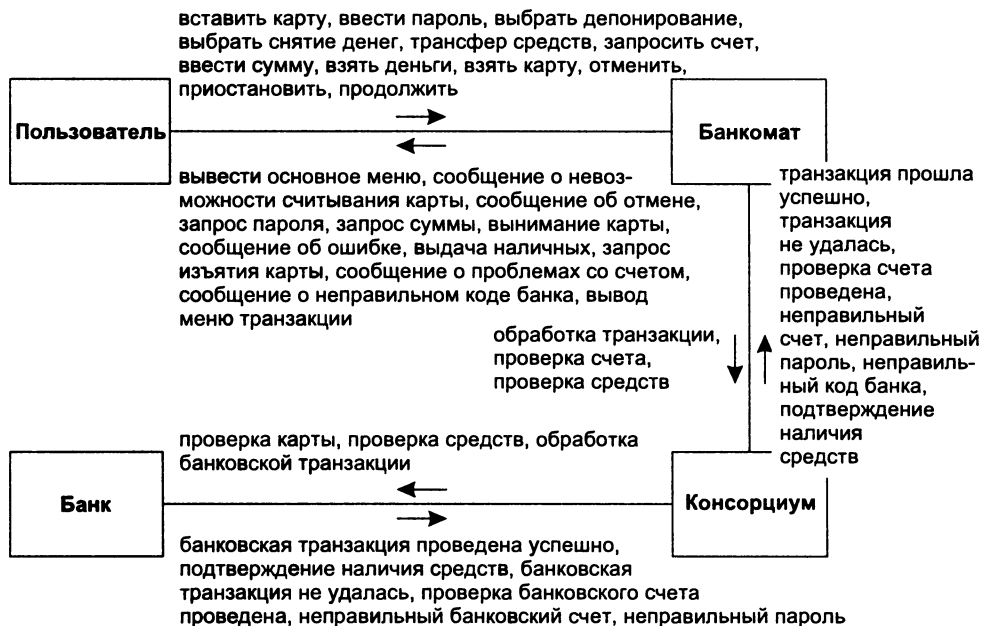


Рис. 13.4. События для примера с банкоматом

13.1.8. Подготовка диаграмм деятельности для сложных вариантов использования

Диаграммы последовательности описывают диалог и взаимодействие действующих лиц, но на них нельзя отразить имеющиеся альтернативы и принятые решения. Вам придется рисовать отдельную диаграмму для основного потока взаимодействия и отдельные диаграммы для каждой ошибочной ситуации и каждой точки принятия решения. Диаграммы деятельности позволяют объединить все это поведение благодаря документированию ветвлений и слияний потока управления. Диаграммы деятельности можно использовать для документирования бизнес-логики на этапе анализа, однако не стоит оправдывать ими ранний переход к реализации.

Пример с банкоматом. На рис. 13.5 показано, что помещение клиентом карты в банкомат может вызвать множество различных последствий. Некоторые варианты отклика указывают на наличие проблем с картой или счетом (банкомат не возвращает карту). Только успешное прохождение проверки разрешает продолжать работу с банкоматом.

13.1.9. Структурирование действующих лиц и вариантов использования

На следующем этапе нужно упорядочить варианты использования при помощи отношений включения, расширения и обобщения (см. главу 8). Это особенно полезно для больших и сложных систем. Как и с моделями классов и состояний, мы

откладываем структурирование до тех пор, пока не будут выписаны все базовые варианты использования. Если выполнить структурирование слишком рано, существует опасность искажения приложения из-за фиксации подсознательных соображений в структуре вариантов использования.

Обобщение можно применять и к действующим лицам. Например, *администратора* можно считать *оператором*, обладающим дополнительными привилегиями.

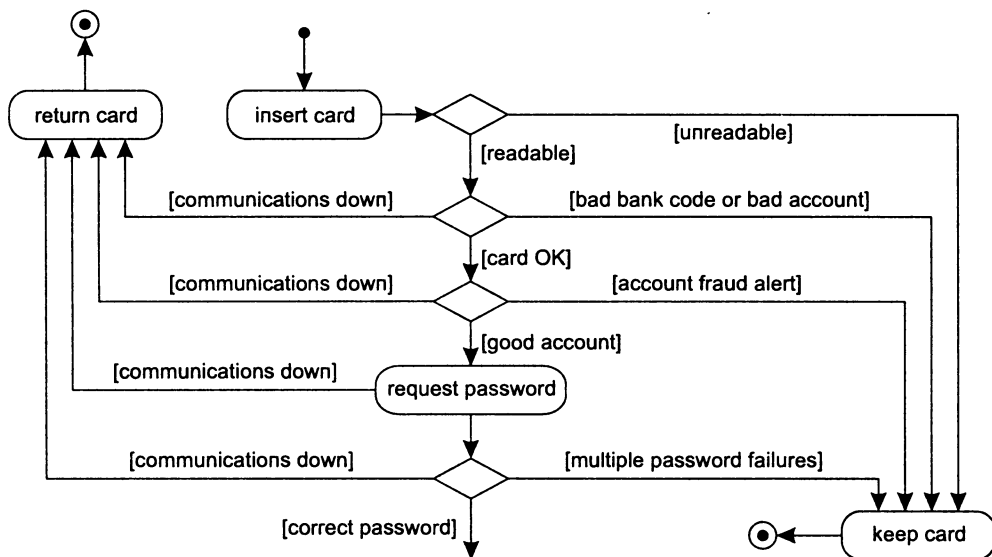


Рис. 13.5. Диаграмма деятельности для проверки карты

Пример с банкоматом. На рис. 13.6 показаны варианты использования, структурированные с применением отношения включения.

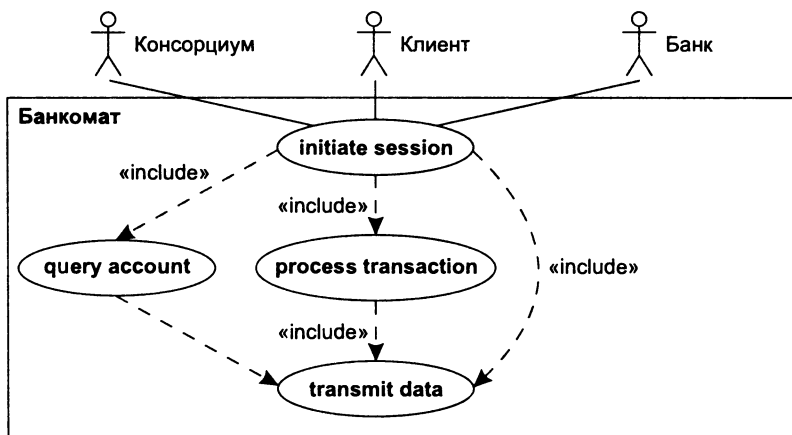


Рис. 13.6. Структурирование вариантов использования

13.1.10. Проверка по модели классов предметной области

На этом этапе модель предметной области и модель приложения должны быть уже хорошо согласованы друг с другом. Действующие лица, варианты использования и сценарии основаны на классах и концепциях модели предметной области. Вспомните, что одним из этапов построения этой модели является проверка маршрутов. На практике эта проверка — первый шаг к выделению вариантов использования.

Сверьте друг с другом модель приложения и модель предметной области, чтобы убедиться в их согласованности. Изучите сценарии и проверьте, что в модели предметной области имеются все необходимые данные. Убедитесь, что эта модель содержит все параметры событий.

13.2. Модель классов приложения

Классы приложения описывают само приложение, а не объекты реального мира, с которыми оно работает. Большинство классов приложения связаны с компьютерами и определяют восприятие приложения пользователями. Модель классов приложения строится в несколько этапов.

1. Указать интерфейсы пользователя (раздел 13.2.1).
2. Определить граничные классы (раздел 13.2.2).
3. Определить управляющие объекты (раздел 13.2.3).
4. Проверить модель классов по модели взаимодействия (раздел 13.2.4).

13.2.1. Определение интерфейсов пользователя

Большинство взаимодействий можно разделить на две части: логика приложения и пользовательский интерфейс. *Пользовательским интерфейсом* (user interface) называется объект или группа объектов, предоставляющих пользователю системы единую точку доступа к объектам предметной области, командам и параметрам приложения. В процессе анализа внимание уделяется прежде всего потокам информации и управления, а не формату представления. Одна и та же программная логика может принимать входные данные из командной строки, считывать из файла, интерпретировать как щелчки мышью, нажатия на сенсорные панели и кнопки, так и поступающие удаленные сигналы, при условии, что внешний интерфейс аккуратно отделен от внутренней части приложения.

На этапе анализа пользовательский интерфейс рассматривается достаточно грубо. Не следует беспокоиться о том, каким конкретно образом будет введена определенная порция данных. Вместо этого нужно попытаться определить команды, которые пользователь должен иметь возможность выполнять. *Командой* (command) называется крупномасштабный запрос некоторой услуги системы. Например, командами могут быть «забронировать билеты» и «найти поисковую строку в базе данных». Формат ввода информации и их запуска на выполнение

изменить относительно несложно, поэтому, в первую очередь, следует разработать сами команды.

Тем не менее вполне допустимо предварительно набросать интерфейсы, чтобы с их помощью визуализировать работу приложения и проверить, не было ли пропущено что-нибудь важное. Можно даже сделать имитацию интерфейса, чтобы пользователи могли попробовать поработать с ним и оценить его. Логiku приложения можно имитировать фиктивными процедурами. Отделение логики от пользовательского интерфейса дает вам возможность оценить ощущения от работы с этим интерфейсом, пока приложение все еще находится в стадии разработки.

Пример с банкоматом. На рис. 13.7 показан возможный вид интерфейса банкомата. Точные детали на этом этапе несущественны. Самое важное — это информация, которой пользователи обмениваются с системой. Иногда пример интерфейса помогает визуализировать работу приложения.

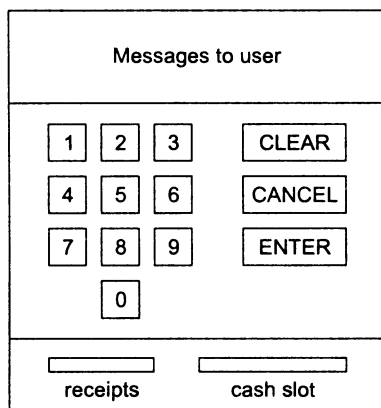


Рис. 13.7. Формат интерфейса банкомата

13.2.2. Определение пограничных классов

Система должна быть способна работать с информацией, принимаемой из внешних источников, но ее внутренняя структура не должна зависеть от этих источников. Часто оказывается полезно определить пограничные классы, которые будут изолировать внутреннюю часть системы от внешнего мира. Пограничным называется класс, который является посредником во взаимодействии системы с внешним источником. Такой класс должен уметь принимать данные в формате одного или нескольких внешних источников и преобразовывать их к формату, с которым работает внутренняя часть системы.

Пример с банкоматом. Полезно будет определить пограничные классы *CashCardBoundary* (ГраницаКарты) и *AccountBoundary* (ГраницаСчета), которые скроют взаимодействие банкомата с консорциумом. Этот интерфейс позволит сделать приложение более гибким и упростит его расширение для работы с несколькими консорциумами.

13.2.3. Определение управляющих объектов

Управляющий объект (controller) — это активный объект, осуществляющий управление внутри приложения. Он принимает сигналы из внешнего мира и от объектов системы, реагирует на них, вызывает операции на объектах системы и передает сигналы во внешний мир. Управляющий объект — это воплощение поведения в форме объекта. С поведением в такой форме проще работать и его проще изменять, чем простой код. В основе многих приложений лежат управляющие объекты, упорядочивающие поведение этих приложений.

Проектирование управляющего объекта сводится, по большей части, к разработке диаграммы состояний этого объекта. В модели классов приложения существование управляющих объектов тоже необходимо отразить, вместе с информацией, которую они обрабатывают, и ассоциациями, которые связывают их с другими объектами системы.

Пример с банкоматом. Из сценариев, показанных на рис. 13.2, следует, что банкомат имеет два главных управляющих цикла. Внешний цикл проверяет клиентов и счета. Внутренний цикл обслуживает транзакции. Каждый из этих циклов наиболее естественно воплотить в управляющем объекте.

13.2.4. Проверка по модели взаимодействия

Построив модель классов приложения, вернитесь к вариантам использования и подумайте о том, как они могут осуществляться этим приложением. Например, если пользователь передает приложению команды, параметры команды должны передаваться каким-либо объектом пользовательского интерфейса. Запрос на саму команду должен исходить из какого-либо управляющего объекта. При наличии корректных моделей классов предметной области и приложения вы должны быть способны имитировать варианты использования при помощи классов. Тут нужно мыслить в терминах прослеживания маршрутов, о чем рассказывалось в главе 3. Ручное моделирование помогает убедиться в том, что все составляющие находятся на своих местах.

Пример с банкоматом. На рис. 13.8 показана предварительная модель классов приложения и классы предметной области, с которыми она взаимодействует. Интерфейсов всего два: один для пользователей, другой — для консорциума. Эти классы в модели приложения представлены заглушками, потому что пока еще не ясно, каким образом они должны быть реализованы.

Обратите внимание, что пограничные классы делают структуру данных более плоской и объединяют информацию, полученную из разных классов предметной области. Для простоты количество пограничных классов и их отношений следует по возможности уменьшать.

Класс *TransactionController* (УправляющийОбъектТранзакций) обрабатывает запросы по счетам и собственно транзакции. Класс *SessionController* (УправляющийОбъектСеансов) управляет классами *ATMsession* (СеансБанкомата), каждый из которых обслуживает клиента. Каждый объект *ATMsession* может иметь, а может и не иметь корректных *CashCard* (БанковскаяКарта) и *Account* (Счет). Класс *SessionController* имеет атрибут *status* (состояние), который может принимать

значения *ready* (готов), *impaired* (ограниченный режим — кончилась бумага или наличные, но некоторые операции все равно могут быть выполнены) и *down* (отказ, например, линии связи). Система ведет журнал объектов *ControllerProblem* (НеполадкаУправляющихОбъектов) с указанием типов неполадок (отказ устройства считывания карт, кончилась бумага, кончились наличные, отказ линий связи и т. д.).

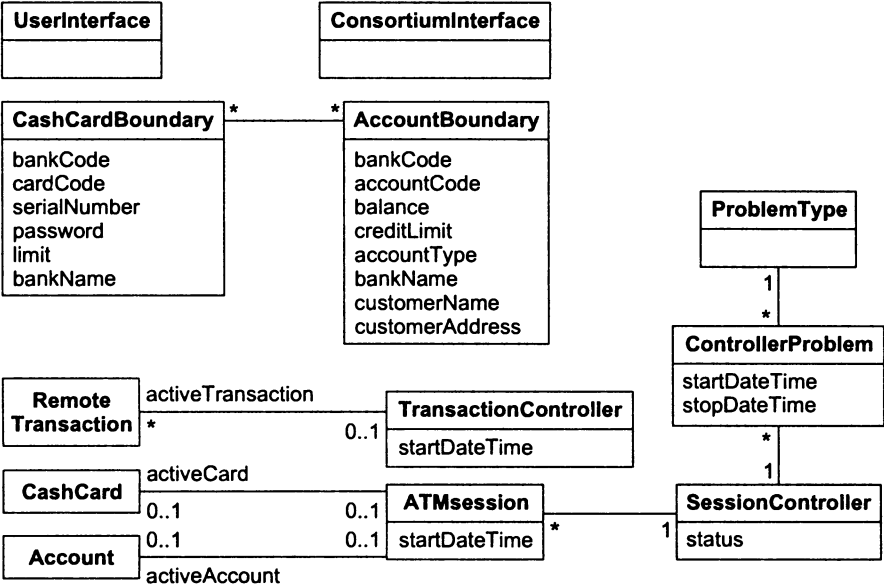


Рис. 13.8. Модель классов приложения

13.3. Модель состояний приложения

Модель состояний приложения описывает состояния классов приложения и, таким образом, дополняет модель состояний предметной области. Классы приложения с большей вероятностью продемонстрируют изменение своего состояния, то есть наблюдать их поведение важнее чем поведение классов предметной области.

Сначала нужно выделить классы приложения, обладающие несколькими состояниями, и с помощью модели взаимодействия определить события, которые влияют на эти состояния. Затем нужно упорядочить допустимые последовательности событий для каждого из классов на диаграмме состояний. После этого нужно проверить различные диаграммы состояний и убедиться, что одинаковые события на них действительно соответствуют друг другу. Наконец, диаграммы состояний нужно проверить по моделям классов и взаимодействия.

- Построение модели состояний приложения осуществляется в несколько этапов.
1. Выделить классы приложения, обладающие состояниями (раздел 13.3.1).
 2. Найти события (раздел 13.3.2).

3. Построить диаграммы состояний (раздел 13.3.3).
4. Проверить по другим диаграммам состояний (раздел 13.3.4).
5. Проверить по модели классов (раздел 13.3.5).
6. Проверить по модели взаимодействия (раздел 13.3.6).

13.3.1. Выделение классов приложения

Модель классов приложения состоит из «компьютерных» классов, видимых пользователям и важных для работы приложения. Вы должны рассмотреть каждый из этих классов и определить, обладает ли он несколькими разными состояниями. Чаще всего состояниями обладают классы пользовательского интерфейса и управляющих объектов. Пограничные классы, напротив, обычно являются статическими и осуществляют импорт и экспорт данных.

Пример с банкоматом. Классы пользовательского интерфейса не обладают состояниями. Возможно, это связано с тем, что наше понимание работы интерфейса недостаточно на данном этапе разработки. Пограничные классы также лишены состояний. Зато управляющие объекты характеризуются важными состояниями, которые мы рассмотрим ниже.

13.3.2. Поиск событий

Ранее вы подготовили набор сценариев для описания модели взаимодействия приложения. Теперь вы должны изучить эти сценарии и выделить из них события. Хотя эти сценарии не обязательно покрывают все возможные ситуации, они не дадут вам пропустить наиболее типичные взаимодействия и в них подчеркиваются наиболее важные события.

Обратите внимание на разницу между последовательностью этапов построения моделей состояний для предметной области и для приложения. В первом случае мы начинали с поиска состояний, после чего выделяли события. Дело в том, что модель предметной области концентрируется на данных. Данные группируются в состояния, которые подвержены влиянию событий. В модели приложения мы начинаем с поиска событий, а затем выделяем состояния. Для модели приложения важно прежде всего поведение, и поэтому варианты использования раскрываются сценариями, которые содержат события.

Пример с банкоматом. Вернемся к сценариям из модели взаимодействия приложения. Там мы обнаружим события: *insert card* (вставка карты), *enter password* (ввод пароля), *end session* (окончание сеанса) и *take card* (извлечение карты).

13.3.3. Построение диаграмм состояний

Следующий этап состоит в построении диаграммы состояний для каждого класса приложения, обладающего существенно зависящим от времени поведением. Выберите один из этих классов и возьмите его диаграмму последовательности. Расставьте события, в которых участвует класс, в виде графа, над дугами которого

напишите названия событий. Интервал между любыми двумя событиями будет состоянием. Дайте каждому состоянию конкретное имя, если оно получается осмысленным, а если нет — оставьте состояние безымянным. Объедините связанные между собой диаграммы последовательности в диаграмму состояний. Эта диаграмма будет последовательностью состояний и событий. Каждый сценарий или диаграмма последовательности определяют один маршрут на диаграмме состояний.

После этого займитесь выделением циклов на диаграмме. Если последовательность событий может повторяться до бесконечности, она образует цикл. Начальное и конечное состояния из цикла совпадают. Если объект «запоминает», что он уже сделал полный проход по циклу, эти два состояния уже не являются идентичными, а потому представление в виде простого цикла будет некорректно. По крайней мере одно состояние цикла должно иметь несколько исходящих переходов, или этот цикл никогда не завершится.

После выделения циклов объедините другие диаграммы последовательности с полученной диаграммой состояний. На каждой диаграмме последовательности найдите точку, в которой она отклоняется от остальных диаграмм. Эта точка соответствует имеющемуся на диаграмме состоянию. Присоедините новую последовательность событий к существующему состоянию в виде альтернативного маршрута. Изучая диаграммы последовательности, вы можете обнаружить дополнительные возможные события, которые могут осуществляться в каждом состоянии. Добавьте на диаграмму и их.

Самое сложное — определить, в какой точке альтернативный маршрут воссоединяется с уже представленными на диаграмме состояний. Два маршрута соединяются в одном состоянии, если объект «забывает», по какому из них он пришел в это состояние. Во многих случаях из знаний о приложении с очевидностью следует, что два состояния идентичны друг другу. Например, бросить две монеты по пять центов в торговый автомат — все равно, что бросить одну монету в десять центов.

Будьте аккуратны с маршрутами, которые кажутся идентичными, но при некоторых обстоятельствах могут отличаться. Например, некоторые системы повторяют последовательность ввода данных, если пользователь допускает ошибку, но через несколько попыток они прекращают повторять ввод. Повторяются одни и те же события, за тем исключением, что система помнит о прошлых ошибках. Эту разницу можно отразить при помощи параметра (например, *number of failures* — число ошибок), в котором будет храниться соответствующая информация. По крайней мере один переход должен зависеть от этого параметра.

Благодарное использование параметров и условных переходов может значительно упростить диаграммы состояний за счет смешивания информации о состояниях с данными. Диаграммы состояний, слишком сильно зависящие от данных, могут сбивать с толку и противоречить здравому смыслу. Другая альтернатива состоит в делении диаграммы состояний на две параллельные диаграммы, используя одну из них для основной последовательности, а другую — для

управляющей информации. Например, поддиаграмма, описывающая ситуацию, в которой пользователю дается возможность сделать одну ошибку, может иметь состояния *No error* и *One error* (Нет ошибок и Одна ошибка).

После рассмотрения всех типичных событий добавьте на диаграмму нетипичные сценарии и исключительные ситуации. Рассмотрите события, которые происходят в неудачные моменты: например, запрос на отмену транзакции, поступающий после начала ее обработки. Если пользователь (или иной внешний агент) не отвечает на запрос системы в должный срок, необходимо породить событие тайм-аута. Обработка пользовательских ошибок часто требует больше размышлений и кодирования, чем типовые последовательности. Обработка ошибок часто усложняет структуру программы, которая в противном случае была бы ясной и компактной, но без нее программы создавать нельзя.

Диаграмма состояний может считаться законченной, если она покрывает все сценарии и учитывает все события, которые могут повлиять на состояния. Вы можете использовать диаграмму состояний для рассмотрения новых сценариев, исследуя влияние событий, обработка которых еще не предусмотрена, на различные состояния. Вопросы типа «что если?» представляют хороший метод проверки полноты модели и ее способности корректно обрабатывать ошибки.

Для описания сложных взаимодействий с независимыми входными данными вы можете использовать вложенные диаграммы состояний (см. главу 6). В противном случае достаточно плоской структуры диаграммы. Описанную выше процедуру построения диаграмм состояния необходимо повторить для каждого класса, поведение которого зависит от времени.

Пример с банкоматом. На рис. 13.9 показана диаграмма состояний для управляющего объекта *SessionController* (УправляющийОбъектСеанса). Центр диаграммы описывает основную часть поведения, связанную с обработкой введенной карты и пароля. Обрыв связи может прервать обработку в любой момент. В этом случае банкомат возвращает карту (или удерживает ее, если обрыв связи сопровождается подозрительными обстоятельствами). После завершения транзакции распечатка чека осуществляется одновременно с возвратом карты. Пользователь может взять карту и чек в произвольном порядке.

На рис. 13.10 и 13.11 показана диаграмма состояний для объекта *TransactionController* (УправляющийОбъектТранзакции), который порождается *SessionController*. (Прочие поддиаграммы рис. 13.10 описаны в упражнениях.) Мы разделили *TransactionController* и *SessionController*, потому что они имеют существенно разное назначение. *SessionController* занимается проверкой пользователей, тогда как *TransactionController* обслуживает запросы по счетам и изменения баланса.

13.3.4. Проверка по другим диаграммам состояний

Проверяйте диаграмму состояний каждого класса на полноту и согласованность. Каждое событие должно иметь отправителя и получателя. В некоторых случаях они оба могут быть одним и тем же объектом. Состояния без предшествующих

или последующих состояний должны вызывать подозрение. Убедитесь, что они соответствуют начальным или конечным точкам последовательностей взаимодействия. Проследите результаты входного события от объекта к объекту через всю систему и убедитесь, что они соответствуют сценариям. Объектам присущ параллелизм. Остерегайтесь ошибок синхронизации, связанных с тем, что какое-либо событие происходит в неподходящий момент. Убедитесь, что одинаковые события на разных диаграммах соответствуют друг другу.

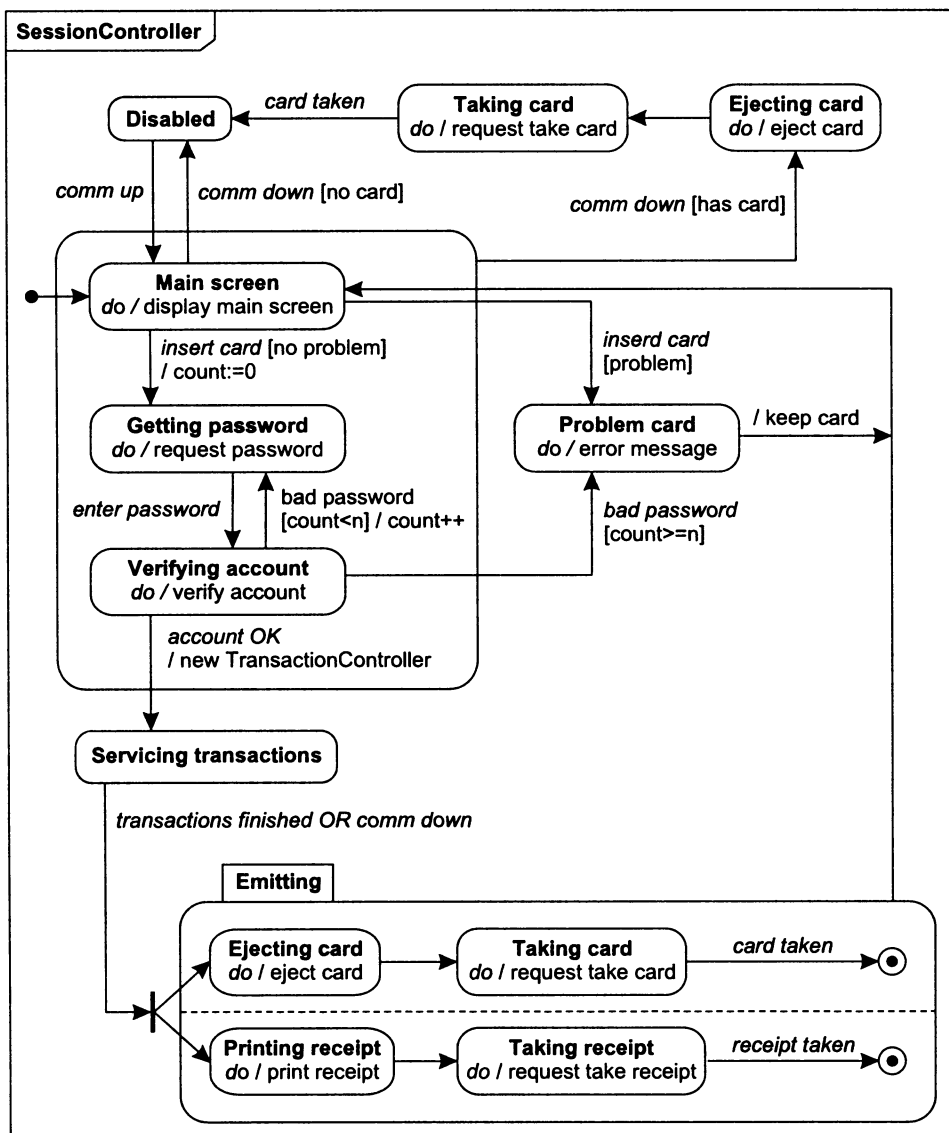


Рис. 13.9. Диаграмма состояний SessionController

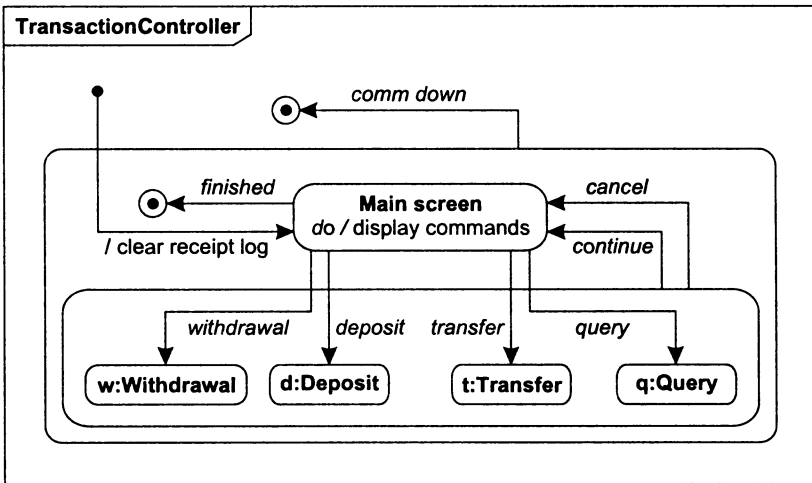


Рис. 13.10. Диаграмма состояний TransactionController

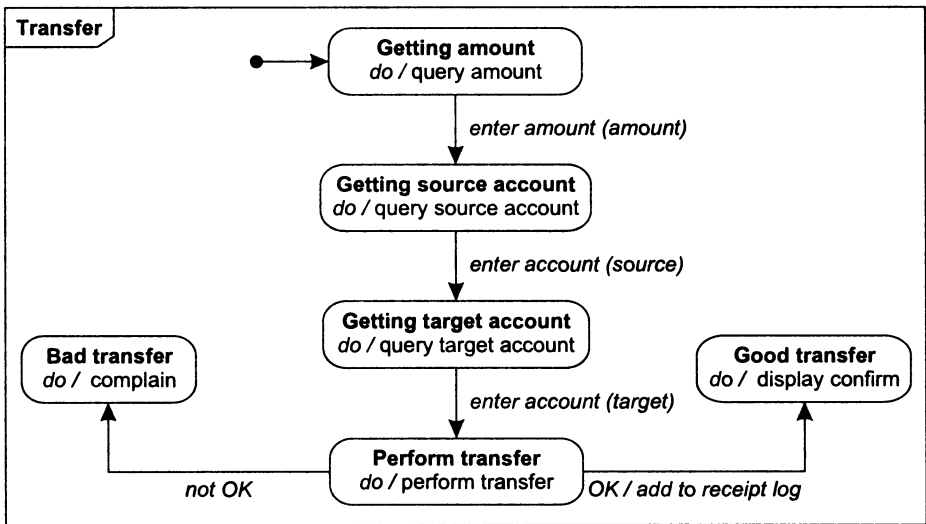


Рис. 13.11. Диаграмма, раскрывающая состояние Transfer

Пример с банкоматом. Управляющий объект *SessionController* вызывает *TransactionController*, а завершение *TransactionController* вызывает возобновление работы *SessionController*.

13.3.5. Проверка по модели классов

Проверьте, что диаграммы состояний согласованы с моделями классов предметной области и приложения.

Пример с банкоматом. Потенциально возможно одновременное обращение нескольких банкоматов к одному счету. Доступ к счету должен контролироваться

таким образом, чтобы только один банкомат мог обновлять его в какой-либо момент времени. Мы не будем углубляться в эту задачу.

13.3.6. Проверка по модели взаимодействия

Когда модель состояний будет готова, вернитесь назад и проверьте ее по сценариям модели взаимодействия. Имитируйте каждую последовательность поведения вручную и убедитесь, что диаграмма состояний дает корректное поведение. В случае обнаружения несоответствий нужно изменить либо диаграмму, либо сценарий. Иногда диаграмма состояний позволяет обнаружить ошибки в сценариях, поэтому не следует заранее предполагать, что они верны.

Затем возьмите модель состояний и проследите разрешенные маршруты. Они описывают все возможные сценарии. Спросите себя, есть ли смысл в этих сценариях. Если его нет, измените диаграмму состояний. Часто эта процедура позволяет обнаружить полезное поведение, которое было пропущено на предыдущих этапах. Обнаружение неожиданной информации, следующей из проекта, а также важных (и иногда кажущихся очевидными) свойств системы указывает на высокое качество этого проекта.

Пример с банкоматом. На данном этапе можно утверждать, что диаграммы состояний выглядят достаточно цельными и согласуются со сценариями.

13.4. Добавление операций

Наш стиль объектно-ориентированного анализа придает гораздо меньшее значение определению операций, чем традиционные методологии разработки, отталкивающиеся от программирования. Нам кажется, что список потенциально полезных операций бесконечен, и трудно выбрать момент, чтобы остановиться и прекратить добавлять их. Операции следуют из определенных источников, и как раз на этом этапе пришло время добавить их в модель. В главе 15 об операциях рассказывается подробно.

13.4.1. Операции из модели классов

Чтение и запись значений атрибутов и связей ассоциаций подразумеваются моделью классов. Показывать их явным образом нет необходимости. В процессе анализа предполагается, что все атрибуты и ассоциации доступны.

13.4.2. Операции из вариантов использования

Большая часть сложной функциональности системы проистекает из вариантов ее использования. В процессе конструирования модели взаимодействия варианты использования определяют деятельность. Часто деятельность соответствует операциям в модели классов.

Пример с банкоматом. Класс *Consortium* (Консорциум) характеризуется деятельностью *verifyBankCode* (проверитьКодБанка), а класс *Bank* (Банк) — дея-

тельностью *verifyPassword* (проверитьПароль). Рисунок 13.5 можно реализовать операцией *verifyCashCard* (проверитьКарту) в классе *ATM* (Банкомат).

13.4.3. Операции «по списку»

Иногда поведение классов в реальном мире предполагает некоторые операции. Мейер (Meyer [Meyer-97]) называет такие операции «операциями по списку» (shopping list operations), потому что они не зависят от приложения, а имеют значение сами по себе. Эти операции позволяют расширить определение класса по сравнению с потребностями текущей задачи.

Пример с банкоматом. К «операциям по списку» можно отнести:

- `account.close();`
- `bank.createSavingsAccount(customer):account;`
- `bank.createCheckingAccount(customer):account;`
- `bank.createCashCardAuth(customer):cashCardAuthorization;`
- `cashCardAuthorization.addAccount(account);`
- `cashCardAuthorization.removeAccount(account);`
- `cashCardAuthorization.close();`

13.4.4. Упрощение операций

Изучите модель классов на предмет наличия подобных операций и различных вариаций одной операции. Постарайтесь расширить определения операций таким образом, чтобы они включали эти вариации и особые случаи. Везде, где это возможно, используйте наследование для сокращения количества различных операций. При необходимости вводите новые суперклассы для упрощения операций, при условии, что их введение не оказывается «насильственным» и неестественным. Размещайте каждую операцию на нужном уровне иерархии классов. В результате такого уточнения модели количество операций сокращается и они становятся более мощными, но составлять их спецификацию проще, чем для исходных операций, потому что они более универсальные и общие.

Пример с банкоматом. Модель банкомата не требует упрощения. На рис. 13.12 мы добавили некоторые операции к модели классов предметной области банкомата из главы 12.

13.5. Резюме

Назначение этапа анализа состоит в том, чтобы постичь суть задачи в достаточной для построения корректного проекта степени. Хорошая аналитическая модель описывает важнейшие черты задачи, но не добавляет к ним артефактов реализации, которые раньше времени ограничили бы возможности проектирования.

Этап анализа делится на анализ предметной области и анализ приложения. Модель предметной области описывает общие знания о приложении. Такая модель

обычно включает модели классов и состояний, но очень редко — модель взаимодействия. Напротив, модель приложения отражает наиболее важные артефакты приложения, видимые пользователям и требующие их одобрения. Для приложения важнее всего модель взаимодействия, однако нельзя сбрасывать со счетов модели классов и состояний.

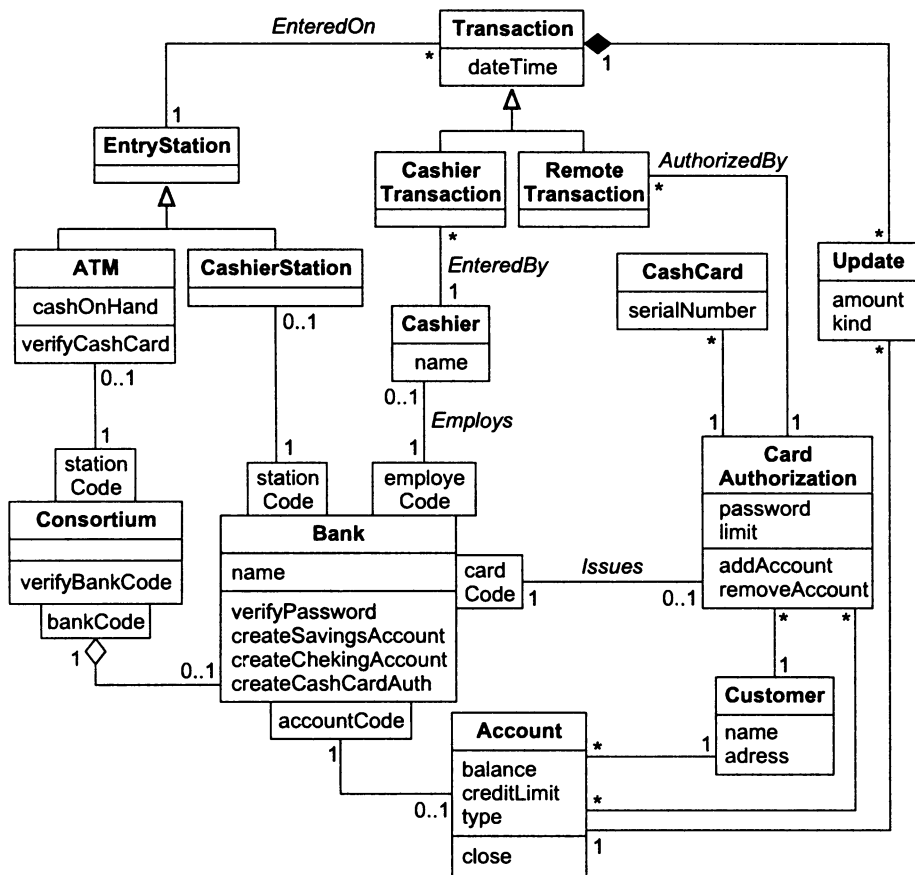


Рис. 13.12. Модель классов предметной области банкомата с некоторыми операциями

Модель взаимодействия приложения описывает взаимодействие системы с внешним миром. Сначала необходимо определить точную границу приложения. Затем нужно ввести действующие лица, описывающие внешние объекты, непосредственно взаимодействующие с системой. Кроме того, необходимо составить варианты использования для видимой извне функциональности. К каждому варианту использования нужно составить сценарии, описывающие типовые последовательности, вариации, крайние и исключительные ситуации. Сложные варианты использования можно дополнить диаграммами деятельности. Варианты использования и действующие лица можно упорядочить при помощи отношений. Нако-

нец, варианты использования следует проверить по модели классов предметной области на предмет несоответствий между ними.

Затем следует дополнить классы предметной области классами приложения. К классам приложения относятся классы интерфейса пользователя, пограничные классы и управляющие объекты. Для их выделения необходимо тщательно изучить варианты использования и сценарии.

На последнем этапе анализа приложения строится модель состояний. Эта модель обычно оказывается богаче модели состояний предметной области и описывает более разнообразное поведение. Сначала нужно выявить классы приложения, обладающие несколькими состояниями, и изучить сценарии взаимодействия на предмет наличия событий, относящихся к этим классам. Самый сложный аспект — объединить сценарии вместе, найти перекрытия и замкнутые циклы. Завершив разработку модели состояний, проверьте согласованность диаграмм состояний друг с другом, а также с моделями классов и взаимодействия.

Мы подчеркиваем необходимость абстрагирования в процессе анализа предметной области. Столь же важно абстрагирование и для анализа приложения. Конструируя свои модели, старайтесь мыслить экспансивно, с расчетом на расширение. Не подчиняйте свое приложение существующим методикам бизнеса, которые со временем могут измениться. Вместо этого старайтесь делать приложение гибким, способным воспринимать и учитывать грядущие изменения.

Таблица 13.1. Ключевые понятия главы

диаграмма деятельности	управляющий объект
действующее лицо	сценарий
анализ приложения	диаграмма последовательности
пограничный класс	операция по списку
вариант использования	граница системы
пользовательский интерфейс	

Библиографические замечания

Бертран Мейер [Meyer-97] дает много полезных сведений о принципах, на которых должен основываться хороший проект. Он рекомендует использовать проектирование снизу вверх, ориентированное на структуру данных, применять выделение «операций по списку», а также утверждает, что в системе не должно быть «главной программы». Для спецификации операций он рекомендует использовать утверждения, пред- и постусловия.

Литература

[Meyer-97] Bertrand Meyer. Object-Oriented Software Construction, Second Edition. Upper Saddle River, NJ: Prentice Hall, 1997.

Упражнения

- 13.1. (4) Подготовьте сценарии нетипичных и исключительных ситуаций из раздела 13.1.6.
- 13.2. (6) Постройте поддиаграммы *Deposit* (Депозит), *Withdrawal* (Снятие) и *Query* (Опрос) с рис. 13.10.
- 13.3. (4) На рис. 13.1 показана диаграмма классов для упражнения 11.6. *Sender* (Отправитель) и *Receiver* (Получатель) — это единственные классы, имеющие существенное поведение, зависящее от времени. Постройте диаграмму последовательности для следующего сценария: *Отправитель* пытается установить соединение с получателем, отправляя пакет, инициирующий транзакцию. *Получатель* успешно считывает пакет и отвечает подтверждением приема. После этого *отправитель* передает пакет, сообщающий о начале файла, который также подтверждается. Затем передается содержимое файла в виде трех пакетов, прием которых подтверждается *получателем*, после чего следуют пакеты, указывающие на достижение конца файла и конца транзакции, которые, как и все остальные, подтверждаются *получателем*.

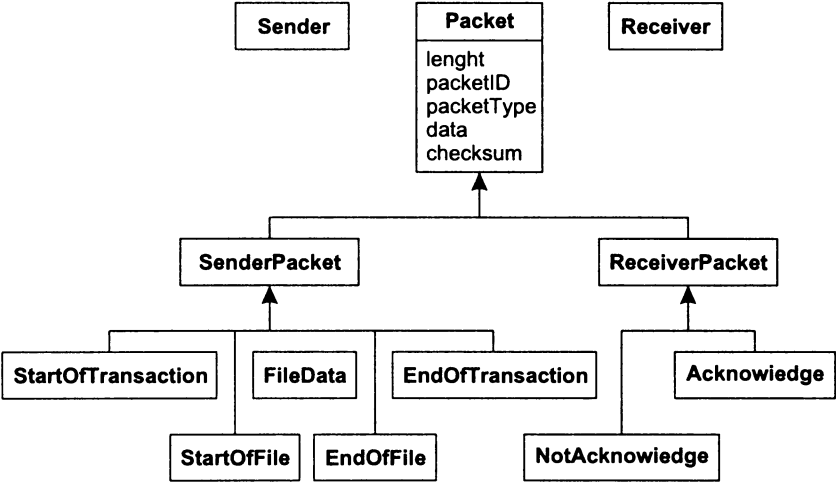


Рис. У13.1. Диаграмма классов для системы передачи файлов

- 13.4. (3) Подготовьте дополнительные диаграммы последовательности для предыдущего примера, учитывающие ошибки, вызванные шумами. Ошибки могут возникать в пакетах любого типа. После этого проверьте свой ответ в предыдущей задаче.
- 13.5. (5) Подготовьте диаграмму состояний системы передачи файлов по диаграммам последовательности, разработанным в упражнениях 13.3 и 13.4.
- 13.6. (8) Составьте диаграмму состояний счетчика пройденного пути для велосипеда, исходя из приведенных ниже сценариев.

- Пользователь включает счетчик на едущем велосипеде.
Счетчик отображает текущее время. Пользователь нажимает кнопку РЕЖИМ.
Счетчик отображает пройденное за текущий день расстояние. Пользователь нажимает кнопку РЕЖИМ.
Счетчик отображает максимальную скорость с момента сброса. Пользователь нажимает кнопку РЕЖИМ.
Счетчик отображает длительность катания с момента сброса. Пользователь нажимает кнопку РЕЖИМ.
Счетчик отображает пройденное расстояние с момента сброса. Пользователь нажимает кнопку РЕЖИМ.
Счетчик отображает среднюю скорость с момента сброса. Пользователь нажимает кнопку РЕЖИМ.
Счетчик отображает текущее время...
- Пользователь включает счетчик на неподвижном велосипеде.
Счетчик отображает текущее время. Пользователь нажимает кнопку РЕЖИМ.
Счетчик отображает суммарное пройденное расстояние. Пользователь нажимает кнопку РЕЖИМ.
Счетчик отображает суммарную продолжительность катания. Пользователь нажимает кнопку РЕЖИМ.
Счетчик отображает пройденное сегодня расстояние. Пользователь нажимает кнопку РЕЖИМ.
Счетчик отображает максимальную скорость с момента сброса. Пользователь нажимает кнопку РЕЖИМ.
Счетчик отображает длительность катания с момента сброса. Пользователь нажимает кнопку РЕЖИМ.
Счетчик отображает пройденное расстояние с момента сброса. Пользователь нажимает кнопку РЕЖИМ.
Счетчик отображает среднюю скорость с момента сброса. Пользователь нажимает кнопку РЕЖИМ.
Счетчик отображает текущее время...
- Счетчик отображает пройденное за день расстояние.
Часы показывают полночь. Начинается новый день.
Счетчик устанавливает пройденное за день расстояние равным нулю.
- Пользователь прекращает движение.
Проходит четыре минуты.
Дисплей счетчика меркнет.
Пользователь нажимает кнопку РЕЖИМ.
Дисплей загорается.

- Пользователь нажимает кнопку РЕЖИМ и не отпускает ее.

Счетчик обнуляет все переменные, вычисленные с момента сброса.

Рассмотрим простой редактор диаграмм из упражнений 12.3–12.8.

- 13.7. (2) Обозначьте границу системы для этого приложения в нескольких предложениях.
- 13.8. (2) Укажите два действующих лица, взаимодействующих с данным приложением.
- 13.9. (4) Приведите по крайней мере четыре варианта использования. Определите каждый из них одним или двумя предложениями. Постройте диаграмму вариантов использования.
- 13.10. (6) Структурируйте варианты использования, имеющие общие элементы, при помощи отношений. Вы можете создавать новые варианты использования. (Замечание для преподавателя: вы должны дать студентам ответ к предыдущему упражнению.)
- 13.11. (4) Подготовьте типичный сценарий построения рис. У12.1. Укажите по крайней мере десять операций с редактором из описания задачи, приведенного в главе 12. Об ошибочных ситуациях пока не беспокойтесь.
- 13.12. (3) Подготовьте три сценария ошибочных действий, отталкиваясь от предыдущего упражнения.
- 13.13. (4) Подготовьте диаграммы последовательностей для сценариев, написанных в предыдущем упражнении.
- Рассмотрим автоматизированную систему подсчета очков из упражнений 12.9–12.13.
- 13.14. (2) Обозначьте границу системы для этого приложения в нескольких предложениях.
- 13.15. (2) Укажите четыре действующих лица, взаимодействующих с данным приложением.
- 13.16. (5) Вот несколько вариантов использования системы: регистрация участника, планирование соревнований, планирование сезона, подсчет очков за упражнение, судейство упражнения, вычисление статистики. Определите каждый из этих вариантов одним-двумя предложениями. Постройте диаграмму вариантов использования.
- 13.17. (3) Подготовьте сценарий настройки системы подсчета очков в начале сезона. Необходимо ввести данные о командах, участниках и судьях. Подготовьте расписание соревнований в сезоне и выберите упражнения для каждого из них. Укажите коэффициенты сложности для разных упражнений. Вы должны ввести по меньшей мере 2 команды, 6 участников, 3 судьи, 3 соревнования и 12 упражнений. Об ошибочных ситуациях пока не беспокойтесь.
- 13.18. (3) Подготовьте три сценария ошибочных действий, отталкиваясь от предыдущего упражнения.

- 13.19. (3) Подготовьте сценарий распечатки и обработки форм предварительной регистрации для системы подсчета очков. В этом сценарии два спортсмена должны поменять свои адреса, а другие два — указать, что они не могут принять участие в соревнованиях. Каждому участнику должен быть присвоен номер.
- 13.20. (6) Подготовьте диаграмму деятельности для следующего расчета. На диаграмме приведите плавательные дорожки для участника соревнований, оператора компьютера, судьи и секретаря.
- Оператор компьютера произносит вслух номер участника, появляющийся на дисплее. Участник проверяет свой номер и выполняет упражнение. Трое судей поднимают таблички со своими оценками. Секретарь читает их вслух. Оператор компьютера вводит произносимые секретарем оценки в компьютер.
- 13.21. (3) Укажите «операции по списку» для системы подсчета очков и разместите их на диаграмме классов.
- 13.22. (5) Для каждого из перечисленных в предыдущем упражнении методов укажите, что именно он должен делать.

Проектирование системы

Выполнив анализ задачи, вы должны решить, каким образом следует подойти к проектированию. *Проектирование системы* (system design) — это выбор высокоуровневой стратегии решения задачи. Такая стратегия называется *архитектурой системы* (system architecture). На этом этапе принимаются решения о разбиении системы на подсистемы, об аппаратной или программной реализации этих подсистем, а также формулируются политики, служащие основой для проектирования классов.

В этой главе вы узнаете о том, какие моменты следует учитывать в процессе формулирования проекта системы. Мы опишем несколько распространенных архитектурных стилей, которые могут использоваться в качестве отправной точки. Этот список не может претендовать на полноту: всегда можно придумать какую-то новую архитектуру. Подходы, описанные в этой главе, рассчитаны на небольшие и средние программные проекты. Крупные и сложные системы, в создании которых участвует более десяти разработчиков, ограничены из-за недостатков взаимодействия между людьми, и потому они требуют гораздо более внимательного отношения к логистике. Большая часть рекомендаций, которые мы здесь приводим, применимы не только к объектно-ориентированным системам.

14.1. Обзор проектирования систем

В процессе анализа мы сосредоточиваемся на том, *что* должно быть сделано, и не задумываемся о том, *как* это следует делать. В процессе проектирования разработчики принимают решения о том, каким образом они будут решать задачу, сначала на высоком уровне, а потом в деталях.

Проектирование системы — это первый этап проектирования, в процессе которого выбирается базовый подход к решению задачи. Разработчики формулируют общую структуру и стиль решения. Архитектура системы определяет ее разбиение

на подсистемы. Кроме того, архитектура создает контекст, в котором принимают-ся последующие решения о детальном устройстве системы. Создание архитектуры требует последовательного выполнения следующих действий.

1. Оценить производительность системы (раздел 14.2).
2. Составить план повторного использования (раздел 14.3).
3. Разбить систему на подсистемы (раздел 14.4).
4. Выявить присущую задаче параллельность (раздел 14.5).
5. Распределить подсистемы по аппаратному обеспечению (раздел 14.6).
6. Спланировать хранилища данных (раздел 14.7).
7. Распределить глобальные ресурсы (раздел 14.8).
8. Выбрать стратегию управления программным обеспечением (раздел 14.9).
9. Обработать пограничные условия (раздел 14.10).
10. Установить приоритеты при компромиссах (раздел 14.11).
11. Выбрать стиль архитектуры (раздел 14.12).

Достаточно часто для выбора архитектуры системы можно использовать аналогию с предыдущими системами. Некоторые виды архитектуры применимы к широкому классу задач. В разделе 14.12 мы приводим обзор нескольких наиболее распространенных видов архитектуры вместе с задачами, для решения которых они используются. Не все задачи можно решить с помощью этих видов архитектуры. Комбинируя их между собой, вы будете получать новые виды архитектуры.

14.2. Оценка производительности

На начальном этапе планирования новой системы вы должны провести приближенную оценку ее производительности. Цель состоит не в том, чтобы получить высокую точность оценки, а в том, чтобы определить, реально ли построить нужную систему. Вы можете ошибиться в два раза в ту или иную сторону, обычно такой точности оказывается вполне достаточно. Все зависит от задачи. Расчет должен быть простым и основанным на здравом смысле. Вам придется делать упрощающие предположения. Не беспокойтесь о деталях: приближайте, оценивайте, а при необходимости — угадывайте нужные значения.

Пример с банкоматом. Представим, что мы проектируем сеть банкоматов для одного банка. Действовать можно так. У банка есть 40 филиалов (в каждом должен быть терминал). Предположим, что такое же количество терминалов расположено в супермаркетах и других магазинах. В загруженный день половина терминалов может быть занята одновременно. (Можно предположить, что все они заняты — результаты от этого изменятся не сильно. Главное — определить приемлемые границы производительности.) Предположим, что каждый клиент завершает один сеанс работы с банкоматом примерно за минуту и большая часть транзакций состоят из одного вноса или снятия денег со счета. Таким образом, пиковая нагрузка на сеть может составлять около 40 транзакций в минуту или одна транзакция в секунду. Этот результат может быть не очень точным, но он показывает,

что нам не потребуется особенно быстродействующее оборудование. Совсем другой результат получился бы, если бы мы оценивали требования к сетевой бирже или к книжному магазину. Для них вычислительная мощность компьютеров становится очень важным критерием.

Аналогичным образом можно оценить потребности в хранении данных. Подсчитайте количество клиентов, оцените объем данных для каждого из них и перемножьте полученные величины. Если речь идет о банке, то требования получатся более жесткие, чем к вычислительной мощности компьютеров, но все же их вряд ли можно будет назвать запредельными. Напротив, для спутниковой системы геофотосъемки ключевыми факторами, определяющими архитектуру, будут потребности в хранении данных и в их передаче.

14.3. Планирование повторного использования

Повторное использование часто называется основным преимуществом объектно-ориентированной технологии, но оно не обеспечивается одним только фактом применения этой технологии. Повторное использование характеризуется двумя существенно разными аспектами: можно использовать существующие вещи, а можно создавать новые с расчетом на повторное использование. Гораздо проще использовать существующее, чем проектировать новое, ориентируясь на неизвестные перспективы повторного использования. Конечно, чтобы мы сейчас могли повторно использовать какие-то сущности, кто-то в прошлом должен был их создать. Суть в том, что большинство разработчиков используют существующее, и лишь небольшая их часть создает новое. Не думайте, что вам придется начинать работу с объектно-ориентированными технологиями с создания новых сущностей. Для этого требуется большой опыт.

Повторно используемыми бывают модели, библиотеки, каркасы и образцы. Наиболее практичным можно назвать повторное использование моделей. Логика модели часто бывает применима ко множеству задач.

14.3.1. Библиотеки

Библиотека (library) — это совокупность классов, которые могут оказаться полезными во множестве контекстов. Эта совокупность должна быть хорошо устроена, чтобы пользователи могли искать нужные им классы. Чтобы построить такую библиотеку, требуется много усилий, и часто бывает трудно решить, где должен быть размещен какой-либо объект. В решении этой задачи может помочь сетевой поиск, но он не заменит хорошей организации библиотеки. Кроме того, классы должны иметь точное и подробное описание, чтобы пользователи могли сами решить, подходят они им или нет. В книге [Korson-92] отмечаются некоторые качества, которыми должны обладать «хорошие» библиотеки классов.

- **Цельность.** Библиотека классов должна охватывать небольшое количество четко определенных тем.
- **Полнота.** Библиотека классов должна полностью описывать поведение по выбранным темам.

- **Согласованность.** Полиморфные операции должны обладать одинаковыми именами и сигнатурами в разных классах.
- **Эффективность.** Библиотека должна предоставлять альтернативные реализации алгоритмов (таких как алгоритмы сортировки), позволяющие выбирать между скоростью и объемом памяти.
- **Расширяемость.** Пользователь должен иметь возможность определять подклассы классов библиотеки.
- **Универсальность.** Везде, где это возможно, должны использоваться параметризованные определения классов.

К сожалению, при интеграции библиотек из нескольких источников могут возникать проблемы [Berlin-90]. Разработчики часто размазывают конкретные решения по многим классам в иерархии наследования. Библиотеки классов могут отвечать политикам, имеющим смысл по отдельности, но несовместимым друг с другом. Такие рассогласования трудно устранить, конкретизировав какой-нибудь класс или добавив собственный код. Вместо этого приходится нарушать инкапсуляцию и изменять исходный код. Эти проблемы настолько серьезны, что именно они фактически ограничивают возможности использования кода из библиотек классов.

- **Проверка аргументов.** Приложение может проверять аргументы в совокупности или по одному во время их ввода. Групповая проверка подходит для интерфейса командной строки: пользователь вводит все аргументы, и только после этого осуществляется проверка. Активный интерфейс проверяет аргументы по одному или независимыми группами, как только пользователь их вводит. Комбинация библиотек классов, некоторые из которых проверяют аргументы группой, а другие — индивидуально, даст в конечном итоге ужасный пользовательский интерфейс.
- **Обработка ошибок.** В разных библиотеках классов используются разные методики обработки ошибок. Методы одной библиотеки могут возвращать код ошибки вызывающей программе, в то время как методы другой библиотеки могут обрабатывать ошибки самостоятельно.
- **Парадигмы управления.** Приложение может управляться событиями (*event-driven control*) или процедурами (*procedure-driven control*). В первом случае пользовательский интерфейс вызывает методы приложения. Во втором случае приложение вызывает методы пользовательского интерфейса. Объединить оба типа управления в одном приложении достаточно сложно.
- **Групповые операции.** Групповые операции часто оказываются неэффективными и неполными. Например, примитив удаления объекта может заблокировать базу данных, произвести удаление, после чего завершить транзакцию. Если вы хотите удалить группу объектов в одной транзакции, библиотека классов должна поддерживать функцию группового удаления.
- **Сборка мусора.** Библиотеки классов по-разному подходят к управлению выделением памяти и предотвращением ее утечек. Например, память для строки может выделяться с возвращением указателя на фактическую строку,

возвращением копии строки или возвращением указателя с доступом только на чтение. Стратегии сборки мусора тоже могут быть разными: *пометка и подметание* (mark and sweep), *счетчик ссылок* (reference counting), *сборка мусора приложением* (последний вариант применяется в C++).

- **Коллизии имен.** Названия классов, открытые атрибуты и методы принадлежат глобальному пространству имен, поэтому вам остается только надеяться на то, что коллизий между разными библиотеками не произойдет. Большинство библиотек классов используют префиксы, сокращающие вероятность коллизий.

14.3.2. Каркасы

Каркас (framework — [Johnson-88]) — это скелет программы, детализация которого позволяет создать полноценное приложение. Детализация чаще всего сводится к конкретизации абстрактных классов поведением, специфичным для данного приложения. С каркасом может поставляться библиотека классов, благодаря наличию которой пользователь может конкретизировать абстрактные классы, выбирая подходящие подклассы из библиотеки, а не программируя их поведение с нуля. Каркас состоит не только из классов, он включает парадигму потока управления и общие инварианты. Обычно каркасы создаются для целой категории приложений, а библиотеки классов для них еще более специализированны и не подходят для решения общих задач.

14.3.3. Образцы

Образец (pattern) — это апробированное решение общей задачи. Разные образцы предназначены для различных стадий цикла разработки программного обеспечения. Существуют образцы для анализа, архитектуры, проектирования и реализации. Лучше использовать существующие образцы, чем заново изобретать решения с нуля. Образцы обычно сопровождаются рекомендациями по их использованию, а также характеристиками для сравнения.

Использование образцов дает множество преимуществ. Одно из них состоит в том, что образец разрабатывался другими людьми и успешно применялся ими на практике. Следовательно, он с большей вероятностью окажется корректным и устойчивым, чем не прошедшее тестирование собственное решение. Кроме того, работая с образцами, вы учите язык, знакомый многим другим разработчикам. Огромное количество книг описывают образцы, тонкости и нюансы работы с ними. Вы можете рассматривать образцы как расширения языка моделирования. Не обязательно мыслить в терминах примитивов, можно оперировать их комбинациями. Образцы — это фрагменты модели, выражающие знания экспертов.

Образец отличается от каркаса. Первый обычно состоит из небольшого количества классов, связанных отношениями. Напротив, каркас обычно содержит на порядок больше классов и охватывает целую подсистему или даже приложение.

Пример с банкоматом. Концепция транзакции подразумевает некоторую возможность повторного использования, потому что транзакции часто встречаются

в компьютерных системах, и существуют коммерческие программы, обеспечивающие их поддержку. Кроме того, повторное использование возможно в рамках инфраструктуры, обеспечивающей взаимодействие консорциума с банкоматами и компьютерами банков.

14.4. Разбиение системы на подсистемы

Для всех приложений, за исключением самых простых, первый этап проектирования системы сводится к делению этой системы на части. Каждая крупная часть системы называется подсистемой. Каждая подсистема обладает чем-то таким, что отличает ее от других подсистем, например единой функциональностью, физическим размещением или выполнением на однотипном оборудовании. Например, компьютер космического корабля может состоять из подсистем жизнеобеспечения, навигации, управления двигателями и проведения космических экспериментов.

Подсистема (subsystem) — это не объект и не функция, а совокупность классов, ассоциаций, операций, событий и ограничений, связанных между собой и имеющих хорошо определенный и (желательно) не слишком обширный интерфейс с другими системами. Подсистема обычно определяется сервисами, которые она предоставляет. *Сервис* (service) — это группа родственных функций, имеющих общее назначение, например обработка ввода-вывода, построение рисунков и выполнение арифметических операций. Подсистема выражает некоторый цельный взгляд на одну часть задачи. Например, файловая система является подсистемой операционной системы. Она состоит из набора связанных абстракций, которые практически не зависят от абстракций в других подсистемах (таких как управление памятью и процессами).

Каждая подсистема связана с оставшейся частью системы хорошо определенным интерфейсом. Интерфейс определяет форму всех взаимодействий и потоки информации через границы подсистемы, но он не ограничивает внутреннюю реализацию подсистемы. Благодаря этому каждая подсистема может разрабатываться независимо от остальных.

Подсистемы следует определять таким образом, чтобы взаимодействия в системе в целом носили главным образом внутренний характер, то есть осуществлялись внутри подсистем. Это сократит зависимости между разными подсистемами. Количество подсистем не должно быть большим: в качестве разумного ограничения можно взять число 20. Каждая подсистема, в свою очередь, может быть разбита на подсистемы более низкого уровня.

Отношения между двумя подсистемами могут быть организованы по принципу клиент-сервер, или же эти подсистемы могут быть одноранговыми. В первом случае (отношение клиент-сервер) клиент вызывает сервер, который выполняет некоторый сервис и возвращает результат. Клиент должен знать интерфейс сервера, но серверу не обязательно знать интерфейсы клиента, поскольку инициатором взаимодействия всегда является клиент.

В одноранговых отношениях каждая подсистема может обратиться к любой другой. Взаимодействие подсистем не обязательно сопровождается немедленным ответом. Одноранговые взаимодействия сложнее, поскольку подсистемы должны

знать интерфейсы всех остальных подсистем. В них могут возникать циклы, трудные для понимания и проектирования. Старайтесь разбивать систему на клиентские и серверные части всегда, когда это возможно, потому что одностороннее взаимодействие гораздо проще проектировать, понимать и изменять, чем двустороннее.

Разбиение системы на подсистемы может быть организовано в виде последовательности горизонтальных уровней или вертикальных отделов.

14.4.1. Уровни

Многоуровневая система (layered system) — это упорядоченное множество виртуальных миров (ярусов), каждый из которых строится в терминах миров, находящихся ниже него, и образует базис для реализации вышестоящих уровней. Объекты каждого уровня могут быть независимыми, но часто между объектами разных уровней наблюдается некоторое соответствие. Знание в данном случае является односторонним: подсистема знает о нижних уровнях, но не о верхних. Между верхними и нижними уровнями существует отношение клиент-сервер.

Например, в интерактивной графической системе окна строятся при помощи операций с экраном, которые состоят из операций с отдельными пикселями, которые сводятся к операциям ввода-вывода с устройствами. Каждый уровень может иметь свой набор классов и операций. Каждый уровень реализуется в терминах классов и операций нижележащих уровней.

Многоуровневая архитектура бывает двух видов: открытая и закрытая. В закрытой архитектуре каждый уровень строится только в терминах того уровня, который находится непосредственно под ним. Это сокращает зависимости между уровнями и облегчает внесение изменений, поскольку интерфейс уровня влияет только на следующий за ним уровень. В открытой архитектуре уровень может использовать функции любого нижележащего уровня. Это сокращает потребность в переопределении операций на каждом уровне, благодаря чему код может получиться более эффективным и компактным. Однако открытая архитектура не способствует сокрытию информации. Изменения в подсистеме могут повлиять на любую вышестоящую подсистему, поэтому такая архитектура получается менее устойчивой, чем закрытая. Оба типа полезны. Проектировщику приходится взвешивать достоинства и недостатки каждой из них.

Обычно в постановке задачи определяются только верхний и нижний уровни. Верхний уровень — это требуемая система, а нижний — это доступные ресурсы (оборудование, операционная система, существующие библиотеки). Если разница между ними слишком велика (а так бывает достаточно часто), вам приходится вводить дополнительные уровни для сокращения концептуальных разрывов между соседними уровнями.

Многоуровневую систему можно перенести на другое оборудование, переписав только один из ее уровней. Нужно стараться вводить по крайней мере один уровень абстрагирования между приложением и любыми сервисами, предоставляемыми операционной системой или оборудованием. Добавьте уровень классов интерфейса, предоставляющих логические сервисы, и отобразите их на конкретные системно-зависимые сервисы.

14.4.2. Разделы

Разделы (partitions) делят систему по вертикали на несколько независимых или слабо связанных между собой подсистем, каждая из которых предоставляет сервисы одного типа. Например, в состав компьютерной операционной системы входят файловая система, диспетчер процессов, система управления виртуальной памятью и система управления устройствами. Подсистемы могут знать о существовании остальных подсистем, но это знание не является глубоким и не создает серьезных зависимостей на уровне проектирования.

Разница между уровнями и разделами состоит в том, что уровни отличаются друг от друга уровнем абстракции. Разделы же просто делят систему на части, находящиеся на одном и том же уровне абстракции. Еще одно отличие состоит в том, что уровни обязательно зависят друг от друга (обычно по типу клиент-сервер). Разделы образуют одноранговую систему и являются относительно независимыми или взаимозависимыми.

14.4.3. Сочетание уровней и разделов

Вы можете разбить систему на подсистемы, сочетая уровни и разделы. Уровни могут разбиваться на разделы, а разделы — на уровни. На рис. 14.1 показана схема типичного приложения, использующего имитацию и интерактивную графику. Крупные системы чаще всего требуют использования комбинации уровней и разделов.

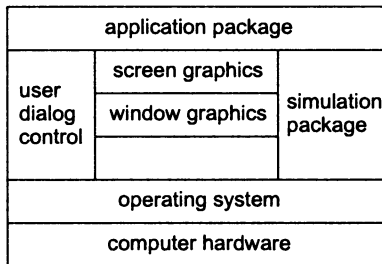


Рис. 14.1. Схема типичного приложения

После определения подсистем высшего уровня необходимо показать информационные потоки. Иногда все подсистемы взаимодействуют между собой, но чаще структура потоков оказывается более простой. Например, вычисления часто организуются в виде конвейера (примером такой структуры является компилятор). Другие системы имеют структуру звезды, в которой главная подсистема управляет всеми взаимодействиями других подсистем. Везде, где это возможно, следует стремиться упрощать топологию и сокращать количество взаимодействий между подсистемами.

Пример с банкоматом. На рис. 14.2 показана архитектура системы банкоматов. Эту систему мы разбиваем на три основных подсистемы: банкоматы, компьютер консорциума, компьютеры банков. Топология — простая звезда: компьютер

консорциума взаимодействует со всеми банкоматами и всеми банковскими компьютерами (линии связи мы обозначили *comm link*). Для обращения к банкоматам и компьютерам банков используются коды банкоматов и банков.

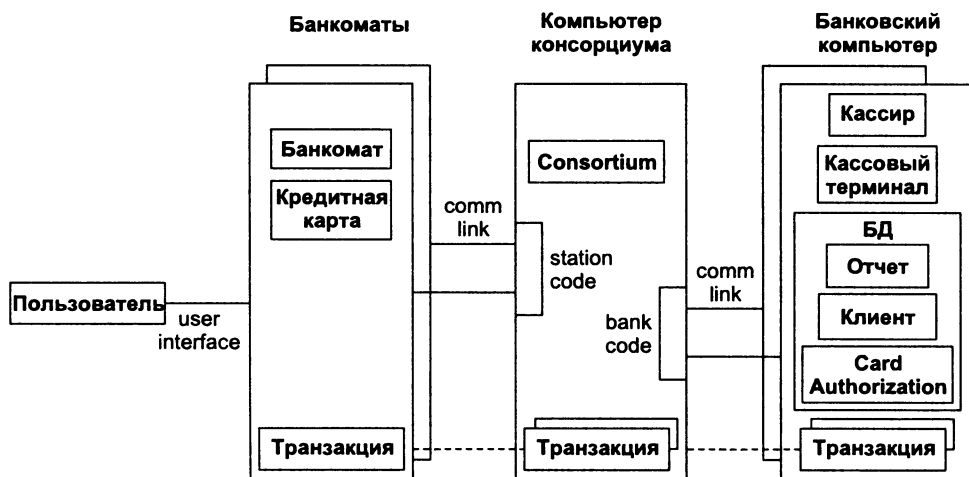


Рис. 14.2. Архитектура системы банкоматов

14.5. Выделение параллелизма

В аналитической модели, в реальном мире и в аппаратном обеспечении все объекты действуют параллельно друг с другом. Однако в программной реализации системы не все объекты оказываются параллельными, потому что один процессор может и не обеспечивать одновременное выполнение множества объектов. На практике вы можете реализовать несколько объектов на одном процессоре только в том случае, если эти объекты не могут быть активными одновременно. Одной из важных задач этапа проектирования системы является выделение объектов, которые должны быть активными одновременно, а также объектов, активность которых является взаимоисключающей. Объекты второй группы можно объединить в один поток управления (задачу).

14.5.1. Выделение неотъемлемой параллельности

Основным средством для поиска параллелизмов является модель состояний. Два объекта являются неотъемлемо параллельными, если они могут одновременно получать информацию о событиях, не взаимодействуя друг с другом. Если события не синхронизированы друг с другом, вы не можете объединить эти объекты в один поток управления. Например, подсистемы управления двигателем и крыльями самолета должны работать параллельно (хоть и не полностью независимо друг от друга). Конечно, лучше всего работать с независимыми подсистемами, потому что их можно распределить по разным аппаратным устройствам без всяких затрат на взаимодействие.

Не обязательно реализовывать неотъемлемо параллельные подсистемы в виде разных аппаратных устройств. Аппаратные прерывания, операционные системы и механизмы поддержки многозадачности предназначены именно для моделирования логической параллельности в однопроцессорной системе. Физически параллельные входные сигналы должны, разумеется, поставляться разными датчиками, но если на выходной сигнал не накладывается никаких временных ограничений, многозадачная операционная система вполне может обеспечить моделирование параллельности. Чаще всего объекты, которые должны быть реализованы в виде отдельных аппаратных устройств, перечисляются в описании задачи.

Пример с банкоматом. Если бы постановка задачи из главы 11 содержала утверждение, что каждый банкомат должен иметь возможность работать автономно в случае отказа центральной системы (возможно, с некоторыми ограничениями на объем транзакций), мы должны были бы обязательно установить на каждый банкомат собственный процессор с полной программой управления.

14.5.2. Определение параллельных задач

Все объекты с концептуальной точки зрения являются параллельными, однако на практике объекты часто зависят друг от друга. Изучая диаграммы состояний отдельных объектов и взаимодействие между ними, которое осуществляется посредством событий, вы можете объединять группы объектов в один поток управления. *Поток управления* (thread of control) — это маршрут, проходящий через несколько диаграмм состояний, на котором только один объект может быть активен в конкретный момент времени. Поток остается внутри одной диаграммы состояний до тех пор, пока объект не пошлет событие другому объекту и не перейдет в состояние ожидания другого события. Тогда поток управления переходит к получателю события и остается у него до тех пор, пока не возвратится к исходному объекту вместе с новым событием. Поток может разделиться в том случае, если объект отправит событие и продолжит свое выполнение.

Только один объект может быть активен в каждом потоке управления в конкретный момент времени. Потоки управления в компьютерных системах реализуются в виде задач.

Пример с банкоматом. Пока банк проверяет счет или обрабатывает транзакцию, банкомат находится в состоянии ожидания. Если центральный компьютер непосредственно управляет банкоматом, мы можем объединить объект банкомата с объектом банковской транзакции в одну задачу.

14.6. Распределение подсистем

Параллельные подсистемы должны быть распределены по аппаратным устройствам: универсальным процессорам или специализированным функциональным блокам. Вы должны:

- оценить потребности каждой подсистемы в вычислительных ресурсах и объеме ресурсов, необходимый для удовлетворения этих потребностей;

- выбрать аппаратную или программную реализацию подсистемы;
- распределить программные подсистемы по процессорам с учетом требований, предъявляемых к производительности, так, чтобы минимизировать взаимодействие между процессорами;
- определить связность физических модулей, реализующих подсистемы.

14.6.1. Оценка требований к аппаратным ресурсам

Решение использовать несколько процессоров или иных аппаратных модулей принимается на основании сведений о потребностях в аппаратных ресурсах, превышающих возможности одного процессора. Количество процессоров зависит от объема вычислений и скорости компьютера. Например, военный радиолокатор выдает слишком большой объем данных в короткий промежуток времени, чтобы его мог обработать один процессор, даже очень быстрый. Поэтому данные должны быть обработаны множеством параллельных компьютеров, чтобы потом на их основании можно было проанализировать степень угрозы.

Проектировщик системы должен оценить требуемую мощность процессора, определив стационарную нагрузку (произведение количества транзакций в секунду на время обработки транзакции). Приближенное значение часто оказывается неточным. Бывает полезно провести несколько экспериментов. Оценочное значение нужно несколько завысить, чтобы учесть переходные эффекты, связанные со случайными флуктуациями нагрузки и синхронизированными всплесками активности. Избыточная емкость зависит от приемлемой частоты отказов, вызванных нехваткой ресурсов. Вы должны учитывать не только стационарную (среднюю) нагрузку, но и пиковую.

Пример с банкоматом. Сам по себе банкомат относительно прост. Он должен обслуживать интерфейс пользователя и выполнять нетрудоемкие локальные вычисления. Каждому банкомату будет достаточно одного процессора. Компьютер консорциума фактически играет роль маршрутизатора: он принимает запросы банкоматов и передает их компьютерам банков. Если сеть банкоматов должна быть достаточно крупной, ее придется разбить на сегменты и установить на компьютер консорциума несколько процессоров, чтобы он не стал узким местом системы. Банковские компьютеры выполняют обработку данных и работают с простыми приложениями базы данных. Поставщики баз данных выпускают как однопроцессорные, так и многопроцессорные версии своих продуктов. Выбор зависит от необходимой пропускной способности и надежности.

14.6.2. Выбор между аппаратным и программным обеспечением

Объектно-ориентированный подход удобен для анализа аппаратных устройств. Каждое устройство — это объект, существующий параллельно с другими объектами (устройствами или программами). Вы должны распределить подсистемы по аппаратным устройствам и программным комплексам. Существует две причины,

по которым некоторая подсистема может быть реализована в виде аппаратного устройства.

- **Стоимость.** Существующее оборудование в точности предоставляет требуемую функциональность. В наше время проще купить математический сопроцессор, чем реализовывать его функции программными средствами. Датчики и эффекторы просто обязаны быть аппаратными устройствами.
- **Производительность.** Система требует более высокой производительности, чем может предоставить универсальный процессор, и при этом существует более эффективное оборудование. Например, в приложениях, связанных с обработкой сигналов, широко используются процессоры, выполняющие быстрое преобразование Фурье (БПФ).

Сложность проектирования системы связана с тем, что вы должны выполнить все внешние требования к оборудованию и программному обеспечению. Объектно-ориентированное проектирование — это не волшебная палочка, которая может решить все проблемы такого рода, однако оно помогает моделировать внешние пакеты. Вы должны учитывать вопросы совместимости, стоимости и производительности. Кроме того, нужно сделать систему гибкой в расчете на изменения в будущем (как проектные изменения, так и усовершенствования готового продукта). Гибкость требует определенных затрат, и именно архитектор должен решать, сколько он готов на нее потратить.

Пример с банкоматом. Проблем с производительностью в нашем приложении нет. Поэтому для банкоматов, консорциума и банков подойдут самые обычные компьютеры.

14.6.3. Распределение задач по процессорам

Проект системы должен описывать распределение программных подсистем по процессорам. Существует несколько причин, по которым это распределение должно быть формализовано.

- **Логистика.** Некоторые задачи должны выполняться в определенных местах, потому что они связаны с управлением оборудованием или требуют параллельного выполнения. Например, на рабочей станции инженера должна быть установлена своя собственная операционная система, чтобы оператор мог работать с ней в случае перебоев в сети.
- **Ограничения на взаимодействие.** Время отклика и поток данных превышают доступную полосу пропускания между задачей и аппаратным устройством. Например, высокоскоростные графические устройства требуют установки собственных контроллеров, потому что они порождают большие объемы данных.
- **Ограничения на вычислительные ресурсы.** Если потребности в вычислительных ресурсах слишком велики, чтобы их мог удовлетворить один процессор, можно установить несколько процессоров. Затраты на взаимодействие можно сократить, назначив задачи, активно обменивающиеся данными, одному процессору. Независимые подсистемы следует распределять по разным процессорам.

Пример с банкоматом. Банкомат не предъявляет серьезных требований к вычислительным или коммуникационным ресурсам. Трафик и вычисления, порождаемые одним пользователем банкомата, относительно невелики. Однако требования логистики в этой задаче нужно учесть. Если банкомат должен иметь возможность работать автономно в случае отказа сети, у него должен быть собственный процессор и программа. В противном случае, для беспроцессорного терминала, работающего с сетью и выполняющего все вычисления удаленно, логику банкомата можно будет упростить до предела.

14.6.4. Определение физической связности

Определив виды и количество аппаратных устройств, вы должны описать их расположение и соединения между ними.

- **Топология соединений.** Выберите топологию для соединения физических устройств. Физическим соединениям часто соответствуют ассоциации модели классов. Отношения клиент-сервер также реализуются в виде физических соединений. Некоторые соединения могут быть косвенными. Стоимость наиболее важных отношений следует минимизировать.
- **Дублируемые блоки.** Вы должны выбрать топологию дублирующих друг друга блоков. Если вы решили повысить производительность, включив в систему несколько однотипных модулей, вы должны указать и их топологию. Модель классов тут не поможет, потому что дублирование блоков обычно применяется для оптимизации на этапе проектирования. Топология дублирующих блоков обычно имеет регулярную структуру, например: линейная последовательность, матрица, дерево, звезда. Вы должны учесть ожидаемые шаблоны поступления данных и предложить параллельные алгоритмы их обработки.
- **Взаимодействия.** Выберите форму каналов коммуникации и протоколы взаимодействия. Точные интерфейсы модулей на этапе проектирования учитывать не нужно, но общие механизмы взаимодействия нужно указать. Например, взаимодействие может быть синхронным, асинхронным или блокирующим. Вы должны оценить пропускную способность и задержку каналов коммуникации и выбрать наиболее подходящие типы соединений.

Даже если соединения являются логическими, а не физическими, их все равно нужно рассмотреть. Например, блоки могут представлять собой задачи, выполняемые в рамках одной операционной системы и связанные между собой средствами межпроцессного взаимодействия (IPC). В большинстве операционных систем вызовы IPC выполняются гораздо медленнее, чем вызовы подпрограмм внутри одной задачи, поэтому их использование в задачах с жесткими временными ограничениями может быть непрактично. В этом случае вам придется объединить жестко связанные задачи в одну и реализовать соединения как вызовы подпрограмм.

Пример с банкоматом. На рис. 14.2 показаны физические соединения между подсистемами. Все банкоматы соединяются с центральным компьютером консорциума, который также соединен с компьютерами банков. Топология — звезда с компьютером консорциума в центре.

14.7. Управление хранилищами данных

Существует несколько способов хранения данных, которые могут использоваться независимо или совместно: структуры данных, файлы и базы данных. Разные варианты хранения обладают разными сочетаниями стоимости, времени доступа, емкости и надежности. Например, приложение на персональном компьютере может работать со структурами данных в памяти и с файлами. Система бухгалтерского учета может использовать базу данных для взаимодействия подсистем.

Файлы — это дешевое, простое и надежное средство хранения данных. Однако операции с файлами являются низкоуровневыми, поэтому в приложение приходится включать дополнительный код, обеспечивающий переход на необходимый уровень абстракции. В разных операционных системах файлы реализуются по-разному, поэтому переносимые приложения требуют аккуратной изоляции уровня файловой системы. Реализации файлов с последовательным доступом стандартизованы лучше всего, а вот команды и форматы хранения файлов с произвольным доступом и индексированных файлов сильно зависят от системы. Ниже перечислены типы данных, которые лучше всего хранить в файлах:

- данные большого объема и низкой информационной плотности (архивные файлы, записи журналов);
- средние по объему данные с простой структурой;
- данные, доступ к которым осуществляется последовательно;
- данные, которые можно полностью загрузить в память.

Еще один вариант хранения данных — *базы данных*, управляемые системами управления базами данных (СУБД). СУБД (в том числе реляционные и объектно-ориентированные) выпускаются разными производителями. Они могут кэшировать часто используемые данные в памяти для оптимизации стоимости и производительности оперативной памяти и дискового пространства. Базы данных облегчают перенос приложений на другое оборудование и на другие платформы, потому что перенос кода СУБД осуществляется его поставщиком. Одним из недостатков СУБД является их сложный интерфейс. Многие языки баз данных плохо интегрируются с языками программирования. Ниже перечислены типы данных, которые лучше всего хранить в базах данных:

- данные, требующие детального обновления множеством пользователей;
- данные, с которыми должны работать несколько приложений;
- данные, требующие координированного обновления посредством транзакций;
- большие объемы данных, требующие эффективной обработки;
- данные, требующие длительного хранения и имеющие большую ценность для организации;
- данные, которые должны быть защищены от неавторизованного доступа и от злоумышленников.

Объектно-ориентированные базы данных не смогли завоевать популярность на широком рынке. Поэтому их следует рассматривать только в том случае, если вы разрабатываете особое приложение, работающее с данными множества типов или требующее обращения к низкоуровневым примитивам для управления данными. К таким приложениям относятся инженерные, мультимедийные приложения, базы знаний и электронные устройства со встроенным программным обеспечением. Для большинства приложений достаточно реляционной базы данных. Такие базы доминируют на рынке, их функциональность вполне достаточна для большинства приложений. При правильном использовании они позволяют очень хорошо реализовать объектно-ориентированную модель, о чем рассказывается в главе 19.

Пример с банкоматом. Типичный банковский компьютер работает с реляционной базой данных. Такие базы обладают высокой производительностью, широко доступны и достаточно дешевы для таких видов приложений.

Банкомат тоже может работать с базой данных, но ее парадигма менее очевидна. Тут можно использовать как реляционную, так и объектно-ориентированную базу данных. Многие объектно-ориентированные базы предоставляют доступ к низкоуровневым примитивам, а урезанная по функциональности база данных может стать основой для дешевого производства программного обеспечения банкоматов. Такая база может и упростить работу банкомата. С другой стороны, реляционные базы данных достигли определенной зрелости, что сокращает затраты на разработку.

14.8. Распределение глобальных ресурсов

Проектировщик системы должен указать глобальные ресурсы и определить механизмы управления доступом к ним. Глобальные ресурсы бывают нескольких видов:

- **физические устройства:** процессоры, ленточные накопители, а также спутники связи;
- **пространство:** дисковое пространство, экран рабочей станции, кнопки мыши;
- **логические названия:** идентификаторы объектов, имена файлов, названия классов;
- **доступ к общим данным:** базы данных.

Если ресурс представляет собой физический объект, то он может самостоятельно контролировать доступ к себе при помощи протокола управления доступом. Если же ресурс представляет собой логическую сущность, то возникает опасность конфликта доступа. Например, один и тот же идентификатор объекта может быть одновременно использован параллельными задачами.

Избежать такого конфликта можно при помощи «охраняющего» объекта, которому будет принадлежать каждый глобальный ресурс и который будет управлять доступом к своему ресурсу. Один охраняющий объект может управлять несколькими общими ресурсами. Доступ к любому общему ресурсу возможен только через его охраняющий объект. Отнесение глобального ресурса к некоторому объекту — это признание индивидуальности ресурса.

Вы можете выполнить логическое разбиение ресурса, распределив его части по разным охраняющим объектам, благодаря чему будет реализован параллельный доступ к частям ресурса. Например, в параллельной распределенной среде одной из стратегий выделения идентификаторов объектов является предварительное распределение диапазонов возможных идентификаторов между процессорами, входящими в состав сети. Каждый процессор выделяет идентификаторы из предоставленного ему диапазона, благодаря чему исчезает потребность в глобальной синхронизации.

В приложениях, предъявляющих жесткие требования к временным ограничениям, стоимость доступа к ресурсу через охраняющий объект иногда оказывается слишком высока. Клиенты в таких системах должны работать с ресурсом напрямую. В этом случае блокировка может быть установлена на отдельные части ресурса. Блокировка — это логический объект, связанный с некоторым подмножеством ресурса, который дает владельцу блокировки право на непосредственный доступ к ресурсу. Охраняющий объект в этом случае тоже должен существовать, он занимается выдачей и снятием блокировки, однако после одной операции взаимодействия с охраняющим объектом для получения блокировки клиент может работать с ресурсом напрямую. Такой подход опаснее, потому что каждый пользователь ресурса сам отвечает за свое поведение при работе с этим ресурсом. Не используйте прямой доступ к общим ресурсам, если есть другие варианты.

Пример с банкоматом. В нашем случае глобальными ресурсами являются банковские коды и номера счетов. Код банка должен быть уникален в контексте консорциума. Номер счета должен быть уникален в контексте банка.

14.9. Выбор стратегии управления программным обеспечением

В аналитической модели взаимодействия представляются как события, передаваемые между объектами. Аппаратное обеспечение в этом отношении близко аналитической модели, а вот в программном обеспечении управление может быть реализовано несколькими способами. Не обязательно, чтобы все подсистемы использовали одну и ту же реализацию, но лучше всего выбирать для всей системы единый стиль управления. Существует два основных варианта управления программной системой: внешнее управление и внутреннее управление.

Внешнее управление (external control) — это поток видимых извне событий между объектами системы. Существует три варианта управления, осуществляемого через внешние события: процедурное последовательное, событийное последовательное и параллельное. Оптимальный вид управления зависит от доступных ресурсов (предоставляемых языком программирования и операционной системой), а также от видов взаимодействия внутри приложения.

Внутреннее управление (internal control) — это поток управления внутри процесса. Оно появляется только на этапе реализации, а потому нельзя сказать, что ему изначально присущи параллельность или последовательность. Проектировщик может разбить процесс на несколько задач для обеспечения логической

ясности или для повышения производительности (в случае наличия нескольких процессоров). В отличие от внешних событий внутренние переходы управления (например, вызовы процедур или вызовы между задачами) находятся под контролем программы и могут быть структурированы так, как удобно. Широко распространены три типа операций передачи управления: вызовы процедур, квазипараллельные вызовы между задачами и параллельные вызовы между задачами. Квазипараллельные задачи (сопрограммы или программные потоки) — это конструкции, разработанные для удобства программиста. Каждый программный поток обладает собственным стеком и адресным пространством, но только один из них может быть активен в конкретный момент времени.

14.9.1. Процедурное управление

В последовательной системе с процедурным управлением ход выполнения определяется кодом программы. Процедуры запрашивают внешние данные и ждут их поступления. Когда входные данные поступают в систему, выполнение программы возобновляется с той процедуры, которая запрашивала эти данные. Значение счетчика команд, содержимое стека вызовов процедур и значения локальных переменных в этом случае определяют состояние системы.

Основное преимущество процедурного управления состоит в том, что его легко реализовать в большинстве широко распространенных языков. Недостаток же в том, что этот вариант требует отображения присущей объектам параллельности на последовательный поток управления. Проектировщик должен преобразовывать события в операции между объектами. Обычно операция соответствует паре событий: событию вывода, которое выполняет вывод и запрашивает ввод, и событию ввода, которое доставляет в систему новые значения. Эта парадигма неудобна для описания асинхронного ввода, потому что программа должна явным образом запрашивать входные данные. Процедурное управление пригодно только в том случае, если модель состояний демонстрирует регулярное чередование событий ввода и вывода. Более гибкие пользовательские интерфейсы и системы управления построить на основе этой парадигмы достаточно сложно.

Обратите внимание, что основные объектно-ориентированные языки, такие как C++ и Java, являются процедурно-ориентированными. Пусть вас не вводит в заблуждение словосочетание «передача сообщений». Сообщение — это вызов процедуры со встроенным оператором *case*, зависящим от класса целевого объекта. Основной недостаток обычных объектно-ориентированных языков в том, что они не поддерживают присущую объектам параллельность. Существуют и параллельные объектно-ориентированные языки, но они пока распространены недостаточно широко.

14.9.2. Событийное управление

В последовательной системе, управляемой событиями, контроль осуществляет диспетчер или монитор, предоставляемый языком или операционной системой или выполненный в виде отдельной подсистемы. Разработчики связывают процедуры приложения с событиями, и диспетчер вызывает эти процедуры, когда

происходят соответствующие им события. Процедуры передают диспетчеру выходные данные или разрешают ввод входных данных, но они не блокируются в ожидании их поступления. Все процедуры возвращают управление диспетчеру (а не сохраняют его до тех пор, пока не будут получены какие-либо данные). Поэтому состояние системы не может быть описано счетчиком программы и стеком. Процедуры должны использовать глобальные переменные для сохранения информации о состоянии, или же локальная информация о состоянии должна поддерживаться диспетчером. Событийное управление реализовать на стандартных языках сложнее, чем процедурное, но результат во многих случаях стоит затраченных усилий.

Управляемые событиями системы получаются более гибкими, чем процедурные. Такие системы моделируют сотрудничество процессов внутри одной многопоточной задачи. Неправильно написанная процедура может заблокировать все приложение, поэтому разработчикам приходится быть внимательными. Особенно удобными получаются событийные подсистемы пользовательского интерфейса.

Используйте управляемую событиями систему для внешнего управления вместо процедурной везде, где это возможно, потому что преобразование событий в программные конструкции в этой парадигме выполнять проще. Такие системы получаются «более модульными» и лучше обрабатывают аварийные ситуации, чем процедурные системы.

14.9.3. Параллельное управление

В параллельной системе управление осуществляется несколькими независимыми объектами параллельно. Каждый из таких объектов представляет собой отдельную задачу. В этой системе события реализуются как односторонние сообщения (не такие, как «сообщения» в объектно-ориентированных языках) между объектами. Задача может ожидать ввода данных, но другие задачи при этом продолжают выполняться. Операционная система планирует выполнение задач и предоставляет механизм постановки событий в очередь, поэтому события не теряются в том случае, если задача в момент их поступления занята чем-то другим. В случае наличия нескольких процессоров разные задачи выполняются действительно параллельно.

14.9.4. Внутреннее управление

В процессе проектирования разработчик раскрывает операции на объектах в виде последовательностей операций более низкого уровня на тех же самых или иных объектах. Внутренние взаимодействия объектов во многом схожи с внешними взаимодействиями, потому что для них могут использоваться те же самые механизмы реализации. Однако существует и важное отличие: внешние взаимодействия всегда подразумевают ожидание событий, потому что объекты не зависят друг от друга и не могут заставить друг друга отвечать именно в нужный момент. Внутренние операции объектов представляют собой часть алгоритма реализации, поэтому их отклик может быть предсказуем. Следовательно, внутренние взаимодействия часто можно рассматривать в виде вызовов процедур: вызывающий объект посылает запрос и ждет на него ответа. Существуют алгоритмы параллельной

обработки, однако очень многие вычислительные задачи хорошо представляются последовательными алгоритмами и могут быть заключены в единственный поток управления.

14.9.5. Другие парадигмы

Мы предполагаем, что читателя интересует главным образом процедурное программирование, но ему не следует забывать о существовании других парадигм. Например, бывают системы, основанные на правилах, системы логического программирования и другие формы программ, не имеющих процедур. Они выражают другой стиль управления: явное управление заменяется декларацией утверждений и неявными правилами вычислений, которые могут быть недетерминированными или сложными. В настоящее время такие языки используются в узких областях, таких как искусственный интеллект и программирование баз знаний, но мы предполагаем, что со временем они начнут применяться шире. Поскольку эти языки принципиально отличаются от процедурных (в том числе и объектно-ориентированных), дальнейшее повествование не будет иметь к ним почти никакого отношения.

Пример с банкоматом. Для банкомата вполне пригодно управление событиями. Банкомат обслуживает одного пользователя в любой момент времени, поэтому параллельности не требуется. Банкомат должен быстро реагировать на действия пользователя, поэтому управление событиями подходит для него гораздо лучше, чем процедурное.

14.10. Учет граничных условий

Проект системы по большей части определяется ее поведением в стационарном режиме, однако вы обязательно должны рассмотреть пограничные условия и учесть следующие моменты.

- **Инициализация.** Система должна перейти из статического начального состояния в приемлемое стационарное состояние. Система должна проинициализировать константы, параметры, глобальные переменные, задачи, охраняющие объекты и, возможно, даже саму иерархию классов. В процессе инициализации лишь небольшая часть функциональности системы обычно бывает доступной. Особенно сложна инициализация системы с параллельными задачами, потому что на этом этапе независимые объекты не должны уйти далеко вперед или отстать от всех остальных.
- **Завершение.** Завершение обычно оказывается проще инициализации, потому что большинство объектов можно просто уничтожить. Задача должна освободить все зарезервированные ею внешние ресурсы. В параллельной системе задача должна уведомлять о своем завершении все остальные задачи.
- **Отказ.** Отказ — это незапланированное завершение системы. Отказ может быть вызван ошибками пользователя, истощением системных ресурсов или

внешней поломкой. Проектировщик системы должен учитывать возможность ее отказа. Отказ может быть вызван и ошибками внутри системы и часто обнаруживается как «невозможное» противоречие. В идеальной системе таких ошибок быть не может, однако хороший проектировщик должен предусмотреть возможность корректного завершения в случае фатальных ошибок. Система должна оставить среду выполнения как можно более «чистой» и сохранить как можно больше информации о причинах отказа, прежде чем окончательно завершить работу.

14.11. Установка приоритетов

Проектировщик системы должен установить приоритеты, которыми нужно будет руководствоваться на последующих этапах разработки. Приоритеты разрешают конфликты между желаемыми, но несовместимыми целями. Например, чаще всего работу системы можно ускорить, добавив в нее памяти, но при этом она будет потреблять больше энергии и стоить дороже. Компромиссы связаны не только с самой программой, но и с процессом ее разработки. Иногда бывает нужно пожертвовать полнотой функциональности для того, чтобы в нужный момент выпустить продукт на рынок. Иногда приоритеты указываются при постановке задачи, но чаще ответственность за совмещение несовместимого ложится на проектировщика.

Проектировщик должен определить относительную важность различных критериев, на основании чего можно будет в дальнейшем идти на компромиссы. Проектировщик не обязан принимать все решения сразу, он просто устанавливает правила, по которым эти решения будут приниматься. Например, первые видеоигры работали на процессорах с ограниченным объемом памяти. Важнейшим приоритетом была экономия памяти, за ним следовала быстрота выполнения. Проектировщикам приходилось использовать все программистские трюки, забывая о переносимости, понятности и удобстве поддержки системы. Другой пример: математические подпрограммы должны работать на множестве систем. Для них жизненно важным является правильность алгоритмов, а также переносимость и понятность. Этими качествами нельзя жертвовать в угоду быстрой разработки.

Решения проектировщика влияют на систему в целом. Успех или провал конечного продукта может зависеть от правильности выбора целей. Если не установить приоритеты в масштабе всей системы, отдельные ее части могут быть оптимизированы по разным параметрам (субоптимизация), в результате чего система будет зря расходовать ресурсы. Даже в небольших проектах программисты часто забывают о настоящих целях и начинают беспокоиться об «эффективности» тогда, когда на самом деле она не имеет принципиального значения.

Установка приоритетов для принятия решений чаще всего дает достаточно туманные результаты. Не следует ожидать точных чисел, как то: скорость 53 %, память 31 %, переносимость 15 %, стоимость 1 %. Приоритеты редко оказываются абсолютными. Если скорость выполнения важнее, чем память, это не означает, что любой прирост скорости, даже самый небольшой, может оправдать неограниченное увеличение расходуемой памяти. Нельзя даже составить полный список

критериев принятия решений. Приоритеты — это утверждение философии проекта. Реальное принятие решений на последующих этапах все равно потребует оценок и интерпретации поставленных целей.

Пример с банкоматом. Банкомат — это продукт для массового рынка. Поэтому важным фактором является стоимость производства. Конечный продукт обязан иметь удобный пользовательский интерфейс. Программное обеспечение должно быть устойчивым к отказам и ошибкам. Стоимость разработки не имеет большого значения, поскольку она может быть разделена на множество копий продукта.

14.12. Распространенные архитектурные стили

Среди существующих систем широко распространены несколько архитектурных стилей, которые могут послужить вам в качестве прототипов для построения ваших приложений. Каждый стиль оптимален для системы определенного типа. Если ваше приложение обладает набором похожих характеристик, вы можете использовать для него соответствующую архитектуру и сэкономить усилия при проектировании. В любом случае архитектурный стиль может послужить, по крайней мере, отправной точкой. Некоторые разновидности стилей перечислены ниже.

- *Пакетное преобразование* (batch transformation) — однократное преобразование всех входных данных (раздел 14.12.1).
- *Непрерывное преобразование* (continuous transformation) — преобразование изменяющегося входного сигнала, выполняемое непрерывно (раздел 14.12.2).
- *Интерактивный интерфейс* (interactive interface) — система, в которой доминируют внешние взаимодействия (раздел 14.12.3).
- *Динамическое моделирование* (dynamic simulation) — система, моделирующая развивающиеся объекты реального мира (раздел 14.12.4).
- *Система реального времени* (real-time system) — система, отвечающая жестким временным ограничениям (раздел 14.12.5).
- *Администратор транзакций* (transaction manager) — система, занимающаяся хранением и обновлением данных, часто с поддержкой параллельного доступа из разных физических точек (раздел 14.12.6).

Конечно, это не полный список известных систем и архитектур, однако мы постарались перечислить все наиболее распространенные стили. Для некоторых задач вам придется придумывать абсолютно новую архитектуру, но в большинстве случаев можно обойтись существующим стилем или какой-либо из его модификаций. Многие задачи сочетают различные аспекты нескольких архитектур.

14.12.1. Пакетное преобразование

Пакетное преобразование (batch transformation) состоит из нескольких вычислительных процессов. Приложение получает входные данные и должно вычислить результат. Никакого взаимодействия с внешним миром в промежутке не предпо-

лагается. В качестве примеров можно привести стандартные вычислительные задачи: компиляторы, программы для расчета бухгалтерских ведомостей, программы автоматического проектирования СБИС, анализа напряжений в мостах и многие другие. Для таких задач модель состояний тривиальна или вовсе вырождена. Модель классов достаточно важна: она описывает ввод, вывод и промежуточные стадии вычислений. Модель взаимодействия документирует вычисления и связывает между собой модели классов. Наиболее важным аспектом решения задачи является определение четкой последовательности этапов.

В прошлом, когда мы работали в отделе исследований General Electric, один наш коллега (Билл Премерлани) написал компилятор, который принимал ASCII-файл с графическими изображениями на входе и генерировал код определений реляционной базы данных на выходе. Эта работа предшествовала появлению коммерчески доступных средств объектно-ориентированного моделирования. На рис. 14.3 показана последовательность этапов. Компилятор состоял из пяти моделей классов: одна для входных данных, другая — для выходных и три — для промежуточных представлений.

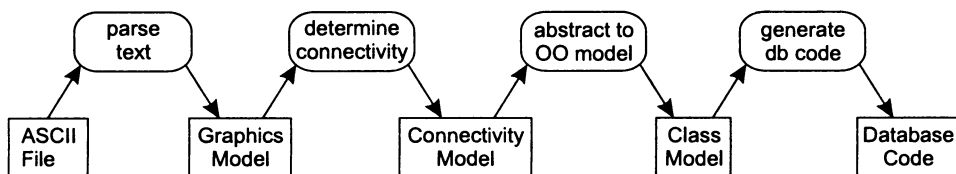


Рис. 14.3. Последовательность этапов компиляции

Пакетное преобразование проектируется в следующей последовательности.

1. Разбить преобразование на этапы, каждый из которых состоит в выполнении части преобразования.
2. Подготовить модели классов для ввода, вывода и промежуточных данных. Каждый этап учитывает только те представления, которые связывают его с соседями.
3. Детализировать каждый этап до тех пор, пока его реализация не станет очевидной.
4. Оптимизировать получившийся конвейер.

14.12.2. Непрерывное преобразование

Непрерывное преобразование (continuous transformation) — это характеристика системы, выходной сигнал которой зависит от изменяющихся сигналов на входах. В отличие от пакетного преобразования, которое вычисляет результат один раз, непрерывное преобразование должно обновлять выходные сигналы достаточно часто (теоретически — непрерывно, однако на практике вычисление осуществляется с большой частотой дискретизации). Из-за жестких временных ограничений система не может вычислять весь набор выходных данных заново каждый раз при изменении входного сигнала (иначе это было бы просто пакетное преобразование).

Вместо этого система должна рассчитывать приращения выходных данных. В качестве типичных примеров приложений этого класса можно привести: обработчики сигналов, оконные системы, инкрементные компиляторы и системы контроля производственных процессов. Для этих систем модели классов, состояний и взаимодействия должны быть примерно такими же, как и для пакетных преобразований.

Один из методов реализации систем такого рода состоит в построении функционального конвейера. Эффект любого изменения входного сигнала последовательно распространяется по конвейеру. Разработчик может добавлять промежуточные и избыточные объекты для повышения производительности конвейера. Некоторые приложения требуют синхронизации значений внутри конвейера. Такие системы выполняют операции за четко определенное время. В них поток операций должен быть аккуратно сбалансирован, чтобы значения поступали в нужное место в нужный момент, не создавая заторов.

На рис. 14.4 показан пример работы графического приложения. Приложение отображает геометрические фигуры в пользовательских координатах в систему координат окна. Затем оно обрезает фигуры по размерам окна. Наконец, каждая фигура сдвигается на экране с учетом положения оконной системы координат.

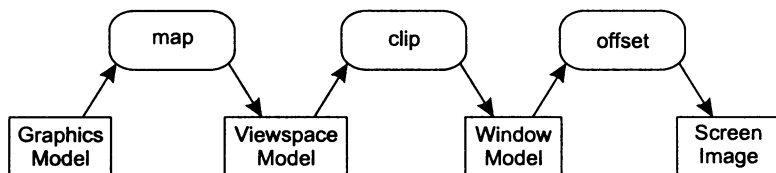


Рис. 14.4. Последовательность этапов работы графического приложения

Непрерывное преобразование проектируется в следующей последовательности.

1. Разбить преобразование на этапы, каждый из которых должен описывать какую-либо часть преобразования.
2. Определить модели входных, выходных и промежуточных данных, как для пакетного преобразования.
3. Продифференцировать каждую операцию так, чтобы работать с приращениями. Таким образом, изменение входного сигнала должно распространяться по конвейеру в виде последовательных приращений.
4. Добавить оптимизирующие промежуточные объекты.

14.12.3. Интерактивный интерфейс

Интерактивный интерфейс (interactive interface) — это система, в которой доминируют взаимодействия с внешними агентами (людьми или устройствами). Внешние агенты не зависят от системы, поэтому она не может их контролировать, а может лишь запрашивать у них ввод данных. Интерактивный интерфейс обычно составляет лишь часть некоторого приложения, которая часто может рассматриваться независимо от вычислений. В качестве примера интерактивной

системы можно привести: интерфейс запросов, основанный на формах, оконный менеджер рабочей станции или панель управления моделированием.

Наиболее важные аспекты интерактивного интерфейса — это протокол взаимодействия системы с внешними агентами, синтаксис возможных взаимодействий, представление выходных данных (варианты отображения на экране), поток управления внутри системы, производительность и обработка ошибок. Интерактивные интерфейсы определяются главным образом моделью состояний. Модель классов описывает элементы взаимодействия (входные и выходные маркеры, форматы представления). Модель взаимодействия описывает взаимодействие диаграмм состояний.

Интерактивный интерфейс проектируется в следующей последовательности.

1. Изолировать классы интерфейса от классов приложения.
2. Использовать предопределенные классы для описания взаимодействия с внешними агентами везде, где это возможно. Например, в оконных системах имеется множество предопределенных окон, меню, кнопок, форм и других классов, готовых к использованию в приложениях.
3. Используйте модель состояний в качестве структуры программы. Интерактивные интерфейсы лучше всего реализуются с использованием параллельного управления (многозадачность) или событийного управления (прерывание). Процедурное управление (выдача запроса и ожидание ответа) удобно только применительно к жестким управляющим последовательностям.
4. Изолируйте физические события от логических. Часто логическое событие может соответствовать множеству физических событий. Например, графический интерфейс может получать входные данные из формы, контекстного меню, от функциональной клавиши, введенной последовательности команд или из файла.
5. Полностью описывайте функции приложения, вызываемые интерфейсом. Убедитесь, что в модели имеется достаточно информации для их реализации.

14.12.4. Динамическое моделирование

Динамическое моделирование (dynamic simulation) — это моделирование, или отслеживание, объектов реального мира. В качестве примеров можно привести моделирование траекторий движения молекул, расчет траекторий космических кораблей, экономические модели и видеоигры. Моделирование проще всего проектировать с использованием объектно-ориентированных технологий. Объекты и операции берутся непосредственно из модели приложения. Управление может быть реализовано двумя способами: внешний управляющий объект может имитировать конечный автомат, либо же объекты могут обмениваться сообщениями между собой, как и происходит в реальном мире.

В отличие от интерактивной системы, внутренние объекты при динамическом моделировании соответствуют объектам реального мира, поэтому модель классов

в таких задачах имеет существенное значение и часто получается достаточно сложной. Модели состояний и взаимодействия также важны.

Динамическая модель проектируется в следующей последовательности.

1. Идентифицировать активные объекты реального мира в модели классов. Эти объекты обладают периодически обновляемыми атрибутами.
2. Идентифицировать дискретные события. Дискретные события соответствуют дискретным взаимодействиям объектов, таким как включение питания или нажатие на тормоз. Дискретные события могут быть реализованы в виде операций над объектами.
3. Идентифицировать непрерывные зависимости. Атрибуты реального мира могут зависеть от других атрибутов или непрерывно изменяться со временем, как, например, высота, скорость или положение рулевого колеса. Эти атрибуты должны обновляться через периодические промежутки времени с использованием методик аппроксимации для минимизации погрешностей квантования.
4. Обычно модель управляется циклом с небольшим шагом по времени. Дискретные события между объектами также часто включаются в этот цикл.

Обычно самое сложное при динамическом моделировании — обеспечить адекватную производительность. В идеальном мире произвольное количество параллельных процессоров могло бы выполнять моделирование в точном соответствии с тем, как события развиваются в реальном мире. На практике проектировщику приходится оценивать вычислительную стоимость каждого цикла и обеспечивать систему достаточными вычислительными ресурсами. Непрерывные процессы приходится аппроксимировать дискретными шагами.

14.12.5. Системы реального времени

Система реального времени (real-time system) — это интерактивная система с жесткими временными ограничениями на время отклика. Системы реального времени бывают жесткими и гибкими. Жесткие — это жизненно важные приложения, требующие гарантированного отклика в заданное время. Гибкие системы реального времени тоже должны быть высоконадежными, но для них допустимо редкое нарушение ограничений. Типичные приложения реального времени — это управление процессами, считывание данных, коммуникационные устройства, управляющие устройства и реле перегрузки.

Проектирование систем реального времени — сложная задача, требующая решения таких подзадач, как обработка прерываний, распределение по приоритетам и координация работы множества процессоров. К сожалению, системы реального времени часто используются в режимах, близких к предельно допустимым, поэтому для достижения необходимой производительности приходится изменять проект системы. В результате теряется переносимость и удобство обслуживания системы. Проектирование систем реального времени — это специальная тема, которую мы в этой книге рассматривать не будем.

14.12.6. Администратор транзакций

Администратор транзакций (transaction manager) — это система, основное назначение которой состоит в хранении и выдаче данных. Большинство администраторов транзакций имеют дело с множеством пользователей, которые одновременно считывают и записывают данные. Администратор должен обеспечивать защиту данных от неавторизованного доступа и от случайных сбоев. Администраторы транзакций часто реализуются в виде надстройки над системой управления базами данных (СУБД). СУБД предоставляет универсальную функциональность для управления данными, которая может быть использована повторно. В качестве примеров администраторов транзакций можно привести системы бронирования авиабилетов, контроля инвентаря и выполнения заказов.

В таких системах доминирует модель классов. Модель состояний бывает важной в отдельных случаях, в частности для описания эволюции объекта, а также ограничений и методов, применяющихся в разные моменты времени. Модель взаимодействия бывает существенной достаточно редко.

Администратор транзакций проектируется в следующей последовательности.

1. Преобразовать модель классов к структурам базы данных (подробнее см. главу 19).
2. Определить параллельные модули — такие, которые не могут использоваться совместно). При необходимости следует добавить новые классы.
3. Определить единицу транзакции — набор ресурсов, которые одновременно задействуются в одной транзакции. Транзакция всегда выполняется целиком или не выполняется вовсе.
4. Выполнить проектирование параллельного управления транзакциями. Большинство СУБД решают эту задачу. Чаще всего системе следует сделать несколько попыток повторить транзакцию в случае ее отмены.

14.13. Архитектура сети банкоматов

Сеть банкоматов — это гибрид интерактивного интерфейса с администратором транзакций. Системы ввода транзакций — это интерактивные интерфейсы. Их назначение состоит во взаимодействии с человеком для ввода информации, необходимой для формулировки транзакции. Для описания этих систем нужно построить модели классов и состояний. Консорциум и банки представляют собой распределенную систему администрирования транзакций. Их назначение состоит в хранении данных и их обновлении по распределенной сети при определенных условиях. Для описания этой подсистемы нужно построить ее модель классов. На рис. 14.2 показана архитектура системы банкоматов в целом.

Постоянные хранилища данных расположены в компьютерах банков. База данных гарантирует, что данные будут непротиворечивыми, и обеспечивает параллельный доступ к этим данным. Банкомат обрабатывает каждую транзакцию как пакетный преобразователь: счет блокируется до тех пор, пока транзакция не будет завершена.

Параллельность возникает из-за наличия множества банкоматов, которые могут быть активны одновременно. Один банкомат может осуществлять только одну транзакцию в какой-либо момент времени, но в каждой транзакции участвует компьютер консорциума и по крайней мере один банковский компьютер. Как показано на рис. 14.2, транзакция соединяет физические устройства. В процессе проектирования каждая часть становится отдельным классом реализации. Компьютер консорциума или банка может одновременно обрабатывать несколько транзакций. Это не создает каких-либо новых проблем, так как база данных синхронизирует одновременный доступ к любому счету.

Компьютеры консорциума и банков будут работать в режиме управления событиями. Каждый из них накапливает входные события в очереди и обрабатывает их по одному в порядке получения. Компьютер консорциума предоставляет минимально необходимую функциональность: он передает сообщения от банкоматов банковским компьютерам и от банковских компьютеров банкоматам. Этот компьютер должен быть достаточно мощным, чтобы обрабатывать все транзакции. Допустимо в редких случаях блокировать транзакции (из-за перегрузки), отправляя пользователю соответствующее уведомление.

Банковский компьютер — единственное устройство, которое должно выполнять нетривиальные процедуры, но и они главным образом сводятся к обновлению данных в базе. Единственная сложность может возникнуть из-за необходимости обработки ошибок. Банковские компьютеры должны иметь достаточную мощность для обработки транзакций в прогнозируемом «худшем варианте», и их дисковое пространство должно быть достаточным для записи всех транзакций в журнал.

Система должна поддерживать операции добавления и удаления банкоматов и банковских компьютеров. Каждый физический модуль должен предусматривать собственную защиту от отказа или отключения от сети. База данных обеспечивает защиту от потери данных. Однако следует предусмотреть возможность отказа во время транзакции: ни клиент, ни банк не должны при этом потерять деньги. Это может потребовать реализации сложного протокола подтверждения. В случае обрыва соединения банкомат должен выводить соответствующее сообщение. Банкомат должен обрабатывать и другие отказы, такие как полный расход наличных или бумаги для чеков.

В финансовых системах наивысший приоритет имеет безопасность транзакций. Если цельность транзакции попадает под малейшее сомнение, банкомат должен прервать ее и выдать пользователю соответствующее сообщение.

14.14. Резюме

После проведения анализа приложения, но перед проектированием классов проектировщик системы должен выбрать базовый подход к решению задачи. Самая высокоуровневая стратегия построения системы называется архитектурой системы.

На начальном этапе планирования новой системы необходимо оценить ее производительность. Вы должны иметь хотя бы приблизительное представление о том, что вам предстоит. Требования к производительности системы должны

быть разумными, и вы должны быть уверены, что вас не ждут неприятные сюрпризы в процессе разработки.

Затем нужно подготовить план повторного использования. Повторное использование часто называют основным преимуществом объектно-ориентированной технологии, но оно не вытекает из одного только факта применения этой технологии. Есть два важных аспекта повторного использования. Большинство разработчиков должны беспокоиться об использовании существующих моделей, библиотек, каркасов и образцов, пригодных для их приложений. Элитные разработчики могут создавать артефакты, которые будут использоваться другими людьми.

Система может быть разделена на горизонтальные слои и вертикальные разделы. Каждый слой определяет свой абстрактный мир, который может полностью отличаться от других слоев. Каждый слой является клиентом по отношению к слою или слоям ниже его и сервером по отношению к вышележащему слою или слоям. Разделы отличаются друг от друга предоставляемыми сервисами. Применение простых топологий, таких как конвейер или звезда, позволяет уменьшить сложность системы. Большинство систем представляют собой комбинации слоев и разделов.

Объекты, которым присуща неотъемлемая параллельность, выполняются одновременно. Такие объекты нельзя объединять в один поток управления. Для их реализации нужно использовать разные аппаратные устройства или, по крайней мере, разные задачи на одном процессоре. Объекты, не являющиеся параллельными, можно объединять в один поток управления и выполнять в одной задаче.

Количество процессоров и специализированных устройств должно быть таким, чтобы производительность системы отвечала предъявленным требованиям. Вы должны распределять объекты по аппаратным устройствам, чтобы использование этих устройств было сбалансированным, а выполнение объектов отвечало требованиям параллельности. Для этого нужно оценить пропускную способность вычислительной подсистемы и предусмотреть поддержку очередей при настройке оборудования. Для ресурсоемких вычислений можно использовать специализированное оборудование. Одной из целей деления аппаратной сети на разделы является сокращение трафика между физически раздельными модулями.

Хранилища данных позволяют четко выделить подсистемы в рамках архитектуры и дают определенную степень постоянства данным приложения. Хранилища могут быть реализованы как структуры данных в памяти, файлы и базы данных. К достоинствам файлов можно отнести простоту, дешевизну и постоянство, но они находятся на слишком низком уровне абстракции и требуют дополнительных усилий при программировании. Базы данных находятся на более высоком уровне абстракции, чем файлы, но они усложняют систему и требуют дополнительных вложений.

Проектировщик системы должен выделить глобальные ресурсы и определить механизмы контроля доступа к этим ресурсам. Наиболее широко применяются следующие механизмы: создание «охраняющего объекта», который делает доступ последовательным, деление глобальных ресурсов на непересекающиеся подмножества, управляемые на более низком уровне, а также блокировка.

Аппаратное управление по определению характеризуется параллельностью. Программное управление может быть процедурным, событийным и параллельным.

Управление процедурной системой осуществляется кодом программы. Значение счетчика команд, содержимое стека и значения локальных переменных определяют состояние системы. Управление событийной системой осуществляется диспетчером или монитором. Процедуры приложения сопоставлены событиям и вызываются диспетчером при осуществлении этих событий. В параллельной системе управление осуществляется сразу несколькими независимыми объектами. Параллельные и управляемые событиями реализации получают гораздо более гибкими, чем те, которые управляются процедурами.

Проект системы описывает главным образом ее поведение в стационарном режиме, однако необходимо уделять внимание и пограничным условиям (инициализация, завершение, отказы).

Важным аспектом системной архитектуры являются компромиссы между скоростью работы и объемом памяти, аппаратным и программным обеспечением, простотой и универсальностью, эффективностью и удобством обслуживания. Принимаемые решения зависят от целей, которые ставятся перед разработчиками приложения. Проектировщик должен установить приоритеты, чтобы обеспечить согласованность принимаемых решений на всех этапах разработки.

Существует набор распространенных архитектурных стилей, пригодных для большинства приложений. К ним относятся два вида функциональных преобразований: пакетное и непрерывное; три вида систем с временной зависимостью: интерактивный интерфейс, динамическое моделирование и системы реального времени; и система с базой данных — администратор транзакций. Большинство приложений представляют собой комбинацию нескольких архитектурных стилей. Обычно для каждой из основных подсистем используется какой-либо один стиль архитектуры. Конечно, существуют и другие архитектурные стили.

Таблица 14.1. Ключевые понятия главы

архитектура	уровень	поток управления
клиент-сервер	раздел	каркас
параллелизм	одноранговые отношения	проектирование системы
управление данными	сервис	
система, управляемая событиями	подсистема	

Библиографические замечания

Простые приложения не требуют значительных усилий от системных инженеров, а вот сложные системы приходится делить на составляющие части и распределять их между соответствующими специалистами. [Clements-02] описывает процесс оценки архитектуры программы. Кратко его можно описать так: группа заинтересованных лиц собирается вместе и определяет критерии, которым должна удовлетворять архитектура, и приоритеты этих критериев. Критерии могут быть оценены количественно при помощи конкретных сценариев. Затем архитектура анализируется на соответствие наиболее высокоприоритетным сценариям.

Образцы достаточно популярны в литературе, им посвящено большое количество книжек. Существуют образцы анализа [Coad-95], архитектуры [Buschmann-96] [Shaw-96], проектирования [Gamma-95] и реализации [Coplien-92]. Регулярные конференции по образцам проводятся уже много лет. Большая их часть финансируется группой Pattern Languages of Programming [PLoP].

Литература

[Berlin-90] Lucy Berlin. When objects collide: Experiences with reusing multiple class hierarchies. ECOOP/OOPSLA 1990 Proceedings, October 21-25, 1990, Ottawa, Ontario, Canada, 181–193.

[Buschmann-96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern-Oriented Software Architecture: A System of Patterns. Chichester, UK: Wiley, 1996.

[Clements-02] Paul Clements, Rick Kazman, and Mark Klein. Evaluating Software Architectures. Boston: Addison-Wesley, 2002.

[Coad-95] Peter Coad, David North, and Mark Mayfield. Object Models: Strategies, Patterns, and Applications. Upper Saddle River, NJ: Yourdon Press, 1995.

[Coplien-92] James O. Coplien. Advanced C++ Programming Styles and Idioms. Boston: Addison-Wesley, 1992.

[Gamma-95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley, 1995.

[Johnson-88] Ralph E. Johnson and Brian Foote. Designing reusable classes. Journal of Object-Oriented Programming 1, 3 (June/July 1988), 22-35.

[Korson-92] Tim Korson and John D. McGregor. Technical criteria for the specification and evaluation of object-oriented libraries. Software Engineering Journal (March 1992), 85–94.

[PLoP] jerry.cs.uiuc.edu/~plop

[Shaw-96] Mary Shaw and David Garlan. Software Architecture. Upper Saddle River, NJ: Prentice Hall, 1996.

Упражнения

14.1. (4) Для каждой из перечисленных ниже систем укажите применимые к ней архитектурные стили: пакетное преобразование, непрерывное преобразование, интерактивный интерфейс, динамическое моделирование, система реального времени, администратор транзакций. Объясните свой выбор. Если к системе подходит несколько стилей, сгруппируйте свойства системы по стилям.

1) **Электронный игрок в шахматы.** Система состоит из шахматной доски со встроенным компьютером, подсветкой и мембранными переключателями. Человек вводит свои ходы, нажимая на фигуры, стоящие на доске, что вызывает срабатывание мембранных переключателей, установленных

под всеми клетками. Компьютер указывает свои ходы, подсвечивая клетки. Человек должен двигать фигуры за компьютер. Компьютер должен делать только разрешенные ходы, не должен допускать, чтобы человек делал неразрешенные ходы, и должен стремиться выиграть.

2) **Симулятор самолета для видеоигры.** Видеоигра уже реализована и состоит из компьютера с джойстиком и клавишами и выходным интерфейсом для подключения к цветному телевизору. Вы должны написать программу, которая будет выводить на экран вид из кабины самолета. Управление самолетом осуществляется при помощи джойстика и клавиш. Изображение должно строиться на основании сведений о местности, хранящихся в памяти. Готовая программа будет продаваться на картриджах, подключаемых к видеоигре.

3) **Микроконтроллер гибкого диска.** Микроконтроллер будет осуществлять функции внутреннего управления. Вы должны написать программу для него. Контроллер служит мостом между компьютером и дисководом. Вы отвечаете за позиционирование головки чтения-записи и за чтение данных. Дискета разбита на дорожки и секторы. Дорожки — это концентрические окружности на дискете. Каждая дорожка разбита на несколько секторов. Архитектура должна поддерживать следующие операции: поиск дорожки 0, поиск заданной дорожки, чтение дорожки, чтение сектора, запись дорожки, запись сектора.

4) **Сонар.** Вы должны описать часть системы, которая занимается обнаружением подводных объектов и вычислением расстояния до них. Для этого система испускает акустический импульс и анализирует эхосигналы. Для выполнения анализа используется корреляционная методика: переданный импульс с временной задержкой умножается на принятый эхо-сигнал и интегрируется по длительности задержки. Если результат оказывается достаточно большим, это указывает на наличие объекта на соответствующем расстоянии от детектора.

14.2. (3) Обсудите реализацию управления для приложений, описанных в предыдущем упражнении.

14.3. (7) Вы — системный архитектор, занимающийся разработкой новой программы обработки сигналов. Вы должны придумать способ сохранения данных в реальном времени. В системе используются АЦП, которые дискретизируют аналоговый входной сигнал со скоростью 16 000 байт/с (128 000 бит/с) в течение 10 секунд. К сожалению, необходимые расчеты невозможно выполнить за это время, поэтому вы должны придумать методику временного хранения данных. Ранее было принято решение ограничить размер буфера величиной 64 000 байтов. В системе имеется дисковод, работающий с 77-дорожечными дискетами общей емкостью 243 000 байтов. Перемещение головки от одной дорожки к другой занимает 10 мс, а поиск начала дорожки после позиционирования головки занимает в среднем 83 мс. Перед началом получения данных дисковод устанавливается на нужную дорожку.

Вы рассматриваете два решения задачи: 1) Запись данных на дискету по мере их поступления. Почему этот метод не сработает? 2) Использование буфера в памяти. Данные помещаются в память и записываются на дискету с максимальной скоростью. Сработает ли этот метод? Опишите его подробнее. Какой буфер потребуется в действительности? Сколько дорожек будет использовано на дискете? Подготовьте несколько сценариев. Опишите возможную систему управления.

- 14.4. (6) Рассмотрим систему для автоматического сверления отверстий в прямоугольных металлических пластинах. Размер и расположение отверстий задаются интерактивно при помощи графического редактора на персональном компьютере. Когда пользователь заканчивает работу над чертежом, специальное периферийное устройство персонального компьютера пробивает отверстия в перфокарте ЧПУ. Эту карту можно использовать на различных промышленных сверлильных станках с ЧПУ, которые способны перемещать сверлильную головку и менять сверла.

Вы занимаетесь программой редактирования чертежей и подготовки перфокарт. На карты записываются последовательности команд на перемещение головки, смену сверл и сверление. Поскольку перемещение сверла занимает некоторое время, а смена его занимает еще больше времени, система должна выбирать достаточно эффективную последовательность сверления. Не обязательно стремиться к поиску абсолютного минимума времени, но, с другой стороны, не следует расходовать его неэффективно. Головка перемещается независимо в двух направлениях, поэтому время на перемещение между двумя отверстиями пропорционально большему из смещений по этим направлениям. Вы должны подготовить проект архитектуры системы. Как вы охарактеризуете стиль системы?

- 14.5. (5) Рассмотрим систему интерактивного символьного преобразования полиномов. Основная идея в том, чтобы предоставить математику возможность более точно и эффективно разрабатывать формулы. Пользователь вводит математические выражения и команды по одной строке за раз. Выражения представляют собой полиномиальные дроби, которые состоят из констант и переменных. Переменным могут присваиваться промежуточные выражения. Выполняемые операции: сложение, вычитание, умножение, деление и дифференцирование по переменной.

Разработайте архитектуру этой системы. Как вы охарактеризуете ее стиль? Каким образом можно сохранять текущее состояние системы, чтобы в дальнейшем иметь возможность продолжить работу?

- 14.6. (4) Архитектура системы из предыдущего упражнения может содержать следующие подсистемы. Вы должны организовать их в разделы и уровни.
- 1) Синтаксис строки — сканирование строки на наличие маркеров.
 - 2) Семантика строки — определение значения введенной строки.
 - 3) Обработка команд — выполнение команд пользователя, проверка ошибок.
 - 4) Конструирование выражения — построение внутреннего представления введенного выражения.

- 5) Применение операции — выполнение операции на одном или нескольких выражениях.
- 6) Сохранение работы — сохранение текущего состояния (контекста).
- 7) Загрузка работы — загрузка предварительно сохраненного состояния (контекста).
- 8) Подстановка — подстановка выражения вместо переменной в другое выражение.
- 9) Рационализация — преобразование выражения в каноническую форму.
- 10) Оценка — замена переменной в выражении на константу и упрощение выражения.

14.7. (6) Рассмотрим систему для редактирования, сохранения и печати диаграмм классов и генерации схемы реляционной базы данных. Система должна поддерживать лишь ограниченное подмножество системы обозначений моделирования классов — классы с атрибутами и бинарные ассоциации с множественностью. Система должна поддерживать функции редактирования: создание класса, создание ассоциации, вырезать, копировать, вставить. Редактор должен «понимать» семантику диаграмм классов. Например, перемещение прямоугольника класса должно выполняться одновременно с растяжением линий, обозначающих ассоциации, в которых этот класс участвует. Удаление класса должно приводить к удалению ассоциаций. Когда пользователь завершает работу с диаграммой, система должна построить соответствующую схему для реляционной базы данных. Обсудите относительные преимущества одной программы, выполняющей обе функции, перед двумя программами, одна из которых редактирует диаграммы классов, а другая генерирует схему по диаграммам классов.

14.8. (6) В предыдущем примере вы должны были учитывать физические и логические аспекты диаграмм классов. К физическим аспектам относятся положение и размеры линий, прямоугольников и надписей. К логическим аспектам относятся связность, классы, атрибуты и ассоциации. Обсудите возможность построения архитектуры с использованием следующих стратегий, а также вопросы редактирования и сохранения диаграмм состояний и генерирования схемы базы данных.

- 1) Моделируются только геометрические аспекты диаграммы классов. Логические аспекты рассматриваются как выводимые.
- 2) Моделируются и геометрические, и логические аспекты диаграммы классов.

14.9. (5) Альтернативный подход к системе, описанной в упражнении 14.7, состоит в использовании коммерческой настольной издательской системы для подготовки диаграммы классов вместо реализации собственного редактора диаграмм. Издательская система может сохранять файлы на языках разметки. Поставщик должен описывать грамматику такого языка.

Сравните два подхода. Первый состоит в построении собственного редактора, понимающего семантику диаграмм классов. Второй состоит в использовании

коммерческой настольной издательской системы для редактирования диаграмм классов. Что произойдет, если выйдет новая версия издательской системы? Можно ли допустить, что пользователь подготовит диаграмму в терминах, которые будет понимать ваш генератор схемы? Стоит ли тратить усилия на реализацию операций копирования и вставки, которые уже реализованы в коммерческих системах? Кто обеспечит поддержку пользователей, если у них возникнут проблемы? Каким образом будет поддерживаться и обслуживаться ваша система? Как скоро вам удастся завершить ее?

14.10. (6) При проектировании систем достаточно часто встает задача хранения данных в таком формате, чтобы они сохранялись в случае падения напряжения в сети или поломки оборудования. Идеальное решение должно быть надежным, дешевым, небольшим, быстрым, бесплатным в обслуживании и простым. Кроме того, оно должно быть устойчиво к нагреву, грязи и влажности. Компромиссные решения часто влияют на функциональные требования. Сравните каждое из перечисленных ниже решений с идеалом. Обратите внимание, что наш список не исчерпывает все возможности.

- 1) Не беспокоиться о потере данных. Каждый раз при включении системы полностью сбрасывать все данные.
- 2) Не отключать питание. Использовать специальные источники питания, в том числе — резервные генераторы.
- 3) Хранить критическую информацию на магнитном диске. Периодически делать полные и добавочные копии на магнитной ленте.
- 4) Использовать батареи для подачи питания в память системы в случае отключения внешнего питания. В таком варианте система может даже предоставлять ограниченную функциональность.
- 5) Использовать специальную память, такую как память на магнитных пузырьках или энергонезависимую память EEPROM.
- 6) Критические параметры вводятся пользователем при помощи переключателей. Существуют коммерческие решения, в том числе микровыключатели в упаковке, подключаемые наподобие интегральной схемы.

14.11. (7) Для каждой из перечисленных ниже систем выберите одну или несколько стратегий хранения данных из предыдущего упражнения. Объясните свой выбор и оцените объем памяти в байтах (по порядку величины).

- 1) **Карманный бухгалтерский калькулятор.** Основной источник питания — солнечная батарея. Выполняет элементарные арифметические действия.
- 2) **Электронная пишущая машинка.** Основной источник питания — аккумуляторы или переменный ток. Работает в двух режимах. В одном режиме документы вводятся построчно. Редактирование осуществляется в пределах одной строки. На ЖК-дисплее отображается 16 символов. В другом режиме можно ввести весь документ, а потом его напечатать. Машинка должна хранить рабочий документ в памяти в течение года в случае отключения питания.

- 3) **Системные часы персонального компьютера.** Основным источником питания — постоянный ток от блока питания включенного компьютера. Системные часы хранят дату и время. Они должны работать в течение пяти лет при выключенном компьютере.
- 4) **Система бронирования авиабилетов.** Основным источником питания — сеть переменного тока. Бесперебойная работа системы должна быть обеспечена любой ценой. Если систему по какой-либо причине нужно выключить, данные не должны быть утеряны.
- 5) **Блок управления и защиты двигателя от перегрева.** Устройство обеспечивает защиту двигателей различной номинальной мощности, рассчитывая температуру двигателя по измеренному току с учетом модели отвода тепла от двигателя. Если расчетная температура превышает некоторое ограничение, двигатель отключается и не может быть запущен заново до тех пор, пока он не остынет. Основным источником питания является сеть переменного тока, которая может быть отключена. Система должна обеспечивать защиту двигателя сразу после своего включения. Параметры моделирования устанавливаются на заводе, однако пользователь должен иметь возможность при необходимости изменить их уже после установки системы. Поскольку температура двигателя не измеряется непосредственно, необходимо непрерывно выполнять моделирование температуры двигателя в течение по крайней мере одного часа после отказа питания, потому что оно может быть включено до того, как двигатель остынет полностью.

14.12. (9) Проектирование формата файлов — стандартная составляющая проектирования многих систем. Удобным средством представления формата файлов являются диаграммы BNF. На рис. У14.1 показана часть диаграммы BNF языка описания классов и бинарных ассоциаций. Нетерминальные символы изображаются прямоугольниками, а терминальные — кружками или прямоугольниками со скругленными углами. На диаграмме определены все нетерминальные символы, за исключением *character* (символ). Диаграмма состоит из классов и ассоциаций. Класс обладает уникальным именем и произвольным количеством атрибутов. Ассоциация обладает необязательным именем и двумя полюсами. Полюс ассоциации характеризуется именем одного из участвующих в ней классов и данными о множественности. Текстовая информация представляется в виде строк, заключенных в кавычки. Символ — это произвольный ASCII-символ, за исключением кавычек.

- 1) Опишите диаграмму, представленную на рис. У14.2, при помощи языка с рис. У14.1.
- 2) Обсудите сходства и различия данных в устройстве хранения и данных «в движении». Например, описание из пункта 1 может быть использовано для хранения диаграммы в файле или для передачи диаграммы из одного места в другое.

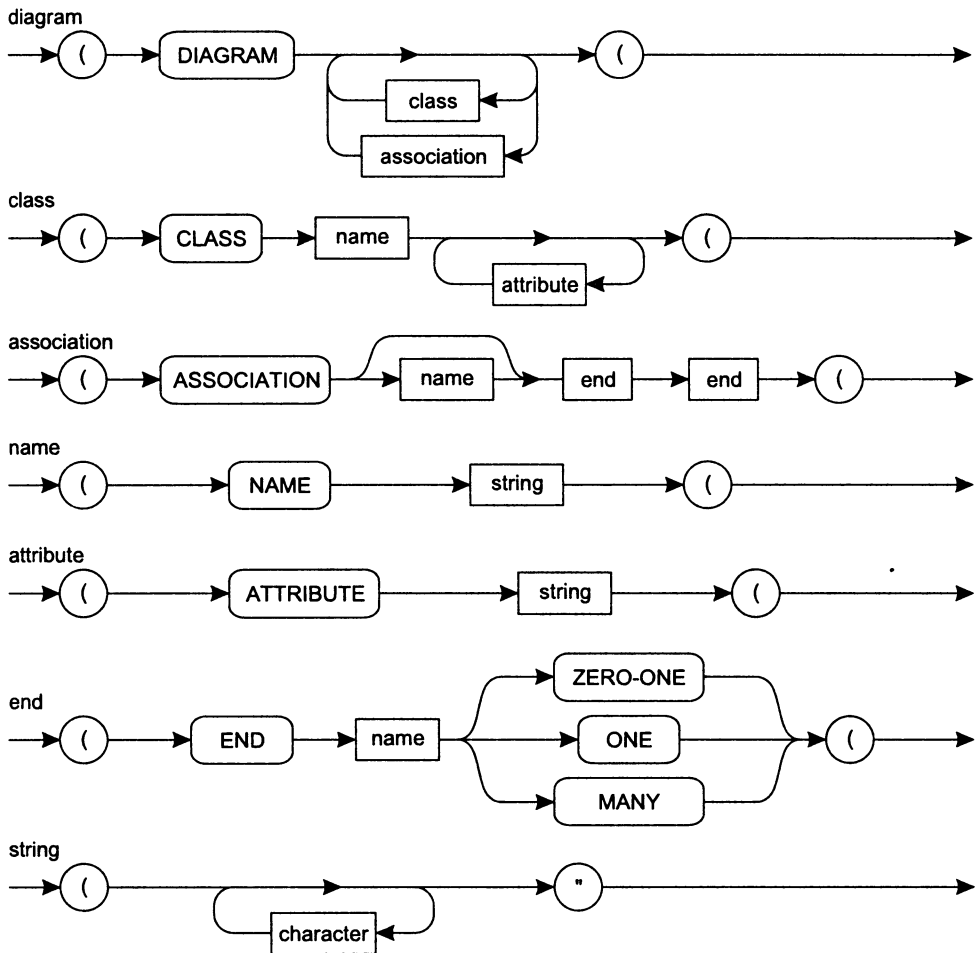


Рис. У14.1. Диаграмма BNF для языка описания классов и ассоциаций

- 3) В этой задаче язык используется для описания структуры диаграмм классов. Придумайте язык для описания двумерных многоугольников. Опишите свой язык при помощи BNF. Опишите на этом языке квадрат и треугольник.

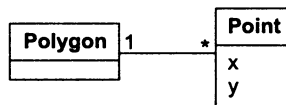


Рис. У14.2. Диаграмма классов многоугольников

- 14.13. (6) В цифровых системах часто возникают проблемы, связанные с повреждением данных из-за шумов или поломок оборудования. Одно из решений состоит в использовании циклического избыточного кода (Cyclic Redundancy

Code — CRC). При сохранении или передаче данных по ним вычисляется код, который добавляется к этим данным. При чтении или приеме данных код вычисляется заново и сравнивается с присланным значением. Соответствие является необходимым, но не достаточным требованием корректности данных. Вероятность обнаружения ошибки зависит от сложности функции вычисления CRC. Некоторые функции могут использоваться не только для обнаружения, но и для коррекции некоторых ошибок. В качестве примера простой функции, позволяющей обнаружить ошибки в одном бите, можно привести проверку четности.

Функция вычисления контрольной суммы может быть реализована аппаратно или программно. Выбор в конкретной задаче зависит от требований к скорости, стоимости, гибкости и сложности. Аппаратное решение будет быстрым, но сделает систему более сложной и дорогостоящей. Программное решение дешевле и более гибкое, но оно может быть недостаточно быстрым, кроме того, оно может усложнить программную составляющую системы.

Для каждой из перечисленных ниже подсистем определите, нужна ли вообще им контрольная сумма CRC. Если нужна, сделайте выбор между аппаратной и программной реализациями. Объясните свой выбор.

- 1) Контроллер гибкого диска.
- 2) Система передачи файлов с одного компьютера на другой по телефонным линиям.
- 3) Плата памяти в компьютере космического корабля.
- 4) Магнитный ленточный накопитель.
- 5) Проверка номера счета (контрольная сумма используется для выявления случайно введенных номеров счетов).

14.14. (6) Рассмотрим программу-планировщика из упражнений 12.16—12.19 и 12.20—12.23.

Такие программы обязательно должны обеспечивать безопасность: доступ пользователя на запись и чтение к различным расписаниям должен четко контролироваться.

Очевидный способ реализации схемы безопасности состоит в хранении списка разрешений доступа для каждой комбинации пользователя и расписания. Однако такая схема может оказаться неудобной для контроля и обслуживания.

Другое решение состоит в том, чтобы дать возможность указывать разрешения для групп пользователей. Пользователь может принадлежать к нескольким группам. Каждая группа может содержать другие группы и отдельных пользователей. Пользователи могут обращаться к расписаниям, разрешения на доступ к которым есть у них самих или у тех групп, в состав которых они входят.

Расширьте модели классов из упражнений 12.19 и 12.23 так, чтобы описать эту модель безопасности. (Замечание для преподавателя: предоставьте студентам наши ответы к упражнениям 12.19 и 12.23.)

Проектирование классов

15

На этапе анализа разработчики системы формулируют, что именно должна делать реализованная система, а на этапе проектирования они вырабатывают стратегию атаки. Цель этапа проектирования классов состоит в том, чтобы довести определения классов и ассоциаций до их конечного вида и выбрать алгоритмы для операций.

В этой главе рассказывается о том, каким образом следует облекать во плоть аналитическую модель, которая после этого становится основой для реализации. Ваши решения должны направляться стратегией, выработанной на этапе проектирования системы. На новом этапе вы должны заняться деталями этой стратегии. Нет необходимости переходить от одной модели к другой, потому что объектно-ориентированная парадигма охватывает и анализ, и проектирование, и реализацию. Эта парадигма применима и к спецификациям, описывающим реальный мир, и к компьютерным реализациям.

15.1. Обзор этапа проектирования классов

Модель анализа описывает информацию, которую должна содержать система, и высокоуровневые операции, которые она должна выполнять. Проектную модель можно было бы создать полностью с нуля, с абсолютно другими классами. Однако в большинстве случаев проще и лучше всего оказывается перенести классы из аналитической модели прямо в проектную. После этого проектирование классов сводится к добавлению деталей и принятию конкретных решений. Более того, использование аналитической модели при проектировании упрощает поддержание согласованности аналитической и проектной моделей в процессе их развития.

В процессе проектирования вам приходится выбирать различные способы реализации классов аналитической модели, с учетом необходимости минимизации времени выполнения, используемой памяти и других затрат. В частности, вы должны воплотить все операции, выбирая алгоритмы и разбивая сложные операции на более простые составляющие. Это разбиение представляет собой итерационный

процесс, повторяющийся на все более низких уровнях абстрагирования. Возможно, вам придется добавлять в модель новые классы для хранения промежуточных результатов во время выполнения программы (во избежание повторного вычисления). Однако следует опасаться чрезмерной оптимизации, так как не менее важными параметрами программы являются простота реализации и поддержки, а также возможность расширения.

Объектно-ориентированное проектирование представляет собой итерационный процесс. Когда вы сочтете модель классов готовой на одном уровне абстрагирования, вы должны будете рассмотреть ее на более низком уровне. На каждом следующем уровне вам придется добавлять новые операции, атрибуты и классы. Возможно, вам даже придется пересмотреть отношения между объектами (в том числе, изменить иерархию наследования). Не удивляйтесь, если вы обнаружите, что итерации повторяются несколько раз.

Проектирование классов делится на несколько этапов.

- 1. Наведение мостов между высокоуровневыми требованиями и низкоуровневыми сервисами (раздел 15.2).
- 2. Реализация вариантов использования через операции (раздел 15.3).
- 3. Формулирование алгоритмов для каждой операции (раздел 15.4).
- 4. Рекурсия вниз для проектирования операций, обеспечивающих реализацию более высокоуровневых операций (раздел 15.5).
- 5. Реорганизация (раздел 15.6).
- 6. Оптимизация доступа к данным (раздел 15.7).
- 7. Воплощение поведения, подлежащего манипуляции (раздел 15.8).
- 8. Коррекция структуры классов для улучшения наследования (раздел 15.9).
- 9. Организация классов и ассоциаций (раздел 15.10).

15.2. Наведение мостов

Рисунок 15.1 демонстрирует суть проектирования. Имеется набор функций или черт, которыми должна обладать ваша система. Имеется набор доступных ресурсов. Вы должны навести мосты через пропасть между ними. Потребности высокого уровня диктуются вариантами использования, командами приложения, системными операциями и службами. К ресурсам относятся инфраструктура операционной системы, библиотеки классов и предшествующие приложения.

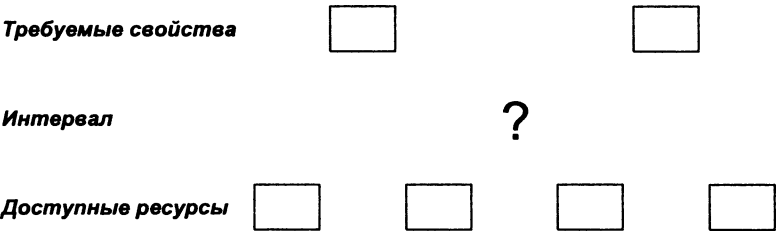


Рис. 15.1. Пропасть между ресурсами и требованиями

Если вы можете сконструировать нужные функции непосредственно из имеющихся ресурсов, задача решена. Например, комиссионер может рассчитать свои комиссионные при помощи электронной таблицы, исходя из различных предположений. В этой задаче имеющиеся ресурсы полностью соответствуют потребностям.

Однако чаще всего задача решается не так просто. Предположим, вы хотите создать интернет-каталог с возможностью заказа товаров. В этом случае вы не сможете сразу же создать систему из электронной таблицы или написать ее на языке программирования, так как пропасть между ресурсами и потребностями слишком велика. Вам придется придумывать какие-то промежуточные элементы, так чтобы каждый элемент выражался через несколько элементов более низкого уровня (рис. 15.2). Более того, если пропасть слишком велика, промежуточные элементы также придется разделить на несколько уровней. Эти элементы могут быть операциями, классами или иными конструкциями UML. Изобретение подходящих промежуточных элементов и составляет самую суть успешного проектирования.

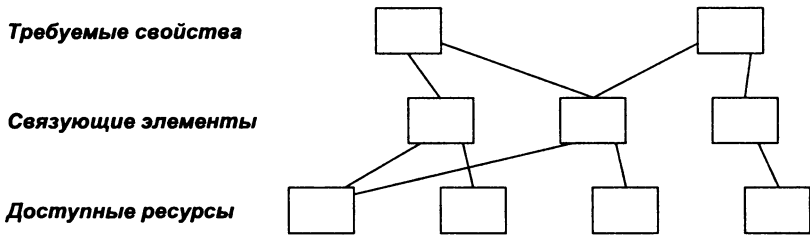


Рис. 15.2. Наведение мостов

Чаще всего промежуточные элементы не выделяются из условия задачи очевидным образом. Может существовать множество способов реализовать высокоуровневую операцию. Вам придется угадать возможный набор промежуточных операций, а затем попытаться реализовать их. Будьте аккуратны с промежуточными операциями, которые похожи друг на друга, но не являются идентичными. Вы можете сократить объем кода и сделать его более ясным, объединив похожие операции в более универсальные. Модифицированные операции могут не так хорошо подходить к некоторым высокоуровневым операциям, однако на этот компромисс приходится идти, так как хорошее проектирование подразумевает оптимизацию системы в целом, а не отдельных ее частей.

Если промежуточные элементы уже были построены раньше, вы можете просто воспользоваться ими, но принцип наведения мостов остается тем же самым. Вам все равно придется искать нужные элементы в каркасе или библиотеке классов и подгонять их друг к другу. Проблема не в том, чтобы создавать отдельные элементы — с этим справится кто угодно. Проблема в том, чтобы надежно соединить между собой эти части и получить из них цельную систему.

Проектирование особенно сложно, потому что это не аналитическая задача. Невозможно вывести идеальную систему, просто изучив системные требования. Комбинаций промежуточных элементов может быть слишком много, чтобы перебрать их все. Поэтому необходимо применять эвристические методики.

Проектирование требует синтеза: вы должны придумывать новые промежуточные элементы и соединять их друг с другом. Это творческая работа, подобная решению головоломок, доказательству теорем, игре в шахматы, строительству мостов или написанию симфоний. Никто не сделает для вас кнопку, нажав которую вы сможете получить готовый проект. Результаты предшествующих этапов процесса разработки помогут вам найти нужное направление, подобно учебникам шахматной игры, инженерным справочникам или лекциям по теории музыки, но в конечном итоге создание проекта требует совершения определенного творческого акта.

15.3. Реализация вариантов использования

В главе 13 мы добавили в модель классов основные операции. Теперь мы должны разбить эти операции, происходящие главным образом из вариантов использования, на составные части.

Варианты использования определяют требуемое поведение, но они не ограничивают его реализацию. Цель проектирования состоит в том, чтобы сделать выбор между возможными вариантами реализации и подготовить ее осуществление. Каждый вариант обладает своими преимуществами и недостатками. Недостаточно просто добиться выполнения необходимого поведения, хотя это и главная задача. Вы должны учесть влияние каждого принятого решения на производительность, надежность, легкость усовершенствования и другие параметры. Проектирование — это процесс реализации функциональности с учетом конфликтующих между собой потребностей.

Варианты использования определяют поведение на уровне системы в целом. В процессе проектирования вы должны изобретать новые операции и объекты, которые будут обеспечивать это поведение. Затем вы должны определить каждую из новых операций в терминах операций более низкого уровня, в которых задействовано большее количество объектов. В конце концов, вы сможете реализовать оставшиеся операции непосредственно в терминах существующих операций. Изобретение промежуточных операций и есть то, что мы называем «наведением мостов».

Начинать нужно с составления списка ответственности варианта использования или операции. Ответственностью называется нечто известное объекту или нечто такое, что он должен сделать [Wirfs-Brock-90]. Ответственность не является четко определенным понятием. Это просто средство, помогающее мыслительному процессу. Например, в театральной интернет-кассе процедура бронирования должна найти свободные места на выбранном спектакле, пометить их как занятые, получить платеж от покупателя, организовать доставку билетов и перевести платеж на соответствующий счет. Занятость мест должна отслеживать система самого театра, она же должна определять цену различных мест и т. д.

Каждая операция может нести ответственность за несколько действий. Некоторые виды ответственности могут быть общими у разных операций. Группируйте виды ответственности в кластеры и старайтесь делать эти кластеры согласованными. Это означает, что каждый кластер должен состоять из родственных видов ответственности, которые могут обслуживаться одной низкоуровневой операцией. В некоторых случаях, когда виды ответственности получают достаточно широко

определенными и независимыми друг от друга, каждый из этих видов становится кластером сам по себе.

После этого нужно определить операцию для каждого кластера видов ответственности. Операции следует определять так, чтобы она не ограничивалась конкретными обстоятельствами. Следует избегать и чрезмерной общности определений, потому что тогда операция получается нечеткой. Цель состоит в том, чтобы предвидеть возможное использование новой операции в будущем. Если операция может быть использована в нескольких частях текущего проекта, достаточно будет сделать ее такой общей, чтобы она покрывала все существующие варианты своего применения.

Наконец, назначать новые низкоуровневые операции классам. Если подходящих классов найти не удастся, вам придется придумать и добавить в модель новый класс низкого уровня.

Пример с банкоматом. Один из вариантов использования из главы 13 назывался *process transaction* (обработка транзакции). Вспомните, что *Transaction* (Транзакция) — это множество классов *Update* (Обновление) и что логика операции зависит от того, является ли транзакция снятием денег со счета, помещением их на счет или переводом денег между счетами.

- *Снятие денег со счета* (withdrawal). Операция снятия денег несет ответственность за множество действий, таких как: получение суммы транзакции от клиента, проверка наличия достаточных средств на счете, проверка соответствия указанной суммы политике банка, проверка наличия достаточных средств в банкомате, выдача наличных, уменьшение суммы на счете, печать клиентского чека. Некоторые из этих действий должны выполняться в контексте транзакции базы данных. Такая транзакция гарантирует, что либо все действия будут выполнены вместе, либо ни одно из них. Это относится, например, к выдаче наличных и уменьшению суммы на счете.
- *Размещение денег на счете* (deposit). Операция размещения денег на счете несет ответственность за такие действия, как: получение суммы транзакции от клиента, получение конверта с деньгами от клиента, печать временной метки на конверте, увеличение счета, печать клиентского чека. Некоторые действия также должны выполняться в контексте транзакции.
- *Трансфер* (transfer). Операция несет ответственность за такие действия, как: получение номера исходного счета, получение номера целевого счета, получение суммы трансфера, проверка наличия достаточных средств на исходном счете, проверка соответствия указанной суммы политике банка, уменьшение суммы на исходном счете, увеличение суммы на целевом счете, печать суммы на чеке. И здесь некоторые действия должны выполняться в контексте транзакции.

Видно, что некоторые виды ответственности присутствуют в нескольких операциях. Например, все операции запрашивают у клиента сумму транзакции. Операции *transfer* (трансфер) и *withdrawal* (снятие) проверяют наличие достаточных средств на счете. Правильный проект должен объединять это поведение и реализовывать его один раз.

15.4. Проектирование алгоритмов

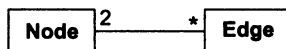
На этом этапе нужно сформулировать алгоритм для каждой операции. Аналитическая спецификация описывает, что именно должна делать операция для своих клиентов, а алгоритм показывает, как это делается. Проектирование алгоритмов также делится на несколько этапов.

1. Выбор алгоритмов, минимизирующих стоимость реализации операций.
2. Выбор структур данных, соответствующих алгоритмам.
3. Определение новых внутренних классов и операций.
4. Назначение операций подходящим классам.

15.4.1. Выбор алгоритмов

Многие операции реализуются очевидным образом, так как они сводятся к прослеживанию модели классов и получению или изменению значений атрибутов или связей. Для записи таких операций удобно использовать систему обозначений языка OCL (см. главу 3).

Некоторые операции не могут быть полностью выражены в виде цепочек прослеживания связей модели классов. Для описания таких ситуаций мы обычно используем псевдокод. Псевдокод помогает обдумывать алгоритм, не углубляясь в тонкости программирования. Например, многие приложения связаны с графами и с транзитивным замыканием (транзитивным замыканием называется набор вершин, которые непосредственно или косвенно связаны с некоторой исходной вершиной). На рис. 15.3 показана простая модель неориентированного графа, а в листинге 15.1 приведен псевдокод, позволяющий вычислить транзитивное замыкание.



```

Node::computeTransitiveClosure () returns NodeSet
  nodes:=createEmptySet;
  return self.TCloop (nodes);
Node::TCloop (nodes:NodeSet) returns NodeSet
  add self to nodes;
  for each edge in self.Edge
    for each node in edge.Node
      /* 2 nodes are associated with an edge */
      if node is not in nodes then node.TCloop(nodes);
    end if
  end for each node
end for each edge

```

Рис. 15.3. Модель неориентированного графа

Листинг 15.1. Псевдокод поиска транзитивного замыкания

```

Node::computeTransitiveClosure () returns NodeSet
  nodes:= createEmptySet;
  return self.TCloop (nodes);
Node::TCloop (nodes:NodeSet) returns NodeSet
  add self to nodes;

```



```

for each edge in self.Edge
  for each node in edge.Node
    /* каждая дуга связывает две вершины */
    if node is not in nodes then node.TCloop(nodes);
  end if
end for each node
end for each edge

```

Когда эффективность не слишком важна, нужно выбирать простые алгоритмы. На практике производительность приложения определяется небольшим количеством операций (так называемыми *узкими местами* — *bottlenecks*). В среднем на 20 % операций приходится 80 % времени выполнения. Остальные операции лучше всего сделать простыми, понятными и удобными для программирования. Творческую активность лучше направить на алгоритмы тех операций, которые могут стать узкими местами приложения. Например, поиск значения в множестве из n элементов методом перебора требует в среднем $n/2$ операций, тогда как бинарный поиск выполняется за $\log n$ операций, а поиск по хэш-таблице вообще в среднем требует менее двух операций. Ниже мы приводим некоторые рекомендации по выбору между альтернативными алгоритмами.

- **Вычислительная сложность.** Как зависит длительность вычислений от размера структур данных? Не обращайтесь внимания на незначительные факторы, влияющие на эффективность. Дополнительный уровень косвенной адресации может значительно повысить ясность алгоритма, а на скорость вычислений он почти не повлияет. Однако сложность алгоритма нужно учитывать обязательно. Время выполнения (или объем занимаемой памяти) может по-разному зависеть от количества входных данных: быть константой, линейно, квадратично или экспоненциально. Например, печально известный алгоритм сортировки «методом пузырька» выполняется за время, пропорциональное n^2 , тогда как большинство альтернативных алгоритмов сортировки выполняются за время, пропорциональное $n \log n$.
- **Простота реализации и понятность.** Для некритических операций можно пожертвовать производительностью в тех случаях, когда имеется более простой алгоритм. По этой причине нужно стараться следовать аналитической модели на этапе проектирования и вносить в нее лишь самые необходимые изменения. Если у вас нет проблем с производительностью, не тратьте время на оптимизацию, потому что это затруднит понимание модели и усложнит программирование.
- **Гибкость.** Раньше или позже большинство программ требуют расширения. Сильно оптимизированные алгоритмы часто затрудняют внесение подобных изменений. В некоторых случаях полезно иметь две версии критических операций: простой и неэффективный алгоритм, который можно реализовать быстро и использовать для проверки системы, и сложный, но эффективный алгоритм, который можно будет сравнивать с простым.

Пример с банкоматом. Взаимодействие между компьютером консорциума и банковскими компьютерами может быть достаточно сложным. Одна из проблем

связана с распределенностью вычислений: компьютер консорциума находится в одном месте, а банковские компьютеры — во множестве других мест. Важно, чтобы компьютер консорциума допускал масштабирование: сеть банкоматов не может оправдать затраты на слишком мощный компьютер консорциума, но этот компьютер должен быть способен обслуживать новые банки по мере их присоединения к консорциуму. Третий вопрос связан с тем, что банковские системы представляют собой отдельные приложения, существующие независимо от системы банкоматов. Это приводит к неизбежным преобразованиям и компромиссам, связанным с необходимостью поддерживать различные форматы данных. По этим причинам выбор алгоритмов, координирующих взаимодействие консорциума и банка, является достаточно важным решением.

Во многих банках имеются сложные системы предотвращения потерь. Поэтому решение о принятии или отклонении запроса от банкомата о снятии денег со счета может определяться не простой формулой, а сложной внутренней логикой. Это решение может зависеть от оценки кредитоспособности клиента, его поведения в прошлом, а также от соотношения баланса на счете и запрашиваемой суммы. Хорошие алгоритмы сокращают потери банков, что сопровождается возрастанием стоимости расходов на разработку.

15.4.2. Выбор структур данных

Алгоритмы работают с определенными структурами данных. На этапе анализа вас интересовала логическая структура информации в системе, но на этапе проектирования вы должны предложить структуры данных, которые позволят реализовать эффективные алгоритмы. Структуры данных не добавляют в аналитическую модель новой информации, они служат тому, чтобы организовать эту информацию в форме, удобной для применения алгоритмов. Многие структуры данных являются экземплярами классов-контейнеров. К таким структурам относятся массивы, списки, очереди, стеки, множества, мультимножества, словари, деревья и их разновидности, такие как очереди с приоритетом и бинарные деревья. Большинство объектно-ориентированных языков содержат универсальные структуры данных в библиотеках предопределенных классов.

Пример с банкоматом. *Transaction* (Транзакция) состоит из множества *Update* (Обновление). Мы должны пересмотреть модель классов — на самом деле, транзакция представляет собой последовательность обновлений, то есть класс *Transaction* должен содержать упорядоченный список *Update*. Обдумывая алгоритмы и изучая логику приложения, вы будете находить недостатки и совершенствовать модель классов.

15.4.3. Определение внутренних классов и операций

В процессе декомпозиции высокоуровневых операций вам придется придумывать новые операции, относящиеся к более низким уровням. Некоторые низкоуровневые операции могут относиться к «операциям по списку» (см. главу 13)

аналитической модели. Однако чаще всего добавления новых операций избежать не удастся.

Раскрытие алгоритмов может вызвать необходимость создания новых классов для хранения промежуточных результатов. Обычно эти классы отсутствуют в постановке задачи, так как они относятся к артефактам приложения.

Пример с банкоматом. Среди действий, относящихся к сфере ответственности варианта использования *process transaction* (обработка транзакции), мы выделили печать клиентского чека. Банкомат должен печатать на чеке все действия клиентов, чтобы они не забывали о них. В аналитической модели класс *Receipt* (Чек) отсутствовал, поэтому нам придется добавить его.

15.4.4. Назначение операций классам

Когда класс имеет отношение к реальному миру, его операции обычно достаточно очевидны. В процессе проектирования добавляются внутренние классы, соответствующие не объектам реального мира, а лишь некоторым их аспектам. Поскольку эти классы придумываются вами, они являются в достаточной степени произвольными, и границы между ними определяются не логической необходимостью, а удобством.

Как выбрать класс, к которому должна быть отнесена операция? Если в операции участвует только один объект, решение будет простым: нужно запросить выполнение операции у этого объекта или приказать ему выполнить ее. Если же объектов несколько, решение усложняется. Вы должны определить объект, играющий в этой операции главную роль. Задайте себе следующие вопросы.

- **Получатель действия.** Есть ли такой объект, над которым выполняют операцию все остальные объекты? В большинстве случаев лучше связывать операцию с ее целью, а не с инициаторами.
- **Запрос или обновление.** Есть ли такой объект, который изменяется в ходе выполнения операции, тогда как другие объекты лишь сообщают сведения о своем состоянии? Изменяющийся объект и является целью данной операции.
- **Центральный класс.** Какой класс находится ближе всего к центру графа всех участвующих в операции классов и ассоциаций? Если классы и ассоциации образуют звезду вокруг одного центрального класса, этот класс является целью операции.
- **Аналогия с реальным миром.** Если бы объекты существовали не в программе, а в реальном мире, какой объект вы бы толкали, перемещали или активировали, чтобы выполнить операцию?

Иногда бывает трудно назначить операцию какому-либо классу, оставаясь в рамках иерархии обобщения. Операции часто приходится перемещать вверх и вниз по этой иерархии в процессе проектирования, по мере уточнения их границ. Более того, определения самих подклассов иерархии тоже часто требуют корректировки.

Пример с банкоматом. Рассмотрим операции варианта использования *process transaction* (обработка транзакции) из раздела 15.3 и попробуем назначить их различным классам.

- *Customer.getAmount()* — получение суммы транзакции от клиента. Значение *amount* будет храниться как атрибут объектов *Update* (Обновление), но мы предполагаем, что эти объекты недоступны в момент ввода суммы транзакции и создаются лишь после проведения ряда проверок. Мы будем хранить значение суммы транзакции во временном атрибуте.
- *Account.verifyAmount(amount)* — проверка наличия запрошенной суммы на балансе.
- *Bank.verifyAmount(amount)* — проверка соответствия суммы политике банка.
- *ATM.verifyAmount(amount)* — проверка наличия достаточного количества купюр в банкомате. Обратите внимание, что в нашей модели имеется несколько методов *verifyAmount*, принадлежащих разным классам. Это удобный способ организации модели. Все методы должны иметь одинаковую сигнатуру.
- *ATM.disburseFunds(amount)* — выдача наличных.
- *Account.debit(amount)* — уменьшение суммы на банковском счете.
- *Receipt.postTransaction()* — печать данных о транзакции на чеке клиента. Вам может показаться, что нам следовало связать классы *Customer* (Клиент) и *Receipt* (Чек), но модель получится более точной, если вместо этого мы свяжем классы *CashCard* (БанковскаяКарта) и *Receipt* (Чек). При прослеживании модели одна банковская карта подразумевает одного клиента, но благодаря связи *CashCard* и *Receipt* мы можем проследить, какие экземпляры *CardAuthorization* (КартаАвторизации) и *CashCard* (БанковскаяКарта) использовались во время конкретного сеанса.
- *ATM.receiveFunds(amount)* — получение конверта с деньгами от клиента. Не совсем очевидно, к какому классу следует отнести этот метод. Мы могли бы назначить его классам *ATM* или *Customer*. Мы решили назначить его классу *ATM*, чтобы обеспечить симметрию с операцией *ATM.disburseFunds*. Печать штампа на конверте мы включили в операцию получения конверта.
- *Account.credit(amount)* — увеличение суммы на банковском счете.
- *Customer.getAccount()* — эта операция подразумевает получение как исходного, так и целевого счетов. Метод должен удовлетворять неявному ограничению: одному клиенту может принадлежать несколько счетов, и он обязан указывать только те счета, которые принадлежат лично ему. Обычно пользовательский интерфейс обеспечивает выполнение этого ограничения, потому что пользователю предоставляются для выбора только его собственные счета.

На рис. 15.4 показана модель классов банкомата из главы 13 с добавленными в этом разделе операциями.

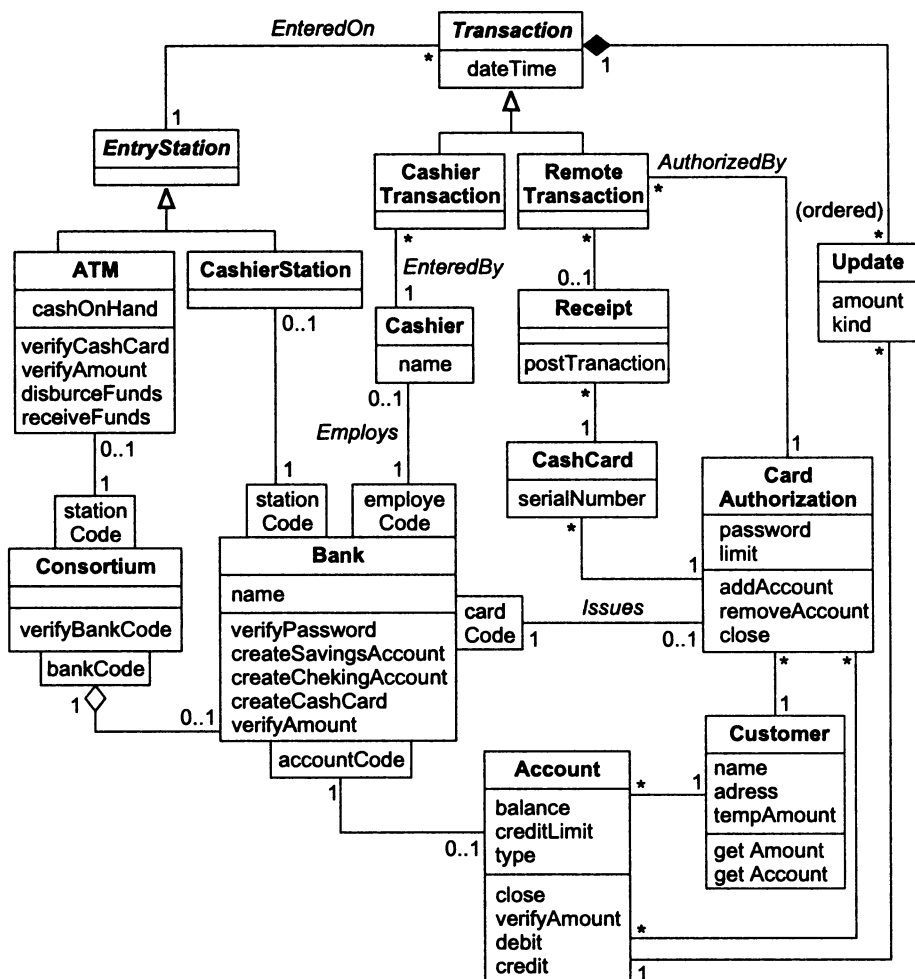


Рис. 15.4. Модель классов банкомата

15.5. Рекурсия вниз

Мы рекомендуем применять многоуровневое упорядочение операций: операции из верхних уровней при этом вызывают операции из нижних уровней. Процесс проектирования обычно осуществляется сверху вниз — вы начинаете с самых высокоуровневых операций, а затем определяете низкоуровневые операции. Можно работать и снизу вверх, но при этом вы рискуете спроектировать операции, которые впоследствии не потребуются. Рекурсия вниз осуществляется двумя основными способами: по функциональности и по механизму.

15.5.1. Уровни функциональности

Рекурсия по функциональности означает, что вы берете требуемую высокоуровневую функциональность и разбиваете ее на операции более низкого уровня. Это естественный способ работы, но при произвольном разбиении операции на части, плохо согласованные с классами, вы можете столкнуться с проблемами. Чтобы избежать этого, объединяйте схожие операции и назначайте операции классам.

Еще одна опасность рекурсии по функциональности заключается в том, что сама рекурсия может сильно зависеть от точной формулировки высокоуровневой функциональности. Небольшое изменение может радикально повлиять на разбиение. Чтобы предотвратить это, нужно прикреплять операции к классам и расширять их функциональность в расчете на последующее повторное использование. Операция должна быть цельной, внутренне согласованной и осмысленной, а не просто произвольной частью кода. Аккуратно распределенные по классам операции подвержены изменениям в гораздо меньшей степени, чем свободно витающие в модели. Такой подход к функциональности основан на том, что виды ответственности системы неизбежно реализуются в ее составляющих.

Пример с банкоматом. В разделе 15.3 мы рассмотрели вариант использования и разбили его на виды ответственности. В разделе 15.4.4 мы распределили получившиеся операции по классам. Эти операции нас вполне устроили, но если бы они плохо согласовывались с моделью классов, нам пришлось бы переработать их.

15.5.2. Уровни механизмов

Рекурсия по механизму означает, что система строится из уровней, необходимых для обеспечения функционирования механизмов реализации. Для предоставления функциональности требуются различные механизмы хранения информации, управления последовательностью операций, координации объектов, передачи информации, выполнения вычислений и других действий. Эти механизмы не присутствуют в высокоуровневых операциях явным образом, но без них система работать не может. Например, чтобы построить высокое здание, нужна инфраструктура поддерживающих балок, вспомогательных трубопроводов, устройств контроля качества. Все это не нужно самим пользователям, которые хотят просто иметь жилье, но это необходимо для того, чтобы сделать возможной реализацию выбранной архитектуры. Аналогичным образом компьютерная архитектура включает целый ряд универсальных механизмов, таких как структуры данных, алгоритмы и образцы. Они могут быть связаны не с конкретной областью приложения, а с выбранным архитектурным стилем.

Например, образец «субъект-представление» связывает несколько представлений с каждым субъектом. Субъект содержит семантическую информацию о некой сущности, а представление передает эту информацию пользователю в некотором формате. Существуют механизмы обновления субъектов и передачи изменений во все представления, а также обновления представления и передачи изменений в субъект. Эта инфраструктура может обслуживать самые разные приложения.

Однако как часть программы эта инфраструктура строится из других, более примитивных механизмов.

В любой крупной системе уровни функциональности и механизмов неизбежно смешиваются. Система, спроектированная исключительно методом рекурсии по функциональности, получается хрупкой и сверхчувствительной к изменениям в требованиях. Система, созданная путем комбинирования различных механизмов, делает очень мало полезного. Проектирование — это подбор оптимального соотношения составляющих.

Пример с банкоматом. Мы уже упомянули некоторые важные механизмы. Системе требуется коммуникационная и распределительная инфраструктура. Компьютеры банков и банкоматы находятся в разных местах и при этом должны быстро и эффективно взаимодействовать между собой. Более того, архитектура должна быть устойчива к ошибкам и сбоям на линии.

15.6. Реорганизация

Исходный проект набора операций неизбежно будет несогласованным, избыточным и неэффективным. Это естественно, потому что большой проект невозможно разработать правильно с первого раза. Вам приходится принимать решения, которые связаны с другими решениями. Неважно, в каком порядке вы будете это делать, — некоторые решения окажутся не вполне оптимальными.

Более того, эволюция проекта одновременно приводит к его деградации. Хорошо, когда операцию или класс можно использовать в нескольких целях. Однако созданная с одной целью операция не может идеально подходить для другой цели. Вы должны пересмотреть свой проект и переработать классы и операции таким образом, чтобы они удовлетворяли всем поставленным целям и были цельными с концептуальной точки зрения. В противном случае приложение получится хрупким, запутанным, неудобным для расширения и обслуживания и в конце концов рухнет под собственным весом.

Марти Фаулер [Fowler-99] определяет *реорганизацию* (refactoring) как изменение внутренней структуры программного обеспечения для усовершенствования проекта без изменения внешней функциональности. Фактически, вы должны отступить на шаг, рассмотреть все множество классов и операций и реорганизовать их так, чтобы облегчить дальнейшую разработку. Реорганизация может показаться вам пустой тратой времени, но она является важной частью любого хорошего инженерного процесса. Недостаточно просто обеспечить нужную функциональность. Если вы планируете в дальнейшем обслуживать собственную систему, вы должны сделать ее проект ясным, понятным и разбитым на отдельные модули. Реорганизация позволяет сохранить жизнеспособность проекта в процессе разработки.

Пример с банкоматом. В разделе 15.4.4 мы рассматривали операции варианта использования *process transaction* (обработка транзакции). Очевидное действие на этапе реорганизации — объединение *Account.credit(amount)* и *Account.debit(amount)* в одну операцию *Account.post(amount)* (изменение суммы на счете). Дополнительные возможности реорганизации откроются после формулирования операций для других вариантов использования.

15.7. Оптимизация проекта

Чтобы спроектировать систему, нужно сначала правильно определить логику ее работы, а затем выполнить оптимизацию. Дело в том, что оптимизировать проект непосредственно в процессе разработки достаточно сложно. Более того, если начать заботиться об эффективности слишком рано, проект получится неудачным. После того как логика будет готова, вы сможете запустить приложение, измерить его производительность, а затем заняться тонкой настройкой. Чаще всего большую часть времени выполняется небольшой участок кода, который и нужно оптимизировать. Лучше сосредоточить усилия на небольших составляющих приложения, чем распределять их равномерно.

Сказанное не означает, что нужно полностью забывать об эффективности на ранних этапах проектирования. Неэффективность некоторых подходов может быть настолько очевидна, что вы даже не будете их рассматривать. Если существует ясный, простой и эффективный способ разработать что-либо — воспользуйтесь им. Но не используйте сложных и неестественных подходов только из-за беспокойства о возможных проблемах с производительностью. Сначала нужно сделать так, чтобы простой проект заработал. Потом его можно будет оптимизировать. Тогда может обнаружиться, что проблема была совсем не в том месте, о котором вы беспокоились.

Проектная модель строится на основе аналитической модели. Аналитическая модель описывает логику приложения, а проектная модель добавляет к ней детали, относящиеся к вопросам разработки. Вы можете оптимизировать неэффективную, но при этом семантически корректную аналитическую модель для повышения производительности, но оптимизированная система будет менее понятной и будет иметь меньше шансов на повторное использование. Вы должны выдерживать баланс между эффективностью и ясностью. Оптимизация проекта включает в себя следующие задачи:

- предоставление эффективных путей доступа;
- реорганизация вычислений для повышения производительности;
- сохранение промежуточных результатов для исключения необходимости повторного вычисления.

15.7.1. Добавление избыточных ассоциаций для повышения эффективности доступа

Наличие избыточных ассоциаций в модели на этапе анализа нежелательно, поскольку они не несут в себе никакой добавочной информации. Однако цели проектирования отличаются от целей анализа: главной задачей становится обеспечение реализации приложения. Можно ли реорганизовать ассоциации для оптимизации критических участков системы? Нужно ли добавить новые ассоциации? Ассоциации аналитической модели могут образовывать далеко не самую эффективную сеть с учетом статистики обращений и их относительных частот.

Рассмотрим проект базы данных о профессиональных качествах сотрудников компании. На рис. 15.5 показана часть аналитической модели классов. Операция

Company.findSkill() возвращает множество сотрудников компании, обладающих заданным качеством. Например, приложению может потребоваться список всех сотрудников, умеющих говорить на японском языке.

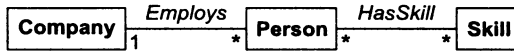


Рис. 15.5. Аналитическая модель базы данных о сотрудниках

Предположим, что в компании работает 1000 сотрудников, каждый из которых в среднем обладает десятью качествами. Простой вложенный цикл должен будет 1000 раз проследить ассоциацию *Employs* (ДержитНаСлужбе) и 10 000 раз проследить ассоциацию *HasSkill* (ОбладаетКачеством). Если на японском языке говорят только 5 сотрудников, отношение общего числа проверок к числу успешных будет 2000.

В эту модель можно внести несколько усовершенствований. Например, для реализации *HasSkill* можно использовать хэш, а не простой неупорядоченный список. Операция может выполнять хэширование за постоянное время, поэтому стоимость проверки наличия данного качества тоже будет постоянной (при условии, что существует только один объект *Skill* для качества *speaks Japanese* (говорит по-японски)). Такая схема сокращает количество проверок с 10 000 до 1000 (по одному на каждого сотрудника).

В случае когда количество успешных результатов, возвращаемых запросом, не будет большим, потому что лишь малая доля объектов удовлетворяет условию проверки, доступ к объектам можно ускорить при помощи индексирования (index). Например, на рис. 15.6 добавлена выводимая ассоциация *SpeaksLanguage* (Говорит-НаЯзыке) между классами *Company* (Компания) и *Person* (Сотрудник), в которой квалификатором является язык. Выводимая ассоциация не добавляет в модель новой информации, но она помогает быстро обращаться к сотрудникам, разговаривающим на конкретном языке. Индексирование требует определенных затрат памяти. Его приходится выполнять каждый раз при обновлении базовых ассоциаций. Необходимость построения индекса должен определять именно проектировщик. Учтите, что если в большинстве случаев в ответах на запросы присутствует значительная доля объектов, индекс не дает существенной экономии времени поиска.

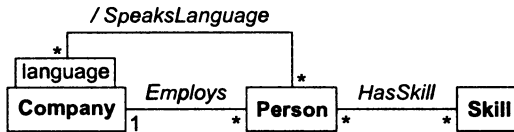


Рис. 15.6. Усовершенствованная модель с выводимой ассоциацией

Вы должны исследовать каждую операцию и выяснить, какие ассоциации отслеживает эта операция для получения необходимых данных. Затем для каждой операции нужно оценить следующие параметры.

- **Частота обращения.** Как часто вызывается эта операция?
- **Веерность.** Какова степень веерности маршрута в модели? Оцените среднее значение элементов в каждой из ассоциаций с кратностью «много».

Перемножьте эти величины, чтобы получить веерность всего маршрута. Связи с кратностью «один» не увеличивают веерности, но они несколько повышают стоимость отдельной операции; о таких небольших эффектах беспокоиться не следует.

- **Избирательность.** Какова доля «попаданий» в конечном классе? Какая доля объектов удовлетворяет критерию поиска? Если в результате прослеживания отбрасывается большая часть объектов, простой вложенный цикл может быть недостаточно эффективным решением.

Для часто вызываемых операций с малой долей возвращаемых результатов нужно обеспечивать индексирование, потому что реализация таких операций с использованием вложенных циклов будет неэффективной.

Пример с банкоматом. В комментарии к операции *postTransaction()* в разделе 15.4.4 мы решили связать *Receipt* (Чек) и *CashCard* (БанковскаяКарта). Однако нам может потребоваться быстро найти клиента по его чеку. Поскольку прослеживание от *CashCard* (БанковскаяКарта) к *CardAuthorization* (КартаАвторизации) не имеет веерности, поиск будет быстрым, а, следовательно, добавлять выводимую ассоциацию необходимости нет.

В США банки должны докладывать об операциях с суммами более 10 000 наличными правительству. Мы могли бы выполнить прослеживание от *Bank* (Банк) к *Account* (Счет), затем от *Account* к *Update* (Обновление) и отфильтровать операции с наличными, сумма которых превышает 10 000 долларов. Однако тут нам потребовалось бы уточнить модель классов, чтобы отличать операции с наличными от безналичных. Мы могли бы расширить определение *kind*, чтобы отличать наличные операции от безналичных. Ускорить эту операцию помогла бы выводимая ассоциация между *Bank* (Банк) и *Update* (Обновление).

15.7.2. Изменение порядка выполнения для повышения эффективности

После корректирования структуры модели классов для сокращения частого прослеживания связей можно приступить к оптимизации алгоритмов. Одним из принципов оптимизации алгоритмов является раннее исключение не дающих результата путей из поиска. Предположим, что вам нужно найти всех сотрудников, умеющих разговаривать на японском и французском языках. Предположим, что на японском разговаривают 5 сотрудников, а на французском — 100. Тогда лучше сначала найти тех, кто разговаривает на японском, а потом проверить, говорит ли кто-нибудь из них на французском. Лучше сокращать область поиска как можно раньше. Иногда приходится изменять порядок вложенности циклов на противоположный относительно исходной спецификации.

Пример с банкоматом. Законодательство США требует от банков сообщать не только об отдельных операциях с суммами более 10 000 долларов наличными, но и о «подозрительных» операциях, которые могут быть попытками обойти ограничение. Например, две операции снятия 5000 долларов, сделанные одна сразу после другой, могут считаться подозрительными.

Предположим, что мы не только проверяем операции с большими суммами, но и по-разному подходим к физическим и юридическим лицам. Допустим, что банк меньше доверяет физическим лицам, и установил для них более низкий порог «подозрительности» операций. Все подозрительные экземпляры класса *Update* (Обновление) можно получить через выводимую ассоциацию (мы должны написать «подозрительная» вместо «более 10 000 долларов» для выводимой ассоциации из раздела 15.7.1), а затем выполнить прослеживание обратно к классу *Account* (Счет), чтобы разделить счета физических и юридических лиц. Затем можно изучить полученные операции и определить, о каких нужно доложить куда следует.

Альтернативное решение состоит в том, чтобы добавить в модель две различные выводимые ассоциации между классами *Bank* (Банк) и *Update* (Обновление): одна для физических лиц, а другая — для юридических. Тогда обратное прослеживание к классу *Account* (Счет) не потребуется.

15.7.3. Сохранение промежуточных результатов

Иногда бывает полезно определить новые классы для кэширования выводимых атрибутов во избежание повторных вычислений. Кэш нужно обновлять в случае изменения любого из объектов, от которых он зависит. Существует три метода обработки таких изменений.

- **Явное обновление.** Проектировщик добавляет в операцию обновления исходного атрибута код, явным образом обновляющий значения выводимых атрибутов, связанных с данным атрибутом.
- **Периодическое вычисление.** В реальных приложениях значения часто изменяются группами. Вы можете реализовать периодическое вычисление всех выводимых атрибутов вместо обновления отдельных атрибутов при каждом изменении значений исходных атрибутов. Периодическое вычисление проще реализовать, чем явное обновление, а потому его реализация меньше подвержена ошибкам. С другой стороны, если данные изменяются последовательно в небольшом количестве объектов, полное периодическое вычисление может быть просто неэффективным.
- **Активные значения.** *Активным* (active value) называется значение, которое автоматически поддерживается в согласованном состоянии со значениями исходных атрибутов. Специальный механизм регистрации контролирует зависимости выводимых атрибутов от исходных атрибутов. Этот механизм отслеживает значения исходных атрибутов и обновляет значения выводимых атрибутов при любых изменениях первых. Подобные механизмы для работы с активными значениями предоставляются некоторыми языками программирования.

Пример с банкоматом. Мы можем добавить в модель класс *SuspiciousUpdateGroup* (ГруппаПодозрительныхОбновлений). Экземпляр этого класса может содержать несколько экземпляров *Update* (Обновление), а экземпляр *Update* может принадлежать нескольким *SuspiciousUpdateGroup*. В новом классе будут храниться выводимые атрибуты, упрощающие контроль подозрительного поведения.

Там же удобно хранить замечания и наблюдения. Экземпляр *SuspiciousUpdateGroup* будет множеством экземпляров *Update*, которые были заподозрены в попытке обойти ограничение в 10 000 долларов.

15.8. Воплощение поведения

Защищенное в коде поведение неизбежно утрачивает гибкость. Его можно выполнять, но нельзя изменять во время выполнения программы. В большинстве случаев этого достаточно, потому что чаще всего ваши потребности будут сводиться именно к выполнению поведения. Однако если вам нужно сохранять, передавать или изменять поведение во время выполнения программы, необходимо прибегнуть к методу *воплощения* поведения (reification).

Воплощение — это превращение в объект сущности, изначально объектом не являвшейся. Поведение обычно соответствует этому описанию. Обычно поведение не относится к тому, чем вы можете манипулировать. Если вы воплотите поведение, вы сможете сохранять его, передавать другим операциям и модифицировать. Воплощение усложняет систему, но позволяет сделать ее значительно более гибкой.

Чтобы воплотить поведение, нужно закодировать его в объект, а во время выполнения — декодировать. Правда, это может создать значительную дополнительную нагрузку на систему (а может и не создать). Если кодирование вызывает процедуры высшего уровня, все расходы сведутся к нескольким лишним ссылкам. Если же все поведение кодируется на другом языке, его приходится интерпретировать, что на порядок медленнее выполнения компилированного кода. Если время выполнения закодированного поведения составляет незначительную часть общего времени выполнения системы, накладные расходы на кодирование можно считать несущественными.

Воплощение рассматривается в упражнении 4.16 (глава 4). Там говорится о том, что рецепт какого-либо блюда можно рассматривать как последовательность операций или как данные в модели классов.

В работе [Gamma-95] перечисляются несколько образцов (patterns), позволяющих воплотить поведение. К ним относится кодирование конечного автомата в виде таблицы с интерпретатором времени выполнения (образец *State*), кодирование последовательности запросов в виде параметризованных объектов команд (образец *Command*), параметризация процедуры в терминах используемой ею операции (образец *Strategy*). Эти методы были придуманы уже давно, но язык образцов удобен для их описания и для оценки их преимуществ и недостатков. Например, образец *Strategy* использовался во времена языка FORTRAN для таких целей, как передача функции подпрограмме интегрирования. В этом языке не было способов гарантировать точное соответствие переданной функции определенным требованиям, поэтому программист легко мог допустить ошибку. Кодируя передаваемую функцию в виде экземпляра класса функций, вы получаете одновременно возможность расширения, обеспечиваемую полиморфизмом, и в то же время можете задавать сигнатуры целого семейства функций. В этом примере объектно-ориентированная технология позволяет сделать решение более ясным по сравнению с более ранними технологиями.

Пример с банкоматом. Мы уже использовали воплощение в нашем примере. Транзакцию можно рассматривать как действие: снятие денег, взнос или трансфер. Мы сделали транзакцию классом, чтобы иметь возможность описывать ее. Функциональность, которая нам нужна, легко может быть получена прослеживанием модели классов.

15.9. Корректировка иерархии наследования

Осуществляя проектирование классов, вы можете корректировать определения классов и операций в целях улучшения согласованности иерархии наследования. Для этого есть несколько методов:

- реорганизация классов и операций;
- абстрагирование общего поведения из групп классов;
- использование делегирования для совместного использования поведения в тех случаях, когда наследование неприемлемо по семантическим соображениям.

15.9.1. Реорганизация классов и операций

Иногда несколько классов определяют одинаковое поведение, так что легко обеспечить наследование этого поведения от общего предка, но чаще всего операции в разных классах являются лишь похожими, а не идентичными. Корректируя определения этих операций, вы можете свести схему к наследованию от одной операции.

Перед тем как прибегать к наследованию, нужно обеспечить соответствие операций. Все операции должны иметь одинаковую сигнатуру, то есть одинаковое количество и тип аргументов и результатов. Кроме того, операции должны иметь одинаковую семантику. Чтобы сделать наследование более удобным, вы можете использовать следующие приемы.

- **Операции с необязательными аргументами.** Можно добиться согласованности сигнатур, добавив к операции необязательные аргументы. Например, операция *draw* для черно-белого дисплея не требует указания цвета, но она может принимать этот параметр и игнорировать его для обеспечения согласованности с цветными дисплеями.
- **Операции, являющиеся частными случаями.** Некоторые операции могут иметь меньше аргументов, потому что они являются частными случаями более общих операций. Реализуйте такие операции, вызывая более общие операции с подходящими значениями параметров. Например, добавление элемента в конец списка — частный случай операции вставки элемента в список с точкой вставки, находящейся после конечного элемента списка.
- **Несогласованные названия.** Схожие атрибуты в разных классах могут иметь разные названия. Дайте атрибутам одинаковые названия и переместите их в общий класс-предок. Это обеспечит лучшую согласованность операций, работающих с этими атрибутами. Ищите похожие операции

с разными названиями. Очень важно иметь единую стратегию именования всех элементов модели.

- **Несогласованные операции.** Если операция определена в отдельных классах некоторой группы, нужно определить операцию у общего предка группы, а для тех классов, в которых она не используется, объявить ее как заглушку. Например, операция поворота имеет смысл для всех геометрических фигур, но для окружности она тождественна отсутствию операции.

Пример с банкоматом. Банкомат может печатать удаленные транзакции на чеке. Логично было бы добавить возможность печатать чеки и для транзакций, осуществляемых с помощью кассира. Однако печать чека для класса *RemoteTransaction* (УдаленнаяТранзакция) требует наличия *CashCard* (БанковскойКарты), тогда как чек для *CashierTransaction* (КассоваяТранзакция) печатается непосредственно для данного экземпляра *Customer* (Клиент). Кроме того, программное обеспечение кассы никак не связано с системой банкоматов. Поэтому у нас должно быть два вида чеков: *RemoteReceipt* (ЧекУдаленныхОпераций) и *CashierReceipt* (ЧекКассовыхОпераций).

15.9.2. Абстрагирование общего поведения

На этапе анализа вы вряд ли сможете выявить все возможности использования наследования, поэтому на этапе проектирования стоит снова уделить время проверке модели классов с целью поиска схожих элементов. Кроме того, в процессе проектирования модель обязательно расширяется новыми классами и операциями. Если в каких-либо классах имеются одинаковые операции и атрибуты, может статься, что эти классы на самом деле являются конкретизацией чего-то более общего, что может быть добавлено в модель на более высоком уровне абстрагирования.

Везде, где имеется общее поведение, вы можете создавать общие суперклассы для тех черт поведения, которые являются одинаковыми у всех потомков, оставляя им реализацию всех прочих черт. Такая трансформация модели классов называется абстрагированием общего суперкласса или общего поведения из модели. Мы рекомендуем вам создавать только абстрактные суперклассы, не имеющие непосредственных экземпляров. Поведение, определяемое таким суперклассом, принадлежит всем экземплярам его подклассов. (Для этого в модель классов всегда можно добавить подкласс *Other* — Прочие.) Например, операция *draw* для геометрической фигуры на дисплее требует настройки и прорисовки геометрии. Прорисовка для разных фигур, таких как окружности, прямые и сплайны, осуществляется по-разному, но для всех фигур можно унаследовать общую настройку (установку цвета, ширины линии и прочих параметров) от абстрактного класса *Figure* (Фигура).

Иногда оказывается полезно создать суперкласс даже в том случае, если в вашем приложении у него будет только один подкласс. Хотя никакой немедленной пользы это действие не принесет, абстрактный суперкласс может быть повторно использован в будущих проектах. Он даже может оказаться достойным включения в вашу собственную библиотеку классов. Когда вы закончите проект, вам стоит посмотреть, какие классы могут повторно использоваться в будущих приложениях.

Абстрактные суперклассы обладают и другими достоинствами, помимо общего и повторного использования. Деление класса на два, позволяющее отделить более конкретные аспекты от более общих, является одним из проявлений модульного принципа организации. Каждый класс представляет собой отдельный компонент системы с хорошо документированным интерфейсом.

Создание абстрактных суперклассов повышает возможность расширения программного продукта. Представьте, что вы разрабатываете модуль контроля температуры для большой автоматизированной системы управления. Вы должны использовать конкретный датчик (модель J55) с конкретным интерфейсом считывания температуры и преобразовывать численные данные в градусы Цельсия по специальной формуле. Все это поведение можно поместить в отдельный класс, каждый экземпляр которого будет соответствовать отдельному датчику в составе системы. Однако если вы примете во внимание, что J55 — не единственная модель датчика, то вы создадите абстрактный суперкласс *Sensor* (Датчик), определяющий поведение, общее для всех датчиков. Конкретный подкласс *SensorJ55* будет осуществлять считывание и преобразование для датчика данной модели.

Когда вам потребуется перевести систему управления на новую модель датчика, достаточно будет всего лишь подготовить подкласс со специализированным поведением, учитывающим особенности новой модели датчика. Общее для любых датчиков поведение уже имеется в суперклассе. Но самое главное, что вам не придется изменить ни единой строчки кода во всей системе управления, использующей эти датчики, потому что интерфейс, определенный в суперклассе *Sensor*, останется тем же самым.

Абстрактные суперклассы облегчают управление конфигурациями в процессе поставки и поддержки программного обеспечения. Представьте, что вы должны поставить свою систему управления на множество заводов, разбросанных по всей стране, и на каждом конфигурация системы будет своей, причем, в частности, будут использоваться разные типы датчиков. Некоторые заводы все еще работают со старой моделью J55, другие уже перешли на новую K99. Третьи работают с датчиками обоих типов. Настройка системы в соответствии с требованиями множества клиентов может потребовать больших усилий.

Вместо этого вы можете поставить всем клиентам одну и ту же версию системы, в которой для каждой модели датчика будет предусмотрен свой подкласс. Когда система запускается, она считывает клиентский файл конфигурации, в котором написано, какие модели датчиков он использует, после чего система создаст экземпляры конкретных подклассов соответствующих моделей. В остальном система работает со всеми датчиками одинаково, так как их интерфейс определяется суперклассом *Sensor*. Вы можете даже реализовать переключение с одного типа датчика на другой в процессе работы (без выключения системы), если добавите возможность создания нового объекта для нового типа датчика.

Пример с банкоматом. Мы уделили особое внимание наследованию в процессе конструирования модели классов. На данном этапе новых возможностей расширения иерархии наследования не появилось. В реальном приложении этап проектирования внесет в систему гораздо больше деталей, а потому и возможностей улучшения наследования будет больше.

15.9.3. Делегирование

Наследование — это механизм реализации обобщения, согласно которому поведение суперкласса используется совместно всеми его подклассами. Совместное использование поведения допустимо только тогда, когда классы действительно находятся в отношении обобщения, то есть когда действительно можно сказать, что подкласс является частным случаем суперкласса. Операции подкласса, перекрывающие соответствующие операции суперкласса, должны предоставлять те же сервисы, что и в суперклассе. Когда класс *B* наследует спецификацию класса *A*, вы можете предполагать, что каждый экземпляр класса *B* является экземпляром класса *A*, потому что эти экземпляры ведут себя соответствующим образом.

Некоторые программисты используют наследование как технологию реализации, вовсе не задумываясь о необходимости поддерживать согласованность поведения. Часто бывает так, что существующий класс реализует часть поведения, которое вы хотите добавить в новый класс, а в остальном эти классы отличаются друг от друга. У программиста возникает соблазн унаследовать поведение от существующего класса, чтобы избежать необходимости реализовывать новый класс полностью. В результате могут возникнуть проблемы, связанные с тем, что остальные унаследованные операции реализуют нежелательное поведение. Мы не рекомендуем *наследовать поведение*, потому что это приводит к ошибкам.

Представьте, что вам нужен класс *Stack* (Стек), а у вас уже есть класс *List* (Список). У вас может возникнуть желание унаследовать *Stack* от *List*. Помещение элемента на стек можно реализовать как добавление элемента к концу списка, а удаление элемента из стека можно реализовать как удаление элемента из конца списка. Но вы унаследуете и операции, добавляющие и удаляющие элементы из произвольных позиций списка. Если они когда-либо будут использованы (по ошибке или для упрощения задачи), класс *Stack* начнет вести себя не так, как ему полагается.

Вместо того чтобы использовать наследование как технологию реализации, вы можете достичь той же цели, связав два класса ассоциацией. После этого один объект сможет вызывать нужные операции другого класса, используя делегирование. Делегирование (delegation) — это перехват операции в одном объекте и отправка ее связанному объекту. Делегирование осуществляется только для тех операций, которые имеют смысл в контексте данного объекта, поэтому опасности случайно унаследовать не имеющие смысла операции просто нет.

Более безопасный вариант класса *Stack* может делегировать операции классу *List* (рис. 15.7). Каждый экземпляр класса *Stack* содержит закрытый экземпляр класса *List*. Фактическую реализацию можно оптимизировать, используя встроенный объект или атрибут-указатель. Операция *Stack.push()* делегируется объекту *List* вызовом его операций *last()* и *add()*, посредством которых осуществляется добавление элемента в конец списка. Операция *Stack.pop()* реализуется аналогичным образом посредством операций *last()* и *remove()*. Возможность повреждения стека из-за добавления или удаления произвольных элементов в данном случае скрыта от клиентов класса *Stack*.

В некоторых языках, таких как C++ и Java, подкласс может наследовать форму суперкласса с выборочным наследованием операций предков и выборочным предоставлением операций клиентам. Это равносильно делегированию, потому что подкласс при этом не является частным случаем суперкласса и не может заменять его.

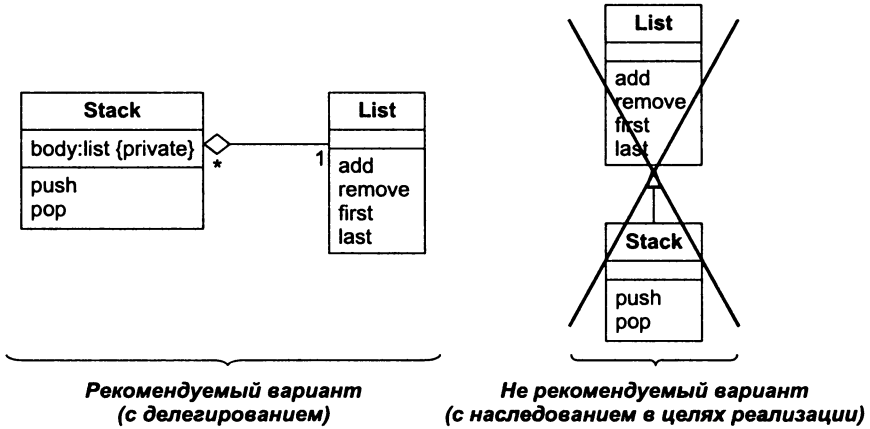


Рис. 15.7. Альтернативные варианты проектирования

Пример с банкоматом. Наша иерархия наследования имеет глубокую и четкую структуру. Именно к такому наследованию и нужно стремиться.

15.10. Организация проекта модели классов

Программы состоят из дискретных физических блоков, которые можно редактировать, компилировать, импортировать и использовать другими способами. В некоторых языках, таких как C и Fortran, блоками являются файлы исходного кода. В Ada и Java имеется явно определенная языковая конструкция под названием *package* (пакет). В работе [Coplien-92] показано, как можно использовать класс C++ для группировки статических функций-членов, вложенных классов, перечислений и констант. Организацию проекта модели классов можно усовершенствовать следующими способами:

- сокрытие внутренней информации от доступа извне;
- контроль согласованности сущностей;
- коррекция определений пакетов.

15.10.1. Соккрытие информации

На этапе анализа мы не учитывали видимость информации, потому что нам было важнее понять устройство приложения. Цели этапа проектирования несколько иные: аналитическая модель изменяется так, чтобы облегчить реализацию и обслуживание. На жизнеспособность проекта сильно влияет отделение внешней спецификации от внутренней реализации. Это называется *сокрытием информации* (information

hiding). Благодаря этому принципу вы можете изменять реализацию класса, не беспокоясь о том, что вам придется изменять код его клиентов. Возникающие в результате границы между классами ограничивают влияние изменений, благодаря чему вам становится легче предсказывать возможные результаты этих изменений.

Существует несколько способов сокрытия информации.

- **Ограничение области прослеживания модели классов.** В предельном случае метод может проследить все ассоциации модели классов, чтобы найти объект и обратиться к нему. Неограниченная видимость допустима на этапе анализа, но в реализации методы, «знающие» о модели слишком много, оказываются недолговечными и сильно страдают от изменений в модели. В процессе проектирования нужно стремиться ограничивать область, прослеживаемую каждым из методов [Lieberherr-88]. Объект должен обращаться только к непосредственно связанным с ним объектам (между ними должны быть прямые ассоциации). Объект может обращаться к косвенно связанным объектам только посредством методов промежуточных объектов.
- **Исключение прямого обращения к чужим атрибутам.** Вообще говоря, подклассы могут обращаться к атрибутам своих суперклассов. Но вот обращения к атрибутам через ассоциации допускать не следует. Вместо этого нужно вызывать операции тех классов, атрибуты которых нужно прочесть или изменить.
- **Проектирование интерфейсов на высоком уровне абстрагирования.** Нужно стремиться минимизировать связи между классами. Для этого можно, в частности, повышать уровень абстрагирования интерфейсов. Можно вызывать метод другого класса для выполнения важной операции, но не стоит беспокоить его по мелочам.
- **Соккрытие внешних объектов.** Используйте пограничные объекты для изоляции внутренней части системы от ее внешнего окружения. Пограничным называется объект, который обеспечивает пересылку запросов и ответов между внутренними и внешними объектами. Пограничный объект принимает внешние запросы в форме, удобной для клиентов, и преобразует их в форму, удобную для внутренней реализации.
- **Избегайте каскадных вызовов методов.** Не допускайте применения метода к результатам, возвращаемым другим методом, за исключением тех случаев, когда класс результата уже является поставщиком методов по отношению к вызывающему классу. Вместо этого напишите метод, который будет сочетать обе операции.

15.10.2. Согласованность сущностей

Согласованность — одно из важнейших качеств, которыми должен обладать проект. Сущность, такая как класс, операция или пакет, является внутренне согласованной, если она устроена по некоторому логичному плану и если все ее составляющие работают вместе для достижения единой цели. Сущность должна иметь одну общую тему; она не может быть совокупностью несвязанных частей.

Метод должен хорошо делать какое-то одно действие. Один метод не должен содержать и политику, и реализацию. *Политика* (policy) — это принятие решений

в зависимости от контекста. *Реализация* (implementation) — это выполнение полностью определенных алгоритмов. Политика подразумевает принятие решений, получение глобальной информации, взаимодействие с внешним миром и интерпретацию частных случаев. Методы, определяющие политику, содержат операции ввода-вывода, обращения к данным и условные операторы. В таких методах нет сложных алгоритмов, но есть вызовы других методов, реализующих нужные алгоритмы. Метод-реализация должен представлять собой код ровно одного алгоритма без всяких решений, предположений, значений по умолчанию или отклонений. Все данные поступают в виде аргументов, поэтому список аргументов у такого метода может быть довольно большим.

Отделение политики от реализации значительно увеличивает вероятность повторного использования. Методы-реализации не содержат никакой зависимости от контекста, а потому могут быть с легкостью задействованы в другой ситуации. Методы-политики обычно приходится переписывать, ориентируясь на специфику нового приложения, но к счастью они обычно получаются достаточно простыми и состоят, главным образом, из высокоуровневых решений и низкоуровневых вызовов.

Рассмотрим в качестве примера операцию, начисляющую проценты на счет «до востребования». Проценты начисляются ежедневно, но если счет закрывается, все проценты за месяц теряются. Программная логика начисления процентов состоит из двух частей: метода-реализации, который начисляет проценты по двум заданным датам, без учета штрафов или иных изменений, и метода-политики, который принимает решение, на каком временном промежутке следует вызывать метод-реализацию. Метод-реализация будет достаточно сложным, но вероятность его повторного использования велика. Метод-политика вряд ли будет использован повторно, но зато его проще написать.

Один класс не должен одновременно служить слишком большому количеству целей. Если класс получается слишком сложным, его можно разбить, используя обобщение или агрегацию. Небольшие составляющие имеют больше шансов на повторное использование, чем крупные блоки. Точные числа указать достаточно сложно, но в качестве простого правила можно использовать следующее: класс следует разбить, если он содержит более 10 атрибутов, 10 ассоциаций или 20 операций. Всегда разбивайте класс, если атрибуты, ассоциации или операции явно распадаются на несколько несвязанных групп.

15.10.3. Коррекция определений пакетов

В процессе анализа вы разбили модель классов на пакеты. Первоначальный способ разбиения может быть не самым удачным с точки зрения реализации. Пакеты нужно определять так, чтобы их интерфейсы были как можно проще и при этом были бы как можно лучше определены. Интерфейс между двумя пакетами состоит из ассоциаций, которые связывают классы одного пакета с классами другого, и операций, которые обращаются к классам через границы пакетов.

В процессе формирования пакетов можно руководствоваться связностью модели классов. В качестве грубого правила подойдет следующее: классы, тесно

связанные друг с другом ассоциациями, должны находиться в одном пакете, тогда как классы, связанные слабо или не связанные вовсе, могут быть в разных пакетах. Конечно, нужно учитывать и другие аспекты. Пакет должен иметь определенную тему, обладать функциональным единством.

Количество операций, прослеживающих некоторую ассоциацию, является удобной мерой важности соответствующей связи между классами. Речь идет не о частоте прослеживаний, а о количестве различных способов использования ассоциации. Сильно связанные классы должны находиться в одном и том же пакете.

Пример с банкоматом. На рис. 15.8 показан последний вариант модели классов предметной области банкомата после выполнения проектирования классов. (Помните, что в полную модель классов входит и модель классов приложения — рис. 13.8.)

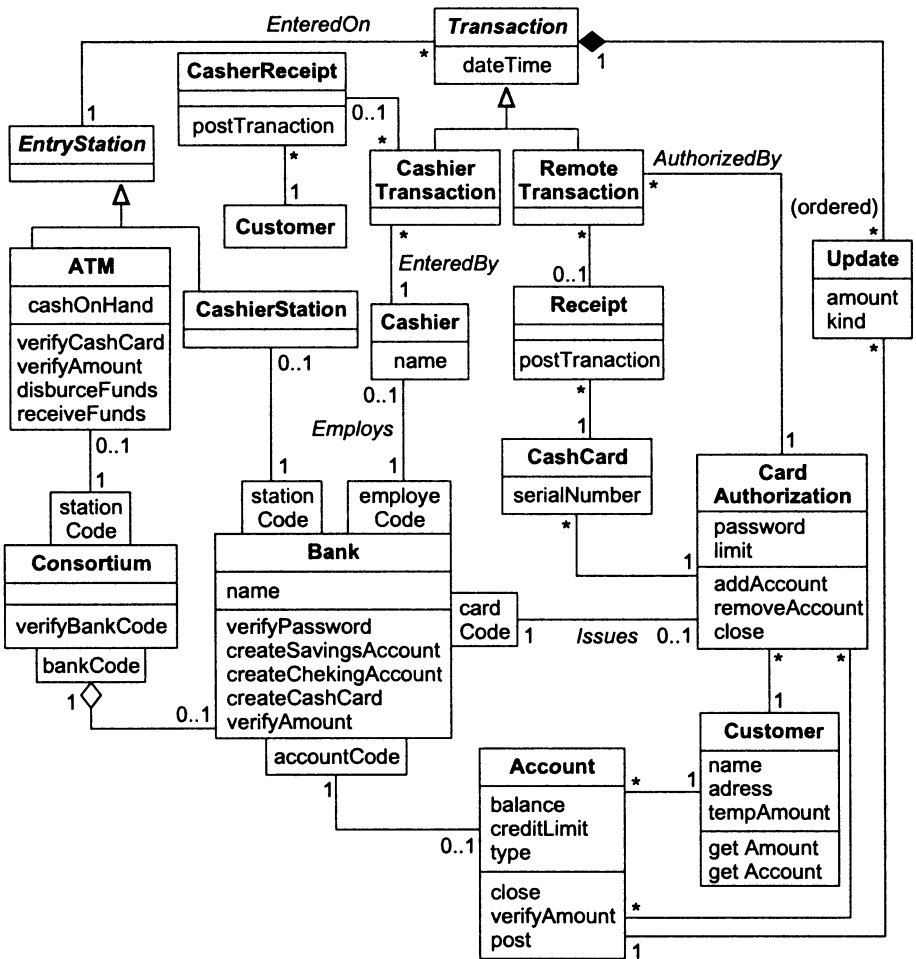


Рис. 15.8. Модель классов предметной области банкомата

15.12. Резюме

Проектирование классов не начинается на пустом месте. Этот процесс состоит в уточнении модели, полученной на предшествовавших этапах. При проектировании классов модель обрastaет деталями: вы проектируете алгоритмы, реорганизуете операции, оптимизируете классы, корректируете наследование и уточняете определения пакетов.

Первый этап проектирования классов состоит в добавлении операций, реализующих варианты использования. Варианты использования определяют требуемое поведение, но не его реализацию. Проектировщик должен придумать операции, которые обеспечат поведение, заданное вариантами использования.

После этого нужно выбрать для каждой операции наиболее подходящий алгоритм. В первую очередь нужно учитывать вычислительную сложность, однако в некоторых случаях можно жертвовать быстротой ради простоты. Высокоуровневые операции рекурсивно определяются через низкоуровневые, которые тоже нужно придумывать. Рекурсия прекращается, когда вы достигаете уже имеющихся операций или таких, реализация которых очевидна.

В исходном проекте множество операций неизбежно будет содержать несоответствия, избыточные элементы и неэффективные конструкции. Это естественно, потому что невозможно создать хороший проект целиком за один раз. Более того, по мере своего развития проект одновременно ухудшается. Операция или класс, созданные для выполнения одной задачи, не будут оптимальными для выполнения другой. Занимаясь проектированием, вы должны периодически реорганизовывать операции для повышения их ясности и устойчивости к изменениям.

Возможно, вам придется переработать аналитическую модель для повышения эффективности. В процессе оптимизации информация, содержащаяся в модели, не удаляется; напротив, к ней добавляются новые, избыточные данные, ускоряющие обращение к элементам модели и сохраняющие промежуточные результаты. Полезно бывает перепланировать алгоритмы и сократить количество подлежащих выполнению операций.

Не забывайте о пользе воплощения (превращения в объект сущности, изначально объектом не являвшейся). Например, помещение вклада на счет, снятие денег со счета и трансфер между счетами обычно считаются операциями, потому что это действия, выполняемые некоторыми объектами. Однако в нашем примере мы сделали транзакцию объектом, чтобы иметь возможность описывать ее.

В процессе проектирования классов можно корректировать определения классов и операций для улучшения качества иерархии наследования. К таким корректировкам относятся изменение списка аргументов метода, перемещение атрибутов и операций из класса в суперкласс, определение абстрактного суперкласса для вынесения в него общего поведения нескольких классов, а также деление операции на наследуемую и конкретизирующую части. Используйте делегирование вместо наследования, если класс подобен другому классу, но не является его подклассом.

Программы приходится разбивать на физические блоки, с которыми работают редакторы и компиляторы. Это удобно и для совместной работы внутри команды программистов. Скрытие информации позволяет гарантировать, что будущие

изменения затронут лишь ограниченный объем кода. Пакеты должны быть цельными, а сущности внутри них должны объединяться одной темой.

Таблица 15.1. Ключевые понятия главы

абстрагирование суперкласса	выводимый атрибут	политика vs реализация
улучшение наследования	реализация наследования	рекурсия
алгоритм	индекс	реорганизация
структура данных	сокрытие информации	воплощение
делегирование	оптимизация	ответственность
выводимая ассоциация	пакет	вариант использования

Библиографические замечания

Алгоритмы и структуры данных рассматриваются в любом базовом курсе информатики. В качестве хорошей литературы по этой теме можно рекомендовать [Aho-83] и [Sedgewick-95].

Добавление индексов и изменение порядка доступа для повышения производительности — широко используемая методика оптимизации баз данных. Примеры ее использования приводятся в [Ullman-02].

Одна из первых попыток найти правила определения видимости, сохраняющие объектно-ориентированную модульность, была сделана в работе [Lieberherr-88]. В [Meyer-97] приводятся рекомендации по стилю использования классов и операций.

В первом издании этой книги этап проектирования классов назывался этапом *проектирования объектов* (object design).

Литература

[Aho-93] Alfred Aho, John Hopcroft, and Jeffrey Ullman. Data Structures and Algorithms. Boston: Addison-Wesley, 1983.

[Coplien-92] James O. Coplien. Advanced C++: Programming Styles and Idiom. Boston: Addison-Wesley, 1992.

[Fowler-99] Martin Fowler. Refactoring: Improving the Design of Existing Code. Boston: Addison-Wesley, 1999.

[Gamma-95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley, 1995.

[Lieberherr-88] K. Lieberherr, I. Holland, A. Riel. Object-oriented programming: an objective sense of style. OOPSLA'88 as ACM SIGPLAN 23, 11 (November 1988), 323-334.

[Meyer-97] Bertrand Meyer. Object-Oriented Software Construction, Second Edition. Upper Saddle River, NJ: Prentice Hall, 1997.

[Sedgewick-95] Robert Sedgewick, Philippe Flajolet, and Peter Gordon. An Introduction to the Analysis of Algorithms. Boston: Addison-Wesley, 1995.

[Ullman-02] Jeffrey Ullman and Jennifer Widom. A First Course in Database Systems. Upper Saddle River, NJ: Prentice Hall, 2002.

[Wirfs-Brock-90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. Designing Object-Oriented Software. Upper Saddle River, NJ: Prentice Hall, 1990.

Упражнения

- 15.1. (6) Рассмотрите варианты использования из упражнения 13.9 и перечислите по крайней мере четыре области ответственности для каждого из них. (Замечание для преподавателя: можно предоставить студентам наш ответ к упражнению 13.9.)
- 15.2. (6) Рассмотрите три первых варианта использования из упражнения 13.16 и перечислите по крайней мере четыре области ответственности для каждого из них. (Замечание для преподавателя: можно предоставить студентам наш ответ к упражнению 13.16.)
- 15.3. (4) Напишите алгоритмы построения перечисленных ниже геометрических фигур в графическом терминале. Фигуры не закрашиваются. Терминал работает в растровом режиме. Укажите все сделанные вами предположения:
 - 1) окружность;
 - 2) эллипс;
 - 3) квадрат;
 - 4) прямоугольник.
- 15.4. (3) Обсудите, подходит ли алгоритм прорисовки эллипса из предыдущего упражнения для построения окружностей, а алгоритм прорисовки прямоугольников для квадратов.
- 15.5. (3) Аккуратное упорядочение операций умножения и сложения позволяет свести к минимуму количество арифметических операций, необходимых для вычисления значения полинома. Например, полином вида $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ можно вычислить почленно, добавляя каждое слагаемое к общей сумме после его вычисления. Это потребует 10 операций умножения и 4 операции сложения. Можно изменить порядок арифметических операций в соответствии со следующей формулой: $x(x(x(xa_4 + a_3) + a_2) + a_1) + a_0$, так что потребуется всего 4 умножения и 4 сложения. Сколько операций потребуется для вычисления полинома n -го порядка каждым из методов? Обсудите преимущества и недостатки каждого метода.
- 15.6. (4) Усовершенствуйте диаграмму классов на рис. У15.1, обобщив классы *Ellipse* (Эллипс) и *Rectangle* (Прямоугольник) посредством класса *GraphicPrimitive* (ГрафическийПримитив). У вас должна получиться единственная ассоциация типа один-к-одному с классом *BoundingBox* (ОграничивающийПрямоугольник). Фактически вы измените значение кратности с 0..1 на ровно 1. На рис. У15.1 класс *BoundingBox* используется *Ellipse* и *Rectangle* совместно. Ограничивающий прямоугольник — это прямоугольник минимальной площади, содержащий соответствующий эллипс или прямоугольник.

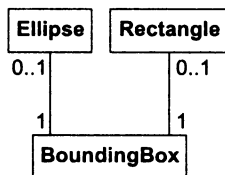


Рис. У15.1. Часть диаграммы с общим классом

- 15.7. (5) Какие классы из предыдущего упражнения должны предоставлять операцию *delete* (удаление), видимую внешнему миру? Удаление означает уничтожение объекта. Поясните свой ответ.
- 15.8. (4) Измените диаграмму классов на рис. У15.2 таким образом, чтобы параметры полей содержались в отдельном классе. Определенные категории страниц могут иметь заданные по умолчанию значения полей, но для некоторых страниц значения по умолчанию могут перекрываться.

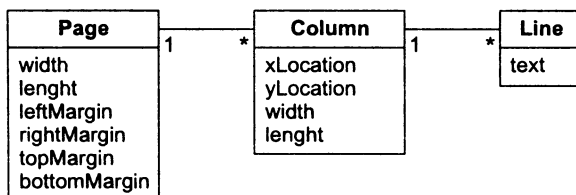


Рис. У15.2. Часть диаграммы классов для газеты

- 15.9. (3) Измените диаграмму на рис. У15.2 таким образом, чтобы можно было определить, на какой странице (*Page*) находится строка (*Line*), без предварительного поиска соответствующей колонки (*Column*).
- 15.10. (7) Напишите псевдокод для каждого метода на рис. У15.3. Операция *initialize* (инициализация) помещает 52 карты в колоду (*Deck*). Операции *insert* (вставка) и *delete* (удаление) принимают в качестве единственного аргумента карту (*Card*) и осуществляют помещение карты в совокупность или удаление ее оттуда. При этом принудительно выполняется перерисовка совокупности. Операция *delete* применима только к верхней карте стопки (*Pile*). Операция *sort* (сортировка) осуществляет сортировку карт на руках (*Hand*) по масти (*suit*) и по старшинству (*rank*).

Класс *Pile* является абстрактным. Операции *topOfPile* и *bottomOfPile* — это запросы. *Draw* (взять) снимает верхнюю карту со стопки и помещает ее в руку (*Hand*), которая передается в качестве аргумента.

Операция *shuffle* (перемешивание) перемешивает колоду. При раздаче (*deal*) карты по одной берутся из колоды и раздаются игрокам на руки (объекты класса *Hand* при этом создаются и возвращаются в виде массива). Операция *display* отображает карту. *Compare* сравнивает карты и определяет, какая из них старше. *Discard* удаляет карту из совокупности и помещает ее в стопку, переданную в качестве аргумента.

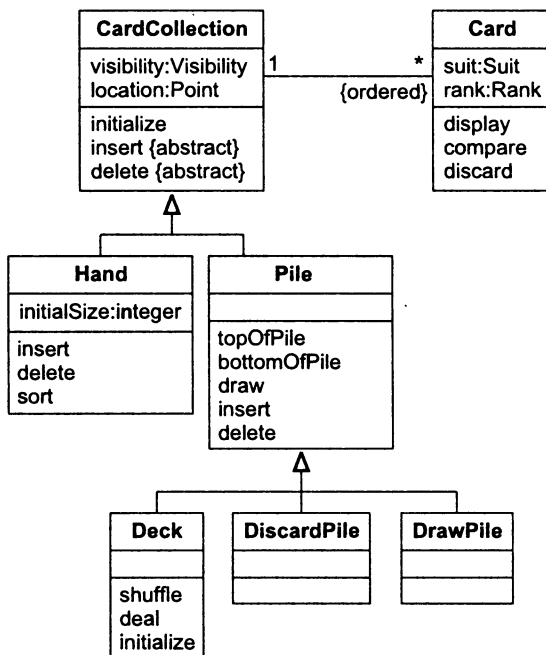


Рис. У15.3. Часть диаграммы классов для программы, играющей в карты

- 15.11. (5) Напишите псевдокод для вычисления суммы очков, набранных за попытку (рис. У12.4).

Каждая попытка участника (*Trial*) оценивается несколькими судьями. Каждый из них поднимает карточку с числом очков. Специально назначенный человек зачитывает очки секретарям. Трое секретарей записывают очки, выбрасывают одно минимальное и одно максимальное значения, после чего суммируют оставшиеся очки. Они сравнивают результаты между собой, чтобы исключить арифметические и иные ошибки. В некоторых случаях они могут попросить зачитать очки повторно. Полученные результаты передаются трем другим секретарям, которые умножают результат на коэффициент сложности и вычисляют среднее значение. Полученные значения сравниваются для исключения арифметических и других ошибок.

- 15.12. Напишите псевдокод для перечисленных ниже операций классов (рис. У12.4). Вам придется добавить на диаграмму ассоциацию *RegisteredFor* (Зарегистрирован) типа многие-ко-многим между множеством выступлений *Events* и упорядоченным списком участников *Competitors*, чтобы отслеживать тех участников, которые зарегистрированы на какое-либо выступление. Планирование попыток осуществляйте с учетом порядка регистрации участников.

- 1) (3) Поиск выступления (*Event*) по упражнению (*Figure*) и соревнованию (*Meet*).
- 2) (3) Регистрация участника (*Competitor*) на выступление (*Event*).

- 3) (3) Регистрация участника (*Competitor*) на все выступления (*Event*) соревнования (*Meet*).
- 4) (5) Выбор и планирование выступлений (*Events*) соревнования (*Meet*).
- 5) (3) Планирование соревнований (*Meet*) сезона (*Season*).
- 6) (4) Распределение судей (*Judge*) и секретарей (*Scorekeeper*) по станциям (*Station*).

15.13. (8) На рис. У15.4 приведена часть диаграммы классов метамодели, которая может использоваться в компиляторе объектно-ориентированного языка. Напишите псевдокод для метода *traceInheritancePath*, который будет проследивать иерархию наследования так, как это описано ниже. Метод возвращает упорядоченный список классов от более общего класса к более конкретному. Проследивание осуществляется только по цепочке обобщений; агрегации и ассоциации не учитываются. Если проследивание не дает результатов, возвращается пустой список. Вы можете исходить из предположения, что множественное наследование не поддерживается.

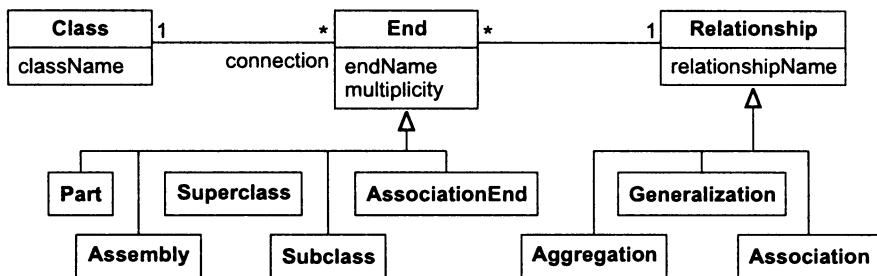


Рис. У15.4. Часть диаграммы классов метамодели

15.14. (8) Усовершенствуйте рис. У15.4, удалив ассоциации классов *End* и *Relationship*, заменив их ассоциациями с подклассами *End* и *Relationship*. Это пример преобразования диаграммы классов. Напишите псевдокод операции *traceInheritancePath* для новой диаграммы классов.

15.15. (7) В соответствии с рис. У15.4 напишите алгоритм операции, которая будет генерировать имя для безымянной ассоциации. На вход операции поступает экземпляр ассоциации (*Association*). Операция должна вернуть глобально уникальное имя *relationshipName*. Если у ассоциации уже есть имя, операция должна вернуть именно его. В противном случае имя генерируется в соответствии со стратегией, которую вы должны придумать самостоятельно.

Какую именно стратегию вы выберете, не так уж важно, но имена обязательно должны быть уникальными, причем по имени вы должны иметь возможность определить, к какой ассоциации оно относится. Вы можете предположить, что все ассоциации являются бинарными. Можете предположить также, что аналогичная операция для класса *End* уже есть: она возвращает имя полюса *endName*, уникальное в контексте данного отношения. Если имя, которое вы формируете, конфликтует с уже имеющимися, измените его таким образом, чтобы оно стало уникальным. Если вы хотите

изменить диаграмму или использовать дополнительные структуры данных, можете это сделать, но не забудьте описать все добавленные элементы.

- 15.16. (7) Усовершенствуйте диаграмму классов (рис. У15.5), трансформировав ее путем добавления класса *PoliticalParty* (ПолитическаяПартия). Свяжите классы *Voter* (Избиратель) и *PoliticalParty* (ПолитическаяПартия) ассоциацией. Объясните, почему это преобразование улучшает модель.
- 15.17. (7) Иногда авиакомпании выпускают на рейсы с недостаточным количеством пассажиров самолеты с меньшим числом мест. Напишите алгоритм перераспределения мест, причем такой, чтобы места с небольшими номерами рядов не переназначались. Предполагается, что количество кресел в ряду в обеих моделях самолетов одинаковое.
- 15.18. (8) Необходимость создания эффективной реализации вынуждает добавлять в модель классы, не содержащиеся в исходной постановке задачи. Например, двухмерная система автоматизированного проектирования может использовать специализированные структуры для определения попадания точек внутрь прямоугольного окна, заданного пользователем.

Один из методов решения задачи заключается в хранении совокупности точек, отсортированных сначала по x , а затем по y . При этом точки, попадающие в прямоугольное окно, можно выделить, не проверяя координаты *всех*.

Постройте диаграмму классов, описывающую совокупность точек, упорядоченных по x и по y . Напишите псевдокод для операций *search* (поиск), *add* (добавление), *delete* (удаление). В качестве аргументов операция *search* принимает описание прямоугольной области и собственно совокупность точек. Операция возвращает множество всех точек входной совокупности, попадающих внутрь прямоугольной области. Операции *add* и *delete* принимают в качестве аргументов точку и совокупность точек (при этом точка добавляется в совокупность или удаляется из нее).

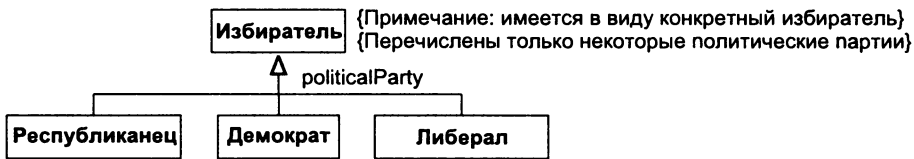


Рис. У15.5. Диаграмма классов, описывающая членство избирателей в политических партиях

- 15.19. (8) Определите, по какому закону время поиска в предыдущем упражнении зависит от количества точек в совокупности. Перечислите все сделанные предположения.
- 15.20. (3) При выборе алгоритма важно оценить его требования к ресурсам. Каким образом время выполнения перечисленных ниже алгоритмов зависит от указанных параметров?
- 1) Алгоритм из упражнения 15.13, глубина иерархии наследования.
 - 2) Алгоритм из упражнения 15.17, количество пассажиров.

Резюме процесса разработки

16

Процесс разработки лежит в основе организованного производства программного обеспечения (рис. 16.1). Мы считаем, что этот процесс должен задействовать объектно-ориентированные концепции и систему обозначений UML.

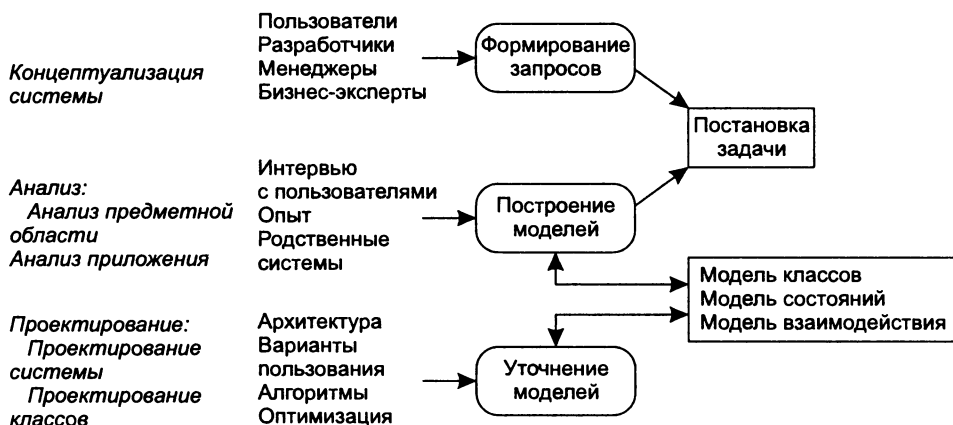


Рис. 16.1. Обзор процесса разработки

Обратите внимание: разработчику не приходится переходить от одной модели к другой, потому что объектно-ориентированная парадигма охватывает этапы анализа, проектирования и реализации. Эта парадигма одинаково хорошо подходит для описания реального мира и компьютерной реализации.

Мы описываем процесс разработки как последовательность этапов, но на практике объектно-ориентированная разработка представляет собой итерационный процесс. Когда модель кажется завершенной на одном уровне абстракций, вы должны рассмотреть следующий, более низкий уровень. На каждом уровне вам придется добавлять новые операции, атрибуты и классы. Возможно, вам

даже придется пересмотреть отношения между объектами (и даже внести изменения в иерархию наследования). Подробнее об итерационной разработке рассказывается в главе 21.

16.1. Концептуализация системы

Концептуализация системы — это рождение приложения. Человеку, который разбирается и в технологии, и в потребностях бизнеса, приходит в голову *идея* приложения. Цель этапа концептуализации — охватить всю картину целиком: зачем нужна система, можно ли ее разработать за разумную цену, покроет ли спрос на систему затраты на ее создание. До этапа концептуализации у вас есть идея нового приложения. После него — постановка задачи, которая является отправной точкой для анализа.

16.2. Анализ

Анализ — это разработка моделей, позволяющих лучше понять требования к системе. Цель анализа — описать, что именно нужно сделать, а не то, каким образом это следует сделать. Вы должны понять проблему прежде, чем начнете решать ее. Важно учесть все доступные источники информации, включая описание задачи, интервью с пользователями, предыдущий опыт и артефакты из родственных систем. На выходе этапа анализа получается набор моделей, детально и полностью описывающих систему. Анализ делится на два этапа: анализ предметной области и анализ приложения.

16.2.1. Анализ предметной области

Анализ предметной области охватывает общие знания о приложении: понятия и отношения между ними, известные экспертам в данной области. Цель состоит в том, чтобы выработать точную, четкую, понятную и корректную модель реального мира. В результате анализа предметной области обычно формируется модель классов и, в некоторых случаях, модели состояний. Реже подготавливаются модели взаимодействия. Самое важное в построении модели предметной области — решить, какую именно информацию следует включить в модель (определить границы приложения) и каким образом ее представить (выбрать уровень абстрагирования).

16.2.2. Анализ приложения

Анализ приложения охватывает компьютерные аспекты системы, видимые пользователям. Объекты приложения — это не просто внутренние проектные решения. Их важность обусловлена тем, что с ними будут иметь дело пользователи. Классы приложения делятся на контроллеры, устройства и пограничные объекты. На этом этапе формируется в первую очередь модель взаимодействия, однако модели классов и состояний тоже не остаются без изменений.

16.3. Проектирование

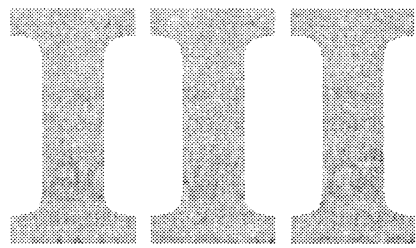
Анализ отвечает на вопрос «что делает приложение», а проектирование говорит «как это делается». Добившись полного понимания устройства приложения на этапе анализа, вы можете переходить к построению практического решения, которое можно будет впоследствии поддерживать. Модель для проектирования может быть совершенно независимой от аналитической модели, но чаще всего оптимальный подход состоит в перенесении классов аналитической модели в проектную модель. После этого проектирование сводится к детализации модели и к принятию решений. Проектирование делится на два этапа: проектирование системы и проектирование классов.

16.3.1. Проектирование системы

Цель проектирования системы — выработать высокоуровневую стратегию (архитектуру) решения задачи по созданию приложения. Выбор архитектуры — это важное решение, обладающее широким спектром последствий. Этот выбор основывается на предъявляемых требованиях и на предыдущем опыте. Проектировщик системы должен определить политику, которая будет направлять действия на следующем этапе.

16.3.2. Проектирование классов

Проектирование классов сводится к расширению и корректировке аналитических моделей реального мира. При этом модель подготавливается к реализации. Проектировщики классов формируют завершенные определения классов и ассоциаций, а также выбирают алгоритмы для операций.



Реализация

В первой и второй части мы описали объектно-ориентированные концепции и процедуры анализа и проектирования, в ходе которых эти концепции применяются. В третьей части речь пойдет об оставшихся этапах процесса разработки программного обеспечения. Мы обсудим конкретные детали реализации систем с использованием C++, Java и баз данных. Объектно-ориентированные модели можно применять с любыми языками, но в этой книге мы не будем рассматривать языки, не относящиеся к числу объектно-ориентированных, потому что большинство разработчиков в наше время такие языки не используют.

В главе 17 обсуждаются вопросы реализации, не зависящие от выбора языка. Речь пойдет о методах реализации ассоциаций, потому что большинство языков не имеют встроенной поддержки элементов этого типа.

В главе 18 рассматриваются принципы реализации объектно-ориентированного проекта на C++ и Java. Мы выбрали именно эти языки потому, что они наиболее популярны среди языков программирования.

В главе 19 показано, как можно реализовать проект при помощи базы данных. Опять же, мы рассматриваем реляционную базу данных потому, что базы такого типа доминируют на рынке. Логично предположить, что объектно-ориентированный проект можно реализовать и в объектно-ориентированной базе данных, но такие базы недостаточно широко распространены и используются только в особых случаях.

В главе 20 приведены рекомендации по стилю программирования на любом языке. В конце концов решением любой задачи становится код программы, а потому от стиля кодирования зависит легкость обслуживания и возможность расширения системы.

Третья часть книги завершает рассказ о том, как нужно разрабатывать приложения с помощью объектно-ориентированных концепций. В четвертой части мы расскажем о вопросах проектирования программного обеспечения, относящихся к большим и сложным приложениям.

Моделирование реализации

Реализация (implementation) — это последний этап разработки, на котором приходится иметь дело со спецификой языков программирования. Реализация должна быть очевидной, почти автоматической, потому что все сложные решения нужно было принять на этапе проектирования. Вам предстоит перевести эти решения на язык программирования. Добавление деталей в процессе написания кода неизбежно, но эти детали не должны затрагивать большие участки программы.

17.1. Обзор реализации

Пришло время пожинать плоды своих трудов. На этапе реализации вы сможете воспользоваться результатами тщательной подготовки к решению задачи, которая была проведена раньше, на этапах анализа и проектирования. В первую очередь нужно заняться вопросами реализации, не зависящими от выбранного языка. Это мы называем *моделированием реализации* (implementation modeling). Данный этап состоит из следующих шагов:

1. Уточнение классов (раздел 17.2).
2. Уточнение обобщений (раздел 17.3).
3. Реализация ассоциаций (раздел 17.4).
4. Подготовка к тестированию (раздел 17.5).

Первые два шага мотивируются теорией преобразований. Преобразование (transformation) — это отображение из области моделей в диапазон моделей. При моделировании нужно ориентироваться на требования заказчика, но при этом смотреть на вещи с абстрактной математической точки зрения.

17.2. Уточнение классов

Иногда бывает полезно подкорректировать классы, прежде чем приступить к написанию кода, чтобы упростить разработку или повысить производительность. Не забывайте, что цель реализации состоит в том, чтобы воплотить в жизнь модели, полученные на этапах анализа и проектирования. Не изменяйте проектную модель, если для этого нет достаточно веских причин. Если же такие причины есть, вы можете рассмотреть следующие возможности.

- **Разбиение класса.** На рис. 17.1 мы можем объединить рабочий и домашний адреса в одном классе или разделить эту информацию между двумя классами. Оба подхода вполне допустимы. Если данных много, их лучше разделить. Если данных не слишком много, их можно объединить.

Разбиение класса может быть усложнено обобщениями и ассоциациями. Например, если класс *Person* (Человек) был суперклассом для каких-то других классов, то после его разбиения подклассам будет сложно получить всю информацию. Придется либо допустить множественное наследование, либо устанавливать ассоциацию между классами *PersonHomeInfo* и *PersonOfficeInfo*. Если же класс *Person* будет связан с другими классами какими-либо ассоциациями, вам придется решать, как их можно изменить.

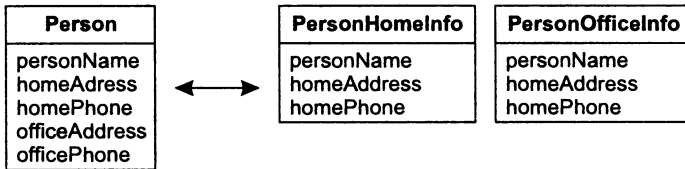


Рис. 17.1. Разбиение класса

- **Слияние классов.** Слияние — это процедура, обратная разбиению. Если бы мы начинали с классов *PersonHomeInfo* и *PersonOfficeInfo* с рис. 17.1, мы могли бы объединить их. На рис. 17.2 показан другой пример с ассоциациями. В данном случае нельзя сказать, что одно представление само по себе лучше другого, потому что с математической точки зрения они идентичны друг другу. Кроме того, нужно учитывать влияние принятых решений на ассоциации и обобщения.

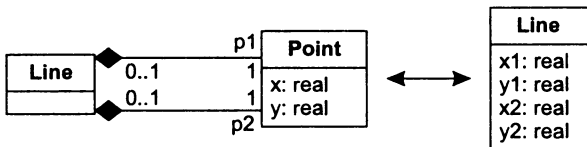


Рис. 17.2. Слияние классов

- **Разбиение и слияние атрибутов.** Описанные выше приемы можно применять и к атрибутам, что демонстрирует рис. 17.3.

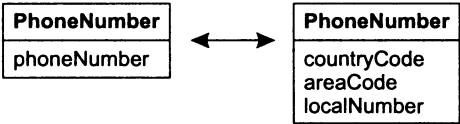


Рис. 17.3. Разбиение и слияние атрибутов

- **Превращение атрибута в класс или класса в атрибут.** Как показано на рис. 17.4, мы можем представить адрес в виде атрибута, одного класса или нескольких родственных классов. Нижний вариант может быть полезен для предварительной загрузки данных в приложение.

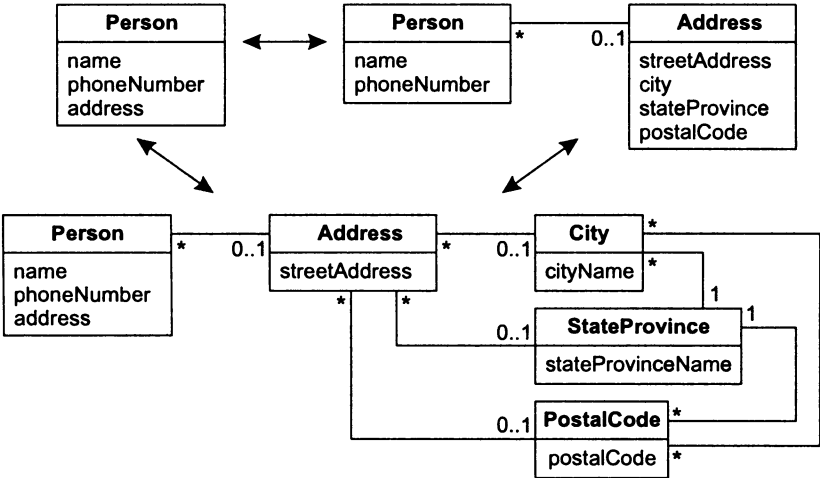


Рис. 17.4. Превращение атрибута в класс и класса в атрибут

Пример с банкоматом. Мы можем разделить *Customer.address* (Клиент.адрес) на несколько классов, если планируем осуществлять предварительную загрузку данных об адресах клиентов. Например, мы можем загрузить значения *city* (город), *stateProvince* (штат/область) и *postalCode* (индекс) для удобства сотрудников службы поддержки.

Мы можем выделить атрибут *Account.type* в отдельный класс. Это упростит программирование специального поведения. Например, можно запрограммировать разные меню для вкладов до востребования и срочных вкладов.

В любом случае, наша модель не так уж велика, и мы готовили ее достаточно аккуратно, так что вряд ли сможем внести в нее много полезных изменений на данном этапе.

17.3. Уточнение обобщений

Корректировать можно не только классы, но и обобщения. Иногда бывает полезно удалить одно обобщение или добавить другое перед переходом к непосредственному кодированию.

На рис. 17.5 показана модель перевода, взятая из одного приложения, которым недавно занимались авторы этой книги. Служба перевода преобразует *Translation-Concept* (ПереводимоеПонятие) в *Phrase* (Фраза) на требуемом языке. *MajorLanguage* (ОсновнойЯзык) — это обычный язык (английский, французский и т. п.). *MinorLanguage* — это диалект языка (американский английский, австралийский английский и т. п.). Все подлежащие переводу записи в базе данных приложения имеют идентификатор *translationConceptID*. Переводчик сначала пытается найти фразу, соответствующую понятию, в указанном *MinorLanguage*, а затем, если не удастся, переходит к соответствующему *MajorLanguage*.

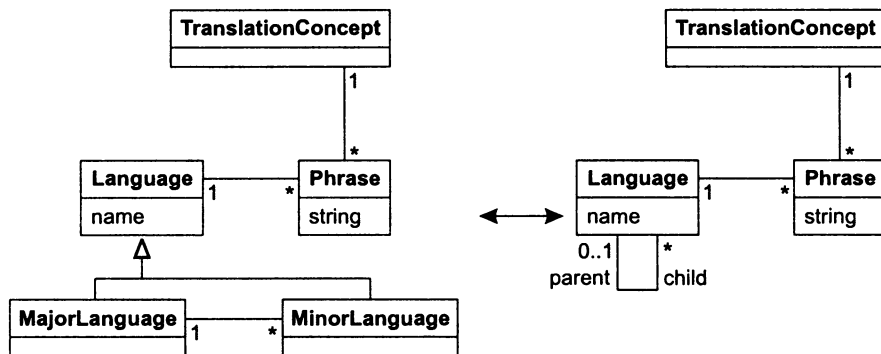


Рис. 17.5. Добавление и удаление обобщений

Для простоты реализации мы удалили обобщение и использовали модель, представленную в правой части рисунка. Поскольку служба перевода отделена от модели приложения, нам не пришлось рассматривать никаких дополнительных обобщений, и упрощение не вызвало проблем.

Пример с банкоматом. В разделе 13.1.1 мы отметили, что модель классов предметной области банкомата охватывает два приложения: собственно банкомат и кассовый терминал. На этапе анализа мы не стали беспокоиться на этот счет, потому что нам нужно было понять бизнес-требования, а клиента в принципе не волнует структура предоставляемых сервисов. Кроме того, нам нужно было обеспечить одинаковое поведение обоих приложений. Однако на этапе реализации мы должны отделить приложения друг от друга и ограничить объем работы. На рис. 17.6 мы удалили из модели классов часть, относящуюся к кассовому аппарату, в результате чего исчезли оба обобщения.

На рис. 17.6 представлена полная модель классов банкомата. Верхняя половина — модель классов предметной области, а нижняя — модель классов приложения. На этой диаграмме мы указали не все имеющиеся в модели операции.

17.4. Реализация ассоциаций

Ассоциации — это «клей», соединяющий в одно целое модель классов. Мы должны сформулировать стратегию реализации ассоциаций. Можно выбрать глобальную стратегию и реализовать все ассоциации одинаковым образом или выбирать конкретную методику для каждой ассоциации в отдельности с учетом того, как

она будет использоваться в приложении. Начнем мы с анализа прослеживани ассоциаций в модели.

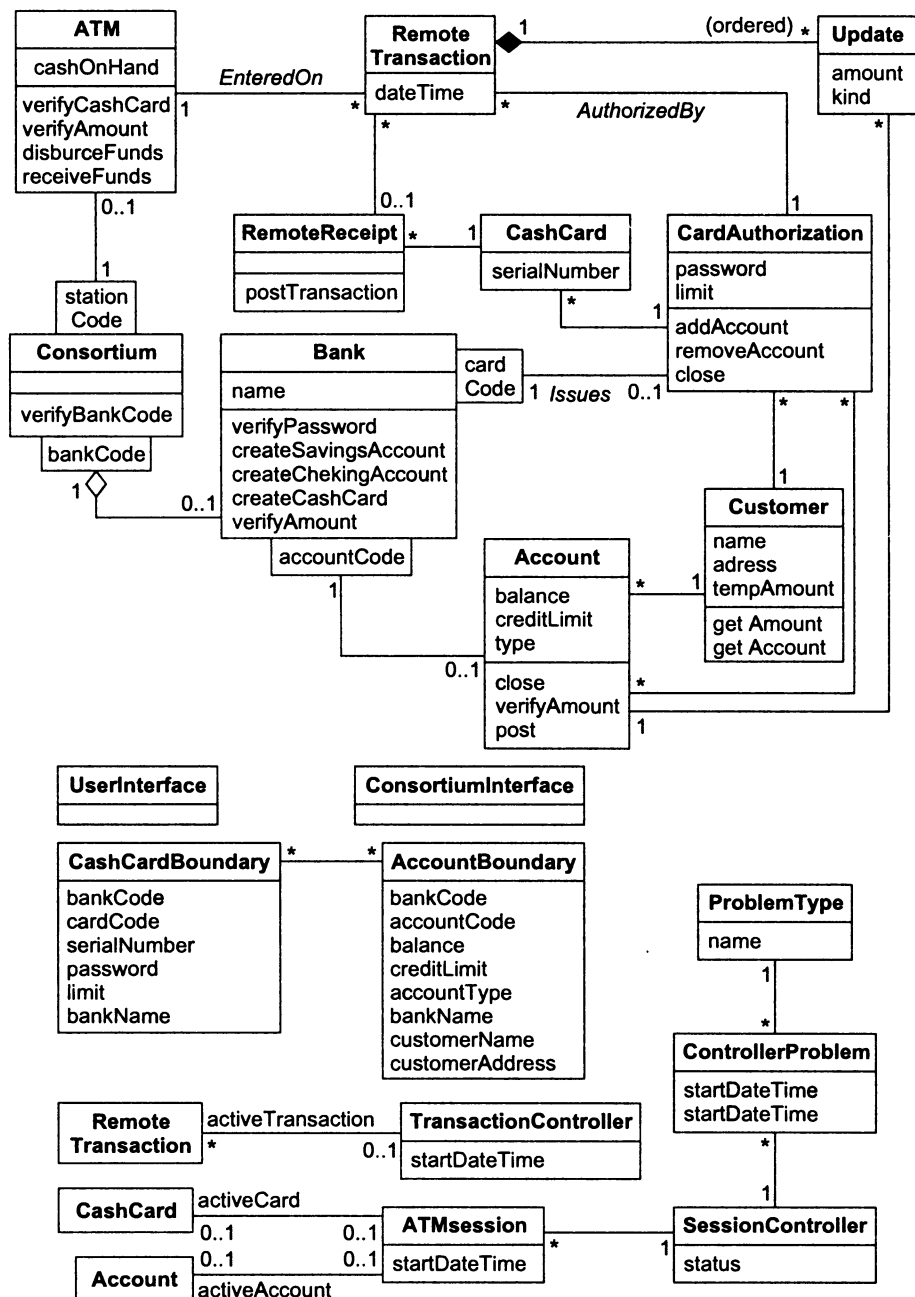


Рис. 17.6. Модель классов банкомата для реализации

17.4.1. Анализ прослеживания ассоциаций

До этого момента мы предполагали, что все ассоциации являются двусторонними. В абстрактном смысле это, несомненно, правильное предположение. Но если в вашем приложении есть ассоциации, которые прослеживаются только в одном направлении, их реализацию можно существенно упростить. Впрочем, не забывайте о том, что в будущем требования могут измениться и вам придется добавлять операцию, которая будет прослеживать эту ассоциацию в обратном направлении.

Для работы с прототипами мы всегда используем двусторонние ассоциации, чтобы иметь возможность быстро добавлять новое поведение и изменять приложение. Однако в рабочей версии приложения некоторые ассоциации оптимизируются. В любом случае, реализация должна быть скрыта. Тогда вам проще будет поменять свое решение.

17.4.2. Односторонние ассоциации

Если ассоциация прослеживается только в одном направлении, ее можно реализовать при помощи указателя — атрибута, содержащего ссылку на объект. Обратите внимание, что в этой главе мы употребляем слово *указатель* (pointer) в логическом смысле. При реализации может быть использован указатель или ссылка выбранного языка программирования или даже внешний ключ базы данных. Если ассоциация имеет кратность «один» (рис. 17.7), указатель тоже будет один. Если же кратность ассоциации «много», придется работать со множеством указателей.

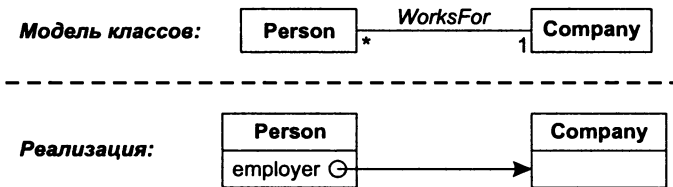


Рис. 17.7. Реализация односторонней ассоциации при помощи указателей

17.4.3. Двусторонние ассоциации

Чаще всего ассоциации прослеживаются в обоих направлениях, хотя и не с одинаковой частотой. Существует три подхода к реализации таких ассоциаций.

- **Односторонняя реализация.** Реализовать в виде одностороннего указателя и выполнять поиск при прослеживании в обратном направлении. Этот подход полезен только в том случае, когда частота прослеживания в двух направлениях отличается очень сильно, а для приложения важно сократить затраты на хранение и обновление информации. Прослеживание в обратном направлении будет при этом дорогостоящей операцией.

- **Двусторонняя реализация.** Реализовать с указателями в обоих направлениях, как показано на рис. 17.8. Этот подход обеспечивает быстрый доступ, но если объект у одного из полюсов ассоциации обновляется, на втором полюсе также необходимо выполнить обновление, чтобы связь осталась согласованной. Этот подход полезен в том случае, когда запросы выполняются чаще, чем обновления.

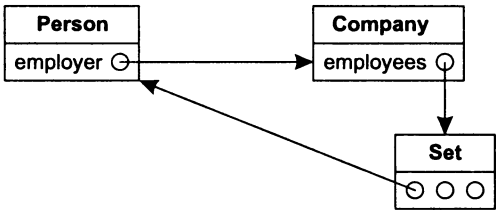


Рис. 17.8. Реализация двусторонней ассоциации через указатели

- **Реализация при помощи объекта.** Реализовать ассоциации при помощи выделенного объекта, не зависящего от обоих связанных классов (рис. 17.9) [Rumbaugh-87]. Объект ассоциации — это множество пар связанных объектов (или троек связанных объектов в том случае, когда ассоциация является квалифицированной), которое хранится в одном объекте переменного размера. В целях повышения эффективности можно реализовать объект ассоциации при помощи двух словарных объектов, каждый из которых будет связан с одним из направлений. При данном подходе доступ получается несколько более медленным, чем при использовании указателей, но с применением хэширования время доступа остается независящим от количества объектов. Этот подход удобен для расширения predetermined классов библиотек, которые не могут быть изменены, потому что в данном случае добавления атрибутов к исходным классам не требуется. Выделенные объекты также удобны для «разреженных» ассоциаций, в которых не участвует большинство объектов класса, потому что память расходуется только на фактически существующие связи.

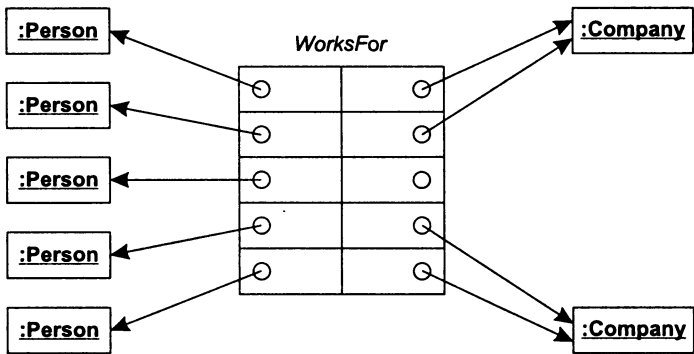


Рис. 17.9. Реализация ассоциации в виде объекта

17.4.4. Сложные ассоциации

Методы реализации сложных ассоциаций зависят от их особенностей.

- **Классы ассоциаций.** Обычно такую ассоциацию превращают в класс. Это позволяет реализовать как атрибуты, так и ассоциации, относящиеся к классу. Однако превращение изменяет смысл модели: у такой ассоциации имеется своя собственная индивидуальность, а потому методы должны каким-то образом принудительно поддерживать зависимость класса ассоциации от участвующих в ассоциации классов.

Если ассоциация относится к типу «один-к-одному» и не имеет дальнейших ассоциаций, ее можно реализовать при помощи указателей, а атрибуты ассоциации хранить в объектах участвующих классов. То же касается ассоциации типа «один-ко-многим»: атрибуты ассоциации в этом случае хранятся в объектах класса, имеющего кратность «много», потому что каждый объект этого класса участвует в ассоциации один раз.

- **Упорядоченные ассоциации.** Используйте упорядоченное множество указателей (аналогично рис. 17.8) или словарь с упорядоченным набором пар (аналогично рис. 17.9).
- **Последовательности.** То же, что и упорядоченная ассоциация, но требует использования списка указателей.
- **Мультимножества.** То же, что и упорядоченная ассоциация, но требует использования массива указателей.
- **Квалифицированные ассоциации.** Реализуйте квалифицированную ассоциацию с кратностью «один» в виде объекта-словаря (см. рис. 17.9). Квалифицированные ассоциации с кратностью «много» встречаются редко. Их можно реализовать в виде словаря множеств объектов.
- **N-арные ассоциации.** Сделайте ассоциацию классом. При этом изменится смысл модели, что придется компенсировать дополнительным кодом (как для классов ассоциаций).
- **Агрегация.** Агрегацию следует рассматривать как обычную ассоциацию.
- **Композиция.** Композицию тоже нужно рассматривать как обычную ассоциацию. Зависимость части от целого придется обеспечивать дополнительным кодом.

Пример с банкоматом. Реализация ассоциаций из модели банкомата рассматривается в упражнении 17.1.

17.5. Тестирование

Аккуратное моделирование приложения сокращает количество ошибок в будущей системе и позволяет снизить затраты на ее тестирование. Однако полностью обойтись без проверки все равно не удастся. Тестирование — это механизм контроля

качества, предназначенный для обнаружения оставшихся ошибок. Кроме того, оно позволяет независимо оценить качество вашего программного обеспечения. Количество ошибок, обнаруженных в процессе тестирования, — индикатор качества программы. С ростом опыта в моделировании эта величина должна сокращаться. Аккуратно записывайте все обнаруживаемые ошибки, а также жалобы клиентов.

Если система достаточно цельная, основную сложность представляет поиск случайных ошибок. Их исправление практически не вызывает затруднений. Напротив, если программа разработана бессистемно, исправить найденные ошибки может быть достаточно сложно.

Тестирование нужно проводить на всех стадиях разработки, а не только на этапе реализации. Однако природа тестирования на разных этапах разная. В процессе анализа вы сравниваете модель с ожиданиями пользователя, задавая вопросы и пытаясь обнаружить в модели ответ на них. В процессе проектирования вы тестируете архитектуру и можете моделировать ее производительность. В процессе реализации вы тестируете реальный код, а модель служит для поиска возможных маршрутов прослеживания.

При тестировании следует переходить от небольших составляющих ко всему приложению в целом. Разработчикам следует в первую очередь проверять свой собственный код, свои классы и методы. Это называется *модульным тестированием* (unit testing). Затем нужно перейти к *тестированию взаимодействия* (integration testing), то есть к проверке согласованности классов и методов. Тестирование взаимодействия осуществляется путем последовательного объединения все больших блоков кода и поведения. Такое тестирование должно проводиться уже на ранних этапах разработки (см. главу 21). Завершающий этап — *системное тестирование* (system testing), при котором проверяется все приложение в целом.

17.5.1. Модульное тестирование

Обычно разработчики проверяют свой собственный код и выполняют тестирование взаимодействия, потому что они понимают конкретную логику приложения и знают, где нужно искать ошибки. Модульное тестирование выполняется по тем же принципам, что и до эпохи объектно-ориентированного программирования: разработчики пытаются перебрать все возможные случаи, использовать особые значения аргументов, а также проверяют свои блоки на устойчивость, наличие ошибок счета объектов и др. Если методы и классы достаточно просты, вы легко сможете подготовиться к тестированию модулей.

Можно снабдить объекты и методы средствами для обнаружения ошибок. Вы можете помещать в код какие-либо *утверждения* (assertions): предусловия, постусловия, инварианты. Ошибки нужно стараться обнаруживать вблизи их источника, а не далее по ходу программы, где их результат может быть не столь очевиден.

Мы поддерживаем рекомендации, касающиеся парной работы программистов и активной инспекции кода. При этом мы рекомендуем выполнять формальный обзор программ (см. главу 22), когда разработчики предоставляют свой код коллегам с целью услышать их мнение.

17.5.2. Системное тестирование

В идеале системное тестирование должно выполняться отдельной командой, не зависящей от разработчиков. На эту роль подходит организация контроля качества (quality assurance). Подход к системному тестированию должен быть выработан на основании аналитической модели, построенной по исходным требованиям. Тестовая среда должна готовиться параллельно с разработкой системы. Тогда тестеры не будут отвлекаться на детали разработки и смогут дать независимую оценку приложения, что сократит вероятность пропуска ошибки. После завершения альфа-тестирования клиенты выполняют бета-тестирование, а затем программа готовится к выпуску законченной версии.

Тестовые сценарии на уровне системы определяются вариантами использования из модели взаимодействия. Дополнительные сценарии можно строить на основании вариантов использования или конечных автоматов. Выберите несколько типичных случаев, но не забудьте и про нетипичные ситуации (нулевое количество проходов цикла, максимальное количество проходов цикла, одновременное осуществление событий — если оно допускается моделью и т. д.). Хорошие тестовые сценарии получаются из нетипичных последовательностей смены состояний конечного автомата, потому что при этом проверяются принятые по умолчанию предположения. Не забудьте уделить внимание производительности: нагрузите систему одновременным распределенным доступом нескольких пользователей, если, конечно, это подразумевалось в системных требованиях.

Используйте тестовую систему по максимуму. *Тестовая система* (test suite) полезна для проверки кода после исправления ошибок и обнаружения ошибок, которые появятся в будущих выпусках той же системы. Автоматизация тестирования системы с интерактивным пользовательским интерфейсом может представлять определенную сложность, но в любом случае вы можете задокументировать тестовые сценарии для последующего использования.

Пример с банкоматом. Мы аккуратно подготовили модель банкомата в соответствии с предлагаемой методологией. Поэтому если бы нам нужно было написать само приложение, с тестированием у нас бы проблем не возникло.

17.6. Резюме

Реализация — это последний этап разработки, на котором приходится иметь дело со спецификой языков программирования. В первую очередь нужно заняться вопросами реализации, не зависящими от выбранного языка. Иногда бывает полезно подкорректировать классы, прежде чем приступить к написанию кода, чтобы упростить разработку или повысить производительность. Делать это без веских оснований не следует.

Ассоциация — ключевая концепция, лежащая в основе модели классов UML, однако эта концепция не поддерживается большинством языков программирования непосредственно. Тем не менее в процессе изучения требований нужно сохранять ясность мысли и пользоваться ассоциациями, а на переходе к реализации заменять их более примитивными конструкциями. Существует два основных

способа реализации ассоциаций: при помощи указателей (в одном или двух направлениях) и при помощи отдельных объектов. Объект ассоциации — это пара объектов-словарей (по одному для каждого направления).

Аккуратное моделирование сокращает вероятность ошибок, но не исключает необходимость тестирования. Вы должны выполнить модульное тестирование, тестирование взаимодействия и системное тестирование. В процессе модульного тестирования разработчики проверяют написанные ими самими классы и методы. Тестирование взаимодействия заключается в объединении различных классов и методов. Системное тестирование — это проверка всего приложения в целом на предмет соответствия выявленным на этапе анализа требованиям.

Таблица 17.1. Ключевые понятия главы

объект ассоциации	тестирование системы
словарь	трансформация
моделирование реализации	модульное тестирование
тестирование интеграции	двусторонняя ассоциация
указатель	односторонняя ассоциация

Библиографические замечания

Разделы 17.2 и 17.3 мотивируются концепцией трансформаций. Исчерпывающее рассмотрение трансформаций приводится в работе [Batini-92]. Дополнительные трансформации описываются в [Blaha-96] и [Blaha-98].

Литература

[Batini-92] Carlo Batini, Stefano Ceri, and Shamkant B. Navathe. Conceptual Database Design: An Entity-Relationship Approach. Redwood City, CA: Benjamin Cummings, 1992.

[Blaha-96] Michael Blaha and William Premerlani. A catalog of object model transformations. Third Working Conference on Reverse Engineering, November 1996, Monterey, CA, 87—96.

[Blaha-98] Michael Blaha and William Premerlani. Object-Oriented Modeling and Design for Database Applications. Upper Saddle River, NJ: Prentice Hall, 1998.

[Rumbaugh-87] James E. Rumbaugh. Relations as semantic constructs in an object-oriented language. OOPSLA'87 as ACM SIGPLAN 22, 12 (December 1987), 466—481.

Упражнения

- 17.1. (7) Реализуйте все ассоциации, имеющиеся на рис. 17.6. Используйте односторонние или двусторонние указатели в соответствии с семантикой задачи. Ответ поясните.

- 17.2. (5) Реализуйте все ассоциации, имеющиеся на рис. У12.3. Используйте односторонние или двусторонние указатели в соответствии с семантикой задачи. Ответ поясните.
- 17.3. (4) Реализуйте все ассоциации, имеющиеся на рис. У15.2. Используйте односторонние или двусторонние указатели в соответствии с семантикой задачи. Ответ поясните.
- 17.4. (3) Реализуйте все ассоциации, имеющиеся на рис. У15.3. Используйте односторонние или двусторонние указатели в соответствии с семантикой задачи. Ответ поясните.
- 17.5. (7) Реализуйте все ассоциации, имеющиеся на рис. У12.4. Используйте односторонние или двусторонние указатели в соответствии с семантикой задачи. Должен ли какой-нибудь из полюсов ассоциаций быть упорядоченным? Ответ поясните.

Объектно-ориентированные языки

18

В этой главе рассматривается реализация готового проекта на языках C++ и Java. Именно эти объектно-ориентированные языки распространены в настоящее время наиболее широко. Цель данного этапа: получить код программы. Мы не пытаемся уместить учебник по C++ или Java в одну главу. Мы хотели подчеркнуть особенности этих языков и рассказать о том, каким образом их следует использовать для реализации моделей.

Материалы для этой главы предоставила Крис Келси, за что мы ей очень благодарны.

18.1. Введение

Реализовать объектно-ориентированный проект на объектно-ориентированном языке довольно легко, поскольку языковые конструкции в данном случае близки к проектным. В этой книге мы расскажем о языках C++ и Java, так как они распространены наиболее широко. Даже если вы работаете с другим языком, большинство принципов от этого не изменятся.

У языков C++ и Java много общего. Язык Java моложе C++ и заимствовал довольно большую часть синтаксиса последнего. В обоих языках имеется жесткая типизация: переменные и значения должны быть заранее известного типа (предопределенного или определенного пользователем). Жесткая типизация повышает надежность программ, потому что позволяет обнаруживать несоответствие аргументов при вызове методов и предоставляет дополнительные возможности для оптимизации.

18.1.1. Введение в C++

Язык C++ был разработан Бьерном Страуструпом в Bell Laboratories в 80-х годах XX века. Целью создания нового языка было расширение широко использовавшегося в то время языка C для обеспечения поддержки объектно-ориентирован-

ных концепций. С++ сохранил возможности низкоуровневого программирования, имевшиеся в С и добавил к ним поддержку высокоуровневых объектно-ориентированных концепций.

Изначально язык С++ был реализован в Bell Labs как препроцессор, который преобразовывал С++ в стандартный язык С. Когда С++ обрел широкую популярность, появились компиляторы с символическими отладчиками и другие мощные средства разработки. Комитет ISO/ANSI стандартизировал язык и его библиотеки в 1998 году. Компиляторы С++ для различных операционных систем выпускаются несколькими крупными производителями, а также фондом свободного программного обеспечения (Free Software Foundation — FSF).

Синтаксис языка С++ включает в себя синтаксис С. Новый язык представляет собой гибрид, в котором некоторые сущности имеют объектные типы, а другие сущности принадлежат к традиционным примитивным типам. Язык сохранил особенности, несовместимые с чистым объектно-ориентированным программированием, такие как глобальные функции, не являющиеся методами какого-либо класса. Однако синтаксис и семантика примитивных и объектных типов хорошо согласованы между собой.

С++ поддерживает обобщение и наследование, а также выбор методов во время выполнения (полиморфизм). Классы могут содержать как полиморфные, так и неполиморфные методы. Полиморфный метод обязательно должен быть объявлен в суперклассе как виртуальный — *virtual*. Эффективность реализации достигается благодаря тому, что каждый объект содержит указатель на таблицу методов его класса. В языке отсутствует базовый тип *Object*, характерный для других объектно-ориентированных языков. С++ допускает множественное наследование.

С++ позволяет не только перекрывать (или подменять) методы при их наследовании, но и перегружать их (*overload*): методы и функции могут иметь одинаковые имена, но разное число или тип параметров. При вызове выбирается метод или функция с соответствующим числом и типом параметров. Можно перегружать и операторы языка С++, что делает возможной интуитивно понятную форму записи (например, $a + b$ вместо $a.add(b)$, где a и b принадлежат к типу *ComplexNumber*).

Спецификаторы доступа обеспечивают инкапсуляцию (сокрытие информации), предоставляя возможность ограничения доступа к членам класса (методам и данным). Члены класса могут быть доступны только методам самого класса (*private*), класса и его подклассов (*protected*) или любого класса, метода или функции (*public*). Класс может предоставлять выборочный доступ к своим закрытым членам, объявляя дружественные классы как *friend*. С++ обеспечивает ограничение доступа на уровне классов, а не объектов, поэтому никаких ограничений между объектами одного класса быть не может. Пространства имен (*namespace*) ограничивают область действия символов, но не влияют на доступность видимых сущностей. Пространства имен были введены для того, чтобы обходить конфликты имен между внешними библиотеками.

С++ позволяет программисту самостоятельно управлять памятью, в частности выбирать собственную стратегию выделения памяти. Для объектов, объявленных во время компиляции, память выделяется статически. Приложение

может динамически выделять память из кучи (heap) во время своего выполнения. Для этого служит операция *new*. Динамически созданный объект существует в памяти до тех пор, пока он не будет удален явным вызовом операции *delete*.

Адрес размещения объекта в памяти не изменяется на протяжении всего его существования. Именно адрес в памяти служит уникальным идентификатором объекта. Программист может получить адрес статического объекта при помощи оператора *&* или адрес динамически созданного объекта в виде *ссылки* (handle) в момент создания этого объекта. Чтобы обратиться к объекту, на который указывает некоторый указатель, данный указатель *разыменовывается* (dereference) при помощи оператора ***. Для любого объекта *O*, расположенного в памяти по адресу *A*, выполняются следующие равенства: $\&O == A$, $*A == O$. Как и в языке C, программист не защищен от ошибок, связанных с неправильным обращением к памяти, ни на этапе компиляции, ни во время выполнения программы. C++ не предоставляет средств для обнаружения таких ошибок во время выполнения программы. Ошибки обычно приводят к непредсказуемым результатам.

Помимо указателей, в C++ поддерживаются *ссылки* (reference), которые с синтаксической точки зрения выступают в роли альтернативных имен объектов. Ссылки должны связываться с существующими объектами в момент своего создания. Ссылка не может быть связана с другим объектом, а потому не может быть и нулевой. В остальном ссылки ведут себя как разыменованные указатели.

Во всех классах имеются конструкторы и деструкторы, которые автоматически вызываются в момент создания и удаления объектов класса. Они обычно используются для инициализации и выполнения действий, которые обязательно должны производиться при удалении объекта (например, освобождения памяти).

Таким образом, C++ представляет собой гибкий язык программирования, характеризующийся ориентированностью на повышенную эффективность выполнения программ, возможностью образования иерархий, а также однородной семантикой. Точность выражений достигается в нем благодаря отказу от простоты.

18.1.2. Введение в Java

Язык Java появился в начале 90-х годов XX века в виде побочного продукта исследований, проводимых корпорацией Sun Microsystems. Эти исследования касались программирования пользовательских устройств. Для данного проекта требовался переносимый и независимый от устройств язык. В 1995 году интерпретатор языка был выложен на веб-сайт (в то время веб-сайты были редкостью), и вскоре возможность выполнения Java-программ с некоторыми ограничениями была добавлена в веб-браузеры, благодаря чему разработчики веб-страниц получили возможность создавать динамическое и интерактивное содержимое. За несколько лет бурного развития Интернета и приложений распределенной архитектуры язык Java стал основным средством для коммерческих разработчиков. Коммерческие, условно-бесплатные и бесплатные средства разработки для Java распространены очень широко. Язык продолжает развиваться, библиотеки для него растут. Корпорация Sun контролирует версии языка и бесплатно предоставляет основные средства для работы с ним для большинства платформ.

Популярность Java обеспечена не столько характеристиками самого языка, сколько его переносимостью и наличием огромного количества библиотек классов. Исходный код на Java компилируется в промежуточный байт-код, который, в свою очередь, выполняется виртуальной машиной Java Virtual Machine (JVM). Виртуальные машины были разработаны почти для всех операционных систем. Имеются также собственные компиляторы Java (также для разных систем).

Подобно тому, как успех C++ определялся наличием большого количества программистов, владевших C, успех Java основывался на тех, кто владел C++ и C. Синтаксис Java во многом подобен синтаксису C++, однако модели работы с объектами и памятью в этих языках существенно отличаются.

Java относится к числу языков с жесткой типизацией, с разделением примитивных и объектных типов. Работа с этими типами осуществляется по-разному: память под примитивы выделяется статически, они интерпретируются как переменные со значениями (операция $i = j$ присваивает переменной i значение переменной j), тогда как объекты выделяются динамически во время выполнения, а работа с ними осуществляется только посредством переменных-ссылок (операция $i = j$ создает альтернативную ссылку i для объекта, на который ссылается j). В Java не предусмотрен синтаксис для описания ссылок на примитивы, но имеются средства для преобразования между примитивами и соответствующими объектными типами, например между *int* и *Integer*. В версии Java 1.5 была добавлена концепция *автоматической упаковки* (autoboxing) — это автоматическое преобразование между примитивами и их объектными обертками в типичных ситуациях, например, при передаче параметров или при использовании совокупностей.

Все объектные типы Java происходят от общего предка *Object*, поэтому на высшем уровне абстрагирования все они совместимы между собой. Проверка типов во время выполнения приводит к передаче сообщения об исключительной ситуации при некорректном преобразовании объектов (например, когда объект должен быть преобразован к типу своего подкласса). Полиморфизм осуществляется автоматически. Программист может запретить его, только если он явным образом запретит определение перекрывающих методов. Java не поддерживает множественное наследование, но позволяет эмулировать его при помощи *интерфейсов* (interfaces) — абстрактных спецификаций классов, которые содержат только константы и объявления методов.

Пакеты (packages) помогают упорядочивать классы и задают область действия идентификаторов. Каждый компилируемый блок исходного кода (файл) может содержать не более одного открытого класса, причем название файла должно совпадать с именем этого класса. В том же файле могут быть определены вспомогательные классы. В самом начале файла объявляется пакет, к которому он относится. Для обращения к открытым элементам других пакетов используется директива *import* (например, *import java.io.** — включение доступа к классам пакета ввода-вывода).

Подобно спецификаторам C++, модификаторы доступа Java позволяют ограничить доступность атрибутов и методов. По умолчанию все атрибуты и методы имеют область видимости *package* и могут быть использованы всеми остальными

методами, определенными в том же пакете. Явным образом можно указать область видимости *private* (доступ внутри класса), *public* (открытый доступ) и *protected* (доступ для класса и его подклассов, определенных в других пакетах). Таким образом, термин *protected* в Java имеет несколько иной смысл, чем в C++.

Управление памятью осуществляется только виртуальной машиной. Весь код на Java существует в контексте класса. Система загружает и выгружает классы по мере необходимости и перемещает код в памяти во время его выполнения. Программисту не нужно знать, где именно находятся объекты; он обращается к ним по соответствующим ссылкам, которые с синтаксической точки зрения могут рассматриваться как переменные объектных типов.

Освобождение памяти производится в процессе *сборки мусора* (garbage collection). Когда все ссылки на объект удаляются, система может вернуть память, занимаемую объектом, в пул. Программист может порекомендовать системе уничтожить объект, но он не может инициировать сборку мусора в конкретный момент времени. Java обеспечивает управление памятью во время выполнения программы (контроль выхода за границы массива или попыток использования нулевых ссылок), и в случае обнаружения соответствующих ошибок передаются сообщения об исключительной ситуации.

Таким образом, язык Java ориентирован на переносимость и безопасность кода, что достигается ценой частичной потери эффективности и гибкости. Для эмуляции множественного наследования используются абстрактные интерфейсы. Для этого языка разработаны обширные библиотеки объектов, обеспечивающих реализацию типичных программ и служащих для интеграции распределенных платформ на системном уровне.

18.1.3. Сравнение C++ и Java

Характеристики обоих языков сравниваются в табл. 18.1.

Таблица 18.1. C++ и Java

	C++	Java
Управление памятью	Доступно программисту. Объект располагается по фиксированному адресу	Управляется системой. Объекты перемещаются в памяти во время выполнения
Модель наследования	Одиночное и множественное наследование. Явный полиморфизм методов. Нет универсального класса-предка. Допускает смешивание иерархий	Одиночное наследование с абстрактными интерфейсами. Автоматический полиморфизм. Универсальный предок Object
Управление доступом и защита объектов	Гибкая модель с защитой при помощи спецификатора const	Нечеткая модель вызывает слабую инкапсуляцию
Семантика типов	Согласована для примитивных и объектных типов	Разная для примитивных и объектных типов

	C++	Java
Организация программы	Функции и данные могут существовать вне классов. Область действия может быть глобальной или определяться пространством имен	Все функции и данные существуют внутри классов. Областью действия может быть пакет
Библиотеки	Главным образом обеспечивающие низкоуровневую функциональность. Богатая библиотека шаблонов контейнеров (структур данных) и алгоритмов	Обширные. Имеются классы сервисов высокого уровня и классы, служащие для интеграции систем
Обнаружение ошибок во время выполнения	Должно быть предусмотрено программистом. Ошибки приводят к непредсказуемому поведению программы	Осуществляется системой. Ошибки приводят к завершению компиляции или выполнения
Переносимость	Исходный код требует компиляции под нужную платформу. Исполняемый код выполняется непосредственно процессором	Скомпилированные в байт-код классы могут быть выполнены на любой платформе, где установлена виртуальная машина. Необходимо наличие виртуальной машины
Эффективность	Отличная	Хорошая. Зависит от реализации виртуальной машины

18.2. Сокращенная модель банкомата

На рис. 18.1 показана часть модели классов банкомата, которую мы рассмотрим в качестве примера реализации. Мы добавили классы *CheckingAccount* (СчетДоВостребования) и *SavingsAccount* (СберегательныйСчет), чтобы иметь возможность обсуждать обобщения.

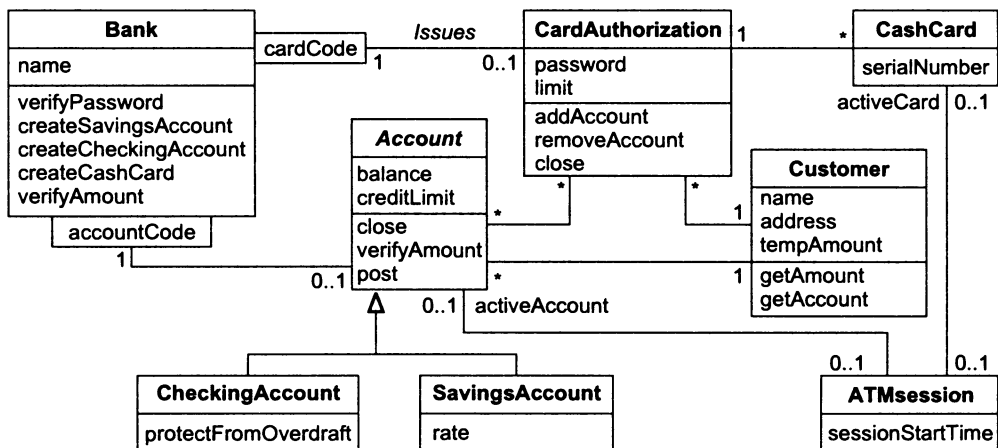


Рис. 18.1. Сокращенная модель классов банкомата, используемая в этой главе

18.3. Реализация структуры

Первый этап реализации объектно-ориентированного проекта — это реализация структуры, определяемой моделью классов. Вы должны выполнить следующие действия.

1. Реализовать типы данных (раздел 18.3.1).
2. Реализовать классы (раздел 18.3.2).
3. Реализовать управление доступом (раздел 18.3.3).
4. Реализовать обобщения (раздел 18.3.4).
5. Реализовать ассоциации (раздел 18.3.5).

18.3.1. Типы данных

Если вы еще не определили типы атрибутов, сейчас самое время сделать это. Некоторые типы данных требуют особого рассмотрения.

Примитивы

Простые значения могут быть числами с плавающей точкой, целыми числами, символами и логическими значениями. Старайтесь использовать числовые значения вместо строковых. Числовые типы более эффективны с точки зрения хранения, обработки и простоты обеспечения цельности атрибутов (см. ниже о перечислениях).

Объектные типы

Объекты могут использоваться для объединения и упорядочения атрибутов. C++ поддерживает физическую вложенность объектов: экземпляр одного объекта создается внутри пространства памяти другого объекта. Java поддерживает только ссылки на объекты, поэтому объект Java внутри другого объекта — это аналог указателя C++, который должен быть явным образом связан с конкретным объектом.

Структуры C++ технически отличаются от классов тем, что их члены по умолчанию открыты для доступа извне. Их удобно использовать для логического объединения данных без методов. Можно добавить конструктор, который будет инициализировать данные. Объект, в котором будет находиться структура, может обеспечивать сокрытие данных этой структуры.

```
struct address {
    string street;
    string city;
    string state;
    address() : street(""), city(""), state("") { }
};

class Customer {
    string name;
    address addr; //атрибут объекта
    float tempAmount;

public:
    // ...
    string City() {return addr.city; } //этот метод следует вызывать извне
```

Аналогичную стратегию можно использовать и в языке Java, однако необходимо обеспечить явное создание экземпляра объекта-атрибута внутри конструктора или инициализатора внешнего объекта. Доступ внутри пакета подразумевает доступ к объектам-атрибутам внутри классов.

```
class Address {
    String street = "";
    String city = "";
    String state = "";
}

public class Customer {
    private String name;
    //явное создание объекта-атрибута:
    private address addr = new Address(); //объект-атрибут
    private float tempAmount;
    // ...
    String city() { return addr.city; } //этот метод следует вызвать извне
}
```

Ссылочные типы

Для реализации ассоциаций можно использовать ссылки на объекты в Java и указатели и ссылки в C++. В Java все переменные-классы представляют собой ссылки на объекты, тогда как в C++ переменная может быть самим объектом. C++ требует особого внимания, чтобы не перепутать объект с ссылкой на него.

Идентификаторы объектов

В объектно-ориентированных языках предусмотрены встроенные механизмы идентификации объектов и средства для проверки индивидуальности объектов. Обычно необходимости создавать явный тип идентификаторов объектов не возникает. Если вам нужен уникальный идентификатор объекта, вы можете получить его от системы во время выполнения программы.

В C++ уникальным идентификатором объекта является адрес объекта в физической памяти, который можно получить применением оператора `&` к самому объекту или ссылке на него. Индивидуальность объекта можно проверить сравнением указателей или адресов. В Java оператор `==` позволяет сравнивать индивидуальности объектов во время выполнения. Если нужен идентификатор объекта, можно воспользоваться методами `hashCode()` и `toString()` универсального суперкласса *Object*, которые возвращают идентификатор объекта в целочисленном и строчном форматах. Адреса памяти в Java недоступны, однако в большинстве систем эти методы реализованы на внутреннем размещении объекта.

Не путайте уникальный идентификатор предметной области (например, номер банковского счета или индивидуальный номер налогоплательщика) с идентификатором объекта. Идентификатор предметной области описывает свойство, относящееся к этой области, тогда как идентификатор объекта — это атрибут, относящийся к системе.

Перечисления

Перечисления (перечислимые типы — *enumerations*) обладают двумя преимуществами: ограниченным множеством значений и их символьным представлением.

Например, множество значений масти игральной карты можно описать перечислением *CLUB, DIAMOND, HEART, SPADE* (ТРЕФА, БУБНА, ЧЕРВА, ПИКА).

Язык C++ позволяет реализовать перечисления непосредственно. Каждое перечисление определяется как отдельный тип, который можно указывать в методах и операторах. Если явные значения элементов не указываются, им присваиваются последовательные целочисленные значения, начиная с 0.

```
enum Card = { CLUB, DIAMOND, HEART, SPADE };
```

Компилятор выполняет неявное преобразование от перечислимого типа к целочисленному, но не в обратную сторону. C++ гарантирует, что размер объектов перечислимого типа достаточен для размещения в памяти максимального значения. Например, переменная, принимающая значения *{TRUE, FALSE}*, может иметь размер в один бит. Перечисления могут быть определены глобально или входить в какие-либо классы.

На практике использование перечислений C++ может быть затруднено сложностью преобразования и необходимостью переопределения операций, работающих с целочисленными аргументами. Лучше всего использовать перечисления для повышения ясности кода, что достигается использованием символьных констант. Скрытые значения атрибутов могут храниться в виде целочисленных переменных, тогда как в интерфейсе объекта можно использовать константы для ограничения значений параметров, а в реализации методов те же константы будут служить граничными значениями.

```
class Car {
public: // перечисление сделаем открытым, чтобы клиенты могли его
    использовать
    enum direction {N,E,S,W};
private:
    int mph; // скорость автомобиля
    int nesw; // направление движения
public:
    // требуется передача параметра типа enum
    Car(int speed, direction dir) : mph(speed), nesw(dir) {}
    // int позволяет использовать ++ и - без перепрыжки для типа enum
    Car& TurnRight() {if (++nesw > W) nesw = N; return *this;}
    Car& TurnLeft() {if (--nesw < N) nesw = W; return *this;}
    // ...
};

int main()
{
    Car (15,Car::E); // клиент использует значение из открытого
    перечисления
    // ...
}
```

Версии Java вплоть до 1.5 не поддерживали понятие перечислимого типа. Перечисления Java аналогичны перечислениям в C++. В отсутствие перечислений для группировки и совместного использования констант можно использовать интерфейсы (см. раздел 18.3.4). Явные модификаторы *public*, *static* и *final* не

обязательны, потому что по умолчанию они действуют на все поля типа *int*, определенные в интерфейсе. Обратите внимание, что интерфейс *Enumeration* из библиотеки классов Java уже отменен, как устаревший. Он предоставлял рудиментарные операции для итерирования и не имеет отношения к перечислимым типам.

```
public interface Card {
    public static final int CLUB = 0;
    public static final int DIAMOND = 1;
    public static final int HEART = 2;
    public static final int SPADE = 3;
    . . .
}
```

18.3.2. Классы

Объектно-ориентированные языки программирования позволяют реализовывать объекты напрямую, непосредственно. Вы должны объявить все атрибуты и методы модели классов внутри соответствующих классов C++ или Java. Кроме того, вам придется добавить атрибуты и методы для реализации ассоциаций (см. раздел 18.3.5). Разумно переносить в программу названия из проектной модели.

Вы можете рассматривать объекты как сущности, предоставляющие сервисы запрашивающим их клиентским объектам. Сервисы — это открытые методы класса. Методы описывают протокол запроса сервисов. Параметры методов — это информация, необходимая объекту для предоставления сервиса, а возвращаемое значение — ответ объекта на клиентский запрос.

Начинайте с общедоступного интерфейса класса, а потом добавляйте внутренние методы и атрибуты, обеспечивающие поддержку открытых методов.

Поскольку открытые методы документируют поведение класса, они обычно приводятся в начале объявления этого класса. Впрочем, это не обязательное условие. И в C++, и в Java разрешение символов осуществляется только после полного считывания определения класса, поэтому методы могут ссылаться друг на друга в любом порядке.

18.3.3. Управление доступом

Для сервисов, которые могут запрашиваться или вызываться клиентами, в классе должны быть определены соответствующие открытые методы. Не подлежащие изменению данные тоже можно объявить как открытые. Все остальные члены класса (атрибуты и методы, служащие для внутренней реализации внешней функциональности) должны быть сделаны невидимыми и недоступными для других объектов и функций.

Для управления доступом клиентов к методам и данным в C++ и в Java используются *спецификаторы доступа* (access specifiers или access modifiers). Основные спецификаторы, применимые к атрибутам и методам, одинаковы в обоих языках. Только методы самого класса могут обращаться к атрибутам и методам, имеющим спецификатор *private* (закрытый). Любой клиент может работать с членами класса, имеющими спецификатор *public* (открытый).

Управление доступом в Java

Жесткая инкапсуляция в Java требует внимательного отношения к пакетам и к спецификаторам доступа. Пакеты Java ограничивают область привилегированного доступа. Название пакета объявляется в самом начале файла исходного кода; таким образом, к одному пакету может быть отнесено несколько файлов. Если метод или атрибут не имеет явного спецификатора *private*, другие методы и классы, определенные в том же пакете, могут свободно обращаться к нему. Если название пакета не указано, класс считается принадлежащим глобальному пакету, и все его члены, за исключением закрытых, доступны всем остальным классам, не принадлежащим каким-либо пакетам.

Мы рекомендуем не помещать классы в глобальный пакет. Более того, все атрибуты и методы, не являющиеся открытыми, следует делать закрытыми, то есть объявлять их как *private*, потому что пакетная область видимости не обеспечивает жесткой инкапсуляции. Доступ к атрибутам следует контролировать при помощи методов-оберток.

В языке Java имеется возможность управления доступом не только к членам классов, но и к самим классам. Класс должен быть объявлен как *public*, чтобы его открытые методы были доступны клиентам, находящимся вне его пакета. См. раздел «Видимость и доступ к классам».

Управление доступом в C++

Спецификаторы доступа C++ действуют на целые секции объявления класса. Все члены класса считаются закрытыми, пока компилятор не встретит явный спецификатор доступа. Спецификаторы могут присутствовать в любом месте объявления класса, и их действие распространяется до следующего спецификатора. Все атрибуты и методы, не относящиеся к интерфейсу класса, опять-таки, следует сделать закрытыми. Ослаблять ограничения доступа нужно только при наличии достаточно веских оснований. Управление доступом в структурах C++ осуществляется по тем же принципам, но с важным отличием: все члены структуры по умолчанию считаются открытыми (см. раздел 18.3.1).

Язык C++ позволяет открывать выборочный доступ к закрытым членам класса при помощи объявления *friend*. Класс может предоставить доступ к функции, методу или классу, но при этом он открывает доступ ко всем своим закрытым членам. Поэтому объявление дружественности следует использовать с осторожностью.

Видимость и доступ к классам

Класс должен быть виден своим потенциальным клиентам, иначе они не смогут распознать его тип и не получат доступа к его открытым членам. В C++ и Java минимальным компилируемым блоком является файл. Классы в одном файле видны друг другу, но на практике рекомендуется не помещать в один файл более одного класса (возможно, со вспомогательными классами). Классы из внешних модулей необходимо явным образом сделать видимыми компилируемому классу.

В Java видимость и доступ определяются структурой пакетов. Классы, находящиеся в разных файлах одного пакета, считаются видимыми друг другу. Пакеты

связаны со структурой файловой системы на диске: все файлы пакета должны находиться в каталоге, имя которого совпадает с именем этого пакета. В процессе компиляции и выполнения программы Java использует переменную окружения `CLASSPATH`, задающую отправную точку для поиска пакетов внутри файловой системы.

Пакет может содержать множество классов, но в каждом файле исходного кода Java может быть только один класс с видимостью *public*, и имя этого класса должно совпадать с именем файла. Внешние по отношению к пакету классы могут использовать только открытые методы открытых классов. Чтобы открытый класс был виден классу из другого пакета, нужно использовать директиву импортирования: *import packagename.classname* или *import packagename.**. Система осуществляет поиск необходимых классов все по тому же маршруту `CLASSPATH`.

Пример с банкоматом. Каждый класс находится в отдельном файле. Например, файл *Bank.java* будет содержать следующие строки:

```
package bankInfo
public class Bank { . . . }
```

Файл *Customer.java* будет содержать текст:

```
package bankInfo
public class Customer { . . . }
```

Файлы *Bank.java*, *Customer.java* и другие классы пакета *bankInfo* должны находиться на диске в папке *bankInfo*. Чтобы классы *Bank* и *Customer* были доступны классу *ATMsession*, который не входит в пакет *bankInfo*, в файл этого класса необходимо включить строки:

```
import bankInfo.*
public class ATMsession { . . . }
```

Директива *import* делает видимыми и доступными открытые методы открытых классов пакета *bankInfo*.

Строгих правил именования пакетов нет, но это не значит, что вам следует использовать глобальный пакет. Не используя классы, вы теряете управление видимостью и доступом. Закрытые атрибуты все равно остаются закрытыми, однако методы, предназначенные для доступа к этим атрибутам, становятся фактически открытыми, а кроме того, открывается доступ к вспомогательным классам, предназначенным для ограниченного использования.

C++ не накладывает ограничения на размещение файлов исходного кода или скомпилированных модулей. Программы C++ делятся на заголовочные файлы, в которых содержатся объявления (в том числе и классов), и файлы реализации, в которых содержится код всех методов, за исключением наиболее простых (которые можно реализовать прямо внутри объявления класса). Для введения символов из одного файла в другой используется директива *#include <filename>* (в случае, если файл находится в папке, предназначенной для хранения таких файлов) или *#include "filename"* (в кавычках указывается полное имя файла). Эта директива ставится в самом начале файла исходного кода. Фактически, эта директива осуществляет подстановку символов из соответствующего заголовочного файла в компилируемый файл исходного кода.

В C++ имеется концепция пространства имен (namespace), которая позволяет ограничивать область действия наименований программных сущностей, однако не ограничивает доступ к ним. Класс помещается в пространство имен, если его объявление находится внутри этого пространства, и полным именем такого класса становится *namespace::classname*. Пространство имен может охватывать несколько файлов исходного кода, и в одном файле может быть определено несколько пространств имен. Символы одного пространства имен могут быть введены в другое при помощи директивы *using*. Стандартные имена библиотеки C++ находятся в пространстве имен *std*.

Если пространство имен не указано, символы считаются глобальными. Поскольку пространства имен ограничивают видимость, но не доступ, они используются главным образом для предотвращения совпадения имен при работе с несколькими библиотеками.

Управление доступом в иерархиях наследования

В обоих описываемых нами языках имеется спецификатор доступа *protected*, но работает он по-разному. В C++ защищенный член класса доступен только методам класса и его подклассов.

В Java защищенные атрибуты и методы имеют пакетный уровень доступа, который расширяется включением методов подклассов, определенных вне пакета.

В Java отсутствует эквивалент защищенному уровню доступа из C++.

Применение управления доступом

В модели UML могут присутствовать обозначения *{public}*, *{private}*, *{protected}*, *{package}* или +, -, #, ~, определяющие уровень доступа к членам класса (который в UML называется *видимостью* — visibility). Отсутствие явных указаний подразумевает, что метод принадлежит к общедоступному интерфейсу класса. В реализации модели необходимо использовать спецификаторы доступа, а вспомогательные атрибуты следует делать закрытыми.

18.3.4. Обобщение

Объектно-ориентированные языки позволяют реализовывать обобщение через наследование. Класс может являться предком-суперклассом для одного или нескольких потомков-подклассов. Подкласс наследует все элементы предка, к которым он может добавлять собственные атрибуты и методы. Подкласс может переопределять методы суперкласса, что позволяет ему по-своему выражать поведение метода с тем же именем и сигатурой (параметрами и возвращаемым значением). Подклассы, в свою очередь, могут выступать в качестве суперклассов по отношению к своим потомкам. В результате получается иерархия классов, на которой каждый уровень выражает расширенное или конкретное поведение.

Механизм наследования не только делает определение подклассов более удобным, он позволяет более абстрактным образом работать с объектами. Наследование делает возможным полиморфизм, благодаря которому тип потомка может выступать в качестве представителя типа предка. В цельной иерархии каждый

подкласс должен уметь реализовывать поведение всех своих суперклассов так, чтобы он в любой ситуации мог осуществить ожидаемое от суперкласса поведение [Liskov-88].

Не обязательно определять все классы иерархии наследования полностью. Абстракции высокого уровня достаточно просто объявить, а реализацию деталей отложить до подклассов. В результате получатся абстрактные классы — описанные, но не реализованные полностью. Абстрактные классы описывают типы на уровне, не обладающем достаточными сведениями для реализации конкретного поведения. Например, любую фигуру можно нарисовать, но пока ее тип неизвестен, отсутствуют сведения о том, как это сделать.

Поскольку абстрактные типы реализованы не полностью, вы не можете создавать объекты этих типов. Вместо этого вы должны создавать конкретные объекты, принадлежащие к типу одного из конкретных подклассов, который должен полностью реализовывать определенное в суперклассе поведение (и свое собственное), и работать с ссылками на эти объекты, имеющими тип суперкласса. На объекты подкласса в этом случае можно ссылаться как на объекты более общего типа. Например, *Square* или *Circle* — конкретные подклассы абстрактного класса *Shape*. Можно работать с ними через переменную-ссылку типа *Shape*. Хотя обе ссылки будут иметь тип *Shape*, при вызове метода *draw()* будет выполняться соответствующее поведение. Приведенная ниже программа на языке Java приведет к построению разных форм.

```
Shape s1 = new Circle();
Shape s2 = new Square();
s1.draw(); s2.draw();
```

Языки C++ и Java реализуют разные модели наследования.

Наследование в Java

Все объекты Java имеют общего предка — класс *Object*. Он содержит минимальный набор методов и полей, необходимых для поддержки концепций программирования, таких как индивидуальность, эквивалентность и параллельность объектов. Наличие общего предка позволяет работать с объектами на максимально общем уровне: например, формировать совокупности различных, не связанных между собой типов. Использование объектов на уровне «бестиповой» абстракции позволяет обойти некоторые ограничения, накладываемые языками с жесткой типизацией.

Наследование реализуется посредством ключевого слова *extends*:

```
class Account {
    private float balance;
    public void Post(float amount) { . . . }
    public float Balance() { return balance; }
}
class SavingsAccount extends Account {
    private float rate; // добавление атрибута — процентной ставки
    float CalcInterest() {
        // расчет причитающихся процентов
    }
}
```

Класс языка Java может быть потомком только одного класса. Однако некоторые преимущества множественного наследования можно реализовать посредством интерфейсов. *Интерфейс* (interface) — это класс, определение которого не содержит реализации. Оно содержит объявления методов (без реализации) и может содержать неизменяемые поля, относящиеся к классу в целом (а не к отдельному объекту), то есть статические. Интерфейсы, как и классы, могут иметь потомков — субинтерфейсы.

Чтобы воспользоваться интерфейсом, класс должен объявить, что он реализует данный интерфейс, и предоставить код для всех методов интерфейса. Класс может расширять только один класс (то есть быть его потомком), но может реализовывать произвольное количество интерфейсов, что позволяет имитировать множественное наследование. Реализующий множество интерфейсов объект можно рассматривать как имеющий любой из типов своих интерфейсов. В нашем примере мы абстрагируем счета, на которые начисляются проценты, в отдельный интерфейс, который будет реализовываться классом *SavingsAccount*. Это гарантирует однородность методов во всех классах, реализующих данный интерфейс, как бы они ни назывались: *SavingsAccount*, *InterestBearingCheckingAccount* или как-либо еще.

```
interface InterestBearingAcct {
    float CalcInterest();
}
class SavingsAccount extends Account
    implements InterestBearingAcct {
    private float rate;
    public float CalcInterest() {
        // реализация вычисления процентов
    }
}
```

Теперь *SavingsAccount* может выступать в программе как объект типа *Account*, *SavingsAccount* или *InterestBearingAcct*.

Помимо конкретных классов и интерфейсов, в Java допустимы и абстрактные классы — не полностью реализованные классы, которые не могут иметь экземпляров, но могут быть предками конкретных подтипов. И сам класс, и его нереализованные методы должны иметь спецификатор *abstract*.

```
public abstract class AbstractExample {
    void method1() { /* . . . */ }
    abstract void method2 ();
}
```

Открытый класс может служить суперклассом подклассу в другом пакете, поскольку он доступен любому клиенту, импортировавшему пакет, в котором такой класс находится. Подклассы, определенные вне пакетов своих суперклассов, могут обращаться не только к открытым, но и к защищенным членам своих суперклассов. Закрытые члены суперклассов подклассам не видны (см. раздел 18.3.3).

Чтобы предотвратить порождение подклассов, класс следует объявить как *final*. Квалификатор *final*, примененный к методу, запрещает его перекрытие.

Наследование в C++

Подклассы C++ не имеют общего предка. Иерархия классов может начинаться с произвольного класса. Суперкласс называется *базовым классом* (base), а подкласс — *порожденным* (derived) *классом*. *Непосредственный базовый класс* (direct base) — это прямой предок порожденного класса, а *косвенный базовый класс* (indirect base) — более удаленный предок.

В C++ в отличие от Java полиморфизм не является автоматическим. Класс может смешивать методы, разрешение которых осуществляется автоматически во время выполнения программы, со всеми прочими методами. Чтобы разрешить полиморфизм, метод нужно объявить как *virtual*. Все прочие методы будут вызываться в соответствии с типом ссылки на объект, а не в соответствии с фактическим типом порожденного класса, даже если в порожденном классе метод перекрывается.

```
class Hello { // . . .
    public:
        void method1() { cout << "hello\n"; }
        virtual void method2() { cout << "hello\n"; }
};
class Goodbye : public Hello { // . . .
    public:
        void method1() { cout << "goodbye\n"; }
        void meothd2() {cout << "goodbye\n";}
};
int main() {
    Goodbye g;
    Hello& h = g; // тот же объект, но ссылка базового типа

    g.method1(); g.method2(); h.method1(); h.method2();
}
```

В результате на экран будет выведен следующий текст:

```
goodbye // method1 не виртуальный, вызывается из подкласса
goodbye // method2 виртуальный, вызывается из подкласса
hello    // method1 не виртуальный, вызывается из базового класса
goodbye  // method2 виртуальный, вызывается из базового класса
```

C++ не позволяет запретить перекрытие методов, однако в иерархии, где часть методов перекрывается, а часть — нет, отсутствие модификатора *virtual* может указывать на желание автора сохранить базовый метод в неизменности.

Спецификация доступа в C++ включена в синтаксис описания наследования. Порожденный класс может иметь тип доступа *public*, *protected* или *private*. Открытое наследование подразумевает, что все открытые методы базового класса остаются открытыми и в порожденном классе.

```
class SavingsAccount : public Account { . . . }
```

При закрытом наследовании все методы базового класса становятся закрытыми. Такое наследование подразумевает, что базовый класс используется для упрощения реализации порожденного класса, но при этом порожденный класс не

является частным случаем базового класса. На практике лучше инкапсулировать базовый тип в качестве члена порожденного класса (с использованием делегирования — см. раздел 15.9.3). Защищенное наследование подразумевает, что открытые методы базового класса доступны только потомкам данного класса. Реально это используется редко.

Абстрактные классы получаются при включении по крайней мере одного чисто виртуального метода, при объявлении которого используется синтаксис инициализации нулем: *void fn() = 0*. Такие классы не могут иметь экземпляров, а порожденные от них классы являются абстрактными до тех пор, пока не реализуют все унаследованные чисто виртуальные методы.

C++ поддерживает множественное наследование. На практике оптимальная комбинация родительских классов представляет собой смесь конкретных типов и абстрактных классов, описывающих требования к поведению. Последние являются чем-то вроде интерфейсов Java, благодаря чему вы получаете возможность определять основные реализации методов там, где это удобно.

```
class Account { // не абстрактный
public:
    /* предположим, что реализация метода Post может быть разной
    для разных типов счетов. Базовая реализация - предлагаемый
    по умолчанию вариант */
    virtual void Post(float amount) { . . . }
    float Balance() { return balance; }
private:
    float balance;
};

class InterestBearingAcct { // абстрактный класс
public:
    virtual float CalcInterest() = 0; // чисто виртуальный метод
    float Rate() { return rate; }
private:
    float rate;
};

class SavingsAccount : public Account,
    public InterestBearingAcct {
public:
    virtual float CalcInterest() {
        // расчет причитающихся процентов
    }
};
```

18.3.5. Ассоциации

Существующие объектно-ориентированные языки не предоставляют непосредственной поддержки концепции ассоциаций. Однако они позволяют с легкостью создавать связи в виде ссылок на объекты или отдельных объектов-ассоциаций (см. раздел 17.4). Ассоциацию следует сделать классом, если имеются атрибуты, характеризующие саму ассоциацию как нечто отдельное. Мы рекомендуем превращать в классы n-арные и квалифицированные ассоциации.

Односторонние ассоциации

Односторонние ассоциации сокращают взаимные зависимости между классами. Когда один класс ссылается на другой, последний должен быть виден и доступен исходному. Ссылающийся класс должен поддерживать ассоциацию своими силами. Обычно он использует интерфейс того класса, на который ссылается. Насколько возможно сократить зависимости, настолько облегчается обслуживание системы и повышается возможность повторного использования.

Например, если бы нам не нужно было получать совокупность сотрудников компании на рис. 17.7, достаточно было бы указателя из класса *Person* на класс *Company*. В языке Java класс *Person* может просто содержать поле типа *Company*.

```
public class Company { . . . }
public class Person {
    private Company employer;
    . . .
}
```

Язык C++ позволяет реализовать атрибуты-ссылки двумя способами. Чаще всего используется указатель, что позволяет изменять ссылку. Если предположить, что человек может менять работодателей, мы получим следующий код на C++. (Этот код допускает отсутствие работодателя — значение *null*, — что не согласуется с кратностью, указанной на рис. 17.7. Допустимые значения кратности должны обеспечиваться методами, изменяющими значение данного аргумента.)

```
class Company { . . . }
class Person {
    Company* employer;
    . . .
}
```

Ссылка на класс подразумевает постоянную связь, в результате которой содержащий эту связь объект зависит от объекта-атрибута. Ссылки должны быть связаны в момент инициализации. Они не могут быть нулевыми и не могут изменять свое значение. Таким образом, в данном случае язык гарантирует наличие объекта ссылки. Если предположить, что счет (*Account*) создается в конкретном банке (*Bank*) и не может быть передан в другой банк и никакой счет не может существовать без банка, получится следующий код:

```
class Bank { . . . }
class Account {
    Bank& bank;
    . . .
}
```

Это требование языка C++ делает необходимым существование экземпляра класса *Bank* перед порождением экземпляра класса *Account* (см. раздел 18.4.1). Частичное ограничение этого рода может быть получено в Java, если объявить атрибут объектного типа как *final*, что гарантирует невозможность другого объекта присваивания переменной-ссылке. Тем не менее требование существования таким образом закодировано быть не может.

При использовании атрибутов-ссылок для реализации ассоциаций следует быть осторожным, чтобы не допустить непредвиденного изменения участвующих в ассоциации объектов. Объект, содержащий атрибут-ссылку, может послужить «черным ходом» к другому объекту, на который он ссылается. Особенно важно следить за инкапсуляцией атрибутов-ссылок, которые должны изменяться и возвращаться только осмысленными и безопасными методами. С++ позволяет повысить уровень защищенности при помощи квалификатора *const*, который можно добавлять в тех случаях, когда изменение объекта в контексте ассоциации не предвидится. Java не позволяет отличать неизменные объекты от изменяющихся.

Двусторонние ассоциации

Двусторонние ассоциации требуют приложения усилий на обоих полюсах связи. Ассоциацию типа «один-к-одному» можно реализовать посредством взаимных ссылок двух объектов или посредством отдельного объекта-ассоциации. Ассоциация типа «один-ко-многим» требует одной ссылки на одном полюсе и совокупности ссылок на другом полюсе (или объекта-ассоциации). Например, клиент может иметь несколько счетов, и мы хотим иметь возможность проследивать соответствующую ассоциацию в обоих направлениях. И в Java, и в С++ имеются типы объектных совокупностей, которые можно использовать для реализации связи на полюсе с кратностью «многое».

```
public class Account {
    private Customer customer; // полюс с кратностью 1
    . . .
}

import java.util.* // для обращения к классу HashSet
public class Customer {
    // совокупность ссылок на счета
    private HashSet accounts = new HashSet();
    . . .
}
```

Классы-ассоциации

Класс-ассоциация позволяет повысить независимость объектов благодаря устранению прямых ссылок на связанные классы, однако при этом падает эффективность работы системы и повышается сложность реализации ассоциаций.

Объект ассоциации представляет собой множество кортежей, каждый из которых содержит одно значение из каждого из участвующих в ассоциации классов. Объект бинарной ассоциации может быть реализован в виде двух объектов-словарей, каждый из которых осуществляет отображение в одном направлении. В библиотеках Java и С++ предусмотрена поддержка отображения объектов. Чтобы инкапсулировать связи и сделать работу с ними интуитивно понятной, отображение можно упаковать во вспомогательный класс, который будет служить диспетчером при создании или использовании участвующих в ассоциации объектов.

Выбор реализации

Если модель не содержит указаний относительно выбора метода реализации ассоциаций, вы можете сделать этот выбор самостоятельно, опираясь на масштаб и архитектуру системы, среду разработки приложения и природу этого приложения. Если вы не хотите модифицировать существующие классы, создайте класс-ассоциацию. С помощью таких классов лучше всего реализуются ассоциации типа «один-ко-многим», в случае если объектов у полюса с кратностью «много» может быть действительно много.

Для простых ассоциаций удобно использовать атрибуты-ссылки. Односторонние ассоциации более эффективны и безопасны. Инкапсуляция ссылок требует особого внимания.

18.4. Реализация функциональности

Закончив реализацию структуры модели, вы можете переходить к реализации функциональности. Для каждого класса нужно указать методы вместе с их сигатурой (название метода, параметры, тип результата). Модель классов подразумевает существование множества методов. Очевидно, что вы должны иметь возможность создавать и уничтожать объекты и связи, а также обращаться к значениям атрибутов. Более того, возможность прослеживания связей модели классов также подразумевает существование определенных методов. Методы также возникают вследствие существования выводимых атрибутов, моделей состояний и взаимодействия.

1. Создание объекта (раздел 18.4.1).
2. Существование объекта (раздел 18.4.2).
3. Уничтожение объекта (раздел 18.4.3).
4. Создание связи (раздел 18.4.4).
5. Уничтожение связи (раздел 18.4.5).
6. Выводимые атрибуты (раздел 18.4.6).

Объекты обладают состоянием, поведением и индивидуальностью. Это отражается жизненным циклом объекта. Объекты создаются системой и должны инициализироваться в правильном состоянии. В С++ имеются средства уничтожения объектов по желанию программиста, тогда как в Java программист может лишь предложить сборщику мусора уничтожить объект. В обоих языках имеются специальные методы, которые запускаются в момент создания объекта, а также методы, позволяющие описать завершение существования объекта.

Объекты могут запрашивать друг у друга сервисы и информацию. Для этого они вызывают методы друг друга. В Java и в С++ поведение объектов вызывается посредством оператора членства «.», который указывает необходимость вызова операнда, находящегося в правой части (метода класса), для указанного целевого объекта. Тот же синтаксис используется для обращения к атрибутам объекта: *X.y* означает атрибут *y* объекта *X*. Обычно атрибуты инкапсулируются, а потому оказываются недоступны внешним объектам.

В контексте класса объектам доступна ссылка на самих себя, которая в обоих языках называется *this*. Явно указывать эту ссылку перед именами членов класса не обязательно. В Java *fn()* равносильно *this.fn()*, а *y=10* означает *this.y=10*. В C++ *this* — это указатель, а потому с ним необходимо использовать специальный оператор членства *this->fn()*.

Чтобы объект мог обратиться к другому объекту за сервисами или информацией, он должен знать имя целевого объекта. Это имя может быть получено из связи: в составе объекта может присутствовать указатель или ссылка на другой объект или через параметр, когда объект получает идентификатор другого объекта при вызове метода. Любой вызов методов целевого объекта осуществляется через идентификатор и ограничивается установленным для этих методов уровнем доступа.

В C++ и Java существует концепция статических (*static*) членов классов, которые используются всеми объектами класса совместно. Жизненный цикл статического элемента полностью отличается от жизненного цикла обычного объекта. К такому элементу можно обратиться через сам класс, независимо от существования экземпляров этого класса (см. раздел 18.4.2).

18.4.1. Создание объекта

В объектно-ориентированных языках объекты обычно создаются динамически (в процессе выполнения программы): программа запрашивает систему о создании экземпляра конкретного класса. В Java это требование действует только на непримитивные типы, тогда как примитивные типы размещаются в памяти статически во время компиляции. В C++ допустимо статическое и динамическое выделение памяти и под примитивные, и под объектные типы. Для создания объектов в обоих языках используется оператор *new*.

```
//C++
// статическое выделение памяти под один счет
Account acct1;

// статическое выделение памяти под массив из 10 счетов
Account accounts1[10];

// динамическое выделение: определение указателя,
// инициализация объекта оператором new
Account* acct2 = new Account;

// динамическое создание массива из 10 счетов
Account* accounts2 = new Account[10];

//Java
// создание ссылки, инициализация объекта оператором new
Account acct = new Account();

// создание массива из 10 ссылок, но не самих счетов
Account accounts = new Account[10];

// создание счетов, на которые будут ссылаться элементы массива
for (int i=0; i<10; i++) accounts[i] = new Account();
```


Когда создается новый объект, система выделяет память под его атрибуты и выполняет другие действия, подразумеваемые началом жизненного цикла объекта. Объектно-ориентированные языки освобождают программиста от необходимости понимания технических деталей реализации объектов. Java и C++ позволяют программисту указывать, какие операции должны быть выполнены во время создания объекта, поэтому вы имеете возможность обеспечить внутреннюю непротиворечивость только что созданного объекта. Как только система завершает свои действия по созданию объекта, вызывается специальный метод этого объекта, называемый *конструктором* (constructor). Конструктор — это метод, не возвращающий никакого значения. Его имя совпадает с именем класса. Он может иметь произвольное количество параметров и может быть перегружен. Например, приведенный ниже код на Java показывает, что класс *Account* (Счет) имеет два конструктора, один из которых требует указания начального баланса (а другой не принимает никаких аргументов).

```
public class Account {
    . . .
    public Account(float OpeningBalance) {
        balance = OpeningBalance;
    }
    public Account() { balance = 0; }
    . . .
}
```

Внутри конструктора можно присваивать значения членам класса, создавать входящие в него объекты и выполнять прочие действия. Конструктор выполняется уже для полностью сформированного объекта, поэтому он может вызывать другие методы и обладает теми же возможностями, привилегиями и ограничениями, как и все остальные методы. Если у объекта не определен конструктор, считается, что он имеет форму $X()\{\}$. Хотя такой конструктор не выполняет никаких действий, он допускает выражения, содержащие операцию создания объекта, имеющую соответствующую форму. После создания параметризованного конструктора неявный конструктор без параметров больше не будет поддерживаться системой. Если он вам нужен, придется определить его самостоятельно (как это сделано в примере выше).

Конструкторы в подклассах тоже не представляют особых сложностей для понимания. Объект, принадлежащий к типу подкласса, наследует атрибуты и методы суперкласса, а также содержит все члены, определенные на уровне подкласса. Можно считать, что сначала создается родительский объект, а потом к нему добавляется часть, определенная в подклассе. Хотя конструкторы не наследуются, они выполняются последовательно от наиболее общих к наиболее конкретным. В случае C++, где допускается статическое создание объектов, объекты-члены создаются рекурсивно (с вызовом их конструкторов) до завершения создания содержащего их объекта. Только после этого запускается конструктор внешнего объекта. Рассмотрим это на примере:

```
class X {
    public:
        X() { cout << "X!"; }
```

```

class Y : public X {
    public:
        Y() { cout << "Y!"; }
};

class Z : public Y {
    public:
        Z() { cout << "Z!"; }
};

int main() {
    Z z; // просто создаем объект Z
    return 0;
}

```

В результате выполнения этой программы будет выведена строка *X!Y!Z!*.

Отсутствие явно определенного конструктора считается плохим стилем программирования. По умолчанию C++ никак не инициализирует члены класса, тогда как Java инициализирует их значениями 0 или *null* в зависимости от типа. В любом случае состояние объекта вряд ли будет корректным с точки зрения модели состояний. Назначение конструктора в том, чтобы программист имел возможность провести инициализацию объекта и операции над ним таким образом, что к началу своей деятельности этот объект был бы полностью сформированным, внутренне согласованным и готовым к работе.

Если над созданным объектом не должны быть выполнены никакие операции, Java позволяет инициализировать элементы прямо в определении класса. В C++ для этого используется список инициализации членов, в котором указываются значения членов, присваиваемые им перед запуском конструктора. В обоих языках операции присваивания можно помещать в тело конструктора, однако инициализация всегда считается предпочтительнее присваивания.

```

//Java
public class Account {
    private float balance = 0; // инициализация
    . . .
    public Account() {} // в конструкторе присваивание уже не нужно
}

//C++
class Account {
    float balance; // указывать значение нельзя
    . . .
public:
    Account() : balance(0) {} //список инициализации
}

```

В C++ имеется также копирующий конструктор, имеющий семантику копирования при присваивании. По умолчанию предоставляется конструктор формы *X(const X& x)*, подразумевающий почленное копирование содержимого. Java не поощряет копирование объектов: программист должен указать, что класс реализует интерфейс *Cloneable*, и перекрыть метод *clone()* класса *Object*.

18.4.2. Существование объекта

Статически выделенные объекты, созданные во время компиляции, которые в C++ могут иметь произвольный тип, а в Java могут быть только примитивными типами и переменными-ссылками, существуют в рамках программного блока, ограниченного фигурными скобками `{}`. Они автоматически уничтожаются, когда точка выполнения выходит за границы этого блока. Динамически выделенные объекты хранятся в памяти до тех пор, пока они не будут уничтожены явно (в C++) или пока не перестанут использоваться программой (в Java).

В обоих языках допустимо определение статических членов класса. Такие члены принадлежат не какому-либо объекту класса, но всему классу в целом, причем работать с ними можно даже в отсутствие экземпляров класса (если, конечно, эти члены являются открытыми). Для этого служит синтаксис `X::StaticMethod()` или `X.StaticMethod()` (C++, Java). Можно обращаться к ним и через объекты класса, но при этом нужно помнить о том, что они не являются членами данного экземпляра.

Статические члены существуют до создания первого объекта данного типа — в момент создания программы C++ или при загрузке класса Java — и не прекращают своего существования до завершения программы. В C++ статические члены существуют в виде независимых объектов (для них выделяется отдельная память), тогда как в Java каждый класс сам по себе существует независимо (имеется экземпляр самого класса), и ему принадлежат все статические члены. Жизненный цикл статических членов класса никак не связан с объектами этого класса. Поскольку статические методы не являются членами классов, синтаксис *this* (и *super*) внутри них применять бессмысленно. Из статического метода можно сослаться на обычные данные, находящиеся внутри объекта, только указав имя этого объекта.

В C++ глобально определенные переменные характеризуются таким же жизненным циклом, что и статические члены классов. Они создаются перед началом программы и удаляются при ее завершении. Глобальные переменные считаются примером плохого стиля программирования, однако глобальные функции используются в C++ достаточно широко. Java не допускает определение функций вне классов, но такие функции часто эмулируются во вспомогательных библиотечных классах, содержащих группы статических методов. Некоторые библиотечные классы Java, такие как *Integer* и *Collections*, полностью или почти полностью состоят из статических методов.

Поскольку статические члены и экземпляры классов создаются отдельно от экземпляров объектов, инициализация их также происходит независимо. Статические члены в C++ объявляются внутри классов, но определяются и инициализируются вне их, тогда как в Java статические поля можно инициализировать как обычные поля или внутри статического инициализирующего блока. Статические члены гарантированно инициализируются перед использованием в программе, но взаимный порядок инициализации этих членов не определен.

18.4.3. Уничтожение объекта

Если объект больше не нужен, его можно уничтожить, а память, которую он занимал, вернуть в пул. Статически выделенные объекты уничтожаются при выходе точки выполнения за границы программного блока. Идентификатор уничтоженного таким образом объекта использовать больше нельзя, потому что это будет ошибкой, — все равно, что позвонить по старому телефону человеку, который переехал.

Срок существования динамически выделенных объектов потенциально может быть неограниченным, поэтому необходимо отслеживать те объекты, которые больше не нужны или существование которых перестает быть корректным, уничтожать их и освобождать занимаемую память. Для этого в C++ и Java предусмотрены разные стратегии. В C++ программист несет ответственность за явное удаление объектов. В Java система сама отслеживает неиспользуемые объекты и удаляет их. Второй подход называется *сборкой мусора* (garbage collection) и реализуется через подсчет количества ссылок на объект. Когда ссылки заканчиваются (идентификаторов объекта больше нет), система помечает объект как подлежащий уничтожению. Периодически запускается сборщик мусора, который освобождает память, занимаемую всеми помеченными объектами, и возвращает ее в системный пул.

Удаление объектов вручную в C++ создает дополнительную нагрузку на программиста, но зато дает возможность жестко контролировать жизненный цикл объекта и полностью моделировать события, происходящие с объектом, включая завершение его существования. Сборка мусора освобождает программиста от заботы о памяти, но ограничивает его контроль над завершением жизненного цикла.

В классах C++ могут быть определены *деструкторы* (destructors), аналогичные конструкторам. Они запускаются автоматически при уничтожении объекта. Деструктор выглядит как конструктор без аргументов, но перед именем класса ставится символ ~, как в следующем примере: `X::~X() {...}`. Деструкторы никогда не принимают аргументов и не возвращают никаких значений.

Обычно в деструкторе выполняются действия, обратные действиям конструктора. Чаще всего к ним относится удаление динамически созданных членов класса, а также увеличение или уменьшение каких-либо счетчиков, освобождение ресурсов и т. д. Деструкторы могут вызывать определенное поведение, которое должно быть осуществлено в конце жизненного цикла объекта.

```
Window :: ~Windows ()
{
    // стирание окна и прорисовка скрытой им области
}
```

Процесс уничтожения противоположен процессу создания: сначала выполняется код деструктора, а затем уничтожаются требующие удаления члены класса (начиная с их деструкторов), после чего вся освобожденная память возвращается на кучу.

Удаление динамически созданного объекта осуществляется оператором *delete*. Пустые квадратные скобки позволяют отличить удаление отдельных элементов массива от удаления массива в целом.

```
class X { . . . }
. . .
X x1 = new X;
X* x2 = new X[20];
. . .
delete x1;
delete [] x2;
```

После удаления объекта C++ рекомендуется сразу же присвоить нулевые значения всем указателям, которые были с ним связаны. Операция удаления, случайно примененная к нулевому указателю, не вызовет сбоя программы, тогда как повторное удаление объекта приводит к непредсказуемым результатам.

Объекты Java не могут быть удалены программистом вручную. Программист может попытаться инициировать уничтожение объекта, установив все ссылки на него равными нулю, и даже может порекомендовать системе выполнить сборку мусора, но гарантировать, что уничтожение будет выполнено в определенный момент, он не может.

Java позволяет классам перекрывать метод *finalize()*, определенный в классе *Object*. Этот метод аналогичен деструкторам C++, однако существенное отличие состоит в том, что невозможно предсказать, в какой момент этот метод будет вызван системой.

18.4.4. Создание связи

Взаимодействие объектов друг с другом приводит к созданию и уничтожению связей между ними. Такие связи должны создаваться во время выполнения поведения (метода), которое устанавливает ассоциацию между объектами, и удаляться во время выполнения другого поведения, которое эту ассоциацию разрушает. Следует избегать методов, осуществляющих непосредственное присваивание, а вместо них определять операции, скрывающие данные и устанавливающие связи только при выполнении определенных условий.

Для установки связи, соответствующей односторонней ассоциации, объект запоминает идентификатор другого объекта, являющегося параметром операции. Двусторонние связи могут быть созданы путем обмена идентификаторами, однако обычно рекомендуется инициировать обмен с одной стороны и инкапсулировать данные на другом конце, чтобы обеспечить непротиворечивую логику и полное обновление связей. Выбор класса, управляющего обменом, определяется логикой приложения.

В приведенном ниже упрощенном примере на Java студенты (*Students*) хранят список текущих курсов лекций (*Courses*), а для каждого курса хранится список студентов. Поскольку решение прослушать курс лекций принимается студентом, а выполнение этого решения зависит от статуса студента, создание связи управляется методом *Student.addClass(Course)*. Этот метод вызывает *Course.enroll(Student)*, который доступен классу *Student* благодаря пакетному уровню доступа. Этот

уровень доступа предотвращает открытый доступ к клиентам (к классу *EnrollmentApplication*), и, таким образом, создание связи может выполняться единственным способом, описанным выше.

```
// файл Course.java
package school;
public class Course {
    private String title;
    private HashSet students = new HashSet();

    public Course (String nm) { title = nm; }
    public String courseName() { return title; }

    boolean enroll(Student stu) {
        // проверка наличия свободных мест на курсе и т.д.
        // в случае положительного ответа возвращается результат
        // метода HashSet.add
        return students.add(stu);
    }

    public void printClassList() {
        System.out.println("Class List for " + courseName() + ":");
        Iterator it = students.iterator();
        while (it.hasNext())
            System.out.println(((Student)it.next()).studentName());
    }
}

// файл Student.java
package school;
public class Student {
    private String name;
    private HashSet classes = new HashSet();

    public Student(String nm) { name = nm; }
    public String studentName() { return name; }
    public boolean addClass(Course crs) {
        // проверка доступности курса студенту, в случае положительного
        // ответа запрашивается добавление студента на курс у класса Course,
        // а потом курс добавляется в список
        return ( crs.enroll(this)) ? classes.add(Crs) : false;
    }

    public void printCourses() {
        System.out.println("Courses for " + studentName() + ":");
        Iterator it = classes.iterator();
        while (it.hasNext())
            System.out.println(((Course)it.next()).courseName());
    }
}

// файл EnrollmentApplication.java
import local.school.*
```

```

public class EnrollmentApplication {
    public static void main(String [] args) {
        Student mike = new Student("mike");
        Student bill = new Student("bill");
        . . .
        Course tt = new Course("Type Theory");
        mike.AddClass(tt);
        bill.AddClass(tt);

        tt.PrintClassList();
        mike.PrintCourses();
    }
}

```

Если на момент создания объекта известно, с какими объектами он будет взаимодействовать, связи можно установить сразу же. Достаточно часто создание объекта должно зависеть от возможности установления связей. В этом случае конструирование объекта следует делегировать специальному методу, который будет проверять необходимые условия перед созданием объекта, потому что конструкторы Java и C++ не могут возвращать ошибку или прерывать процесс создания. Средством для создания объектов может быть специальный класс или статический метод (см. раздел 18.4.2).

В приведенном ниже примере на C++ объект *Transaction* (Транзакция) не будет создан до тех пор, пока не будет проверен номер счета и получен идентификатор этого счета (с которым устанавливается связь).

```

class Transaction {
    . . .
protected:
    // закрытый или защищенный конструктор, наличие которого исключает
    // возможность создания объекта извне вызовом его конструктора
public:
    . . .
    static Transaction* MakeTransaction
        (const char* acctNumber) {
        Account* acct;
        // возврат нулевого указателя показывает, что транзакция
        // запрещена
        if ( /* счет не в порядке */ ) return 0;
        // возврат транзакции для корректного счета
        return new Transaction(*acct);
        . . .
    }
};

```

18.4.5. Удаление связи

Удаление связи означает процедуру, обратную ее созданию. Ссылка обычно разрывается тем методом, действия которого приводят к разрушению ассоциации. В большинстве случаев это означает, что атрибут-ссылка устанавливается равным нулю или что из совокупности ссылок изымается один элемент.

Например, студент (*Student*) из нашего примера может прекратить слушать курс лекций (*Course*).

```
// Student.dropClass(Course):
    public boolean dropClass(Course crs) {
        if (!classes.contains(crs)) return false;
        if (!crs.drop(this)) return false;
        classes.remove(classes.indexOf(crs));
        return true;
    }

// Course.drop(Student):
    boolean drop(Student stu) {
        if (!students.contains(stu)) return false;
        students.remove(students.indexOf(stu));
        return true;
    }
```

Если связи были созданы между уже существовавшими объектами, эти объекты, скорее всего, продолжают существовать и после уничтожения связей. Нужно следить за тем, чтобы объект, на который ссылался другой объект, был доступен по каким-либо ссылкам после уничтожения связи. В противном случае может возникнуть утечка памяти в C++, или же объект, оставшийся без ссылок на него, будет уничтожен сборщиком мусора в Java.

В C++ создание и уничтожение связей осуществляется парой конструктор-деструктор. Эти связи обычно отражают захват и освобождение ресурсов.

Связи между объектами, существование которых зависит друг от друга, могут требовать удаления одного из объектов при разрыве. Это, в свою очередь, может потребовать удаления или обновления других объектов, связанных с уничтожаемым. Здесь можно провести аналогию с базами данных, где удаление записи может каскадно распространиться на связанные с ней записи или быть запрещено для сохранения целостности существующих записей.

18.4.6. Производные атрибуты

Для достижения точности и актуальности желательно рассчитывать значения производных атрибутов по независимым атрибутам в момент обращения к первым. Обычно время, затрачиваемое на это вычисление, пренебрежимо мало по сравнению с затратами на регулярное обновление и на хранение избыточных данных.

Широко распространена ошибка, связанная с включением в объекты избыточных данных о состоянии. Состояние часто можно определить по значениям остальных атрибутов (Java):

```
public class Account {
    // неправильно: избыточные данные для счета с превышением кредита
    private boolean overdrawn;
    private float balance;
    . . .
    // правильно: определение состояние счета по запросу
    public boolean isOverdrawn() { return balance < 0; }
```


Состояние объекта можно определить и по его связям (C++):

```
class Customer {
    . . .
    List<Account> accts;
public:
    bool hasOverdrawnAccount() {
        // перебор всех счетов клиента
        // возвращение true в случае наличия счетов
        // с превышением кредита
    }
}
```

Или же состояние может зависеть от существования или отсутствия связей (C++):

```
class Telephone {
    Telephone* connectedTo;
    . . .
public:
    // есть ли соединение с другим телефоном?
    // если ссылка ненулевая, то есть
    bool isBusy() { return connectedTo != 0; }
    . . .
}
```

18.5. Практические рекомендации

Вот несколько советов по реализации проектов на C++ и Java.

- **Перечисления.** Используйте перечислимые типы для повышения ясности кода и ограничения возможных значений (раздел 18.3.1).
- **Пакеты Java.** Не используйте пакет по умолчанию. Указывайте имена пакетов и с их помощью управляйте доступом (раздел 18.3.3).
- **Управление доступом.** Объявляйте все атрибуты и недоступные другим классам методы как закрытые — *private*. Ослабляйте ограничение только в том случае, если для этого есть веские основания (раздел 18.3.3).
- **Дружественные классы.** Воздерживайтесь от использования спецификатора *friend* в C++, потому что он нарушает инкапсуляцию (раздел 18.3.3).
- **Интерфейсы Java.** Реализуйте множественное наследование посредством интерфейсов. Интерфейс объявляет методы и постоянные атрибуты, а также позволяет указывать тип объектов (раздел 18.3.4).
- **Закрытое наследование в C++.** Избегайте закрытого наследования классов. Вместо него лучше использовать делегирование (раздел 18.3.4).
- **Односторонние ассоциации.** Используйте односторонние ассоциации везде, где нет необходимости в прослеживании связи в двух направлениях. Односторонние ассоциации легче поддерживать, они сокращают взаимную зависимость объектов (раздел 18.3.5).
- **Ссылки C++ и квалификатор final в Java.** Обращайте внимание на полюса ассоциаций, которые должны быть привязаны к объектам в момент

инициализации и не подлежат дальнейшим изменениям. Эта семантика полностью сохраняется ссылками C++ и частично реализуется посредством квалификатора *final* в Java (раздел 18.3.5).

- **Конструкторы.** Отсутствие явно определенного конструктора в классе считается плохим стилем программирования. Конструктор обеспечивает инициализацию объекта и перевод его в корректное начальное состояние (раздел 18.4.1).
- **Удаление в C++.** После удаления объекта в C++ рекомендуется сразу же обнулить все указатели на этот объект во избежание случайного повторного его удаления, которое может привести к катастрофическим последствиям (раздел 18.4.3).
- **Удаление связи.** При удалении связи желательно следить за доступностью участвовавших в ней объектов или контролировать их удаление, если оно необходимо. В противном случае возникнет утечка памяти в C++, или же объект может быть непредвиденно удален сборщиком мусора Java (раздел 18.4.5).

18.6. Резюме

Реализовать объектно-ориентированный проект на объектно-ориентированном языке относительно легко, потому что конструкции языка близки конструкциям модели. В этой главе мы рассмотрели два наиболее популярных объектно-ориентированных языка: C++ и Java.

Первый этап воплощения объектно-ориентированного проекта — реализация структуры модели классов. Сначала следует задать тип всех атрибутов. Везде, где это возможно, вместо строковых значений лучше использовать числовые. Числовые типы лучше размещаются в памяти и более эффективно обрабатываются, а также облегчают контроль согласованности атрибутов.

Затем следует определить сами классы. Лучше всего начинать с общедоступного интерфейса класса, а затем добавлять внутренние методы, атрибуты и встроенные классы, необходимые для реализации методов интерфейса.

Особое внимание следует уделять управлению доступом. Контроль доступа позволяет инкапсулировать атрибуты и методы и ограничить обращения к ним. Основные спецификаторы доступа в C++ и Java совпадают. Закрытые члены класса доступны только его собственным методам. Открытые члены класса доступны всем клиентам. Спецификатор *protected* в C++ открывает доступ самому классу и его подклассам. В Java защищенные (*protected*) члены доступны методам того же пакета. Таким образом, пакеты Java служат не только для упорядочения кода, но и для управления доступом. Объявление дружественности в C++ (*friend*) позволяет организовать выборочный доступ к закрытым членам класса.

Объектно-ориентированные языки позволяют реализовать обобщение через наследование. C++ поддерживает множественное наследование непосредственно, а Java — посредством концепции интерфейсов. Интерфейс Java — это спецификация класса, из которой полностью исключены детали реализации.

Полиморфизм в Java осуществляется автоматически. Порождение подклассов можно запретить посредством квалификатора *final*. Этот квалификатор, приме-

ненный к методам, предотвращает их перекрытие. В C++ полиморфизм управляется программистом при помощи квалификатора *virtual*.

Объектно-ориентированные языки не имеют встроенной поддержки ассоциаций. Однако их можно легко реализовать посредством указателей или ссылок или выделенных объектов-ассоциаций. Односторонние ассоциации сокращают взаимную зависимость классов, поэтому их следует использовать всегда, когда прослеживание планируется осуществлять только в одном направлении. C++ позволяет реализовать полюса ассоциации двумя способами. Указатель позволяет изменять связь или делать ее нулевой, тогда как ссылка получает свое значение в момент инициализации и не может быть изменена или сделана нулевой. В Java можно объявить атрибут объектного типа как *final*, что запрещает изменение связи, но не требует ее изначальной инициализации.

Если ассоциацию нужно прослеживать в двух направлениях, используйте двустороннюю реализацию. Вы можете добавить указатель или ссылку (или множество указателей или ссылок) в каждый класс или создать отдельный объект-ассоциацию. Двусторонние указатели создают определенную избыточность, поэтому необходимо внимательно следить за их согласованностью.

Закончив реализацию структуры модели классов, переходите к реализации методов. Аккуратно определяйте конструкторы объектов, обеспечивайте корректную инициализацию. Столь же внимательно следует относиться и к деструкторам, не забывая освобождать системные или программные ресурсы.

Не стоит хранить в объектах избыточные данные. Точность и надежность достигаются вычислением значений выводимых атрибутов в момент обращения к ним.

Таблица 18.2. Ключевые понятия главы

абстрактный	производные данные	пакет	private
модификатор доступа	деструктор	указатель	new
спецификатор доступа	перечисление	полиморфизм	interface
ассоциация	сборка мусора	ссылка	friend
конкретный	обобщение	virtual	final
конструктор	пространство имен	static	public
тип данных	перегрузка	protected	

Библиографические замечания

В книге [Stroustrup-94] обсуждаются вопросы, связанные с реализацией проектов на языках программирования.

Литература

[Arnold-00] Ken Arnold, James Gosling, and David Holmes. The Java Programming Language, Third Edition. Boston: Addison-Wesley, 2000.

[Gosling-00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. Java Language Specification, Third Edition. Boston: Addison-Wesley, 2000. (<http://java.sun.com/docs/books/jls/>)

[Liskov-88] Barbara Liskov. Data abstraction and hierarchy. SIGPLAN Notices, 23, 5 (May 1988).

[Stroustrup-94] Bjarne Stroustrup. The Design and Evolution of C++. Boston: Addison-Wesley, 1994.

[Stroustrup-98] Bjarne Stroustrup. The C++ Programming Language, Third Edition. Boston: Addison-Wesley, 1997.

Упражнения

18.1. (1) Задайте тип данных для каждого атрибута на рис. У18.1.

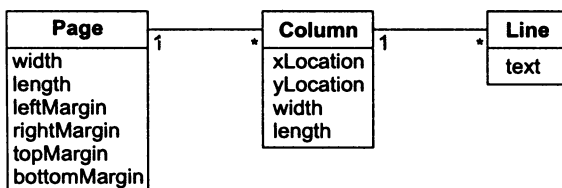


Рис. У18.1. Часть диаграммы классов системы верстки

18.2. (4) Рассмотрим модель на рис. У18.2. В этом упражнении вы можете не обращать внимания на атрибуты, методы и ассоциацию. Комментарий к модели приводится в упражнении 15.10.

- 1) Объявите все классы на C++ с учетом наследования. Следите за уровнем доступа.
- 2) Объявите все классы на Java с учетом наследования. Следите за уровнем доступа и пакетами.

18.3. (6) На рис. У18.2 атрибут *visibility* управляет отображением лицевой стороны карты. Добавьте ответ к этому упражнению к ответу на упражнение 18.2. Продолжайте игнорировать методы и ассоциации. (Замечание для преподавателя: вы можете предоставить студентам наш ответ к упражнению 18.2.)

- 1) Объявите все атрибуты, за исключением *location*, на C++. Определите перечислимые типы.
- 2) Объявите все атрибуты, за исключением *location*, на Java. Определите перечислимые типы.

18.4. (7) Добавьте в свои ответы к упражнению 18.3 ассоциацию *CardCollection* — *Card*. Вы можете воспользоваться шаблоном (о них мы в этой книге почти не рассказываем). Подумайте о реализации ассоциаций и почитайте о стандартной библиотеке и шаблонах в Интернете или книге о C++.

18.5. (6) Объявите методы в программе из упражнения 18.3. См. упражнение 15.10.

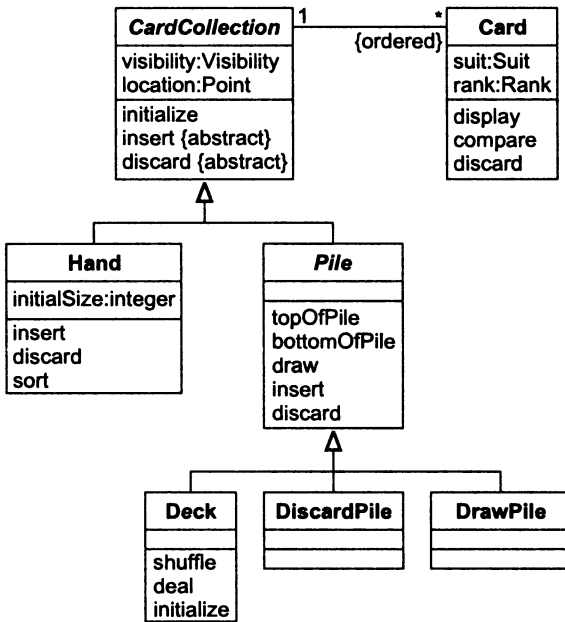


Рис. У18.2. Часть диаграммы классов компьютерного игрока в карты

18.6. Напишите программу на C++ или Java, включая объявления классов и методов. Программа должна реализовать предложенную ниже модель при помощи указателей:

- 1) (9) ассоциация типа «один-к-одному», прослеживаемая в обоих направлениях;
- 2) (6) ассоциация типа «один-ко-многим», прослеживаемая в направлении от одного ко многим. Ассоциация не упорядочена;
- 3) (6) ассоциация типа «один-ко-многим», прослеживаемая в направлении от одного ко многим. Ассоциация упорядочена;
- 4) (8) ассоциация типа «многие-ко-многим», прослеживаемая в обоих направлениях. В одном из них она упорядочена, в другом — нет.

18.7. (7) Опишите возможные стратегии управления памятью при условии, что автоматический сборщик мусора недоступен. Ответ должен содержать рекомендации, которые программист мог бы использовать в процессе кодирования.

- 1) **Система для работы с текстом.** Система часто создает одну большую строку в непрерывном участке памяти путем объединения нескольких строк меньшей длины. Вы не можете расходовать память зря и не можете установить ограничение сверху на длину строки или на количество соединяемых строк. Напишите псевдокод, который будет соединять строки и освобождать неиспользуемую память.
- 2) **Многопроходный компилятор.** Объекты создаются динамически. На каждом проходе проверяются объекты, созданные на предыдущем проходе,

и создаются объекты для обработки при следующем проходе. Виртуальное адресное пространство системы, в которой будет работать компилятор, практически не ограничено, а операционная система имеет хороший алгоритм своппинга. Методы динамического выделения и освобождения памяти из стандартной библиотеки неэффективны. Обсудите преимущества и недостатки двух альтернативных вариантов: 1) забыть о сборке мусора и дать операционной системе выделить большой объем виртуальной памяти; 2) освобождать память после удаленных объектов при помощи сборщика мусора.

- 3) **Программное обеспечение для длительной эксплуатации, например банковское или авиадиспетчерское.** Вы работаете с той же системой и библиотекой, что и в упражнении 18.7(2). Обсудите преимущества и недостатки возможных подходов.
- 4) **Метод, который может создать и вернуть объект, использующий большой объем памяти.** Обсудите преимущества и недостатки двух подходов: 1) каждый раз при вызове метода он удаляет созданный им в предыдущий раз объект, если таковой обнаруживается; 2) каждый раз при вызове метода он может создать новый объект. Об удалении старого объекта должен заботиться вызывающий метод. Прокомментируйте эти подходы.

18.8. (6) Как можно разбить модель на рис. У12.4 по пакетам Java?

18.9. (7) Напишите объявления классов модели на рис. У18.3 на Java и C++. Реализуйте модель целиком. Следите за управлением доступом и используйте пакеты. Не забудьте о конструкторах и деструкторах. Реализуйте все ассоциации как двусторонние. Комментарии к модели приведены в упражнениях главы 12.

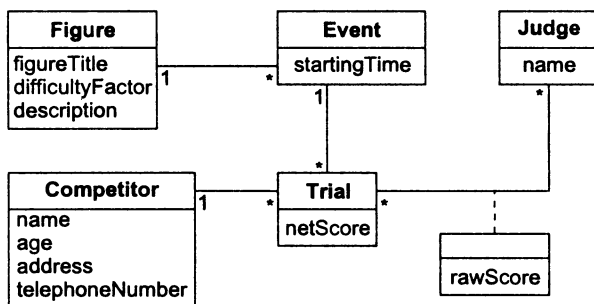


Рис. У18.3. Неполная диаграмма классов системы подсчета очков

18.10. (6) Напишите программу на Java или C++, соответствующую псевдокоду из упражнения 15.11. Ваш код должен быть инкапсулированным. (Замечание для преподавателя: можно предоставить студентам наш ответ к упражнению 15.11.)

- 18.11. (8) Напишите программу на C++ или Java для каждого из выражений OCL из раздела 3.5.3. Предположите, что ассоциации реализованы как двусторонние. Выражение OCL может не учитывать инкапсуляцию, но код, который вы напишете, должен быть максимально скрытым.
- 18.12. (6) Реализуйте все ассоциации, в которых участвуют классы *Box*, *Link*, *LineSegment* и *Point* на рис. У18.4. Воспользуйтесь C++ или Java. Обратите внимание, что редактор позволяет создавать связи только между парами прямоугольников.
- 18.13. (7) Реализуйте операцию *cut* для класса *Box* на рис. У18.4 на языке C++ или Java. Для простоты предположите, что *cut* удаляет вырезаемые объекты и не помещает их в буфер. Операция должна распространяться с прямоугольников на связи между ними. Обновляйте все затрагиваемые операцией ассоциации. Позаботьтесь об освобождении памяти (только для C++). Вы можете предположить, что изображение на экране будет обновлено другим методом.
- 18.14. (7) Напишите метод на C++ или Java, который будет создавать связь между двумя прямоугольниками. На вход метода подаются два прямоугольника и список точек. Метод должен обновить ассоциации и при необходимости создать экземпляры объектов. Вы можете предположить, что изображение на экране будет обновлено другим методом. Напишите метод для уничтожения связи.

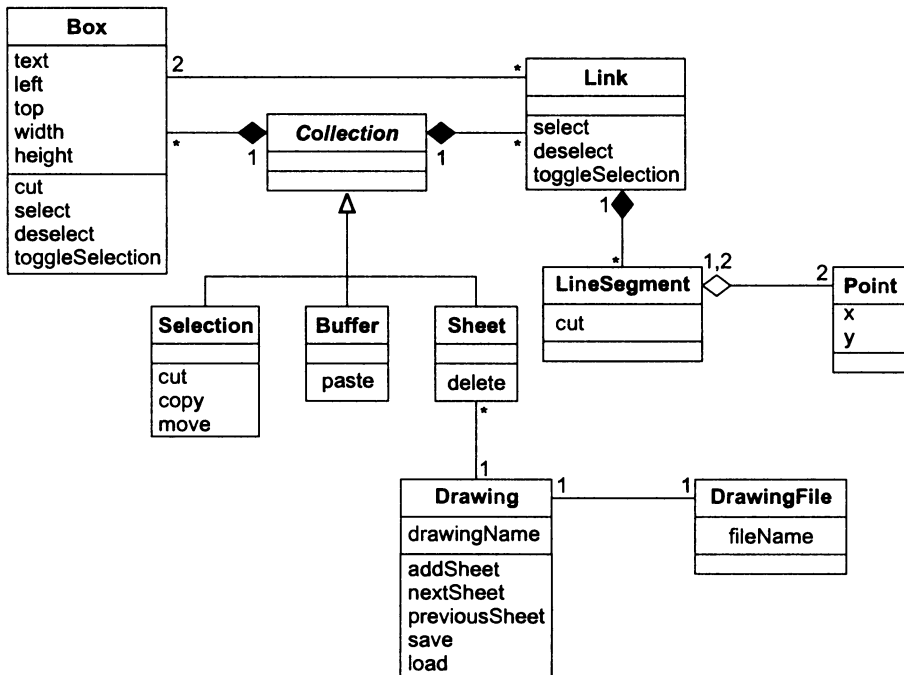


Рис. У18.4. Диаграмма классов редактора диаграмм

- 18.15. (8) На языке C++ или Java реализуйте перечисленные ниже запросы для модели на рис. У18.4.
- 1) Дан прямоугольник, нужно найти все непосредственно связанные с ним прямоугольники.
 - 2) Дан прямоугольник, нужно найти все непосредственно или косвенно связанные с ним прямоугольники.
 - 3) Дан прямоугольник и связь. Определить, участвует ли прямоугольник в этой связи.
 - 4) Дан прямоугольник и связь. Найти второй прямоугольник, соединенный этой связью с первым.
 - 5) Даны два прямоугольника, найти все связи между ними.
 - 6) Дано множество выделенных объектов и лист. Определить, какие связи соединяют выделенный прямоугольник с невыделенным.
- 18.16. Даны два прямоугольника и связь, получить упорядоченное множество точек. Первая точка соответствует соединению связи с первым прямоугольником, последняя точка — соединению связи со вторым прямоугольником, а промежуточные точки соответствуют изломам связи.

Объектно-ориентированная парадигма может быть приспособлена не только к программному коду, но и к базам данных. Как ни странно, модели UML можно реализовать не только в объектно-ориентированных, но и в реляционных базах данных. В результате получаются эффективные, согласованные, легко расширяемые системы.

Подготовка к реализации в базе данных начинается с этапа анализа, описанного в главе 12. Этот этап заканчивается построением модели предметной области. Прочие методологические главы из второй части применимы главным образом к программному коду и в значительно меньшей степени — к базам данных. Эта глава продолжает рассказ с того места, на котором он закончился в главе 12. Каким образом можно отобразить модель в структуры базы данных и оптимизировать результат для достижения максимальной производительности? Как можно связать базу данных с программным кодом? Для читателей, мало знакомых с предметом, мы даем небольшое введение в базы данных.

Речь пойдет в первую очередь о реляционных базах, потому что они доминируют на рынке. Объектно-ориентированные базы данных пригодны только для приложений определенного рода, поэтому мы коротко расскажем о них ближе к концу главы.

19.1. Введение

19.1.1. Концепции баз данных

База данных (database) — это долговременное *самодокументированное* (self-descriptive) хранилище данных, размещаемое в одном или нескольких файлах. Именно «самодокументированность» отличает базу данных от набора обычных файлов: в базе хранится структура данных или *схема* (schema), а не только сами данные.

Система управления базой данных, СУБД (database management system — DBMS) — это программное обеспечение, управляющее доступом к данным. Одной из основных целей объектно-ориентированных технологий является расширение возможностей повторного использования. В приложениях, активно работающих с данными, СУБД может заменить значительную часть кода. Обращаясь к сервисам СУБД, вы достигаете того же эффекта, что и при повторном использовании собственного кода. Есть и другие причины, по которым можно обратиться к СУБД.

- **Защита данных.** СУБД защищают данные от случайной порчи, связанной со сбоями аппаратного обеспечения, ошибками носителей и приложений.
- **Эффективность.** СУБД основаны на эффективных алгоритмах, предназначенных для работы с большими объемами данных.
- **Многопользовательский режим.** Множество пользователей могут работать с базой данных одновременно.
- **Работа с несколькими приложениями.** Несколько приложений могут одновременно считывать данные из базы и записывать их туда. База данных — нейтральный посредник, обеспечивающий взаимодействие между программами.
- **Качество данных.** Вы можете указать правила, которым должны удовлетворять данные. СУБД может контролировать качество данных в дополнение к тому контролю, который обеспечивается приложениями.
- **Распределенность данных.** Данные могут быть разделены между разными сайтами, организациями и платформами. СУБД обеспечивает согласованность фрагментированных данных.
- **Безопасность.** СУБД позволяет разрешить работу с данными только ограниченному кругу авторизованных пользователей.

19.1.2. Концепции реляционных баз данных

Реляционная база данных (relational database) хранит данные в виде таблиц. Реляционная СУБД (relational DBMS — RDBMS) управляет таблицами данных и связанными структурами, которые повышают функциональность и эффективность использования таблиц. РСУБД значительно выиграли от наличия ясного определения (изобретателем реляционных баз данных был E.F. Codd) и стандартного языка (SQL [Melton-93]). Все РСУБД поддерживают основные команды SQL, посредством которых можно определять таблицы, работать с данными в этих таблицах и контролировать доступ к ним. РСУБД отличаются друг от друга поддерживаемыми типами данных, оптимизацией, доступом из программ и системными данными. Стандарт SQL постепенно охватывает все более широкие области его применения. Три главных аспекта РСУБД — структура данных, операторы и ограничения.

- **Структура данных.** Реляционная база данных представляет собой совокупность таблиц. Каждая таблица имеет конкретное количество столбцов и произвольное число строк. В каждой ячейке таблицы хранится определенное значение. На рис. 19.1 показан пример из двух таблиц. Таблица

Person (Человек) состоит из пяти столбцов и четырех строк; таблица *Company* (Компания) состоит из трех столбцов и трех строк. Полужирным выделены столбцы, в которых записывается *основной ключ* (primary key) — о том, что это такое, будет рассказано ниже. Обратите внимание, что Jack Brown нигде не работает. Значение *null* означает, что фактическое значение атрибута в данной строке неизвестно или неприменимо.

В РСУБД используются специальные методы ускорения доступа, такие как индексирование, хэширование и сортировка, потому что обычные таблицы работают слишком медленно для большинства практических задач. Эти методы реализованы прозрачно для пользователя и не видны командам записи и чтения. РСУБД самостоятельно определяет, в каких случаях для ускорения доступа следует использовать соответствующую методику, и задействует ее. Обновление вспомогательных структур при изменении содержимого таблиц также осуществляется автоматически.

Person table				
personID	lastName	fistName	address	employer
1	Smith	Jim	314 Olive St.	1001
5	Brown	Moe	722 Short St.	1002
999	Smith	Jim	1561 Main Dr.	1001
14	Brown	Jane	722 Short St.	NULL

Company table		
companyID	companyName	address
1001	Ajax Widgets	33 Industrial Dr.
1002	AAA liquors	724 Short St.
1003	Win-more Sports	1877 Broadway

Рис. 19.1. Пример базы данных из двух таблиц

Операторы. Язык SQL содержит операторы для работы с таблицами. Оператор *select* считывает данные из таблицы. Синтаксис выглядит примерно следующим образом (ключевые слова мы выделили прописными буквами):

```
SELECT списокСтолбцов
FROM списокТаблиц
WHERE условиеИстинно
```

С логической точки зрения СУБД объединяет все таблицы в одну временную таблицу (фактическая реализация гораздо более эффективна). Аргумент оператора *FROM* показывает, из каких столбцов нужно считывать данные. Условие *WHERE* позволяет выбрать определенные строки. В ответ на запрос РСУБД возвращает полученные данные. В языке SQL имеются также команды для добавления, удаления и обновления данных в таблицах. Интерактивные команды SQL ориентированы на работу с множествами. Они оперируют целыми таблицами, а не отдельными строками или значениями.

Аналог SQL для работы с приложениями позволяет оперировать конкретными строками.

- **Ограничения.** РСУБД может накладывать на данные множество ограничений, которые определяются в составе структуры базы данных. Если данные не удовлетворяют какому-либо из ограничений, РСУБД откажется сохранять их и вернет пользователю сообщение об ошибке.

Возможный или *потенциальный ключ* (candidate key) — это сочетание столбцов, уникально идентифицирующих каждую строку в таблице. Комбинация должна быть минимальной, то есть в нее должны входить только те столбцы, которые необходимы для уникальной идентификации. Ни один столбец в возможном ключе не может содержать нулевые значения.

Основной ключ (primary key) — это возможный ключ, который предпочтительнее всего использовать для обращения к записям таблицы. Основным ключ у каждой таблицы может быть только один, и обычно он в ней имеется. На рис. 19.1 мы выделили основные ключи полужирным шрифтом.

Внешний ключ (foreign key) — это ссылка на возможный ключ (обычно — на основной ключ) в другой таблице. С его помощью таблицы соединяются между собой. На рис. 19.1 столбец *employer* (работодатель) является внешним ключом таблицы *Person* (Человек), который ссылается на столбец *companyID* (идентификаторКомпании) в таблице *Company* (Компания). Недопустимо было бы установить значение *employer* для *Мое Brown* равным 1004, потому что в таблице *Company* это значение не определено. Если строку *Ajax Widgets* удалить из таблицы *Company*, придется также удалить обе строки человека *Jim Smith* или установить для них значение *employer* = *null*. Итак, связи между внешними и основными ключами служат для переходов между таблицами.

19.1.3. Нормальные формы

Нормальная форма (normal form) — это правило формирования таблиц реляционных баз данных, помогающее повысить согласованность их содержимого. Таблицы, удовлетворяющие нормальным формам высокого уровня, хранят меньшее количество избыточных или противоречивых данных. Разработчики могут нарушать нормальные формы по каким-либо веским причинам, например для повышения производительности базы данных, которая чаще считывается, чем обновляется. Ослабление требований называется *денормализацией* (denormalization). Очень важно стараться как можно точнее следовать нормальным формам и нарушать их только тогда, когда это действительно необходимо.

Нормальные формы впервые стали использоваться в 70–80-х годах XX века. В те времена разработчики формировали базы данных, создавая списки требуемых полей, которые приходилось разбивать на осмысленные группы перед сохранением их в базы данных. Для этого и служили нормальные формы. С их помощью поля разбиваются на группы на основании зависимостей между этими полями.

К сожалению, зависимости слишком легко пропустить. Если это происходит, структура базы данных получается ущербной.

Модели UML предоставляют более удобный способ для подготовки баз данных. Вместо того чтобы начинать с отдельных полей, разработчики мыслят сразу в терминах больших групп полей, то есть классов. Модели UML не исключают применения нормальных форм. Формы применимы к данным независимо от подхода, использовавшегося в процессе разработки.

Однако моделирование на языке UML делает ненужной проверку нормальных форм. Если разработчику удастся построить цельную модель, она заведомо будет удовлетворять нормальным формам. Обратное также верно: плохая модель не будет удовлетворять нормальным формам. Более того, если разработчику не удастся построить цельную модель, он, скорее всего, не сможет найти все внутренние зависимости между полями данных, необходимые для построения правильных нормальных форм. Модель строить не так сложно, как искать зависимости.

Вывод таков: разработчик может проверить нормальные формы после завершения моделирования, но такая проверка не является необходимой.

19.1.4. Выбор СУБД

Чтобы построить свое приложение, вы должны выбрать конкретную СУБД. Основные функции определены стандартом SQL, поэтому вам остается выбрать поставщика РСУБД исходя из прагматических соображений.

- **Доля на рынке.** Основными игроками являются Oracle, IBM и Microsoft. Смогут ли другие компании бороться с ними, сказать сложно. Вы можете также рассмотреть возможность использования РСУБД с открытым кодом, например MySQL или PostgreSQL.
- **Производитель и поддержка третьих фирм.** РСУБД — это значительное вложение средств для компании; поддержка РСУБД требует постоянного взаимодействия с производителем или сторонними фирмами.
- **Другие приложения.** Вы сократите расходы на администрирование и лицензирование, если будете использовать продукты того же производителя для других приложений.

Выпуская новый продукт, основные производители ненадолго вырываются вперед в соревновании с конкурентами, но затем новую версию выпускает кто-то другой. В среднем функции и производительность РСУБД от разных поставщиков отличаются друг от друга не слишком значительно, поэтому ориентироваться на них при выборе продукта бессмысленно.

19.2. Сокращенная модель банкомата

На рис. 19.2 показана часть модели банкомата, которая будет использована в этой главе в качестве примера. Мы добавили классы *CheckingAccount* (ТекущийСчет) и *SavingsAccount* (НакопительныйСчет), чтобы иметь возможность обсудить вопросы, связанные с обобщением. Кроме того, мы добавили класс *Address* (Адрес).

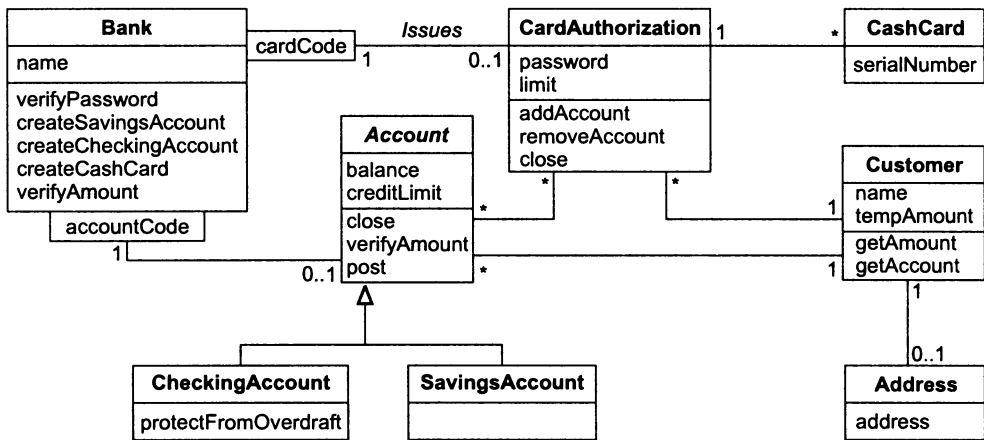


Рис. 19.2. Сокращенная модель банкомата для реализации

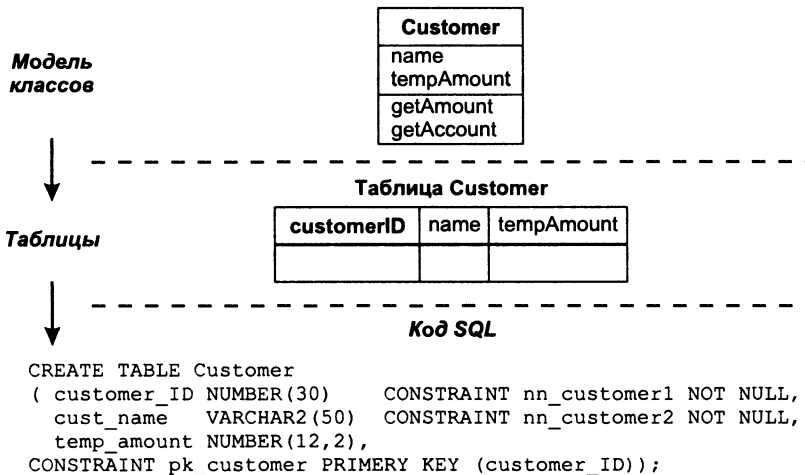
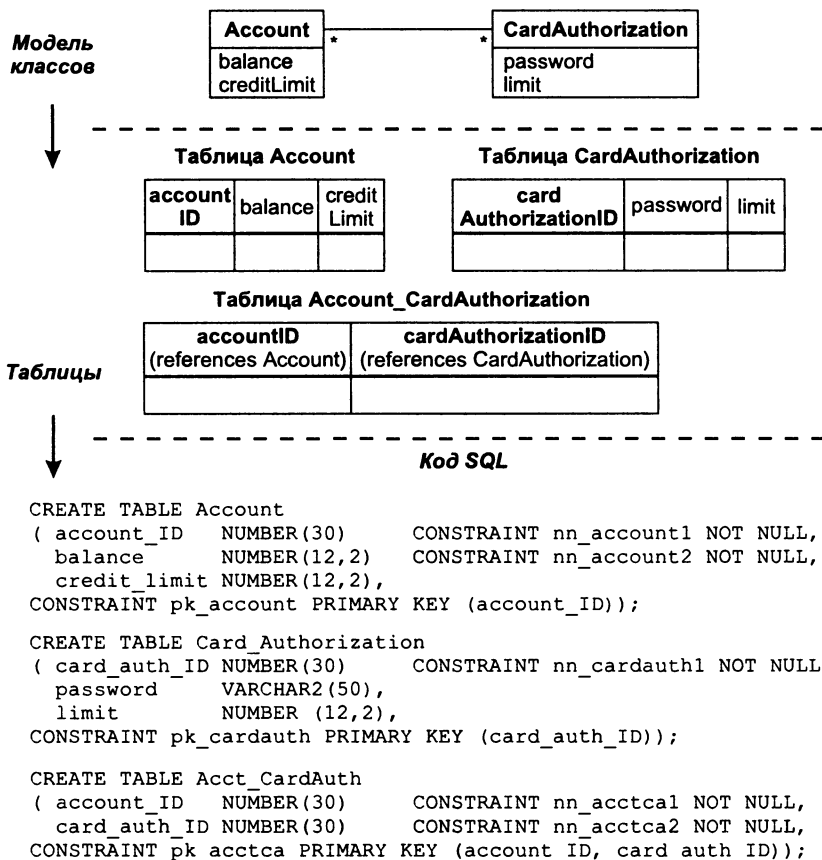
19.3. Реализация структуры — основы

Модель классов достаточно легко перевести в код на языке SQL. Для этого существуют автоматические средства, но вам все равно нужно знать правила этого преобразования. Тогда вы сможете понять, что именно делают ваши программы, и сможете проверить результаты их работы. РСУБД обеспечивают хорошую поддержку классов и ассоциаций, но не поддерживают наследование, поэтому вам придется использовать обходные методики. Реализация структуры осуществляется в приведенной ниже последовательности.

- 1. Реализация классов (раздел 19.3.1).
- 2. Реализация ассоциаций (раздел 19.3.2).
- 3. Реализация обобщений (раздел 19.3.4).
- 4. Реализация индивидуальности (раздел 19.3.5).

19.3.1. Классы

Чаще всего каждый класс отображается в свою собственную таблицу, а каждый его атрибут становится столбцом этой таблицы (рис. 19.3). Можно добавить столбцы для идентификатора объекта и для ассоциаций (об этом мы расскажем чуть ниже). Полужирным шрифтом мы выделили основной ключ. Ключевые слова языка SQL выделены заглавными буквами. Обратите внимание, что операции не воздействуют на структуру таблицы. Мы выбрали типы данных и ограничения, которые кажутся разумными в свете нашей задачи. Во всех примерах мы будем использовать синтаксис Oracle. *nn_customer1* и *nn_customer2* — названия, которые мы дали ограничениям на значения полей (эти поля не могут быть нулевыми). *pk_customer* — название ограничения, которое накладывается на основной ключ.

**Рис. 19.3.** Реализация классов**Рис. 19.4.** Реализация ассоциаций типа «многие-ко-многим»

19.3.2. Ассоциации

Правила реализации ассоциаций учитывают их кратность.

- **«Многие-ко-многим»**. Реализуйте ассоциацию в виде таблицы и сделайте основным ключом этой таблицы комбинацию основных ключей участвующих в ассоциации классов (рис. 19.4). Если у ассоциации есть атрибуты, они станут дополнительными столбцами этой таблицы.
- **«Один-ко-многим»**. Каждый класс у полюса «один» становится внешним ключом в таблице класса с кратностью «много» (рис. 19.5). Если бы у полюса «один» было имя, мы могли бы использовать его в качестве имени внешнего ключа. Мы предполагаем, что порядковый номер (*serialNumber*) банковской карты (*CashCard*) уникален.

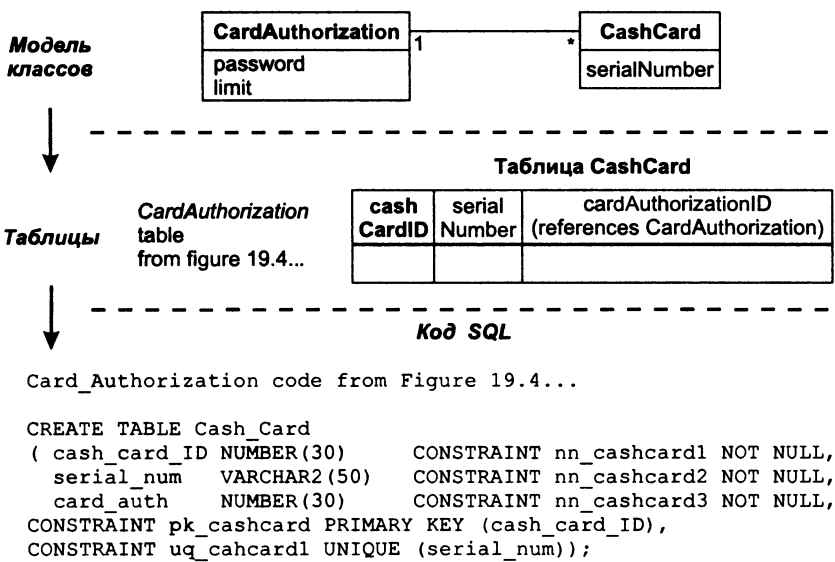
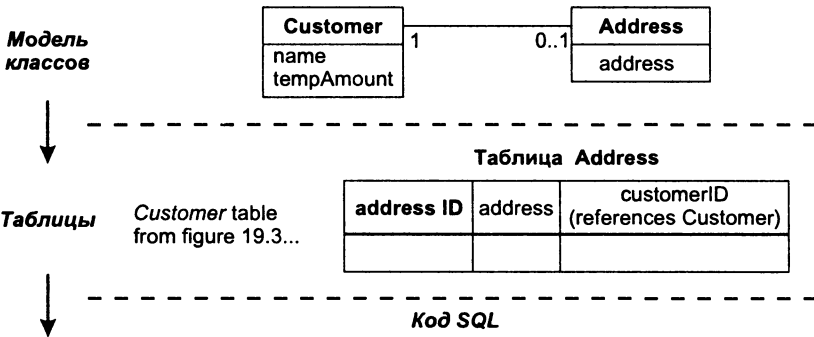


Рис. 19.5. Реализация ассоциаций типа «один-ко-многим»

- **«Один-к-одному»**. Такая кратность встречается тностредко. В этом случае внешний ключ можно поместить в таблицу любого класса (рис. 19.6).
- **N-арные ассоциации**. Тоже встречаются редко. Можно рассматривать их как ассоциации типа «многие-ко-многим» и создавать для них отдельную таблицу. Чаще всего основной ключ таблицы n-арной ассоциации сочетает в себе основные ключи участвующих в ассоциации таблиц.
- **Классы ассоциаций**. Это ассоциация, которая одновременно является классом. Правильно описать зависимости будет проще, если такую ассоциацию сделать отдельной таблицей, независимо от ее края.
- **Квалифицированные ассоциации**. Для них действуют те же правила, что и для таких же ассоциаций без квалификатора. Поэтому на рис. 19.7 мы поступили с квалифицированной ассоциацией, как с ассоциацией типа

«один-ко-многим» (у каждого банка может быть много счетов). *ckn* (n — номер) обозначает возможный ключ. У многих квалифицированных ассоциаций квалификатор входит в состав возможного ключа.

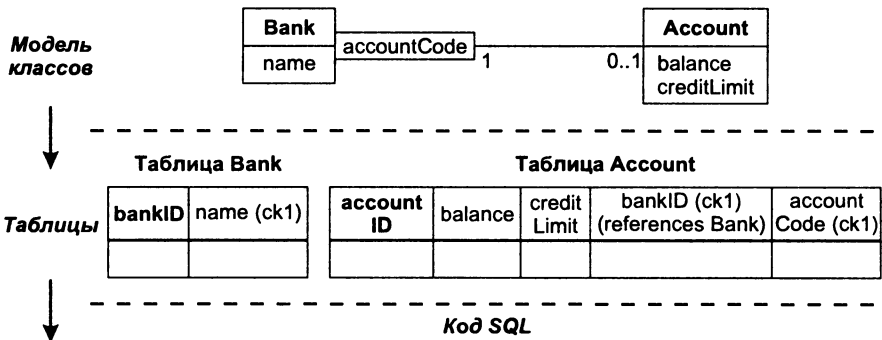


Customer code from Figure 19.3...

```

CREATE TABLE Address
( address_ID NUMBER(30) CONSTRAINT nn_address1 NOT NULL,
  address VARCHAR2(200) CONSTRAINT nn_address2 NOT NULL,
  customer_ID NUMBER(30) CONSTRAINT nn_address3 NOT NULL,
  CONSTRAINT pk_address PRIMARY KEY (address_ID));
  
```

Рис. 19.6. Реализация ассоциации типа «один-к-одному»



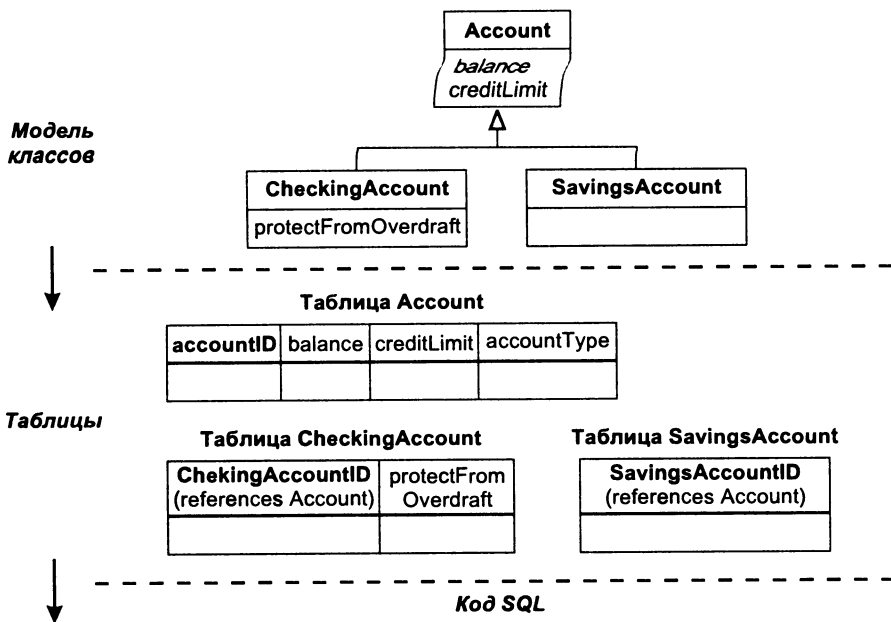
```

CREATE TABLE Bank
( bank_ID NUMBER(30) CONSTRAINT nn_bank1 NOT NULL,
  bank_name VARCHAR2(50) CONSTRAINT nn_bank2 NOT NULL,
  CONSTRAINT pk_bank PRIMARY KEY (bank_ID),
  CONSTRAINT uq_bank1 UNIQUE (bank_name));

CREATE TABLE Account
( account_ID NUMBER(30) CONSTRAINT nn_account1 NOT NULL,
  balance NUMBER(12,2) CONSTRAINT nn_account2 NOT NULL,
  credit_limit NUMBER(12,2)
  bank_ID NUMBER(30) CONSTRAINT nn_account3 NOT NULL,
  account_code VARCHAR2(50) CONSTRAINT nn_account4 NOT NULL,
  CONSTRAINT pk_account PRIMARY KEY (account_ID),
  CONSTRAINT uq_accont1 UNIQUE (bank_ID, account_code));
  
```

Рис. 19.7. Реализация квалифицированной ассоциации

множества обобщений (*accountType*), указывает таблицу подклассов для каждой записи суперкласса. Многоуровневое наследование реализуется последовательно, по одному уровню за раз.



```

CREATE TABLE Account
( account_ID    NUMBER(30)          CONSTRAINT nn_account1 NOT NULL,
  balance       NUMBER(12,2)        CONSTRAINT nn_account2 NOT NULL,
  credit_limit  NUMBER(12,2),
  account_type  VARCHAR2(20)        CONSTRAINT nn_account3 NOT NULL,
  CONSTRAINT pk_account PRIMARY KEY (account_ID));

CREATE TABLE Checking_Account
( chk_acct_ID   NUMBER(30)          CONSTRAINT nn_chkacct1 NOT NULL,
  protect_odrft VARCHAR2(1)         CONSTRAINT nn_chkacct2 NOT NULL,
  CONSTRAINT pk_chkacct PRIMARY KEY (chk_acct_ID));

CREATE TABLE Savings_Account
( sav_acct_ID   NUMBER(30)          CONSTRAINT nn_savacct1 NOT NULL,
  CONSTRAINT pk_savacct PRIMARY KEY (sav_acct_ID));
  
```

Рис. 19.8. Реализация обобщения

Обратите внимание, что названия основного ключа на нашем рисунке разные, но объект должен иметь одинаковые значения основного ключа во всей иерархии наследования. Поэтому «текущий счет Джо» может иметь одну строку в таблице *Account* с идентификатором *account_ID* = 101 и другую строку в таблице *Checking_Account* с идентификатором *chk_acct_ID* = 101. Мы предпочитаем связывать имена идентификаторов с именами классов (*account_ID*, *chk_acct_ID*, *sav_Acct_ID*), а не использовать одно и то же имя для всех таблиц, служащих для реализации обобщения. Это упрощает работу с многоуровневыми обобщениями.

Обратите внимание, что исключать из этой схемы подклассы, не имеющие атрибутов (подобно *SavingsAccount* на рис. 19.8), не следует. Достижимое повышение производительности редко оказывается значительным, но зато при этом усложняется обеспечение зависимостей внешних ключей (см. раздел 19.4.1).

- **Множественное наследование.** Для его реализации придется создавать отдельную таблицу суперклассов и отдельную таблицу подклассов. Для множественного наследования с перекрывающимися классами нужно создавать одну таблицу для каждого суперкласса, одну таблицу для каждого подкласса и одну таблицу для обобщения.

19.3.4. Индивидуальность

В каждой таблице, за редкими исключениями, должен быть основной ключ. В качестве этого ключа мы использовали индивидуальность объекта. Мы предпочитаем именно такой подход, хотя раньше никак не комментировали это, но нужно отметить, что в литературе по базам данных часто встречается и другой подход. На рис. 19.9 сравниваются два возможных варианта реализации индивидуальности объектов.

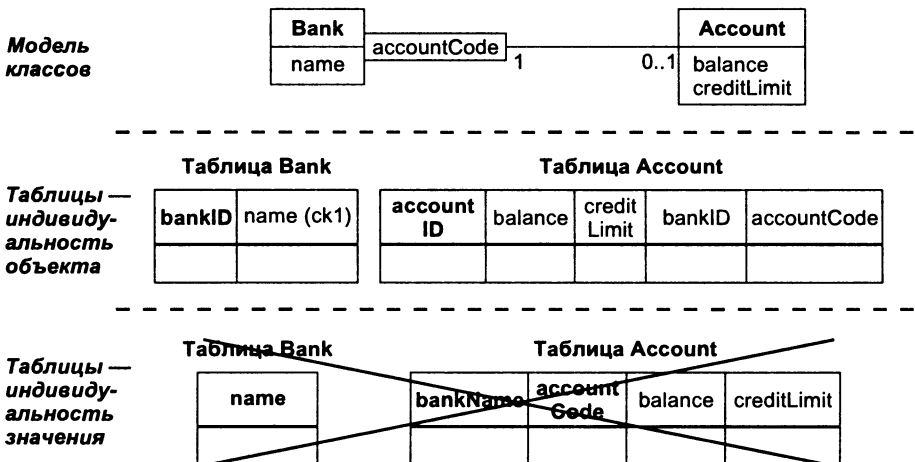


Рис. 19.9. Индивидуальность объекта или значения

- **Индивидуальность объекта.** В каждую таблицу каждого класса добавляется произвольный численный атрибут (идентификатор объекта), который делается основным ключом этой таблицы. Основным ключ каждой таблицы ассоциации состоит из идентификаторов участвующих в ней классов.

Идентификатор объекта представляет собой единственный атрибут, занимающий небольшой объем памяти и однородный в размере. Большинство РСУБД работают с идентификаторами особенно эффективно. Однако использование идентификаторов объектов затрудняет чтение базы данных в процессе отладки и обслуживания. Идентификаторы затрудняют и слияние баз данных, потому что их значения в разных базах могут совпадать, а в таком случае требуется переназначение идентификаторов. Эти числа не должны быть видны пользователям.

- **Индивидуальность значения.** Каждый объект идентифицируется некоторой комбинацией реальных атрибутов. Основным ключ таблицы ассоциации в этом случае также состоит из основных ключей участвующих в ассоциации классов.

Такая индивидуальность обладает своими преимуществами и недостатками. Основные ключи имеют собственный смысл, что облегчает отладку базы данных. Однако их может быть сложно изменить. Изменение может распространиться на другие таблицы. Для некоторых классов трудно найти естественные идентификаторы, связанные с реальным миром.

Мы рекомендуем использовать индивидуальность объектов для приложений РСУБД. Однородность и простота перевешивают возможные дополнительные затраты на отладку. Более того, индивидуальность объектов согласуется с принципами объектной ориентированности: объекты обладают собственной индивидуальностью, никак не связанной с их свойствами. Объектно-ориентированные языки реализуют индивидуальность при помощи указателей или таблиц поиска указателей, а идентификатор — аналогичная по смыслу конструкция для баз данных.

19.3.5. Основные правила реализации РСУБД

В табл. 19.1 перечислены основные правила реализации структуры модели в РСУБД. Этим правилам следуют большинство систем для генерации баз данных из моделей UML.

Таблица 19.1. Основные правила реализации РСУБД

Концепция	Конструкция UML	Рекомендуемые правила реализации
Класс	Класс	Отобразить каждый класс в таблицу, а каждый атрибут — в столбец этой таблицы

Концепция	Конструкция UML	Рекомендуемые правила реализации
Ассоциация (названия полюсов становятся названиями внешних ключей)	Многие-ко-многим	Использовать отдельную таблицу
	Один-ко-многим	Использовать встроенный внешний ключ
	Один-к-одному	
	N-арная	Использовать отдельную таблицу
Агрегация	Ассоциация-класс	
	Квалифицированная	То же, что и для неквалифицированной
	Агрегация	То же, что и для ассоциации
Обобщение	Композиция	
	Одинокое наследование	Создание отдельных таблиц для суперкласса и каждого из подклассов
	Раздельное множественное наследование	
	Перекрывающееся множественное наследование	То же, что и при раздельном множественном наследовании, но с таблицей самого обобщения, связывающей записи суперкласса и подкласса

19.4. Реализация структуры — дополнительные вопросы

В предыдущем разделе мы рассказывали о том, каким образом следует определять таблицы базы данных для каждой из конструкций модели классов UML. В этом разделе мы рассмотрим дополнительные аспекты, связанные с повышением производительности и обеспечением качества данных. В ходе реализации модели вам предстоит выполнить следующие действия.

- Реализовать внешние ключи (раздел 19.4.1).
- Реализовать проверочные ограничения (раздел 19.4.2).
- Реализовать индексы (раздел 19.4.3).
- Рассмотреть возможные представления (раздел 19.4.4).

19.4.1. Внешние ключи

Внешние ключи возникают в процессе реализации ассоциаций и обобщений. После определения внешнего ключа РСУБД гарантирует отсутствие повисших ссылок: РСУБД откажется выполнить обновление, результатом которого может стать повисшая ссылка. Поскольку приложение может быть уверено в корректности внешних ключей, оно может не выполнять дополнительную проверку их правильности.

Многие РСУБД могут обеспечивать распространение эффектов удалений и обновлений, влияющих на внешние ключи. Если вы используете индивидуальность объектов (в соответствии с нашими рекомендациями), необходимости распространения обновлений нет, потому что идентификаторы никогда не изменяются. Однако полезно бывает определить реакцию системы на удаление, о чем мы расскажем ниже.

Для каждой таблицы подкласса при реализации обобщений необходимо указать параметр *on delete cascade* (каскадное удаление). В листинге 19.7 приведены определения внешних ключей, которые необходимо добавить к листингу 19.6. Вспомните, что обобщение структурирует описание объекта: каждый уровень обобщения дает часть этого описания. Приложение должно объединить записи суперкласса и подкласса, чтобы получить объект в целом. Удаление суперкласса приводит к каскадному удалению соответствующей записи подкласса, и это удаление распространяется вниз по всем уровням обобщений.

Листинг 19.7. Внешние ключи для обобщения — реализация каскадного удаления

```
ALTER TABLE Checking_Account ADD CONSTRAINT fk_chkacct1
    FOREIGN KEY chk_acct_ID
REFERENCES Account ON DELETE CASCADE;
```

```
ALTER TABLE Savings_Account ADD CONSTRAINT fk_savacct1
    FOREIGN KEY sav_acct_ID
REFERENCES Account ON DELETE CASCADE
```

Хорошо было бы иметь возможность распространять удаление записи подкласса вверх к таблице суперкласса, однако SQL не поддерживает подобную операцию. Вам придется написать программный код, который будет обеспечивать распространение удаления вверх по иерархии наследования.

Внешние ключи необходимо определять и для воплощения ассоциаций, как показано в листинге 19.8. Действие при удалении зависит от смысла модели. Например, если мы хотим, чтобы удаление записи клиента приводило к удалению соответствующей записи адреса — нужно указывать параметр *on delete cascade* (первый оператор в листинге 19.8). Если же мы хотим запретить удаление клиента, у которого есть счета (чтобы избежать случайной потери важных данных), нужно указать параметр *on delete no action* (никаких действий при попытке удаления). В этом случае, чтобы удалить клиента, пользователь системы должен будет сначала удалить все счета этого клиента. Синтаксис Oracle позволяет просто не указывать действие при операции *delete* (второй оператор в листинге 19.8).

Листинг 19.8. Внешние ключи для ассоциации

```
ALTER TABLE Address ADD CONSTRAINT fk_address1
    FOREIGN KEY customer_ID
REFERENCES Customer ON DELETE CASCADE;
```

```
ALTER TABLE Account ADD CONSTRAINT fk_account2
```

```
FOREIGN KEY customer_ID
REFERENCES Customer;
```

19.4.2. Проверка ограничений

SQL позволяет устанавливать общие ограничения, которые определяют допустимые значения перечислимых типов. Это особенно полезно при реализации названия набора обобщений. В листинге 19.9 мы добавляем такое ограничение к листингу 19.6.

Листинг 19.9. Обязательное задание названия набора обобщений

```
ALTER TABLE Account ADD CONSTRAINT enum_account1
CHECK (account_type IN ('Checking_Account', 'Savings_Account'));
```

19.4.3. Индексы

В большинстве РСУБД создание индексов является побочным эффектом ограничения *unique* на основной и возможный ключи. Индекс — это структура данных, отображающая значения из одного столбца в строки таблицы базы данных. Вам следует создать индекс для каждого внешнего ключа, на который не действуют ограничения какого-либо основного или потенциального ключа. Например, наличие основного ключа *Acct_CardAuth* (рис. 19.4) заставляет РСУБД построить индекс, что обеспечивает быстрый доступ к *account_ID* и к комбинации *account_ID*+*card_auth_ID*. Дополнительный индекс для *card_auth_ID* (листинг 19.10) обеспечит столь же быстрый доступ к этому столбцу в отдельности.

Наличие индексов критически важно для обеспечения быстродействия базы данных. Индексирование внешних ключей обеспечивает быстрое комбинирование таблиц. Отсутствие индексов замедляет работу РСУБД на несколько порядков. Индексы внешних ключей должны быть неотъемлемой частью базы данных, потому что их несложно включить, и нет никаких причин, чтобы этого не делать.

Администратор базы данных может определить дополнительные индексы для ускорения частых запросов, а также использовать специальные механизмы настройки, предусмотренные в той или иной СУБД.

Листинг 19.10. Определение индекса

```
CREATE INDEX index_acctcal ON Acct_CardAuth (card_auth_ID);
```

19.4.4. Представления

Вы можете определить представление для каждого подкласса, чтобы консолидировать унаследованные данные и упростить доступ к объекту. Представление — это таблица, динамически вычисляемая РСУБД. В листинге 19.11 приведен пример создания представления для подкласса *CheckingAccount*. Вы можете свободно считывать любые данные из представления, однако запись данных через представление РСУБД поддерживают лишь частично. Ограничения зависят от конкретного продукта.

Листинг 19.11. Создание представления

```
CREATE VIEW view_checking_account AS
  SELECT chk_acct_ID, balance, credit_limit, protect_odrft
  FROM Account A, Checking_Account CA
  WHERE A.account_ID = CA.chk_acct_ID;
```

19.4.5. Дополнительные правила реализации моделей UML в РСУБД

В табл. 19.2 приведены дополнительные правила, касающиеся реализации моделей классов в РСУБД. Эти правила включены в большинство систем автоматической генерации баз данных из моделей.

Таблица 19.2. Дополнительные правила реализации

Концепция	Правило реализации
Класс	Определить контрольное ограничение для каждого атрибута с перечислимым типом
Ассоциация, агрегация, композиция	Вводить внешние ключи. Указать действие при удалении в зависимости от смысла модели: on delete cascade или on delete no action
	Определить контрольное ограничение для каждого атрибута с перечислимым типом
	Определить индексы для всех встроенных внешних ключей, на которые не распространяются ограничения, наложенные на основные и возможные ключи
Обобщение	Вводить внешние ключи. Указать действие при удалении в зависимости от смысла модели: on delete cascade или on delete no action
	Определить проверочное ограничение для каждого набора обобщений
	Рассмотреть возможность создания представления для консолидации унаследованных данных и для упрощения чтения объектов

19.5. Реализация структуры из примера с банкоматом

На рис. 19.10 показаны все таблицы для модели на рис. 19.2. В листинге 19.12 приведен код SQL, который создает соответствующие структуры в базе данных Oracle. Каждый оператор *sequence* создает счетчик, используемый для выделения идентификаторов. Например, *seq_bank* используется для выделения *bank_ID* при создании очередного объекта *Bank*. Мы упорядочили код по смыслу, поэтому его придется изменить, чтобы выполнить в реальной базе данных. Сначала выполняются операции создания таблиц и счетчиков, затем создания индексов и наконец — операции изменения таблиц. Мы не включили в листинг операции создания представлений для *CheckingAccount* и *SavingsAccount*.

Таблица Bank

bankID	name (ck1)

Таблица Customer

customerID	name	tempAmount

Таблица Account

account ID	balance	credit Limit	bankID (ck1) (references Bank)	account Code (ck1)	account Type	customerID (references Customer)

Таблица CardAuthorization

card AuthorizationID	password	limit	bankID (ck1) (references Bank)	card Code (ck1)	customerID (references Customer)

Таблица CashCard

cashCardID	serialNumber	cardAuthorizationID (references CardAuthorization)

Таблица Account_CardAuthorization

accountID (references Account)	cardAuthorizationID (references CardAuthorization)

Таблица SavingsAccount

savingsAccountID (references Account)

Таблица CheckingAccount

checkingAccountID (references Account)	protectFrom Overdraft

Таблица Address

address ID	address	customerID (references Customer)

Рис. 19.10. Таблицы РСУБД для сокращенной модели банкомата

Листинг 19.12. Код SQL, реализующий модель классов банкомата

```

CREATE TABLE Bank
( bank_ID          NUMBER(30)          CONSTRAINT nn_bank1 NOT NULL,
  bank_name        VARCHAR2(50)        CONSTRAINT nn_bank2 NOT NULL,
  CONSTRAINT pk_bank PRIMARY KEY (bank_ID),
  CONSTRAINT uq_bank1 UNIQUE (bank_name));

CREATE SEQUENCE seq_bank;

CREATE TABLE Customer
( customer_ID      NUMBER(30)          CONSTRAINT nn_customer1 NOT NULL,
  cust_name        VARCHAR2(50)        CONSTRAINT nn_customer2 NOT NULL,
  temp_amount      NUMBER(12,2),
  CONSTRAINT pk_customer PRIMARY KEY (customer_ID));

```

Листинг 19.12 (продолжение)

```

CREATE SEQUENCE seq_customer;

CREATE TABLE Card_Authorization
( card_auth_ID      NUMBER(30)          CONSTRAINT nn_cardauth1 NOT NULL,
  password          VARCHAR2(50),
  limit             NUMBER(12,2),
  bank_ID           NUMBER(30)          CONSTRAINT nn_cardauth2 NOT NULL,
  card_code         VARCHAR2(50)        CONSTRAINT nn_cardauth3 NOT NULL,
  customer_ID       NUMBER(30)          CONSTRAINT nn_cardauth4 NOT NULL,
  CONSTRAINT pk_cardauth PRIMARY KEY (card_auth_ID),
  CONSTRAINT uq_cardauth1 UNIQUE (bank_ID, card_code));

CREATE SEQUENCE seq_cardauth;

CREATE INDEX index_cardauth1 ON Card_Authorization (customer_ID);

ALTER TABLE Card_Authorization ADD CONSTRAINT fk_cardauth1
  FOREIGN KEY bank_ID
  REFERENCES Bank;

ALTER TABLE Card_Authorization ADD CONSTRAINT fk_cardauth2
  FOREIGN KEY customer_ID
  REFERENCES Customer;

CREATE TABLE Checking_Account
( chk_acct_ID       NUMBER(30)          CONSTRAINT nn_chkacct1 NOT NULL,
  protect_odrft     VARCHAR2(1)         CONSTRAINT nn_chkacct2 NOT NULL,
  CONSTRAINT pk_chkacct PRIMARY KEY (chk_acct_ID));

ALTER TABLE Checking_Account ADD CONSTRAINT fk_chkacct1
  FOREIGN KEY chk_acct_ID
  REFERENCES Account ON DELETE CASCADE;

ALTER TABLE Checking_Account ADD CONSTRAINT enum_chkacct1
  CHECK (protect_odrft IN ('Y','N'));

CREATE TABLE Account
( account_ID        NUMBER(30)          CONSTRAINT nn_account1 NOT NULL,
  balance           NUMBER(12,2)        CONSTRAINT nn_account2 NOT NULL,

  credit_limit      NUMBER(12,2),
  bank_ID           NUMBER(30)          CONSTRAINT nn_account3 NOT NULL,
  account_code      VARCHAR2(50)        CONSTRAINT nn_account4 NOT NULL,
  account_type      VARCHAR2(20)        CONSTRAINT nn_account5 NOT NULL,
  customer_ID       NUMBER(30)          CONSTRAINT nn_account6 NOT NULL,
  CONSTRAINT pk_account PRIMARY KEY (account_ID),
  CONSTRAINT uq_account1 UNIQUE (bank_ID, account_code));

CREATE SEQUENCE seq_account;

CREATE INDEX index_account1 ON Account (customer_ID);

ALTER TABLE Account ADD CONSTRAINT fk_account1
  FOREIGN KEY bank_ID
  REFERENCES Bank;

```

```
ALTER TABLE Account ADD CONSTRAINT fk_account2
    FOREIGN KEY customer_ID
    REFERENCES Customer;

ALTER TABLE Account ADD CONSTRAINT enum_account1
    CHECK (account_type IN ('Checking_Account', 'Savings_Account'));

CREATE TABLE Acct_CardAuth
( account_ID          NUMBER(30)          CONSTRAINT nn_acctca1 NOT NULL,
  card_auth_ID        NUMBER(30)          CONSTRAINT nn_acctca2 NOT NULL,
  CONSTRAINT pk_acctca PRIMARY KEY (account_ID, card_auth_ID));

CREATE INDEX index_acctca1 ON Acct_CardAuth (card_auth_ID);

ALTER TABLE Acct_CardAuth ADD CONSTRAINT fk_acctca1
    FOREIGN KEY account_ID
    REFERENCES Account;

ALTER TABLE Acct_CardAuth ADD CONSTRAINT fk_acctca2
    FOREIGN KEY card_auth_ID
    REFERENCES Card_Authorization;

CREATE TABLE Savings_Account
( sav_acct_ID         NUMBER(30)          CONSTRAINT nn_savacct1 NOT NULL,
  CONSTRAINT pk_savacct PRIMARY KEY (sav_acct_ID));

ALTER TABLE Savings_Account ADD CONSTRAINT fk_savacct1
    FOREIGN KEY sav_acct_ID
    REFERENCES Account ON DELETE CASCADE;

CREATE TABLE Cash_Card
( cash_card_ID        NUMBER(30)          CONSTRAINT nn_cashcard1 NOT NULL,
  serial_num          VARCHAR2(50)        CONSTRAINT nn_cashcard2 NOT NULL,
  card_auth_ID        NUMBER(30)          CONSTRAINT nn_cashcard3 NOT NULL,
  CONSTRAINT pk_cashcard PRIMARY KEY (cash_card_ID),
  CONSTRAINT uq_cashcard1 UNIQUE (serial_num));

CREATE SEQUENCE seq_cashcard;
CREATE INDEX index_cashcard1 ON Cash_Card (card_auth_ID);

ALTER TABLE Cash_Card ADD CONSTRAINT fk_cashcard1
    FOREIGN KEY card_auth_ID
    REFERENCES Card_Authorization;

CREATE TABLE Address
( address_ID          NUMBER(30)          CONSTRAINT nn_address1 NOT NULL,
  address             VARCHAR2(200)       CONSTRAINT nn_address2 NOT NULL,
  customer_ID         NUMBER(30)          CONSTRAINT nn_address3 NOT NULL,
  CONSTRAINT pk_address PRIMARY KEY (address_ID));

CREATE SEQUENCE seq_address;

CREATE INDEX index_address1 ON Address (customer_ID);

ALTER TABLE Address ADD CONSTRAINT fk_address1
    FOREIGN KEY customer_ID
    REFERENCES Customer ON DELETE CASCADE;
```

19.6. Реализация функциональности

Обычно базы данных используются для построения приложений. Самое важное при реализации — это структура базы данных, но функциональность приложения (пользовательский интерфейс, сложная логика и другое поведение) тоже очень важна. Модель UML — это первый шаг к объединению базы данных и программы, потому что она дает единый подход к обоим аспектам. Тем не менее вы все равно должны рассмотреть дополнительные вопросы, связанные с реализацией функциональности.

- Связь языка программирования с базой данных (раздел 19.6.1).
- Преобразование данных (раздел 19.6.2).
- Инкапсуляция и оптимизация запросов (раздел 19.6.3).
- Использование кода SQL (раздел 19.6.4).

19.6.1. Связь языка программирования с базой данных

Стили реляционных баз данных и стандартных языков программирования существенно отличаются друг от друга, что затрудняет взаимодействие между ними. Базы данных объявляются декларативно: разработчики описывают данные, которые им нужны, а не процедуры получения этих данных. Большинство языков программирования используют императивное описание и требуют выражения логики приложения в виде последовательности шагов. Для объединения баз данных с языками программирования существует множество методик, и очень важно иметь представление о всех доступных средствах.

- **Препроцессор и постпроцессор.** Препроцессоры и постпроцессоры полезны для пакетных приложений. Основная идея проста: создать запрос для базы данных, получить входной файл, запустить приложение, проанализировать его вывод и сохранить результаты в базе данных. Недостаток состоит в том, что взаимодействие с базой данных посредством промежуточных файлов может быть достаточно неудобным. Препроцессор должен запросить всю необходимую информацию, прежде чем вызывать приложение. Столь же трудной может быть обработка выходных файлов приложения, особенно в сложном формате. Эта методика полезна для старых программ или для сертифицированного программного обеспечения, которое не может быть модифицировано.
- **Файлы сценариев.** Иногда вам нужно просто создать файл с командами РСУБД. Например, ввод команды `@filename` в интерактивном приглашении SQL (SQL Plus) системы Oracle запускает выполнение команд, содержащихся в файле `filename`. Разработчики могут использовать язык сценариев операционной системы для запуска множества сценариев и для управления их выполнением. Сценарии полезны для простого взаимодействия с базой данных, например для создания структур. Они полезны и для прототипирования.

- **Встроенные команды СУБД.** Еще одна методика состоит во включении команд SQL в код приложения. Она пропагандируется во многих книгах по базам данных. К сожалению, получающиеся в результате программы довольно сложно читать и обслуживать. Основная проблема состоит в том, что базы данных концептуально отличаются от большинства языков программирования. Мы не рекомендуем прибегать к этому методу.
- **Интерфейс программирования приложений (API).** Более удачная альтернатива предыдущему методу — инкапсулировать запросы на чтение и запись из базы данных внутри специальных методов приложения, которые все вместе будут предоставлять интерфейс к базе данных. Эти методы можно написать на фирменном языке доступа к базе данных, таком как Oracle PL/SQL, или на стандартном языке типа ODBC или JDBC. Интерфейс изолирует взаимодействие с базой данных от остальных частей приложения. API помогает разделить задачи управления данными, логику приложения и пользовательский интерфейс, что согласуется с принципами инкапсуляции.
- **Хранимые процедуры.** *Хранимая процедура* (stored procedure) — это программный код, хранящийся в базе данных. Некоторые РСУБД требуют использования таких процедур для достижения максимальной эффективности. Эти процедуры позволяют приложениям совместно использовать некоторую функциональность. Однако не рекомендуется помещать функциональность, относящуюся только к конкретному приложению, внутрь базы данных, потому что ее сможет использовать любое приложение, а это противоречит принципам инкапсуляции. Правила работы с хранимыми процедурами разные в разных СУБД.
- **Язык четвертого поколения.** *Язык четвертого поколения* (fourth-generation language — 4GL) — это каркас для приложений баз данных, обеспечивающий вывод на экран, простейшие расчеты и формирование отчетов. Языки четвертого поколения широко распространены и позволяют значительно сократить сроки разработки приложения. Они особенно удобны для простых приложений и для прототипирования. Для приложений со сложной логикой они не пригодны.
- **Общий уровень.** Общий уровень скрывает базу данных и предоставляет простые команды для доступа к данным (такие как *getRecordGivenKey* и *writeRecord*) [Blaha-98]. Вы можете писать код приложения с использованием команд этого уровня и игнорировать существование базы данных. Удачно выбранный уровень позволяет упростить программирование приложения, однако он может снизить производительность и ограничить доступ к функциональности базы данных.
- **Система, управляемая метаданными.** Приложение косвенно обращается к данным, начиная с чтения описания данных (то есть метаданных), после чего оно формирует запрос для считывания самих данных. Например, РСУБД сначала обращаются к системным таблицам, а затем к данным. Приложения, управляемые метаданными, могут быть достаточно сложны. Эта методика подходит для каркасов (см. раздел 14) и для самообучающихся приложений.

В некоторых случаях бывает полезно сочетать различные методики в одном приложении. Например, разработчик может использовать API и реализовать часть функциональности в виде хранимых процедур. В табл. 19.3 перечислены все методики связи баз данных с языками программирования.

19.6.2. Преобразование данных

Обработка данных, хранящихся в устаревших форматах, важна для создания новых приложений и для обмена данными между разными приложениями. Многие приложения оказываются довольно неудачными, поэтому переработка их данных может потребовать значительного времени и усилий. Преобразование данных имеет несколько основных аспектов.

- **Очистка данных.** Вы должны устранить ошибки в имеющихся данных. Ошибки возникают из-за неправильного ввода пользователем, ошибок в модели, ошибок в базе данных и сбоев приложения. Например, приложение могло допустить ввод адресов с некорректным почтовым индексом. Некоторая комбинация полей могла задумываться как уникальная, но если это не обеспечивалось структурой базы данных, гарантировать такую уникальность тяжело.
- **Обработка недостающих данных.** Нужно решить, что делать с недостающими данными. Можете ли вы добыть их откуда-либо еще, оценить возможные значения или просто ввести нулевые значения? Тут вам могут помочь пользователи системы.
- **Перенос данных.** Необходимость переноса данных из одного приложения в другое возникает довольно часто. Для такой операции удобно использовать стандартный язык, например XML.
- **Слияние данных.** Источники данных могут содержать перекрывающуюся информацию. Например, в одной системе могут содержаться сведения о счетах, а в другой — сведения об адресах. Новому приложению могут потребоваться данные обоих типов. Для источников со сложной структурой информации лучше предварительно создать ее модель, а затем разработать схему слияния.
- **Изменение структуры данных.** Обычно исходные структуры отличаются от целевых, поэтому данные приходится корректировать. Например, одно приложение может хранить телефонный номер в единственном поле; другое может хранить код страны, код города и местный номер отдельно в трех полях. Одинаковые по смыслу поля могут иметь разные названия, типы данных и размеры. Данные могут быть закодированы по-разному, например, пол может быть мужским или женским, М или Ж, 1 или 2 и т. д.

Обработку данных следует начинать с их загрузки во вспомогательные таблицы, отражающие структуру оригинала. Например, если старое приложение использует файлы, а новое будет работать с реляционной базой данных, разумно будет создать по одной таблице для каждого файла. Каждый столбец файла отображается в столбец соответствующей таблицы, с тем же типом и размером данных. Большинство РСУБД с легкостью могут выполнить такую операцию.

После того как данные будут загружены во вспомогательные таблицы базы данных, с ними уже можно работать посредством команд SQL. Это часто оказывается проще, чем писать программу на каком-либо другом языке. Вспомогательные таблицы дают возможность пользоваться всей мощью запросов для преобразования данных из старого формата к новому.

Достаточно часто удается найти альтернативные источники данных. Данные о клиентах можно получить из записей отдела продаж, отдела обслуживания или от внешней службы маркетинга. Для устранения избыточности следует в первую очередь загрузить данные из наиболее точного источника, затем загрузить следующий и т. д. Помещайте данные во вспомогательные таблицы, чтобы SQL имел возможность устранить уже существующие записи. В противном случае вы можете получить две записи для одного клиента и тому подобные ошибки. Этот простой подход позволяет избежать противоречий в данных, и база данных получается ориентированной на наиболее точные источники информации.

19.6.3. Инкапсуляция и оптимизация запросов

В разделе 15.10.1 мы подчеркнули важность инкапсуляции (сокрытия информации). Из этого следовало, что возможности прослеживания модели классов следует ограничивать. К сожалению, принципы инкапсуляции противоречат принципам оптимизации запросов РСУБД.

В соответствии с принципами инкапсуляции, объект должен обращаться только к непосредственно связанным с ним объектам. Обращение к прочим объектам осуществляется посредством методов промежуточных объектов. Инкапсуляция увеличивает устойчивость приложения: локальные изменения в модели приложения вызывают локальные (а не глобальные) изменения в коде.

Системы оптимизации РСУБД рассматривают запрос с логической точки зрения и генерируют эффективный план выполнения. Если запрос сформулирован максимально широко, оптимизатор обладает максимальной свободой при генерации этого плана. Максимальная производительность обычно достигается при соединении нескольких таблиц в одном запросе SQL, а не разделением логики по нескольким операторам.

Отсюда следует, что инкапсуляция повышает устойчивость приложения, но снижает возможности оптимизации. Напротив, широкая постановка запросов повышает возможности оптимизации, но при этом незначительное изменение в приложении может повлиять на множество запросов. Простого решения этого противоречия для РСУБД не существует. Возможны три различные ситуации:

- **Запутанная программа.** Если программный код достаточно сложен, а падение производительности не слишком велико, этот код следует инкапсулировать.
- **Простая программа и хорошая производительность запросов.** Запросы следует формулировать широко, если это повышает производительность РСУБД, а программный код и сами запросы не слишком сложны для формулирования и переформулирования при возможных изменениях модели классов.

- **Простая программа и низкая производительность запросов.** Как это ни странно, иногда фрагментация запросов приводит не к падению, а к повышению производительности. Идеального оптимизатора запросов не существует, иногда его приходится направлять вручную.

19.6.4. Использование кода SQL

Методы всегда можно написать на каком-нибудь языке программирования, но иногда оптимальной альтернативой этому является использование кода SQL. Опытный разработчик пишет запросы SQL быстрее, чем код программы. Более того, код SQL выполняется быстрее, содержит меньше ошибок и легче поддается расширению. Производительность SQL выигрывает от сокращения трафика взаимодействия (вычисление производится только на сервере) и от использования устойчивых алгоритмов РСУБД.

В качестве примера рассмотрим полную модель банкомата из главы 17. Предположим, что мы хотим подготовить месячный отчет по транзакциям для каждого счета. Мы могли бы опросить базу данных, а потом работать с результатами отдельных запросов. Альтернативный подход состоит в использовании команды SQL, которая сразу же создаст необходимый отчет (листинг 19.13). В нашем примере именами с двоеточиями обозначены переменные программы, передаваемые запросу SQL.

Листинг 19.13. Перенос функциональности в код SQL

```
SELECT T.date_time, U.amount, U.kind
FROM Bank B, Account A, Update U, Remote_Transaction T
WHERE B.bank_ID = A.bank_ID AND
      A.account_ID = U.account_ID AND
      U.transaction_ID = T.transaction_ID AND
      B.bank_name = :aBankName AND
      A.account_code = :anAccountCode AND
      T.date_time >= :aStartDate AND
      T.date_time <= :anEndDate
ORDER BY T.date_time;
```

19.7. Объектно-ориентированные базы данных

Объектно-ориентированная база данных представляет собой долговременное хранилище объектов, состоящих из данных и поведения. В обычных языках программирования объекты прекращают свое существование после завершения программы. В объектно-ориентированной базе данных существование объектов не ограничивается рамками выполнения программы. Объектно-ориентированная СУБД (object-oriented DBMS — OODBMS) управляет данными, программным кодом и связанными структурами, формирующими объектно-ориентированную базу данных. В отличие от РСУБД ООСУБД значительно отличаются друг от друга синтаксисом и возможностями.

ООСУБД появились на рынке относительно недавно. Коммерческое использование РСУБД началось в 70-х годах XX века, тогда как ООСУБД начали появляться только в 90-х. Разработка ООСУБД имела две основные цели.

- **Программисты недовольны РСУБД.** Многие программисты не способны понять принципы РСУБД и хотят работать с чем-то более привычным. РСУБД отвечают декларативным принципам (запросы описывают характеристики, которым должны отвечать требуемые данные), тогда как большинство языков программирования имеют императивную природу (выражаются в виде последовательности шагов). Более того, РСУБД плохо сопрягаются с большинством языков программирования, а разработчикам хочется иметь более удобный интерфейс.

Нужно отметить, что перечисленные причины не являются достаточно вескими для выбора ООСУБД при реализации проекта. РСУБД доминируют на рынке, и это положение не изменится в ближайшем будущем. Программистам не следует переходить к ООСУБД из-за разочарования в РСУБД, им лучше научиться пользоваться РСУБД. С коммерческой точки зрения РСУБД являются более зрелыми, они характеризуются надежностью, масштабируемостью и легкостью администрирования.

- **Потребность в особых функциях.** РСУБД не обладают возможностями, которые нужны некоторым специальным приложениям. ООСУБД позволяют работать с более широкими типами данных и обращаться к низкоуровневым примитивам.

Эта причина может считаться достаточной для перехода на ООСУБД. Если достаточно сложное приложение очень важно для вашего бизнеса, ООСУБД может упростить его разработку. Инженерные приложения, системы мультимедиа и искусственного интеллекта в некоторых случаях выигрывают от использования ООСУБД.

19.8. Практические рекомендации

В этом разделе мы приводим некоторые практические рекомендации по реализации объектно-ориентированного проекта в реляционной базе данных.

- **Нормальные формы.** Нормальные формы применимы к структуре базы данных независимо от подхода, использовавшегося при разработке модели. Однако если ваша объектно-ориентированная модель непротиворечива, необходимости проверять нормальные формы нет (раздел 19.1.3).
- **Классы.** Отобразите каждый класс в таблицу, а каждый атрибут класса — в столбец этой таблицы (раздел 19.3.1).
- **Ассоциации.** Для простых ассоциаций типа «один-к-одному» и «один-ко-многим» используйте встроенный внешний ключ. Для всех прочих ассоциаций создавайте отдельную таблицу (раздел 19.3.2).
- **Обобщения.** При единичном наследовании отобразите суперкласс и все его подклассы в отдельные таблицы (раздел 19.3.3.).
- **Индивидуальность.** Мы рекомендуем использовать объектную индивидуальность. Этот подход обладает существенными преимуществами (раздел 19.3.4.).

- **Внешние ключи.** Определяйте ограничения для всех внешних ключей. Для каждого подкласса укажите параметр *on delete cascade* для внешнего ключа его суперкласса. Для внешних ключей некоторых ассоциаций тоже следует указывать этот параметр, в зависимости от смысла модели (раздел 19.4.1).
- **Перечисления.** Используйте ограничения SQL для работы с перечислимыми типами (раздел 19.4.2).
- **Индексы.** Создавайте индексы для всех внешних ключей, которые не являются основными или возможными ключами. Вы можете создать дополнительные индексы для часто выполняемых запросов, а также использовать специальные механизмы оптимизации РСУБД (раздел 19.4.3).
- **Представления.** Определение представлений позволяет консолидировать объекты, разделенные на таблицы суперклассов и подклассов. Представления удобны для чтения данных, но не для записи, которая поддерживается РСУБД лишь частично (раздел 19.4.4).
- **Взаимодействие с языком программирования.** При реализации взаимодействия программы с базой данных следует рассмотреть все имеющиеся возможности (раздел 19.6.1).
- **Преобразование данных.** Полезно бывает загрузить данные из файлов в вспомогательные таблицы. После этого обработка данных может осуществляться мощными средствами SQL. Код на этом языке писать быстрее и проще, чем на других языках программирования (раздел 19.6.2).
- **Инкапсуляция и оптимизация запросов.** Помните о противоречии между двумя этими принципами. Принимайте отдельное решение по каждому вопросу (раздел 19.6.3).
- **Объектно-ориентированные базы данных.** О них следует вспоминать только тогда, когда их возможности действительно необходимы приложению (раздел 19.7).

19.9. Резюме

Система управления базой данных — это программное обеспечение, предоставляющее универсальную функциональность для хранения и чтения данных, а также для управления доступом к ним. СУБД защищает данные от случайных потерь и делает их доступными для совместного использования. Существует несколько парадигм СУБД, но для разработки новых приложений чаще всего используются реляционные СУБД. Существуют и объектно-ориентированные СУБД, но их использование ограничено по прагматическим соображениям.

Объектно-ориентированные модели очень удобны для реализации в РСУБД. Разработчик может мыслить абстрактно, откладывая детали реализации на потом. При правильной реализации непротиворечивые объектно-ориентированные модели превращаются в расширяемые, эффективные и легкие для понимания

базы данных. В табл. 19.1 и 19.2 приведены общие правила реализации структуры РСУБД.

При реализации программной логики следует стремиться к максимально широкому использованию языка SQL.

Таблица 19.3. Ключевые понятия главы

потенциальный ключ	реализация обобщений
связь языка программирования с базой данных	индекс
преобразование данных	нормальная форма
база данных	нуль
система управления базой данных (СУБД)	объектно-ориентированная СУБД
внешний ключ	первичный ключ
индивидуальность	реляционная СУБД
реализация ассоциаций	SQL
реализация классов	представление

Библиографические замечания

Принципы СУБД и РСУБД описаны во множестве хороших книг. В качестве отличного учебника, объясняющего концепции баз данных, можно порекомендовать [Elmasri-00]. Примеры приложений, использующих объектно-ориентированные базы данных, можно найти в [Chaudhri-98].

Материал этой главы изложен более подробно в [Blaha-98]. Там можно найти сведения о файлах, РСУБД и ООСУБД (в частности, ObjectStore). Наш подход к базам данных согласуется с подходами других авторов, в частности с [Muller-99].

В работе [Chang-03] описаны средства сопряжения баз данных с графическим интерфейсом пользователя посредством промежуточных текстовых файлов.

Литература

[Blaha-98] Michael Blaha and William Premerlani. Object-Oriented Modeling and Design for Database Applications. Upper Saddle River, NJ: Prentice Hall, 1998.

[Chang-03] Peter H. Chang. A platform independent middleware architecture for accessing databases on a database server on the Web. IEEE Conference on Electro/Information Technology. Indianapolis, 2003.

[Chaudhri-98] Akmal B. Chaudhri and Mary Loomis. Object Databases in Practice. Upper Saddle River, NJ: Prentice Hall PTR, 1998.

[Elmasri-00] Ramez Elmasri and Shamkant B. Navathe. Fundamentals of Database Systems, Third Edition. Redwood City, CA: Benjamin/Cummings, 2000.

[Leavitt-00] Neal Leavitt. Whatever happened to object-oriented databases? IEEE Computer, August 2000, 16–19.

[Melton-93] Jim Melton and Alan R. Simon. Understanding the New SQL: A Complete Guide. San Francisco: Morgan Kaufmann, 1993.

[Muller-99] Robert J. Muller. Database Design for Smarties: Using UML for Data Modeling. San Francisco: Morgan Kaufmann, 1999.

Упражнения

19.1. (8) На рис. У19.1 показаны четыре модели классов для ориентированных графов. Ориентированный граф состоит из множества ребер и множества вершин. Каждое ребро соединяет две вершины и характеризуется стрелкой, указывающей направление движения. К одной вершине может подходить произвольное количество ребер. Пара вершин может быть соединена несколькими ребрами. Ребро может соединять вершину саму с собой. На рис. У19.1, а граф представляет собой ассоциацию типа «многие-ко-многим» между вершинами. Ориентированность указывается полюсами *from* и *to*. На рис. У19.1, б граф представляет собой ассоциацию типа «многие-ко-многим» между ребрами. Квалификаторы *end1* и *end2* представляют собой перечислимые типы с возможными значениями *to* и *from*, которые указывают, какие концы ребер соединяются между собой. На рис. У19.1, в и вершины, и ребра рассматриваются как объекты. Две ассоциации *to* и *from* хранят соединения между ними (по одной ассоциации на каждое ребро). На рис. 19.1, г каждое соединение представлено как квалифицированная ассоциация. Каждый конец ребра соединяется ровно с одной вершиной. Квалификатор *end* относится к перечислимому типу. Какая диаграмма лучше всего моделирует граф? Объясните преимущества и недостатки каждого варианта. Что произойдет, если пара вершин будет соединена несколькими ребрами? Может ли ребро соединять вершину саму с собой? Что произойдет, если к какой-то вершине присоединено только одно ребро?

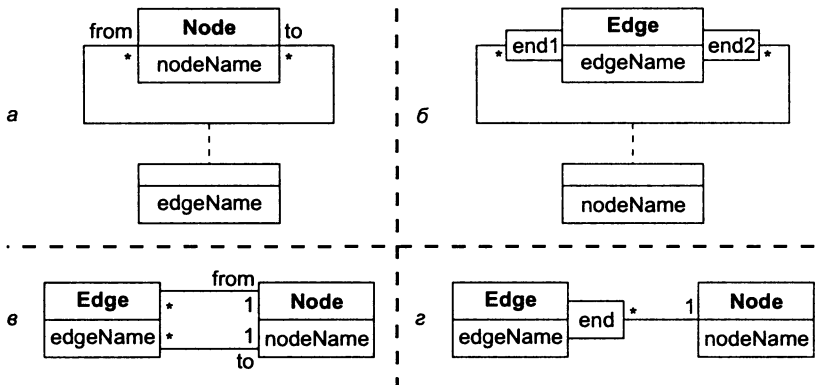


Рис. У19.1. Альтернативные модели классов ориентированного графа

19.2. (6) Нарисуйте таблицы для каждой модели из предыдущего примера. Укажите основной, возможный и внешний ключи. Используйте индивидуальность объектов.

- 19.3. (4) Напишите код на SQL для создания пустой базы данных для таблиц, соответствующих рис. У19.1, в и У19.1, г. Недостающую информацию дополняйте по своему усмотрению.
- 19.4. (3) Заполните таблицы базы данных, созданные командами SQL из предыдущего упражнения, в соответствии со структурой ориентированного графа на рис. У19.2.
- 19.5. Подготовьте запросы SQL для решения приведенных ниже задач. Вы должны работать с моделью классов на рис. У19.1, г. Четвертое задание потребует от вас дополнить SQL псевдокодом.
- 1) (3) По имени ребра определить две соединяемых им вершины.
 - 2) (3) По имени вершины определить все входящие и исходящие ребра.
 - 3) (5) По паре вершин определить соединяющие их непосредственно ребра (в любом направлении).
 - 4) (8) По одной вершине определить ее транзитивное замыкание (вершины, до которых можно добраться из данной вершины). Каждое ребро нужно проследивать в направлении от конца *from* к концу *to*.

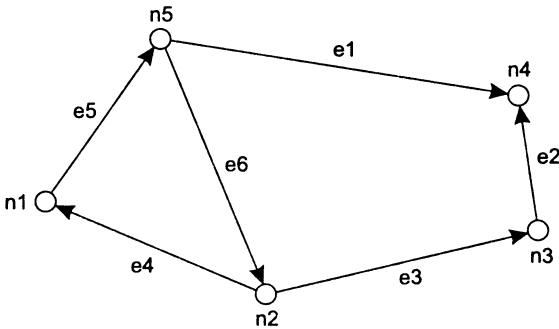


Рис. У19.2. Пример ориентированного графа

- 19.6. (6) Нарисуйте таблицы для модели на рис. У19.3. Выражение — это бинарное дерево членов, образуемое из констант, переменных и арифметических операторов. Использование унарного минуса не разрешено.
- 19.7. (4) Напишите код на SQL для создания пустой базы данных с таблицами из предыдущего примера.

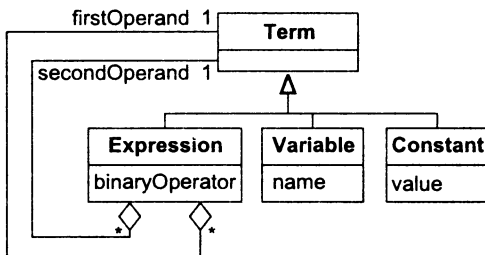


Рис. У19.3. Модель классов для выражений

- 19.8. (5) Заполните таблицы из предыдущего примера для выражения $(X + Y/2)/(X/3 - Y)$. Порядок операторов должен учитывать наличие скобок, но сами скобки в базу данных не попадают.
- 19.9. (7) Нарисуйте таблицы для рис. У19.4. Документ состоит из пронумерованных страниц. Каждая страница содержит множество рисунков: эллипсов, прямоугольников, ломаных, текстовых строк и групп объектов. Эллипсы и прямоугольники находятся внутри своих прямоугольников. Ломаная — это последовательность прямолинейных сегментов, определяемая соединяемыми этими сегментами точками. Текстовая строка характеризуется начальной точкой и шрифтом. Все ассоциации и агрегации считайте неупорядоченными. В этом упражнении вы можете не учитывать упорядоченность точек ломаной.
- 19.10. (7) Переделайте таблицы из предыдущего примера с учетом упорядоченности ассоциации *Polyline_Point*. База данных должна возвращать точки ломаной в правильной последовательности. (Замечание для преподавателя: можно предоставить студентам наш ответ к предыдущему упражнению.)
- 19.11. (6) Переделайте таблицы из упражнения 19.9 так, чтобы они отражали измененную модель классов на рис. У19.5. Обсудите преимущества и недостатки новой модели. Не учитывайте упорядоченность ассоциации *Points*.

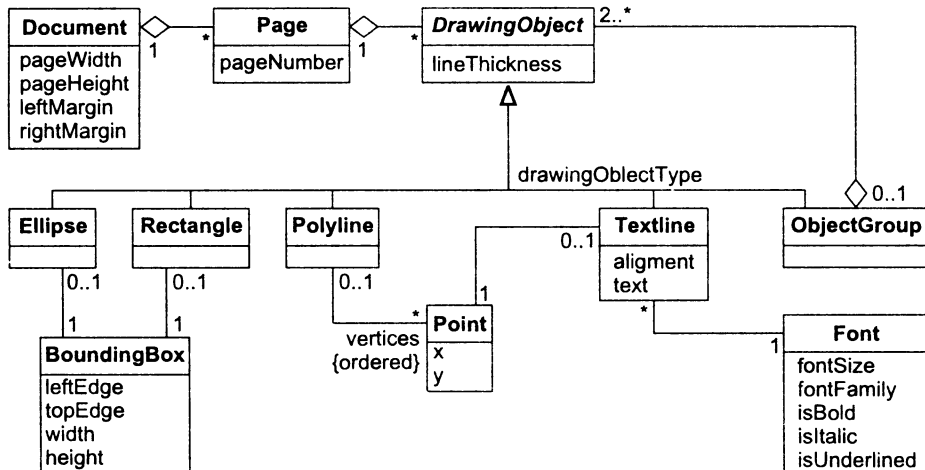


Рис. У19.4. Модель классов настольной издательской системы

- 19.12. (5) Напишите код SQL для создания пустой базы данных с таблицами из упражнения 19.9.
- 19.13. (5) Преобразуйте команды SQL из листинга 19.14 в модель классов. В таблице хранятся расстояния между парами городов.
- 19.14. (4) Для таблиц из упражнения 19.13 напишите запрос SQL, который будет определять расстояние между двумя городами по их названиям.

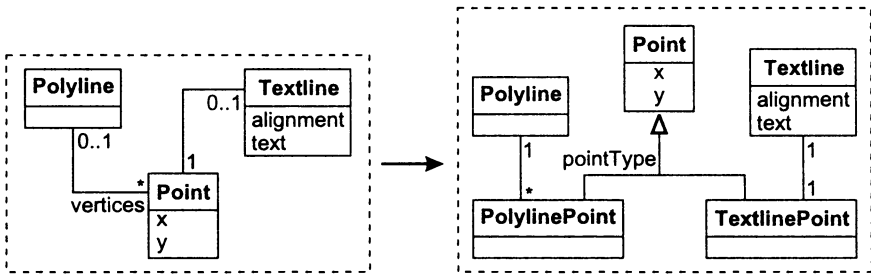


Рис. У19.5. Обобщение точки для предыдущей модели

Листинг 19.14. Создание таблиц для хранения расстояний между городами

```
CREATE TABLE City
( city_ID          NUMBER(30)      CONSTRAINT nn_city1 NOT NULL,
  city_name        VARCHAR2(255)   CONSTRAINT nn_city2 NOT NULL,
  CONSTRAINT pk_city PRIMARY KEY (city_ID),
  CONSTRAINT uq_city UNIQUE (city_name));

CREATE SEQUENCE seq_city;

CREATE TABLE Route
( route_ID         NUMBER(30)      CONSTRAINT nn_route1 NOT NULL,
  distance         NUMBER(20,100)  CONSTRAINT nn_route2 NOT NULL,
  CONSTRAINT pk_route PRIMARY KEY (route_ID));

CREATE SEQUENCE seq_route;

CREATE TABLE City_Distance
( city_ID          NUMBER(30)      CONSTRAINT nn_dist1 NOT NULL,
  route_ID         NUMBER(30)      CONSTRAINT nn_dist2 NOT NULL,
  CONSTRAINT pk_dist PRIMARY KEY (city_ID, route_ID));

CREATE INDEX index_dist1 ON City_Distance (route_ID);

ALTER TABLE City_Distance ADD CONSTRAINT fk_dist1
  FOREIGN KEY city_ID
  REFERENCES City;

ALTER TABLE City_Distance ADD CONSTRAINT fk_dist2
  FOREIGN KEY route_ID
  REFERENCES Route;
```

Стиль программирования

20

Любой шахматист, повар или лыжник может подтвердить, что знать в теории и уметь делать на практике — очень разные вещи. Это относится и к написанию программ. Недостаточно знать базовые конструкции языка и уметь оперировать ими. Опытный программист создает понятные программы, ориентируясь не только на сиюминутные потребности. При этом он руководствуется идиомами программирования, упрощенными правилами, предостерегающими советами и профессиональными секретами. Хороший стиль особенно важен в объектно-ориентированном программировании, потому что главное преимущество этого подхода заключается в возможности создания понятных расширяемых программ, допускающих повторное использование.

20.1. Объектно-ориентированный стиль

Хорошие программы — это нечто большее, нежели просто реализованные функциональные требования. Программы, разработанные в соответствии с нормативами, получаются более правильными, расширяемыми, удобными в отладке и допускающими повторное использование. Большая часть правил, применимых к обычным программам, действуют и для объектно-ориентированных программ. Чисто объектно-ориентированные конструкции, такие как наследование, требуют применения новых правил. Мы разбили все правила хорошего стиля программирования на несколько категорий. Некоторые правила относятся сразу к нескольким категориям:

- повторное использование (раздел 20.2);
- расширяемость (раздел 20.3);
- устойчивость (раздел 20.4);
- программирование «в большом» (раздел 20.5).

20.2. Повторное использование

Программное обеспечение, допускающее возможность повторного использования, позволяет сократить расходы на проектирование, кодирование и тестирование благодаря амортизации затрат на нескольких приложениях. Сокращение объема кода облегчает понимание программы, что, в свою очередь, снижает количество ошибок. Повторное использование возможно и в обычных языках программирования, но объектно-ориентированные языки значительно повышают его вероятность.

20.2.1. Виды повторного использования

Повторное использование возможно в нескольких ситуациях. Во-первых, один и тот же код может использоваться несколько раз внутри одного разрабатываемого приложения. Во-вторых, код предыдущих приложений может использоваться в новых приложениях. В обоих случаях действуют похожие правила. Для реализации многократного использования внутри одного приложения нужно искать избыточные последовательности кода и использовать средства языка программирования для консолидации этих последовательностей (рефакторинг). В результате применения описанного правила программы получаются меньше по объему, а их отладка значительно ускоряется.

Планирование повторного использования в будущем схоже с инвестированием и требует предвидения. Маловероятно, что отдельный класс будет использоваться в нескольких приложениях. Более вероятно использование тщательно продуманных подсистем, таких как абстрактные типы данных, графические пакеты и библиотеки численного анализа. В этом отношении могут оказаться полезными образцы и каркасы (см. главу 14).

20.2.2. Правила хорошего стиля для повторного использования

Собранные в этом разделе правила помогут вам повысить вероятность повторного использования в рамках одного приложения и между разными приложениями.

- **Методы должны быть цельными.** Метод считается цельным, если он выполняет одно действие или группу тесно связанных действий. Если он выполняет два несвязанных действия, разбейте его на два метода.
- **Методы должны быть небольшими.** Если метод получается слишком большим, разбейте его на более простые. Большим можно считать метод, размер которого превышает одну-две страницы кода. Разбивая метод на части, вы можете добиться их повторного использования даже в том случае, если весь метод в целом вам больше никогда не пригодится.
- **Методы должны быть согласованными.** Схожие методы должны иметь схожие названия, порядок аргументов, типы данных, возвращаемые значения и ошибочные ситуации. Везде, где это возможно, стремитесь обеспечивать параллельность. Методы одной операции должны иметь согласованную сигнатуру и семантику.

В операционной системе UNIX имеется множество примеров несогласованности функций. Например, в библиотеке C есть две несогласованные функции для вывода строк: *puts* и *fputs*. Первая записывает строку в стандартный поток вывода и заканчивает ее символом перевода строки. Вторая записывает строку в указанный файл, но не завершает ее символом перевода строки. Старайтесь избегать подобных несоответствий.

- **Отделяйте политику от реализации.** «Политические» методы принимают решения, обрабатывают аргументы и воплощают общий контекст. Эти методы переключают управление между реализующими методами. Политические методы должны осуществлять проверку состояния и контроль ошибок, но они не должны непосредственно выполнять вычисления или реализовывать сложные алгоритмы. Политические методы часто зависят от приложения, но их легко писать и читать.

Реализующие методы содержат конкретную детализированную логику, но не принимают решений о необходимости выполнения каких-либо действий. Если в таком методе возникает ошибочная ситуация, он должен только возвратить сообщение об ошибке, но не обрабатывать ее. Реализующие методы выполняют конкретные расчеты и часто содержат сложные алгоритмы. Им не нужен доступ к глобальному контексту, они не должны принимать решения, содержать значения по умолчанию или переключать поток управления. Поскольку суть реализующих методов сводится к автономным алгоритмам, они часто оказываются полезными в различных контекстах.

Не объединяйте политику и реализацию в одном методе. Изолируйте ядро алгоритма в отдельный реализующий метод. Для этого требуется абстрагирование параметров из политического метода и передача их в виде аргументов реализующему методу.

Например, метод масштабирования окна в два раза — это политический метод. Он должен установить коэффициент масштабирования окна и вызвать реализующий метод, который масштабирует окно на произвольный коэффициент. Если впоследствии вы решите изменить масштабный коэффициент по умолчанию на какой-либо другой, например на 1.5, достаточно будет всего лишь изменить параметр в политическом методе, а реализующий метод, который осуществляет собственно масштабирование, изменять не потребуется.

- **Учитывайте все возможности.** Напишите методы для всех возможных ситуаций, а не только для тех, которые требуются в данный момент. Например, если вы пишете метод для получения последнего элемента списка, напишите заодно метод для первого элемента. Это не только повысит вероятность повторного использования, но и рационализирует область применения родственных методов.
- **Стремитесь к максимальному расширению применимости метода.** Старайтесь обобщать типы аргументов, предусловия и постусловия, предположения о действии метода и контекст, в котором он работает. Закодируйте осмысленные действия для пустых аргументов, граничных значений и недопустимых значений. Часто небольшое увеличение объема кода метода значительно расширяет область его применения.

- **Не используйте глобальную информацию.** Сокращайте внешние ссылки. Ссылка на глобальный объект предполагает некоторый контекст использования метода. Часто всю необходимую информацию можно передать в виде аргументов. Можно хранить глобальную информацию в составе целевого объекта, чтобы другие методы тоже могли обращаться к ней.
- **Не используйте методов с состояниями.** Если поведение метода существенно зависит от предшествующей истории, вероятность его повторного использования значительно снижается. Старайтесь заменять такие методы на методы, не использующие информацию о состояниях. Например, приложение для обработки текста требует реализации операций вставки и замены. Один из возможных подходов состоит в установке флага *insert* (вставка) или *replace* (замена) и вызове операции *write*, которая вставляет или заменяет текст в зависимости от состояния флага. В другом подходе используются две операции *insert* и *replace*, которые осуществляют соответствующие действия без всякой информации о состоянии системы. Недостаток методов с информацией о состоянии заключается в том, что состояние объекта в одной из частей приложения может повлиять на метод, вызванный в другой части приложения.

20.2.3. Использование наследования

Приведенные выше рекомендации повышают вероятность повторного использования кода. В некоторых случаях методы различных классов кажутся похожими, но не достаточно похожи для того, чтобы заменить их одним унаследованным методом. Существует несколько методик разбиения методов на части для упрощения наследования кода.

- **Выделение общих частей.** Самый простой подход заключается в вынесении общего кода в новый метод, который будет вызываться каждым из двух похожих методов. Общий метод может быть перенесен в класс-предок. Фактически, это сводится к вызову подпрограммы (рис. 20.1).



Рис. 20.1. Повторное использование с выделением общей части

- **Выделение различий.** В некоторых случаях удобнее вынести отличительные черты методов разных классов в отдельные методы, а оставшийся код оставить в общем методе. Этот подход работает в тех случаях, когда общего у методов больше, чем различий. Как показано на рис. 20.2, новый метод вызывает операцию, которая реализуется по-разному методами разных классов. В некоторых случаях для общего метода приходится создавать абстрактный класс. Этот подход упрощает добавление подклассов, поскольку методы подклассов содержат только отличительные черты.

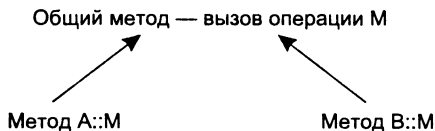


Рис. 20.2. Повторное использование с выделением отличий

Выделение общего и различий хорошо иллюстрирует пакет для построения графиков числовых данных. *DataGraph* — абстрактный класс, в котором содержатся общие для подклассов данные и операции. Один из методов *DataGraph* называется *draw*. Он состоит из следующих этапов: построение границы, масштабирование данных, построение осей, построение графика данных, построение заголовка и легенды.

Подклассы *DataGraph*, к которым относятся *LineGraph*, *BarGraph* и *ScatterGraph*, строят границы, заголовки и легенды одинаково, но по-разному реализуют масштабирование данных, построение осей и данных. Каждый подкласс наследует методы *drawBorder*, *drawTitle* и *drawLegend* от абстрактного класса *DataGraph*, но сам определяет собственные методы *scaleData*, *drawAxes* и *plotData*. Метод *draw* достаточно определить только один раз в классе *DataGraph*. Он наследуется всеми подклассами. Каждый раз при вызове метода *draw* вызываются *drawBorder*, *drawTitle* и *drawLegend* из суперкласса и *scaleData*, *drawAxis* и *plotData* из подкласса. Чтобы добавить новый подкласс, достаточно написать три его специализированных метода.

- **Делегируйте.** Иногда наследование может повысить объем повторного использования кода в программе, однако классы по смыслу не находятся в отношениях предок-потомок. Не поддавайтесь искушению воспользоваться наследованием только в целях облегчения реализации. Применяйте делегирование. Наследование нужно применять только тогда, когда оно семантически корректно. Наследование означает, что каждый экземпляр подкласса действительно является экземпляром суперкласса. Все операции и атрибуты суперкласса должны быть применимы и к подклассам. В результате неправильного применения наследования программы получают сложными в обслуживании и труднорасширяемыми. Объектно-ориентированные языки сами по себе не накладывают ограничений на семантику наследования. В такой ситуации правильнее применять механизм делегирования. Метод одного класса вызывает метод другого класса для фактического выполнения каких-либо действий. Поскольку передача управления осуществляется явным образом, вероятность возникновения каких-либо побочных эффектов существенно снижается. Названия методов в классах могут отличаться: в каждом классе они должны соответствовать истинному смыслу операций.
- **Инкапсулируйте внешний код.** Часто возникает желание использовать код, который был разработан для другого приложения в рамках иных соглашений об интерфейсах. Вместо того чтобы вставлять в свою программу непосредственный вызов внешнего кода, лучше инкапсулировать его внутри метода или класса. Это позволит изменить или заменить внешний пакет без крупномасштабной модификации вашего кода.

Например, если вы занимаетесь разработкой приложения для численных расчетов и хотите использовать существующий надежный код для вычисления обратной матрицы, можно написать класс Матрица, который будет инкапсулировать функциональность, предоставляемую внешним пакетом. Этот класс может содержать метод вычисления обратной матрицы, принимающий в качестве аргумента допуск на сингулярность и возвращающий новую матрицу, обратную к данной.

20.3. Возможность расширения

Чаше всего программное обеспечение приходится расширять в совершенно непредвиденных направлениях. Рекомендации по улучшению возможностей повторного использования повышают и возможность расширения. Кроме них имеются дополнительные правила хорошего стиля, представленные в этом разделе.

- **Инкапсулируйте классы.** Класс считается инкапсулированным, если его внутренняя структура скрыта от других классов. К реализации класса должны обращаться только его собственные методы. Многие компиляторы самостоятельно оптимизируют методы внешних классов, обеспечивая прямой доступ к реализации данного класса, но программист не должен делать этого сам. Уважайте другие классы, никогда не лезьте за данными, которые находятся внутри них.
- **Указывайте видимость методов.** Открытые методы доступны извне класса. Их интерфейсы должны быть опубликованы. Интерфейс открытого метода, который используется другими классами, изменить очень сложно, поэтому нужно стремиться свести количество таких методов к минимуму и тщательно продумывать их в процессе разработки. Скрытые методы находятся внутри класса, их можно удалять или изменять, и эти действия никак не затронут другие классы. Защищенные методы обладают промежуточной степенью видимости (см. раздел 18.1).

Аккуратное использование видимости (раздел 4.1.4) облегчает понимание классов и делает код более эластичным. Закрытые и защищенные методы скрывают ненужные детали от клиентов класса. В отличие от открытого метода, закрытый метод не может быть применен вне своего контекста, поэтому он может учитывать предусловия или информацию о состоянии класса.

- **Скрывайте структуры данных.** Не экспортируйте структуры данных из метода. Внутренние структуры данных зависят от выбранного алгоритма. Если вы экспортируете их, вы не сможете впоследствии поменять алгоритм.
- **Избегайте прослеживания нескольких связей и каскадных вызовов методов.** Метод не должен иметь полного знания о модели классов. Он должен прослеживать связи со своими соседями и должен вызывать их методы, но ему не следует прослеживать связь между его соседом и еще каким-либо классом, потому что эта связь не должна быть непосредственно видимой данному классу. Вместо этого класс должен вызвать метод соседнего класса,

если ему нужно обратиться к данным других классов. Если сеть ассоциаций изменится, изменения в методах будут минимальными.

Не допускайте применения одного метода к результатам выполнения другого метода, если класс результата еще не доступен в виде атрибута, аргумента, соседнего класса или класса из библиотеки низкого уровня. Напишите новый метод в исходном целевом классе, чтобы он выполнял составной метод. Описанные принципы были предложены в работе [Lieberherr-89].

- **Не используйте множественное ветвление на основании типа объекта.** Применяйте полиморфизм. Множественное ветвление можно использовать для проверки внутренних атрибутов объекта, но не для выбора поведения в зависимости от типа объекта. Выбор операций в зависимости от типа объекта — это самая суть полиморфизма, о чем забывать не следует.
- **Проектируйте в расчете на переносимость.** Системно-зависимые операции должны быть сконцентрированы в небольшом числе методов. Это облегчит перенос программного обеспечения на другие платформы. Например, при использовании базы данных можно изолировать доступ к ней. Это облегчит переключение на базу данных другого производителя и даже смену парадигмы (например, переход с реляционной базы данных на объектно-ориентированную).

20.4. Устойчивость

Вы должны стремиться к эффективности методов, но не за счет устойчивости. Метод считается устойчивым, если он не вызывает аварийного завершения даже при получении некорректных параметров. Никогда не жертвуйте устойчивостью к ошибкам пользователя.

- **Предусматривайте защиту от ошибок.** Программа должна защищаться от некорректного ввода и не давать сбить себя с толку. Методы должны проверять входные данные, которые могут вызвать ошибку.

Ошибки бывают двух типов. Анализ позволяет обнаружить ошибки, которые существуют в предметной области. Для таких ошибок можно указать адекватную реакцию системы. Например, банкомат должен обрабатывать ошибки сканирования карт и ошибки сетевого взаимодействия. Существуют также низкоуровневые системные ошибки, связанные с программными аспектами. К ним относятся ошибки выделения памяти, ошибки ввода-вывода, сбои аппаратного обеспечения. Программа должна контролировать возникновение системных ошибок и быть способной, по крайней мере, корректно завершить работу, если никакие другие действия уже не возможны.

Старайтесь предотвращать ошибки программирования и включать в программу диагностическую информацию. В процессе разработки бывает полезно вставлять в программу утверждения, которые помогут отыскивать ошибки. Впоследствии проверочные операции можно будет удалить для повышения эффективности работы. Языки с жесткой типизацией предоставляют

более надежную защиту от ошибок, связанных с несоответствием типов, а контрольные утверждения можно использовать в любом языке. В частности, не забывайте о проверке границ массивов.

Оптимизируйте программу только после того, как она заработает. Не занимайтесь оптимизацией, пока программа не начнет работать. Часто программисты тратят слишком много времени на оптимизацию кода, который выполняется относительно редко. Измерьте производительность отдельных частей программы. Чаще всего большую часть времени программа тратит на выполнение небольшого набора операций. Изучите приложение чтобы понять, какие критерии производительности для вас важны, например производительность в худшем случае или частота вызова методов. Если метод может быть реализован несколькими способами, оцените альтернативы по затратам памяти и времени с учетом простоты реализации. Старайтесь избегать избыточной оптимизации, так как любая оптимизация снижает расширяемость, возможность повторного использования и понятность кода. Если методы инкапсулированы правильно, вы сможете заменить их оптимизированными версиями, не меняя других частей программы.

Создавайте объекты сразу в корректном состоянии [Vermeulen-00]. В противном случае кто-нибудь может вызвать конструктор, а после него не вызвать нужный инициализирующий метод. Код всегда должен быть логически безупречным, это тоже одно из правил хорошего стиля в программировании.

Проверяйте аргументы. Открытые методы, доступные клиентам класса, должны досконально проверять передаваемые аргументы, потому что клиенты могут нарушать подразумеваемые ограничения. Эффективность закрытых и защищенных методов можно повысить, если предположить, что их аргументы имеют корректные значения. При реализации закрытых методов можно положиться на то, что все необходимые проверки осуществляются в открытых методах.

Не пишите свои и не используйте чужие методы, аргументы которых невозможно проверить. Например, печально известная функция *scanf* в UNIX считывает строку во внутренний буфер, не проверяя размер этого буфера. Этот недостаток используется для написания вирусов, вызывающих переполнение буфера в программном обеспечении, которое не осуществляет самостоятельной проверки аргументов.

Исключайте предопределенные ограничения. Везде, где это возможно, используйте динамическое выделение памяти для создания структур данных без предопределенных ограничений. Предсказать максимальную емкость структур данных в приложении достаточно сложно, поэтому не устанавливайте вообще никаких пределов. Время жестких ограничений на количество записей в таблице символов, на имена пользователей и файлов должно давно закончиться. Большинство объектно-ориентированных языков предоставляют отличные средства динамического выделения памяти.

Снабжайте программу средствами для отладки и контроля производительности. Подобно тому, как проектировщики интегральных схем снабжают

их контрольными точками, вы должны встраивать в свой код средства для тестирования, отладки, сбора статистики и контроля производительности. Уровень средств зависит от используемой среды программирования. Вы можете вставить отладочные операторы в методы, запуск которых будет зависеть от установленного уровня отладки. Отладочные операторы могут выводить сообщения при запуске и завершении работы метода, а также печатать выборочные значения переменных.

Для облегчения понимания поведения классов в них можно добавить код, осуществляющий сбор статистических данных. Некоторые операционные системы (в частности, UNIX) предоставляют средства для профилирования выполнения приложения. Обычно эти средства возвращают информацию о количестве вызовов каждого метода и затратах времени на его выполнение. Если в вашей системе подобные средства отсутствуют, вы можете встроить их в код самостоятельно.

20.5. Программирование крупных систем

Программирование крупных систем — это создание крупных программных комплексов командами программистов. В таких проектах первостепенной задачей становится обеспечение взаимодействия сотрудников, для чего используются соответствующие методики конструирования программного обеспечения. Помимо них вы можете руководствоваться следующими общими правилами хорошего стиля.

- **Не начинайте программировать раньше времени.** Продумывайте систему как можно дольше и тщательнее и только потом воплощайте ее в код. В конце концов, код все равно придется написать, чтобы создать приложение. Но писать код тяжело, а вносить в него изменения еще сложнее. Модели гораздо более «податливы», поскольку они разрабатываются на высоком уровне и не содержат лишних деталей. Гораздо лучше проработать свои идеи в моделях, достичь полного понимания системы и только затем переходить к написанию кода.
- **Делайте методы удобочитаемыми.** Осмысленные имена переменных повышают удобочитаемость метода. Затраты на ввод нескольких лишних символов значительно меньше, чем на устранение последствий неправильного понимания вашего кода другим программистом. Не используйте непонятных сокращений. Заменяйте длинные выражения с большой глубиной вложения на временные переменные. Не используйте одну и ту же переменную для разных целей внутри одного метода, даже если области действия этой переменной не перекрываются. Большинство компиляторов в любом случае оптимизируют переменные. Незначительное повышение эффективности не окупит потери удобочитаемости.
- **Делайте методы понятными.** Метод можно считать понятным, если его код может понять кто-либо, кроме его создателя (а также сам создатель через некоторый промежуток времени после написания метода). Небольшие цельные методы в среднем получают более понятными.

- **Используйте те же названия, что и в модели классов.** Названия сущностей в программе должны соответствовать названиям из модели классов. В программе могут использоваться дополнительные сущности, служащие целям реализации модели, но сущности, переносимые из модели в программу, должны сохранять свои исходные имена. Следование этому правилу повышает возможности отслеживания изменений, документирования, а также улучшает понятность кода. Можно использовать дополнительные соглашения, помогающие избегать конфликтов имен. К таким соглашениям относятся префиксы имен.
- **Аккуратно выбирайте имена.** Имена должны точно описывать операции, классы и атрибуты, к которым они относятся. Выберите один стиль именования и следуйте ему всегда. Например, вы можете давать операциям имена вида *действиеОбъект* (например, *addElement* или *drawHighlight*). Создайте словарь часто используемых действий. К примеру, не стоит использовать и *new*, и *create* в одной программе, если только они не несут разный смысл. Во многих объектно-ориентированных языках названия методов формируются автоматически из названий классов и операций.

Не используйте одинаковые имена методов для семантически различающихся операций. Как показано на рис. 20.3, все методы с одинаковыми именами должны иметь одну и ту же сигнатуру (количество и тип аргументов) и одинаковый смысл.

Circle::area()	~~Matrix::invert()~~ — performs matrix inversion
Rectangle::area()	~~Figure::invert()~~ — turns figure upside down
Правильно	Старайтесь избегать

Рис. 20.3. Названия методов

- **Следуйте правилам программирования.** Команды, работающие над проектами, должны следовать правилам программирования, установленным в их организациях, или внешним стандартам, таким как [Vermeulen-00]. Правила программирования касаются таких вопросов, как формат имен переменных, выделение управляющих структур отступами, документирование методов в их заголовках, а также встроенное документирование.
- **Используйте пакеты.** Группируйте тесно связанные классы в пакеты (см. раздел 4.11).
- **Документируйте классы и методы.** Документация метода должна описывать его назначение, функции, контекст, входные и выходные данные, а также все предположения и предусловия, касающиеся состояния объекта. Вы должны описать не только детали алгоритма, но также причины его выбора. Основные части метода следует документировать комментариями внутри этого метода.
- **Избегайте дублирования кода.** В работе [Baker-95] рассматриваются два приложения с объемом кода более миллиона строк (одно из приложений — система X Window). Было обнаружено, что не менее 12 % кода было

получено копированием, которого можно было избежать. В результате копирования код разбухает, что затрудняет его поддержку и понимание.

- **Публикуйте спецификацию.** Спецификация — это контракт между создателем класса и его клиентами. После написания спецификации создатель класса не может нарушить ее условия, потому что в противном случае он причинит вред клиентам. Спецификация содержит объявления методов, но их должно быть достаточно, чтобы клиент мог использовать эти методы. Доступность в сети описания класса и его составляющих помогает обеспечить корректность использования этого класса. Ниже приведены примеры спецификаций.

Спецификация класса

Название класса: *Circle*

Версия: 1.0

Описание: Эллипс, большая и малая полуоси которого равны.

Суперклассы: *Ellipse*

Составляющие:

Закрытые атрибуты:

center: Point — координаты центра

radius: Real — радиус окружности

Открытые методы:

draw(Window) — построение окружности в окне

intersectLine(Line): Set of Points — поиск точек пересечения отрезка и окружности, возвращает множество, содержащее от 0 до 2 точек

area(): Real — вычисляет площадь круга

perimeter(): Real — вычисляет длину окружности

Закрытые методы: отсутствуют

Спецификация операции

Операция: *intersectLine(line: Line): Set of Points*

Исходный класс: *GeometricFigure*

Описание: Возвращает множество точек пересечения геометрической фигуры с отрезком. Множество может содержать произвольное неотрицательное количество точек. Точки касания присутствуют в множестве в единственном числе. Если отрезок совпадает с прямолинейным сегментом фигуры, в множество включаются только две конечные точки прямолинейного сегмента.

Статус: Абстрактная операция в исходном классе, должна быть перекрыта.

Входные данные:

self: GeometricFigure — геометрическая фигура, для которой осуществляется поиск

line: Line — отрезок, для которого ищутся точки пересечения

Возвращает:

Множество точек пересечения. Множество может содержать произвольное количество точек.

Побочные эффекты: Никаких.

Ошибочные ситуации:

Если фигуры не пересекаются, возвращает пустое множество.

Если отрезок совпадает с линейным сегментом фигуры, множество будет содержать только две конечные точки этого линейного сегмента.

Если фигура содержит ограничиваемую ей площадь, рассматривается только граница этой фигуры.

Спецификация метода

Метод: *Circle::intersectLine(line: Line) Set of Points*

Описание: Поиск точек пересечения (от 0 до 2) окружности и отрезка. Возвращает множество точек пересечения. Если отрезок является касательной к окружности, множество будет содержать одну точку.

Входные данные:

self:Circle — окружность

line:Line — отрезок

Возвращает:

Множество точек пересечения. Множество может содержать от 0 до 2 точек.

Побочные эффекты: Никаких.

Ошибочные ситуации:

Если фигуры не пересекаются, возвращает пустое множество.

Если отрезок является касательной к окружности, возвращает точку касания.

Если радиус окружности равен 0, возвращает единственную точку, при условии, что центр окружности лежит на отрезке.

20.6. Резюме

Хороший стиль программирования помогает максимально реализовать преимущества объектно-ориентированного программирования. Наибольший выигрыш достигается благодаря сокращению затрат на обслуживание и усовершенствования, а также благодаря возможности повторного использования кода в новых приложениях. Правила хорошего стиля объектно-ориентированного программирования добавляют к обычным правилам новые, которые распространяются на уникальные концепции, присущие только объектно-ориентированному подходу.

Одной из важных целей объектно-ориентированной разработки является повышение возможностей повторного использования классов и методов. Повторное использование внутри одного приложения подразумевает поиск и консолидацию схожих участков кода. Планирование повторного использования в будущих приложениях требует больших затрат. Вероятность использования повышается, если

методы делаются небольшими, цельными и локальными. Важно отделять политику от реализации. Чтобы использовать наследование, можно разбить общий метод на вспомогательные методы, часть из которых унаследовать от исходного класса, а другую часть определить в подклассах. При необходимости совместного использования методов между классами, не находящимися в родственных отношениях, следует применять делегирование.

Большинство программ когда-нибудь приходится расширять. Способность к расширению можно повысить благодаря инкапсуляции, сокращению зависимостей, использованию методов для обращения к атрибутам других классов и ограничения видимости методов.

Не жертвуйте устойчивостью ради эффективности. Поскольку объекты содержат ссылки на свои классы, они менее уязвимы к несоответствию типов, чем обычные программные переменные. Внутри метода можно осуществлять динамическую проверку соответствия объектов каким-либо предположениям. Программы должны быть защищены от ошибок пользователя и системы. Не следует забывать об утверждениях, которые помогают отлавливать ошибки программирования.

Написание больших программ командами программистов требует более высокой дисциплинированности, тщательного документирования и активного взаимодействия между участниками проекта. Жизненно важно, чтобы методы были удобочитаемыми и хорошо документированными.

Таблица 20.1. Ключевые слова главы

делегирование	оптимизация	устойчивость
документирование	программирование крупных систем	видимость
инкапсуляция	рефакторинг	
расширяемость	повторное использование	

Библиографические замечания

Объектно-ориентированное программирование требует корректного воплощения концепций приложения в языковые конструкции. Большинство принципов, применимых к обычному программированию, распространяются и на объектно-ориентированное программирование. Хорошим руководством по стилю программирования является книга [Kernighan-99].

Программирование крупных систем хорошо описано в известной книге [Brooks-95]. Детальный список правил по программированию на Java приводится в [Vermeulen-00], и большая часть этих правил применима и к другим языкам.

Литература

[Baker-95] Brenda S. Baker. On finding duplication and near-duplication in large software systems. Second IEEE Working Conference on Reverse Engineering, July 1995, Toronto, Ontario, 86—95.

[Brooks-95] Frederick P. Brooks, Jr. The Mythical Man-Month, Anniversary Edition. Boston: Addison-Wesley, 1995.

[Kernighan-99] Brian W. Kernighan, Rob Pike. The Practice of Programming. Boston: Addison-Wesley, 1999.

[Lieberherr-89] Karl J. Lieberherr, Arthur J. Riel. Contributions to teaching object-oriented design and programming. OOPSLA'89 as ACM SIGPLAN 24, 11 (November 1989) 11–22.

[Vermeulen-00] Allan Vermeulen, Scott W. Ambler, Greg Bumgardner, Eldon Metz, Trevor Misfeldt, Jim Shur, Patrick Thompson. The Elements of Java Style. Cambridge, UK: Cambridge University Press, 2000.

Упражнения

- 20.1. (4) Одна из методик повторного использования кода сводится к помещению метода в список аргументов другого метода. Например, одна из операций, определенных на бинарном дереве, — упорядоченная печать. Подпрограмма *print(node)* распечатывает значения дерева, корнем которого является узел *node*. Она включает рекурсивный вызов *Print(node.leftSubtree)*, который осуществляется при наличии левого поддерева, затем выполняется печать значения корневого узла, а после этого выполняется рекурсивный вызов для правого поддерева. Этот подход может быть обобщен на другие операции. Приведите не менее трех примеров операций, определенных на узлах бинарного дерева. Напишите псевдокод операции *orderedVisit(node, method)*, которая будет последовательно применять метод *method* к узлам дерева, корнем которого является узел *node*.
- 20.2. (3) Объединение схожих методов в одну операцию повышает вероятность повторного использования кода. Объедините приведенные в примерах методы в одну операцию при помощи переопределения, расширения или обобщения. Укажите атрибуты, необходимые для отслеживания обоих типов счетов.
 - 1) *cashCheck(normalAccount, check)* — если значение *check* меньше баланса на счете *normalAccount* (обычныйСчет), оплатить чек и вычесть деньги со счета. В противном случае возратить чек.
 - 2) *cashCheck(reserveAccount, check)* — если значение *check* меньше баланса на счете *reserveAccount* (резервныйСчет), оплатить чек и вычесть деньги со счета. В противном случае проверить баланс резерва. Если перевод резервных средств позволяет оплатить счет без превышения лимита, оплатить чек и обновить значения балансов. В противном случае возратить чек.
- 20.3. (4) В листинге У20.1 приведена функция на языке С, создающая новую страницу в системе автоматизированного проектирования. Страница представляет собой именованную двухмерную область, которая может содержать текст и графику и может быть отображена на экране. Для полного описания системы может потребоваться несколько страниц. Функция в листинге

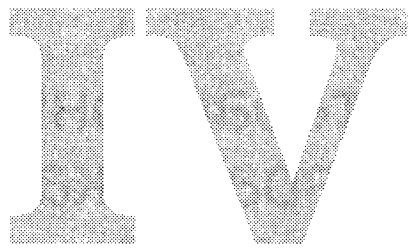
создает новую вертикальную или горизонтальную страницу, после чего образует ее имя из заданного корня и суффикса. Она вызывает следующие функции языка C: *strlen* (длина строки), *strcpy* (копирование строки), *strcat* (конкатенация строк), *malloc* (выделение памяти). Типы данных *SheetType* и *Sheet* определены вне функции в том же модуле. Функции *strlen*, *strcpy*, *strcat* вызовут сбой системы, если в качестве любого аргумента им передать 0. Таким образом, предлагаемый вариант функции может аварийно завершить работу при возникновении ошибок нескольких типов. К ошибке приведет нулевое значение аргументов *rootName* и *suffix*, а также недопустимое значение *sheetType*. Возможен отказ выделения памяти при вызове *malloc*.

- 1) Перечислите все возможные варианты аварийного завершения функции. В каждом случае укажите последствия.
- 2) Перепишите функцию таким образом, чтобы она не могла аварийно завершиться из-за перечисленных вами ошибочных ситуаций. Вместо этого она должна печатать сообщение об ошибке, что было бы полезно при отладке содержащей ее программы.

Листинг У20.1. Создание страницы

```
Sheet createSheet (sheetType, rootName, suffix)
SheetType sheetType;
char *rootName, *suffix;
{   char *malloc(), *strcpy(), *strcat(), *sheetName;
    int strlen(), rootLength, suffixLength;
    Sheet sheet, vertSheetNew(), horizSheetNew();
    rootLength = strlen(rootName);
    suffixLength = strlen(suffix);
    sheetName = malloc(rootLength + suffixLength + 1);
    sheetName = strcpy(sheetName, rootName);
    sheetName = strcat(sheetName, suffix);
    switch(sheetType)
    {   case VERTICAL:
        sheet = vertSheetNew();
        break;
        case HORIZONTAL:
        sheet = horizSheetNew();
        break;
    }
    sheet->name = sheetName;
    return sheet;
```

Разработка программного обеспечения



К этому моменту вы уже научились довольно многому, прочитав первые три части нашей книги. Вы познакомились с объектно-ориентированными концепциями и их обозначениями на языке UML. Вы изучили процесс применения этих концепций и получили представление о тонкостях реализации моделей на языках C++ и Java, а также в базах данных. Последняя часть книги рассматривает некоторые вопросы разработки программного обеспечения более подробно.

Мы все время подчеркивали, что разработка программ должна осуществляться итерационно. В книге изложение процесса неизбежно должно было получиться линейным, но нам не хотелось бы, чтобы у читателей создалось ложное впечатление. Разработка программ редко оказывается прямолинейной. В главе 21 мы исследуем ее итерационную природу.

По нашему опыту, большинство организаций заинтересованы в применении объектно-ориентированной технологии и особенно в объектно-ориентированном моделировании. Однако многим оказывается сложно ввести эту технологию на практике, потому что они не имеют представления о том, как это следует делать. В главе 22 рассказывается о том, каким образом можно получить отдачу от объектно-ориентированной технологии и внедрить ее в вашей организации.

Приложения редко создаются с нуля. Обычно разработка основывается на опыте, полученном от предшествующих приложений. Устаревшие системы часто задают множество требований к их последователям. Из-за них возникают дополнительные сложности, связанные с преобразованием данных и интеграцией связанных систем. В главе 23 мы поговорим об этих вопросах.

Четвертая часть завершает нашу книгу. Изучив ее, вы будете готовы перейти к самостоятельному объектно-ориентированному моделированию и его практическому применению. Мы будем рады вашим комментариям и рассказам, которые помогут нам углубить собственное понимание предмета. Пишите нам по адресу blaha@computer.org.

Итерационная разработка

В письменном изложении процесс разработки может показаться линейным, но на самом деле это нежелательный артефакт носителя информации. Истинная природа процесса разработки программного обеспечения — итерационная. На ранних этапах невозможно точно предвидеть устройство системы, из-за чего впоследствии приходится возвращаться к пройденным этапам, исправлять допущенные ошибки и вносить усовершенствования. *Итерационная разработка* (iterative development) — это процесс, который делится на последовательность этапов или итераций, на каждом из которых достигается более точное приближение к требуемой системе, чем на предыдущих этапах [Rumbaugh-05].

21.1. Обзор итерационной разработки

Мы предпочитаем итерационный подход к разработке всем прочим подходам. Эту главу мы начнем со сравнения итерационной разработки с двумя другими распространенными подходами: водопадной разработкой (раздел 21.2) и быстрым прототипированием (раздел 21.3). Затем мы рассмотрим следующие вопросы, связанные с итерационной разработкой:

- масштаб итерации (раздел 21.4);
- выполнение итерации (раздел 21.5);
- планирование следующей итерации (раздел 21.6);
- моделирование и итерационная разработка (раздел 21.7);
- определение рисков (раздел 21.8).

21.2. Итерационная и водопадная модели

В 80-х и 90-х годах прошлого века водопадный подход был доминирующей парадигмой жизненного цикла программ [Larman-03]. Как показано на рис. 21.1, в рамках этого подхода разработчики переходят от этапа к этапу в жесткой ли-

нейной последовательности без всяких возвратов. Начало следующего этапа возможно только после завершения предыдущего.

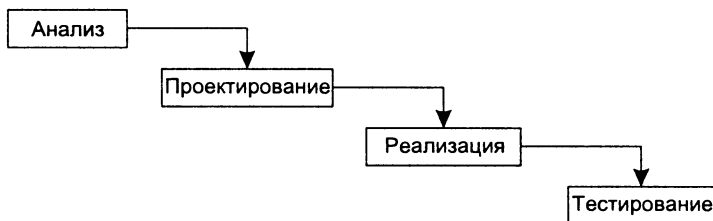


Рис. 21.1. Водопадный подход

На собственном опыте сообщество разработчиков убедилось в неэффективности этого подхода для большинства приложений [Sotirovski-01]. Водопадный подход пригоден для хорошо понятных приложений, где выход этапов анализа и проектирования вполне предсказуем, но такие приложения встречаются довольно редко. Для большинства приложений характерна неопределенность требований. Кроме того, в этом подходе работоспособная система получается только после завершения всех этапов. Это затрудняет оценку успешности выполнения проекта и корректировку отклонений.

Итерационная разработка, напротив, характеризуется наличием множества ключевых событий или вех (milestones) и позволяет обнаруживать просчеты на ранних стадиях, когда система еще гибкая и податливая к изменениям, которые обходятся дешевле. Очевидно, что итерационная разработка более совершенна.

21.3. Итерационная разработка и быстрое прототипирование

Быстрое прототипирование (rapid prototyping — рис. 21.2) состоит в быстрой разработке части программы и ее оценке в процессе практического использования. Полученный опыт учитывается при повторном проходе цикла разработки. В конечном итоге последний прототип поставляется заказчиком в виде готового приложения или же осуществляется переход к другому подходу. Быстрое прототипирование обладает схожими с итерационной разработкой преимуществами. Отличие состоит в том, что в первом подходе часто приходится отбрасывать имеющийся код, тогда как во втором подходе код аккумулируется.

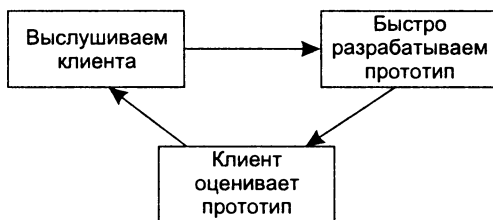


Рис. 21.2. Быстрое прототипирование

Прототипирование представляет собой проверку концепций, при которой часто приходится отбрасывать частично готовые варианты программы. Итерационная разработка тоже допускает исключение части кода при изменениях, но такое исключение не является намеренным.

Быстрое прототипирование ориентировано на взаимодействие с заказчиком и помогает быстро определять его требования. Оно может быть полезно для демонстрации технической возможности реализации в тех случаях, когда речь идет о технологии, которая может вызвать затруднения. Недостаток состоит в том, что отказаться от написанного кода может быть сложно. Заказчики часто путают успешный прототип с готовым продуктом и не понимают, что прототип предназначен только для демонстрации и может быть лишен устойчивой инфраструктуры. Некоторые клиенты рассматривают отбрасывание кода как выбрасывание денег на ветер. Им трудно понять, что истинная ценность прототипов состоит в том опыте, который приобретается с их помощью.

Итерационная разработка позволяет сохранить те же преимущества, при условии, что итерации делаются достаточно короткими, а промежуточные версии демонстрируются клиенту. Оба метода дают заказчику возможность регулярно контролировать успешность выполнения проекта. Они также помогают разработчикам устранять возможные проблемы на ранних стадиях.

21.4. Масштаб итераций

Итерационная разработка состоит из последовательности итераций. Количество итераций и их длительность зависят от масштабов проекта. В коротком проекте длительностью не более шести месяцев длина итерации может составлять две—четыре недели. В большом проекте длительностью в несколько лет итерации могут продолжаться три—четыре месяца. Слишком короткие итерации создают избыточные накладные расходы. Слишком длинные итерации не позволяют получить достаточного количества контрольных точек для оценки выполнения проекта и выполнения промежуточных корректировок. Вы должны стремиться к тому, чтобы все итерации были одинаковыми, но в некоторых случаях может возникнуть потребность удлинить итерацию, например, при проработке глубокой инфраструктуры или сложных составляющих.

Определите масштаб итераций. В качестве цели удобно выбрать минимальный объем работы, который дает на выходе ощутимый материальный результат. Важные для приложения составляющие нужно создавать как можно раньше. Это относится и к ключевым участкам кода, которые часто выполняются приложением. Следите за тем, чтобы функциональность внутри системы была сбалансирована. Разработчики могут иметь любимые методики и предпочитать работать с разными аспектами, но план в целом должен быть сбалансирован и нацелен на максимально быструю реализацию законченных самостоятельных частей приложения. Каждая итерация должна обеспечивать как минимум что-то одно из нижеследующего: экономическую отдачу, расширение функциональности, улучшение взаимо-

действия с пользователем, повышение эффективности, повышение надежности или укрепление инфраструктуры (последнее необходимо для обслуживания и для последующих итераций).

Хороший базис для оценки задают варианты использования. В рамках одной итерации можно сосредоточиться на нескольких вариантах использования. Конечно, не обязательно, чтобы итерация реализовывала вариант использования целиком. Вы можете реализовать основную функциональность на первой итерации, расширенную функциональность на следующей, а обработку ошибок добавить на третьей итерации. Однако не следует разбивать вариант использования по слишком большому количеству итераций. Помимо вариантов использования, вы должны разделить по итерациям внутренние сервисы — механизмы и службы, предоставляющие инфраструктуру или обеспечивающие поддержку для реализации более высокоуровневых операций. Эти сервисы идентифицируются на этапах планирования архитектуры и проектирования классов.

Если приоритет какой-либо из составляющих необходимо повысить, приоритет другой составляющей должен быть понижен. Это позволит избежать синдрома одинаковой важности всех частей. Никогда не бывает так, чтобы все было одинаково важно, просто менеджеры и разработчики боятся делать сложный выбор или признавать, что времени на все не хватает. Отслеживая длительность итераций и корректируя их содержимое, вы вынуждены смотреть на вещи трезво и представлять ход выполнения плана.

Не обязательно предоставлять клиенту результаты каждой итерации. С точки зрения разработчика, самое главное — не останавливаться, оставаться в рамках графика и следить за тем, чтобы части приложения подходили друг к другу. С точки зрения клиента установка промежуточных продуктов после каждой итерации может потребовать слишком больших расходов. Для упрощения логистики можно объединить несколько промежуточных версий перед передачей их заказчику.

21.5. Выполнение итерации

Каждая итерация должна начинаться с единой для всех разработчиков базовой версии и заканчиваться новой базовой версией. В конце итерации разработчики должны интегрировать все версии артефактов системы и проверить их. Это позволит всем работать с одинаковым набором предположений и не отставать от изменений системы. Выполнение этого правила — обязательное условие успеха проекта.

Некоторым разработчикам такое правило может показаться неудобным. Кажется, что эффективнее продолжать разрабатывать подсистему, не тратя время на ее интеграцию с остальными частями системы. Действительно, если команда делится на мелкие группы, которым приходится часто взаимодействовать друг с другом, это не так удобно. Но это жизненно важно для успеха проекта в целом. Если группы слишком долго работают сами по себе, они расходятся в своих предположениях, интерфейсах и по другим параметрам. Осуществление интеграции может оказаться сложным и дорогостоящим. Но часто оказывается, что предположения, заложенные в разные подсистемы, просто несовместимы друг с другом.

Тогда приходится отслеживать изменения или идти на программистские ухищрения, чтобы решить возникшие проблемы.

Чтобы закончить итерацию, сдать ее результаты, протестировать и интегрировать ее в систему, команда должна структурировать свою деятельность. Это требует затрат на планирование, которые, впрочем, окупаются в долгосрочной перспективе в масштабах всей системы.

Второе правило гласит, что на выходе каждой итерации должна получаться исполняемая версия программы. Недостаточно написать код, который нельзя выполнить. Чтобы протестировать код, его необходимо запустить. После этого можно проверить интеграцию подсистем, обнаружить и устранить несовместимости на ранних этапах. Более того, именно исполняемый код является лучшей мерой успешного выполнения проекта. Очень легко обмануть себя и других, если исполняемого кода нет. Так можно пропустить серьезные недочеты или недооценить сложность отладки и интеграции подсистем.

В каждой итерации должно быть отведено достаточно времени на все этапы разработки. Внутри итерации программа пишется по уменьшенной водопадной модели. Вы переходите от анализа к проектированию, реализации, тестированию и интеграции. Водопадный подход вполне эффективен в небольших масштабах, где он обеспечивает систематическую разработку функциональности. Проблемы возникают только в том случае, если принимаемые решения нельзя изменить на более поздних этапах. В итерационном подходе неправильные решения можно изменить на последующих итерациях, поэтому они не угрожают проекту в целом.

Отведите в своем плане достаточно времени для тестирования. Очень важно проводить тестирование по ходу работы и не откладывать его на поздние итерации. Вся суть итерационной разработки в том, чтобы строить надежные системы маленькими шагами.

21.6. Планирование следующей итерации

После каждой итерации вы должны оценить выполнение проекта и пересмотреть план на следующую итерацию. Как соотносится длительность прошедшей итерации с запланированной? Правильно ли вы распределили задачи между разработчиками? Доволен ли клиент выполнением работ? Возникли ли какие-нибудь очевидные проблемы или задачи для следующей итерации? Получилась ли программа стабильной, или вам придется отвести больше времени на реорганизацию на следующей итерации?

Конечно, если итерация оказалась успешной, выполнение плана можно продолжать. В противном случае не нужно бояться отбросить неправильные решения и внести необходимые коррективы. Приложение получится расширяемым, обслуживаемым и жизнеспособным, только если у него есть надежная основа. Вы должны регулярно получать отзывы пользователей. Они особенно важны на ранних этапах, потому что вам нужно, чтобы пользователи впитали особенности вашего приложения и подумали о его применении в повседневном бизнесе. Более того, пользователи могут помочь определить, когда работы отклонятся от заданного направления.

21.7. Моделирование и итерационная разработка

Моделирование естественным образом дополняет итерационную разработку. Одной из целей итерационной разработки является раннее обнаружение проблем в программном обеспечении. То же можно сказать и о моделировании. В работе [Sotirovski-01] автор выразительно охарактеризовал философию итерационной разработки словами «быстрый отказ». Проблемы неизбежны, но лучше их обнаружить на ранних этапах, чем на более поздних. Опытный разработчик может обнаружить некоторые проблемы еще в модели и тем самым сократить объем работ в одной итерации. В итоге разработка идет быстрее и проще. Итерационная разработка, разумеется, не оправдывает хакерских приемов и отказа от вдумчивого моделирования.

Как показано на рис. 21.3, начинать следует с аккуратного моделирования приложения, что позволит выявить требования, после чего следует построить модель для первой итерации. Затем можно снова вернуться к модели и провести следующую итерацию и так далее, чередуя моделирование с итерационной разработкой. Моделирование позволяет раньше обнаружить ошибки и дает ощущение направления и непрерывности последовательности итераций. Моделирование может и должно осуществляться быстро, чтобы не замедлять график проекта.

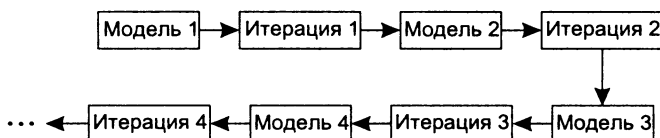


Рис. 21.3. Моделирование и итерационная разработка

В табл. 21.1 моделирование сравнивается с итерационной разработкой. Обе методики способствуют фиксации требований, но делают это по-разному. Моделирование помогает клиентам представить возможности программного обеспечения еще до его создания. Итерационная разработка помогает им увидеть программу в процессе ее развития, чтобы они могли выступить с комментариями или изменить направление приложения усилий разработчиков.

Ничто не может сравниться с моделированием по способности повысить качество приложения. В работе [Brooks-87] утверждается, что «концептуальная цельность — важнейший критерий при проектировании системы». Моделирование предназначено для облегчения понимания и усовершенствования сути приложения. Итерационная разработка подразумевает частое тестирование, а потому также способствует усовершенствованию, хотя и не столь эффективна сама по себе, как вместе с моделированием.

Моделирование повышает производительность благодаря быстрому выявлению недостатков при мысленных экспериментах, позволяющих обойтись без отброшенного кода. Итерационная разработка также дает значительный вклад в производительность, потому что требует интеграции на ранних стадиях, что исключает создание несовместимых компонентов или неудачных редакций.

По определению моделирование не участвует в отслеживании проекта. Моделирование относится к ранней стадии разработки программного обеспечения, поэтому не может служить критерием оценки хода проекта как целого. Итерационная разработка гораздо лучше подходит для отслеживания проекта, потому что частый выпуск рабочего кода не оставляет места для споров о том, что сделано, а что нет. Вследствие этого график проекта становится более предсказуемым.

21.8. Идентификация рисков

Самое важное в планировании итерации — минимизация рисков. Встречаться с опасностями нужно как можно раньше, а не откладывать их до конца проекта (что иногда случается). Разновидностей возможных рисков существует довольно много.

- **Технические риски.** Предлагаемое техническое решение может оказаться неправильным или неприемлемым. Если вы займетесь техническими вопросами на раннем этапе, возможно, вы успеете найти другое решение до того, как будет слишком поздно, и прежде, чем на плохом фундаменте будет построена вся система.
- **Технологические риски.** Технология, на которую вы рассчитываете, может оказаться недоступной или несоответствующей требованиям. Испытывайте технологии, задействованные в важнейших частях системы, как можно раньше.
- **Риски приемлемости для пользователя.** Пользователям может не понравиться интерфейс или функциональность системы. Итерационная разработка позволяет пользователям испытать часть системы в то время, когда стиль ее оформления еще может быть изменен достаточно легко.
- **Риски планирования.** Всегда существует шанс, что проект не завершится вовремя. Итерационная разработка снижает этот шанс, поскольку предоставляет точные критерии выполнения. Если наблюдается отставание от графика, вы можете пожертвовать частью функциональности. Даже если проект завершится позже назначенного срока, у вас все равно будет работающая система, которую вы сможете предъявить в этот срок. Система с 90 % функциональности гораздо лучше, чем реализованная на 90 %, но абсолютно неработоспособная система в водопадной модели.
- **Риски персонала.** Ключевые лица могут выйти из проекта в самый неудачный момент. Итерационная разработка характеризуется частыми контрольными точками со стабильными выпусками системы. Модели гарантируют, что итерации тщательно продумываются и документируются. Потеря ключевых лиц не может остаться незамеченной, но, по крайней мере, у вас будет шанс подобрать им замену.
- **Рыночные риски.** Требования к приложению всегда могут измениться. Моделирование и итерационная разработка дают вам гибкость и быстроту реагирования на эти изменения.

На каждой итерации вы должны идентифицировать риски, распределять их по приоритетам и в первую очередь заниматься рисками с наивысшими приоритетами. Это позволит вам снизить первоочередные риски на ранних стадиях, когда у вас еще есть запас времени и вы можете перейти к альтернативным вариантам. Планы итераций *должны* учитывать возможность изменений. Сама рабочая обстановка должна быть ориентирована на изменения.

21.9. Резюме

Разработке программного обеспечения присущи проблемы, связанные с непониманием, неправильной оценкой и непредвиденными изменениями. Моделирование делает ваше приложение гибким. Итерационная разработка делает гибким процесс создания этого приложения. Она предоставляет возможность регулярной проверки хода работ и раннего обнаружения недостатков.

Итерационная разработка отличается от водопадного подхода. Водопадный подход предполагает идеальное предвидение и строгую последовательность разработки. Это неудачный подход, значение которого было сильно преувеличено в литературе.

Итерационная разработка отличается и от быстрого прототипирования. В последнем, сложные вопросы решаются методом исследований при помощи написания кода, который в случае выбора неудачного решения отбрасывается. Итерационная разработка подразумевает деление проекта на небольшие части, вероятность неудачи в рамках которых достаточно мала. Оба метода достаточно ценны.

Количество и длительность итераций зависят от масштабов проекта. Если итерации слишком малы, они создают чрезмерные накладные расходы. Если итерации слишком велики, контрольных точек для оценки хода проекта становится слишком мало. Обычно разумная длина итерации составляет от нескольких недель до нескольких месяцев. Определять охват итерации следует методом ранжирования рисков по приоритетам. Начинайте с самых серьезных рисков и переоценивайте приоритеты по завершении каждой итерации. Недопустимо затягивать график итерации, расширяя функциональность.

Интегрируйте подсистемы в единое целое в ходе разработки, не откладывая это до конца. Если команды разработчиков предоставить самим себе, они неизбежно разойдутся в своих предположениях и в описаниях интерфейсов. При слишком поздней интеграции вам придется отслеживать изменения или устранять различия в коде. В конце итерации должен получаться исполняемый выпуск системы, который должен быть протестирован. Исполняемый код является лучшей мерой хода выполнения.

Моделирование — естественное дополнение итерационной разработки. Оба метода повышают качество, продуктивность и предсказуемость разработки программного обеспечения. Некоторым разработчикам кажется, что моделирование замедляет разработку и мешает работе. Если вы моделируете быстро и охватываете моделью самую суть приложения, никому и в голову не придет сказать такое.

Разработка приложения связана с определенными рисками. Итерации должны быть структурированы таким образом, чтобы самые серьезные угрозы устранялись в первую очередь.

Таблица 21.1. Ключевые понятия главы

риски разработки	итерационная разработка	быстрое прототипирование
интеграция	моделирование	тестирование
итерация	прототип	водопадная модель

Библиографические замечания

[Larman-03] содержит подробный рассказ об истории итерационной разработки со множеством ссылок.

Литература

[Brooks-87] Frederick P. Brooks, Jr. No silver bullet; Essence and accidents of software engineering. IEEE Computer, April 1987, 10–19.

[Larman-03] Craig Larman and Victor R. Basili. Iterative and incremental development: a brief history. IEEE Computer, June 2003, 47–56.

[Rumbaugh-05] James Rumbaugh. The Unified Modeling Language Reference Manual, Second Edition. Boston: Addison-Wesley, 2005.

[Sotirovski-01] Drasko Sotirovski. Heuristics for iterative software development. IEEE Software, May/June 2001, 66–73.

Управление моделированием

22

Моделирование необходимо для разработки качественного программного обеспечения, но перейти к его практическому применению может быть довольно сложно. Можно искренне хотеть работать с моделями, но затягивать их фактическое использование. Моделирование требует изменений культуры, которые должны активно стимулироваться самой организацией.

22.1. Обзор управления моделированием

Мы часто сталкиваемся с тем, что сотрудники организации хотят использовать модели в своей работе, но не знают, с чего им следует начать. В этой главе мы рассказываем о том, каким образом можно внедрить моделирование на предприятии. Здесь мы затронем следующие темы:

- виды моделей (раздел 22.2);
- ловушки моделирования (раздел 22.3);
- сеансы моделирования (раздел 22.4);
- организация персонала (раздел 22.5);
- методики изучения (раздел 22.6);
- методики обучения (раздел 22.7);
- технические средства (раздел 22.8);
- оценка затрат на моделирование (раздел 22.9).

22.2. Виды моделей

Используемые на практике модели можно разделить на несколько категорий, каждая из которых обладает своим назначением, характеристиками и содержанием.

Люди часто путают модели между собой и забывают о причинах, по которым они решили создавать модель определенного типа.

- **Модель приложения.** Это одна из наиболее распространенных моделей, о которой и рассказывается в нашей книге. Модель приложения помогает разработчикам понять требования и закладывает фундамент для построения соответствующего приложения. В качестве примера можно привести наш пример с банкоматом.
- **Метамодель.** Метамоделю подобны моделям приложений, но сложнее их. Они часто используются для сложных приложений. В качестве примера можно привести каркасы (глава 14) и редактор моделей классов.
- **Модель предприятия.** Такая модель описывает целую организацию или какой-либо важный ее аспект. Модели приложений и метамодели используются для построения программного обеспечения. Модели предприятий предназначены совсем для другой цели, а именно для согласования концепций между несколькими приложениями и для понимания устройства предприятия. По своей природе модели предприятия имеют более широкую область охвата по сравнению с моделями приложений, но они редко оказываются более глубокими, так как им нужно описать только наиболее типичные аспекты. В качестве примера можно привести модель всего программного обеспечения какого-либо банка.
- **Оценка продукта.** Модели можно использовать при приобретении программного обеспечения. Вам нужно подготовить несколько моделей: модель ваших требований и модель программы для каждого из конкурирующих продуктов. Модель требований отличается от модели приложения только тем, что в ней могут отсутствовать мелкие детали, так как вы не собираетесь создавать по ней само приложение. Производители редко предлагают модели своих продуктов, но вы часто можете создать свою собственную модель при помощи инженерного анализа (reverse engineering — [Blaħa-04]). Модель поможет вам оценить качество приложения и понять его сильные и слабые стороны, а также область его применения. В качестве примера можно рассмотреть такую ситуацию. Если вы хотите купить программное обеспечение для банкомата, вместо того чтобы писать его самостоятельно, модель поможет вам сделать выбор.

22.3. Ловушки моделирования

Преимущества моделирования неоспоримы, но у него есть и недостатки, которые нужно стремиться обойти.

- **Паралич анализа.** Некоторые разработчики так глубоко уходят в моделирование, что им не удается его закончить. Такое часто случается с аналитиками, которые не занимаются собственно разработкой. Бывает это и с теми, кто только начинает заниматься моделированием, делает это неэффективно и не знает, когда его можно считать завершенным.

Решение: план проекта поможет вам избежать паралича на этапе анализа благодаря выделению определенного времени под конкретные задачи. План должен предусматривать затраты на моделирование и ожидаемый выход. Оценку качества модели и степени выполнения проекта вы можете получить при помощи формальных обзоров. Это полезно, если у тех, кто занимается моделированием, есть опыт разработки.

Параллельное моделирование. Нам приходилось сталкиваться с тем, что организации строили «запасные» модели, основанные на разных парадигмах. Часто бывает, что объектно-ориентированное моделирование осуществляется параллельно с моделированием баз данных без всякого взаимодействия между командами разработчиков. В командах объектно-ориентированных разработчиков большинство программистов не разбираются в базах данных. Профессионалы по работе с базами данных используют свои привычные методики и часто незнакомы с объектно-ориентированной методологией. В литературе между двумя концепциями лежит такая же пропасть: как сообщество объектно-ориентированных программистов, так и сообщество специалистов по базам данных имеет свой собственный стиль и жаргон, и лишь немногие специалисты могут считать себя принадлежащими к обоим лагерям одновременно. Ограничения существующих средств разработки еще более усиливают разделение.

Решение: как ни странно, проблема больше в терминологии и в стиле, а не в сути дела. Лучше всего просто не забывать о различиях культур. Пусть разработчики создают модели обоих типов, если им это удобно. Это один из способов обойти недостатки современных средств. Самое главное, чтобы разработчики работали с моделями. Если эти модели разные, то нужно, чтобы они часто согласовывались между собой. Здесь помогает итерационная разработка — разработчикам приходится согласовывать усилия на каждой итерации.

Неспособность мыслить абстрактно. Многим людям трудно мыслить абстрактно. Они неспособны научиться искусству моделирования. Они быстро переходят в режим программирования и не могут отойти от задачи на шаг и взглянуть на нее в целом.

Решение: единственное лекарство — как можно больше практики. Неопытным разработчикам, которые учатся моделировать, нужно решать упражнения. Они должны работать с реальными приложениями под руководством опытного наставника. Те, кто так и не научатся моделировать, должны быть задействованы в других задачах.

Избыточная область охвата. Модель должна отражать реальный мир, но лишь ту его часть, которая имеет отношение к вашим целям. Некоторые разработчики склонны к утрате сосредоточения, они помещают в свои модели много избыточных сведений. В принципе, неплохо охватывать моделью чуть больше, чем нужно для конкретного приложения, потому что до его создания точные его границы предсказать трудно. Но не стоит пытаться охватить гораздо больше, чем требуется, потому что такая модель будет спекулятивной и на ее основе тяжело будет создать что-то полезное.

Решение: чтобы обойти эту ловушку, используйте план проекта и проводите регулярные проверки (рецензирование). Эксперты по бизнесу должны понимать, чем занимаются разработчики модели, потому что таким образом они будут получать представление о возможностях и ограничениях создаваемого приложения.

- **Недостаток документации.** Слишком часто нам приходится сталкиваться с недокументированными моделями. Одних диаграмм недостаточно, к ним нужны пояснения. Комментарий должен объяснять читателю суть каждой диаграммы и определять используемую терминологию. Он должен разъяснять тонкости и обосновывать все противоречивые решения, а также описывать примеры, иллюстрирующие трудные вопросы.

Решение: вы должны настаивать на том, чтобы модели документировались комментариями или словарями данных. Не ленитесь читать эти комментарии.

- **Недостаток технического рецензирования.** Так же часто мы сталкиваемся с недостатком технического рецензирования. Каждый сотрудник или небольшая группа работает в одиночку и не хочет делиться своим опытом, знаниями и талантом. Разработчики не обсуждают друг с другом свои проекты, потому что обсуждение не влияет на подлежащие сдаче объемы работ и не поощряется руководством. Подобно тому, как эксперты по бизнесу являются источником требований к приложению, коллеги-разработчики могут рассказать вам много полезного о методах вычислений и о том опыте, который они приобрели в родственных приложениях. Формальные осмотры (рецензии) помогают устранить ошибки еще до этапа тестирования.

Решение: каждый проект должен пройти рецензирование хотя бы один раз, а лучше — несколько. Если осмотр один, его следует осуществить после завершения основы модели и архитектуры. Если осмотров будет несколько, их следует проводить после завершения моделирования и разработки архитектуры на каждой основной итерации. Руководству следует поддерживать свободный, критический и конструктивный стиль общения. Оно должно сделать финансирование проекта зависящим от рецензирования. Помните, что рецензии должны быть техническими, а не менеджерскими, то есть ориентированными на углубление технологии, а не на информирование руководства. В статье [Boehm-01] отмечается, что экспертная оценка позволяет обнаружить 60 % ошибок в программном обеспечении, поэтому техническое рецензирование, несомненно, является достаточно важным. Мы рекомендуем делать совещания, посвященные техническим осмотрам, небольшими (не более 10 человек) и приглашать на них только тех, кто активно участвует в проекте.

22.4. Сеансы моделирования

В главе 12 мы рассказывали об анализе предметной области — построении моделей реального мира, с помощью которых проясняются требования приложения. Когда вы приобретете опыт моделирования, вы сможете использовать другие

методы взаимодействия с пользователями для получения от них необходимых входных данных. Мы рассмотрим три альтернативы: скрытое моделирование, циклическое моделирование и моделирование на месте («в прямом эфире»), обсудим их преимущества и недостатки.

22.4.1. Скрытое моделирование

Самый популярный подход к моделированию выглядит так. Сначала разработчик общается с экспертами, записывает их комментарии (сформулированные требования или описания вариантов использования), а затем уходит к себе и занимается моделированием. Многие аналитики выбирают именно этот подход, потому что они могут сосредоточиться на словах пользователя во время общения с ним, а потом спокойно заняться самой моделью в одиночестве. На протяжении нескольких собраний пользователи отвечают на вопросы и самостоятельно делятся теми сведениями, которые кажутся им важными. После каждого собрания аналитик объединяет комментарии пользователей, в результате чего модель постепенно совершенствуется. В этом подходе обычно модель пользователям не показывают.

Лучше встречаться сразу с несколькими пользователями, чем беседовать с ними поодиночке. Пользователи стимулируют память друг друга. При беседе один на один пользователь может чувствовать себя скованно, чего не происходит на групповых совещаниях. Большинство аналитиков предпочитает встречаться с группами пользователей, объединенными по интересам. Например, на одном собрании могут присутствовать только продавцы, на другом — только конструкторы.

Скрытое моделирование (back-room modeling) обладает следующими преимуществами и недостатками.

- **Преимущества.** Предъявляет минимальные требования к опыту, а потому подходит разработчикам, которые только начинают заниматься моделированием.
- **Недостатки.** Циклическое взаимодействие с пользователями опытным разработчикам моделей кажется обременительным. Медлительность взаимодействия может представлять проблему и для пользователей, поскольку им приходится участвовать в нескольких интервью. Аналитiku приходится аккуратно записывать информацию, чтобы ничего не забыть.

22.4.2. Циклическое моделирование

Циклическое моделирование сложнее скрытого, но более эффективно с точки зрения определения требований. Аналитiku все равно приходится встречаться с небольшими группами пользователей, объединенными интересами или функциями, но при этом пользователи видят модель. Когда пользователи формулируют свои требования, аналитик выполняет прослеживание модели и пытается удовлетворить этим требованиям. Простые вопросы могут быть решены непосредственно во время собрания, а сложные — отложены на потом.

Мы назвали этот подход *циклическим моделированием* (round-robin modeling), потому что аналитик показывает модель каждой группе пользователей до тех пор, пока все их возможные потребности не будут учтены. Здесь требуется несколько

итераций, поскольку одна группа пользователей может поднять вопрос, требующий согласования с предыдущей группой. При скрытом моделировании группы тоже опрашиваются последовательно, но разница в том, что пользователи не видят саму модель.

Начинается циклическое моделирование с затравочной модели, основывающейся на имеющейся деловой документации. Не стоит начинать с пустого листа бумаги, потому что таким образом вы будете напрасно расходовать время и испытывать терпение пользователей. Затравочная модель стимулирует дискуссию. Пользователи видят, что аналитик хорошо подготовился, и могут сосредоточиться на сложных вопросах.

Во время собраний мы сообщаем пользователям, что они выступают в роли экспертов по бизнесу и что нам нужна их помощь в определении требований к системе. Мы выступаем в роли экспертов по компьютерам, поэтому заботу о деталях они могут предоставить нам. Обычно после этого пользователи облегченно вздыхают. Мы не задерживаемся на формализмах и объясняем используемые обозначения по мере продвижения работы. У участников редко возникают проблемы, потому что мы непрерывно объясняем свою модель.

Циклическое моделирование обладает следующими преимуществами и недостатками.

- **Преимущества.** Требуется меньше собраний, чем при скрытом моделировании. Поскольку модель находится перед глазами, аналитик может решить часть вопросов непосредственно во время собраний. При скрытом моделировании аналитик делает заметки, а потому может пропустить нужные ему детали.
- **Недостатки.** Циклическое моделирование все равно требует нескольких итераций, и оно малоэффективно для обмена идеями между группами пользователей. При наличии противоречий прийти к компромиссу может быть достаточно сложно. Аналитик выступает в невыгодной роли посредника между конфликтующими группами пользователей. Этот недостаток присущ также и скрытому моделированию. Некоторые пользователи не понимают моделей или боятся их, и аналитику приходится заботиться об этом.

22.4.3. Моделирование на месте

Моделирование на месте (live modeling) подходит для экспертов по моделированию. Мы приглашаем на собрание 10–20 человек с различными интересами (разработчиков, менеджеров, специалистов по различным областям бизнеса). Модель создается непосредственно в процессе ведения собрания, в соответствии с предложениями и комментариями. Обычно нам удается не отставать от диалога, а при необходимости мы берем тайм-аут. Модель, которую мы строим при помощи редактора, отображается проектором на большой экран. Сеанс обычно длится два часа, трех сеансов бывает достаточно для построения 80 % структуры модели с 50 классами. Большие и сложные модели требуют дополнительных сеансов. Между сеансами разработчик чистит диаграммы, документирует модель и решает оставшиеся вопросы.

Процесс стимулируется размером и разнообразием состава группы. Те, кто поначалу участвовали в дискуссии неохотно, видят реакцию других и чувствуют потребность вступить в беседу, чтобы защитить свою точку зрения. Комментарии одного участника обычно вызывают комментарии другого. Особенно успешно идет моделирование в присутствии скептиков.

Моделирование на месте можно начинать с пустого листа, но аналитик должен заранее подготовиться и изучить будущее приложение. Идеи поступают очень быстро, и аналитик должен быть готов к этому. Иногда мы подготавливаем затравочную модель, если у нас есть для нее информация. Обычно мы просим клиента подготовить список требований для стимулирования дискуссии в случае пауз.

Мы активно участвуем в беседе, а не просто записываем информацию. Мы задаем вопросы и уточняем ответы, которые кажутся нам неудовлетворительными. Мы вносим предложения, основываясь на своем опыте. Окончательное решение всегда остается за экспертами по бизнесу. Время от времени мы сталкиваемся со сложными проблемами моделирования, решение которых мы откладываем до следующего собрания.

Часто активную дискуссию вызывают названия. Это может быть полезно, потому что хорошие названия исключают неправильное понимание модели. Дискуссии стимулируют выдачу связанной информации. Мы заставляем экспертов по бизнесу придумывать хорошие названия — краткие, четкие и не допускающие разных интерпретаций.

Моделирование на месте обладает следующими преимуществами и недостатками.

- **Преимущества.** Для мастеров моделирования это наилучший способ получения входных данных от пользователей. Мы постоянно практикуем моделирование на месте, и клиентам нравится быстрое продвижение вперед. Участники обладают разными знаниями и по-разному воспринимают предметную область. Совместное участие в одном совещании дает им возможность согласовать свои взгляды. Значительный положительный эффект дается тем, что участникам приходится разговаривать друг с другом. Люди, у которых обычно нет времени и желания общаться, оказываются вместе и ведут беседу.
- **Недостатки.** Аналитик должен быть очень опытным в моделировании, способным вести собрания и умеющим обращаться с редактором моделей. Немногие разработчики сочетают в себе эти качества. Моделирование на месте удобно для выявления структуры системы — классов и отношений между ними. Оно не так эффективно для поиска атрибутов, потому что большую группу людей трудно заставить сосредоточиться на мелких деталях. Атрибуты следует извлекать из других источников информации. Моделирование на месте непригодно для сложных приложений, например приложений с метамоделями. Для таких задач мы рекомендуем использовать скрытое моделирование.

В табл. 22.1 сведены преимущества и недостатки трех видов моделирования.

Таблица 22.1. Преимущества и недостатки различных подходов к проведению сеансов моделирования

	Скрытое моделирование	Циклическое моделирование	Моделирование на месте
Суть метода	Запись комментариев пользователей и построение модели в их отсутствие	Модель демонстрируется пользователям, но разрабатывается в их отсутствие	Модель разрабатывается в процессе встречи со всеми пользователями
Требуемый опыт	Низкий	Средний	Очень высокий
Производительность (для модели с 50 классами)	Низкая (около 15 собраний по 2 часа)	Средняя (около 12 собраний по 2 часа)	Очень высокая (около 3 собраний по 2 часа)
Рекомендация	Оптимально для неопытного разработчика моделей	Оптимально для разработчика с опытом	Оптимально для очень опытного разработчика

22.5. Организация персонала

На рис. 22.1 показано решение по организации персонала, оптимальное для крупной организации. Несколько экспертов объединяются в технологическую группу, которая обслуживает разработчиков, организованных по областям деятельности. В табл. 22.2 описываются роли технологической группы и прикладных групп.



Рис. 22.1. Корпоративная структура

Технологическая группа работает в масштабе целой организации. Она распространяет стандарты и методики вычислений, обслуживает прикладные группы. В эту группу следует поместить лучших специалистов по моделям, чтобы их опыт был доступен всем остальным. Технологическая группа не должна заниматься созданием приложений, для этого есть прикладные группы. Технологическая группа должна отвечать за сложные вопросы. Размер группы должен быть небольшим, чтобы организация не несла слишком больших расходов.

Роль прикладных групп несколько иная. Они должны изучать область деятельности и передавать знания между родственными приложениями. Члены прикладных групп должны тесно взаимодействовать с коллегами из той же области деятельности. Разработчики должны хорошо владеть средствами моделирования (хотя лучшие из них находятся в технологической группе).

Таблица 22.2. Технологические и прикладные группы

	Технологическая группа	Прикладная группа
Перспектива	Организация в целом	Область деятельности
Задачи	Распространение стандартов и методик вычислений, обслуживание моделей предприятия, поддержка прикладных групп	Создание приложений, оценка продуктов для приобретения
Требуемый опыт моделирования	Эксперт	Свободное владение
Количество групп	Одна	Несколько
Размер группы	Небольшой (в целях ограничения расходов)	Определяется потребностями области деятельности

В некоторых фирмах используются иные организационные структуры: все разработчики моделей помещаются в технологическую группу и по необходимости выделяются в помощь разработчикам приложений. Мы не рекомендуем использовать такую схему. Моделирование настолько сильно стимулирует понимание сути и диалог между сотрудниками, что его нужно распространять по всему предприятию. (По этой причине оказывается полезно обучить моделированию бизнесменов и маркетологов.) Моделирование — это лингва-франка разработчиков программного обеспечения. Разработчики приложений должны сами создавать модели для своих нужд.

22.6. Методики изучения

Для изучения моделирования существует несколько методик. Некоторые подходят для самостоятельного использования. Другие требуют поддержки со стороны организации. Этот раздел ориентирован как на студентов, так и на практиков.

- **Обучение и воспитание.** Университеты и коммерческие учебные учреждения предлагают курсы обучения концепциям моделирования и их применению. Обучение лучше проходить незадолго до начала реального проекта (в идеале — за несколько недель). Постарайтесь не увеличивать промежутков до начала работы, иначе вы успеете забыть слишком много. Обучение следует укреплять воспитанием. Разработчикам нужна помощь в применении изученного материала. Неопытным разработчикам моделей может не хватать уверенности в правильности их действий. Недостаточно привлечь внешние ресурсы к обслуживанию проекта. Необходимо обеспечить передачу знаний от сторонних разработчиков к своим собственным.
- **Командная работа.** Модели приложений должны создаваться небольшими командами, которые изначально должны состоять из разработчиков, экспертов по бизнесу и внешних консультантов. После создания нескольких приложений внешние консультанты уже не потребуются. Командная

работа позволяет распространять знания о компьютерных технологиях и о бизнесе внутри фирмы.

- **Семинары.** Периодическое проведение семинаров эффективно для образования сотрудников. Фирма должна способствовать проведению технических семинаров разработчиками. На семинарах разработчики беседуют и обмениваются идеями. Они узнают о различных проектах и обретают знания для продвижения своих собственных проектов. Семинары обеспечивают разработчикам взаимную поддержку в борьбе с трудностями моделирования.
- **Непрерывное образование.** Разработчики должны стремиться к поиску новых идей и применению лучших методов, выработанных программистским сообществом. Периодическое посещение технических конференций и профессиональных собраний очень полезно в этом отношении. Ценные идеи можно черпать из книг и журналов.
- **Техническое рецензирование.** Формат технического рецензирования способствует общению и обогащает опыт как разработчика, так и рецензентов (см. раздел 22.3).

22.7. Методики обучения

Существует несколько методов обучения моделированию, обладающих определенными преимуществами и недостатками.

- **Ознакомление.** Моделировать можно научиться только в процессе моделирования. В течение нескольких лет мы испытывали различные методики и пришли к выводу, что быстрее всего начать моделирование можно с ознакомления. Во время сеансов моделирования (см. раздел 22.4) мы пропускаем введение в моделирование (несмотря на то, что многие присутствующие никогда раньше не слышали о нем) и сразу приступаем к моделированию задачи. Через несколько сеансов присутствующие уже продвигаются довольно далеко в работе над своим приложением и одновременно узнают довольно много о самом моделировании.
- **Практика.** Студентам необходим большой объем практических занятий и упражнений. Они должны решать задачи из самых разных областей, с различными входными данными, на разных уровнях абстрагирования и разной сложности.
- **Исправление ошибок.** Студенты часто допускают ошибки в процессе моделирования, и предусмотреть их все бывает довольно сложно. Исправление ошибок представляет собой значительную часть процесса обучения. Студенты могут учиться на своих ошибках и на ошибках своих сокурсников в процессе совместной работы с ними и на презентациях. В учебном учреждении можно сделать презентации обязательными. На наших учебных курсах мы обычно делаем их добровольными.

- **Реализация.** Очень важно, чтобы студенты понимали простоту реализации моделей. Они должны понимать, что любую модель, которую они могут придумать, можно реализовать в устойчивой, предсказуемой и эффективной системе. Как только они поймут это, они должны перестать заботиться о реализации и мыслить непосредственно в терминах моделей.
- **Ученичество.** Преподавать можно не только группе, но и индивидуально. Новичок может обучиться моделированию, работая над каким-либо проектом совместно с опытным разработчиком. Это обучение лучше всего подходит для того, кто уже начал изучать моделирование самостоятельно.
- **Образцы.** Когда мы создаем модели, мы всегда мыслим в терминах образцов (глава 14). Мы сталкиваемся с различными ситуациями, анализируем лежащую в основе математику, а затем переходим к образцу. Образец — это испытанное верное решение стандартной задачи, которое было проанализировано экспертами и признано эффективным. Существует множество видов образцов. Есть образцы анализа, архитектуры, проектирования и реализации. Самая большая проблема — понять, когда нужно применить образец. Кроме того, образец никогда не может охватить всю модель целиком.

22.8. Средства

Для решения любой серьезной задачи по моделированию необходимы специальные средства — редакторы моделей, средства для управления конфигурацией, генерирования кода, симуляции, компиляции, отладки, профилирования и т. д. Мы не пытаемся перечислить здесь все доступные средства, потому что их слишком много. Мы рассмотрим только те средства, которые непосредственно относятся к моделированию, и упомянем некоторых доминирующих производителей.

22.8.1. Средства моделирования

Для больших приложений (50 классов и более) требуются тяжеловесные средства моделирования. Повышение производительности — отнюдь не главное преимущество средства моделирования. Главное состоит в том, что оно может сделать проникновение в суть задачи более глубоким. Средства моделирования помогают экспертам работать быстрее, они упорядочивают информацию о классах в удобном для поиска формате. Новичкам они помогают следить за синтаксисом и избегать наиболее распространенных ошибок. В качестве примеров таких средств можно привести IBM Rational Rose XDE, Rhapsody, Magic Draw, Together/J и Enterprise Architect.

Небольшие приложения менее требовательны к средствам. Однако и здесь средство моделирования может оказаться полезным, потому что моделирование обеспечивает ясность мышления, а использование средства упрощает конструирование модели. Учитывая доступность недорогих средств моделирования, а также наличие лицензирования на предприятие, разумную причину для отказа от средства моделирования придумать довольно сложно.

22.8.2. Средства управления конфигурациями

Крупные приложения состоят из множества файлов: файлов программ (исходный код, компилированный код, исполняемый код), документации (для пользователей, администраторов, обслуживающих) и данных (конфигурации, метаданных, тестовых данных). На практике бывает сложно координировать все эти файлы. Для решения этой задачи и служат средства управления конфигурациями. Они повышают эффективность работы разработчиков и сокращают вероятность потери полезных файлов.

В книге [Pressman-97] перечисляются пять главных задач, составляющих управление конфигурациями.

- **Идентификация.** Средство управления конфигурациями должно обеспечивать идентификацию каждого файла конфигурации и его связь со всеми прочими файлами.
- **Управление версиями.** Организация должна иметь возможность отслеживать копии файла в процессе его изменения с течением времени. Иногда бывает необходимо вернуться к старым файлам (например, для определения причин ошибки).
- **Управление изменениями.** Организация должна определить ответственного за принятие изменений, а также за синхронизацию деятельности сотрудничающих разработчиков. Наиболее распространен контроль файлов в момент открытия для изменений и сохранения обновленной версии (check-in and check-out control).
- **Аудит.** Средство управления конфигурациями хранит журнал доступа. Пользователи могут обратиться к журналу и определить даты изменения файлов и авторов этих изменений.
- **Отчет о состоянии.** Необходимо иметь возможность уведомлять сотрудников об изменениях.

Если моделирование осуществляет один человек, ответственно относящийся к резервному копированию, он может обойтись и без средства управления конфигурациями. По мере увеличения количества моделей и разработчиков такой подход становится все более затруднительным. Для управления несколькими моделями и координации усилий нескольких человек нужно использовать средства управления конфигурациями.

Наиболее известны IBM Rational ClearCase, Merant PVCS, Microsoft Visual SourceSafe, CVS.

22.8.3. Генераторы кода

Многие средства моделирования позволяют генерировать код приложения. Обычно такое средство может генерировать объявления данных для программного кода и для базы данных (если вы работаете с ней). Некоторые средства могут генерировать и алгоритмы, но эту задачу автоматизировать сложнее. Например, такое средство может генерировать программный код по диаграмме состояний.

Независимо от используемого средства разработчики должны с осторожностью относиться к генерированному коду, проверять его на правильность и эффективность. Некоторые средства генерируют плохой код с малозаметными ошибками. Другие средства, к нашему удивлению, допускают грубые ошибки в коде. Если программисты будут читать код, генерируемый программным средством, они будут лучше представлять, что именно оно делает.

22.8.4. Средства интерпретации моделей

Для предсказания поведения и производительности готового приложения можно использовать средства симуляции моделей. Некоторые средства моделирования содержат встроенные механизмы для этой цели. В качестве примера можно привести программу i-Logix Statemate, которая может симулировать диаграммы состояний.

22.8.5. Репозиторий

Репозитории важны для разработки приложений, потому что в них хранятся метаданные, позволяющие различным средствам взаимодействовать друг с другом. Репозиторий является посредником между программными средствами. Поскольку репозиторий работает с метаданными, внедрить его довольно сложно, но зато работа с ним может увеличить эффективность остальных средств. Репозитории производятся Allen System Group, Computer Associates, IBM и Microsoft.

22.9. Оценка затрат на моделирование

Разработка программы — это коммерческое предложение. Бизнесмены оценивают затраты на создание программы, ожидаемую выручку и экономию. Моделирование обычно составляет малую часть (гораздо менее 10 %) общих затрат на создание приложения. Затраты на моделирование зависят от следующих факторов.

- **Сложность приложения.** Конкретные приложения всегда проще, чем абстрактные. Проще создать программу для обработки звонков клиентов, чем создать систему для всех видов взаимодействия с клиентами.
- **Профессионализм.** Опытный разработчик может работать на порядок быстрее неопытного. Кроме того, опытный разработчик скорее создаст качественную модель с правильным абстрагированием.
- **Средства.** Хорошо, если разработчик может работать с мощным средством моделирования и умеет это делать.
- **Размер модели.** Время на создание модели нелинейно возрастает с ее размером. Можно считать, что время создания пропорционально количеству классов в степени 1.5. Таким образом, создание модели с 500 классами требует в 30 раз больше времени, чем создание модели с 50 классами.
- **Рецензирование.** Рецензирование сокращает количество итераций, необходимых для построения модели.

С учетом всех этих факторов большинство моделей создаются за срок от двух недель до шести месяцев.

22.10. Резюме

В этой главе мы рассмотрели несколько тем, призванных помочь внедрению технологии моделирования на предприятии.

Модели можно разбить на несколько категорий, с разным назначением, характеристиками и содержанием. Многие разработчики не учитывают это. Например, модель предприятия не может содержать все особенности приложений, потому что иначе она станет неудобной. Модели могут использоваться для оценки приобретаемых продуктов.

Модели обладают множеством достоинств, но, как и любая другая технология, приводят к дополнительным рискам. Мы идентифицировали основные риски и указали действия по их снижению.

Существует несколько подходов к взаимодействию с пользователями в процессе моделирования. Мы перечислили некоторые виды взаимодействия (скрытое моделирование, циклическое моделирование, моделирование на месте), их преимущества и недостатки.

Чтобы изучить моделирование, сотрудник может прибегнуть к различным методам (обучение, воспитание, командная работа, семинары, непрерывное обучение, техническое рецензирование).

Учить моделированию тоже можно по-разному. Наиболее успешно ознакомление на практике, потому что изучить моделирование можно только в процессе моделирования. Преподавателям следует проводить со студентами практические занятия по созданию моделей и заниматься с ними исправлением ошибок. Студенты должны понять, что модель всегда можно реализовать. Опытные разработчики моделей могут распознавать и применять образцы.

Для разработки серьезного приложения необходимо использовать программные средства. Мы перечислили некоторые существующие программы и указали критерии для их выбора, а также для оценки затрат на моделирование.

Библиографические замечания

В этой книге мы почти не говорим о вопросах, связанных с управлением, таких как планирование проекта, оценка проекта, оценка затрат, метрики, оценка персонала и динамика команд. Эти важные темы рассмотрены в других книгах, таких как [Pressman-97] и [Blaha-01].

В книге [Colwell-03] автор описывает свой собственный опыт работы с рецензиями проектов и подчеркивает их важность для качества программ.

[Berndtsson-04] описывает свой девятилетний опыт преподавания курсов объектно-ориентированного анализа и проектирования. Он приходит к следующим заключениям:

- **Успешность программирования — не индикатор успешности моделирования.** 65 % студентов, успешно завершивших курс объектно-ориентированного программирования, плохо справлялись с объектно-ориентированным моделированием.

- **Успешность моделирования может служить индикатором успешности программирования.** 84 % студентов, успешно завершивших курс моделирования, справились и с программированием.
- **Основная трудность моделирования — абстрактность.** Оценки курса объектно-ориентированного моделирования в высокой степени коррелируют с курсом распределенных систем (столь же высокий уровень абстрактности).

[Box-00] заключает, что объектно-ориентированная технология требует более высокого уровня абстрагирования, чем структурное программирование. Изучение абстракции требует значительных усилий.

Литература

[Berndtsson-04] Mikael Berndtsson. Teaching object-oriented modeling and design. Draft paper, 2004.

[Blaha-01] Michael R. Blaha. A Manager's Guide to Database Technology: Building and Purchasing Better Applications, Upper Saddle River, NJ: Prentice Hall, 2001.

[Blaha-04] Michael Blaha. A copper bullet for software quality improvement. IEEE Computer, February 2004, 21–25.

[Boehm-01] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. IEEE Computer, January 2001, 135–137.

[Box-00] Roger Box and Michael Whitelaw. Experiences when migrating from structured analysis to object-oriented modeling. Fourth Australasian Computing Education Conference, Melbourne, Australia, December 4–6, 2000, 12–18.

[Colwell-03] Bob Colwell. Design reviews. IEEE Computer, October 2003, 8–10.

[Pressman-97] Roger S. Pressman. Software Engineering: A Practitioner's Approach, Fourth Edition. New York: McGraw-Hill, 1997.

Унаследованные системы

23

Чаще всего разработка — это не создание новых приложений, а усовершенствование существующих. Очень редко приходится творить новое приложение с нуля. Даже в этом редком случае обычно приходится собирать информацию из существующих приложений и обеспечивать интеграцию с ними. Вы можете использовать технические требования, идеи, данные и код унаследованных приложений.

Сложно изменить приложение, если вы не понимаете его устройство. Если приложение было разработано с использованием объектно-ориентированных моделей, причем моделей точных, они помогут вам понять и усовершенствовать это приложение. Если моделей не было или они были утеряны, вам придется начать с построения модели существующей структуры.

23.1. Инженерный анализ

Инженерный анализ (reverse engineering) — процесс изучения артефактов реализации и выявления лежащей в их основе логики. Инженерный анализ применялся к аппаратному обеспечению: разгадывание проектов по готовым продуктам было широко распространено [Rekoff-85]. Модели помогают инженерному анализу, потому что они позволяют выражать как абстрактные концепции, так и детали реализации.

При построении новых приложений инженерный анализ позволяет сохранить полезную информацию из старых приложений. Не обязательно увековечивать старые ошибки. Аналитик должен определить, что следует сохранить, а что — отбросить. Модели, полученные в результате такого анализа, следует рассматривать как источник требований для нового приложения.

Инженерный анализ требует принятия решений, интерпретации системы разработчиком и поэтому не может быть полностью автоматизирован. Программные средства позволяют механически проанализировать весь объем кода, но им сложно выявить человеческие решения. Многие средства позволяют генериро-

вать начальную модель, но эта модель будет немногим более, чем визуальным представлением структуры программы. Аналитик должен справиться по меньшей мере с двумя проблемами: получением малопонятной или утраченной информации и описанием скрытого поведения.

23.1.1. Инженерный анализ и разработка

Таблица 23.1 демонстрирует, что инженерный анализ противоположен по смыслу нормальной разработке (forward engineering): вы начинаете с существующего приложения и пытаетесь вернуться к требованиям, по которым оно строилось.

Таблица 23.1. Инженерный анализ и разработка

Разработка	Инженерный анализ
Разработать приложение по имеющимся требованиям	Определить требования по имеющемуся приложению
Более точная. У разработчика есть требования, и он обязан создать приложение, которое будет им отвечать	Менее точный. Реализация может дать различные требования, в зависимости от интерпретации аналитика
Предписывающая. Разработчикам говорят, как им следует работать	Адаптивный. Аналитик должен понять, что именно сделал разработчик
Более зрелая. Широко доступен опытный персонал	Менее зрелый. Опытные аналитики встречаются редко
Требует времени. Может занимать месяцы и даже годы	Может быть выполнен в 10–100 раз быстрее, чем разработка (дни или недели)
Модель должна быть корректной и полной, иначе приложение не будет работать	Модель может быть несовершенной. Частичная информация все равно будет полезной

23.1.2. Входные данные для инженерного анализа

При проведении инженерного анализа нужно быть изобретательным и рассматривать все возможные источники входных данных. Доступная информация сильно зависит от конкретной задачи.

- **Программный код.** Исходный код программы может быть богатым источником информации. Специальные средства помогут вам понять поток управления и структуры данных программы. Комментарии, описательные имена переменных, функций и методов еще более углубят ваше понимание системы.
- **Структура базы данных.** Если приложение работает с базой данных, вы можете многое узнать из этой базы. База данных описывает структуру данных и задает ограничения на них, причем точным и явным образом.
- **Данные.** Если доступны данные, вы можете определить большую часть их структуры. Хорошо написанная программа и дисциплинированные пользователи обычно дают более качественные данные, нежели предписывает структура данных. В больших системах можно воспользоваться небольшим

объемом данных, чтобы получить предварительные заключения, а затем проверить эти заключения на дополнительных данных. Проверка безусловно не может доказать правильность заключений, но чем больше данных вы проверите, тем более вероятной будет правильность заключений.

- **Формы и отчеты.** Осмысленные заголовки и формат проясняют структуру данных и логику их обработки. Определения форм и отчетов особенно полезны, если доступна их связь с переменными. Эмпирический подход состоит во вводе известных, но необычных значений переменных для определения связей между формами и переменными.
- **Документация.** Задачи отличаются друг от друга качеством, количеством и характером документации. Документация создает контекст инженерного анализа. Особенно полезно иметь руководство пользователя. Может иметься и словарь данных (список важных сущностей в программе с их определениями). Относитесь к документации с осторожностью, потому что она может оказаться устаревшей и несогласованной с самим приложением.
- **Понимание приложения.** Если вы хорошо понимаете приложение, вы сможете лучше разработать интерфейсы. Может быть, вам удастся связаться с экспертами по данному приложению, которые ответят на ваши вопросы и приведут обоснования своих решений. Вы можете еще более укрепить свою модель, воспользовавшись сведениями из родственных приложений.
- **Тестовые ситуации.** Тестовые ситуации позволяют проверить нормальный поток управления и необычные ситуации. Иногда они дают важные сведения.

23.1.3. Выходные данные инженерного анализа

Инженерный анализ дает несколько видов полезных выходных данных.

- **Модели.** Модель отражает область применения и назначение приложения. Она является отправной точкой для понимания исходной программы и построения любых новых программ.
- **Отображения.** Вы можете связать атрибуты модели с переменными. С меньшей точностью вы можете связать программный код с моделями состояний и взаимодействий.
- **Журналы.** Аналитики должны записывать свои наблюдения и нерешенные вопросы. Журнал отражает принимаемые решения и их обоснование.

23.2. Построение модели классов

Всегда начинайте с построения модели классов приложения. Это даст вам возможность понять классы и отношения между ними. Мы рекомендуем строить модель классов в три этапа, которые мы называем восстановлением реализации, восстановлением проекта и восстановлением анализа.

23.2.1. Восстановление реализации

Прежде всего постарайтесь быстро изучить приложение и создайте начальную модель классов. Если программа написана на объектно-ориентированном языке, вы можете восстановить классы и обобщения непосредственно. В противном случае вам придется изучать структуры данных и операции и определять классы вручную. Система может быть спроектирована не очень удачно, поэтому результат может оказаться неудовлетворительным. Старайтесь не делать никаких дальнейших предположений, ограничьтесь только определением классов. Полезно иметь начальную модель, описывающую реализацию системы.

23.2.2. Восстановление проекта

Теперь нужно протестировать приложение таким образом, чтобы восстановить ассоциации. Обычно ассоциация реализуется как атрибут-указатель. Кратность ассоциации в прямом направлении обычно бывает очевидна. Кратность в противоположном направлении обычно не декларируется, и вам придется определить ее из общих соображений или путем изучения кода.

Во многих системах для реализации ассоциации с кратностью «много» используются совокупности указателей. В этом случае изначальная модель будет указывать на класс совокупности, а не на класс элементов. Вы должны перенести ассоциацию в класс элементов и соответствующим образом скорректировать кратность. Классы совокупностей — это механизмы реализации, которые не должны присутствовать в большинстве аналитических и проектных моделей. Вы можете указать эти классы как рекомендуемую реализацию для ассоциаций в направлении с кратностью «много».

Иногда при помощи указателей ассоциации реализуются в обоих направлениях. В этом случае необходимо идентифицировать парные указатели и объединять их в одну ассоциацию. Необходимо проверить все классы, в которых имеются парные указатели друг на друга. Если вы работаете с программным средством, оно создаст по одной ассоциации на каждый указатель. Вы должны удалить одну из ассоциаций и перенести соответствующую информацию на вторую ассоциацию.

Нижнее ограничение на кратность обычно имеет значение 0 или 1. Значение 0 соответствует ситуации, когда целевой объект инициализируется где-то в исходном коде, но момент проведения инициализации неочевиден. Значение 1 означает, что целевой объект инициализируется в момент создания объекта, например, в конструкторе или блоке инициализации.

23.2.3. Восстановление анализа

На последнем этапе вы должны интерпретировать модель, уточнить ее и сделать более абстрактной. Удалите все оставшиеся артефакты реализации, устраните ошибки. Избавьтесь от избыточной информации или пометьте ее как избыточную. Это подходящий момент для пересмотра модели. Является ли она удобочитаемой и согласованной? Согласуйте результаты инженерного анализа с моделями других приложений и с документацией. Покажите свою модель экспертам по приложению и учтите их рекомендации.

Если исходный код не был объектно-ориентированным, вам придется выводить обобщения исходя из подобия и различий структуры и поведения. Для выделения агрегаций и композиций требуется понимание приложения и внимательное изучение кода. Например, одинаковое время существования объектов может указывать на их участие в композиции. Понимание приложения и кода требуется также для определения квалификаторов и классов ассоциации.

Вы можете добавить в модель пакеты, при помощи которых организуются классы, ассоциации и обобщения. Классы из нескольких файлов можно объединить в один пакет или, наоборот, разбить большой файл кода по нескольким пакетам.

23.3. Построение модели взаимодействия

Назначение каждого метода обычно бывает достаточно очевидным, а то, каким образом объекты взаимодействуют друг с другом для решения стоящих перед системой задач, обычно бывает достаточно сложно понять просто исходя из кода. Проблема в присущей коду редуцированности: он описывает работу каждой отдельной составляющей. Система же имеет смысл только как целое: взаимодействия объектов друг с другом придают ей этот смысл. Расширить ваше понимание системы поможет модель взаимодействия.

Чтобы добавить методы в модель классов, вы можете воспользоваться разделением на слои. В данном случае слой (slice) — это подмножество программы, сохраняющее определенную проекцию ее поведения [Weiser-84]. Для разделения на слои нужно выбрать некоторый исходный код, подлежащий сохранению. Затем нужно пометить все операторы, используемые этим кодом. Таким образом, вы получите проекцию фрагмента поведения исходной программы. Разделение на слои позволяет преобразовать процедурный код в объектно-ориентированное представление.

Для программистов, как отмечает [Weiser-84], мышление в терминах слоев довольно естественно. Удобство работы с ними обуславливается следующими факторами: 1) их можно идентифицировать автоматически; 2) слои получаются меньше, чем исходная программа; 3) они выполняются независимо друг от друга; 4) каждый слой в точности воспроизводит проекцию поведения исходной программы [Weiser-84].

Для представления полученного таким образом метода можно использовать диаграмму деятельности. Она поможет вам понять последовательность обработки и поток данных между различными объектами. После этого вы сможете сконструировать диаграммы последовательности из диаграмм деятельности, чтобы упростить дальнейшую работу.

23.4. Построение модели состояний

Если вы занимаетесь анализом пользовательского интерфейса, модель состояний может быть довольно полезна. В остальных случаях модели состояний не имеют особого значения.

Если вам нужно построить модель состояний, действуйте следующим образом. В входе у вас имеются диаграммы последовательности, полученные после построения модели взаимодействий. Вы должны объединить диаграммы последовательности одного класса, упорядочив события и добавив условия и циклы (см. главу 13).

Изучение кода и проведение динамического тестирования позволит вам дополнить диаграммы последовательности. Полезно найти все возможные состояния каждого класса, для которого нужно сформулировать модель состояний. Инициация и завершение соответствуют конструкторам и деструкторам объектов.

23.5. Рекомендации по проведению инженерного анализа

В этом разделе мы собрали несколько важных подсказок, которые следует иметь в виду в процессе выполнения инженерного анализа и построения моделей классов, взаимодействия и состояний.

- **Отделяйте предположения от фактов.** Инженерный анализ выдает множество гипотез. Вы должны достичь полного понимания приложения прежде, чем сможете сделать твердые заключения. По мере проведения анализа вам придется возвращаться к предыдущим решениям, проверять и изменять их.
- **Приспосабливайтесь.** Процесс инженерного анализа должен соответствовать задаче. Сами задачи и доступные данные очень сильно отличаются в разных случаях. Для инженерного анализа вам понадобятся мозги — это все равно, что решать большую головоломку.
- **Будьте готовы к различным интерпретациям.** При инженерном анализе ответ может быть не единственным. Альтернативные интерпретации дают разные модели. Чем больше имеется информации, тем меньше остается простора для индивидуальных интерпретаций.
- **Не бойтесь приближенных результатов.** Вполне допустимо потратить разумное количество времени на извлечение 80 % информации о сути приложения. Получить оставшиеся 20 % вы сможете с помощью методик обычной разработки (таких как интервью с опытными пользователями). Многих нервирует неточность результатов, поскольку она означает работу с иной парадигмой, отличающейся от парадигмы разработки приложений.
- **Будьте готовы к нестандартным конструкциям.** Иногда даже опытные разработчики используют необычные конструкции. В некоторых случаях вам не удастся построить полную и точную модель, потому что такой модели никогда не существовало.
- **Следите за согласованностью стиля.** Программное обеспечение обычно разрабатывается в рамках единой стратегии. Даже нарушения хорошего стиля должны быть одинаковыми во всем приложении. Поэтому вы можете определить стратегию даже по части приложения.

23.6. Обертка

Иногда попадаются хрупкие и трудные для понимания приложения, написанные очень давно, с утерянной документацией и без поддержки разработчиков. Любые изменения могут угрожать жизнеспособности этих приложений. Поэтому многие организации стремятся ограничить внесение изменений в такие приложения. Вместо этого код изолируют, а вокруг него строят обертку.

Обертка (wrapper) — это совокупность интерфейсов, управляющих доступом к системе. Она состоит из пограничных классов, предоставляющих соответствующие интерфейсы, а потому должна разрабатываться в соответствии с принципами объектно-ориентированного программирования. Методы пограничных классов вызывают методы системы, используя существующие операции. Пограничный метод может задействовать несколько существующих операций и объединять данные из различных источников. Вызовы методов унаследованного приложения могут выглядеть сумбурно, но пограничные классы скрывают это от внешних клиентов. Исходный код может и не быть объектно-ориентированным. Обертки сохраняют форму унаследованного программного обеспечения и дают возможность использовать его функциональность.

Функциональность существующих приложений может быть трудной для понимания, непредсказуемой или сложной. Часто достаточно бывает извлечь ядро функциональности, наиболее фундаментальное, простое в использовании и хорошо протестированное. В этом случае обертка даст удобный интерфейс этой функциональности. Некоторые составляющие старого приложения теряются, но обычно лишь наиболее сомнительные.

Новую функциональность обычно можно добавить в виде отдельного пакета. Проектируйте этот пакет в соответствии с принципами объектно-ориентированного программирования. Старайтесь минимизировать взаимодействие с существующей системой и сохранять однородность этого взаимодействия.

В качестве примера рассмотрим веб-приложение. Исходное банковское приложение написано на Коболе и работает на старом аппаратном обеспечении. Обертка предоставляет доступ к логике Кобола через объектно-ориентированные методы, которые можно присоединить к современному веб-интерфейсу. Унаследованный код все равно будет нуждаться в обслуживании, но на это никак не повлияет существование веб-интерфейса. Фактически исполняемый код веб-приложения может быть неаккуратным, но он будет работать до тех пор, пока обслуживание логики на Коболе не будет затрагивать методы обертки.

Обертка обычно используется лишь в качестве временного решения, потому что она бывает серьезно ограничена организацией унаследованного программного обеспечения (обычно акцидентальной). Сама по себе комбинация старого кода, обертки и нового кода в новом формате становится настолько неудобной, что ее приходится переписывать заново.

Снид предлагает использовать XML для взаимодействия унаследованного программного обеспечения с внешним миром [Sneed-01]. Он также отмечает, что обертки служат не только технологическим, но и общественным целям. Тех, кто занимается обслуживанием старого кода, обертка не беспокоит. Обертки помогают

программистам сохранять неизменными их мысленные модели программного обеспечения. Это помогает в повседневном обслуживании заключенного в обертку кода.

23.7. Обслуживание

В литературе по программному обеспечению обслуживание рассматривается как нечто монолитное, но мы предпочитаем точку зрения Райлича и Беннетта. Согласно [Rajlich-00], программное обеспечение проходит пять стадий:

- **Начальная разработка.** Разработчики создают программное обеспечение.
- **Развитие.** Программа претерпевает серьезные изменения в функциональности и архитектуре. Для поддержания его качества может быть использован рефакторинг.
- **Обслуживание.** Доступные технические таланты сокращаются. Изменения сводятся к небольшим исправлениям и простым корректировкам функциональности. На этом этапе программа начинает неуклонно устаревать. Здесь уместно использовать технологию создания оберток.
- **Постепенное прекращение работы.** Производитель продолжает получать прибыль, но уже планирует вывод программы из употребления.
- **Прекращение работы.** Продукт удаляется с рынка, клиенты переводятся на другое обеспечение.

Эти пять этапов не разделены жесткими стенами. Технический уровень качества программы непрерывно убывает с течением времени. Авторы отмечают, что данную последовательность этапов могут переживать отдельные версии программных продуктов, сменяемые новыми версиями.

Цель разработки программного обеспечения состоит в том, чтобы замедлить спад и продержать программу на этапах развития и обслуживания как можно дольше. В любом случае, менеджеры обычно стремятся к тому, чтобы избежать внезапного устаревания программы, и стараются заранее предусмотреть такой исход.

23.8. Резюме

Предметом разработки чаще всего бывает не создание новых приложений, а развитие существующих. Вы, как разработчик, должны уметь модифицировать существующие приложения и обеспечивать интеграцию с ними. Вы можете унаследовать от старого приложения требования, идеи, данные и код. Основная технология, используемая в работе с унаследованными приложениями, — инженерный анализ.

Назначение инженерного анализа — сохранение информации из старых систем и перенесение ее в новые системы. Инженерный анализ не предназначен для увековечения прошлых ошибок. Вы должны исключить все ошибки, которые вам удастся обнаружить. Это просто дополнительный источник требований для новых приложений, однако важность его велика. Входных данных для инженерного анализа может быть довольно много, и вы должны быть готовы использовать их все. Выходными данными инженерного анализа являются модели.

Начинайте работу с построения модели классов. Перенесите в нее классы, ассоциации и обобщения. Мы рекомендуем строить модель классов в три этапа, которые мы называем восстановлением реализации, восстановлением проектирования и восстановлением анализа. На каждом этапе количество решений по интерпретации программы возрастает.

Затем приступайте к построению модели взаимодействия, связывая ваше понимание поведения программы с моделью классов. Вы можете начать с процедурного кода и при помощи разделения на слои получить порции логики, относящиеся к различным объектам. Разделение на слои позволяет преобразовать процедурный код в объектно-ориентированный. В конечном итоге вы должны представить модель взаимодействия в виде совокупности диаграмм деятельности и последовательности.

При необходимости постройте модель состояний. Делать это нужно в последнюю очередь. При этом полезно обращаться к диаграммам последовательности.

Инженерный анализ отличается от разработки и требует другого образа мышления. Мы привели несколько советов, помогающих настроиться на нужный лад.

Обертки — альтернативный метод работы с унаследованными приложениями. Вы можете рассматривать такое приложение как черный ящик и создать вокруг него удобные интерфейсы. После этого новые приложения смогут обращаться к функциональности старого посредством обертки.

Таблица 23.2. Ключевые понятия главы

прямое конструирование	инженерный анализ
поддержка	обертка

Библиографические замечания

[Bachman-89] утверждает, что большинство информационных систем строятся на базе старых программ, и только в исключительных случаях создается что-то новое. Это относится не только к информационным системам, но и к программному обеспечению вообще.

Книга [Chikofsky-90] оказала больше влияние на стандартизацию терминологии инженерного анализа. [Kollmann-01] рассказывает о том, каким образом следует восстанавливать модель классов из исходного кода на основании тестовых запусков. Тестовые запуски часто не могут полностью подтвердить гипотезы о модели классов, но они могут помочь понять ее или подсказать правильное решение. В книге [Sneed-96] приводится базовый подход к преобразованию процедурной программы на языке Кобол в объектно-ориентированное представление.

Литература

[Bachman-89] Charles W. Bachman. A personal chronicle: Creating better information systems, with some guiding principles. IEEE Transactions on Knowledge and Data Engineering 1, 1 (March 1989), 17—32.

[Chikofsky-90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, January 1990, 13–17.

[Kollmann-01] Ralf Kollmann and Martin Gogolla. Application of UML associations and their adornments in design recovery, *IEEE Eighth Working Conference on Reverse Engineering*, October 2001, Stuttgart, Germany, 81–90.

[Rajlich-00] Vaclav T. Rajlich and Keith H. Bennett. A staged model for the software life cycle. *IEEE Computer*, July 2000, 6671.

[Rekoff-85] MG Rekoff, Jr. On Reverse Engineering, *IEEE Transactions on Systems, Man, and Cybernetics SMC-15*, 2 (March/April 1985), 244–252.

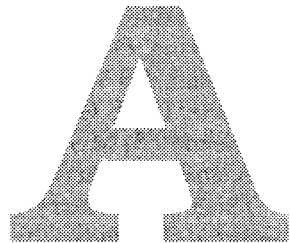
[Sneed-96] Harry M. Sneed. Object-oriented COBOL recycling. *IEEE Third Working Conference on Reverse Engineering*, November 1996, Monterey, CA, 169–178.

[Sneed-01] Harry M. Sneed. Wrapping legacy COBOL programs behind an XML interface. *IEEE Eighth Working Conference on Reverse Engineering*, October 2001, Stuttgart, Germany, 189–197.

[Weiser-92] M. Weiser. Programmers use slices when debugging. *Communications of the ACM* 25, 7 (July 1982), 446–452.

[Weiser-84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering* 10, 4 (July 1984), 352–357.

Система графических обозначений UML

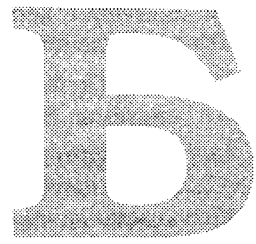


На внутренней стороне обложки книги приведены графические обозначения, используемые в моделях классов, состояний и взаимодействия. Вы можете использовать их в качестве краткого справочника при построении или интерпретации диаграмм. Однако мы должны предупредить вас, что новичку будет сложно разобраться в этих обозначениях. Чтобы понять концепции, для которых используются эти обозначения, вы должны изучить первую часть нашей книги. Чтобы научиться применять обозначения и концепции в течение жизненного цикла программного обеспечения, изучите главы 2 и 3. Для поиска нужного материала можно использовать алфавитный указатель.

За исключением названий концепций и примечаний, все элементы диаграмм, текстовые названия и символы пунктуации входят в систему графических обозначений. Названия на диаграммах (такие как *Class*, *attribute1*, *operation* и *event2*) указывают на элементы, к которым они относятся. Вы можете изменить синтаксис имен и объявления атрибутов и сигнатур, чтобы согласовать их с синтаксисом языка реализации.

Большинство приведенных элементов не относятся к обязательным (особенно на ранних этапах моделирования). Даже на этапе проектирования не следует перегружать модель лишними названиями и обозначениями. Например, если у полюсов ассоциации указаны имена, давать имя самой ассоциации уже не обязательно. Мы не указывали необязательные элементы, потому что хотели продемонстрировать фактическую систему обозначений UML, не усложняя ее дополнительными метаобозначениями.

Вы можете свободно копировать приведенные примеры обозначений UML с внутренней стороны обложки этой книги. Электронную версию можно скачать с сайта <http://www.modelsoftcorp.com>.



Краткий словарь

В этом словаре собраны термины, используемые в объектно-ориентированном моделировании на стадиях анализа, проектирования и реализации.

- **abstract class** (абстрактный класс) — класс, не имеющий непосредственных экземпляров. В UML имя абстрактного класса выделяется курсивом или ключевым словом {abstract}, которое ставится под именем или после него (ср. concrete class).
- **abstract operation** (абстрактная операция) — операция без реализации. Конкретный потомок класса с абстрактной операцией должен предоставить метод, реализующий эту операцию. В UML имя абстрактной операции выделяется курсивом или после ее имени ставится ключевое слово {abstract}.
- **abstraction** (абстракция) — фокусирование на важнейших аспектах приложения, игнорирование незначительных деталей.
- **access modifier** (модификатор доступа) — (в Java) средство управления доступа к методам и данным посредством задания видимости public, private, protected и package.
- **access specifier** (спецификатор доступа) — (в C++) средство управления доступом к методам и данным посредством задания значений видимости public, private и protected, а также с использованием объявления дружественности friend.
- **activation** (активация) — период выполнения объекта. В UML обозначается узким прямоугольником (синоним — focus of control).
- **active object** (активный объект) — объект, обладающий собственным потоком управления (ср. passive object).
- **activity** (деятельность) — спецификация выполняемого поведения.
- **activity diagram** (диаграмма деятельности) — диаграмма, на которой показана последовательность этапов сложного процесса.

- activity token (маркер деятельности) — маркер, размещение которого на символе деятельности показывает прогресс выполнения.
- actor (действующее лицо) — непосредственный внешний пользователь системы. В UML обозначается значком «человечка» (stick man).
- aggregation (агрегация) — разновидность ассоциации, в которой целое (совокупность) состоит из частей. Агрегация часто называется отношением «часть-целое» и может образовывать многоуровневую иерархию вложенности. Агрегация обладает свойствами транзитивности и антисимметричности. В UML обозначается небольшим пустым ромбом, который ставится поверх полюса ассоциации у класса-совокупности (ср. composition).
- analysis (анализ) — этап разработки, на котором изучается задача из реального мира. Целью этапа является получение представления о требованиях к решению без планирования возможной реализации.
- ancestor class (класс-предок) — класс, являющийся непосредственным или косвенным суперклассом некоторого другого класса (ср. descendant class).
- API — см. *Application Programming Interface*.
- application analysis (анализ приложения) — второй подэтап анализа, на котором исследуются компьютерные аспекты приложения, видимые пользователям.
- application programming interface (интерфейс программирования приложений) — совокупность методов, обеспечивающих функциональность приложения.
- architecture (архитектура) — высокоуровневая стратегия решения задачи создания приложения.
- assembly (совокупность) — класс объектов, состоящий из частей объектов.
- association (ассоциация) — описание группы связей с общей структурой и семантикой. В UML обозначается линией, соединяющей классы. Линия может состоять из нескольких сегментов.
- association class (класс ассоциации) — ассоциация, которая одновременно является классом. Подобно связям ассоциации, экземпляры класса ассоциации получают свою индивидуальность от экземпляров связанных классов. Подобно обычным классам, класс ассоциации может иметь атрибуты, операции и участвовать в ассоциациях. В UML обозначается прямоугольником (как класс), прикрепленным к ассоциации пунктирной линией.
- association end (полюс ассоциации) — конец ассоциации. Бинарная ассоциация имеет два полюса, тернарная — три и т. д.
- attribute (атрибут) — именованное свойство класса, описывающее значение, которым обладает каждый объект класса. В UML атрибуты указываются во втором отделе прямоугольника класса.
- automatic transition (автоматический переход) — переход, запускающийся автоматически, когда завершается деятельность, связанная с исходным состоянием.

- **bag** (мультимножество) — неупорядоченная совокупность элементов, допускающая наличие дубликатов. В UML обозначается ключевым словом {bag}, которое может ставиться у полюса ассоциации.
- **base class** (базовый класс) — (в C++) суперкласс.
- **batch transformation** (пакетное преобразование) — стиль архитектуры, обозначающий последовательное преобразование данных между входом и выходом. В начале задаются входные данные, а цель — получить выходные. В процессе вычислений никакого взаимодействия с внешним миром нет (ср. continuous transformation).
- **boundary class** (пограничный класс) — класс объектов, обеспечивающий сопряжение системы с внешним источником.
- **call-by-reference** (вызов по ссылке) — в языке программирования; механизм передачи аргументов методу, состоящий в передаче адресов этих аргументов (ср. call-by-value).
- **call-by-value** (вызов по значению) — в языке программирования; механизм передачи аргументов методу, в котором передаются копии значений данных. Если аргумент изменяется, новое значение не будет передано во внешнюю по отношению к методу часть системы (ср. call-by-reference).
- **candidate key** (потенциальный ключ) — в реляционной базе данных; комбинация столбцов, уникально идентифицирующих каждую строку таблицы. Комбинация должна быть минимально возможной. В нее должны входить только те столбцы, которые действительно нужны для уникальной идентификации. Ни один из этих столбцов не может быть нулевым.
- **cardinality** (кардинальное число, мощность) — количество элементов в совокупности (ср. multiplicity).
- **change event** (событие изменения) — событие, вызываемое выполнением какого-либо логического условия. Подразумевается, что условие непрерывно проверяется, и как только его значение изменяется с «ложно» на «истинно», происходит событие изменения. В UML обозначается ключевым словом when, за которым следует логическое выражение в скобках (ср. guard condition).
- **changeability** (изменяемость) — указывает на то, может ли некоторое свойство (например, полюс ассоциации) измениться после присвоения начального значения. Возможные значения: changeable (может меняться) и readonly (может только инициализироваться).
- **class** (класс) — описание группы объектов, обладающих схожими свойствами (атрибутами), поведением (операциями и диаграммами состояний), отношениями с другими объектами и семантикой. В UML обозначается прямоугольником, имя которого указывается в верхнем отделе.
- **class design** (проектирование классов) — этап разработки, на котором расширяется и оптимизируется аналитическая модель. После этого этапа можно переходить к реализации системы.

- class diagram (диаграмма классов) — графическое представление классов и их отношений, а следовательно, и всех возможных объектов (ср. object diagram).
- class model (модель классов) — описание структуры объектов системы, включая их индивидуальность, отношения с другими объектами, атрибуты и операции.
- classification (классификация) — группировка объектов, обладающих схожими структурами данных и поведением.
- client (клиент) — подсистема, запрашивающая услуги от другой подсистемы (ср. server).
- coherence (согласованность, цельность) — свойство элемента, такого как класс, операция или пакет, заключающееся в упорядоченности в соответствии с некоторым непротиворечивым планом, а также в служении всех частей единой цели.
- completion transition (переход по завершении) — переход, который запускается автоматически по завершении деятельности, связанной с исходным состоянием.
- composite state (составное состояние) — состояние, содержащее другие состояния и предоставляющее для них общее поведение (ср. nested state).
- composition (композиция) — разновидность агрегации с дополнительными ограничениями. Часть может принадлежать только одному целому. Более того, после отнесения части к некоторому целому срок жизни этой части становится равным сроку жизни целого. В UML обозначается небольшим сплошным ромбом, который ставится поверх полюса ассоциации около класса-совокупности (ср. aggregation).
- concrete class (конкретный класс) — класс, который может иметь непосредственные экземпляры (ср. abstract class).
- concurrent (параллельный) — характеристика двух и более видов деятельности или событий, которые могут перекрываться во времени.
- condition (условие) — см. *guard condition*.
- constraint (ограничение) — логическое условие, в которое входят элементы модели (объекты, классы, атрибуты, ассоциации и наборы обобщений). Простые ограничения обозначаются в UML текстовой строкой в фигурных скобках или в прямоугольнике комментария. Сложные ограничения можно выражать на объектном языке ограничений (Object Constraint Language).
- constructor (конструктор) — (в C++ и Java), операция инициализации созданного экземпляра класса (ср. destructor).
- container class (класс-контейнер) — класс объектов-контейнеров. В качестве примеров можно привести множества, массивы, словари и ассоциации.
- continuous transformation (непрерывное преобразование) — система, в которой выход активно зависит от изменяющегося входа и должен периодически обновляться (ср. batch transformation).

- control (управление) — аспект системы, описывающий последовательность операций, осуществляемых в ответ на воздействия.
- controller (управляющий объект) — активный объект, осуществляющий управление внутри приложения.
- database (база данных) — постоянное хранилище данных вместе с описаниями, которое может состоять из нескольких файлов. Наличие описания данных отличает базу данных от обычных файлов.
- database management system (система управления базой данных) — программное обеспечение, управляющее доступом к базе данных.
- data dictionary (словарь данных) — определение всех элементов модели (классов, ассоциаций, атрибутов, операций и значений перечислений), а также обоснование ключевых решений, принятых в процессе моделирования.
- DBMS (СУБД) — см. *database management system*.
- default value (значение по умолчанию) — значение, используемое для инициализации атрибута или в качестве аргумента метода.
- delegation (делегирование) — механизм реализации, в котором объект, в ответ на операцию на самом себе, передает эту операцию другому объекту.
- denormalization (денормализация) — нарушение нормальной формы. Разработчикам следует нарушать нормальные формы только в том случае, если для этого имеются достаточные основания, например, для повышения производительности в узком месте системы (см. *normal form*).
- derived class (производный класс) — (в C++) подкласс.
- derived element (производный элемент) — (в UML) элемент, определяемый в терминах других элементов. Производными могут быть классы, атрибуты и ассоциации. Не путайте термин *derived* в UML с тем же термином в C++. Производный класс в C++ означает подкласс и не имеет никакого отношения к производному элементу UML. Для обозначения производных элементов в UML используется косая черта, которая ставится перед именем элемента.
- descendant class (класс-потомок) — класс, являющийся непосредственным (прямым) или косвенным потомком другого класса (ср. *ancestor class*).
- destructor (деструктор) — (в C++) операция удаления существующего экземпляра класса, который больше не нужен (ср. *constructor*).
- development (разработка) — создание программного обеспечения.
- development life cycle (жизненный цикл разработки) — подход к управлению процессом создания программного обеспечения.
- development stage (этап разработки) — этап процесса создания программного обеспечения. В этой книге описывается следующая последовательность этапов: концептуализация системы, анализ предметной области, анализ приложения, проектирование системы, проектирование классов, моделирование реализации и реализация. Хотя этапы проектирования упорядочены, их не обязательно выполнять последовательно для разных частей приложения. Мы не ориентировались на водопадную разработку.

- dictionary (словарь) — неупорядоченная совокупность пар объектов с допустимыми дубликатами. Каждая пара представляет собой связь между ключом и элементом. Ключ может использоваться для поиска элемента.
- direction (направление) — аргумент операции или метода может быть входным (in), выходным (out) или входным с возможностью изменения (inout).
- do activity (текущая деятельность) — деятельность, продолжающаяся длительное время. В UML обозначается префиксом do /, за которым следует название деятельности.
- domain analysis (анализ предметной области) — первый подэтап анализа, на котором рассматриваются предметы реального мира, несущие семантику приложения.
- dynamic binding (динамическая привязка) — форма разрешения методов, связывающая метод с операцией во время выполнения программы в зависимости от класса одного или нескольких целевых объектов.
- dynamic simulation (динамическое моделирование) — архитектурный стиль системы, занимающейся моделированием объектов реального мира или слежением за ними.
- effect (действие) — ссылка на поведение, выполняемое в ответ на событие. В UML обозначается косой чертой, за которой следует название деятельности.
- encapsulation (инкапсуляция) — отделение внешней спецификации от внутренней реализации (синоним information hiding).
- enterprise model (модель предприятия) — модель, описывающая всю организацию или какой-либо важный аспект организации.
- Entity-Relationship (ER) model (модель Сущность-Связь) — графический подход к моделированию, предложенный Питером Ченом (Peter Chen), описывающий изображение сущностей и отношений между ними. Модель классов UML основана на модели ER.
- entry activity (деятельность при входе) — деятельность, выполняемая при входе в состояние. В UML обозначается префиксом entry /, который ставится перед названием деятельности (ср. exit activity).
- enumeration (перечисление) — тип данных, имеющий конечное число значений. В UML обозначается ключевым словом «enumeration», которое ставится над именем перечисления в верхнем отделе прямоугольника. Во втором отделе приводятся значения перечисления.
- ER — см. *Entity-Relationship model*.
- event (событие) — происшествие в определенный момент времени (ср. state).
- event-driven control (событийное управление) — подход, согласно которому управление осуществляется диспетчером или монитором, предоставляемым языком, подсистемой или операционной системой. Разработчики связывают методы приложения с событиями, а диспетчер вызывает методы при осуществлении соответствующих событий (ср. procedure-driven control).

- **exit activity** (деятельность при выходе) — деятельность, выполняемая непосредственно перед выходом из состояния. В UML обозначается префиксом **exit** / (ср. **entry activity**).
- **extend** (расширение) — отношение вариантов использования, добавляющее поведение к варианту использования. В данном случае расширение добавляется к базовому варианту использования. В случае отношения включения (**include**) базовый вариант использования явным образом содержит включаемый вариант использования. В UML обозначается пунктирной стрелкой от расширяющего варианта использования к базовому. Над стрелкой ставится ключевое слово «**extend**» (ср. **include**).
- **extensibility** (расширяемость) — качество программного обеспечения, позволяющее добавлять к нему новую функциональность с незначительными изменениями существующего кода.
- **extent of a class** (экстент класса) — множество всех объектов класса.
- **feature** (составляющая) — атрибут или операция.
- **final** (для класса Java) — директива, запрещающая создание подклассов.
- **final** (для метода Java) — директива, запрещающая перегрузку метода.
- **fire** (запустить фактически) — начало фактического выполнения перехода.
- **focus of control** (фокус управления) — период выполнения объекта. В UML обозначается узким прямоугольником (синоним **activation**).
- **foreign key** (внешний ключ) — в реляционной базе данных, ссылка на потенциальный ключ (обычно — на основной ключ). Это «клей», которым таблицы соединяются между собой.
- **forward engineering** (прямое конструирование) — построение приложения от общих требований к конечной реализации (ср. **reverse engineering**).
- **fourth-generation language** (язык четвертого поколения) — каркас для приложений баз данных, обеспечивающий вывод на экран, простые вычисления и составление отчетов.
- **framework** (каркас) — скелетная структура программы, требующая детализации для построения полноценного приложения.
- **friend** — (в C++) декларация, разрешающая выборочный доступ к составляющим класса. Класс, содержащий декларацию **friend**, получает доступ к именованной функции, методу или классу.
- **garbage collection** (сборка мусора) — в языках программирования, механизм автоматического освобождения памяти от структур данных, обращение к которым более невозможно.
- **generalization** (обобщение) — частичное упорядочение элементов (классов, сигналов, вариантов использования) по сходствам и отличиям. В UML обозначается треугольником, вершина которого помещается около обобщающего элемента (ср. **specialization**).

- **generalization set name** (имя набора обобщений) — перечислимый атрибут, указывающий, какой аспект объекта абстрагируется конкретным обобщением.
- **guard condition** (сторожевое условие) — логическое выражение, истинность которого необходима для осуществления перехода. Сторожевое условие проверяется только один раз, в момент осуществления переключающего события, и переход запускается только в том случае, если сторожевое условие оказывается истинно. В UML сторожевое условие указывается в квадратных скобках после события.
- **identifier** (идентификатор) — один или несколько атрибутов реализации, однозначно отличающие объект от всех остальных.
- **identity** (индивидуальность) — неотъемлемое свойство объекта, позволяющее отличать его от всех остальных.
- **implementation** (реализация) — этап разработки, на котором проект превращается в код на языках программирования и в структуры баз данных.
- **implementation inheritance** (наследование в реализации) — некорректная реализация, создатели которой пытаются повторно использовать существующий код, но добиваются этого ценой построения нелогичной структуры приложения, которая может помешать обслуживанию системы в будущем.
- **implementation method** (метод реализации) — метод, реализующий конкретные вычисления на явно заданных аргументах, но не принимающий решений, зависящих от контекста (ср. *policy method*).
- **implementation modeling** (моделирование реализации) — этап разработки, на котором добавляются тонкие детали, зависящие от языка реализации. Моделирование реализации непосредственно предшествует самой реализации.
- **include** (включение) — отношение вариантов использования, в котором один вариант использования добавляется внутрь последовательности поведения другого варианта использования. В UML обозначается пунктирной стрелкой от исходного (включающего) варианта использования к целевому (включаемому). Над стрелкой ставится ключевое слово «include» (ср. *extend*).
- **index** (индекс) — структура данных, сопоставляющая одно или несколько значений атрибута объектам или строкам таблицы базы данных, в которых хранятся значения. Индексы используются для оптимизации (для быстрого поиска объектов и строк), а также для обеспечения уникальности.
- **information hiding** (сокрытие информации) — см. *encapsulation*.
- **inheritance** (наследование) — механизм реализации отношения обобщения.
- **integration testing** (тестирование интеграции) — тестирование кода, написанного несколькими разработчиками, для проверки согласования классов и методов (ср. *unit testing* и *system testing*).

- **interaction model** (модель взаимодействия) — модель, описывающая совместную деятельность объектов для достижения результата. Это целостный взгляд на поведение множества объектов, тогда как модель состояний дает редуцированное представление поведения каждого объекта в отдельности.
- **interactive interface** (интерактивный интерфейс) — стиль архитектуры системы, в которой преобладают взаимодействия между системой и агентами, такими как люди, устройства или другие программы.
- **interface** (в Java) — спецификация класса, не позволяющая порождать экземпляры. Такой класс может содержать только константы и объявления методов.
- **iterative development** (итерационная разработка) — разработка системы, представляющая собой процесс, разбитый на множество этапов (итераций), на каждом из которых достигается более совершенное приближение к желаемому результату по сравнению с предыдущим этапом (ср. *rapid prototyping* и *waterfall development*).
- **iterator** (итератор) — в языке программирования, конструкция, управляющая итерацией по всему диапазону значений или по всей совокупности объектов.
- **layer** (уровень) — подсистема, предоставляющая различные сервисы, обладающие одинаковым уровнем абстракции. Уровень может быть основан на подсистемах с более низким уровнем абстракции (ср. *partition*).
- **leaf class** (листовой класс) — класс, не имеющий подклассов. Должен быть конкретным. В Java — синоним обозначения *final*.
- **library** (библиотека) — совокупность классов, которые могут повторно использоваться в разных приложениях.
- **life cycle** (жизненный цикл) — см. *development life cycle*.
- **lifeline** (линия жизни) — период времени, в течение которого существует объект.
- **link** (связь) — физическое или концептуальное соединение между объектами. Связь является экземпляром ассоциации. В UML обозначается линией между объектами. Линия может состоять из нескольких прямолинейных сегментов.
- **lock** (блокировка) — логический объект, связанный с определенным подмножеством ресурса. Этот объект дает владельцу блокировки право непосредственно работать с ресурсом.
- **member** (составляющая класса, в C++) — данные или методы класса.
- **metaclass** (метакласс) — класс, описывающий другие классы.
- **metadata** (метаданные) — данные, описывающие другие данные.
- **method** (метод) — реализация операции класса. В UML методы перечисляются в третьем отделе прямоугольника класса (ср. *operation*).
- **method caching** (кэширование методов) — в языках программирования, оптимизация поиска методов, состоящая в том, что адрес метода ищется только

один раз, когда операция впервые применяется к классу, а затем хранится в таблице, связанной с этим классом.

- **method resolution** (разрешение методов) — в языках программирования, процесс сопоставления операции над объектом методу, соответствующему классу этого объекта.
- **methodology** (методология) — в производстве программного обеспечения, процесс организованного производства программного обеспечения с использованием набора готовых методов и соглашений об обозначениях.
- **model** (модель) — абстракция некоторого аспекта задачи. Модели выражаются при помощи различных диаграмм.
- **modularity** (модульность) — деление системы на группы тесно связанных объектов.
- **multiple inheritance** (множественное наследование) — вид наследования, позволяющий классу иметь несколько суперклассов и наследовать черты от всех предков (ср. *single inheritance*).
- **multiplicity** (кратность) — по отношению к полюсу ассоциации, количество экземпляров одного класса, которые могут быть связаны с одним экземпляром другого класса. Кратность — это ограничение на размер совокупности. В UML обозначается интервалом чисел или специальным символом *, обозначающим слово «много» (ноль и более) (ср. *cardinality*). По отношению к атрибуту — возможное количество значений каждого экземпляра атрибута. Чаще всего принимает значения [1] (обязательное единственное значение), [0..1] (необязательное единственное значение) и [*] (много).
- **namespace** (пространство имен) — (в C++) служит для задания семантической области символов и для разрешения конфликтов имен.
- **n-ary association** (n-арная ассоциация) — ассоциация, обладающая тремя и более полюсами. В UML обозначается ромбом, соединенным с участвующими классами при помощи линий. Если у ассоциации есть имя, оно указывается курсивом около ромба.
- **navigability** (возможность навигации) — направление перехода по бинарной ассоциации в реализации. Возможные значения: никакого, в одном направлении, в обоих направлениях. В UML обозначается стрелкой, которая ставится у полюса ассоциации, прикрепленного к целевому классу.
- **navigation** (прослеживание связей, навигация) — переход по ассоциациям и обобщениям модели классов от исходных объектов к целевым.
- **nested state** (вложенное состояние) — состояние, берущее поведение у композитного состояния и добавляющее свое собственное поведение (ср. *composite state*).
- **new** — (в C++ и Java) оператор создания объектов.
- **normal form** (нормальная форма) — в реляционной базе данных, критерий проектирования базы данных, повышающий согласованность данных.

- n-tier architecture (многоуровневая архитектура) — расширение трехуровневой архитектуры, допускающее произвольное количество уровней в приложении (ср. three-tier architecture).
- null (нуль) — специальное значение атрибута, обозначающее: неизвестно или неприменимо.
- object (объект) — концепция, абстракция или нечто такое, что может быть уникально идентифицировано и имеет смысл в рамках приложения. Объект является экземпляром класса.
- Object Constraint Language (объектный язык ограничений) — язык формулирования ограничений, входящий в состав UML. OCL может использоваться для прослеживания моделей классов.
- object diagram (диаграмма объектов) — графическое представление отдельных объектов и их отношений (ср. class diagram).
- object identity (индивидуальность объекта) — в реляционной базе данных, использование искусственно введенного номера для идентификации каждой записи в таблице (ср. value-based identity).
- Object Management Group (группа управления объектами) — форум по стандартизации, которому принадлежат права на UML.
- object-orientation (объектная ориентированность) — стратегия организации систем в виде совокупностей взаимодействующих объектов, объединяющих в себе данные и поведение.
- OCL — см. *Object Constraint Language*.
- OMG — см. *Object Management Group*.
- OO — см. *object-oriented*.
- OO database (объектно-ориентированная база данных) — база данных, воспринимающая объекты как смесь данных и поведения (ср. relational database).
- OO-DBMS (объектно-ориентированная СУБД) — СУБД, поддерживающая постоянные объекты (помимо временных, поддерживаемых объектно-ориентированными языками программирования (ср. relational DBMS)).
- OO development (объектно-ориентированная разработка) — методика разработки программного обеспечения, в которой в качестве основы используются объекты.
- OO programming language (объектно-ориентированный язык программирования) — язык, поддерживающий объекты (сочетающие индивидуальность, данные и операции), разрешение методов и наследование.
- operation (операция) — функция или процедура, которая может применяться к объектам класса или объектами класса (ср. method).
- ordered (упорядоченный) — отсортированная совокупность элементов, в которой недопустимо наличие дубликатов. В UML обозначается ключевым словом {ordered}, которое ставится, например, у полюса ассоциации (ср. sequence).

- **origin class** (исходный класс) — самый верхний класс иерархии, в котором определена некоторая черта.
- **overloading** (перегрузка) — в языке программирования, связывание одного имени с несколькими методами, сигнатуры которых отличаются количеством или типом аргументов. Вызов перегруженной операции разрешается во время компиляции в зависимости от типов аргументов.
- **override** (подмена или перекрытие) — определение метода для операции взамен унаследованного метода для той же операции.
- **package** (пакет) — конструкция моделей классов, представляющая собой группу элементов (классов, ассоциаций, обобщений и вложенных пакетов), объединенных общей темой. В UML обозначается прямоугольником с закладкой. Название пакета указывается внутри прямоугольника.
- **partition** (раздел) — подсистема, предоставляющая некоторый сервис параллельно с другими подсистемами. Раздел может сам состоять из подсистем более низкого уровня (ср. *layer*).
- **passive object** (пассивный объект) — объект, не имеющий собственного потока управления (ср. *active object*).
- **pattern** (образец, паттерн) — параметризованная выборка из модели, характеризующаяся важностью и повторяемостью. Образец имеет математическую природу и эффективен при повторном использовании в различных приложениях.
- **peer** (равные, одноранговые) — несколько подсистем, взаимно зависящих от сервисов, предоставляемых друг другу (ср. *client* и *server*).
- **persistent object** (постоянный объект) — объект, который хранится в базе данных и может существовать в течение нескольких прогонов программы.
- **policy method** (стратегический метод) — метод, принимающий решения, зависящие от контекста, но вызывающий другие (реализующие его) методы для выполнения конкретных расчетов (ср. *implementation method*).
- **polymorphism** (полиморфизм) — принимает разные формы, суть его состоит в том, что поведение операции может быть различным для разных классов.
- **primary key** (первичный ключ) — в реляционной базе данных; потенциальный ключ, который предпочтительнее использовать для обращения к записям таблицы. Таблица может иметь не более одного первичного ключа. Обычно в каждой таблице такой ключ имеется.
- **private** (закрытая) — обозначение видимости, означающее доступность лишь методам класса-владельца.
- **procedure-driven control** (процедурное управление) — подход, согласно которому управление осуществляется кодом программы. Процедуры запрашивают ввод внешних данных и блокируются в ожидании его. При поступлении входных данных выполнение программы возобновляется с той процедуры, которая запросила эти данные. Значение счетчика команд, состояние стека процедур и значения локальных переменных определяют состояние всей системы (ср. *event-driven control*).

- **programming-in-the-large** (программирование крупных систем) — создание больших и сложных программ с участием команд разработчиков.
- **protected** (защищенная) — обозначение видимости в C++, означающее доступность методам класса-владельца, его потомков и классов из того же пакета.
- **public** (открытая) — обозначение видимости, означающее доступность методам любого класса.
- **qualified association** (квалифицированная ассоциация) — ассоциация, у которой один или несколько атрибутов, относящихся к полюсу со значением кратности «много», позволяют отличать объекты друг от друга. Эти атрибуты называются квалификаторами. В UML обозначается небольшим прямоугольником, который ставится в конце линии ассоциации около исходного класса.
- **qualifier** (квалификатор) — атрибут, позволяющий отличать друг от друга объекты, находящиеся у полюса ассоциации с кратностью «много». В UML квалификатор помещается внутрь небольшого прямоугольника, который ставится в конце линии ассоциации около исходного класса.
- **race condition** (ситуация гонок) — ситуация, в которой порядок получения параллельных сигналов может повлиять на конечное состояние объекта.
- **rapid prototyping** (быстрое прототипирование) — быстрая разработка части системы для экспериментирования и оценки. Прототипирование позволяет подтвердить концепцию системы, а модель-прототип часто не предназначается для реального использования (ср. *iterative development* и *waterfall development*).
- **real-time system** (система реального времени) — стиль архитектуры, описывающий интерактивную систему с жесткими временными ограничениями на действия, где недопустимы задержки.
- **refactoring** (рефакторинг) — изменение внутренней структуры программного обеспечения с целью его усовершенствования без изменения внешней функциональности.
- **reference** (ссылка) — значение атрибута одного объекта, ссылающееся на другой объект.
- **reflection** (рефлексия) — свойство системы, состоящее в том, что она может динамически исследовать собственную структуру и делать заключения о том, в каком состоянии она находится.
- **region** (область) — часть диаграммы состояний.
- **reification** (воплощение) — превращение в объект чего-либо не являющегося таковым.
- **relational database** (реляционная база данных) — база данных, в которой сами данные представлены в виде таблиц (ср. *OO database*).
- **relational DBMS** (реляционная СУБД) — СУБД, управляющая таблицами данных и связанными структурами, повышая функциональность и эффективность использования таблиц (ср. *OO-DBMS*).

- responsibility (ответственность) — нечто известное объекту или обязательное для выполнения им. Ответственность — это не четкая концепция, а способ организации.
- reverse engineering (инженерный анализ) — процесс анализа артефактов реализации и восстановления лежащей в основе логики проекта (ср. forward engineering).
- robust (устойчивость) — свойство программного обеспечения не допускать катастрофических отказов в случае нарушения некоторых требований, закладывавшихся в проект.
- scenario (сценарий) — последовательность событий, происходящих в процессе одного конкретного выполнения системы.
- schema (схема) — структура данных в базе.
- scope (область действия) — указание на то, относится ли составляющая к объекту (динамическая) или к классу в целом (статическая). Обозначается подчеркиванием.
- sequence (последовательность) — упорядоченная совокупность элементов, в которой допускается наличие дубликатов. В UML обозначается ключевым словом {sequence}, которое ставится около полюса ассоциации (ср. ordered).
- sequence diagram (диаграмма последовательности) — диаграмма, на которой показываются участники взаимодействия и последовательности сообщений, которыми они обмениваются.
- server (сервер) — подсистема, предоставляющая сервисы другим подсистемам (ср. client).
- service (сервис) — группа связанных функций или операций, имеющих общее назначение.
- shopping-list operation (операция по списку) — операция, имеющая смысл сама по себе. Бертран Мейер ввел этот термин для обозначения операций, которые вводятся исходя из внутренних потребностей классов, а не из нужд приложения. Иногда операции подсказываются поведением классов в реальном мире.
- signal (сигнал) — явная односторонняя передача информации от одного объекта к другому. В UML обозначается ключевым словом «signal», которое ставится над названием класса сигнала в верхнем отделе прямоугольника. Во втором отделе перечисляются атрибуты сигнала.
- signal event (событие сигнала) — событие отправки или получения сигнала.
- signature (сигнатура) — количество и тип аргументов операции и тип ее результата.
- single inheritance (единственное наследование) — вид наследования, в котором у каждого класса может быть только один суперкласс (ср. multiple inheritance).
- software engineering (разработка программного обеспечения) — систематический, дисциплинированный и поддающийся количественному опре-

делению подход к разработке, эксплуатации и поддержке программного обеспечения.

- **specialization** (конкретизация) — уточнение класса в различных разновидностях. Специализация имеет тот же смысл, что и обобщение, но в противоположном направлении (сверху вниз) (ср. *generalization*).
- **SQL** — стандартный язык взаимодействия с базой данных.
- **state** (состояние) — абстракция значений и связей объекта. В UML обозначается прямоугольником со скругленными краями, в котором указывается необязательное название состояния (ср. *event*).
- **state diagram** (диаграмма состояний) — граф, вершинами которого являются состояния, а направленными дугами — переходы между состояниями.
- **state model** (модель состояний) — описание связанных со временем и последовательностью операций аспектов системы. Модель состояний состоит из множества диаграмм состояний, каждая из которых описывает один класс с важным поведением, зависящим от времени.
- **static** — (в C++ и Java) данные и методы, относящиеся не к экземпляру класса, а к самому классу.
- **stored procedure** (храняемая процедура) — в реляционной базе данных, метод, который хранится в самой базе.
- **strong typing** (жесткая типизация) — в языке программирования, требование объявления типов всех переменных (ср. *weak typing*).
- **subclass** (подкласс) — класс, добавляющий конкретные атрибуты, операции, диаграммы состояний и ассоциации к обобщающему объекту (ср. *superclass*).
- **submachine** (вложенный конечный автомат) — диаграмма состояний, которая может вызываться как часть другой диаграммы состояний. В UML вызов вложенного конечного автомата обозначается указанием его названия после названия локального состояния через двоеточие.
- **substate** (подсостояние) — состояние, выражающее аспект параллельного поведения родительского состояния.
- **subsystem** (подсистема) — важная часть системы, объединенная какой-либо тематикой. Система может разбиваться на уровни или разделы, состоящие из подсистем.
- **superclass** (суперкласс) — класс, в котором определяются общие атрибуты, операции, диаграммы состояний и ассоциации (ср. *subclass*).
- **swimlane** (плавательная дорожка) — столбец на диаграмме деятельности, показывающий человека или организацию, выполняющую деятельность; раздел.
- **system** (система) — рассматриваемое приложение.
- **system architecture** (архитектура системы) — см. *architecture*.
- **system boundary** (граница системы) — граница области системы, определяющая, что входит в состав системы, а что — нет.

system conception (концептуализация системы) — этап разработки, на котором зарождается приложение.

system design (проектирование системы) — этап разработки, на котором проектировщик формулирует архитектуру и устанавливает общие политики проектирования.

system testing (тестирование системы) — проверка приложения в целом (ср. unit testing и integration testing).

table (таблица) — в реляционной базе данных, упорядоченная структура данных, обладающая определенным числом столбцов и произвольным числом строк.

ternary association (тернарная ассоциация) — ассоциация с тремя полюсами. В UML обозначается ромбом, соединенным линиями с классами, участвующими в ассоциации. Если ассоциация имеет имя, оно пишется курсивом около ромба.

this — (в C++ и Java) название целевого объекта метода.

thread of control (поток управления) — один из путей выполнения программы, модели состояний или другого предоставления графа потоков управления.

three-tier architecture (трехуровневая архитектура) — подход, в котором выделяются управление данными, функциональность приложения и пользовательский интерфейс. Уровень управления данными включает схему базы данных и сами данные. Уровень приложения включает методы, формирующие логику приложения. Уровень пользовательского интерфейса содержит формы и отчеты, представляемые пользователю (ср. n-tier architecture).

time event (событие времени) — событие, вызванное наступлением момента абсолютного времени или истечением интервала относительного времени. В UML 2.0 абсолютное время обозначается ключевым словом when, за которым в скобках следует выражение, вычисление которого позволяет определить время. Временной интервал обозначается ключевым словом after, за которым следует выражение в скобках, вычисление которого позволяет определить длительность интервала.

transaction manager (администратор транзакций) — архитектурный стиль; система управления базой данных, основная функция которой состоит в хранении и организации доступа к информации.

transient object (временный объект) — объект, существующий только в памяти и исчезающий при завершении выполнения приложения. То есть это обычный программный объект (ср. persistent object).

transition (переход) — мгновенная смена одного состояния другим. В UML обозначается линией (которая может состоять из нескольких прямолинейных сегментов), соединяющей исходное состояние с целевым. Стрелка указывает на целевое состояние.

transitive closure (транзитивное замыкание) — в теории графов, множество вершин, доступных через некоторую последовательность ребер.

- UML — см. *Unified Modeling Language*.
- Unified Modeling Language (унифицированный язык моделирования) — всесторонний комплект объектно-ориентированных моделей, предназначенный для полного описания программного обеспечения и для других целей. UML был разработан под покровительством OMG.
- UML1 — неформальное обозначение первого выпуска UML, который был одобрен в 1997 году.
- UML2 — неформальное обозначение второго выпуска UML, который был одобрен в 2004 году. Эта книга основана на UML2.
- unit testing (модульное тестирование) — тестирование кода классов и методов самими разработчиками (ср. integration testing и system testing).
- use case (вариант использования) — связанная часть функциональности, предоставляемой системой действующим лицам через взаимодействие с ними. В UML обозначается эллипсом, внутри которого указывается название варианта использования.
- use case diagram (диаграмма вариантов использования) — графическое обозначение вариантов использования и действующих лиц.
- user interface (пользовательский интерфейс) — объект или группа объектов, предоставляющих пользователю системы доступ к объектам, командам и параметрам приложения.
- value (значение) — элемент данных. Значение представляет собой экземпляр атрибута.
- value-based identity (индивидуальность, основанная на значениях) — в реляционной базе данных, использование комбинации реальных атрибутов для идентификации всех записей таблицы (ср. object identity).
- view (представление) — в реляционной базе данных, таблица, динамически вычисляемая СУБД.
- virtual — (в C++) операция, которая может быть перекрыта потомком.
- visibility (видимость) — способность метода ссылаться на черту другого класса. В UML видимость обозначается следующими префиксами: + (открытая), # (защищенная), - (закрытая) и ~ (пакетная).
- waterfall development (водопадная модель разработки) — выполнение этапов разработки программного обеспечения в жесткой последовательности без возвратов (ср. iterative development и rapid prototyping).
- weak typing (слабая типизация) — в языке программирования, отсутствие требования задания типов всех переменных в программе (ср. strong typing).
- wrapper (обертка) — совокупность интерфейсов, открывающих доступ в систему.

Ответы к избранным упражнениям

Мы выбирали, к каким упражнениям давать ответы, руководствуясь следующими принципами. В первую очередь были включены короткие ответы к упражнениям из ключевых глав, упражнениям, дополняющим материал главы, ключевым упражнениям из последовательности заданий, ответы, проясняющие тонкие или сложные моменты, а также прототипы реальных задач. В большинстве задач правильный ответ может быть не единственным, поэтому руководствуйтесь нашими ответами только как примерами.

- 1.5. 2) В уголовных расследованиях могут использоваться комбинации фотографий, отпечатков пальцев, исследования на группу крови, анализы ДНК и зубные карты. Все это позволяет идентифицировать живых или мертвых людей, принимающих участие или являющихся предметом расследования.
- 4) Телефонные номера позволяют идентифицировать практически любой телефон мира. Телефонный номер состоит из кода страны, кода области или города и местного номера. Он также может включать необязательный добавочный номер. На предприятиях могут быть собственные телефонные сети с иными соглашениями. В зависимости от расположения вашего собственного телефона некоторые части номера можно опускать. С другой стороны, для вызова номера из другой области может потребоваться набор префикса выхода на междугородную линию.

В США большинство местных номеров состоят из 7 цифр. Для междугородных звонков приходится набирать префикс 0 или 1, код города (3 цифры) и местный номер (7 цифр). Чтобы позвонить в Париж, нужно

набрать код доступа к международной линии (011), код страны (33), код города (1) и местный номер (8 цифр). Код доступа к линии не входит в состав идентификатора телефона.

- 7) Один из способов предоставить сотрудникам ограниченный доступ состоит в использовании специальной электронной карты. Конечно, если сотрудник потеряет такую карту и не сообщит об этом, ей может воспользоваться кто-то другой. В качестве альтернативных способов можно указать проверку по фотографии, проверку отпечатков пальцев и распознавание голоса.

1.8. 1) Электронные микроскопы, очки, телескопы, прицелы бомбометания и бинокли улучшают зрение. Все они, за исключением сканирующего электронного микроскопа, работают с отраженным или преломленным светом. Очки и бинокли рассчитаны на работу с двумя глазами, остальные устройства предназначены для одного глаза. Телескопы, прицелы и бинокли служат для увеличения удаленных предметов. Микроскоп увеличивает очень маленькие объекты. Очки могут как увеличивать, так и уменьшать, в зависимости от дефекта зрения, на устранение которого они рассчитаны. Этот список можно было бы дополнить оптическими микроскопами, камерами и увеличительными стеклами.

- 2) Трубы, клапаны, краны, фильтры и датчики давления — водопроводные устройства, рассчитанные на определенную температуру и давление. Кроме того, нужно учитывать их совместимость с различными жидкостями. Клапаны и краны позволяют управлять потоком жидкости. Все предметы, за исключением датчика давления, обладают двумя соединениями. Для них может быть построена кривая зависимости потока от давления. Все эти элементы являются пассивными. В качестве дополнительных примеров можно привести насосы, резервуары и соединительные трубки.

2.3. 1) Для трансатлантического кабеля главным критерием является сопротивляемость соленой воде. Кабель должен долго лежать на дне океана без всякого обслуживания. Необходимо учесть взаимодействие океанской флоры и фауны с кабелем, а также влияние давления и солености воды на срок его службы. Соотношение прочности к весу важно при прокладке кабеля. Стоимость — важный экономический фактор. Электрические параметры важны с точки зрения потерь мощности и искажения сигналов.

- 3) Для проводов, используемых в электрических цепях самолетов, важен вес, потому что от него зависит общий вес самолета. Прочность изоляции должна быть достаточной, чтобы противостоять истиранию, вызванному вибрацией. Огнеупорность важна для того, чтобы исключить возгорания в полете.

3.2. На рис. ОЗ.2 показана диаграмма классов для многоугольников и точек. Многоугольник может состоять не менее чем из трех точек.

Кратность ассоциации зависит от того, каким образом точки идентифицируются. Если точка идентифицируется по ее координатам, точки можно считать общими, и ассоциация имеет вид «многие-ко-многим». Если же каждая точка может принадлежать ровно одному многоугольнику, несколько точек могут иметь одинаковые координаты. Это различие демонстрирует следующий ответ.

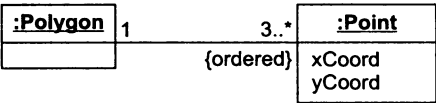


Рис. ОЗ.2. Диаграмма классов для многоугольника и точек

- 3.3 1) На рис. ОЗ.3 показаны объекты и связи для двух треугольников с общей стороной в модели, где каждая точка принадлежит ровно одному многоугольнику.
- 2) На рис. ОЗ.4 показаны объекты и связи для двух треугольников с общей стороной в модели, где точки могут быть общими.

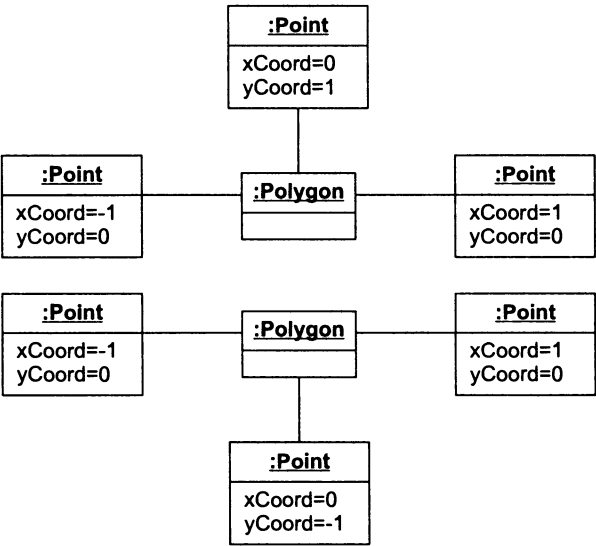


Рис. ОЗ.3. Диаграмма объектов (каждая точка принадлежит одному многоугольнику)

3.20. Графы встречаются во множестве приложений. Возможно несколько вариантов модели, в зависимости от вашей точки зрения. На рис. ОЗ.23 показано точное представление неориентированных графов, соответствующее описанию в упражнении. В вашем ответе может отсутствовать класс UndirectedGraph, но при этом ответ будет уже не таким точным.

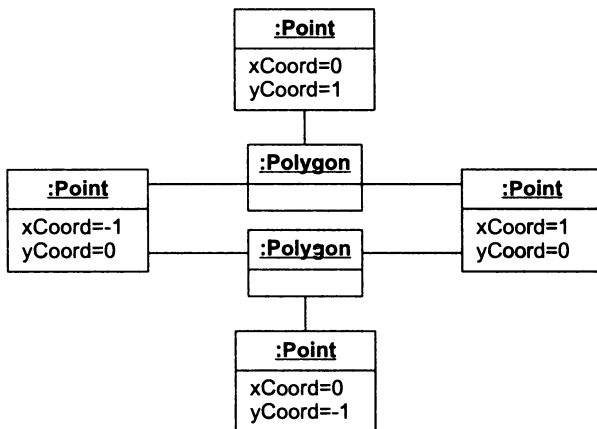


Рис. 03.4. Диаграмма объектов (каждая точка принадлежит нескольким многоугольникам)

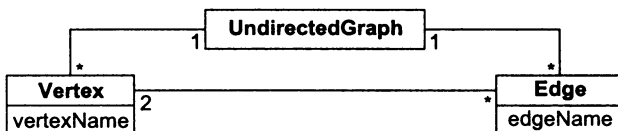


Рис. 03.23. Диаграмма классов для неориентированных графов

Мы обнаружили, что для некоторых запросов, связанных с графами, бывает полезно сделать ассоциацию между вершинами и ребрами классов, как показано на рис. 03.24.

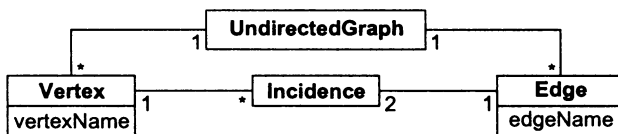


Рис. 03.24. Диаграмма классов неориентированных графов (инцидентность описывается классом)

3.23. На рис. 03.27 показана диаграмма классов, описывающая ориентированные графы. Различие между двумя вершинами ребра обеспечивается квалифицированной ассоциацией. Квалификатор может иметь значения *from* и *to*.

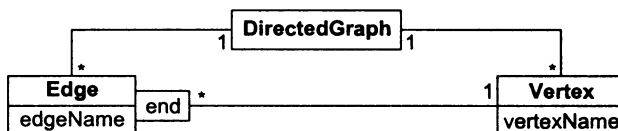


Рис. 03.27. Диаграмма классов ориентированных графов с квалифицированной ассоциацией

На рис. ОЗ.28 показано другое представление ориентированных графов. Различие между двумя вершинами обеспечивается введением двух разных ассоциаций.

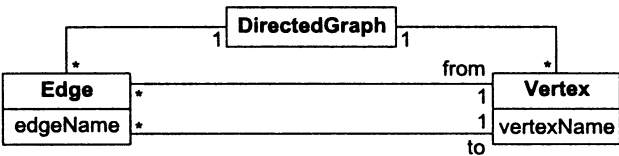


Рис. ОЗ.28. Диаграмма классов ориентированных графов с двумя ассоциациями

Преимущество квалифицированной ассоциации состоит в том, что для поиска вершин, относящихся к данному ребру, достаточно опросить одну ассоциацию. Если квалификатор не указан, можно найти обе вершины. Указав квалификатор, вы можете найти вершину, находящуюся с нужного конца ребра.

Преимущество использования двух ассоциаций состоит в том, что вы устраняете необходимость работать с перечислимыми значениями квалификатора end.

3.25. На рис. ОЗ.30 показана диаграмма классов для системы ссуд на автомобили, в которой указатели заменены ассоциациями.

В такой форме искусственное ограничение на количество работодателей у человека было исключено. Обратите внимание, что в этой модели владельцу может принадлежать несколько машин. На один автомобиль может быть выдано несколько ссуд. Банки ссужают деньги людям, компаниям и другим банкам.

3.28. На рис. ОЗ.34 показана диаграмма классов для задачи об обедающих философах. Ассоциации «один-к-одному» описывают взаимное расположение философов и вилок. Ассоциация *InUse* (*Используется*) показывает, кто именно использует вилки. Возможны и другие представления, в зависимости от вашей точки зрения. Диаграмма объектов поможет вам лучше понять эту задачу.

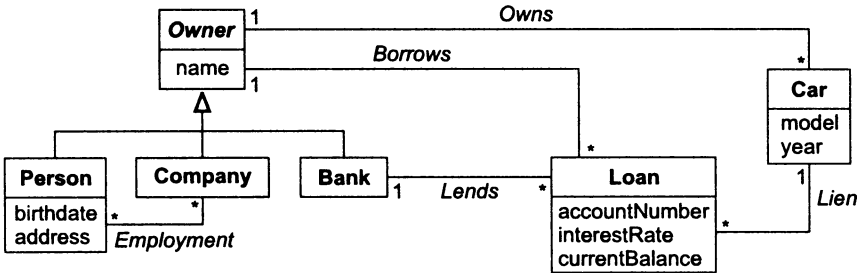


Рис. ОЗ.30. Диаграмма классов для автокредитования

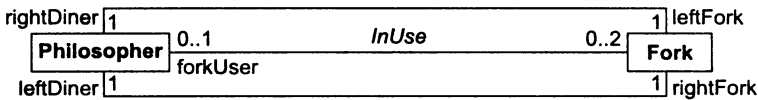


Рис. 03.34. Диаграмма классов для задачи об обедающих философях

- 3.31. Приведенное ниже выражение на OCL возвращает множество авиакомпаний, самолетами которых летал пассажир в течение года.

`aPassenger.Flight->SELECT(getYear(date)=aGivenYear).Airline.Name->asSet`

Оператор `asSet` языка OCL удаляет дубликаты одной и той же авиалинии.

- 3.34. На рис. УЗ.13, *а* обозначено, что подписка имеет выводимую индивидуальность. Рисунок УЗ.13, *б* придает подписке самостоятельное значение и делает ее отдельным классом. Вторая модель лучше, потому что большинство журналов снабжаются кодами подписки на этикетках. Этот код можно хранить в качестве атрибута. Код подписки предназначен для идентификации подписки. Подписка не идентифицируется сочетанием человека и журнала, поэтому *Subscription* следует сделать классом. Более того, один человек может иметь несколько подписок на один журнал. Такую ситуацию может описать только вторая модель.

- 4.2. Диаграмма классов на рис. 04.2 обобщает классы *Selection* (Выделение), *Buffer* (Буфер) и *Sheet* (Лист) суперклассом *Collection* (Совокупность). Обобщение способствует повторному использованию кода, потому что многие операции одинаково применимы ко всем подклассам. Шесть отношений агрегации с исходной диаграммы были заменены двумя. Теперь структура диаграммы отражает ограничение, состоящее в том, что каждый экземпляр *Box* и *Line* должен принадлежать ровно одному экземпляру *Buffer*, *Selection* или *Sheet*.

- 4.4. На рис. 04.3 показана диаграмма классов для графического редактора. Требование, состоящее в том, что в *Group* (Группа) должно содержаться 2 и более *DrawingObject* (ГрафическийОбъект), выражается кратностью 2..* у полюса агрегации *DrawingObject* и *Group*. Тот факт, что *DrawingObject* не обязательно должен входить в *Group*, отражается кратностью 0..1.

Эту диаграмму можно переделать, сделав *Circle* (Окружность) частным случаем *Ellipse* (Эллипс), а *Square* (Квадрат) — частным случаем *Rectangle* (Прямоугольник).

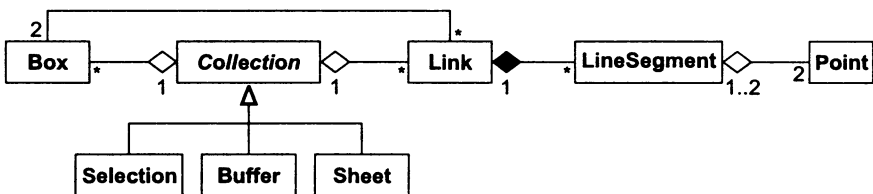


Рис. 04.2. Обобщение классов *Selection*, *Buffer* и *Sheet* классом *Collection*

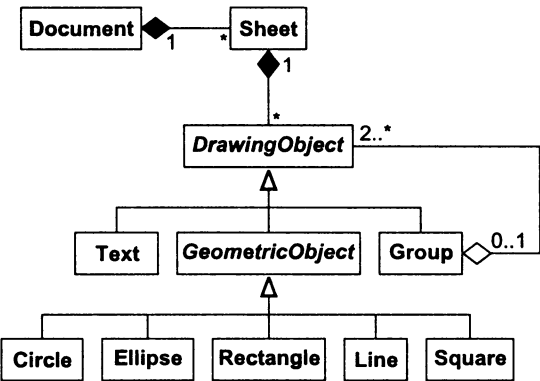


Рис. О4.3. Диаграмма классов для графического редактора, поддерживающего группировку

Мы предполагаем, что *DrawingObject* (*ГрафическийОбъект*) принадлежит какому-либо *Sheet* (*Лист*) и срок существования первого ограничен сроком существования последнего. Кроме того, мы предполагаем, что каждый *Sheet* (*Лист*) принадлежит одному *Document* (*Документ*). Поэтому оба отношения являются композициями.

4.5. На рис. О4.4 показана диаграмма классов с несколькими классами электрических машин. Мы добавили на эту диаграмму атрибуты.

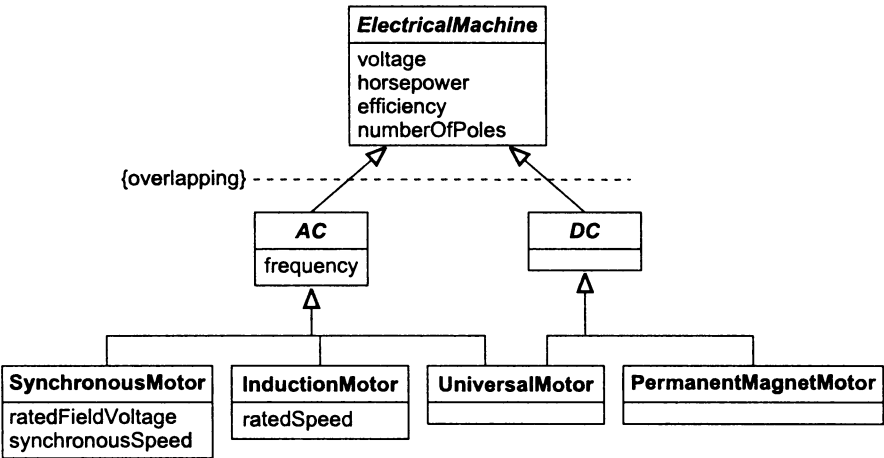


Рис. О4.4. Частичная таксономия для электрических машин

- 4.6. На рис. О4.5 перекрывающаяся комбинация классов была выделена в собственный класс, что позволило отказаться от множественного наследования.
- 4.8. На рис. О4.6 показана метамодель следующих концепций UML: класс, атрибут, ассоциация, полюс ассоциации, кратность, название класса, название атрибута.

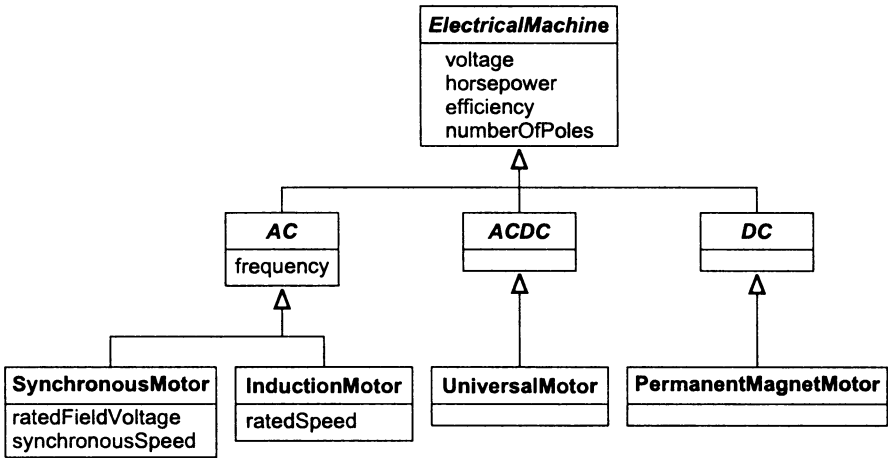


Рис. О4.5. Исключение множественного наследования

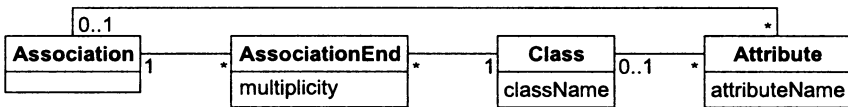


Рис. О4.6. Метамоделю некоторых концепций UML

4.10. Диаграмма классов на рис. У4.3 не поддерживает множественное наследование. Класс может иметь несколько ролей обобщений подкласса, участвующего во множестве обобщений.

4.11. Чтобы обнаружить суперкласс обобщения по рис. У4.3, сначала следует создать запрос по ассоциации между *Generalization* (Обобщение) и *GeneralizationRole* (РольОбобщения), чтобы получить множество всех ролей конкретного экземпляра *Generalization* (Обобщение). Затем нужно выполнить последовательный поиск по этому множеству, чтобы найти роль, значение *roleType* (*munПоли*) которой совпадает с суперклассом. (Остается надеяться, что такой экземпляр обнаружится только один, потому что соответствующее ограничение в модель не заложено.) Наконец, нужно проследить ассоциацию между *GeneralizationRole* (РольОбобщения) и *Class* (Класс), чтобы найти суперкласс.

На рис. О4.9 показана модификация этой диаграммы, упрощающая поиск суперкласса. В этом случае достаточно создать запрос для ассоциации между *Generalization* (Обобщение) и *SuperclassRole* (РольСуперкласса). Затем нужно создать запрос для ассоциации между *SuperclassRole* (РольСуперкласса) и *Class* (Класс), чтобы найти соответствующий экземпляр *Class* (Класс).

На рис. О4.10 показана другая метамодель обобщения, поддерживающая множественное наследование. В этом случае поиск суперкласса сводится к запросу ассоциации *Superclass* (Суперкласс).

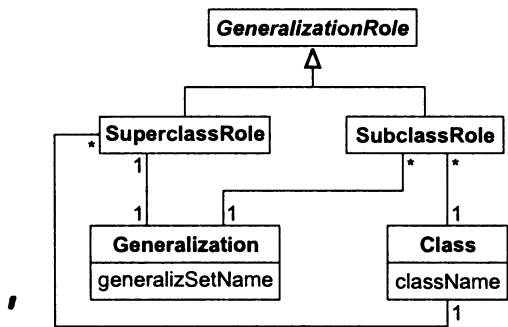


Рис. 04.9. Мета модель обобщений с раздельными ролями подклассов и суперклассов

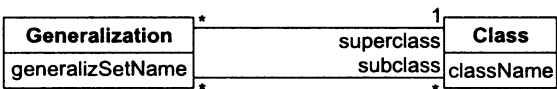


Рис. 04.10. Упрощенная метамодель отношения обобщения

Мы не утверждаем, что метамодель на рис. 04.10 — самая лучшая из всех метамodelей обобщения. Она всего лишь упрощает запрос, рассматриваемый в данном упражнении. Выбор модели определяется, прежде всего, ее назначением.

Приведенный ниже запрос выполняет поиск суперкласса по обобщению для рис. У4.3:

```
aGeneralization.GeneralizationRole->SELECT (roleType=
'superclass').Class
```

То же для рис. 04.9:

```
aGeneralization.SuperclassRole.Class
```

То же для рис. 04.10:

```
aGeneralization.superclass
```

4.16. Простой модели классов с рис. 04.14 достаточно для описания приведенных в примере данных рецепта.

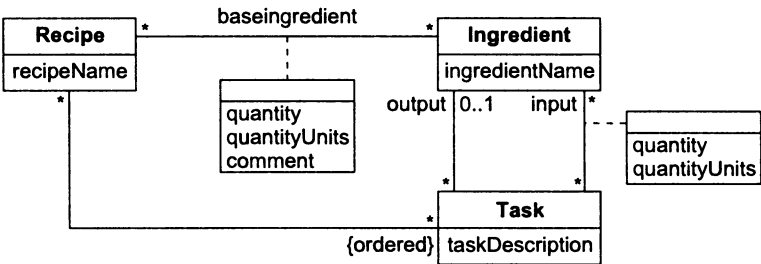


Рис. 04.14. Простая модель классов для рецептов

4.17. На рис. О4.15 показано наше первое решение этого упражнения. Мы просто добавили ассоциацию, которая связывает исходные ингредиенты с заменяющими. У этой модели два недостатка.

Во-первых, модель неудобна для описания взаимозаменяемых ингредиентов. Например, в некоторых рецептах допускается свободная замена масла, маргарина и растительного жира друг на друга. В соответствии с нашей моделью каждую пару взаимозаменяемых ингредиентов пришлось бы хранить отдельно. Поэтому нам пришлось бы ввести следующие пары: (масло, маргарин), (масло, жир), (маргарин, масло), (маргарин, жир), (жир, масло) и (жир, маргарин).

Во-вторых, взаимозаменяемость ингредиентов не абсолютна, она зависит от рецепта.

На рис. О4.16 показана усовершенствованная модель классов, в которой оба недостатка были устранены.

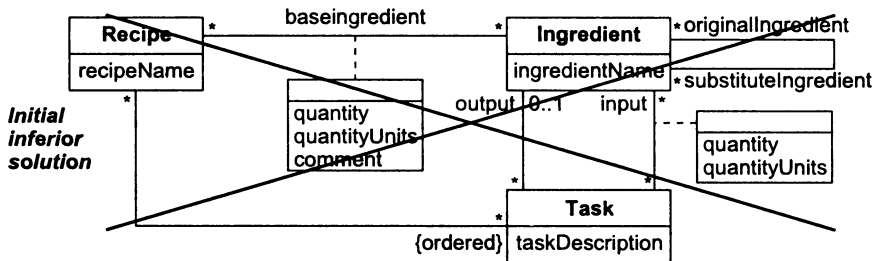


Рис. 04.15. Первая модель классов для рецептов с альтернативными ингредиентами

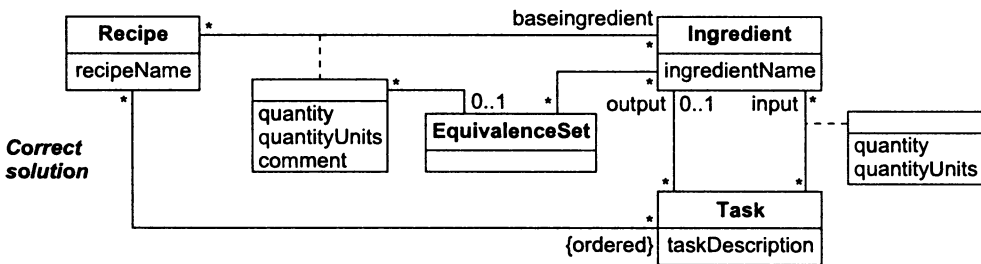


Рис. 04.16. Усовершенствованная модель классов для рецептов с альтернативными ингредиентами

5.2. На рис. О5.2 событие А соответствует нажатию кнопки А. На этой диаграмме отпускание кнопки не имеет отдельного значения, а потому оно не показано (хотя очевидно, что вы должны отпустить кнопку, прежде чем нажать ее снова). Обратите внимание, что новое событие нажатия не может быть порождено до тех пор, пока нажата хотя бы одна из клавиш. Вы можете считать это ограничением на входящие события, которое не обязательно показывать на диаграмме состояний (хотя отразить его тоже не было бы ошибкой).

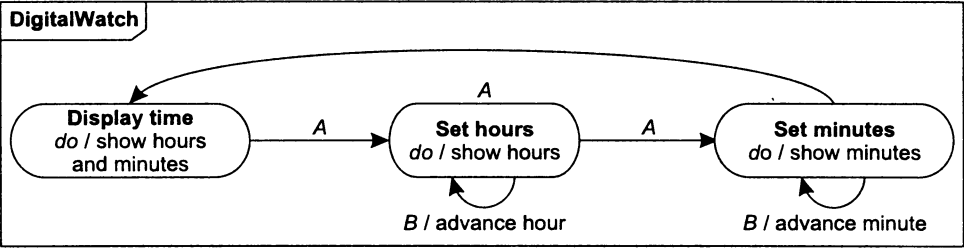


Рис. 05.2. Диаграмма состояний для простых цифровых часов

5.6. На рис. 05.6 показана полная диаграмма состояний для системы управления двигателем.

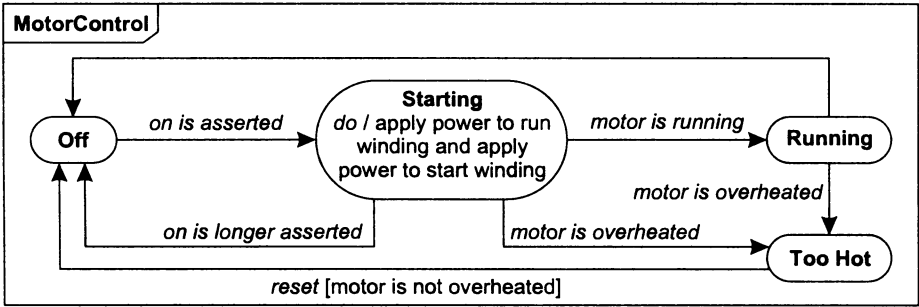


Рис. 05.6. Диаграмма состояний для системы управления двигателем

5.11. На рис. 05.11 показана диаграмма состояний. Обратите внимание, что даже простые диаграммы состояний могут описывать сложное поведение. Событие *change* (изменение) происходит при извлечении свечи из подсвечника или при помещении ее обратно. Условие *at north* (на север) считается выполненным, когда книжный шкаф находится за стеной. Условия *at east*, *south* и *west* выполняются, когда шкаф смотрит вперед, назад или в сторону. Когда вы впервые увидели книжный шкаф, он находился в состоянии *Stopped* (Покой) и указывал на юг (*at south*). Когда ваш друг вынул свечу, событие *change* привело к тому, что шкаф перешел в состояние *Rotating* (Вращение). Когда шкаф повернулся на север (*north*), это условие перевело его в состояние *Stopped*. Когда ваш друг вставил свечу обратно, еще одно событие *change* снова привело шкаф в состояние *Rotating*, в котором он находился до тех пор, пока не стал указывать на север (*north*). Когда вы вытащили свечу еще раз, шкаф снова стал вращаться и повернулся бы на 360 градусов, если бы вы не помешали ему. Возвращение шкафа на место силой не описывается системой управления, а потому не должно быть отражено на диаграмме. Когда вы снова вставили свечу, очередное событие *change* привело его в состояние *Rotating*. Когда вы снова вытащили свечу, новое событие *change* привело шкаф в состояние *Stopping* (Остановка). Через четверть оборота условие *at north, east, south or west* (указывает в любую опреде-

ленную сторону) было выполнено, в результате чего шкаф перешел в состояние *Stopped* (Покой).

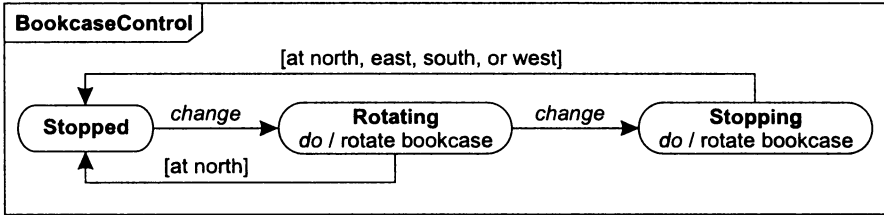


Рис. 05.11. Диаграмма состояний управления книжным шкафом

Чтобы попасть в потайной ход с самого начала, вам нужно было вытащить свечу, а потом быстро вставить ее обратно прежде, чем шкаф повернулся на четверть оборота.

- 6.1. У прожектора и колес поезда имеются собственные диаграммы состояний (рис. 06.1 и 06.2). Обратите внимание, что стационарное состояние колеса содержит несколько вложенных состояний.

Мы указали основные начальные состояния для прожектора и колес. Фактическое начальное состояние колес может быть произвольным (любым из состояний *power off*). Система работает циклично и не зависит от начального состояния, поэтому указывать его не обязательно. Во многих аппаратных системах начальные состояния не определяются.

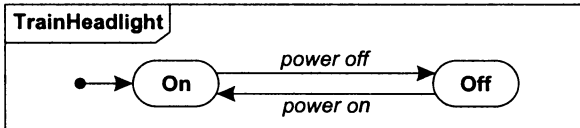


Рис. 06.1. Диаграмма состояний для прожектора игрушечного поезда

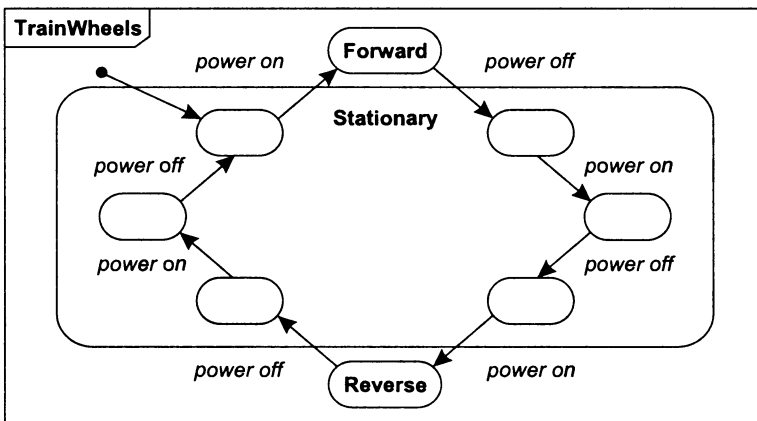


Рис. 06.2. Диаграмма состояний для колес игрушечного поезда

6.3. На рис. 06.4 мы добавили состояние *Motor On* (*Двигатель Включен*), чтобы описать схожесть состояний запуска и работы. Мы показали переход из состояния *Off* (*Выключен*) в состояние *Starting* (*Запуск*). Вместо этого мы могли бы провести переход от *Off* (*Выключен*) к *Motor On* (*Двигатель Включен*) и сделать *Starting* (*Запуск*) начальным состоянием *Motor On*. Обратите внимание, что деятельность *apply power to run winding* (подать питание на рабочую обмотку) была вынесена как из состояния *Running* (*Работа*), так и из состояния *Starting* (*Запуск*).

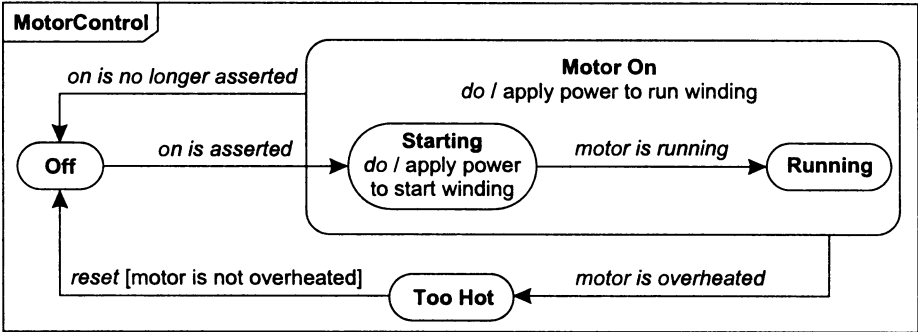


Рис. 06.4. Диаграмма состояний для системы управления двигателем

6.4. На рис. 06.5 показана усовершенствованная диаграмма состояний двигателя. Обратите внимание, что переход из *Off* (*Выключен*) к *Forward* (*Вперед*) или *Reverse* (*Назад*) вызывает неявный переход в состояние *Starting* (*Запуск*), которое является основным начальным состоянием нижней параллельной поддиаграммы. Запрос на выключение приводит к переходу из обеих параллельных поддиаграмм в состояние *Off*.

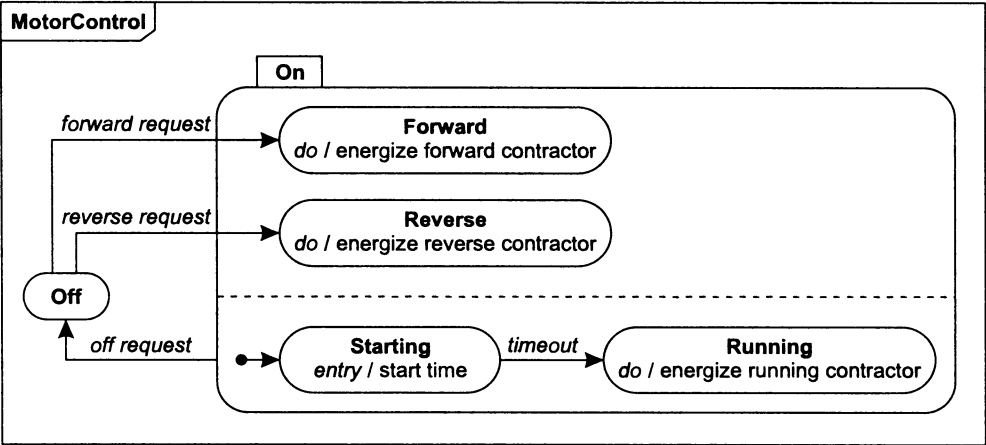


Рис. 06.5. Усовершенствованная диаграмма состояний для управления асинхронным электродвигателем

7.1. Вот ответы для обычного книжного магазина.

1) В качестве действующих лиц можно перечислить:

- *Customer (Клиент)* — человек, инициирующий покупку товара.
- *Cashier (Кассир)* — сотрудник, авторизованный для выполнения проверки покупок на кассе.
- *Payment verifier (Система проверки оплаты)* — удаленная система, подтверждающая возможность использования кредитной или дебетной карты.

2) В качестве вариантов использования можно перечислить:

- **Покупка товаров.** Клиент приносит один или несколько товаров на кассу и платит за них.
- **Возвращение товаров.** Клиент приносит обратно купленные ранее товары и получает за них деньги.

3) На рис. 07.1 показана диаграмма вариантов использования.

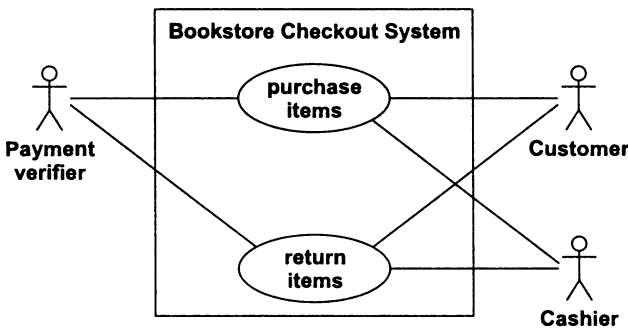


Рис. 07.1. Диаграмма вариантов использования для системы контроля покупок в книжном магазине

4) Мы приводим нормальные сценарии для каждого варианта использования. Правильных ответов можно придумать много.

- **Покупка товаров.** Клиент приносит товары на кассу. Кассир сканирует каждый товар. Кассир вычисляет общую сумму, включая налог. Кассир спрашивает о форме оплаты. Клиент дает кредитную карту. Кассир сканирует карту. Система проверки подтверждает возможность оплаты. Клиент подписывает чек.
- **Возврат товаров.** Клиент приносит купленный ранее товар на кассу. Клиент показывает чек о покупке. Кассир отмечает, что покупка была сделана за наличные. Кассир принимает товары и выдает клиенту деньги наличными.

5) Исключительные сценарии для каждого варианта использования.

- **Покупка товаров.** Клиент приносит товары на кассу. Кассир сканирует каждый товар. На одном из товаров штрих-код поврежден, поэтому кассир идет к полке, чтобы узнать цену.

- **Возврат товаров.** Клиент приносит купленный ранее товар на кассу. Клиент потерял чек о покупке. Клиент получает квитанцию в подтверждение возврата товара, но не денежную компенсацию.

6) На рис. О7.2 показана диаграмма последовательности для первого сценария из пункта 4). На рис. О7.3 показана диаграмма последовательности для второго сценария из пункта 4).

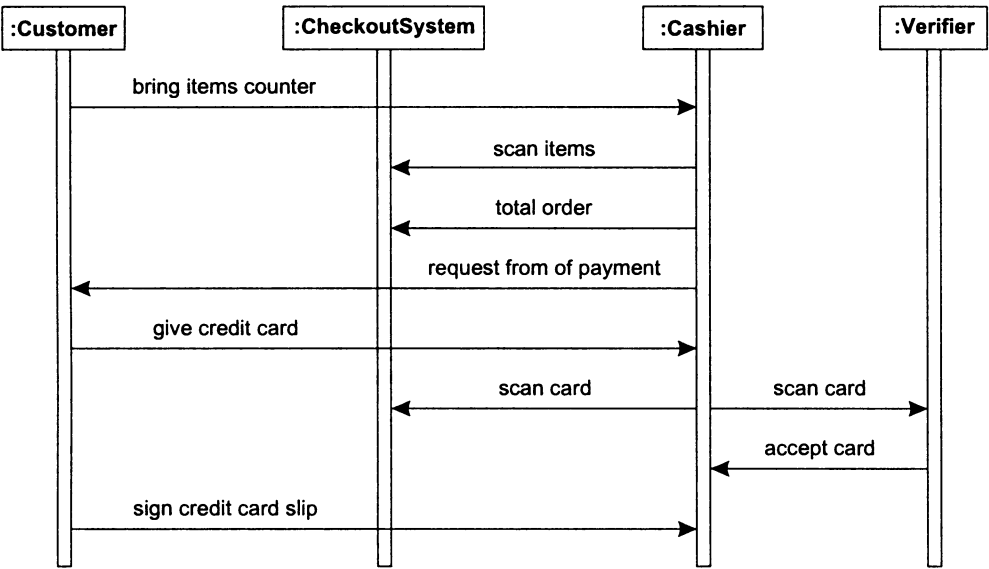


Рис. 07.2. Диаграмма последовательности для покупки товаров

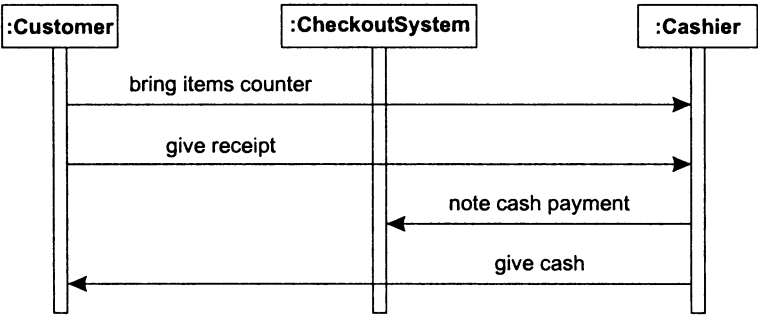


Рис. 07.3. Диаграмма последовательности для возврата товаров

7.8. На рис. О7.12 показана диаграмма деятельности для вычисления счета в ресторане.

8.1. Вот наши ответы для электронной бензоколонки.

1) На рис. О8.1 показана диаграмма вариантов использования.

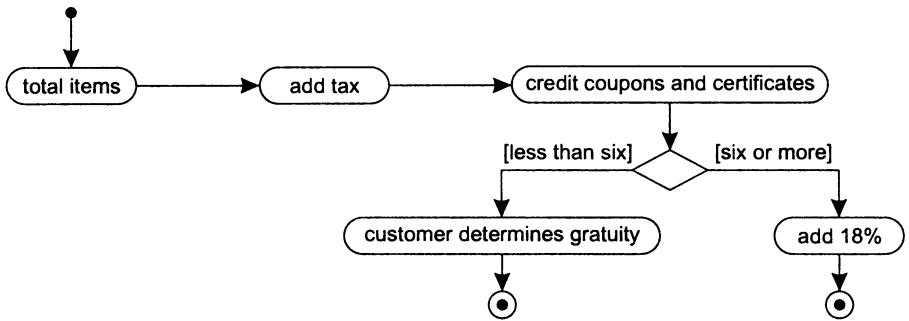


Рис. 07.12. Диаграмма деятельности для вычисления счета в ресторане

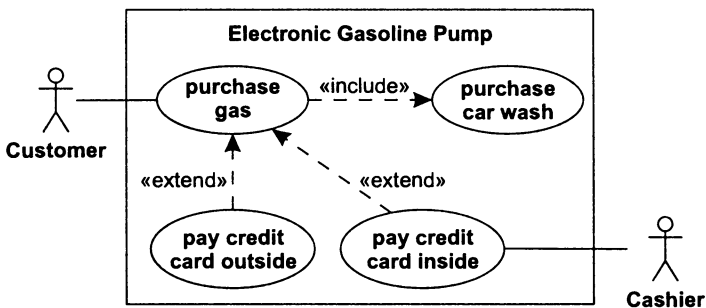


Рис. 08.1. Диаграмма вариантов использования для электронной бензоколонки

2) Действующих лиц двое:

- **Клиент.** Человек, инициирующий покупку бензина.
- **Кассир.** Человек, обслуживающий оплату по кредитной карте и контролирующий продажу бензина.

3) Вариантов использования четыре:

- **Покупка бензина.** Залить бензин и заплатить за него наличными.
- **Мойка машины.** Клиент решает заодно помыть машину и платит за это наличными.
- **Оплата по кредитной карте.** Вместо наличных, для оплаты бензина и необязательной мойки машины используется кредитная карта, обрабатываемая непосредственно электронной бензоколонкой.
- **Оплата по кредитной карте кассиру.** Вместо наличных, для оплаты бензина и необязательной мойки машины используется кредитная карта, которая обрабатывается кассиром вручну.

8.6. На рис. 08.6 показана диаграмма последовательности вычисления стоимости портфеля ценных бумаг.

11.1. Уточним концепцию антиблокировочной тормозной системы для автомобиля.

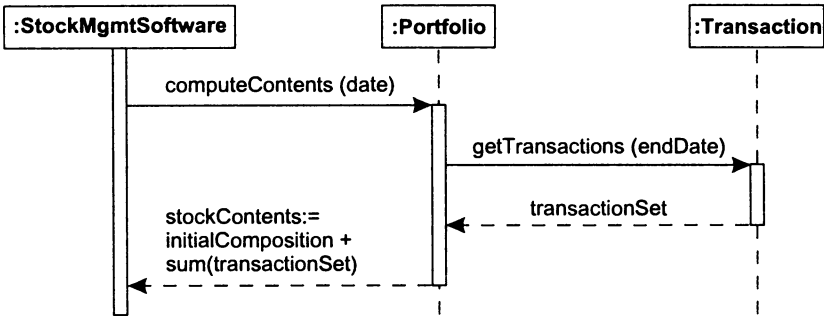


Рис. 08.6. Диаграмма последовательности вычисления стоимости портфеля ценных бумаг

1) Антиблокировочная тормозная система может быть ориентирована на массовый рынок. Если она будет более дешевой и безопасной, чем современные модели, то правительство может сделать обязательной ее установку на все автомобили. (Для определения конкретной стоимости и безопасности, при которых это может произойти, потребуется дальнейшее исследование.)

Заинтересованных лиц несколько. Автовладельцам выгодно повышение безопасности и минимальный ущерб удобству вождения. Производители стремятся снизить стоимость и измерить выигрыш, чтобы предлагать новый товар в своей рекламе. Правительство стремится к статистическому повышению безопасности без снижения топливной экономичности.

Если новая система будет дешевой, будет хорошо работать и не будет влиять на удобство вождения, потенциальными клиентами будут все автовладельцы. На дорогих машинах можно устанавливать дорогую антиблокировочную систему.

- 2) В качестве желательных характеристик можно указать: эффективность предотвращения блокировки колес, возможность обнаружения чрезмерного износа тормозов, получение данных для упрощения обслуживания автомобиля. Нежелательные характеристики: снижение топливной экономичности, снижение удобства вождения, усложнение обслуживания.
- 3) Антиблокировочная тормозная система должна работать с тормозами, рулевой системой и электроникой автомобиля.
- 4) Существует вероятность того, что антиблокировочная система откажет, в результате чего произойдет несчастный случай и будет возбуждено дело в суде. Кроме того, достичь полного понимания взаимодействия антиблокировочной системы с тормозами может быть довольно сложно.

12.9. Нужно исключить следующие потенциальные классы.

- Избыточные классы: *Спортсмен*, *Участник соревнований*, *Индивидуум*, *Человек*, *Регистрирующийся* (все они дублируют класс *Соревнующийся*).

- Нечеткие или несущественные классы: *Спина, Карта, Заключение, Угол, ИндивидуальныйПриз, КомандныйПриз, Попытка, СинхронноеПлавание*.
- Атрибуты: адрес, возраст, среднее количество очков, имя спортсмена, дата, коэффициент сложности, суммарный счет, судейские очки, счет, название команды.
- Конструкции, относящиеся к реализации: файлДанныхОЧленахКоманды, списокЗапланированныхСоревнований, группа, номер.
- Выводимый класс: возрастнаяКатегория вычисляется по возрасту Соревнующегося.
- Операции: вычислениеСреднего, регистрация.
- Не входит в область модели: Групповые выступления.

После удаления всех этих классов у нас останутся только *Соревнующийся, Соревнование, СольноеВыступление, Судья, Лига, Этап, Сектор, Команда, Сезон, Секретарь* и *Попытка*.

12.12. Для описания запросов мы воспользуемся комбинацией OCL и псевдокода.

В некоторых ответах мы прослеживаем целые цепочки связей. В главе 15 объясняется, что каждый класс должен иметь ограниченные сведения о модели классов и операции класса не должны прослеживать ассоциации, в которых он сам не участвует. Мы нарушили этот принцип, чтобы ответ был более простым. В более устойчивой системе нам бы пришлось использовать промежуточные операции, позволяющие исключить проследивание длинных цепочек.

1) Поиск всех членов команды.

```
Team::retrieveTeamMembers ()
returns set of competitors
return self.competitor;
```

3) Найти суммарное количество очков, полученных участником за данное упражнение на данном соревновании. Эту задачу можно решить несколькими способами. Мы приводим только один пример.

```
Competitor::findNetScore (figure, meet)
returns netScore
    event:= meet.event intersect figure.event;
    / * приведенный выше код должен возвращать ровно одно
        соревнование, иначе это будет ошибка реализации. Это
        ограничение присутствует в постановке задачи неявно
        и не было отражено в модели классов. */
    trial := event.trial intersect self.trial;
    if triall == NIL then return ERROR
    else return trial.netScore;
end if
```

5) Найти среднее количество очков, набранных данным участником по всем упражнениям на данных соревнованиях.

```
Competitor::findAverage (meet) returns averageScore
    trials:= meet.event.trial intersect self.trial;
    if trials == NIL then return ERROR
```

```

else
    /* вычислить среднее, как в пункте 4) */
    return average;
end if;

```

- 7) Найти множество всех участников каких-либо соревнований в данном сезоне.

```

Season::findCompetitorsForAnyEvent ()
returns set of competitors
return self.meet.event.trial.competitor;

```

- 12.14. Усовершенствованные диаграммы показаны на рис. O12.7—O12.10. Диаграмма с рис. O12.7 лучше, потому что `dateTime` — всего лишь атрибут. Диаграмма с рис. O12.8 лучше, потому что *UniversityClass* (*УниверситетскийКурс*), скорее всего, будет классом с атрибутами и операциями. Третья тернарная ассоциация не атомарна, потому что комбинация *Seat* (*Место*) и *Concert* (*Концерт*) определяют *Человека* (*Person*). Четвертая тернарная ассоциация также не атомарна, ее можно переформулировать как две бинарных ассоциации.
- 13.14. Приложение работает с данными о соревнованиях лиги синхронного плавания. Система хранит выставленные судьями очки и вычисляет различные среднестатистические данные.
- 13.15. В качестве действующих лиц можно указать соревнующегося, секретаря, судью и команду.

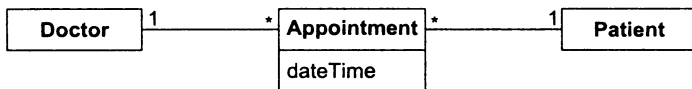


Рис. O12.7. Диаграмма классов для встреч

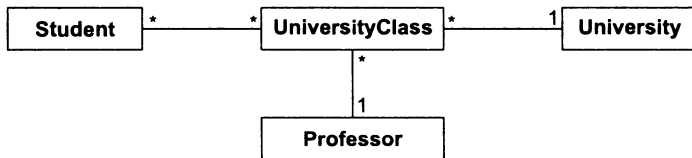


Рис. O12.8. Диаграмма классов для университетских курсов

- 13.16. Вот определения вариантов использования, диаграмма которых показана на рис. O13.12.
- **Регистрация участника.** Новый участник добавляется в систему. Вводится его имя, возраст, адрес и название команды, к которой он принадлежит. Участнику назначается номер.
 - **Планирование соревнования.** Участники распределяются по упражнениям. Определяются стартовые времена. Секретари и судьи распределяются по секторам.

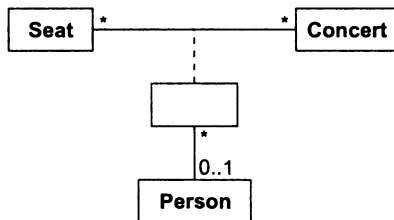


Рис. 012.9. Диаграмма классов для системы бронирования

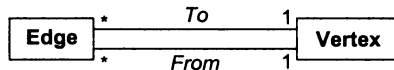


Рис. 012.10. Диаграмма классов для неориентированных графов

- **Планирование сезона.** Определяется список соревнований в рамках сезона. Для каждого соревнования определяется дата, список упражнений и список соревнующихся команд.
- **Оценка упражнения.** Судья следит за выступлением участника на упражнении и выставляет ему оценку.
- **Определение очков.** Секретарь получает от судей их оценки и вычисляет суммарное количество очков.
- **Вычисление статистики.** Система вычисляет требуемую статистику, такую как максимальный индивидуальный результат в упражнении или суммарный командный результат на соревновании.

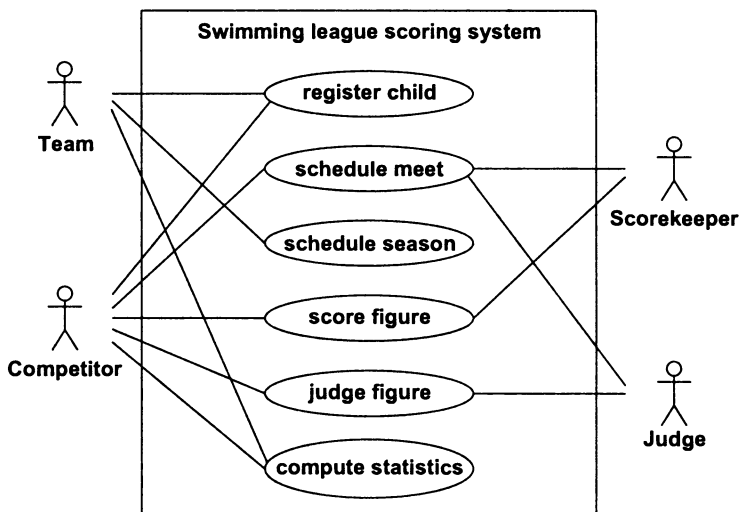


Рис. 013.12. Диаграмма вариантов использования для системы подсчета очков лиги синхронного плавания

13.21. На рис. 013.14 показаны «операции по списку» (не все).

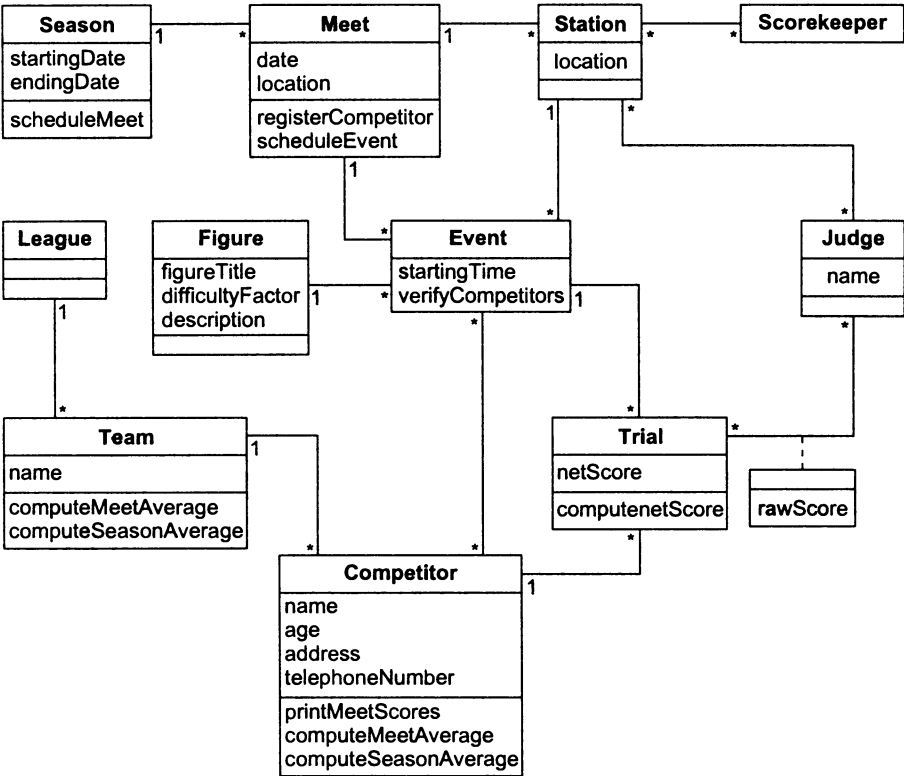


Рис. O13.14. Часть диаграммы классов для системы подсчета очков (с операциями)

14.6. На рис. O14.1 показан один из возможных вариантов разбиения.

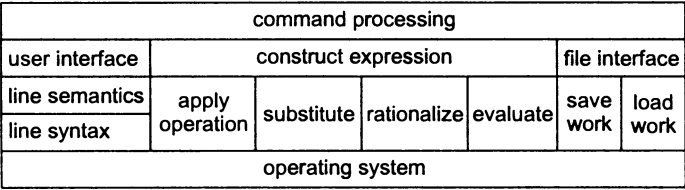


Рис. O14.1. Структурная схема интерактивной символьной системы для работы с полиномами

14.7. Одна программа обеспечивает быстрое выявление и исправление ошибок, устраняет необходимость реализации интерфейса между двумя программами. В этом случае любые ошибки, обнаруженные системой в процессе преобразования диаграммы классов в схему базы данных, могут быть быстро переданы пользователю для исправления. Кроме того, части программы, отвечающие за редактирование и преобразование, могут работать с одними и теми же данными, благодаря чему исключается потребность в интерфейсе (например, файле), который служил бы для передачи диаграммы классов из одной программы в другую.

Деление функциональности между двумя программами снижает требования к памяти и делает возможной независимую разработку программ. Требования к памяти, предъявляемые единой программой, будут приблизительно соответствовать сумме требований двух отдельных программ. Поскольку обе программы будут, скорее всего, нуждаться в больших объемах памяти, их объединение может вызвать проблемы с производительностью. Деление на две программы упрощает их разработку, которая может осуществляться независимо. Изменения в одной программе с меньшей вероятностью затронут другую программу. Две программы отлаживать проще, чем одну большую. Если интерфейс между программами будет жестко определен, проблемы в системе легко можно будет связать с одной из ее частей, которой они вызваны.

Еще одно преимущество деления системы на две части состоит в повышении гибкости. Редактор может использоваться с другими преобразующими программами, например с системой, генерирующей объявления классов на каком-либо языке программирования. Генератор схемы базы данных может быть адаптирован к другим графическим редакторам.

14.10. Рассмотрим каждое из возможных решений.

- 1) *Не беспокоиться о потере данных.* Каждый раз при включении системы полностью сбрасывать все данные. Это самый дешевый и самый простой подход. Его относительно легко запрограммировать, потому что достаточно всего лишь написать подпрограмму инициализации, которая будет запускаться каждый раз при включении питания и будет давать пользователю возможность ввести параметры. Однако этот подход неприемлем для систем, которые должны обеспечивать непрерывное обслуживание клиентов, или для тех, где потеря данных при отключении питания недопустима.
- 3) *Хранить критическую информацию на магнитном диске.* Периодически делать полные и добавочные копии на магнитной ленте. Этот подход характеризуется средней стоимостью и средним уровнем сложности. В случае отказа питания система прекращает работу. Для работы с диском и магнитной лентой требуется работоспособная система. Для работы с лентами требуется присутствие оператора, что исключает возможность применения данного решения для полностью автоматизированных приложений.
- 5) *Использовать специальную память, такую как память на магнитных пузырьках или энергонезависимую память EEPROM.* Это дешевый подход, допускающий автоматизацию. Однако в отсутствие питания система все равно не сможет работать. Кроме того, энергонезависимая память может накладывать дополнительные ограничения, такие как количество операций перезаписи или объем хранимых данных. Для сохранения важных параметров при обнаружении отказа питания может потребоваться специальная программа.

- 14.11. 1) *Карманный бухгалтерский калькулятор*. Не беспокоиться о потере данных. Все остальные варианты слишком дороги. Такие калькуляторы продаются по низкой цене и используются бухгалтерами. Требования к объему памяти — порядка 10 байтов.
- 3) *Системные часы персонального компьютера*. Требуется всего несколько байтов памяти, но часы должны работать при отключении питания. Дешевое решение — резервная батарейка. Электронные часы могут работать от такой батареи в течение 5 лет и более.
- 5) *Блок управления и защиты двигателя от перегрева*. Требуется порядка 10—100 байтов памяти. Для этого приложения важным критерием является цена. Источник бесперебойного питания стоит слишком дорого. Ленточные и дисковые накопители слишком ненадежны, учитывая условия работы. Нужно использовать переключатели, специальные виды памяти и резервную батарею. При помощи переключателей можно вводить параметры (интерфейс системе управления требуется в любом случае). В специальной памяти можно хранить вычисленные данные. Батарейку можно использовать для продолжения работы при отключении питания, однако в данном приложении батарея вызовет сложности в обслуживании. Мы предложили бы пересмотреть последнее требование, например установить датчик для измерения температуры двигателя или предположить, что двигатель может быть нагрет при первом включении.

- 14.12. 1) Описание диаграммы выглядит следующим образом:

```
(DIAGRAM
  (CLASS
    (NAME "Polygon"))
  (CLASS
    (NAME "Point")
    (ATTRIBUTE "x")
    (ATTRIBUTE "y"))
  (ASSOCIATION
    (END (NAME "Polygon") ONE)
    (END (NAME "Point") MANY)))
```

- 14.13. Аппаратное решение работает быстрее всего, но стоит дороже. Программное решение дешевле и более гибкое, но может быть недостаточно быстрым. Программный подход следует использовать везде, где его производительности хватает. В универсальных системах он предпочтительнее благодаря гибкости. В специальных системах добавочные схемы интегрируются с дополнительным оборудованием.

Вообще говоря, существует еще один подход — программно-аппаратный (firmware). Обычно аппаратный контроллер вычисляет контрольную сумму в соответствии с закодированной в постоянной памяти и недоступной извне программой. Этот подход мы относим к аппаратным решениям.

- 1) *Контроллер гибкого диска*. Используйте аппаратный подход. Гибкость тут не нужна, потому что такой контроллер представляет собой спе-

циализированную систему. Потребность в высокой производительности обусловлена высокой скоростью передачи данных.

- 3) *Плата памяти в компьютере космического корабля.* Используйте аппаратный контроль памяти. Это пример специального приложения, где функциональность можно интегрировать в микросхемы плат памяти. Скорость передачи данных очень высокая.
- 5) *Проверка номера счета.* Используйте программный подход. Скорость передачи данных очень низкая. Система, обрабатывающая номер счета, скорее всего, работает на универсальном компьютере.

15.6. Рисунок О15.1 накладывает ограничение, отсутствующее на рис. У15.1. Каждый *BoundingBox* (*Ограничивающий Прямоугольник*) соответствует ровно одному экземпляру *Ellipse* (*Эллипс*) или *Rectangle* (*Прямоугольник*). Одним из критериев качества модели классов является отражение ограничений системы в структуре этой модели.

Мы указали возможность вычисления объекта *BoundingBox*, который определяется параметрами геометрической фигуры и не несет дополнительной информации.

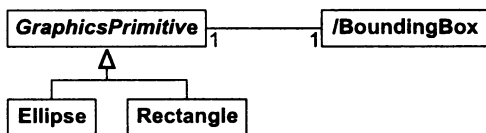


Рис. О15.1. Усовершенствованная диаграмма классов для ограничивающего прямоугольника

15.9. Выводимая ассоциация на рис. О15.3 обеспечивает прямое прослеживание от *Page* (*Страница*) к *Line* (*Строка*). Выводимые сущности ускоряют выполнение некоторых запросов, но требуют дополнительных затрат ресурсов на обновление выводимых данных при изменении исходных данных. Ассоциация *Page_Line* (*Страница_Строка*) представляет собой композицию ассоциаций *Page_Column* (*Страница_Колонка*) и *Column_Line* (*Колонка_Строка*).

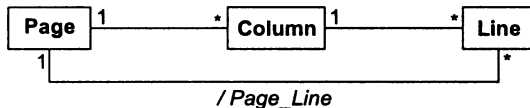


Рис. О15.3. Усовершенствованная модель издательской системы

15.13. Ниже приведен набросок решения. В нашем коде отсутствуют утверждения, которые обычно включаются для проверки правильности. В частности, следовало бы включить оператор для обработки ситуации, когда полюс является подклассом, а отношение не является обобщением. Если код взаимодействует с пользователями или внешними источниками данных, лучше всего включать в него операторы обработки ошибок

в качестве альтернативной ветви при проверке условий, которые «должны» быть истинными.

```

traceInheritancePath (class1, class2): Path
{
    path := new Path;
// поиск пути от class1 как от потомка class2
    classx := class1;
    while classx is not null do
        add classx to front of path;
        if classx = class2 then return path;
        classx := classx.getSuperclass();
// не удалось найти путь от class1 вверх до class2
// ищем путь от class2 вверх к class1
    path.clear();
    classx := class2;
    while classx is not null do
        add classx to front of path;
        if classx = class1 then return path;
        classx := classx.getSuperclass();
// два класса не состоят в непосредственной связи
// возвращается пустой путь
    path.clear();
    return path;
}

Class::getSuperclass (): Class
{
    for each end in self.connection do:
        if the end is a Subclass then:
            relationship := end relationship;
            if relationship is a Generalization then:
                otherEnds := relationship.end;
                for each otherEnd in otherEnds do:
                    if otherEnd is a Superclass then:
                        return otherEnd.class
    return null;
}

```

- 15.16. На рис. О15.5 показана усовершенствованная модель. Членство в политической партии — это не неотъемлемое свойство избирателя, а изменяемая ассоциация. Новая модель лучше описывает избирателей, не принадлежащих к политическим партиям, и позволяет изменять принадлежность к партиям. Если бы избиратели могли принадлежать сразу к нескольким партиям, мы легко могли бы изменить кратность в своей модели. В нашем случае партия — это экземпляр класса *PoliticalParty*, а не подкласс и ее не нужно указывать в модели явным образом. Новые партии могут добавляться в систему без изменения модели вместе с атрибутами для их описания.

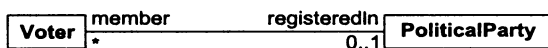


Рис. О15.5. Усовершенствованная модель с воплощением партий

15.18. Левая модель на рис. О15.6 показывает индекс множества точек, реализованный через ассоциацию с двойным квалификатором. Ассоциация сортируется сначала по квалификатору x , а затем по квалификатору y . Поскольку индексирование — это оптимизация, модель содержит избыточную информацию, которая уже присутствует в объектах класса *Point*.

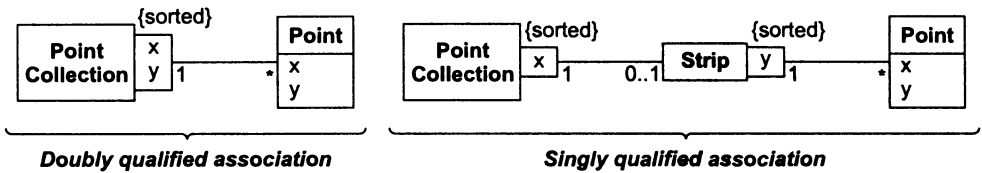


Рис. О15.6. Модели для упорядоченной совокупности точек

Правая модель отражает ту же систему, но с одиночными квалификаторами ассоциаций. Мы добавили избыточный класс *Strip* (*Полоса*), характеризующий все точки с заданной координатой x . Правую модель проще реализовать в большинстве систем, поскольку структура данных с единственным ключом сортировки имеется в большинстве библиотек классов. В фактической реализации для представления ассоциации могут использоваться бинарные деревья, связанные списки или массивы.

Приведенный ниже код определяет методы поиска, добавления и удаления точек.

```

PointCollection::search (region: Rectangle): Set of Point
{
    создать новое пустое множество точек;
    сканировать значения  $x$  в ассоциации до  $x \geq \text{region.xmin}$ ;
    while квалификатор  $x \leq \text{region.xmax}$  do:
        сканировать значения  $y$  для значения  $x$  до  $y \geq \text{region.ymin}$ ;
        while квалификатор  $y \leq \text{region.ymax}$  do:
            добавить  $(x, y)$  в множество точек;
            перейти к следующему значению  $y$ ;
        перейти к следующему значению  $x$ ;
    вернуть множество точек;
}

PointCollection::add (point: Point)
{
    сканировать значения  $x$  в ассоциации до  $x \geq \text{point.x}$ ;
    if  $x = \text{point.x}$  then
        сканировать значения  $y$  для значения  $x$  до  $y \geq \text{point.y}$ ;
        вставить точку в ассоциацию в текущей позиции;
}

PointCollection::delete (point: Point)
{
    сканировать значения  $x$  в ассоциации до  $x \geq \text{point.x}$ ;
    if  $x = \text{point.x}$  then
        сканировать значения  $y$  для значения  $x$  до  $y \geq \text{point.y}$ ;

```

```

    if y = point.y then
        для каждой точки совокупности с текущими значениями x, y
            if точка совокупности = point then
                удалить точку совокупности; return;
    вернуть ошибку «точка не найдена»;
}

```

Обратите внимание, что операцию сканирования нужно реализовать через бинарный поиск, чтобы получить логарифмическое, а не линейное время поиска. Если значения заканчиваются, операция поиска передает управление следующему оператору.

17.2. Стрелка указывает, что ассоциация реализуется в соответствующем направлении.

- *Text* <-> *Box*. Пользователь может редактировать текст, а прямоугольник должен изменять свой размер в соответствии с текстом, поэтому необходим указатель от *Text* к *Box*. Текст может существовать только в прямоугольниках, поэтому предполагается, что пользователь может перетащить прямоугольник, в результате чего текст также должен переместиться. Поэтому необходим указатель от *Box* к *Text*.
- *Connection* <-> *Box*. Прямоугольник можно перетащить, и при этом переместятся его соединения, поэтому необходим указатель от *Box* к *Connection*. Можно перетащить связь, и тогда переместятся ее соединения с прямоугольниками, поэтому необходим указатель от *Connection* к *Box*. Очевидного способа упорядочения в данном случае нет.
- *Connection* <-> *Link*. То же объяснение, что и в предыдущем пункте.
- *Collection* -> *ordered Box*. По заданной совокупности мы должны иметь возможность найти входящие в нее прямоугольники. Необходимость прослеживания в обратном направлении, судя по всему, отсутствует. Упорядочение прямоугольников необходимо для отслеживания их расположения на экране (передний план или задний план).
- *Collection* -> *ordered Link*. То же объяснение, что и в предыдущем пункте.

18.6. 1) Мы приводим пример кода на Java, который реализует двустороннюю ассоциацию между классами A и B типа «один-к-одному» при помощи указателя в каждом классе. Каждый класс обеспечивает поддержку своего собственного полюса ассоциации и обращается к другому классу для обслуживания второго полюса. Каждый класс содержит внутренний атрибут `_updateInProgress` (_выполняетсяОбновление), который позволяет прервать бесконечную рекурсию. Мы приводим только атрибуты и методы, необходимые для реализации ассоциации.

Мы приводим код только для одного класса, потому что код во втором классе должен быть точно таким же, за тем исключением, что классы A и B и объекты a и b должны быть заменены на B, A и b, соответственно. Таким образом, поле `private B b` должно стать `private A a`, а метод `A.SetB(B newB)` станет `B.SetA(A newA)`. Обратите внимание, что мы

до минимума сократили проверку ошибок и опустили возвращение логических переменных или перечислимых типов, а также обработку исключений. Этот код предполагает наиболее жесткий уровень управления доступом. Классы считаются принадлежащими к разным пакетам, а потому могут обращаться только к открытым элементам друг друга.

```
// Java
// класс A с ассоциацией типа один-к-одному с классом B

import BPackage.*;

public class A {
    private B b = null;
    private boolean _updateInProgress = false;

    // проверка, есть ли B для A
    public boolean hasB () {
        return b != null;
    }

    // установка ассоциации A с newB
    public void setB (B newB) {
        if (newB == null) return; //не устанавливаем ассоциацию,
    // нужно вызвать removeB
        if (_updateInProgress) return; //разрываем рекурсию
        if (b = newB) return; // этот A уже связан с B
        if (newB.hasA()) return;
            // newB должен быть свободен
        if (hasB()) removeB();
            // удаление текущей ассоциации, т.к. допустимо только 1:1
        _updateInProgress = true;
        newB.setA(this);
            // запрашиваем newB об обновлении его полюса
        b = newB; // обновление данного полюса
        _updateInProgress = false;
    }

    // Удаление ассоциации с B
    // в ассоциации 1:1 аргумент при удалении не требуется
    public void removeB() {
        if (hasB() == false ) return; // нечего удалять
        if (_updateInProgress) return; // разрываем рекурсию

        _updateInProgress = true;
        b.removeA();
            // запрос на удаление
        b = null; // удаление своего полюса
        _updateInProgress = false;
    }
};
```

Достаточно часто классы, ссылающиеся друг на друга в своих интерфейсах, помещаются в один пакет, а потому могут иметь расширенные знания друг о друге. В таком случае можно передать ответственность за управление обоими полюсами одному классу. Это обеспечит оптимизацию, позволит централизовать код обновления и исключит потребность в средствах

прерывания рекурсии (типа `_updateInProgress`). Однако это повышает уровень взаимной зависимости классов.

Это не обязательно плохо и не обязательно противоречит принципам инкапсуляции. Логическая зависимость уже существует, она отражена в структуре интерфейсов. Возможность локализации операций позволит сделать код более безопасным, более аккуратным и даже более инкапсулированным (с точки зрения внешнего наблюдателя), несмотря на то, что внутренние элементы одного класса станут доступными другому классу. Рассмотрим ситуацию, когда способность управлять завершением связи должна быть предоставлена только участвующему в ней объекту. В языке Java открытые методы `remove()` могут быть сделаны пакетными, и после помещения А и В в один пакет вызов этих методов можно будет осуществлять только участникам связи, но не внешним пакетам.

С другой стороны, можно сделать указатели классов А и В открытыми, что позволит одному из полюсов выполнять обновления полностью. Приведем важнейшие составляющие этого решения на C++.

```
//C++
class B; // объявление

class A {

    friend class B; // можно указывать и для отдельных функций
    // friend void B::setA(A&);
    // friend void B::removeA();
    B* b;

    void removeB(); // В может попросить А удалить связь с В

public:
    bool hasB () { return b != 0; }
    void setB (B& newB);
        // или использовать указатель, чтобы разрешить null b
};

void A::setB(B* newB)
{
    if (b == &newB) return; // этот А уже связан с В
    if (newB.hasA()) return; // newB должен быть свободен
    if (hasB()) removeB();
        // удаление текущей связи
    b = &newB; // обновление своего полюса
    b->a = this;
}

void A::removeB()
{
    if ( !hasB() ) return; // удалять нечего
    if (b->a != this ) return; // связь не двусторонняя

    b->a = 0; // удаление указателя b на этот А
    b = 0;   // удаление своего полюса
}
```


19.13. Из постановки задачи мы заключаем, что маршруту принадлежат два города. Это нельзя вывести только из кода на SQL.



Рис. O19.12. Модель классов для рис. У19.6

19.14. Код на SQL для определения расстояния между двумя городами на рис. У19.6.

```

SELECT distance
FROM Route R, City C1, City C2, City_Distance CD1, City_Distance CD2
WHERE    C1.city_ID = CD1.city_ID AND
         CD1.route_ID = R.route_ID AND
         R.route_ID = CD2.route_ID AND
         CD2.city_ID = C2.city_ID AND
         C1.city_name = :aCityName1 AND
         C2.city_name = :aCityName2;
    
```

20.3. 1) Это пример плохого стиля программирования. Предположение, что аргументы имеют разрешенные значения, а вызванные функции будут хорошо себя вести, вызовет неприятности в процессе тестирования и интеграции программы.

Приведенные ниже операторы вызовут сбой программы при условии, что `strlen == 0`.

```

rootLength = strlen(rootName);
suffixLength = strlen(suffix);
    
```

Приведенный ниже оператор сделает `sheetName` нулевым, если закончится память, в результате чего возникнет сбой при обращении к `strcpy`.

```

sheetName = malloc(rootLength + suffixLength + 1);
    
```

Следующие операторы вызовут сбой программы, если любой из их аргументов будет нулевым.

```

sheetName = strcpy(sheetName, rootName);
sheetName = strcat(sheetName, suffix);
    
```

Если `sheetType` имеет недопустимое значение, оператор `switch` оставит `sheet` без определенного значения. Кроме того, по какой-либо причине функции `vertSheetNew` или `horizSheetNew` могут вернуть нуль. В любом случае приведенный ниже оператор вызовет сбой программы:

```

sheet->name = sheetName;
    
```

Алфавитный указатель

С—U

C++, 372–376, 378, 379–385,
387–397, 399–408

ER, 74

FSF, 373

Java, 372, 374–376, 378–397,
399–408

JVM, 375

Null, 69

OCL, 67

 построение выражений, 68

UML

 история, 27

А—Б

абстрагирование, 35

 изменение уровня, 240

 поведения, 342

абстракция, 24

агрегация, 93

 параллелизм, 143

администратор транзакций, 311

анализ, 21, 203

 инженерный, 480

 итерации, 245

 обзор, 218

 предметной области, 218

аналитическая модель

 наследование, 235

 образец, 241

 проверка маршрутов, 236

 словарь данных, 224

аргумент

 направление, 48

архитектура системы, 21, 286

 стили, 306

ассоциация, 49

 n-арная, 91

 анализ прослеживания, 365

 двусторонняя, 365

 исключение, 225

 квалифицированная, 58

 класс, 56

 односторонняя, 365, 389

 полюс, 53

 реализация, 363, 367, 388

 упорядочение, 55

атрибут, 18, 43, 45

 выводимый, 400

 выделение, 231

 обозначение, 45

 значение по умолчанию, 46

 исключение, 232

 реализация, 400

база данных, 409

 выбор, 413

 инкапсуляция, 431

 нормальная форма, 412

 объектно-

 ориентированная, 432

 реализация, 420

 реализация ассоциаций, 416

 реализация обобщений, 418

 реализация

 функциональности, 428

 реляционная, 410

 сохраненная процедура, 429

 схема, 409

библиотека, 288

быстрое прототипирование, 457

В—Д

вариант использования, 37,

162, 261

 включение, 180, 267

 диаграмма, 165

 идентификация, 261

 начальные и конечные

 события, 262

 обобщение, 182, 267

 расширение, 181, 267

 реализация, 326

 структурирование, 267

 сценарий, 263

взаимодействие

 моделирование, 259

видимость, 89

 включение, 180

 водопадная разработка, 457

 воплощение, 102

 воплощение поведения, 340

 время жизни, 186

 глобальные ресурсы, 300

 действие, 126

 действующее лицо, 162, 261

 выделение, 260

 делегирование, 99, 344

 денормализация, 412

 деятельность, 126, 172

 модель, 171

 параллельная, 174

 при входе, 127

 при выходе, 127

 текущая, 126

диаграмма

 вариантов использования, 165

 деятельности, 37, 267

 выполняемая, 175

 классов, 37, 44

 конечного автомата, 37

 последовательности, 37,

168, 266

 состояний, 37, 122

 построение, 273

диаграмма состояний

 построение, 244, 273

 проверка, 244

динамическое

 моделирование, 309

З—К

запрос, 106

значение, 45

индивидуальность, 17, 43

инженерный анализ, 480

 входные данные, 481

 выходные данные, 482

 модель взаимодействия, 484

 модель классов, 482

 модель состояний, 484

 рекомендации, 485

инкапсуляция, 24, 373

интерактивный интерфейс, 308

интерфейс пользователя, 269

использование

 повторное, 25

 совместное, 25

итерационная разработка, 456

 и моделирование, 461

 идентификация рисков, 462

итерация

 выполнение, 459

 масштаб, 458

 планирование, 460

каркас, 290
 квалификатор, 58
 класс, 18, 43
 абстрактный, 95
 ассоциации, 56
 базовый, 387
 выделение, 221
 группировка в пакеты, 241
 назначение операций, 331
 обозначение, 44
 пограничный, 270
 подкласс, 18
 порожденный, 387
 проектирование, 22, 205
 реализация, 381
 суперкласс, 18
 экземпляр, 18
 классификация, 18
 ключ
 внешний, 412
 возможный, 412
 основной, 412
 количество элементов, 51
 композиция, 94
 конечный автомат
 вложенный, 139
 конструктор, 393
 контроллер, 271
 концептуализация, 21
 концепция
 проработка, 210
 кортеж, 49
 кратность, 50, 51, 87
 куча, 374

Л—М

линия жизни, 186
 маркер деятельности, 175
 маршрут
 прослеживание, 236
 метаданные, 101
 метамодель, 466
 метод, 19, 47
 перегрузка, 373
 перекрытие, 373
 подмена, 373
 политика, 346
 реализация, 346
 сигнатура, 47
 много, 51
 множественное наследование,
 97, 373, 375, 376, 386, 388,
 401, 402
 моделирование
 генерация кода, 476
 изучение, 473
 ловушки, 466
 на месте, 470

моделирование (*продолжение*)
 обучение, 474
 организация персонала, 472
 оценка затрат, 477
 программные средства, 475
 сеансы, 468
 скрытое, 469
 управление, 465
 циклическое, 469
 модель, 23, 34
 аналитическая, 37, 219
 взаимодействия, 23, 37,
 161, 259
 приложения, 259
 виды, 465
 деятельности, 171
 для оценки продукта, 466
 классов, 23, 42
 предметной области, 220
 назначение, 34
 последовательности, 167
 предметной области, 21
 приложения, 21, 466
 проектная, 37
 состояний, 23, 37
 модель классов, 36
 итерационная
 разработка, 237
 навигация, 66
 приложения, 269
 проверка, 271
 уточнение, 361
 модель состояний
 предметной области, 242
 приложения, 272
 пример, 147
 мультимножество, 55

Н—П

набор обобщений, 62
 наведение мостов, 324
 навигация, 66
 наследование, 18
 в аналитической модели, 235
 множественное, 97
 подкласс, 18
 подмена, 63
 суперкласс, 18
 непрерывное
 преобразование, 307
 нормальная форма, 412
 обертка, 486
 область действия, 88
 обобщение, 59, 182
 применение, 62
 реализация, 384
 сигналов, 142
 уточнение, 362
 образец, 241, 290
 обучение, 206
 объект, 17, 42
 время жизни, 186
 идентификатор, 46, 379
 индивидуальность, 43
 обозначение, 44
 параллелизм, 144
 пассивный, 186
 поток, 190
 управляющий, 271
 объектная
 ориентированность, 17
 ограничение, 103
 использование, 105
 ООСУБД, 432, 433, 435
 операция, 18, 19, 43, 46
 по списку, 279
 распространение, 95
 оптимизация
 ассоциаций, 336
 выводимых атрибутов, 339
 индексирование, 337
 ориентированность
 объектная, 17
 основной ключ, 411
 оценка производительности, 287
 пакет, 106
 Java, 375
 группировка классов, 241
 пакетное преобразование, 306
 параллелизм, 143, 144
 выделение, 294
 синхронизация, 145
 параллельность
 неотъемлемая, 294
 переход, 121
 запуск, 121
 по завершении, 128
 перечисление, 86, 379
 плавательная дорожка, 189
 поведение, 43
 абстрагирование, 342
 воплощение, 340
 повторное использование,
 25, 441
 виды, 441
 наследование, 443
 планирование, 288
 хороший стиль, 441
 поддержка, 206
 подкласс, 18, 59
 подмена, 63
 подсистема, 291
 полиморфизм, 19, 373, 376,
 384, 387, 403
 полюс ассоциации, 53
 последовательность, 55
 диаграмма, 168

постановка задачи, 213
 поток объектов, 190
 поток управления, 295
 потомок, 61
 предметная область
 модель классов, 220
 предок, 61
 приложение
 модель классов, 269
 модель состояний, 272
 приоритеты проекта, 305
 программирование
 в большом, 448
 проект
 оптимизация, 336
 реорганизация, 335
 проектирование, 21
 алгоритмов, 328
 выбор алгоритмов, 328
 выбор структур данных, 330
 классов, 22, 324
 наведение мостов, 324
 проектировка
 приоритеты, 305
 производный элемент, 105
 прослеживание, 66
 анализ, 365
 процедура, 186
 процесс
 разработки, 202

Р—С

развертывание, 206
 раздел, 293
 разработка
 быстрое
 прототипирование, 457
 водопадная, 207, 457
 жизненный цикл, 206
 итерационная, 207, 456
 объектно-ориентированная,
 20, 21
 процесс, 202
 распространение, 95
 расширение, 181, 445
 реализация, 22, 205, 360
 ассоциаций, 363
 классов, 381
 моделирование, 360
 на объектно-
 ориентированном
 языке, 372
 структуры, 378

рекурсия, 333
 вниз, 333
 по механизму, 334
 по функциональности, 334
 реорганизация, 335
 классов и операций, 341
 репозиторий, 477
 РСУБД, 410—414, 420—425,
 428—435
 сборка мусора, 376, 396
 связь, 49
 сервис, 291
 сигнал, 117
 обобщение, 142
 отправка, 189
 получение, 189
 сигнатура, 47
 синергия, 26
 система
 архитектура, 21, 286
 выделение параллелизма, 294
 глобальные ресурсы, 300
 границные условия, 304
 изобретение, 209
 концептуализация, 203
 многоуровневая, 292
 определение границ, 260
 оценка
 производительности, 287
 пограничные классы, 270
 проектирование, 204, 286
 разбиение на
 подсистемы, 291
 раздел, 293
 распределение
 подсистем, 295
 реального времени, 310
 тестовая, 369
 хранение данных, 299
 ситуация гонок, 129
 скрытое моделирование, 469
 словарь данных, 224
 слой, 484
 см. *OCL*, 67
 событие, 116
 внешнее, 265
 времени, 118
 изменения, 118
 параллельное, 117
 поиск, 273
 сигнала, 117
 совместное использование, 25

согласованность
 сущностей, 346
 сокрытие информации, 345
 составляющая, 18
 состояние, 119
 вложенное, 139
 композиционное, 140
 разложение, 138
 сохраненная процедура, 429
 сторожевое условие, 121
 СУБД, 410, 411, 413, 423, 429,
 432, 434, 435
 суперкласс, 18, 59
 схема, 409
 сценарий, 167, 263
 выделение событий, 265
 особые ситуации, 265

Т—Э

тестирование, 205, 367, 368
 модульное, 368
 системное, 369
 тестовая система, 369
 тип
 объектный, 378
 управление
 внешнее, 301
 внутреннее, 301, 303
 параллельное, 303
 процедурами, 289
 процедурное, 302
 событийное, 302
 событиями, 289
 управление доступом, 381
 управление
 конфигурациями, 476
 управляющий объект, 271
 диаграмма состояний, 275
 упрощение операций, 279
 уровень
 системы, 292
 условие
 сторожевое, 121
 устойчивость, 446
 утверждение, 368
 фокус управления, 186
 хранилище данных, 299
 циклическое
 моделирование, 469
 экземпляр, 43
 класса, 18
 элемент
 производный, 105

Class Model Notation – Basic Concepts

Object:

objectName:ClassName

objectName:ClassName

attributeName = value

...

Class:

ClassName

ClassName

attribute

attribute : DataType[attMult]

attribute : DataType[attMult] = defaultValue

...

operation

operation (arg1:Name1, ...) : ResultType

...

Association Class:

Class1

Class2

AssocName

attribute

...

operation

...

Generalization (Inheritance):

Superclass



Superclass1

Superclass2

Package:

PackageName

Link:

object1:Class1

AssociationName

object2:Class2

Association:

Class1

AssociationName

assocEndNm1

assocEndNm2

Class2

Multiplicity of Associations:

1

Class

Exactly one

*

Class

Many (zero or more)

0..1

Class

Optional (zero or one)

1..*

Class

One or more

Ordered, Bag, Sequence:

{ordered}

Class

*

{bag}

Class

*

{sequence}

Class

*

Qualified Association:

Class1

qualifier

Class2

Enumeration:

«enumeration»

EnumName

enumValue1

enumValue2

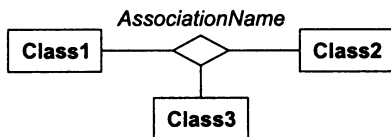
...

Comment:

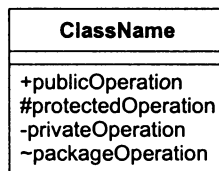
...informal text...

Class Model Notation – Advanced Concepts

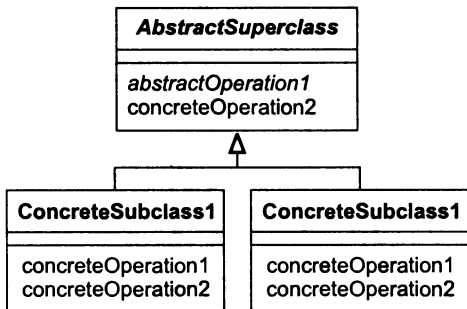
Ternary Association:



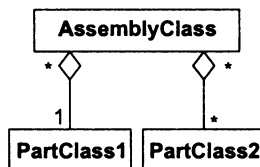
Visibility:



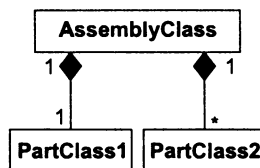
Abstract and Concrete Class:



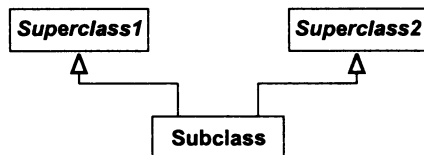
Aggregation:



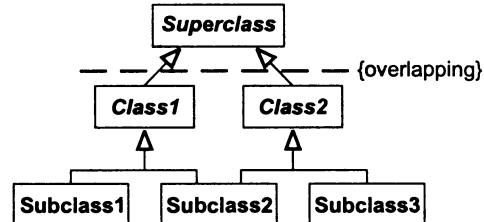
Composition:



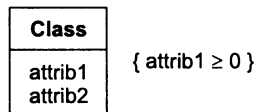
Multiple Inheritance, Disjoint:



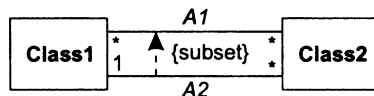
Multiple Inheritance, Overlapping:



Constraint on Objects:



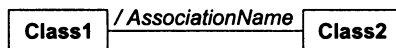
Constraint on Links:



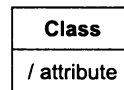
Derived Class:



Derived Association:

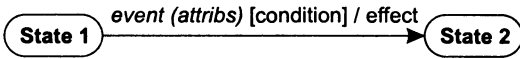


Derived Attribute:



State Model Notation

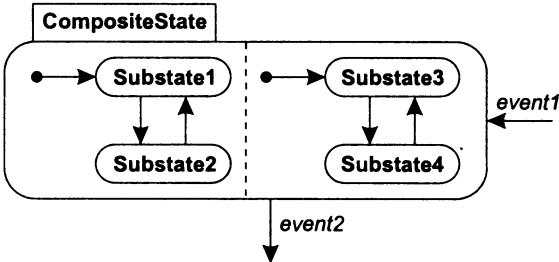
Event causes Transition between States:



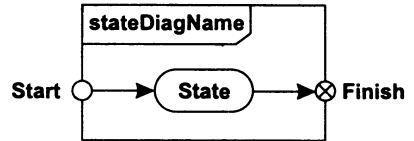
Initial and Final States:



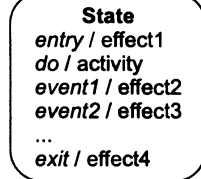
Event causes Transition between States:



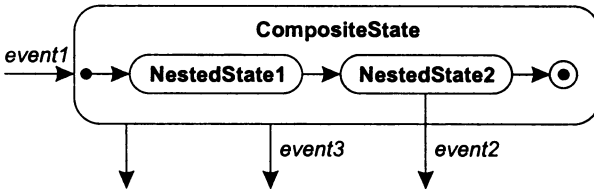
Entry and Exit Points:



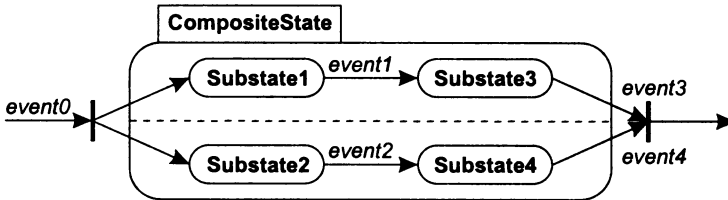
Activities while in a State:



Nested State:



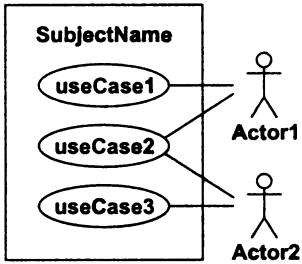
Splitting of control:



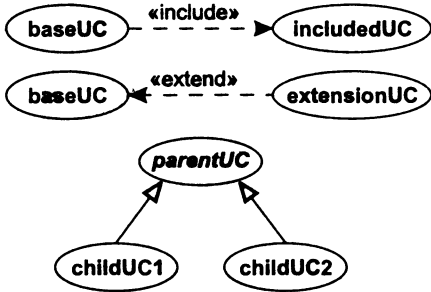
Synchronization of control:

Interaction Model Notation

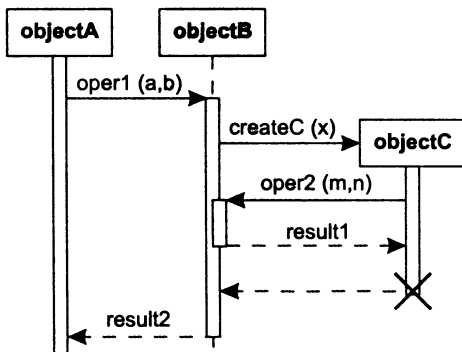
Use Case Diagram:



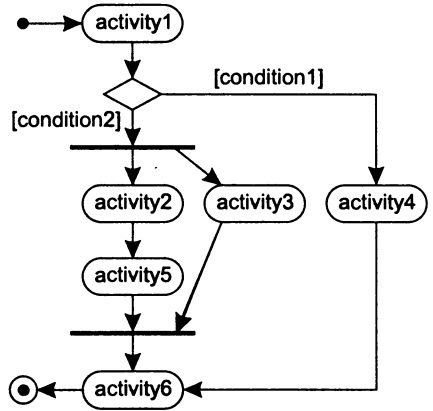
Use Case Relationships:



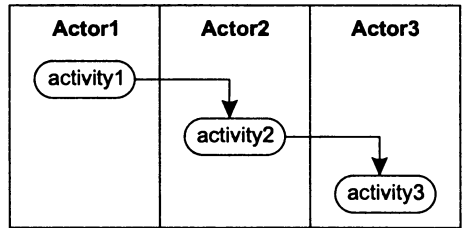
Sequence Diagram:



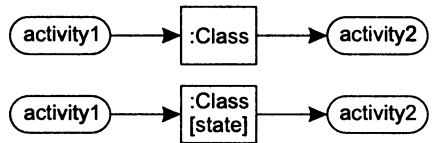
Activity Diagram:



Activity Diagram with Swimlanes:



Activity Diagram with Object Flows:





UML 2.0



Объектно-ориентированное моделирование и разработка

Перед вами поистине революционная книга, посвященная базовым принципам объектно-ориентированного мышления. Своей универсальностью она выгодно отличается от множества книг, описывающих характерные черты какого-нибудь одного языка программирования. Новое издание этого бестселлера обновлено в соответствии со стандартом UML 2.0. Авторы объясняют суть важнейших концепций объектно-ориентированного программирования, представляют способы реализации этих идей при разработке ПО с использованием языков C++ и Java, а также реляционных баз данных. В книге есть задания и множество советов, что делает ее очень практичной.

Особенности второго издания.

- Полностью обновленный текст, остающийся при этом в рамках стилистики, присущей первому изданию. В соответствии со стандартом UML 2.0 добавлены описания вариантов использования, диаграмм деятельности и диаграмм последовательности.
- Рассматривается более устойчивый процесс, предполагающий возможность итеративной разработки, моделирования имплементации и обратной инженерии.
- Множество упражнений, снабженных уникальным 200-страничным онлайн-методическим пособием для преподавателей.
- Особое внимание уделено практической и решению конкретных задач.

Д-р Майкл Блаха и д-р Джеймс Рамбо (совместно с Биллом Премерлани, Фредом Здди и Биллом Лоренсеном) являются создателями объектно-ориентированного метода, представленного в первом издании книги. Этот метод, чрезвычайно популярный в среде разработчиков, лег в основу Унифицированного языка моделирования (UML).

Тема: Программирование/UML

Уровень пользователя: опытный/эксперт

ПИТЕР®

197198, Санкт-Петербург, а/я 619
тел.: (812) 703-73-74, postbook@piter.com

61093, Харьков-93, а/я 9130
тел.: (057) 712-27-05, piter@kharkov.piter.com

www.piter.com — вся информация о книгах и веб-магазин

ISBN 5-469-00814-2



9 785469 008149