Константин Грибачев

Delphi
и Model Driven Architecture

Разработка приложений баз данных

- философия MDA;
- основы UML и OCL;
- объектное пространство;
- навигация по UML-модели;
- компоненты бизнес-уровня;
- OCL-запросы;
- подписка (subscribing);
- трансляция OCL в SQL;
- хранение данных в РСУБД и XML.

ПИТЕР

ББК 32.973.233-018
УДК 681.3.06
Г82

Грибачев К. Г.
Г82 **Delphi и Model Driven Architecture. Разработка приложений баз данных.** — СПб.: Питер, 2004. — 348 с.: ил.

ISBN 5-469-00185-7

Данная книга посвящена новейшей технологии разработки приложений баз данных в Delphi, основанной на концепции архитектуры, управляемой моделями (Model Driven Architecture — MDA). Читатель познакомится с идеологией MDA в целом, освоит базовые основы языка UML, познакомится с мощным и лаконичным диалектом языка OCL, благодаря которому гораздо легче и быстрее решаются задачи доступа к данным, чем при традиционной разработке с применением SQL-запросов. В книге подробно, с разбором множества конкретных примеров, описываются практические шаги по созданию MDA-приложений в Delphi. Читатель на практике убедится, как быстро и просто можно создавать сложные приложения, работающие практически с любой СУБД, при этом зачастую вообще не использ>я SQL и в ряде случаев даже не прибегая к написанию программного кода. Книга в первую очередь адресована читателям, знакомым с традиционными подходами создания приложений баз данных в Delphi, однако она может быть полезна и всем тем, кто желает познакомиться с принципиально новой технологией создания приложений в XXI веке — MDA.

ББК 32.973.233-018
УДК 681.3.06

ISBN 5-469-00185-7

Краткое содержание

Предисловие . . . 13
От автора . . . 15
Введение . . . 17

Часть I. Обзор MDA-архитектуры . . . 20
Глава 1. MDA-архитектура и язык UML . . . 21
Глава 2. Обзор Borland MDA . . . 47
Глава 3. Быстрый старт . . . 60
Глава 4. Модель приложения . . . 81
Глава 5. OCL — язык объектных ограничений . . . 104

Часть II. Создание MDA-приложений . . . 130
Глава 6. Объектное пространство . . . 131
Глава 7. Дескрипторы (Handles) . . . 143
Глава 8. Использование OCL . . . 160
Глава 9. Графический интерфейс . . . 179
Глава 10. Работа с уровнем данных . . . 205
Глава 11. Использование сторонних визуальных компонентов . . . 235
Глава 12. Генерация кода . . . 251
Глава 13. Механизм «подписки» . . . 272
Глава 14. Обзор дополнительных возможностей . . . 289
Глава 15. Продукты сторонних производителей . . . 295

Заключение . . . 302
Приложение А. Скрипт для транслитерации имен UML-модели . . . 303
Приложение Б. ECO — развитие Borland MDA для платформы Microsoft .NET . . . 310
Приложение В. Библиотека для работы с Object Space . . . 326
Приложение Г. Часто задаваемые вопросы . . . 338
Приложение Д. Интернет-источники . . . 343
Список литературы . . . 344
Алфавитный указатель . . . 345

Содержание

Предисловие . . . 13

От автора . . . 15

Как появилась эта книга . . . 15

Благодарности . . . 16

Введение . . . 17

Для кого написана книга . . . 18

Преодоление трудностей . . . 18

Обзор содержания . . . 18

От издательства . . . 19

Часть I. Обзор MDA-архитектуры . . . 20

Глава 1. MDA-архитектура и язык UML . . . 21

История . . . 21

Структура и состав . . . 21

Предпосылки появления MDA . . . 22

Модель приложения. Типы моделей . . . 23

Этапы разработки MDA-приложений . . . 24

Недостатки традиционного подхода . . . 25

Преимущества MDA . . . 26

Состояние и перспективы MDA . . . 27

Концепции реализации . . . 28

Возможные последствия внедрения MDA . . . 28

Унифицированный язык моделирования UML . . . 29

Общие сведения . . . 29

Бизнес-правила . . . 30

Диаграмма классов . . . 30

Классы . . . 31

Отношения . . . 32

Классы-ассоциации . . . 35

Пакеты . . . 36

UML-модель и схема БД . . . 37

UML-редакторы . . . 39

Основы работы в Rational Rose 39
Другие UML-редакторы 45
Резюме 45

Глава 2. Обзор Borland MDA 47
Что такое Borland MDA (BMDA) 47
История развития 49
Borland Enterprise Studio 49
Borland Delphi 7 Studio Architect 50
C#Builder Architect 51
Возможности и специфика 51
Преимущества для разработчиков 52
Последовательность изучения 54
Возможные трудности 54
Bold for Delphi — основа Borland MDA 55
Инсталляция и обзор инструментов 56
Резюме 58

Глава 3. Быстрый старт 60
Создание простого MDA-приложения 60
Создание бизнес-уровня 60
Создание модели приложения 61
Создание графического интерфейса 66
Создание уровня данных 68
Работа с приложением 69
Модификация модели приложения 73
Обсуждение результатов 76
Создание стандартных приложений 77
Резюме 79

Глава 4. Модель приложения 81
Роль модели в Borland MDA 81
Тег-параметры (tagged values) 83
Rational Rose как средство разработки моделей для Borland MDA 83
Настройка Rational Rose 84
Создание модели в Rational Rose и ее импорт в Borland MDA 86
Настройка тег-параметров модели в Rational Rose 92
Настройка тег-параметров модели во встроенном UML-редакторе 96
Экспорт модели из встроенного редактора в Rational Rose 99
Другие способы импорта-экспорта моделей 100
Инструменты встроенного редактора моделей 101
Средства настройки свойств модели 101
Средства настройки атрибутов класса 102
Настройка ролей ассоциаций 102
Резюме 103

Глава 5. OCL — язык объектных ограничений 104
Общие сведения 104
Роль OCL в Borland MDA 104
Модель для изучения 104
Доступ к классам и атрибутам из OCL 106
Базовые элементы и понятия 106
Типы данных 106
Операции сравнения в OCL 107
Типы возвращаемых выражений 107
Арифметические операторы 108
Коллекции 108
Навигация по модели 108
Операции над коллекциями 109
Навигация по коллекции 109
Выборка (фильтрация) и исключение 110
Операции с несколькими множествами 111
Сортировка элементов коллекции 112
Логические операции над коллекциями 113
Операция Collect 114
Вычисления над коллекциями 116
Работа с типами в OCL 117
Обработка типов для элементов 117
Обработка типов для коллекций 118
Операции логического выбора 118
Прочие операции 119
Операции над строками 119
Операции с датами и временем 120
Форматы дат и времени 120
Использование OCL 121
Задание выражений для вычисляемых атрибутов 121
Вычисляемые ассоциации 122
Формирование ограничений (Constraints) 125
Особенности диалекта OCL в среде Bold for Delphi 128
Расширяемость диалекта OCL в Bold 129
Резюме 129

Часть II. Создание MDA-приложений 130

Глава 6. Объектное пространство 131
Понятие Object Space 131
Состав и структура ОП 133
Класс TBoldElement 136
Свойства TBoldElement 136
Методы TBoldElement 137

Класс TBoldSystem 137
Свойства TBoldSystem 137
Методы TBoldSystem 138
Работа с объектным пространством 138
Программное управление атрибутами объектами ОП 139
Программное управление объектами ОП 141
Резюме 142

Глава 7. Дескрипторы (Handles) 143
Роль дескрипторов 143
Классификация дескрипторов 143
BoldSystemHandle 144
BoldSystemTypeInfoHandle 145
BoldExpressionHandle 146
BoldDerivedHandle 149
BoldVariableHandle и BoldOCLVariables 151
BoldListHandle 154
BoldCursorHandle 155
BoldReferenceHandle 158
BoldUnloaderHandle 158
Резюме 159

Глава 8. Использование OCL 160
Роль Object Constraint Language в Borland MDA 160
Условная модель 160
Создание приложения 161
Встроенный OCL-редактор 162
Формирование цепочек дескрипторов 165
Связь OCL и графического интерфейса 167
OCL и вычисляемые данные 168
Использование OCL в программе 173
Программная работа с классом TBoldElement и его наследниками с использованием OCL-вычислений 176
OCL-репозиторий 177
Резюме 178

Глава 9. Графический интерфейс 179
Особенности визуальных MDA-компонентов 180
BoldLabel 180
BoldEdit 182
BoldGrid 182
BoldSortingGrid 184
BoldComboBox 185
BoldListBox 187
BoldCheckBox 187

BoldPageControl 188
BoldTreeView 190
BoldImage 195
Рендереры 198
Автоформы 199
Общие сведения. Структура автоформы 199
Ограничения 201
Генерация автоформ 202
Управление отображением атрибутов 202
Дополнительные инструменты 203
BoldCaptionController 203
Резюме 204

Глава 10. Работа с уровнем данных 205
Функции уровня данных 205
Структура и состав компонентов 206
Работа с СУБД 207
Подключение уровня данных 207
Генерация схемы базы данных 210
Состав системных таблиц 212
Структура генерируемых таблиц 213
Использование адаптеров СУБД 214
Создание собственных адаптеров СУБД 215
Требования к СУБД 216
Состав программных модулей адаптера СУБД 217
Преимущества использования адаптеров 217
Использование BoldActions для работы с БД 218
Язык SQL в MDA-приложениях 220
Идеология применения SQL 220
Использование дескриптора BoldSQLHandle 222
Механизм OCL2SQL 226
Оптимизация работы в клиент-серверной архитектуре 228
Bold и «тонкие базы данных» 228
Работа с несколькими СУБД 229
Использование XML-документов 230
Сохранение данных в XML-файлах 230
Практический пример использования XML 231
Резюме 234

Глава 11. Использование сторонних визуальных компонентов 235
Средства управления внешними компонентами 236
BoldDataSet — шлюз для использования DB-компонентов 241
Использование механизма подписки и программного кода 245
Резюме 249

Глава 12. Генерация кода 251
Процедура генерации 252
Структура и состав генерируемых модулей 255
Практическое использование кода модели 263
Работа с классами и атрибутами 263
Работа с ассоциациями 264
Операции 268
Резюме 271

Глава 13. Механизм «подписки» 272
Описание и реализация 272
События и подписка 272
Основные классы и реализация 273
Программное использование подписки 274
Использование BoldPlaceableSubscriber 277
Программирование вычисляемых атрибутов 279
Обратно-вычисляемые атрибуты 283
Резюме 287

Глава 14. Обзор дополнительных возможностей 289
Регионы 289
Жизненный цикл связанных объектов 290
Трехзвенная архитектура 291
Удаленное подключение к БД посредством SOAP 291
Синхронизация объектных пространств 292
Сервер управления блокировками 292
Object Lending Library (OLLE) 292
Эволюция модели и БД 293
Многоязыковая поддержка 294
Средства отладки 294
Резюме 294

Глава 15. Продукты сторонних производителей 295
BoldExpress Studio 295
BoldGridPro 296
OCL Extensions 298
Bold TCP OSS 298
Bold SOAP Server (BSS) 299
BoldRave 299
deBold 299
pHGantTimePackage и phGrid_BA 300
Резюме 301

Заключение 302

Приложение А. Скрипт для транслитерации имен UML-модели 303
Особенности
Создание и установка: 304
Использование
Исходный текст скрипта 305
Приложение Б. ECO — развитие Borland MDA для платформы Microsoft .NET 310
О продукте C#Builder Architect 310
Основные возможности ECO 311
Создание простого ECO-приложения 311
Генерация шаблона 311
Разработка UML-модели 312
Формирование бизнес-уровня 315
Создание графического интерфейса 317
Автоформы 322
Создание уровня данных 323
Специфика ECO и отличия от Bold 323
Графический интерфейс 324
Бизнес-уровень и UML-моделирование 324
Уровень данных 324
Резюме 325
Приложение В. Библиотека для работы с Object Space 326
Назначение 326
Причины и цели создания 326
Описание 327
Основные процедуры и функции 327
Вспомогательные процедуры и функции 328
Примеры использования 329
Добавление и изменение объектов ОП 329
Поиск и связывание объектов 329
Предупреждения 330
Приложение Г. Часто задаваемые вопросы 338
Общие вопросы 338
СУБД 339
Компоненты и OCL 340
Приложение Д. Интернет-источники 343
Список литературы 344
Алфавитный указатель 345

Предисловие

Книга, которую вы держите в руках, не совсем характерна для российского книжного рынка. В ней вы не найдете банальностей, связанных с применением палитры компонентов и особенностями синтаксиса языка Pascal. Предназначена она широкому кругу пользователей Delphi, но посвящена не «всему понемногу», как большинство издаваемых нынче книг, а одной конкретной технологии — Borland MDA.

Сейчас многие компании используют UML-моделирование при проектировании приложений самого разного масштаба, а для их реализации часто применяют Delphi. Однако подчас модели, созданные с помощью средств UML-моделирования, таких как Rational Rose или Borland Together, остаются лишь иллюстрациями в проектной документации, а приложения создаются теми же способами, что и десять лет назад, — данные проектируются практически отдельно от UML-модели, клиентское приложение создается вручную, при этом любые изменения в постановке задачи (и даже исправление ошибок постановки или проектирования) влекут за собой дорогостоящие переделки клиентского и серверного кода — и вот уже проект имеет все шансы выйти за рамки сроков или бюджета...

Все мы знаем, что бизнес-процессы у самых успешных компаний меняются стремительно — и уж точно быстрее, чем разрабатываются приложения. Поэтому технологии разработки, позволяющие безболезненно вносить изменения в готовый или почти готовый проект на всех уровнях, включая и уровень требований, необходимы ныне как никогда. Borland MDA — это одна из таких технологий. Ее применение поможет значительно сократить время выполнения проектов, снизить затраты на их реализацию, избежать утомительных рутинных операций — а значит, позволит всем нам работать эффективнее и с большим интересом.

Несмотря на крайне высокую востребованность и присутствие в составе Delphi в течение уже без малого двух лет, технология Borland MDA продолжает оставаться практически неизвестной ни разработчикам, ни руководителям проектов, ни архитекторам приложений. Причиной этого является, с одной стороны, практи-

чески полное отсутствие публикаций и книг об этой технологии и у нас, и на Западе, а с другой стороны, ее «нетрадиционность». Применение этой технологии требует преодоления определенных барьеров, одним из которых, на мой взгляд, является присущий многим из нас консерватизм. Именно преодолению консервативного подхода к разработке приложений с базами данных может помочь данная книга. Будучи уникальным и практически единственным на сегодняшний день источником сведений о Borland MDA (не считая небольшого цикла статей, опубликованных ее автором в одном из компьютерных журналов), она написана доступным языком, идеально построена с точки зрения компоновки материала и охватывает все вопросы, связанные с данной технологией. Автор данной книги Константин Грибачев щедро делится с читателями своими знаниями и идеями, нередко необычными и уникальными, и, на мой взгляд, грех этим не воспользоваться.

Мне остается только пожелать автору дальнейших успехов, а всем нам — чтобы было больше таких книг и в России, и в других странах.

С наилучшими пожеланиями

Наталия Елманова,
руководитель проектов ЗАО «Резидент»,
ответственный редактор журнала «КомпьютерПресс».

От автора

Как появилась эта книга

Парадокс, но данная книга появилась в результате недостатка терпения и нежелания заниматься «рутиной» — именно таким качествам своего характера автор благодарен за то что в конце концов эта книга увидела свет. Объяснение довольно простое: после нескольких лет разработки приложений баз данных в Delphi автору наконец по-хорошему надоело из раза в раз повторять по сути одни и те же рутинные операции (создавать таблицы и поля баз данных, строить графический интерфейс и т. д.). Тем более что в современных условиях, когда жизнь и требования быстро меняются, нередко приходилось заниматься основательной переделкой уже практически отлаженного приложения, что вызывало понятные и нехорошие чувства и мысли. И больше года назад, когда терпение истощилось окончательно, автор начал поиски некоего универсального продукта, который бы «все делал сам». Конечно же, такого продукта автор не нашел. Но «волшебный инструмент» найти все-таки удалось — им оказался программный продукт Bold for Delphi малоизвестной шведской компании BoldSoft.

Я, конечно, не сразу осознал это «волшебство». Из отзывов компаний, применяющих Bold for Delphi, следовало, что он якобы в 5-10 раз сокращает сроки разработки. Мне это напомнило телевизионную рекламу. Но попробовать-то надо, раз уж потратил время... В результате сейчас я могу сказать совершенно ответственно: обманывали авторы отзывов. Не в 5 раз Bold сокращает время, а гораздо больше. Работу, на которую раньше уходили месяцы, теперь можно было делать за считанные дни. Это стало очевидно примерно через пару недель использования Bold for Delphi. И тогда у меня возник всего один закономерный вопрос: почему же такая уважаемая и любимая мною компания, как Borland, до сих пор «не взяла его себе», неужели они не понимают явных преимуществ такого приобретения? Надо сказать, что Borland «не подвел», и еще через пару недель компания BoldSoft была им приобретена, в результате чего уникальный программный продукт Bold

for Delphi стал принадлежать компании Borland. И примерно через месяц Bold for Delphi официально вошел в состав версии Borland Delphi 7 Studio Architect. Что еще можно сказать об инструменте Bold for Delphi, чтобы это опять не походило на рекламу? То, что он практически неизвестен широкому кругу наших разработчиков. То, что литературы по нему нет никакой. То что, по сути, единственным источником технической информации по этому продукту являлись интернет-конференции, где автору пришлось провести немало ночных часов. И, наконец, то, что на его базе уже созданы новейшие средства быстрой разработки приложений, вошедшие в последние продукты компании Borland. А что касается «волшебства» — в упомянутых интернет-конференциях употребляется по этому поводу словосочетание — «Bold magic». И в самом деле, часто при работе с этим продуктом возникает ощущение присутствия чуда. И никто из разработчиков, применивших его на практике хотя бы один раз, уже не возвращается к традиционным «старым» методам. Потому что второе значение английского слова «Bold» — это, оказывается, «смелый». И он действительно очень «смелый» — он заставляет в корне пересмотреть всю привычную традиционную технологию разработки приложений баз данных. Все кажется поначалу непривычным, непонятным, но зато потом можно исключительно быстро и «приятно» создавать сложные приложения. У автора этой книги на освоение продукта Bold for Delphi до минимального уровня ушел не один месяц. Но окружающая нас жизнь ускоряется. И мне бы искренне хотелось, чтобы данная книга помогла своим читателям сократить этот срок до нескольких дней, так же, как сокращает время разработки «смелый» и уникальный программный продукт Bold for Delphi.

Благодарности

Автор выражает свою благодарность издательству «Питер» и лично руководителю данного проекта Анатолию Адаменко за поддержку, терпение и взаимопонимание в процессе работы над книгой. Также автор благодарен редактору Юрию Штенгелю за качество и оперативность проделанной им работы. Особую благодарность хочется выразить Наталии Елмановой, благодаря поддержке и советам которой был опубликован цикл статей в журнале «Компьютер Пресс», вследствие чего вероятность появления этой книги сильно повысилась. Автор также благодарит московское представительство компании Borland и лично Сергея Орлика за предоставленное программное обеспечение. И, конечно, я очень благодарен моей семье за понимание и бесконечное терпение, проявленное во время работы над этой книгой.

Константин Грибачев

Введение

В книге освещаются вопросы практического создания приложений, построенных на базе архитектуры MDA (Model Driven Architecture). Основным принципом этой архитектуры является наличие *модели,* которая определяет состав, структуру и поведение приложения. Концепция MDA-архитектуры в последние годы активно развивается мировым сообществом IT-компаний, входящих в состав консорциума OMG (Object Management Group). Внедрение MDA на практике обеспечит качественно более высокий платформенно-независимый уровень разработки, устранение рутинных операций по ручному созданию программного кода и структуры базы данных и даст ряд других преимуществ, которые будут рассмотрены в этой книге. В качестве конкретного инструмента MDA в этой книге описывается программный продукт Bold for Delphi, входящий в состав Borland Delphi 7 Studio Architect. По мнению автора, версия Delphi Architect совершенно незаслуженно обойдена вниманием компьютерной литературы, поскольку ее применение позволяет на практике кардинально сократить время разработки приложений баз данных, значительно снизить объем рутинных операций, обеспечить безболезненные переводы приложения для работы с другими СУБД. Во многих случаях технология Borland MDA позволяет обойтись и без использования языка SQL, применив вместо него мощный и гибкий диалект языка OCL (Object Constraint Language). Существует мнение, что MDA-технология в основном предназначена для проектирования и создания больших корпоративных информационных систем. Безусловно, это так. Но неоспоримо и то, что и для средних и малых приложений применение рассматриваемого в этой книге инструмента Bold for Delphi является не менее эффективным. При его использовании небольшие офисные локальные приложения баз данных могут быть созданы буквально за считанные дни, а впоследствии, при необходимости, с минимальными усилиями могут быть преобразованы в серьезные клиент-серверные системы для работы практически с любыми серверами СУБД. В настоящее время фактически отсутствует литература по MDA-архитектуре вооб-

ще и по продукту Bold for Delphi в частности, и автор надеется, что данная книга хотя бы отчасти заполнит эту нишу.

Для кого написана книга

Для всех читателей-разработчиков, интересующихся новейшими технологиями создания приложений баз данных и вопросами практического применения этих технологий. Конечно, в первую очередь книга адресована всем разработчикам приложений в Delphi. При этом практический опыт создания приложений баз данных желателен, но не обязателен. Далее станет ясно, что в ряде случаев при разработке MDA-приложений с использованием Bold for Delphi, вообще говоря, не обязательно быть специалистом в области реляционных баз данных, так как разработка ведется на другом, более высоком уровне. Этим объясняется, в частности, и достигаемая очень высокая скорость разработки. На практике при знакомстве с продуктом Bold for Delphi случаются ситуации, когда опытным разработчикам приложений баз данных труднее приступить к освоению этого инструмента, чем начинающим. Это объясняется смещением акцента с уровня СУБД на несколько непривычный *бизнес-уровень,* то есть уровень управления.

Преодоление трудностей

Принципы разработки MDA-приложений поначалу кажутся непривычными, заставляя изменить мировоззрение разработчика. Кроме того, создание таких приложений требует и новых знаний — в частности, знания основ языка унифицированного моделирования UML. Именно по этой причине в книге содержатся краткое изложение основ UML и описание работы с UML-редакторами. Необходимо сказать, что освоение MDA-архитектуры потребует определенных усилий от читателя. Однако это обстоятельство не должно быть помехой, ведь результатом освоения описываемых в книге инструментов будет совершенно другое качество работы. Для облегчения понимания материала в данную книг}' включено достаточное количество иллюстрированных конкретных примеров, показывающих, как и что должно делаться в различных ситуациях. Также сделана попытка продемонстрировать разные методы решения одной и той же задачи для лучшего понимания основных принципов практической работы.

Обзор содержания

В первой главе книги представлен общий обзор MDA-архитектуры, ее место и роль, а также проведены некоторые сравнения MDA-подходов с традиционной идеологией и практикой разработки. Вторая глава также является обзорной и показывает истоки и историю развития технологии Borland MDA. Главу 3 рекомендуется читать за компьютером, поскольку она впервые дает пошаговое описание практического алгоритма разработки на примере простого приложения. В последующих главах первой части описаны базовые понятия - модель, UML и OCL. К этим главам (особенно к главе 5 о языке OCL) можно возвращаться и в дальнейшем, используя

их как справочный материал. Вторая часть книги посвящена собственно продукту Bold for Delphi и последовательно описывает все составные части и инструменты с многочисленными примерами практического применения. В главе 15 дан обзор программных продуктов сторонних производителей, созданных специально для использования совместно с Bold for Delphi. В приложениях А и В содержатся дополнительные материалы, такие как исходные тексты скриптов и программной библиотеки, использование которых может быть полезно при практической работе с Bold for Delphi. Приложение Б содержит обзор возможностей новейшей реализации технологии Borland MDA — ECO. Ответы па часто задаваемые вопросы по продукту Bold for Delphi содержатся в приложении Г. И, наконец, в приложении Д содержится список интернет-ссылок на источники различной информации по продукту Bold for Delphi.

От издательства

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

Мы будем рады узнать ваше мнение!

На веб-сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

Обзор MDA-архитектуры

MDA-архитектура и язык UML

В этой главе, являющейся, по сути, вводной, даются основные знания об архитектуре MDA и ее базовом инструменте — языке UML.

История

Model Driven Architecture (MDA) дословно переводится как «архитектура, управляемая моделью». История ее разработки насчитывает всего несколько лет. Основным разработчиком MDA является консорциум OMG (Object Management Group), его официальный сайт в Интернете находится по адресу http://www.omg.org. В настоящее время в консорциум OMG входит более 800 компаний — производителей программного и аппаратного обеспечения. Главной задачей OMG является разработка стандартов и спецификаций, регламентирующих применение новых информационных технологий на различных аппаратных и программных платформах. Широко известны такие продукты и технологии, как UML и CORBA, в разработке которых активно участвует OMG.

Сейчас основное внимание консорциума OMG сосредоточено на развитии технологии MDA.

Архитектура MDA предлагает новый интегральный подход к созданию многоплатформенных приложений. В этой главе мы остановимся лишь на главных моментах, необходимых для понимания дальнейшего материала. Более полное изложение концепции MDA можно найти в [1].

Структура и состав

Архитектура **MDA** базируется на трех основных «китах», или элементах:

- UML (Unified Modelling Language) — унифицированный язык моделирования;

- MOF (MetaObject Facility) - абстрактный язык для описания информации о моделях (язык описания метамоделей);
- CWM (Common Warehouse Metamodel) - общий стандарт описания информационных взаимодействий между хранилищами данных.

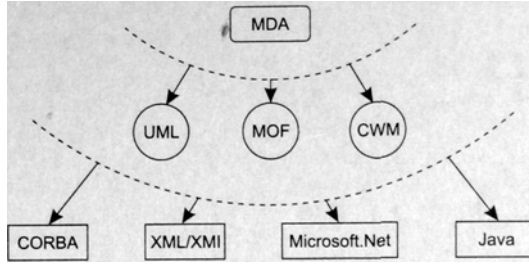


Рис. 1.1. Схема взаимодействия MDA с программными технологиями

Структура взаимодействия MDA с различными технологиями разработки программного обеспечения представлена на рис. 1.1. На центральном уровне структуры находится собственно MDA, которая «разворачивается» наружу посредством второго уровня, содержащего вышеперечисленные базовые составляющие — UML, MOF и CWM, — и, наконец, на третьем, внешнем уровне представлены некоторые из широко известных в настоящее время программных платформ разработки: CORBA, XML, .NET, JAVA. Отметим, что на этом внешнем уровне, по замыслу OMG, могут и должны быть размещены, в принципе, и все возможные будущие технологии разработки. Этим подчеркивается тот факт, что OMG считает архитектуру MDA не просто новой технологией, а скорее «метатехнологией» создания приложений. Последняя отныне будет и единственно актуальной — вне зависимости от развития и появления новых средств разработки, которые MDA уже «заранее интегрировала» в себя.

Предпосылки появления MDA

В последнее десятилетие наблюдается процесс активной разработки различных информационных технологий уровня предприятия, таких, как перечисленные выше программные платформы типа Microsoft .NET, CORBA, JAVA и т. д. Каждая из подобных технологий является достаточно сложной и объемной программной системой, содержащей множество взаимно увязанных составляющих, интерфейсов, объединенных массой специальных правил и ограничений. И хотя каждая из этих технологий, как правило, претендует на комплексное решение всех проблем построения информационных систем, это, как показывает практика, с одной стороны, не мешает всем им существовать одновременно, а с другой — не гарантирует, что в ближайшее время не появятся новые технологии или платформы разработки. В настоящее время уже очевидно, что наличие довольно большого количества подобных платформ и активизация процесса их разработки отнюдь не способствуют

ускорению их практического внедрения. Главная причина этого явления — отсутствие единого механизма интеграции. Руководителям крупных организаций и предприятий сначала приходится тратить значительные материальные средства для приобретения и разработки информационных систем, их сопровождения, обучения персонала и т. д. А впоследствии, в случае перехода на другую платформу, приходится, по сути, все повторять заново, опять тратя при этом материальные ресурсы. На крупных предприятиях нередко возникает необходимость внедрения и одновременной эксплуатации нескольких информационных систем, построенных с использованием различных платформ разработки, которые при этом должны взаимодействовать между собой. В этом случае требуемые затраты на высококвалифицированный персонал могут легко выйти за рамки допустимых, поскольку специалисты, владеющие одновременно несколькими подобными технологиями, встречаются весьма редко, а их «цена» растет отнюдь не пропорционально количеству освоенных программных платформ. Перечислениеподобных ситуаций можно продолжить, но уже из сказанного очевидно, что необходимо каким-то образом решать проблемы интеграции, чтобы унифицировать все имеющиеся ныне и предполагаемые в будущем разнообразные технологии разработки программного обеспечения информационных систем. Задачу эту можно решать по крайней мере двумя способами. Первый способ — внешняя унификация информационных интерфейсов. Одним из распространенных средств реализации такого подхода в настоящее время является язык XML. Этот путь облегчает задачу внешней интеграции, но не обеспечивает решения проблемы повторного проектирования систем при переходе на другую платформу. Консорциум OMG предлагает идти другим путем, интегрируя платформы «изнутри», то есть путем создания платформенно-независимых моделей.

Модель приложения. Типы моделей

В основе MDA лежит идея выделения этапа разработки логики функционирования приложения (бизнес-логики) в качестве самостоятельного и обязательного. То есть, согласно концепции MDA, разработка приложения должна начинаться с этапа создания *модели приложения,* которая определяет состав, структуру и поведение будущего программного продукта.

Модель представляет собой совокупность элементов и их связей, отражающих облик будущего программного продукта, но при этом важно, что этот облик формулируется в достаточно абстрактном виде — то есть без привязки к конкретным языкам или средам программирования. Модель скорее отражает предметную область, в рамках которой ведется проектирование, и содержит абстрактное описание сущностей: объектов, свойств, методов и связей. В данном случае термин «абстрактное» необходимо понимать как «не зависящее от платформы разработки». Сами объекты при этом могут (и должны) быть описаны достаточно подробно. Например, модель приложения, предназначенного для управления кадровым составом фирмы, может содержать объекты типа «сотрудник», «подразделение» и т. д., каждый из которых при необходимости может быть подробно описан несколькими десятками атрибутов (свойств или параметров). Но при этом такое подробное описание создается не на языке программирования, а посредством языка унифи-

UML (Unified Modelling Language). Далее в этой главе мы подробно познакомимся с основами языка UML, а сейчас лишь отметим, что UML — это платформенно-независимый язык, специально созданный (при непосредственном участии консорциума OMG) для применения в системах объектно-ориентированного анализа и проектирования (ООАП). Сразу необходимо сделать важное уточнение: до сих пор речь шла о платформенно-независимых моделях, для которых в MDA используется термин PIM (Platform Independent Model). Естественно, что ставя своей целью создание приложений, функционирующих на разнообразных платформах, MDA не может ограничиться только PIM-моделями, то есть необходимо каким-то образом «адаптировать» абстрактные PIM-модели к конкретным средам и платформам разработки. Поэтому кроме PIM-моделей MDA-архитектура содержит понятие платформенно-зависимых моделей — PSM (Platform Specific Model). Эти модели как раз и выполняют роль своеобразных «адаптеров» или «драйверов», обеспечивая корректное отображение абстрактной PIM-модели на конкретную программную среду. В данном случае под программной средой, или платформой, подразумеваются конкретные технологии создания приложений, например: Microsoft .NET, Sun One, J2EE, CORBA и т. д. Необходимо сразу отметить, что в концепции MDA предусматриваются механизмы, реализующие взаимодействие между PSM-моделями, тем самым обеспечивая не только многоплатформенность, но и информационную интеграцию приложений, «генерируемых» для различных платформ разработки.

Этапы разработки MDA-приложений

Циклограмма разработки MDA-приложений в общем виде включает три основных этапа (рис. 1.2). На первом этапе разработчик, исходя из поставленной перед ним задачи и знаний о предметной области, формирует платформенно-независимую PIM-модель. При ее создании он полностью абстрагируется от особенностей конкретных программных или аппаратных средств. На втором этапе создаются одна или несколько платформенно-зависимых моделей PSM, которые являются своеобразными «адаптерами» или «драйверами», обеспечивающими интеграцию PIM с одной или несколькими технологиями разработки программных ПРОДУКТОВ

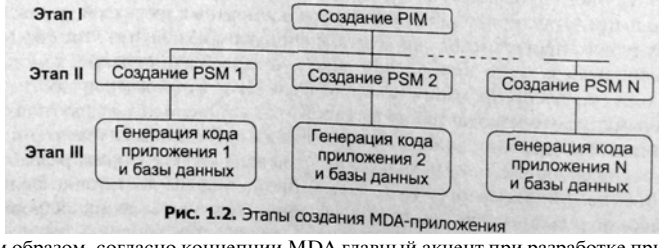


Рис. 1.2. Этапы создания MDA-приложения

Таким образом, согласно концепции MDA главный акцент при разработке приложений переносится с собственно этапа программирования на этап создания модели. При этом, создав один раз модель, разработчик получает принципиальную возможность генерации приложений для разных аппаратных и программных платформ

Недостатки традиционного подхода

Ознакомившись кратко с основными идеями, заложенными в MDA-архитектуру, давайте вернемся к привычному подходу разработки приложений, и попытаемся сформулировать его недостатки и ограничения. На рис. 1.3 в самом общем виде показаны традиционные этапы разработки.

Рис. 1.3. Этапы традиционной разработки

На первом этапе создается проект приложения на основе требований к разработке. Он может быть описан словесно или представлен в графическом виде с отображением структуры и состава будущего приложения. Как правило, при обычном проектировании разработчик уже имеет представление, на какой платформе будет создаваться и функционировать будущее приложение, поэтому проект может включать специфические для выбранной платформы элементы. Например, проектируя иерархию классов приложения при программной реализации на языке C++, разработчик вправе использовать и заложить на этом этапе возможность множественного наследования, а для разработки на Delphi — применять другие методы. На втором этапе происходит собственно программирование приложения на выбранном языке. Наконец, третий этап включает проведение отладки и тестов. Тесты в данном случае подразумевают и демонстрацию приложения заказчику. На практике в подавляющем большинстве случаев разработка на этом не заканчивается вследствие того, что третий этап выявляет некоторое количество замечаний. Замечания могут быть как объективными, то есть вызванными некорректно реализованным этапом программирования, так и субъективными — в этом случае заказчик или менеджер уточняют постановку задачи. Поэтому процесс разработки достаточно сложного приложения всегда носит итерационный характер, то есть включает в себя возврат к предыдущим этапам. И вот здесь в практике традиционной разработки наблюдается следующее устойчивое явление — разработчики «забывают» возвращаться на этап I (пунктирная линия на рис. 1.3). Вместо пересмотра проекта приложения зачастую происходит просто модификация кода, то есть повторение этапа II (сплошная линия на рис. 1.3). К чему это приводит? Для относительно простых приложений ничего страшного, в принципе, не происходит, код приложения просто модифицируется, количество итерации в этом случае невелико, объем кода также невелик. Совершенно другая ситуация с крупными программными проектами, особенно при их коллективной разработке. Происходит «отрыв» проекта от фактически разработанного программного обеспечения, то есть проект с каждой итерацией все больше «расходится» с собственной

реализацией. В результате во многих случаях разработка становится неуправляемой модификация кода - все более затруднительной, так как, по сути, отсутствует документированность в виде проекта. При наличии коллектива программистов ситуация усугубляется по той же причине - отсутствие документа-проекта приводит к увеличению количества ошибок, нарушению связей между отдельными частями программного продукта. Стоит ли говорить о том, что к подобной разработке практически невозможно вернуться через какое-то время для ее доработки или создания на ее базе нового продукта. К сожалению, описанная ситуация не так уж редко встречается на практике.

Другое ограничение при традиционной разработке — необходимость перепрограммирования приложения «вручную» или его перепроектирования при переходе на другую платформу. Несмотря на некоторые имеющиеся средства многоплатформенной разработки, нельзя утверждать на 100 %, что при создании больших программных систем полностью отсутствует необходимость «ручной» доработки программного кода.

Нельзя не упомянуть и о такой довольно часто встречающейся на практике проблеме, как отсутствие ясного «взаимопонимания» между коллективом разработчиков и заказчиком и плохая управляемость разработкой. В силу изложенных выше причин, когда фактически отсутствует описание приложения, менеджер и заказчик полностью теряют контроль над состоянием разработки, так как выполнение этой функции потребовало бы от них владения языками программирования и других специфических знаний, что на практике встречается редко.

Также очевидно, что создание больших программных систем традиционными методами, включающими этап «ручного» кодирования, требует достаточно больших временных затрат, и разработка таких приложений растягивается на годы.

Мы рассмотрели только некоторые ограничения и недостатки традиционного подхода к разработке приложений. Настало время узнать, что же в этом плане предлагает MDA.

Преимущества MDA

Циклограмма создания MDA-приложений (см. рис. 1.2) также содержит потенциальную возможность итерационной разработки. Однако в этом случае разработчик возвращается на этап I и при необходимости корректирует PIM-модель приложения. А поскольку (по крайней мере, такие намерения декларирует концепция MDA) PSM-модель и генератор кода в идеале должны быть полностью отработаны и функционировать «в автоматическом режиме», постольку все изменения PIM должны реализоваться в измененном коде приложения без искажений. Здесь уместно провести некоторую аналогию с прикладной программой и драйверами операционной системы: если прикладная программа использует некий стандартный драйвер, и он штатно функционирует, то при корректном изменении прикладного кода не произойдет никаких неожиданностей в работе программы в целом. Аналогию можно и несколько расширить: если мы заменим имеющийся драйвер на драйвер другого устройства (например, модернизировав в персональном компьютере звуковую карту), а прикладную программу оставим без изменений, то наше приложение должно по-прежнему штатно функционировать, взаимодействуя

уже с другим устройством. То есть, создав один раз PIM-модель и заменяя потом «драйверы» — PSM, — мы добьемся функционирования нашего приложения на совершенно разных платформах. Уже из сказанного очевидно, какие преимущества дает архитектура MDA. К этому можно добавить еще ряд полезных качеств нового подхода.

- Кардинальное повышение производительности разработки. По сути, при использовании MDA-архитектуры вся разработка сводится к корректному формированию PIM-моделей, устраняется этап «ручного» программирования.
- Документированность и легкость сопровождения. PIM-модель в MDA играет роль как проекта, так и основного документа — описания приложения. В достаточно компактном виде PIM-модель содержит все сведения о программе.
- Централизация логики функционирования. В отличие от традиционного подхода, где логика работы приложения «разбросана» по программному коду, в MDA она сосредоточена в одном месте — в PIM-модели. Приложение изменяет свое поведение при изменении PIM-модели.
- Облегчение доступности и управляемости разработки. С точки зрения заказчика или менеджера наличие платформенно-независимой PIM-модели резко облегчает понимание проекта в целом и управление разработкой. Это объясняется тем, что PIM-модель «не привязана» к специфическим особенностям сред программирования и по этой причине не содержит сложных или непонятных заказчику/менеджеру элементов и конструкций. Как мы увидим в дальнейшем, UML-диаграммы, представленные в графическом виде, являются достаточно наглядными и по существу не требуют знания программирования или теоретических основ разработки реляционных баз данных.

Состояние и перспективы MDA

Естественно, что создание такой технологии, как MDA, требует достаточно длительного времени. В конце 2001 г. консорциум OMG выпустил документ «Model Driven Architecture — A Technical Perspective», являющийся первой предварительной спецификацией MDA. На очереди следующие спецификации и документы.

MDA не является, по замыслу OMG, конкурентом какой-либо из существующих технологий создания приложений (CORBA, .NET, J2EE и т. д.). MDA находится на более высоком уровне обобщения процесса разработки, позволяя на этапе создания PIM-модели абстрагироваться от этих платформ, на следующем этапе выбрать одну или несколько платформ разработки и создать соответствующий набор PSM-моделей и, наконец, на этапе генерации кода получить приложение, функционирующее на этих платформах. И как, вероятно, справедливо полагает OMG, этот подход будет работать не только для существующих в настоящее время технологий разработки, но и для любых создаваемых в будущем, путем построения для них соответствующих адаптеров — PSM-моделей.

Если намерения OMG будут реализованы, то в будущем сценарий создания приложений может выглядеть примерно так: создается платформенно-независимая модель, выбирается один или несколько готовых «адаптеров-платформ», далее запускается некоторый программный «MDA-генератор» — и на выходе получается приложение базы данных с готовым графическим интерфейсом. При необходимости изменения вносятся в модель, и процедура генерации повторяется.

Сейчас вышесказанное может вызывать здоровый скептицизм, однако, судя по серьезности намерений OMG и заслуженному авторитету этой организации, а также учитывая, что реализация и внедрение MDA является сейчас одним из ее важнейших стратегических проектов, эти планы рано или поздно будут реализованы.

Концепции реализации

Необходимо подчеркнуть, что на настоящем этапе концепция MDA не является полностью проработанной: более того, существует несколько своеобразных «конкурирующих» подходов к ее реализации. Из них можно выделить два основных — это, условно выражаясь, подходы «интерпретатора» и «генератора». В чем их суть и различие?

В концепции «интерпретатора» основное внимание уделяется «внедрению» PIM-модели в исполняемый файл приложения и доступу к ней во время выполнения приложения. При этом приложение как бы «знает» свою модель на этапе выполнения и функционирует в соответствии с этой моделью, «интерпретируя» ее во время выполнения. Таким образом, например, появляется принципиальная возможность во время выполнения приложения узнать состояние любого элемента модели, или даже изменить это состояние. При этом подходе можно обойтись даже без генерации кода, содержащего описание модели, так как модель хранится в специальном формате внутри исполняемого файла.

Напротив, в концепции «генератора» все внимание сосредоточено на как можно более полной и универсальной реализации автоматической генерации кода приложения, с использованием оптимизации кода по различным параметрам: скорости, размеру и т. д. Этот подход несколько больше схож с CASE-системами.

Трудно сейчас сказать, какой из описанных подходов в результате «одержит победу». Забегая вперед, можно только отметить, что описываемый в этой книге программный продукт Bold for Delphi достаточно гармонично сочетает в себе преимущества обеих концепций.

Возможные последствия внедрения MDA

Легко видеть, что само понятие «разработчик программного обеспечения» может при внедрении MDA довольно сильно видоизмениться. Со смещением акцента на создание модели разработкой приложений будут заниматься не столько программисты, сколько специалисты, владеющие описываемой предметной областью. Возможно, что также в какой-то степени «пострадает» традиционное деление специалистов на разработчиков баз данных и разработчиков приложений баз данных. Как будет показано в следующих главах, уже сейчас возможно при разработке MDA-приложений практически полностью абстрагироваться от знания конкретной

СУБД; более того, во многих случаях нет необходимости и использовать язык SQL, поскольку рассматриваемые в этой книге инструменты MDA предоставляют возможность работать на более «высоком» уровне (бизнес-уровне), где становится абсолютно не важным знание конкретной структурной схемы базы данных или состава полей ее таблиц.

Однако программисты-разработчики вряд ли останутся без работы, так как, с одной стороны, создание MDA-инструментария само по себе является чрезвычайно интересной, сложной и объемной задачей для них. А с другой стороны, внедрение MDA уже сейчас избавляет и самих программистов от рутинной работы, передавая большую ее часть искусственному программному интеллекту — инструментам реализации MDA.

Унифицированный язык моделирования UML

Архитектура MDA ставит во главу угла модель приложения и поэтому непосредственно связана с языком, на котором такие модели создаются, — языком унифицированного моделирования UML

Общие сведения

Язык UML (Unified Modelling Language) является основным инструментом MDA для создания модели приложения. Обладая довольно выразительными графическими возможностями, UML является одновременно и средством описания, и средством документирования разработки.

В настоящее время имеется довольно много литературы, описывающей язык UML (см., например, [2]). Учитывая важность UML для понимания и использования MDA, мы в этой главе кратко остановимся на его основных характеристиках и концепциях.

Причиной появления UML стала необходимость в унификации разнообразных подходов к описанию моделей бизнес-приложений. В последнем десятилетии прошлого века появилось несколько десятков вариантов инструментария для создания подобных моделей, все они были не согласованы между собой, что мешало разработке CASE-средств и приводило к трудностям их внедрения. CASE-средства (Computer Aided Software Engineering — дословно «разработка программного обеспечения с помощью компьютера») в то время использовались в основном в качестве универсальных графических инструментов для визуального проектирования реляционных баз данных с возможностями последующей автоматической генерации БД.

У истоков разработки языка Unified Modelling Language (UML) стоит компания Rational Software, автор одного из первых CASE-средств — Rational Rose. В 1995 году консорциум OMG включается в работу по стандартизации UML, к разработке языка активно подключаются и другие компании, и после выхода нескольких промежуточных версий в 1997 году появляется версия UML 1.0.

Сегодня последней стандартизованной OMG версией является UML 1.4, на заключительном этапе находится разработка версии 2.0. Развитие UML в настоящее время координируется OMG, который считает разработку и продвижение этого языка одним из своих стратегических направлений.

Можно выделить следующие характерные черты UML.

- UML является языком визуального моделирования, то есть обеспечивает наглядное графическое представление модели в виде одной или нескольких схем.
- UML не является языком программирования и не содержит алгоритмов и операторов в обычном смысле, он в первую очередь является средством описания.
- UML, являясь платформенно-независимым языком, абстрагируется от специфики конкретных языков программирования и средств разработки.

Язык UML разрабатывался как универсальное средство объектно-ориентированного проектирования сложных систем, имеющее наглядный графический интерфейс. Одновременно с этим ставились задачи использования UML и в качестве удобного инструмента документирования разработок.

UML базируется на объектно-ориентированном подходе и содержит *диаграмму классов* для описания структуры и состава модели. Диаграмма классов является основой для формирования модели приложения и играет важнейшую роль при разработке MDA-приложений.

Бизнес-правила

Тщательно продуманная диаграмма классов содержит основной объем необходимой информации о *бизнес-правилах,* в значительной степени определяющих корректное функционирование будущего приложения.

Термин «бизнес-правила» здесь и в дальнейшем означает условия и ограничения, накладываемые моделью на всю совокупность понятий моделируемого приложением окружающего мира. Любое бизнес-правило можно сформулировать и на естественном языке. Например, предложения типа «каждый сотрудник работает только в одной организации» или «книга должна быть издана хотя бы одним издательством» представляют собой некоторые бизнес-правила. Однако очевидно, что использование естественного языка в проектировании информационных систем приводит к ряду сложностей в силу хотя бы его неоднозначности и «нестрогости». При создании UML были предприняты значительные усилия с целью предоставления максимальной гибкости разработчику, формулирующему бизнес-правила, при условии существования рамок формальных языков. Этим объясняется и появление языка объектных ограничений OCL (Object Constrained Language), который специально был создан для замены естественного языка при описании условий и ограничений, накладываемых на элементы модели. Язык OCL играет чрезвычайно важную роль в технологиях Borland MDA, рассматриваемых в данной книге. Он будет подробно рассмотрен в последующих главах. Сейчас лишь отметим, что язык OCL формально является частью UML.

Диаграмма классов

В состав стандарта UML входят 8 типов основных диаграмм, исчерпывающим образом описывающих структуру, состав и поведение моделируемого приложения. Все описываемые в этой книге инструменты создания MDA-приложений исполь-

зуют в настоящее время только один из этих типов диаграмм — диаграмму классов (class diagram). Диаграмма классов является основным источником платформенно-независимой информации (PIM) для формирования модели приложения в Borland MDA.

Диаграмма классов занимает центральное место в объектно-ориентированном анализе и проектировании (ООАП) программных систем. Она описывает статическую структуру системы, то есть состав элементов *(классов),* состав их *атрибутов* и *операций,* а также связи между классами *(отношения).* В диаграмме классов не содержится информации о развитии системы во времени.

Классы

В основе диаграммы классов лежит понятие *класс.* Не углубляясь сейчас в детали, будем считать, что понятие класса знакомо разработчикам, использующим методологию объектно-ориентированного программирования. Класс изображается в UML-модели в виде прямоугольника, разделенного в общем случае на три части (рис. 1.4).

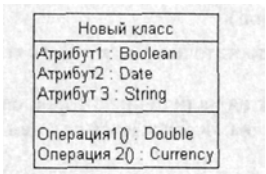


Рис. 1.4. Представление класса в UML

В верхней части отображается имя класса. Это обязательный параметр. Во второй сверху части отображаются *атрибуты* класса, возможно указание их типов и значений по умолчанию. В нижней части класса отображаются названия *операций* и, возможно, списки аргументов и типы возвращаемых результатов. Атрибутов и операций у класса может быть несколько. С точки зрения программиста атрибуты и операции аналогичны соответственно свойствам и методам в объектно-ориентированном программировании (ООП). Примером реального класса может служить, например класс Сотрудник, описывающий сотрудника фирмы (рис. 1.5).

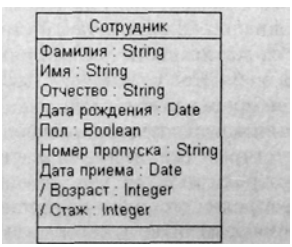


Рис. 1.5. Пример класса с вычисляемыми атрибутами

В этом примере присутствуют атрибуты, названия которых начинаются с символа «/» к /Возраст- и /Стаж. Этим символом отмечаются *вычисляемые* (derived) атрибуты, значения которых получаются как производные от других атрибутов и вычисляются в ходе работы приложения. В данном примере возраст и стаж могут быть вычислены по атрибутам Дата рождения и Дата приема соответственно.

Класс, который не может иметь ни одного объекта (экземпляра), является *абстрактным* классом. В этом случае его название отображается курсивом.

Отношения

Классы в UML могут быть связаны друг с другом посредством различного типа *отношений* (relationship)

Отношения сводятся к четырем базовым типам:

- зависимость (dependency);
- ассоциация (association);
- обобщение (generalization);
- реализация (realization).

Для дальнейшего нам достаточно рассмотреть только два из них: *ассоциация* и *обобщение*.

Ассоциация показывает на наличие некоторой связи между классами, например, «отдел—сотрудник». Она отображается сплошной линией (рис. 1.6).

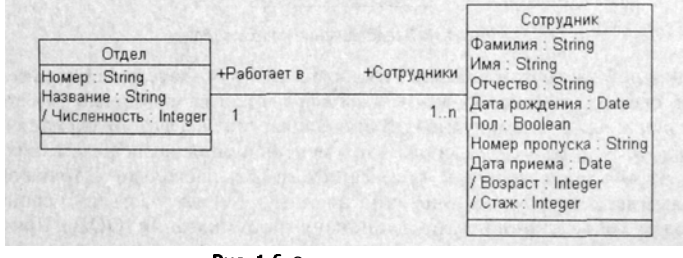


Рис. 1.6. Отношение ассоциации

Линия ассоциации может иметь стрелку на одном из концов, говорящую о «направлении» данной ассоциации. Наличие такой стрелки, как мы увидим далее (см. главу 4), указывает на возможность «навигации» в данном направлении, то есть просмотра значений элементов класса, на который указывает стрелка. Если стрелка отсутствует, такой просмотр возможен для обоих связанных классов. Ассоциация может иметь имя, в этом случае оно располагается над линией ассоциации (на рис. 1.6 имя отсутствует). Концы линии ассоциации помечаются названиями, которые обозначают роли классов, участвующих в отношении. В данном примере ассоциация имеет роли Сотрудники и Работает в. Кроме того, концы ассоциации имеют обозначения размерности, которые указывают на кратность отношения. Так, из рассмотрения рис. 1.6 можно сделать следующие выводы относи-

тельно описываемой модели (то есть восстановить заложенные в нее бизнес-правила):

- каждый отдел включает одного или несколько сотрудников (кратность 1..n);
- каждый сотрудник работает только в одном отделе (кратность 1).

Легко заметить кажущуюся аналогию между ассоциацией в UML и реляционными отношениями типа «один-ко-многим» и «многие-ко-многим» в БД. Однако продолжить эту аналогию не удастся. Во-первых, в реляционной БД связь «многие-ко-многим» не может соединять две таблицы, обязательно требуется промежуточная (связующая) таблица. В UML это вполне обычная ситуация. Во-вторых, кратность в UML может быть, в принципе, любой, в том числе и нулевой. Так, если бы мы на конце ассоциации, указывающей на сотрудника, написали O..n, то это бы означало, что могут существовать отделы, не имеющие ни одного сотрудника. А если бы мы задали кратность 15..50, то это бы означало, что численность сотрудников в отделе не может быть меньше 15 и больше 50.

В некоторых ситуациях отношение ассоциации может не связывать два класса, а применяться к единственному классу. Рассмотрим UML-модель, изображенную на рис. 1.7.

Рис. 1.7. Ассоциация с единственным классом

В этой модели оба конца ассоциации подключены к одному и тому же классу и определяют следующие бизнес-правила:

- каждый сотрудник может находиться в подчинении только у одного сотрудника (например, руководителя) либо не подчиняться ни одному из сотрудников (если он сам является руководителем) — кратность отношения 0..1;
- каждый сотрудник может руководить несколькими другими сотрудниками (если он является руководителем) либо не руководить ни одним (если он является рядовым сотрудником) — кратность отношения 0..n.

Использование ассоциаций в таком варианте нередко встречается при создании моделей приложений. Отношение ассоциации в качестве частного, но самостоятельно существующего в UML варианта включает тип отношения «агрегация».

Отношение «агрегация» описывает ситуации, когда классы связаны как «часть и целое», то есть один класс входит в другой. Такое отношение представляется в UML в виде линии с ромбиком на конце, причем ромбик отображается на стороне «целого» (рис. 1.8).

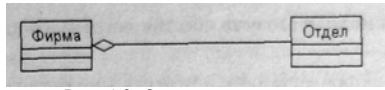


Рис. 1.8. Отношение агрегации

Данный тип отношения показывает, из каких составных частей состоит конкретный класс, то есть реализует декомпозицию целого на составляющие элементы.

Следующим типом рассматриваемых отношений является *обобщение,* или *генерализация.*

Данный тип отношения указывает, что между классами существует связь типа «предок–потомок», аналогичная отношению наследования в ООП. Такая связь отображается линией со стрелкой в форме пустого треугольника, причем треугольник-стрелка указывает на «предка», то есть на родительский класс. Родительский класс в UML носит название *суперкласс.*

Рис. 1.9. Отношение обобщения

На рис. 1.9 изображено отношение обобщения между родительским классом Человек и дочерним классом Гражданин. При этом необходимо учитывать, что все атрибуты класса-родителя автоматически принадлежат и классу-потомку, хотя в классе-потомке они не отображаются. Поэтому класс Гражданин, кроме отображаемых

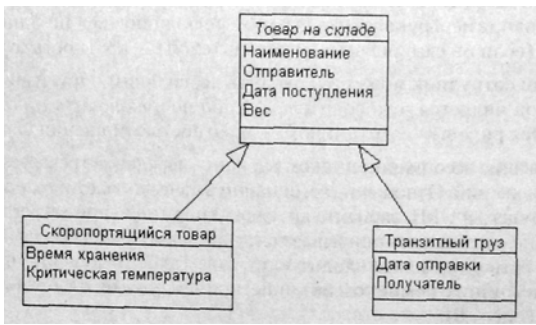


Рис. 1.10. Наследование от абстрактного класса

в нем атрибутов Гражданство и Номер удостоверения личности, будет включать и атрибуты Имя, Вес и Рост. Отношения обобщения довольно часто используются при построении моделей, при этом в качестве класса-родителя (суперкласса) нередко выступает какой-нибудь абстрактный класс. Например, на рис. 1.10 представлен фрагмент модели, описывающей складской терминал.

Из этого примера виден общий принцип построения подобных отношений обобщения — выделение в абстрактный суперкласс всех наиболее общих свойств (атрибутов), присущих каждому классу-потомку, при этом классы-потомки могут представлять сильно отличающиеся между собой объекты.

Отношения обобщения могут образовывать цепочки наследуемых классов, обеспечивая все большую конкретизацию описания классов-потомков по принципу «от общего — к частному» (рис. 1.11).

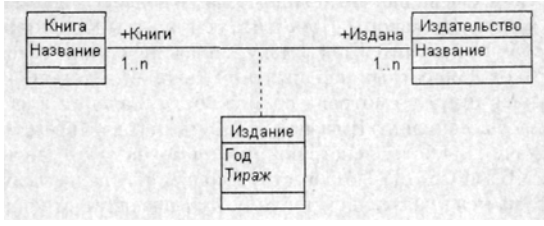


Рис. 1.11. Цепочка отношений обобщения

Использование отношений обобщения позволяет устранить повторное описание одинаковых свойств-атрибутов у классов-потомков и сосредоточиться на их специфических качествах.

Необходимо четко различать отношения агрегации и обобщения. В случае агрегации класс, представляющий «часть целого», не обязан иметь общих атрибутов с классом, представляющим «целое», он является в этом смысле независимым. Наоборот, класс-потомок в отношении обобщения всегда обладает полным набором атрибутов класса-родителя. Если рассматривать аналогию из объектно-ориентированного программирования, то в случае ассоциации типа «агрегация» в состав атрибутов класса «целого» включается атрибут с типом класса «часть».

Классы-ассоциации

Отдельным важным элементом диаграммы классов являются классы-ассоциации. Эти классы не обладают «самостоятельностью», а предназначены для дополнительного описания свойств какой-либо ассоциации, и неразрывно связаны с последней. Рассмотрим модель, содержащую два класса, — Книга и Издательство, каждый со своим набором атрибутов (рис. 1.12).

Эти классы связаны ассоциацией, имеющей кратности ролей на обоих концах вида 1..п, что описывают следующие бизнес-правила:

- книга может быть издана одним или несколькими (многими) издательствами;
- издательство может издать одну или несколько книг.

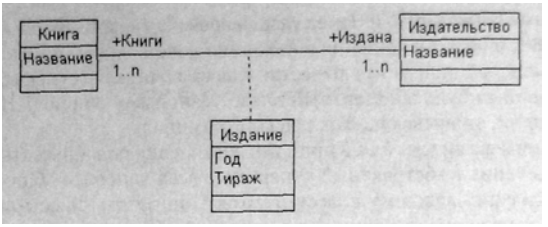


Рис. 1.12. Модель, включающая класс-ассоциацию

Далее предположим, что перед разработчиком ставится следующая задача: обеспечить сохранение информации о годе и тираже издания каждой книги каждым издательством. Очевидно, что эта информация должна содержаться в атрибутах какого-то класса. Но какого? Поместить ее в классы Книга или Издательство простым способом не получится, поскольку данная информация зависит от обоих классов, то есть от конкретных пар значений Книга—Издательство. Именно для подобных случаев и предусмотрена возможность создания классов-ассоциаций. Создав класс-ассоциацию Издание с атрибутами Год и Тираж, мы решаем поставленную задачу. Для более ясного понимания можно провести аналогию с реляционными СУБД (РСУБД). По существу ассоциация, в данном случае на уровне РСУБД, отображается в связующую таблицу, разрешающую отношение «многие-ко-многим» между таблицами Книга и Автор. В этой связующей таблице должны присутствовать два обязательных поля, условно назовем их: Код_книги и Код_издательства. Добавление в модель класса-ассоциации Издание можно тогда представить как простое добавление в указанную связующую таблицу еще двух полей: Год и Тираж (рис. 1.13).

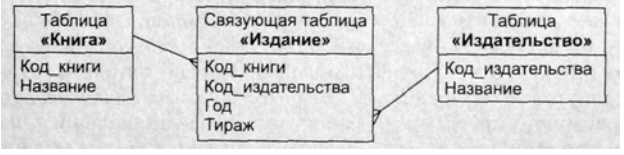


Рис. 1.13. Аналог класса-ассоциации в реляционной базе данных

Важно иметь в виду, что класс-ассоциация не может существовать отдельно от самой ассоциации, которую он представляет, и при удалении ассоциации из модели он будет автоматически удален.

Пакеты

В языке UML существует такое понятие, как *пакет*. Пакеты - это своеобразные «контейнеры» для классов модели, объединяющие в себе некоторые, определяемые пользователем, наборы классов вместе с их отношениями. Пакеты используются для группировки сходных по назначению классов, позволяя представить большую модель, содержащую сотни классов, в виде совокупности нескольких пакетов. Для пакета UML использует специальное обозначение (рис. 1.14),

содержащее имя пакета, которое присваивается разработчиком. Каждый класс модели может принадлежать только одному пакету. Существует возможность включения пакетов в другие пакеты, в этом случае вложенные пакеты называются подпакетами.

Рис. **1.14.** Обозначение пакета на диаграмме UML

UML-модель и схема БД

В этой главе мы не раз прибегали к аналогиям, основанным на структуре построения реляционных баз данных. И хотя с точки зрения функционального назначения и области применения, наверное, не совсем правомерно проводить сравнительный анализ принципов построения диаграмм классов UML и структурных схем реляционных баз данных, все же имеет смысл попытаться это сделать. Это полезно, так как большинство разработчиков, имеющих традиционный опыт создания приложений баз данных, при переходе на MDA-технологию сталкиваются с некоторыми трудностями. Как показывает практика, нередко в этой ситуации накопленный «традиционный» опыт даже является помехой.

На первый взгляд вроде бы справедливо выявить следующие основные аналогии:

- Класс (UML) - Таблица (РСУБД);
- Ассоциация (UML) -Связь (РСУБД).

Рассмотрим их подробнее. Понятия класса и таблицы внешне во многом схожи, однако существуют принципиальные моменты, не позволяющие поставить между ними знак равенства. Во-первых, класс, как известно из ООП, обладает таким ключевым качеством, как наследование. На UML-диаграмме классов такая связь между классами отображается, как мы уже знаем, отношением обобщения. Аналога в РСУБД нет. Во-вторых, в составе класса могут присутствовать операции, что является отражением другого важного качества класса в ООП — поведения (наличия методов класса). Прямых аналогов в РСУБД также не имеется. Косвенными аналогами некоторого «поведения» в РСУБД можно признать такие сущности, как триггеры, посредством которых реализуются реакции на определенные события.

Перейдем к ассоциациям и связям. Первое различие, сразу обращающее на себя внимание, — ассоциации в UML связывают классы, а не отдельные атрибуты классов.

В РСУБД связи подключаются к конкретным ключевым полям таблиц. Легко заметить, что понятия ключевых полей, вторичных ключей и индексов в UML просто отсутствуют (хотя при необходимости в каких-то ситуациях можно включать в класс дополнительные атрибуты для этих целей). Второе важное отличие — ассоциации в UML могут иметь производные кратности отношений на концах,

например, «многие-ко-многим» или даже «<0..5> к <37..154>». РСУБД в этом случае требует наличия связующей таблицы, а возможности кратности ограничены понятиями «один-ко-многим» и «один-к-одному».

Таким образом, даже из вышеприведенного краткого и неполного сравнения очевидно что кажущиеся аналогии таковыми не являются Поэтому при разработке модели на основе диаграммы классов UML нецелесообразно и даже вредно базироваться на структурной схеме реляционной базы данных. На самом деле, оазироваться при этом надо на предметной области, одним из основных инструментов описания которой и является диаграмма классов языка UML.

У читателя после сказанного может возникнуть закономерный вопрос - а каким образом в таком случае вообще функционируют MDA-приложения баз данных «нарушающие» основные принципы функционирования РСУБД? Ответ кратко можно сформулировать примерно так - в составе рассматриваемых в этой книге инструментов реализации Borland MDA имеются специальные средства, осуществляющие прозрачным для разработчика способом «объектно-реляционное отображение», при котором, по сути, происходит преобразование диаграммы классов (модели) в структурную схему РСУБД. Забегая вперед, стоит отметить, что при работе с Borland MDA разработчик практически не занимается созданием базы данных. Более того, он даже может не знать ее структуру и состав и не использовать при работе язык SQL. Подробнее эти вопросы будут рассмотрены в последующих главах.

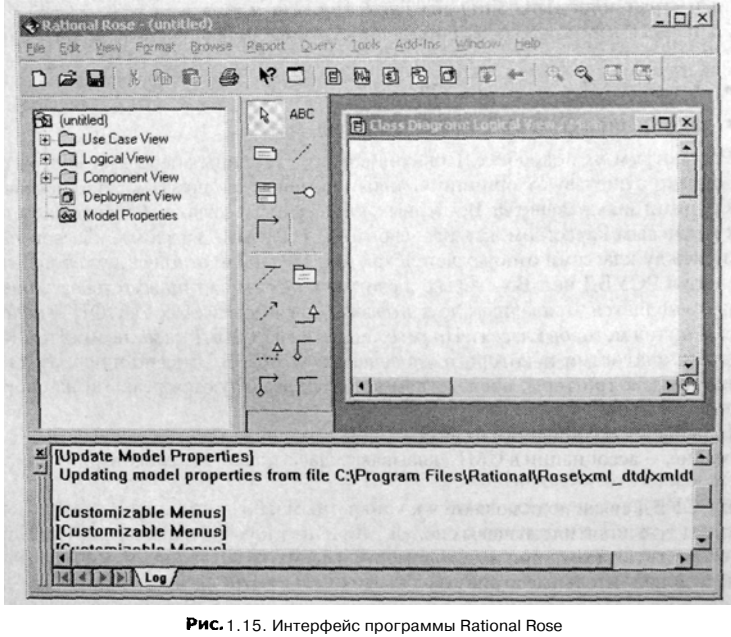


Рис. 1.15. Интерфейс программы Rational Rose

UML-редакторы

В заключение этой главы дадим краткий обзор инструментов, обеспечивающих разработку UML-моделей. Таких инструментов (UML-редакторов) в настоящее время существует довольно много. Самым первым из них был программный продукт Rational Rose, созданный в 1998 году. Он далеко не потерял своей актуальности, и активно применяется многими разработчиками и по сей день. Rational Rose представляет собой универсальный CASE-инструмент для моделирования и разработки приложений, имеющий чрезвычайно развитые программные интерфейсы с различными языками и средами программирования. Разработчиком Rational Rose является компания Rational Software, стоящая у истоков создания языка UML. В 2002 г. она была приобретена фирмой IBM.

Основы работы в Rational Rose

Здесь мы очень кратко познакомимся с принципами работы и возможностями редактора Rational Rose. Более полную информацию читатель может найти, например, в книге [2]. UML-редактор Rational Rose является стандартом де-факто среди инструментов UML-моделирования, обладая полным набором инструментов для создания всех основных диаграмм, предусмотренных стандартами языка UML. На рис. 1.15 представлен общий вид интерфейса программы Rational Rose.

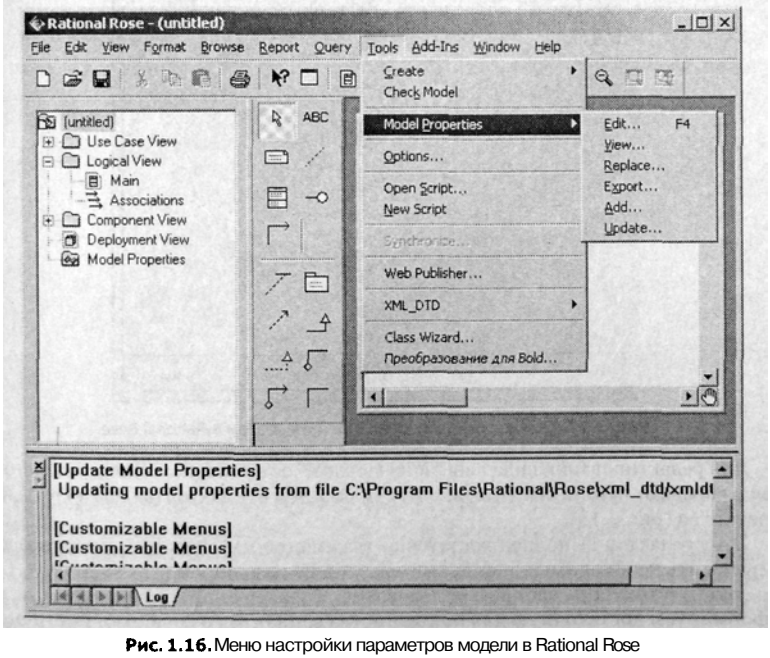


Рис. 1.16. Меню настройки параметров модели в Rational Rose

Rational Rose предоставляет разработчику богатые возможности по настройке модели. Для этого после создания новой UML-модели (команда File > New) можно выбрать соответствующие пункты меню Tools (рис. 1.16).

Команда Tools ▸ Model Properties • Edit... вызывает главное окно редактирования свойств UML-модели (рис. 1.17). Для начала настроим для UML-модели инструментальную панель, которая включает визуальные элементы, используемые при создании моделей (классы, пакеты, ассоциации).

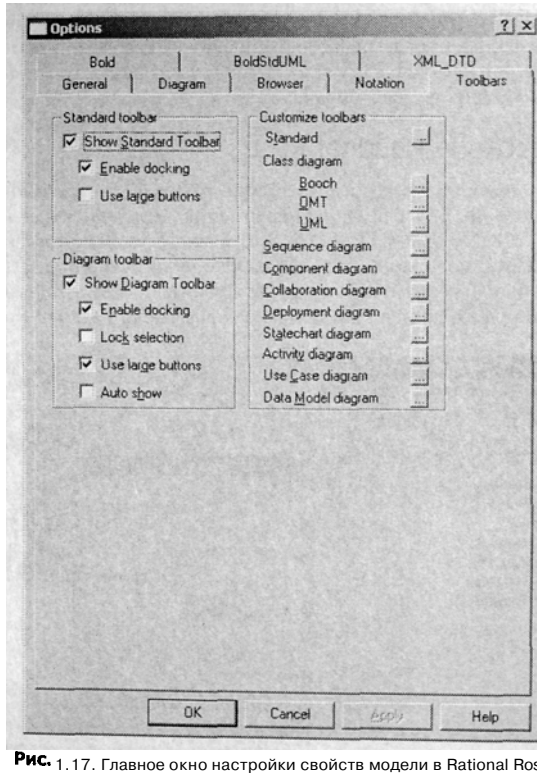


Рис. 1.17. Главное окно настройки свойств модели в Rational Rose

Для редактирования вида панели на вкладке Tolbars нажмем кнопку с многоточием справа от пункта UML (см. рис. 1.17) и попадем в редактор набора визуальных элементов (рис. 1.18).

Этот редактор выполнен достаточно традиционным способом, позволяющим переносить необходимые элементы между полным набором всех элементов (левое окно) и текущим набором пользователя (правое окно). Таким образом мы сформируем достаточную для большинства вариантов разработки инструментальную панель (рис. 1.19).

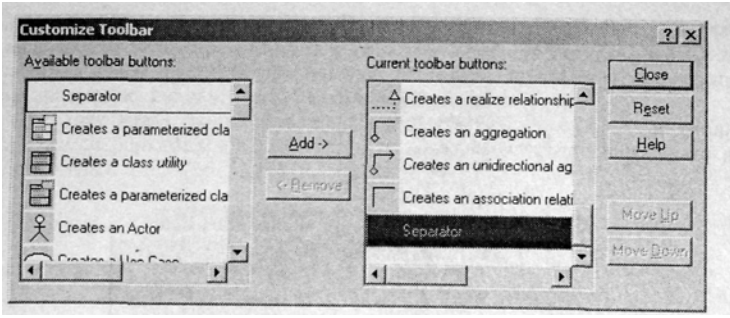


Рис. 1.18. Редактор набора визуальных элементов Rational Rose

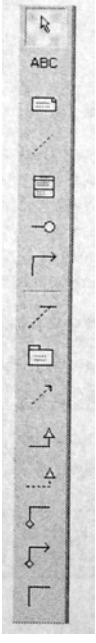


Рис. 1.19. Настроенная инструментальная панель

Использование этой панели для создания модели не представляет трудностей. Для добавления в модель какого-нибудь элемента, например нового класса, необходимо выбрать его на инструментальной панели и затем щелкнуть на свободном поле модели — при этом отобразится новый класс с параметрами по умолчанию. Для индивидуальной настройки класса достаточно вызвать правой кнопкой мыши контекстное меню для выбора вида настроек.

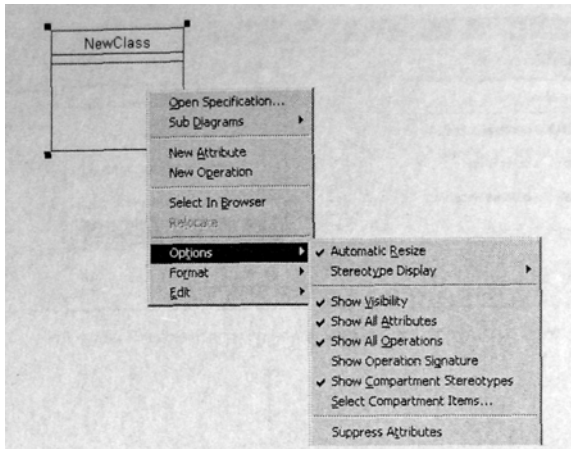


Рис. 1.20. Вложенные меню настроек параметров класса

При этом появятся последовательно вложенные меню (рис. 1.20), содержащие многочисленные возможности настроек параметров класса. С их помощью можно настроить визуальные параметры класса, видимость атрибутов, операций, параметры автоматической подстройки размеров изображения класса и т. д. Для быстрого перехода к настройкам класса необходимо дважды щелкнуть по его изображению. В результате отобразится главное окно настроек параметров класса (рис. 1.21).

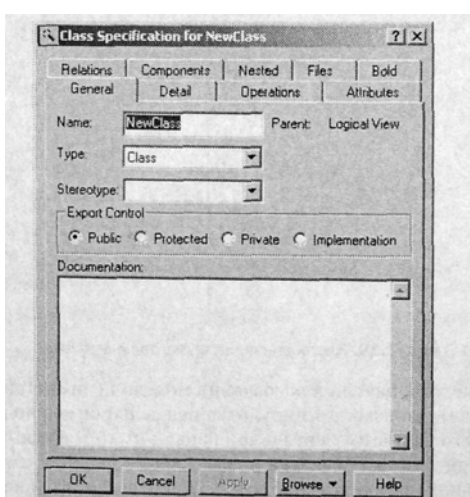


Рис. 1.21. Главное окно настроек параметров класса

Это окно имеет несколько вкладок. На вкладке General можно задать имя класса, тип и ряд других параметров. Кроме того, здесь же в специальном окне Documentation разработчик может ввести дополнительную текстовую описательную информацию, предназначенную для целей документирования.

На вкладке Detail можно задать подробную информацию для модели — является ли класс сохраняемым в БД (Persistent) или временным (Transient), является ли он абстрактным и т. д. (рис. 1.22).

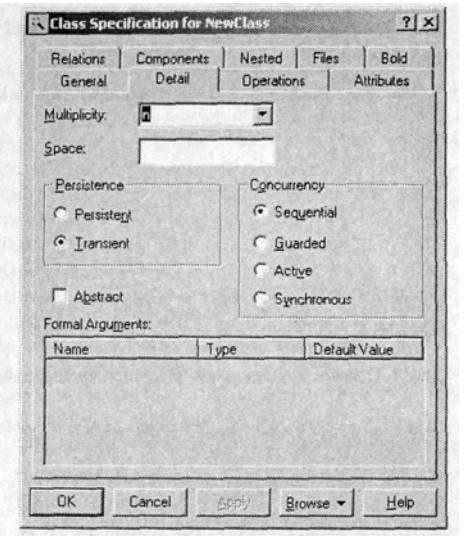


Рис. 1.22. Детальные настройки параметров класса

Пользуясь описанными возможностями, создадим простую модель (рис. 1.23), содержащую два класса, связанные ассоциацией. Для создания ассоциации необходимо выбрать ее элемент на инструментальной панели, затем щелкнуть на первом классе и провести линию до второго.

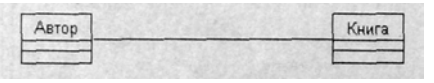


Рис. 1.23. Вид ненастроенной модели в Rational Rose

Теперь кратко опишем процедуру настройки ассоциации. Дважды щелкнув по линии ассоциации, откроем окно настроек ее параметров (рис. 1.24). Оно также содержит несколько вкладок. На вкладке General можно задать имя ассоциации и названия ролей для каждого класса.

На вкладках Role A Detail (рис. 1.25) и Role B Detail можно настроить подробные параметры для каждой роли ассоциации — например, задать имена ролей и их кратности.

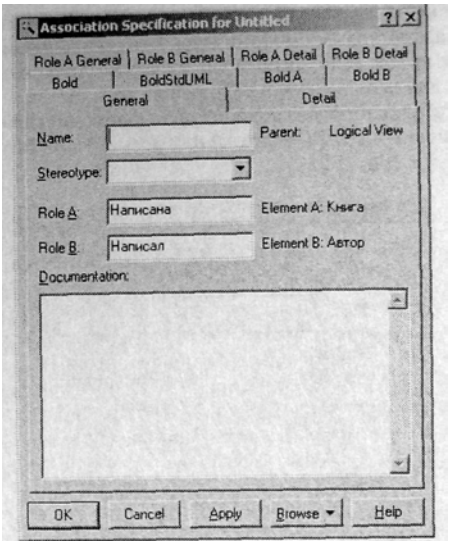


Рис. 1.24. Главное окно настроек параметров ассоциации

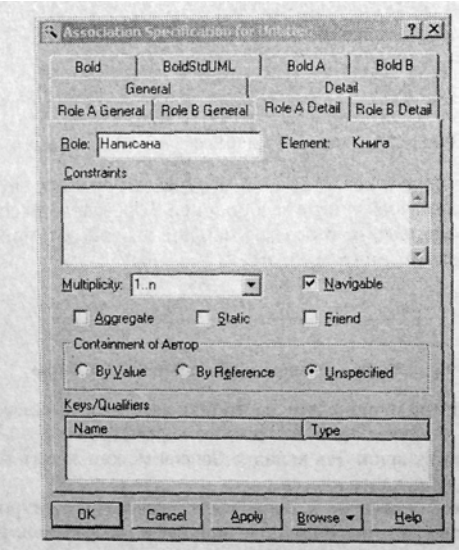


Рис. 1.25. Настройка параметров роли

После выполнения указанных настроек наша модель приобретет вид, представленный на рис. 1.26.

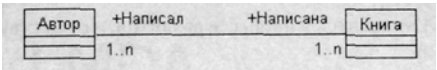


Рис. 1.26. Вид настроенной модели

Другие UML-редакторы

Практически все современные CASE-системы обладают встроенными UML-редакторами. Например, программный продукт PowerBuilder компании PowerSoft, также предоставляет широкие возможности для создания UML-диаграмм. Начиная с версии Delphi 6 появилась возможность интеграции в эту среду разработки UML-редактора ModelMaker. Этот программный инструмент также позволяет создавать сложные диаграммы классов и, кроме этого, обладает развитыми возможностями по взаимодействию с кодом разрабатываемого в Delphi приложения.

В состав новейших сред разработки Borland, рассматриваемых в приложении к этой книге, включены собственные средства создания UML-диаграмм, которые тесно интегрированы со средой разработки. Необходимо отметить, что с появлением стандарта языка XMI (XML Metadata Interchange — язык обмена метаданными XML) появилась принципиальная возможность разработки UML-модели в любом редакторе, поддерживающем спецификацию XMI, с возможностью последующего изменения ее в другом редакторе или использования этой модели в средах разработки MDA-приложений, например в программных инструментах Borland MDA.

Резюме

В этой главе дан краткий обзор основных концепций новой технологии разработки приложений — Model Driven Architecture (MDA), и ее сравнение с традиционными подходами, а также представлена краткая информация о языке унифицированного моделирования UML.

Предпосылками появления MDA является существование большого количества трудно интегрируемых между собой программных платформ разработки, каждая из которых является сложной и дорогостоящей в создании, внедрении и сопровождении программной системой.

В основе MDA лежит принцип создания платформенно-независимой модели приложения (PIM-модели), которая определяет структуру, состав и поведение будущего программного продукта.

Существуют также платформенно-зависимые (PSM) модели, играющие роль адаптеров для различных платформ разработки.

Наличие PIM- и PSM-моделей позволяет автоматически сгенерировать код приложения и, при необходимости, базу данных.

Изменение PSM позволяет автоматически получить функционально идентичное приложение для другой платформы без изменения пользовательского кода.

Показано, что применение MDA имеет важные преимущества перед традиционным подходом разработки приложений — по производительности, переносимости, управляемости и т. д.

Технология MDA находится в процессе развития, и существует несколько концепций ее реализации.

Внедрение MDA должно привести к значительным изменениям роли и места программистов-разработчиков и специалистов в области СУБД.

MDA базируется на языке унифицированного моделирования UML.

Язык UML является платформенно-независимым языком и предназначен для создания моделей.

UML не является языком программирования, а представляет собой язык графического описания, что реализуется посредством диаграмм.

Диаграмма классов, входящая в состав UML, предназначена для описания статической структуры и состава модели. Диаграмма классов является основным источником информации о бизнес-правилах и единственной диаграммой UML, используемой в инструментарии Borland MDA для формирования PIM-модели.

Основными элементами такой диаграммы являются классы и отношения между ними. Существуют специальные классы-ассоциации для реализации хранения дополнительных свойств отношений.

В результате краткого сравнительного анализа диаграммы классов UML и структуры реляционных баз данных сделан принципиальный вывод о нецелесообразности проведения аналогий между ними.

Обзор Borland MDA

Данная глава представляет собой обзор инструментальных средств и технологий, разработанных компанией Borland для реализации MDA-приложений.

Что такое Borland MDA (BMDA)

В этой книге под общим названием «Borland MDA» объединяются идеология, технологии и инструментальные средства, входящие в программные продукты компании Borland, которые предназначены для практической реализации MDA-архитектуры в процессе разработки приложений. Термин «Borland MDA» (сокращенно BMDA) появился в конце 2002 года, когда компанией Borland были приобретены несколько фирм-разработчиков программного обеспечения, занимающихся созданием инструментария для внедрения элементов MDA в процесс разработки. Более полная информация об этом будет представлена в следующем разделе.

Отметим, что наряду с термином «Borland MDA» в ряде источников, относящихся в том числе и к самой компании Borland, используется также и другой, похожий термин — «Model Driven Development», в переводе означающий «разработка, управляемая моделью». Однако, на взгляд автора, подобная трактовка является несколько «узкой», поскольку роль описываемых в этой книге инструментов Borland MDA не ограничивается только этапом разработки приложений. Иначе рассматриваемые в этой книге инструменты Borland MDA следовало бы отнести к классу CASE-систем. Но это не соответствует действительности. Как будет продемонстрировано в дальнейшем, технология и инструменты Borland MDA имеют большое значение на всех этапах создания приложений, включая и этап исполнения программ.

На рис. 2.1 представлена достаточно условная общая схема, иллюстрирующая состав элементов, образующих совокупность программных средств Borland MDA. В нее входят:

- **Средства проектирования и моделирования.** Предназначены для создания модели приложения. Включают в себя UML-редакторы, средства проверки целостности и непротиворечивости UML-модели, редакторы языка OCL, и инструменты импорта-экспорта моделей.
- **Средства разработки.** Обеспечивают программную реализацию объектного пространства, интерфейсы с уровнем представления и уровнем данных. Включают в себя классы и компоненты, используемые разработчиком при создании приложения.
- **Средства генерации.** Обеспечивают генерацию кода приложения на базовом языке (в Delphi — Oblect Pascal). Генерируемый код представляет собой программную реализацию всех элементов модели на базовом языке. Кроме того, эти средства обеспечивают также генерацию схем реляционных баз данных.
- **Средства отладки.** Предназначены для отладки MDA-приложений. Такие приложения требуют отладки на уровне объектов и сообщений объектного пространства.
- **Средства интеграции.** Обеспечивают интеграцию со средой разработки, с одной стороны, и необходимые настройки связей с уровнем данных (СУБД), с другой стороны.
- **Средства интерпретации и управления.** Обеспечивают функционирование на этапе исполнения, включая интерпретацию и доступ к элементам модели, контроль целостности объектного пространства, контроль взаимодействия с уровнями данных и графическим интерфейсом.

Рис. 2.1. Состав Borland MDA

В этой книге мы ограничим рассмотрение Borland MDA только средой программирования Delphi 7 Architect Studio, однако уже сейчас эта технология (существенно переработанная) внедрена еще в один программный продукт - Borland

C#Builder, а в недалеком времени, похоже, будет интегрироваться и в другие среды разработки компании Borland.

История развития

Рисунок 2.2 иллюстрирует этапы развития технологии Borland MDA.

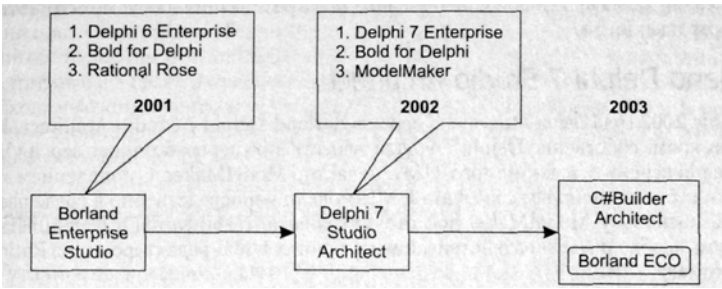


Рис. 2.2. Развитие продуктов Borland, поддерживающих MDA

Borland Enterprise Studio

Первым продуктом компании Borland, поддерживающим MDA-технологию, можно считать программный комплекс Borland Enterprise Studio, вышедший в 2001 году В состав его поставки входили следующие основные программные продукты:

- Borland Delphi 6 Enterprise (компания Borland);
- Bold for Delphi версия 3 (фирма BoldSoft);
- Rational Rose (компания Rational Software).

ПРИМЕЧАНИЕ

Кроме вышеперечисленных, в состав Borland Enterprise Studio входили и другие программные продукты (Borland Application Server и т. д.), однако они не представляют интереса в контексте нашего описания.

Идеология использования этого состава инструментов, по замыслу Borland, была примерно следующей — разработчик создавал модель приложения в CASE-системе Rational Rose на языке UML, затем, на следующем этапе, посредством пакета компонентов Bold for Delphi эта модель «транслировалась» в среду разработки Delphi 6, где и происходила окончательная «доводка» приложения и разработка интерфейса пользователя. Это — очень упрощенная схема, и на практике такой процесс мог выглядеть по-другому. Далее в этой книге мы подробнее познакомимся с этой технологией создания MDA-приложений на примере продукта Bold for Delphi (Delphi 7 Architect Studio). А что касается Borland Enterprise Studio в целом, то возможной причиной недостаточно широкого распространения этого продукта и предлагаемых подходов являлась, с одной стороны, недоработанность

продукта Bold for Delphi версии 3, а с другой стороны, несмотря на безусловно заслуженный авторитет Rational Rose, ее возможности при таком составе программных продуктов явно использовались далеко не в полной мере. По сути, Rational Rose применялась здесь только в качестве UML-редактора. Другими словами, Borland Enterprise Studio фактически представляла собой набор относительно слабо интегрированных между собой продуктов, созданных разными разработчиками. И, хотя сама по себе идея увязки всего процесса разработки в единый комплекс, безусловно, была передовой и правильной, но ее реализация на тот момент «отставала» от идеологии.

Borland Delphi 7 Studio Architect

В конце 2002 года была выпущена версия Borland Delphi 7 Studio Architect. В ее состав, кроме собственно Delphi 7, вошла существенно переработанная версия Bold for Delphi версии 4, и, кроме того, UML-редактор ModelMaker. С появлением этой версии стало возможным создавать UML-модели непосредственно в среде разработки, поскольку ModelMaker достаточно тесно интегрируется с Delphi. Таким образом, необходимость задействования сторонних UML-редакторов типа Rational Rose отпала.

С другой стороны, предлагаемые фирмой BoldSoft в этой версии Bold for Delphi, подходы и решения, оказались настолько удачными, что через месяц после выхода описываемой версии Delphi 7 Studio, компания Borland приобрела фирму BoldSoft.

ПРИМЕЧАНИЕ

Практически в то же время компания Borland приобрела также фирмы TogetherSoft и StarBase.

После этого события продукт Bold for Delphi стал принадлежать компании Borland. Здесь уместно сказать несколько слов о фирме BoldSoft. Являясь членом консорциума OMG, и непосредственным участником разработки языка OCL, шведская компания BoldSoft MDE Aktiebolag впервые предложила законченную концепцию практической реализации MDA в приложениях, создаваемых в Delphi и C++Builder, и разработала для этих целей программные продукты Bold for Delphi и Bold for C++Builder. Причем стоит подчеркнуть, что первая версия Bold for Delphi для Delphi 5 появилась задолго до принятия консорциумом OMG концепции развития MDA. Таким образом, фирма BoldSoft не стала дожидаться окончательной выработки правил и рекомендаций OMG и еще до выпуска соответствующих стандартов на практике реализовала свое видение MDA. Bold for Delphi оказался весьма удачным продуктом. Достаточно сказать, что он уже в течение нескольких лет успешно используется в информационной системе парламента Швеции, а также и в ряде других компаний. По отзывам разработчиков, применение этого продукта позволило повысить эффективность и сократить сроки создания программных систем до 10 раз. Достаточно тесная взаимная интеграция составляющих Delphi 7 Studio Architect программных продуктов плюс существенно переработанные версии самих продуктов дали свой результат. Именно после появления этой версии Delphi о компании Borland стали говорить не только как о разработчике RAD-сред (сред быстрой разработки программных продуктов - Rapid Application Development), но и как о разработчике продуктов, поддерживающих полный цикл создания приложений - от этапа проектирования до этапа генерации кода (ALM - Application LifeCycle Management).

C#Builder Architect

И, наконец, в конце 2003 года появляется качественно новый продукт компании Borland для платформы Microsoft .NET — это C#Builder Architect. Его принципиальная новизна заключается не только в том, что он поддерживает перспективную платформу .NET. С точки зрения MDA-архитектуры C#Builder Architect также предстает в качественно новом облике — после его появления можно констатировать, что процесс интеграции MDA-инструментов и сред разработки Borland принципиально завершился. Технология Borland MDA, получившая в C#Builder Architect название Borland ECO (Enterprise Core Objects), теперь полностью интегрирована в среду разработки, включая графический UML-редактор, а также необходимые компоненты и классы для программной поддержки MDA. В приложении к этой книге мы дадим краткий обзор возможностей C#Builder Architect и технологии Borland ECO.

Возможности и специфика

Для различных программных продуктов компании Borland конкретные наборы возможностей технологии BMDA довольно сильно различаются. Здесь мы познакомимся с основными возможностями, общими для всех продуктов компании Borland, поддерживающих разработку MDA-приложений. К таким возможностям можно отнести следующие.

- Создание UML-моделей. Для этих целей используется встроенный UML-редактор (текстовый или графический).
- Автоматическая генерация программного кода на языке Object Pascal.
- Автоматическая генерация структуры реляционных баз данных. Структура (схема) базы данных генерируется на основе созданной модели.
- Поддержка модификации базы данных с сохранением информации(DataBase Evolution), Специальный инструментарий позволяет изменить (естественно, в допустимых пределах и в зависимости от выбранной СУБД) схему базы данных без потери уже имеющихся данных.
- Возможность хранения базы данных в XML-документе без использования СУБД.
- Использование OCL (Object Constraint Language) в качестве основного средства формирования информационного интерфейса между различными уровнями приложения (СУБД, бизнес-уровень, графический интерфейс).

Принципиальным моментом при использовании BMDA является трехуровневая схема создания приложения, которая включает уровень данных, бизнес-уровень и графический интерфейс пользователя. И в данном случае это не абстракция, а реальность, воплощенная в конкретные наборы компонентов BMDA, с которыми имеет дело разработчик. Так, если обычно при создании приложения баз данных в Delphi визуальные компоненты (сетки, окна редактирования и т. д.) подключаются к полям или таблицам БД, то при работе с BMDA все они подключаются к промежуточному слою — объектам бизнес-уровня. Формирование биз-

нос-уровня приложения — одна из основных функции Borland MDA. **Другая основная функция - обеспечение взаимодействия между бизнес-уровнем и уровнем данных (СУБД) - объектно-реляционное отображение и взаимодействие.** Это взаимодействие включает автоматическую, прозрачную для разработчика, трансляцию ОСЬ в операторы SQL, выполнение операций с таблицами БД и т. д. Подробнее обо всем этом будет рассказано в последующих главах.

И, наконец, основное отличие Borland MDA от упомянутых выше CASE-средств — BMDA функционирует не только на этапе разработки приложения, но и на этапе его исполнения. Любое CASE-средство, сколь бы оно ни было совершенным, предназначено для реализации только этапов проектирования и моделирования. Оно, конечно, может включать в себя также возможности генерации кода и генерации базы данных. Но после запуска приложения CASE-система уже не функционирует, ее «присутствия» уже, по сути, нет. Функционирование BMDA коренным образом отличается. Сохраняя модель приложения в исполняемом файле, BMDA на этапе выполнения приложения использует ее с целью управления бизнес-уровнем, контроля целостности объектного пространства, управления взаимодействием бизнес-уровня с уровнем данных и графическим интерфейсом. Здесь необходимо отметить, что эта особенность BMDA является достаточно специфичной и отличающей эту технологию от остальных. В главе 1 рассматривались различные концепции реализации MDA-архитектуры. Если следовать приведенной там терминологии, то способность BMDA сохранять «знание» модели на этапе исполнения позволяет отнести эту технологию к «интерпретаторам». Однако это не совсем соответствует истине, поскольку BMDA также обладает возможностью генерации кода. Но BMDA — это и не генератор кода. В дальнейшем будет продемонстрировано, что при использовании отдельных версий BMDA (в частности, Bold for Delphi) в ряде случаев вообще необязательно генерировать код классов модели. В следующей главе мы создадим приложение баз данных, в котором вообще будут отсутствовать как код классов модели, так и пользовательский код. Немного забегая вперед, поясним, что информацию о модели BMDA сохраняет не в генерируемом коде, а в специальном компоненте.

Резюмируя вышесказанное, можно попытаться дать следующее **определение технологии Borland MDA:**

Borland MDA — это, с одной стороны, технология и среда разработки, позволяющая на этапе создания формировать объектное пространство (бизнес-уровень) и реализовывать бизнес-логику приложения, а с другой - программная система, обеспечивающая на этапе выполнения функционирование бизнес-уровня и его интеграцию с СУБД (уровнем данных) и графическим интерфейсом пользователя.

Преимущества для разработчиков

Какие же преимущества дает использование технологии Borland MDA разработчикам приложений? Здесь мы опять ограничимся перечислением лишь основных достоинств этой технологии.

- Единый подход ко всем этапам разработки - от проектирования модели до разработки приложения, заключающийся в том, что разработчик на всех эта-

пах работает с одними и теми же сущностями — объектами модели. Здесь отсутствует разрыв между красивой схемой-моделью и программированием приложения СУБД, так как разработчик «не опускается» на уровень базы данных, он даже может не знать, какова структура БД и какие таблицы в ней присутствуют. Разработчик всегда использует те имена и сущности, которые он сам включил в модель.

- Полностью устраняется этап «ручного» создания базы данных. Все таблицы, поля, индексы, ключи генерируются автоматически в соответствии с моделью. Для использования конкретной СУБД достаточно подключить и настроить один из адаптеров баз данных, входящих в состав BMDA. Есть возможность создания собственных адаптеров баз данных.
- Модификация базы данных превращается в тривиальный процесс — после внесения необходимых изменений в модель достаточно просто сгенерировать новую базу данных. Становится не принципиально, какую именно СУБД использовать: при смене СУБД само приложение и его код не меняются.
- Использование языка OCL позволяет полностью абстрагироваться от SQL-диалекта конкретной СУБД. Язык SQL в ряде случаев становится практически ненужным и используется достаточно редко, хотя возможность его задействования сохраняется.

Необходимо отметить, что средства автоматической генерации баз данных и даже классов приложений существовали и ранее. Достаточно привести в пример Rational Rose и PowerBuilder компании Powersoft. Существует и такой программный продукт, как Delphi RoseLink компании Ensemble Systems, являющийся «мостом» между CASE-системой Rational Rose и Delphi. Основные его функции — генерация кода на Object Pascal и обратное проектирование. Однако, генерируемый код не содержит реализацию функциональности. Генерируются только описания — определения классов, интерфейсов и т. д.

Borland MDA отнюдь не ограничивается этим, поскольку, интегрируясь в среду Borland Delphi, предоставляет разработчику полный набор визуальных и невизуальных компонентов, достаточный для реализации «объектного пространства» (Object Space) приложения. Поэтому разработчик получает возможность работать не на уровне кода и таблиц БД, а на уровне объектов внутри этого объектного пространства.

Другими словами, используя Borland MDA, разработчик:

- не создает базу данных, а формирует модель приложения на языке UML;
- работает не с таблицами, полями и ключами базы данных, а с объектами созданной им модели приложения — классами и их атрибутами;
- подключает визуальные компоненты для отображения данных не к таблицам БД, а к объектам модели;
- не пишет запросы на языке SQL, а формирует предложения на гибком и мощном диалекте языка OCL.

Как принято говорить, в этом случае разработка ведется на *бизнес-уровне*.

Все вышесказанное будет наглядно, с использованием конкретных примеров продемонстрировано в последующих главах.

Последовательность изучения

Далее в этой книге будет рассмотрено практическое применение технологии Borland MDA на примере Delphi 7 Studio Architect и Bold for Delphi. Базовые основы и принципы технологии BMDA были заложены и получили развитие именно в Delphi 7 Studio Architect. Для лучшего понимания идеологии и практического освоения технологии ECO (Enterprise Core Objects), реализованной в C#Builder Architect, весьма полезно предварительно ознакомиться с «истоками» этой технологии, заложенной в Delphi 7. Это целесообразно сделать по следующим причинам: во-первых, в ECO во многом присутствует определенная преемственность принципов и подходов. Во-вторых, в текущей версии C#Builder Architect возможности ECO в ряде случаев более ограничены по сравнению с имеющимися в Delphi 7 Architect. Это объясняется пока еще неполной и незавершенной на сегодняшний день «трансляцией» Bold for Delphi на новую программную платформу Microsoft .NET. Поэтому вполне вероятно, что в будущих версиях и обновлениях программных продуктов компании Borland технология ECO будет также обновлена и дополнена, и, скорее всего, это будет сделано и с целью дооснащения ECO теми качествами, которые присутствовали в Bold for Delphi. Следовательно, и с этой точки зрения полезно ознакомиться с «истоками» этой технологии.

И, наконец, следует принять во внимание тот немаловажный факт, что **Delphi 7 Studio Architect является единственным на сегодняшний день продуктом Borland, обеспечивающим разработку MDA-приложений для платформы Windows (Win32).** Если компания Borland в будущем продолжит «линейку» Delphi под платформу Win32, то, безусловно, и в этих новых версиях Delphi технология Borland MDA будет сохранена и расширена.

Возможные трудности

Необходимо заранее отметить, что на пути практического овладения технологией Borland MDA разработчик может столкнуться с некоторыми трудностями.

Во-первых, Borland MDA — это сложная и объемная программная система. Например, программная реализация рассматриваемой в этой книге версии Bold for Delphi включает тысячи новых классов, атрибутов и методов. Во-вторых, в настоящее время технической информации по Borland MDA совершенно недостаточно. Практически единственным источником конкретной информации по технологии на момент написания этих строк являются интернет-конференции на новостном сервере Borland (news://forums.borland.com), где размещены две основные группы новостей по Borland MDA: borland.public.delphi.modeldrivenarchitecture.general и borland.public.delphi.modeldrivenarchitecture.thirdparty. И, наконец в-третьих, Borland MDA - это качественно новая технология разработки, можно сказать, что это целый новый мир, и при переходе на эту технологию должно довольно резко перестроиться мировоззрение разработчика. Здесь все непривычно с точки зрения традиционных методов и средств. Поэтому на практике вполне возможна довольно парадоксальная ситуация - чем меньше у разработчика традиционного опыта создания приложений баз данных, тем легче ему будет освоиться в Borland MDA.

Эти предостережения, конечно, не означают, что данную технологию очень трудно освоить — на самом деле уже сотни разработчиков в мире относительно давно и с успехом применяют Borland MDA в своих программных разработках.

Bold for Delphi — основа Borland MDA

Как уже говорилось ранее в этой главе, программный продукт Bold for Delphi, созданный шведской компанией BoldSoft, в настоящее время является собственностью компании Borland. Ранние версии Bold for Delphi, разработанные для Delphi 5 и Delphi 6, в настоящее время недоступны для разработчиков, и поэтому единственным продуктом реализации MDA в Delphi сейчас является Borland Delphi 7 Studio Architect (существует также возможность обновления версии Borland Delphi 7 Studio Enterprise). Все текущие и последующие обновления Borland MDA осуществляет компания Borland (в 2003 году вышла последняя обновленная версия Bold for Delphi 4.0.0.21, доступная для скачивания зарегистрированным пользователям).

Что же представляет собой Bold for Delphi? С точки зрения эксплуатации разработчиком — это пакет компонентов, устанавливаемый отдельно от собственно среды разработки Delphi. Bold for Delphi включает более 100 визуальных и невизуальных компонентов. Во «внутренностях» пакета содержится около 1700 классов.

Ниже перечислены основные возможности Bold for Delphi:

- встроенный текстовый UML-редактор для создания моделей;
- тесная интеграция с CASE-системой Rational Rose в части создания, настройки, импорта и экспорта UML-моделей;
- поддержка языка XMI в части импорта и экспорта UML-моделей;
- встроенный интеллектуальный редактор OCL-выражений;
- автоматическая генерация программного кода классов модели на языке Object Pascal;
- возможность создания полноценных приложений без генерации кода классов модели;
- автоматическая генерация схемы реляционных баз данных, доступных через программные интерфейсы BDE, ADO, DBExpress, SQLDirect; также поддерживаются СУБД Interbase/Firebird и СУБД DBISAM;
- возможность работы с пользовательскими («нестандартными») СУБД;
- возможность хранения базы данных в XML-документе без использования СУБД;
- поддержка модификации схемы базы данных с сохранением информации (DataBase Evolution);
- использование OCL (Object Constraint Language) для реализации гибких и мощных запросов к объектному пространству, а также для взаимодействия между различными уровнями приложения (СУБД, бизнес-уровень, графический интерфейс);

- настраиваемый механизм объектно-реляционного отображения, включающий в себя автоматическую трансляцию операторов OCL в SQL;
- механизм «подписки» на события, возникающие в системе;
- автоматически генерируемые графические формы для отображения и редактирования данных;
- создание многозвенных приложений и «тонких» клиентов на базе DCOM;
- собственные средства отладки MDA-приложений.

Даже основных перечисленных возможностей достаточно для кардинального повышения эффективности разработки приложений баз данных. Надо также отметить, что использование Bold for Delphi особенно «выгодно» при разработке приложений для «больших» баз, включающих 500 и более таблиц. Хотя этот продукт с успехом может применяться и для локальных СУБД, и для автономных баз данных, где его использование также позволяет в разы сократить время разработки.

Такие уникальные возможности продукта Bold for Delphi не могли остаться незамеченными сторонними фирмами. В настоящее время существует несколько программных пакетов различных фирм-производителей программного обеспечения, созданных специально для технологии Bold for Delphi. Краткий обзор этих продуктов будет дан в главе 15.

Инсталляция и обзор инструментов

Последняя версия продукта Delphi 7 Studio Architect (trial-версия для ознакомления) доступна для загрузки с сайта компании Borland. При инсталляции Delphi 7 Studio Architect Trial Edition на экране появится заставка (рис. 2.3), позволяющая выбрать устанавливаемые продукты: Delphi 7, ModelMaker и Bold for Delphi.

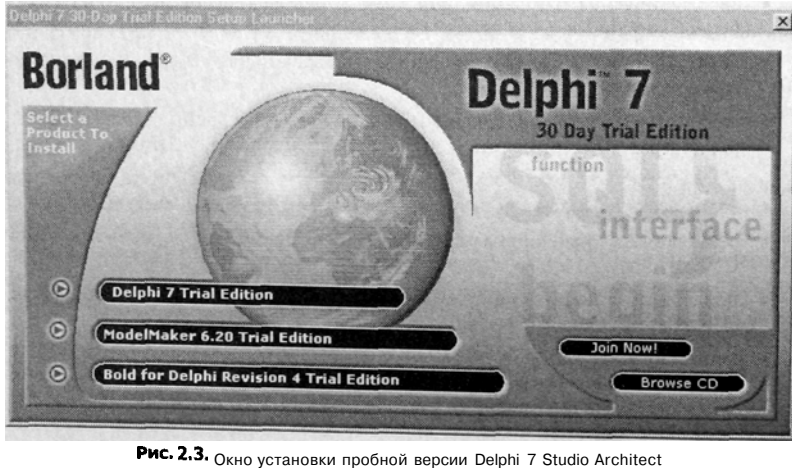


Рис. 2.3. Окно установки пробной версии Delphi 7 Studio Architect

Продукты устанавливаются в порядке их перечисления. После установки Delphi 7 рекомендуется установить также и UML-редактор ModelMaker, позволяющий создавать UML-диаграммы классов (хотя это и не обязательно). Если при инсталляции Bold будет предложено выбрать состав установки (рис. 2.4), рекомендуется пометить все компоненты.

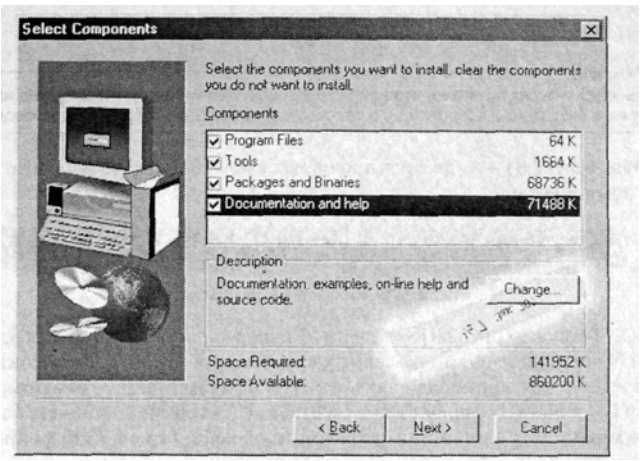


Рис. 2.4. Выбор компонентов для установки

После установки на палитре компонентов Delphi появятся новые вкладки, содержащие визуальные и невизуальные компоненты, предназначенные для создания MDA-приложений. Основные из них представлены ниже.

BoldHandles (рис. 2.5) — содержит невизуальные компоненты для формирования бизнес-уровня приложения.

Рис. **2.5.** Компоненты для формирования бизнес-уровня

BoldPersistence (рис. 2.6) — содержит невизуальные компоненты для взаимодействия с уровнем данных (СУБД). Уровень данных в BMDA носит название Persistence Laver.

Рис. 2.6. Компоненты для взаимодействия с СУБД

BoldControls (рис. 2.7) — содержит визуальные и невизуальные компоненты для создания графического интерфейса пользователя.

Стоит еще раз обратить внимание на то, что визуальные компоненты этой версии BMDA взаимодействуют не с данными, а с объектами бизнес-уровня. По этой

причине BMDA имеет собственные аналоги таких визуальных компонентов, как метка (Label), сетка (Grid) и т. д., обладающие дополнительными необходимыми для такого взаимодействия свойствами.

Рис. 2.7. Компоненты для создания графического интерфейса

ПРИМЕЧАНИЕ

В дальнейшем будет показано, что при необходимости во многих случаях можно использовать и обычные визуальные компоненты, в том числе и компоненты сторонних производителей.

BoldMisc (рис. 2.8) — содержит прочие вспомогательные визуальные и невизуальные компоненты.

Рис. 2.8. Вспомогательные компоненты

Bold COM Handles (рис. 2.9) — содержит невизуальные компоненты для формирования бизнес-уровней многозвенных приложений. Такие приложения позволяют распределять функциональность между «звеньями» таким образом, что появляется возможность создавать так называемые «тонкие» клиенты баз данных, которые характеризуются минимальными требованиями с точки зрения наличия средств доступа к данным. Более подробная информация об этом будет представлена далее.

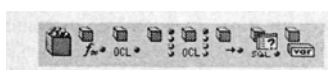


Рис. 2.9. Компоненты для работы с многозвенными приложениями

Bold COM Controls (рис. 2.10) — содержит визуальные и невизуальные компоненты для формирования графического интерфейса многозвенных приложений.

Рис. 2.10. Компоненты для создания графического интерфейса многозвенных приложений

Остальные вкладки Bold for Delphi будут рассмотрены отдельно, при описании дополнительных возможностей продукта.

Резюме

В данной главе приведен обзор программных продуктов компании Borland, поддерживающих разработку MDA-приложений.

История создания таких продуктов насчитывает несколько лет. Приведена информация об общих характеристиках и возможностях программных средств

Borland MDA, а также о преимуществах, которые они предоставляют разработчику, сокращая время разработки приложений баз данных практически на порядок Базовой основой MDA-технологии Borland является созданный ранее программный продукт — пакет компонентов Bold for Delphi, вошедший в версию Borland Delphi 7 Studio Architect. Данная версия Delphi в настоящее время является единственкьш программным инструментом Borland, позволяющим создавать MDA-приложения для платформы Win32 (Windows). Дана информация об инсталляции пакета Bold for Debhi и о составе и назначении основных его компонентов

Быстрый старт

Чтобы быстро ознакомиться с возможностями новой технологии, в этой главе мы рассмотрим, как на практике создается MDA-приложение с использованием Bold for Delphi.

Создание простого MDA-приложения

Создание бизнес-уровня

Создадим отдельную папку для нового Delphi-проекта. Создадим в Delphi новый проект, состоящий из одной формы, и сохраним его в этой папке. Пусть по умолчанию он будет сохранен с именем projectl.dpr, а модуль — с именем по умолчанию unit1.pas.

На панели компонентов Delphi выберем вкладку BoldHandles. Поместим на форму следующие три компонента с вкладки BoldHandles:

- BoldModel1 (компонент, обеспечивающий хранение модели);
- BoldSystemHandle1 (основной компонент-дескриптор объектного пространства);
- BoldSystemTypeInfoHandle1 (основной компонент-дескриптор типов модели).

Эти компоненты реализуют основу «объектного пространства» (Object Space) нашего приложения. Для правильного функционирования они должны быть связаны между собой и настроены следующим образом.

Для компонента BoldSystemTypelnfoHandlel в инспекторе объектов присвоим свойству BoldModel значение BoldModell (оно появится в раскрывающемся списке). Кроме того, присвоим свойству UseGeneratedCod значение False (рис. 3.1), задав этим, что генерация кода для классов модели нашего приложения производиться не будет.

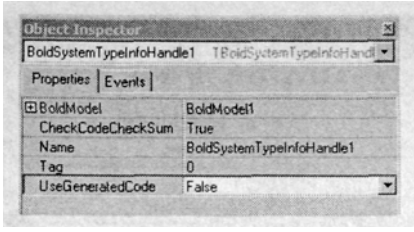


Рис. 3.1. Настройка BoldSystemTypeInfoHandle1

Для компонента BoldSystemHandle1 в инспекторе объектов присвоим свойству SystemTypeInfoHandle значение BoldSystemTypeInfoHandle1 (оно также появится в раскрывающемся списке). Кроме того, присвоим свойству AutoActivate значение True (рис. 3.2). Так обеспечивается активизация объектного пространства «по первому требованию».

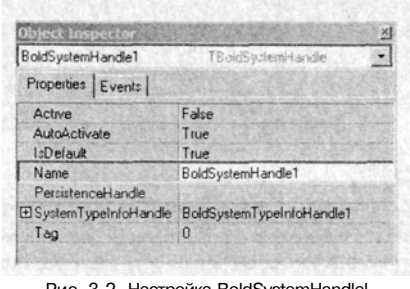


Рис. 3.2. Настройка BoldSystemHandle1

Мы создали прототип бизнес-уровня для нашего приложения. Последовательность вышеуказанных действий практически одинакова и всегда повторяется при создании любого приложения с использованием Bold. Однако бизнес-уровень пока не наполнен функциональным содержанием, так как к этому моменту еще не сделано главное, без чего не может функционировать ни одно MDA-приложение, а именно — не создана модель приложения, в соответствии с которой оно будет работать.

Создание модели приложения

Для создания модели в нашем примере мы будем использовать встроенный в Bold for Delphi редактор моделей. Для перехода в редактор дважды щелкнем на компоненте BoldModel1, расположенном на нашей форме, или правой кнопкой мыши вызовем на этом компоненте контекстное меню и перейдем на его первый пункт Open Bold UML Editor. Появится окно встроенного редактора UML (рис. 3.3).

Этот редактор позволяет создавать UML-модели (в текстовом режиме), а также обеспечивает реализацию основных функциональных возможностей Bold на этапе разработки приложения — таких, как экспорт и импорт моделей из других редакторов, модификация, верификация и тонкая настройка моделей, генерация

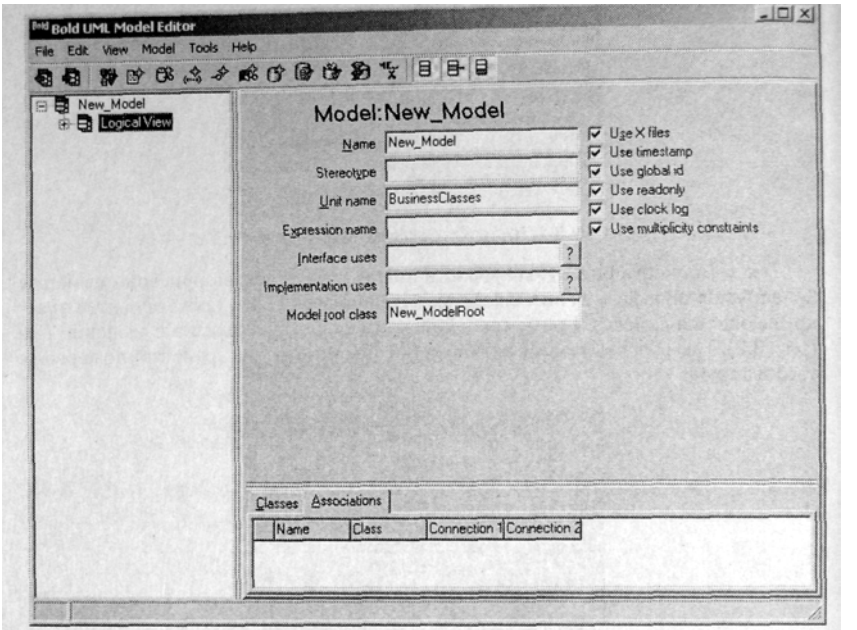


Рис. 3.3. Интерфейс встроенного редактора UML

баз данных и кода. На настоящем этапе мы его используем для создания модели нашего простого приложения.

Прежде чем приступить к непосредственному созданию модели в UML-редакторе, она должна быть сформулирована неформально. Например, будем считать, что разрабатываемое приложение должно обеспечивать работу с библиотечным каталогом, содержащим информацию об авторах и написанных ими книгах.

Для упрощения примем, что автор характеризуется своей фамилией, а книга — названием.

Кроме того, специально введем еще одно искусственное условие — у каждой книги может быть только один автор.

Таким образом, после некоторой дальнейшей конкретизации, неформально будущую модель приложения можно описать следующими основными предложениями (то есть набором бизнес-правил):

- описываемая предметная область содержит множество авторов и множество книг;
- автор однозначно идентифицируется текстовым атрибутом — фамилией;
- книга однозначно описывается текстовым атрибутом — названием;
- автор может написать много книг;
- книга может быть написана только одним автором.

Назначение UML-редактора — преобразовать неформальное описание бизнес-правил в формальную модель на языке UML.

Теперь можно приступить к созданию модели.

Правой кнопкой мыши щелкнем на пункте LogicalView в верхней части левой панели редактора и выберем в появившемся контекстном меню пункт NewClass — создание нового класса. Проделаем эту операцию еще один раз, и получим результат, представленный на рис. 3.4.

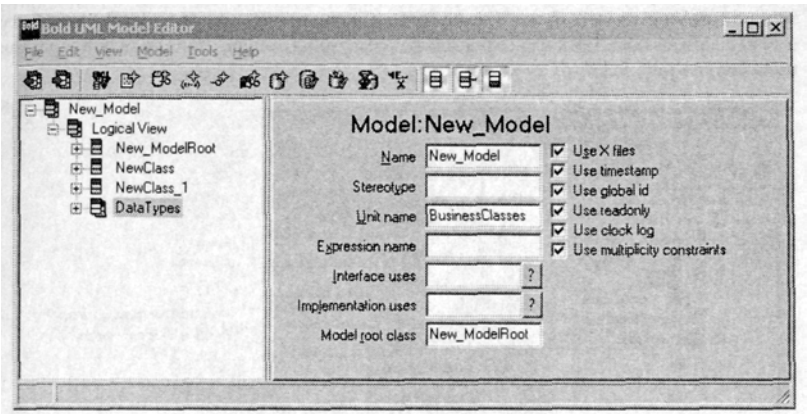


Рис. **3.4.** Вид модели после создания новых классов

Новые классы по умолчанию получили имена NewClass и NewClass_1. Для удобства дальнейшей работы переименуем их. Для этого в левой панели редактора выберем пункт NewClass, перейдем в правую панель редактора и введем в поле Name

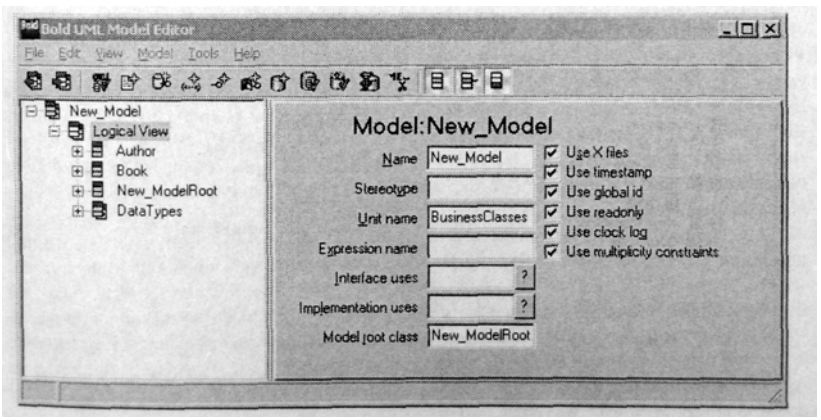


Рис. 3.5. Вид модели с переименованными классами

имя нашего класса - Author, очистив прежнее имя. Для второго класса NewClass_1 назначим аналогично имя Book.

После этого наша модель примет вид, представленный на рис. 3.5.

Теперь создадим атрибуты наших классов. Автор книги, как мы договорились раньше, описывается в нашей модели фамилией, а книга — названием. Следовательно, класс Author будет иметь один атрибут, назовем его aname, класс Book также будет иметь единственный атрибут, которому дадим название btitle. Для создания атрибута выберем в левой панели редактора нужный класс, вызовем контекстное меню и выберем в нем пункт NewAttribute. Создадим по одному новому атрибуту для классов Author и Book и раскроем дерево модели на левой панели редактора, как показано на рис. 3.6.

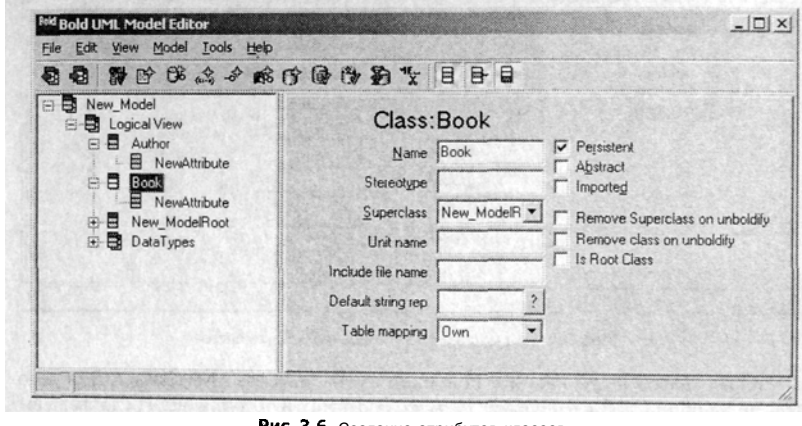


Рис. 3.6. Создание атрибутов классов

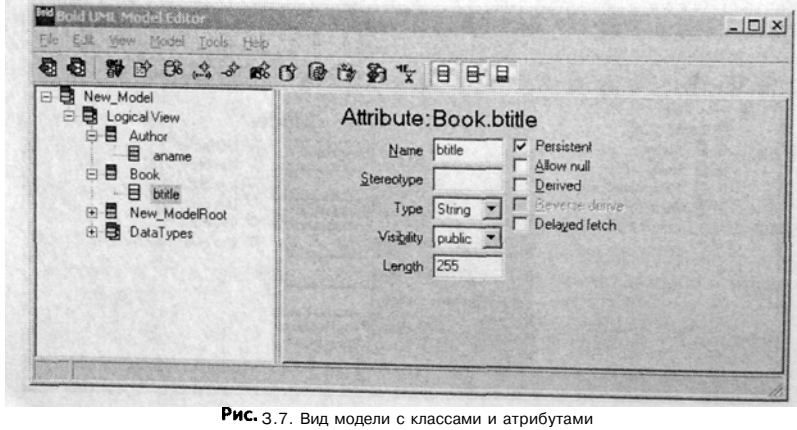


Рис. 3.7. Вид модели с классами и атрибутами

По умолчанию новые атрибуты получили имена (каждый в своем классе) NewAttribute. Переименование атрибутов осуществляется так же, как и классов. Сделаем это и получим модель в соответствии с рис. 3.7.

Классы и атрибуты, созданные нами, пока не полностью моделируют сформулированные выше бизнес-правила. А именно, наша модель пока ничего «не знает» о том, что авторы пишут книги, а книги написаны авторами. То есть между классами должна еще появиться некоторая связь, или отношение. Для создания такого отношения необходимо построить ассоциацию, связывающую наши классы. Поступим так же, как и при создании новых классов, выберем пункт LogicalView в дереве модели на левой панели редактора, затем вызовем правой кнопкой мыши контекстное меню, но теперь выберем в нем пункт NewAssociation. После этого раскроем дерево модели как показано на рис. 3.8.

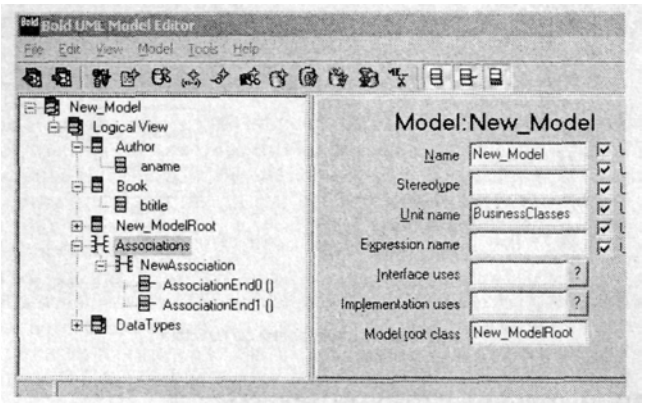


Рис. 3.8. Вид модели, включающей ассоциацию

Слева в дереве модели мы видим, что созданная ассоциация имеет два подпункта — AssociationEndO и AssociationEndl. Это автоматически созданные названия для концов нашей ассоциации. Концы ассоциации характеризуют роли классов (см. главу 1). Определим следующие роли для концов нашей ассоциации. Выделим роль ассоциации AssociationEndO и в правой панели редактора введем в поле Name имя роли byAuthor, а в поле Class выберем из раскрывающегося списка класс Author. Установим значение поля Multiplicity (кратность отношения) равным 1..1. Аналогично выелим роль ассоциации AssociationEndl и в правой панели редактора введем в поле Name имя роли writes, а в поле Class выберем из раскрывающегося списка класс Book. Установим значение поля Multiplicity равным 1..*. После этого модель примет вид в соответствии с рис. 3.9.

Теперь давайте немного подумаем, что мы только что сделали с ролями нашей ассоциации и почему. Как говорилось раньше, в нашей модели принято упрощающее предположение о том, что книга может быть написана только одним автором. Именно по этой причине мы установили свойство Multiplicity у роли byAuthor равным 1..1, то есть «один и только один (автор)*. С другой стороны, наша модель

по умолчанию допускает, что автор может написать много книг, и поэтому на другом конце нашей ассоциации writes свойству Multiplicity присвоено значение 1..* (см рис 3 9) то есть, другими словами «одна, две... много (книг)». Осталось теперь понять, почему 1..1 относится именно к автору, а 1..* – к книгам. Эту привязку сделали мы сами, когда назначали классы для концов нашей ассоциации, и на рис. 3.9 наглядно видна эта привязка - над свойством Multiplicity расположено окошко с названием соответствующего класса.

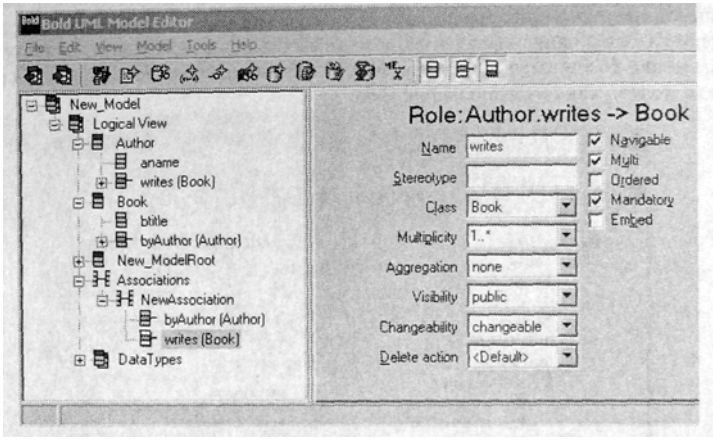


Рис. 3.9. Задание ролей ассоциации

Создание графического интерфейса

Итак, к настоящему моменту мы создали основу бизнес-уровня и модель приложения. Дошла очередь до графического интерфейса.

Сразу стоит напомнить об одном важном обстоятельстве, упомянутом в предыдущей главе. При работе с Bold необходимо привыкнуть к наличию бизнес-уровня в вашем приложении, который существует не абстрактно, а представлен специальными компонентами-дескрипторами для доступа к каким-либо структурам или данным. Эти компоненты не являются визуальными, а служат своеобразными трансляторами и реализаторами запросов, которые посылает графический уровень к бизнес-уровню для получения от последнего необходимой информации. Впоследствии мы убедимся, что наличие бизнес-уровня обеспечивает чрезвычайную гибкость и удобство. А сейчас мы используем один из таких невизуальных компонентов для поддержки нашего интерфейса. Перейдем па вкладку BoldHandles и поместим на форму компонент-дескриптор списка BoldListHandle1. В инспекторе объектов назначим свойству RootHandle этого компонента значение BoldSystemHandle1 (его можно ввести вручную или выбрать из раскрывающегося списка).

Свойство Expression дескриптора списка содержит текстовое представление OCL-выражения (см. главу 5). Введем следующее OCL-выражение ~Author.allinstances, как

показано на рис. 3.10. То есть дескриптор списка в нашем случае сформирует «OCL-запрос» к бизнес-уровню, который в данном случае означает «хочу получить все экземпляры класса Author».

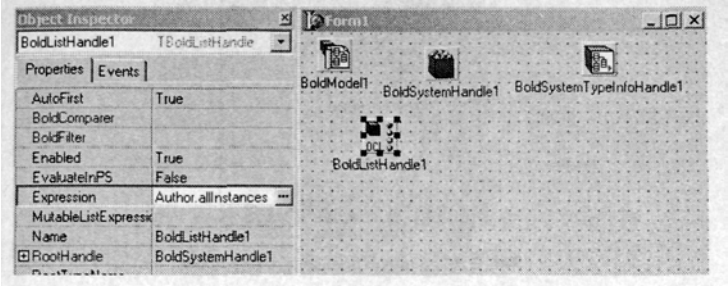


Рис. 3.10. Настройка дескриптора списка

Теперь создадим собственно графический интерфейс, для этого на вкладке BoldControls имеется достаточное количество визуальных компонентов. Мы выберем два из них — BoldGrid и Bold Navigator, которые, как легко понять, являются аналогами соответствующих компонентов DBGrid и DBNavigator, используемых при создании приложений баз данных в Delphi традиционным способом. Настроим визуальные компоненты следующим образом — для обоих в инспекторе объектов установим источник информации — свойству BoldHandle присвоим значение BoldListHandlel, после этого щелкнем правой кнопкой мыши на компоненте BoldGridl и выберем пункт Create Default Columns — то есть «создать столбцы по умолчанию». В результате у компонента BoldGridl автоматически отобразится заголовок столбца aname (рис. 3.11).

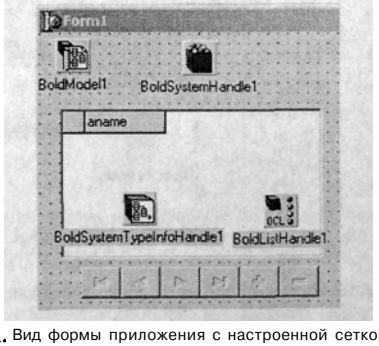


Рис. 3.11. Вид формы приложения с настроенной сеткой BoldGrid

Все. Мы создали наше первое очень простое приложение и можем отправить его на выполнение. Пользуясь кнопками навигатора, можно добавить несколько авторов (рис. 3.12), удалять или редактировать их. Но, как легко убедиться, после выхода из приложения и повторного его запуска все введенные данные исчезают.

Это происходит потому, что наше приложение пока не содержит уровня данных (Persistence Layer).

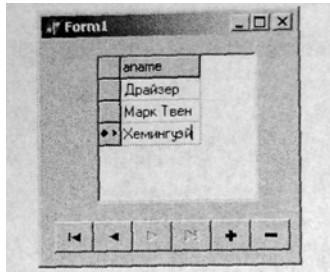


Рис. 3.12. Вид работающего приложения

Создание уровня данных

Чтобы не отвлекаться сейчас на создание базы данных, настройку псевдонимов и тому подобных вещей, для построения уровня данных используем очень удобную возможность, предоставляемую средой Bold for Delphi, — хранение данных в XML-документе. Для этого с вкладки BoldPersistence поместим на форму компонент — файловый дескриптор XML-данных BoldPersistenceHandleFileXML1.

Для настройки компонента в инспекторе объектов назначим его свойству BoldModel значение BoldModel1, а в свойстве FileName введем имя файла, например 1.xml, как показано на рис. 3.13.

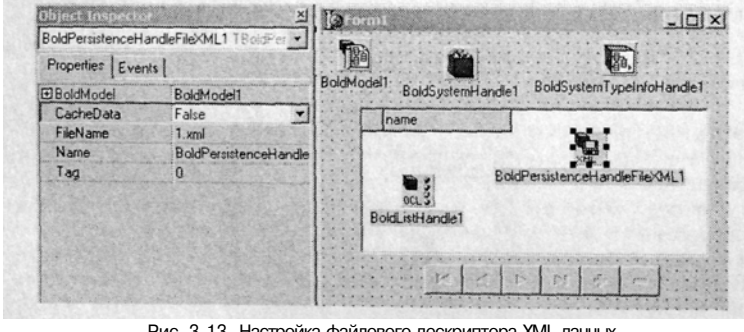


Рис. 3.13. Настройка файлового дескриптора XML-данных

Кроме того, для компонента BoldSystemHandle1 присвоим в инспекторе объектов свойству PersistenceHandle имя вновь введенного компонента BoldPersistenceHandleFileXML1 (рис. 3.14).

Чтобы наше приложение сохраняло свои данные, в процедуру обработки события OnClose нашей формы введем строку кода:

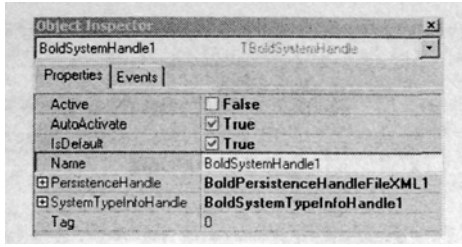


Рис. 3.14. Настройка свойств привязки к уровню данных

```
procedure TForm1.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
  BoldSystemHandle1.UpdateDatabase;
end;
```

Теперь, как легко убедиться, наше приложение приобрело способность сохранять данные, и можно подробнее изучить его возможности.

Работа с приложением

На первый взгляд, пока ничего особенного мы не получили. Создали, пусть и не совсем традиционным способом, но вполне обычную форму для ввода и редактирования данных по авторам. Кстати, почему-то при этом не сделали форму ввода для названий книг — вправе отметить внимательный читатель. Но первое впечатление в данном случае обманчиво. Чтобы начать знакомство с «магией» Borland MDA, давайте добавим к нашему приложению один программный модуль — допишем в объявление Uses нашего модуля Unit1 модуль с названием BoldAFPDefault, и запустим приложение.

Добавим несколько авторов, как показано на рис. 3.12. Теперь выберем автора Драйзер в сетке и дважды щелкнем по этой строке. При этом автоматически откроется новое окно. Мы не создавали эту форму, за нас это сделала среда Borland M DA. Форма имеет несколько вкладок, одна из которых называется writes. Перейдя на нее, мы обнаруживаем форму для ввода названий книг. Если, не закрывая новой формы, перейти на другого автора и опять дважды щелкнуть на его имени, то появится еще одна новая форма для ввода книг, написанных этим автором, и т. д. Таким образом, мы можем занести названия книг всех трех авторов (рис. 3.15).

Отметим, что созданные автоматически формы (автоформы) ведут себя абсолютно независимо от положения курсора на главной форме, и мы можем редактировать книги каждого автора совершенно автономно. Также интересно убедиться, что можно «перетаскивать» книги от одного автора к другому, если перетаскивать мышью не название книги (второй столбец), а серую ячейку первого столбца на форму другого автора. Для такого «перетаскивания» можно использовать и область на автоформе, помеченную изображением стрелки с точкой (см. рис. 3.15).

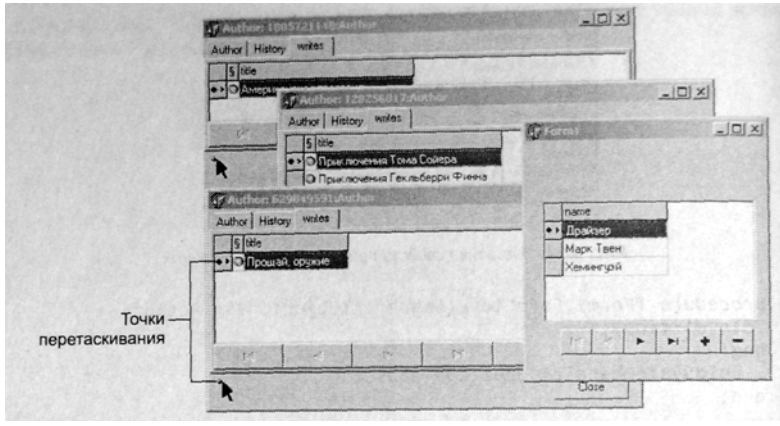


Рис. 3.15. Вид приложения с автоформами

Внимательный читатель без труда обнаружит, что название writes вкладки на автоформе есть не что иное как название соответствующей роли ассоциации в нашей модели. А где же, в таком случае, название второй роли? Для того чтобы увидеть его, выберем какую-нибудь книгу на автоформе и дважды щелкнем на ее названии (рис. 3.16).

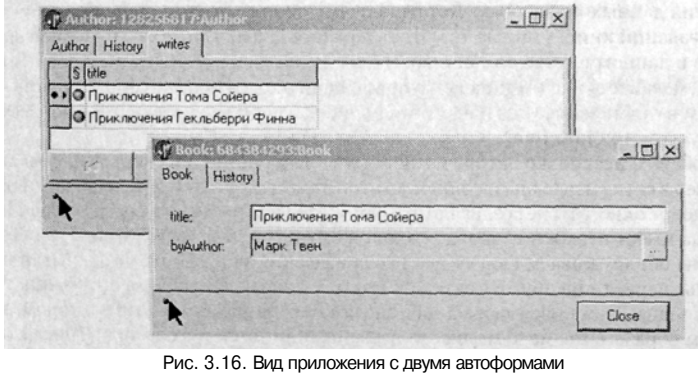


Рис. 3.16. Вид приложения с двумя автоформами

При этом появится новая автоформа, на которой мы увидим название книги, и под ней серое поле с именем автора, а слева от него заголовок byAuthor — это и есть вторая роль нашей ассоциации.

Учитывая, что мы практически не написали пока ни одной строки кода, не программировали события «drag&drop» мыши, не создавали форму для ввода названий книг, можно констатировать, что полученное приложение обладает совсем неплохой функциональностью.

Чтобы видеть более полную картинку происходящего, добавим на главную форму визуальные компоненты BoldGrid2 и BoldNavigator2 из вкладки BoldControls для отображения и управления списком книг. Для получения списка книг из бизнес-уровня нам понадобится также второй компонент — дескриптор списка BoldList-Handle2 из вкладки BoldHandles. Настроим его следующим образом.

В инспекторе объектов назначим свойству RootHandle этого компонента значение BoldSystemHandlel (его можно ввести вручную или выбрать из раскрывающегося списка).

Для свойства Expression введем OCL-выражение Book.allinstances (рис. 3.17).

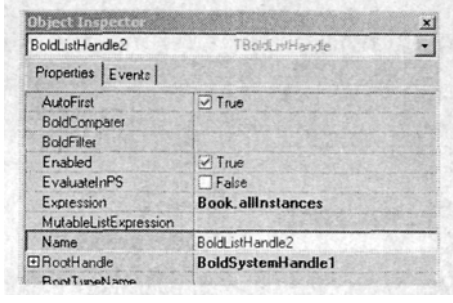


Рис. 3.17. Настройка второго дескриптора списка

Настроим визуальные компоненты BoldGrid2 и BoldNavigator2 следующим образом: для обоих в инспекторе объектов установим источник информации — свойству BoldHandle присвоим значение BoldListHandle2. После этого щелкнем правой кнопкой мыши на компоненте BoldGrid2 и выберем пункт Create Default Columns (создать столбцы по умолчанию). В результате у компонента BoldGrid2 появится заголовок столбца btitle (рис. 3.18).

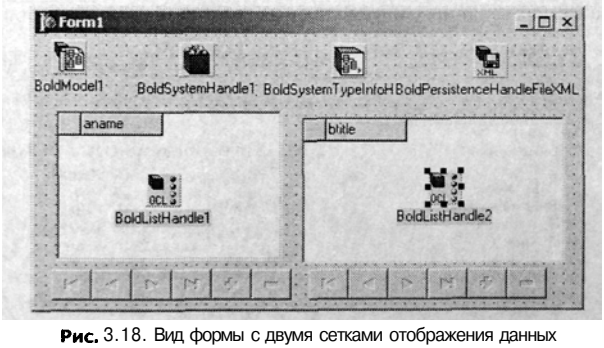


Рис. 3.18. Вид формы с двумя сетками отображения данных

Видно, что названия столбцов aname и btitle в сетках BoldGridl и BoldGrid2 среда Bold автоматически выбирает из назначенных нами атрибутов н классах Author и Book. Это происходит, когда мы создаем столбцы по умолчанию. Но это, есте-

ственно не означает, что мы не можем настроить таблицы BoldGrid по-другому (например, присвоить им русскоязычные названия, см. главу 9). просто сеичас мы выбираем наиболее легкий и быстрый путь.

Запустим наше усовершенствованное приложение. Добавим с помощью навигатора двух авторов: Ильф и Петров. Добавим с помощью второго навигатора книгу 12 стульев. Щелкнем дважды на названии этой книги (рис. 3.19).

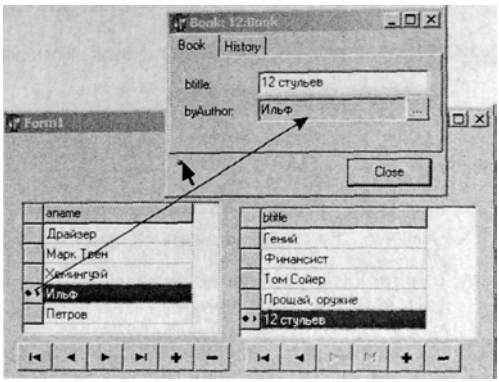


Рис. 3.19. Работающее приложение с двумя сетками и автоформой

Перетащим мышью автора Ильф из сетки на автоформу, на серое поле с именем автора (стрелка на рис. 3.19). Мы увидим, что книге 12 стульев присвоился автор Ильф. Перетащим автора Петров на то же место, и увидим, что автором книги стал Петров, а Ильф исчез. Мы понимаем, что у данной книги в действительности два автора - Ильф и Петров. Но при создании модели нами было введено бизнес-правило - у каждой книги может быть только один автор, и наше приложение, функционируя в рамках заданной модели, не позволяет нам добавить второго.

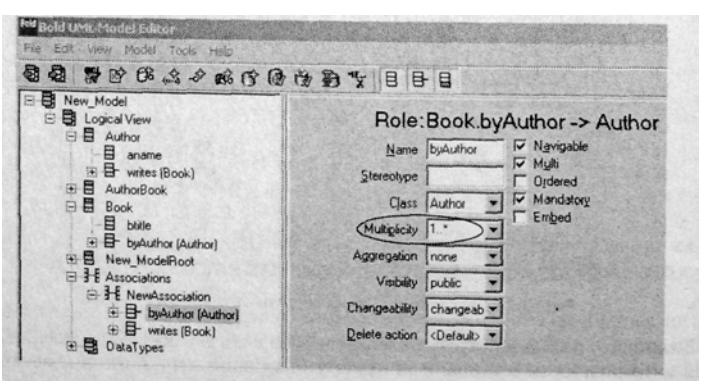


Рис. 3.20. Изменение кратности роли в UML-редакторе

Модификация модели приложения

Давайте посмотрим, что произойдет, если мы удалим ограничение на количество авторов у книги. Вместо этого сформулируем следующее бизнес-правило — «каждая книга может иметь одного или нескольких авторов». Для этого откроем редактор моделей, выберем роль byAuthor и установим ее свойство Multiplicity (кратность отношения) равным 1..* (рис. 3.20).

Запустим приложение. И... получим сообщение об ошибке (рис. 3.21).

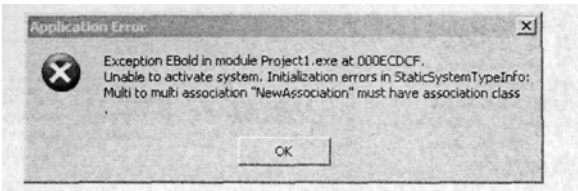


Рис. 3.21. Run-time-сообщение о необходимости наличия класса-ассоциации

Если же мы до запуска приложения попытаемся сохранить нашу форму с измененной моделью, то также получим сообщение об ошибке (рис. 3.22).

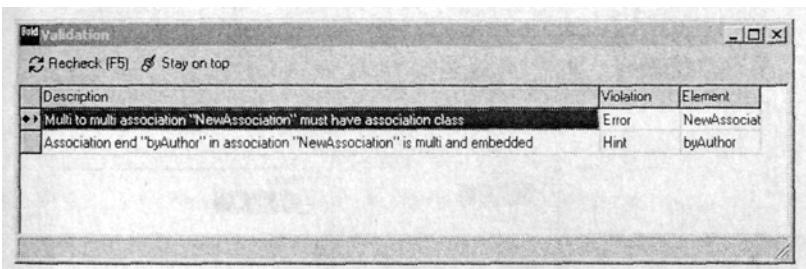


Рис. 3.22. Design-time-сообщение о необходимости наличия класса-ассоциации

Оба сообщения говорят о необходимости наличия в модели класса-ассоциации, обеспечивающего хранение информации о связи «многие-ко-многим». В самом деле, если проводить аналогию с реляционными базами данных, то для случаев подобных связей необходимо создавать дополнительную связующую таблицу. В нашем случае среда Bold for Delphi позволяет создать такой класс автоматически. Для этого в главном меню встроенного UML-редактора выберем команду Tools ▸ UnBoldify Model и сразу после этого — Tools ▸ Boldify Model. В результате в дереве модели появится новый класс (рис. 3.23) с названием NewAssociation.

При этом если выбрать в дереве модели ассоциацию, то справа в окошке Class мы увидим имя нового автоматически созданного класса (см. рис. 3.23).

Сохраним наш проект, удалим файл 1.xml и снова запустим приложение. Добавим тех же двух авторов с помощью первого навигатора. Добавим с помощью второго навигатора книгу 12 стульев.

Щелкнем дважды на названии книги. И мы увидим, что автоформа изменилась (рис. 3.24).

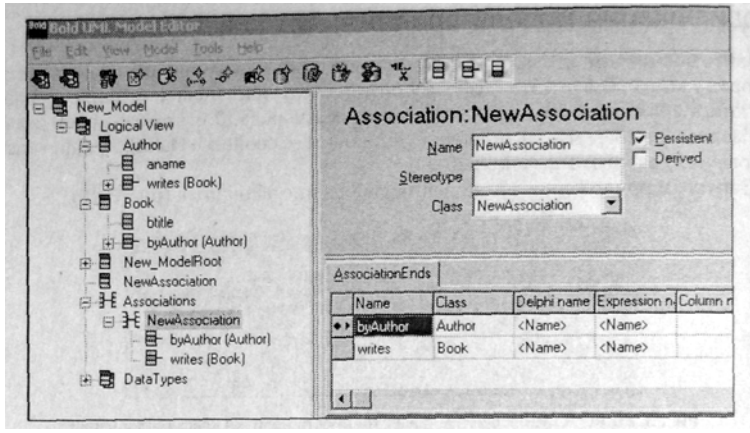


Рис. 3.23. Вид модели с автоматически созданным классом-ассоциацией

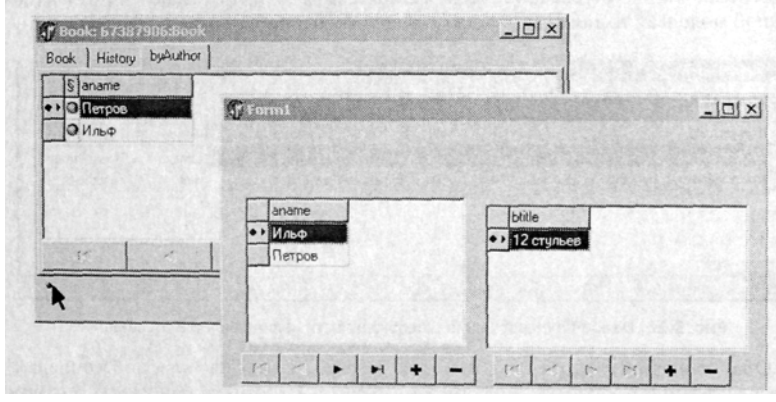


Рис. 3.24. Новый вид автоформы для ввода нескольких авторов

Теперь вместо одного окошка для имени автора мы обнаруживаем форму, позволяющую к данной книге назначать произвольное множество авторов. Перетащим последовательно обоих авторов на автоформу и получим, что теперь у книги 12 стульев оба автора имеются в наличии (см. рис. 3.24).

Только что мы с вами пронаблюдали на практике проявление характерной и основной черты MDA-приложений, а именно - поведение MDA-приложения определяется не кодом приложения, а UML-моделью. Ведь мы не исправили ни строчки кода (да и кода-то, строго говоря, нет, по крайней мере, написанного нами), а получили новую функциональность нашей программы. Почему? Потому что мы изменили модель приложения, и этого оказалось вполне достаточно для изменения его поведения.

Продолжим изучение свойств созданного приложения. Добавим нового автора Драйзер. Добавим во вторую таблицу книги: Американская трагедия, Гений, Финансист. Щелкнем дважды на авторе Драйзер, откроем вкладку writes на появившейся автоформе автора, после чего перетащим мышью эти книги из главной формы на автоформу автора (рис. 3.25). Еще раз напомним, что при перетаскивании нужно помещать курсор не на названии книги или автора, а на ячейку первого столбца, выделенную символами ромбика и стрелки. Эти символы показывают текущее положение указателя.

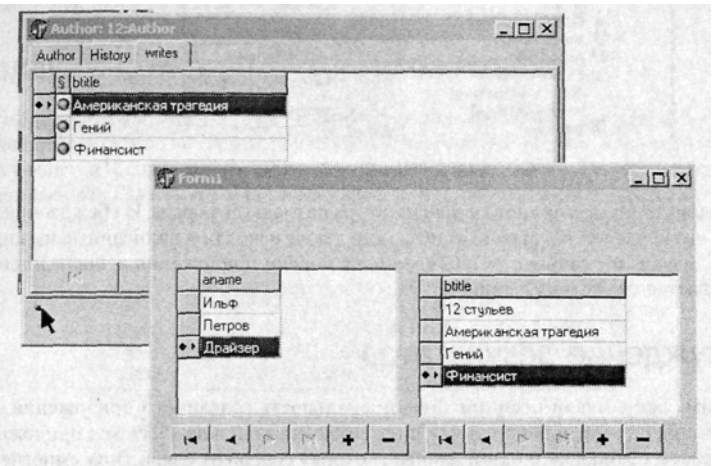


Рис. 3.25. Добавление нескольких авторов на автоформу

А теперь, когда автору Драйзер мы присвоили эти книги, с помощью первого навигатора удалим самого автора. При этом возникнет окно с требованием подтверждения операции удаления, после подтверждения Драйзер исчезнет из списка авторов, автоформа автора при этом автоматически закроется. Но его книги попрежнему будут фигурировать в списке на главной форме. Такое поведение приложения не всегда является допустимым. Автора нет, а его книги присутствуют. То есть налицо некоторое «нарушение целостности данных», как это принято говорить при работе с СУБД.

Попробуем изменить ситуацию. Сформулируем измененное ранее бизнес-правило немного по-другому — «каждая книга должна быть написана по крайней мере одним автором». Тем самым мы говорим, что не должно существовать книг, у которых отсутствует автор.

Для реализации этого правила снова обратимся к нашей модели, выберем роль ассоциации writes и изменим значение параметра Delete action с <Default> на Cascade (рис. 3.26).

Снова запустим наше приложение. Добавим автора Драйзер, которого мы удалили в прошлый раз, и назначим ему уже известным способом три книги: Американская трагедия, Гений и Финансист.

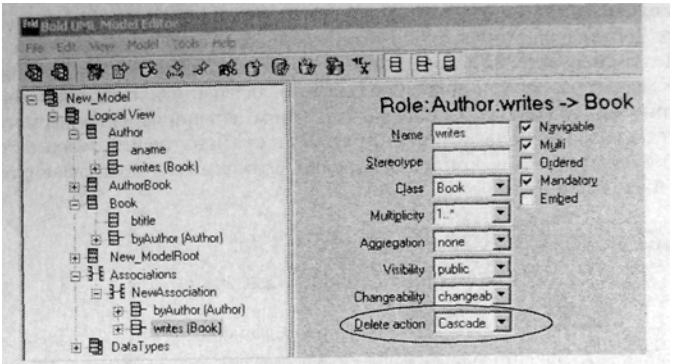


Рис. 3.26. Изменение свойств UML-модели

Теперь попытаемся снова удалить автора из главной формы. И убеждаемся, что после его удаления из правого списка книг также исчезли и написанные им книги. Опять мы с вами увидели, как изменение в модели приложения непосредственно сказывается на его поведении.

Обсуждение результатов

Оценим, уже с новой позиции, функциональность созданного приложения. Мы построили, с точки зрения прикладных возможностей, классическое приложение для работы с локальной базой данных, которая содержит связи типа «многие-ко-многим». При этом:

- не создавалась собственно база данных, а создавалась UML-модель;
- не создавались таблицы и поля, первичные и вторичные ключи и индексы, а создавались классы и атрибуты, ассоциации и роли;
- не создавались связи типа «мастер—подчиненный» и «многие-ко-многим», а назначались размерности ролей (кратности отношений);
- не создавались промежуточные связующие таблицы для реализации отношений «многие-ко-многим»;
- не использовался язык SQL;
- не программировались формы для ввода и редактирования данных, они были созданы автоматически;
- не программировался интерфейс «drag&drop».

Читатели, знакомые с разработкой приложений баз данных в Delphi, могут, наверное, реально оценить затраты времени на создание подобного приложения традиционным способом. А также оценить объем работы при внесении изменений в структуру базы данных, что мы, по сути, и делали, когда изменяли размерности связей для авторов в нашей модели.

К этому можно добавить, что даже такое простое приложение обладает достаточной гибкостью, не требуя при этом, как мы увидели, написания программного кода. Наше приложение вообще не содержит программного кода (имеется в виду, конечно, пользовательский программный код).

Также стоит отметить, что мы пошли не оптимальным путем, когда использовали встроенный в Bold редактор моделей. Он, к сожалению, не является графическим. Как будет видно в дальнейшем, использование графических UML-редакторов может существенно облегчить и еще более ускорить работу по созданию и модификации модели приложений.

Создание стандартных приложений

В приведенном выше примере мы создавали приложение «вручную». Однако, после установки пакета Bold for Delphi, появляется возможность некоторой автоматизации этого процесса. Если в главном меню интегрированной среды Delphi последовательно выбрать File • New • Other • BoldSoft и в открывшемся окне выбрать (единственный) пункт с названием Default Bold Application (рис. 3.27), то будет автоматически сгенерирована заготовка для нового Bold-приложения.

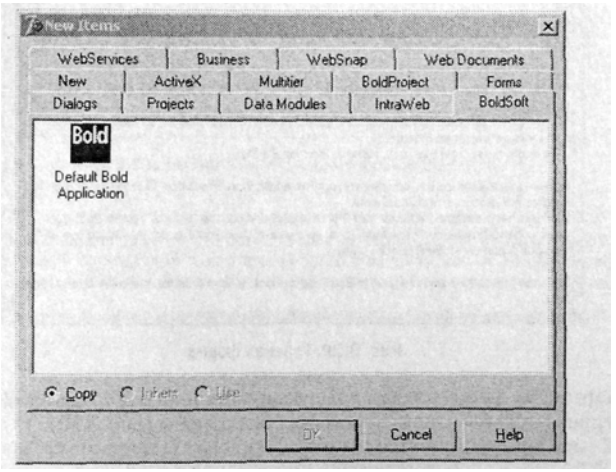


Рис. 3.27. Создание стандартного приложения

Эта заготовка содержит три формы:

- модуль данных dmMain, он включает набор Bold-компонентов для создания объектного пространства, компонент для связи с RationalRose, адаптер СУБД Interbase, а также стандартные Delphi-компоненты ActionList1 и IBDatabase1 (рис. 3.28);
- главную форму frmMain, она содержит главное меню, а также текстовое поле с инструкциями (на английском языке) по дальнейшим действиям разработчика (рис. 3.29);

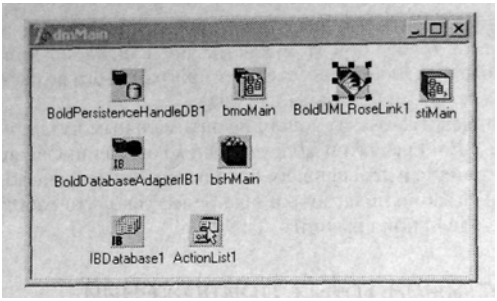


Рис. 3.28. Модуль данных, созданный автоматически

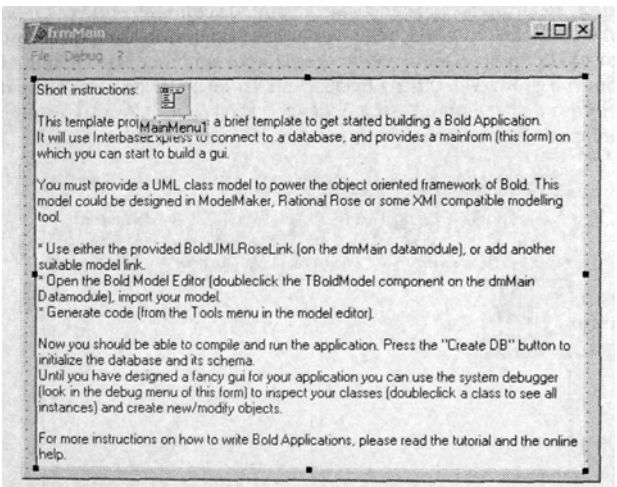


Рис. 3.29. Главная форма

«стартовую» форму frmStart, содержащую кнопки для создания базы данных, активизации объектного пространства и отмены (рис. 3.30).

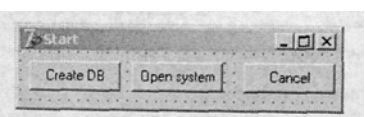


Рис. 3.30. Стартовая форма

Применение указанного способа создания стандартных «заготовок» приложений весьма удобно, однако требует наличия дополнительных знаний для дальнейшей настройки компонентов и разработки собственно приложения. Далее в этой

книге будут даны все необходимые сведения. Сейчас лишь дополнительно отметим, что при необходимости можно использовать и способ добавления таких стандартных «заготовок» по отдельности, то есть не всего приложения в целом, а конкретной формы или модуля данных. Для этого в меню Delphi нужно последовательно выбрать File • New • Other • BoldProject и в открывшемся окне выбрать компонент, который требуется добавить в проект (рис. 3.31).

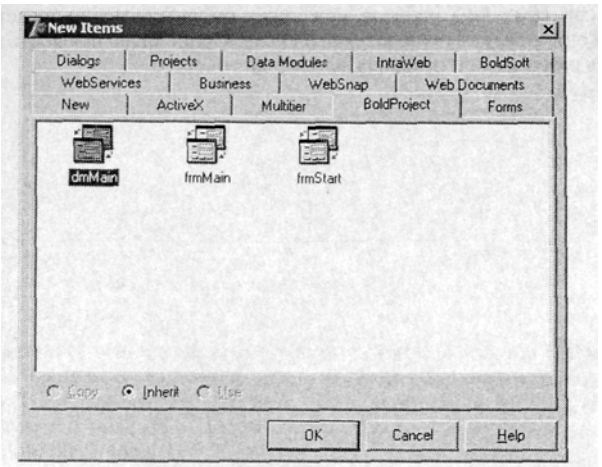


Рис. 3.31. Добавление отдельных форм в Boil-проект

ПРИМЕЧАНИЕ

Вкладка BoldProject для раздельного добавления стандартных форм становится доступна только после создания стандартной «заготовки» нового Bold-приложения. То есть, если был начат «обычный» проект в Delphi, эта вкладка отображаться не будет и воспользоваться ею будет невозможно.

Резюме

На примере простого приложения продемонстрировано использование MDA-инструментария.

- На практике показано, что MDA-приложение организовано в виде «трехслойного пирога», и даже простейшее такое приложение содержит три уровня: уровень данных, бизнес-уровень и прикладной уровень (графический интерфейс).
- Продемонстрировано, что графический интерфейс обращается к данным через посредника - бизнес-уровень, для чего существуют специальные невизуальные компоненты.

- Показано, что функционирование MDA-приложенин невозможно без создания модели. И, с другой стороны, поведение приложения определяется этой моделью не абстрактно, а совершенно конкретно.
- Даже на таком простом приложении продемонстрированы преимущества использования MDA-технологии. Экстраполируя рассмотренный простой пример на корпоративные информационные системы, содержащие сотни классов и тысячи атрибутов, можно уже сейчас примерно представить, почему описываемая технология позволяет кардинально повысить эффективность разработки и сопровождения таких систем.

Модель приложения

Как уже говорилось ранее, модель приложения является основой MDA-технологии. Модель содержит состав, структуру и элементы поведения разрабатываемого приложения. В этой главе мы рассмотрим практическую работу с UML-моделями, а также научимся использовать CASE-систему Rational Rose совместно с инструментарием Bold for Delphi.

Роль модели в Borland MDA

С точки зрения Bold for Delphi модель имеет следующее функциональное назначение:

- определяет состав и тип классов и атрибутов классов;
- фиксирует наличие связей между классами, определяет тип этих связей и их размерность;
- определяет условия и ограничения, накладываемые на классы и их атрибуты: такие условия формируются на диалекте языка OCL, встроенного в Bold;
- содержит информацию для адаптации к среде разработки Delphi в виде набора специальных *тег-параметров*;
- содержит информацию для адаптации к СУБД, также в виде набора специальных тег-параметров.

В Bold, таким образом, отсутствует явное четкое разделение между PIM и PSM-моделями MDA (см. главу 1), и вся необходимая информация содержится в одной общей модели. Функции PSM-модели в основном выполняет набор тег-параметров (*tagged values),* которые будут описаны далее в этой главе. Кроме того, такие функции могут выполнять и некоторые компоненты Bold, являющиеся адаптерами СУБД (см. главу 10).

Bold for Delphi использует модель для реализации следующих основных функций.

Генерация кода. Под генерацией кода понимается автоматическое создание модулей описаний (.inc-файлов) и функциональных модулей (.pas-файлов) на языке Object Pascal, содержащих полную реализацию классов модели приложения (подробнее см. главу 12). Строго говоря, сам по себе Bold не нуждается в генерации кода. Как мы уже увидели на примере простого приложения (см. главу 3) и увидим в дальнейшем, можно создавать достаточно серьезные приложения, не используя генерации кода вообще. Однако в некоторых случаях она бывает необходимой. Общая стратегия здесь выглядит примерно так — если поддерживаемых Bold возможностей языка OCL недостаточно для реализации, например, желаемых методов или вычисляемых атрибутов на бизнес-уровне, то разработчик вправе «опуститься» на уровень программного кода. Кроме того, генерация кода необходима, если классы модели содержат *операции*. Ее можно произвести из встроенного редактора модели (Model Editor) на этапе разработки приложения. Кроме генерации кода Bold обеспечивает и возможность генерации интерфейсов, необходимых для функционирования распределенных DCOM-приложений.

Генерация схемы базы данных. Структура базы данных (то есть набор описаний таблиц, полей, индексов и ключей) может быть сгенерирована автоматически как из встроенного редактора модели на этапе разработки, так и программно во время выполнения приложения. Bold также поддерживает синхронизацию структуры базы данных с изменяющейся во времени моделью приложения *(Model Evolution* — эволюция модели), при этом сохраняется уже имеющаяся в БД информация. Такая возможность в Bold носит специальное название — эволюция базы данных *(DataBase Evolution)*. Необходимо иметь в виду, что между составом классов модели и составом таблиц генерируемой базы данных нет взаимнооднозначного соответствия. Во-первых, ряд таблиц БД Bold использует для своих собственных нужд и создает их автоматически. Во-вторых, не все классы модели требуют сохранения информации объектов в БД (persistent class), а могут быть временными (transient) классами; кроме того, даже в persistent-классах модели часто присутствуют вычисляемые (derived) атрибуты, которые не сохраняются в базе данных. И в-третьих, в ряде случаев принципиально невозможно в реляционной БД реализовать некоторые виды отношений. Так, если в UML-модели вполне допустимо соединять два класса ассоциацией, имеющей на обоих концах кратности большие 1, то в реляционной базе данных для реализации такого вида отношений («многие-ко-многим») требуется создание промежуточной связующей таблицы. Поэтому при генерации схемы базы данных в подобных случаях Bold «самостоятельно» и незаметно для пользователя способен генерировать такие таблицы.

Интерпретация модели во время выполнения для управления поведением приложения. Вся информация о модели сохраняется в компоненте TBoldModel и на этапе выполнения приложения используется для управления объектным пространством, для контроля его целостности, а также для управления взаимодействием бизнес-уровня с уровнем данных и графическим интерфейсом. Можно сказать, что **объектное пространство (Object Space) в Borland MDA является экземпляром модели,** по аналогии с тем, как объект является экземпляром класса в ООП. Управление каждым объектным пространством обеспечивается компонентом TBoldSystemHandle. Информацию о типах, содержащихся в модели, этот компонент получает из компонента TBoldSystemTypeInfoHandle.

Тег-параметры (tagged values)

Кроме UML-модели, BMDA для своего функционирования использует набор специальных переменных-параметров (tagged values), или тег-параметров. Они необходимы для взаимодействия со средой разработки и СУБД, а также для дополнительных настроек модели.

Появление тег-параметров не случайно. Если UML-модель можно рассматривать как платформенно-независимую PIM-модель (см. главу 1), то совокупность тег-параметров можно рассматривать в качестве платформенно-зависимой PSM-модели. Будучи по этой причине связанными с PIM, тег-параметры классифицируются по принадлежности к элементам иерархической структуры модели. Таким образом, существуют отдельные наборы тег-параметров для следующих элементов иерархии модели:

- модель в целом;
- класс;
- ассоциация;
- атрибут;
- роль;
- операция.

С другой стороны, тег-параметры можно классифицировать и по функциональной принадлежности:

- общие;
- используемые для генерации кода и интерфейсов;
- используемые для отображения на уровень данных (Persistence Mapping);
- используемые для описания принадлежности к уровню данных (Persistence).

Разработчик также может создавать собственные тег-параметры и использовать их для своих целей. Значения тег-параметров доступны как на этапе разработки приложения, так и во время его выполнения. Borland MDA описываемой версии содержит несколько десятков тег-параметров. Мы подробнее познакомимся с ними при описании технологии разработки моделей приложений.

Rational Rose как средство разработки моделей для Borland MDA

В предыдущем разделе были рассмотрены основные функции модели приложения в Borland MDA, а также кратко описаны элементы диаграммы классов UML. Настоящая глава посвящена практической разработке модели MDA-приложения в графических UML-редакторах на примере CASE-системы Rational Rose. Следует отметить, что после включения в версию 4 Bold for Delphi полной поддержки функций импорта и экспорта модели в формате XMI (XML Metadata Interchange — язык обмена метаданными XML) в принципе появилась возможность разрабатывать модели приложений для Borland MDA в любом UML-редакторе, поддержи-

вающем этот формат (например, в PowerDesigner компании Sybase). Кроме того, в состав Delphi 7 Studio входит инструмент ModelMaker, также включающий в себя развитые средства UML-моделирования и некоторые средства интеграции с Bold. Тем не менее, с большой долей уверенности можно утверждать, что на настоящий момент именно Rational Rose остается наиболее удобным средством разработки UML-моделей для Borland MDA. Дело в том. что в качестве инструмента создания модели для Bold CASE-система Rational Rose занимает особое место среди программных средств, обладающих графическим UML-редактором. Взаимодействие с Rational Rose заложено в Borland MDA начиная с ранних версий продукта Bold for Delphi, и заложено достаточно основательно. Это взаимодействие реализуется посредством технологии COM (Component Object Model — модель компонентных объектов). Подробное описание COM приводится во многих источниках, например в [3].

С точки зрения COM, Rational Rose является *сервером автоматизации,* выполняющим запросы клиента (*контроллера*)— среды разработки Borland MDA. Благодаря такому тесному механизму взаимодействия обеспечиваются следующие полезные функциональные возможности:

- автоматический запуск Rational Rose по запросу из среды Delphi;
- импорт UML-моделей и тег-параметров из Rational Rose в Bold;
- экспорт UML-моделей и тег-параметров из Bold в Rational Rose;
- доступ к тег-параметрам Bold при разработке модели в Rational Rose;
- адаптация Rational Rose к конкретным версиям Bold.

Перечисленные функции позволяют объединить на практике удобные выразительные средства графического интерфейса Rational Rose с возможностью реализации тонкой настройки модели приложения в среде Borland MDA.

Настройка Rational Rose

Перед созданием модели в редакторе Rational Rose необходимо обеспечить его предварительную настройку для работы с Bold for Delphi. Для начала следует убедиться в наличии Bold for Delphi в составе активных расширений (Add-In) Rational Rose. Для этого командой Add-Ins • Add-In Manager... нужно вызвать окно менеджера расширений (рис. 4.1) и, при необходимости, активизировать пункт Bold for Delphi.

Для настройки под конкретную версию Bold for Delphi используется специальный файл свойств (Properties Files) с названием BfD.pty, который создается при инсталляции Bold for Delphi в папке ...\Program Files\Boldsoft\BfDR40D7Arch\Rose\.

Для его использования в Rational Rose нужно с помощью команды Tools > Model Properties ▸ Update... выбрать вышеуказанный файл. После этого необходимо вызвать окно настроек модели (Tools • Model Properties У Edit), перейти на вкладку Bold, выбрать из раскрывающегося списка параметр Project и установить для свойства PTYVersion (самое нижнее в списке) новое значение (рис. 4.2, номер версии модели 6.4 соответствует последнему обновлению продукта).

На этом этап настройки Rational Rose закончен. При каждом обновлении версии Bold for Delphi эту настройку следует повторять.

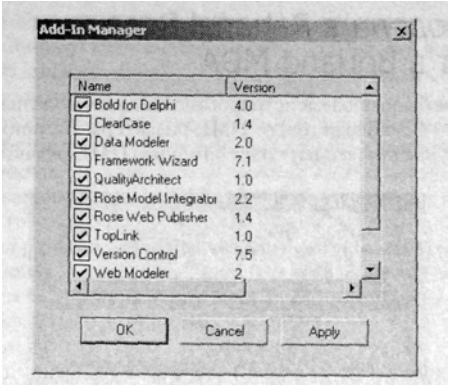


Рис. 4.1. Настройка активных расширений Rational Rose

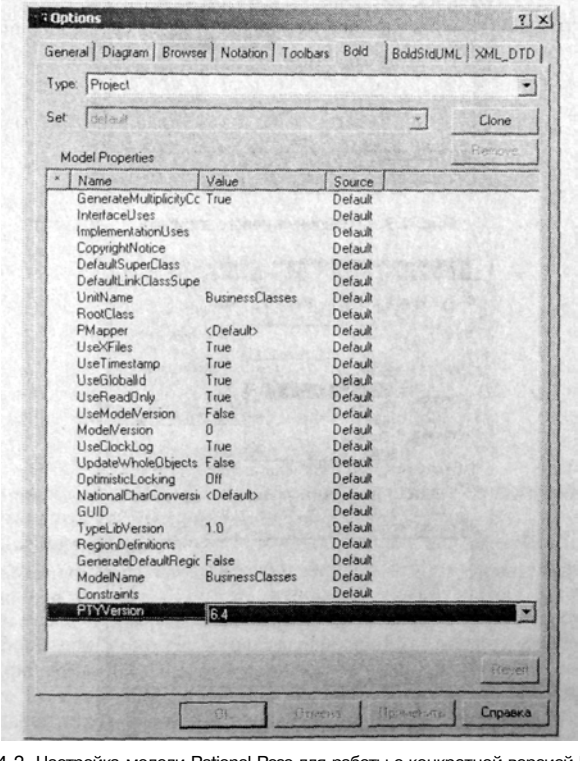


Рис 4.2. Настройка модели Rational Rose для работы с конкретной версией BMDA

Создание модели в Rational Rose и ее импорт в Borland MDA

В предыдущей главе было продемонстрировано создание модели приложения средствами встроенного в Bold текстового UML-редактора. На примере аналогичной модели посмотрим, как это реализуется с помощью UML-редактора Rational Rose.

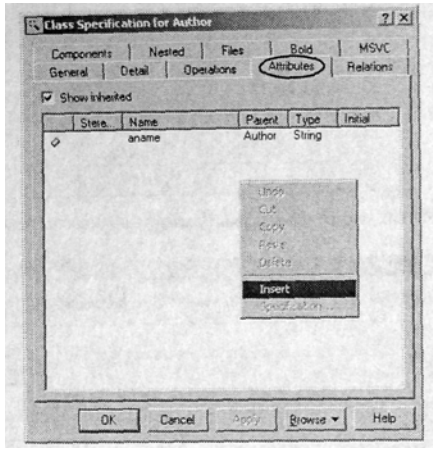


Рис. 4.3. Добавление нового атрибута

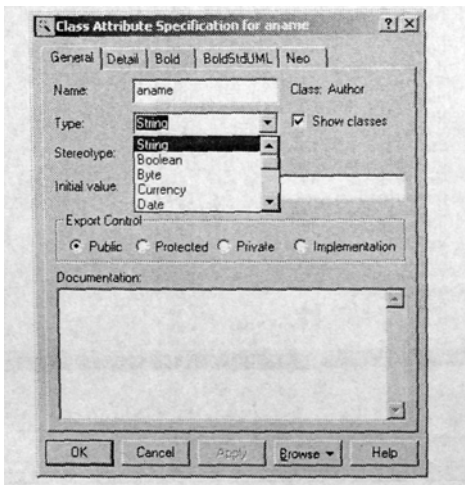


Рис. 4.4. Настройка атрибута

Запустим приложение Rational Rose, добавим в модель два класса — Author и Book (основы работы в Rational Rose описаны в главе 1). Добавим в класс Author новый атрибут. Для этого, дважды щелкнув на изображении класса, войдем в окно его настройки, и выберем вкладку Attributes (рис. 4.3).

Вызвав правой кнопкой мыши контекстное меню, выберем пункт Insert, после чего в списке атрибутов появится новая запись с именем атрибута по умолчанию — name. Щелкнув дважды по этой записи, войдем в окно настройки атрибута, где присвоим ему название aname и зададим строковый тип String, используя раскрывающийся список (рис. 4.4).

Аналогичным образом в класс Book добавим строковый атрибут btitle.

Соединим классы ассоциацией и, щелкнув дважды по линии ассоциации, перейдем в окно ее настроек. На вкладке General введем названия ролей нашей ассоциации — writes и byAuthor, как показано на рис. 4.5.

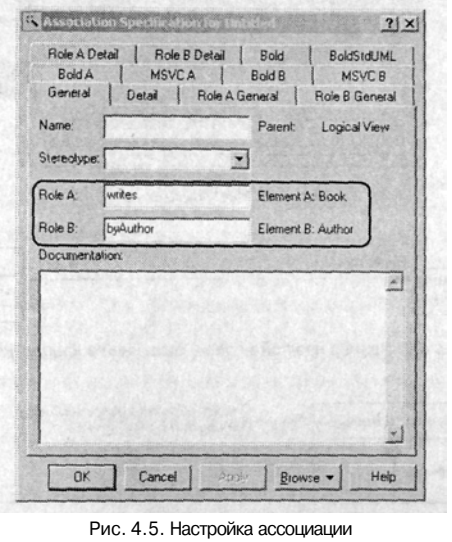


Рис. 4.5. Настройка ассоциации

Далее перейдем на вкладку Role A Detail для настройки первой роли ассоциации. Зададим кратность этой роли 1..n (рис. 4.6).

На вкладке Role B Detail аналогично настроим вторую роль. В результате мы получим простую модель, изображенную на рис. 4.7. Сохраним созданную модель в файле, например lib.mdl.

Перейдем в среду Delphi и создадим простой проект, состоящий из одной формы. Со вкладки BoldHandles поместим на форму компоненты BoldModel1, BoldSystemHandle1, BoldSystemTypeInfoHandle1 и настроим их аналогично разобранному ранее простому приложению. Со вкладки BoldMisc поместим на форму компонент BoldUMLRoseLink — он предназначен для связи с моделью RationalRose — и установим в его свойстве filename имя файла сохраненной нами модели — lib.mdl (рис. 4.8).

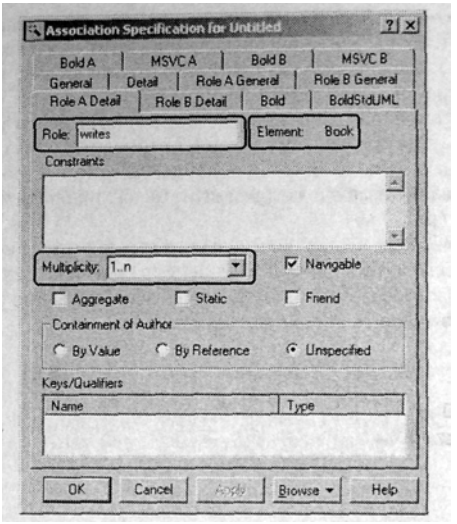


Рис. 4.6. Настройка роли

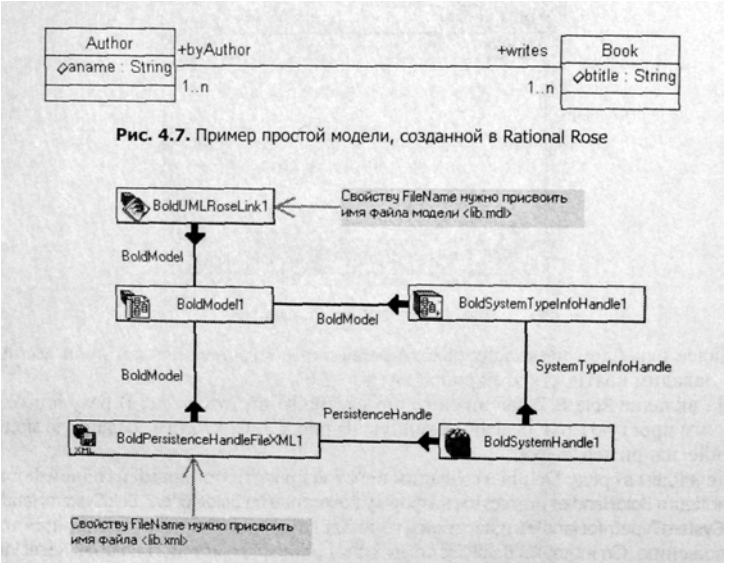


Рис. 4.8. Настройка Bold-компонентое приложения

Теперь все готово для импорта модели Rational Rose в среду Borland MDA. Для этого, щелкнув дважды по компоненту BoldModell, откроем встроенный UML-редактор Bold и запустим импорт, нажав на второй справа значок со стрелкой в панели инструментов (рис. 4.9).

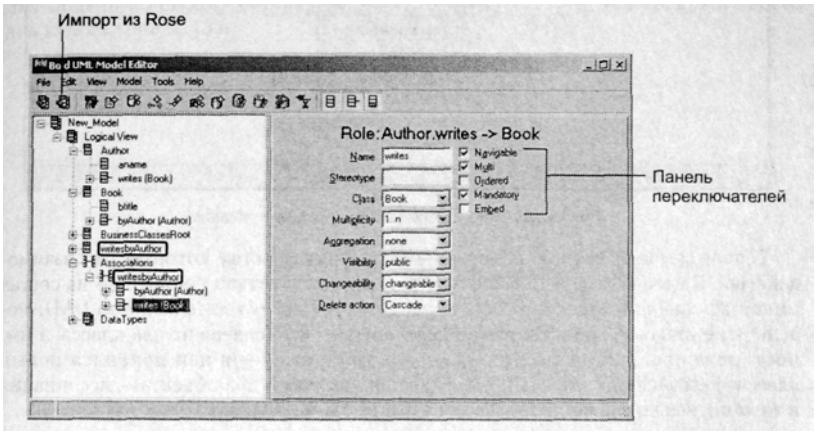


Рис. 4.9. Импорт модели Rational Rose в среду Delphi

Возникнет окно-предупреждение с предложением подтвердить импорт (рис. 4.10). Предупреждение связано с тем, что в результате операции импорта текущая модель будет заменена новой, поэтому информация о текущей модели может быть утеряна. При необходимости эту модель следует сохранить заранее либо при появлении данного окна-предупреждения, отказавшись от импорта.

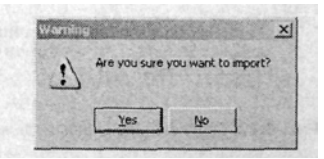


Рис. 4.10. Предупреждение об импорте модели

После подтверждения будет произведен импорт из Rational Rose в среду Borland MDA. При этом, если программа Rational Rose не была предварительно запущена, произойдет ее автоматический запуск. Операция импорта для больших моделей, содержащих несколько сотен классов, может занимать несколько минут. Ее ход можно наблюдать в строке состояния встроенного редактора моделей. После окончания импорта полезно проверить корректность импортированной модели. Это можно сделать, выбрав в главном меню встроенного редактора моделей команду Tools > Consistency Check. При этом будет произведена проверка достаточности и непротиворечивости заданной в UML-модели информации. По результатам проверки будет показано окно с результатами (рис. 4.11).

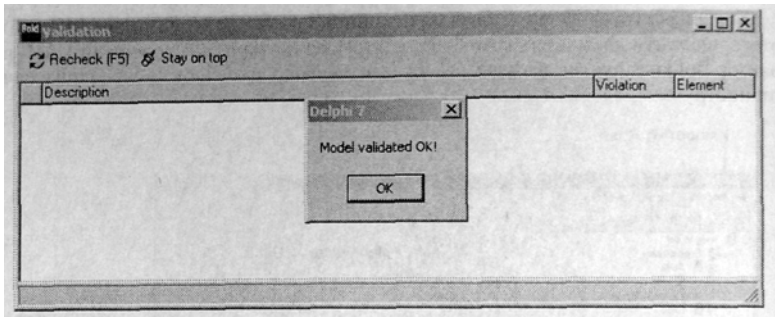


Рис. 4.11. Окно с результатами проверки модели

Теперь самое время проиллюстрировать преимущества, которые дает взаимодействие Rational Rose и Borland MDA. Если внимательно посмотреть на состав импортированной модели, то можно увидеть, что в результате импорта UML-модель изменилась. Так, исходная модель (см. рис. 4.7) содержит два класса, а импортированная в Bold модель включает три класса — в ней появился новый класс writesbyAuthor (рис. 4.12, где выделены рамками два объекта— ассоциация и автоматически созданный класс с названием writesbyAuthor).

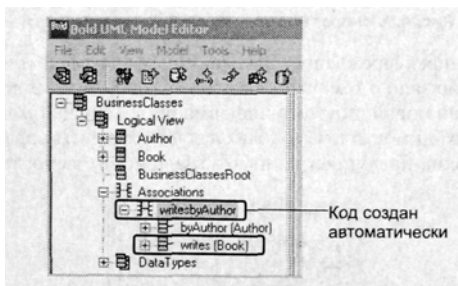


Рис. 4.12. Болдифицированная модель

Причина появления этого класса нами уже обсуждалась в предыдущей главе — он необходим для реализации отношения «многие-ко-многим» между классами Author и Book. Но когда мы создавали модель во встроенном UML-редакторе Bold, необходимо было совершать дополнительные операции для создания такого класса. Теперь же можно убедиться: взаимодействие Borland MDA и Rational Rose настолько «интеллектуально», что не ограничивается собственно передачей информации об элементах модели (классах, атрибутах, ассоциациях). В нашем случае при импорте был автоматически добавлен указанный новый класс. Если заменить в модели кратность хотя бы одной роли на «1» и снова произвести импорт, то легко убедиться, что промежуточный класс исчезнет. Таким образом, Borland MDA, как уже говорилось, в ряде случаев способна самостоятельно добавлять в модель необходимые элементы. Такая операция носит название *«болдифи-*

кация модели» — boldification, а обратная ей — *unboldification*. При необходимости в Bold-редакторе можно увидеть как болдифицированную модель (см. рис. 4.12), так и исходную, для этого достаточно выбрать в меню Tools ▸ Boldify Model или Tools ▸ Unboldify Model соответственно. При этом неболдифицированная модель по своему составу аналогична исходной и также содержит всего два класса (рис. 4.13).

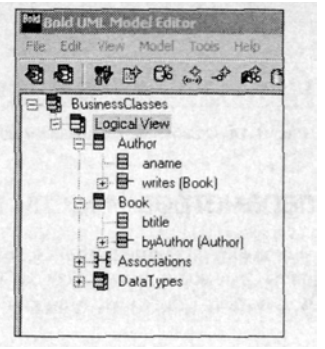


Рис. 4.13. Неболдифицированная модель

Таким образом, болдифицированная модель отличается от исходной наличием дополнительных объектов, автоматически создаваемых средой Borland MDA.

Повторим, что в данном случае принудительная болдификация модели была вызвана тем, что Borland MDA сохраняет объекты модели в реляционных базах данных, где существуют определенные ограничения — в нашем случае два класса объединены отношением «многие-ко-многим», поэтому для сохранения объектов этих классов в РСУБД необходимо наличие дополнительной связующей таблицы. И если представить модель, содержащую десятки и сотни классов, то преимущество использования программного взаимодействия Borland MDA и Rational Rose таково: все подобные связующие таблицы (точнее, классы для ассоциаций), присутствующие в отношениях «многие-ко-многим», будут сгенерированы автоматически, и при создании модели об этом заботиться не нужно. То есть при импорте автоматически производится болдификация модели. Напомним, что принудительно болдифицировать модель можно и из встроенного текстового UML-редактора, как мы это делали в предыдущей главе. В этом легко убедиться, если проделать следующие шаги.

1. Удалите класс writesbyAuthor из модели.
2. Проверьте корректность модели (команда Tools • Consistency Check), получив при этом сообщение об ошибке (рис. 4.14).
3. Отмените болдификацию (команда Tools • Unboldify Model).
4. Произведите болдификацию (команда Tools ▸ Boldify Model).

В результате легко убедиться, что удаленный нами дополнительный класс ассоциации writesbyAuthor создастся вновь, и модель снова станет корректной.

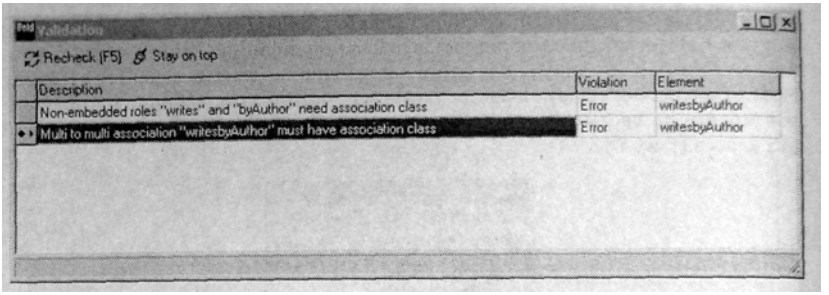


Рис. **4.14.** Ошибки верификации модели

Настройка тег-параметров модели в Rational Rose

Вернемся в Rational Rose и несколько видоизменим нашу модель, а именно - будем использовать русскоязычные названия классов, атрибутов и ролей (рис. 4.15). Для этого отредактируем в Rational Rose названия классов, атрибутов и ролей ассоциации.

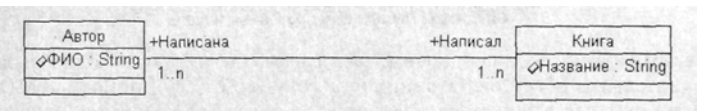


Рис. 4.15. UML-модель с использованием кириллицы

Сохраним новую диаграмму классов в файле, например librus.mdl. Теперь приступим к настройке модели. Для этого воспользуемся некоторыми тег-параметрами, доступ к которым обеспечивается непосредственно из редактора Rational Rose. Щелкнем дважды по классу Автор, перейдем на вкладку Bold и модифицируем следующие тег-параметры (рис. 4.16):

- DelphiName — присвоим значение TAuthor, этот тег-параметр отвечает за преобразование имени класса UML-модели в идентификатор класса, используемый в среде Delphi;
- ExpressionName — присвоим значение Author, этот тег-параметр отвечает за преобразование имени класса модели в имя объекта, используемое для обозначения класса в выражениях на языке OCL;
- TableName — присвоим значение Author, этот тег-параметр отвечает за преобразование имени класса модели в имя таблицы СУБД, используемой для хранения информации об объектах данного класса.

По умолчанию три вышеуказанных тег-параметра имели вид T<Name>, <Name>, <Name> соответственно, что означает автоматическую подстановку вместо <Name> соответствующего имени класса модели. При этом в нашем случае генерировались бы русскоязычные идентификаторы, что, например, для идентификатора класса в Delphi является недопустимым (в нашем случае для класса Автор было бы сгенерировано имя ТАвтор).

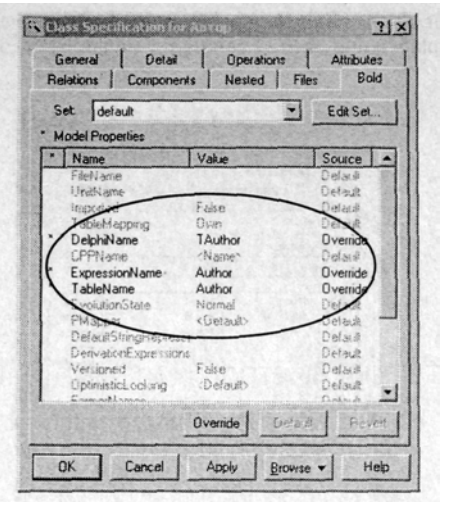


Рис. 4.16. Настройка тег-параметров в Rational Rose

Для решения проблемы автоматического преобразования имен модели из национальных кодировок в составе Bold имеются специальные средства отображения имен, активизация которых обеспечиваются, в частности, тег-параметром проекта NationalCharConversion. Однако для их использования в общем случае придется перекомпилировать исходные тексты Bold for Delphi. Возможны и другие способы решения этой проблемы, например средствами Rational Rose — путем написания специального скрипта на языке BasicScript.

ПРИМЕЧАНИЕ

В приложении А представлен листинг, содержащий исходный код на языке BasicScript полнофункционального скрипта для Rational Rose. Скрипт производит транслитерацию русскоязычных имен элементов диаграммы классов для использования в модели приложения для Bold for Delphi. Там же содержатся инструкции по использованию этого скрипта.

Естественно, в диаграмме классов можно использовать и только англоязычные имена, тогда редактирование вышеописанных тег-параметров не потребуется.

Аналогично настроим тег-параметры в спецификации класса Книга и присвоим им следующие значения:

- DelphiName — значение TBook;
- ExpressionName — значение Book;
- TableName — значение Book.

Аналогичным образом отредактируем атрибуты классов — вместо русскоязычных идентификаторов ФИО и Название введем FIO и Nazvanie соответственно. (Для редактирования тег-параметров у атрибутов необходимо перейти в окно спецификации данного атрибута и выбрать вкладку Bold.)

Теперь перейдем к настройке ассоциации. Вызовем ее спецификацию, перейдем на вкладку Bold A и изменим значение тег-параметров следующим образом (см. рис. 4.17).

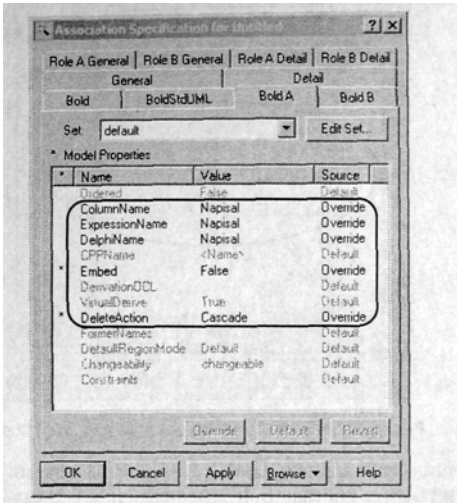


Рис. 4.17. Настройка тег-параметров для первой роли ассоциации

- ColumnName — присвоим значение Napisal. Этот тег-параметр задействуется Borland MDA в случае единичной кратности роли для обозначения названия столбца таблицы БД.
- ExpressionName — присвоим значение Napisal.
- DelphiName — присвоим значение Napisal.
- Embed — присвоим значение False. Этот тег-параметр определяет «встраивание» роли отношения в класс. Встраиваться могут только роли с единичной кратностью. Для понимания этого проведем аналогию с реляционными СУБД, где в таблицах, связанных отношением «один-ко-многим», подчиненная таблица должна содержать «встроенное» ключевое поле, связанное с уникальным ключом мастер-таблицы. Если два класса связаны ассоциацией, обе роли которой имеют кратность больше 1, то независимо от значения тег-параметра Embed Bold обеспечит создание дополнительного связующего класса, как мы убедились ранее. Поэтому в данном случае мы могли и не изменять значение этого параметра.
- DeleteAction — присвоим значение Cascade. С этим тег-параметром мы уже имели дело, когда создавали простое приложение. Он отвечает за правила удаления подчиненных объектов при удалении главного.

Аналогично изменим значения тег-параметров и для второй роли нашей ассоциации (рис. 4.18).

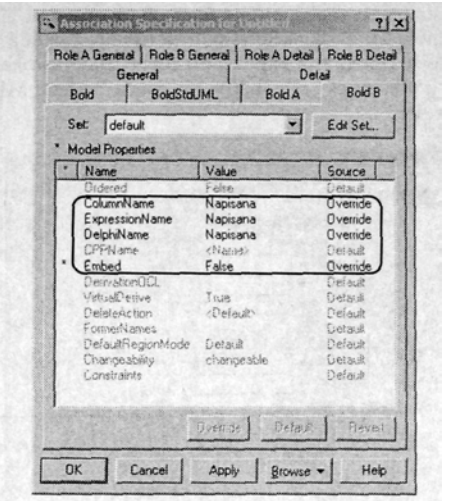


Рис. 4.18. Настройка тег-параметров для второй роли ассоциации

Кроме этого, в спецификации ассоциации перейдем на вкладку Bold и присвоим тег-параметру LinkClassName значение LinkNapisalNapisana для обозначения имени класса ассоциации (рис. 4.19).

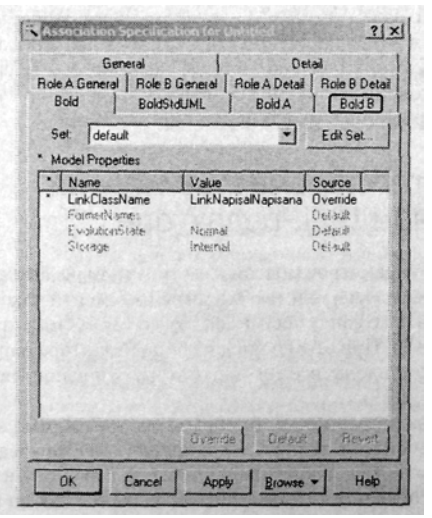


Рис. 4.19. Настройка тег-параметра класса ассоциации

Теперь сохраним модель, вернемся в наше приложение Delphi, изменим свойство FileName компонента BoldUMLRoseLink1 на имя файла новой модели librus.mdl и произведем импорт. После этого проверим корректность модели (команда Tools ▸ Consistency Check) и убедимся (рис. 4.20), что среда Borland MDA восприняла все сделанные изменения.

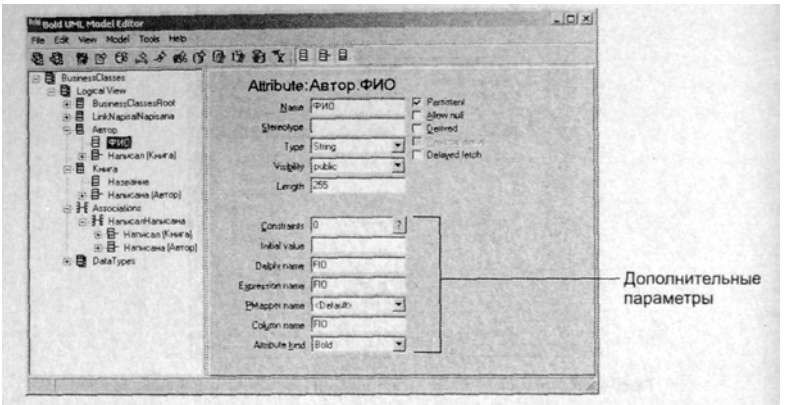


Рис. 4.20. Вид модели с русскоязычными идентификаторами

При этом названия всех классов и атрибутов после импорта остались русскоязычными, хотя на панели дополнительных параметров (она может быть вызвана в Bold-редакторе по нажатию Ctrl+A) мы увидим измененные нами значения. Если повторить действия по созданию простого приложения, описанные в предыдущей главе, то легко увидеть, что во всех заголовках столбцов таблиц, а также на автоформах, все обозначения станут русскоязычными, поскольку они берутся из названий классов, атрибутов и ролей модели приложения.

Настройка тег-параметров модели во встроенном UML-редакторе

Для настройки тег-параметров модели во встроенном редакторе Borland MDA следует выделить в дереве модели необходимый элемент (класс, атрибут, ассоциацию, роль) и выбрать команду Tools ▸ Edit tagged values (или просто нажать комбинацию клавиш Ctrl+T). При этом появится окно редактора параметров, на вкладке Bold которого представлены все тег-параметры, относящиеся к выбранному элементу модели (рис. 4.21).

Для редактирования конкретного параметра необходимо выделить его в списке параметров и ввести новое значение в правом окне тег-редактора. Кроме того, в тег-редакторе можно удалять тег-параметры или добавлять новые (пользовательские) параметры. Значения тег-параметров доступны как во время проектирования приложения, так и программно, во время его выполнения.

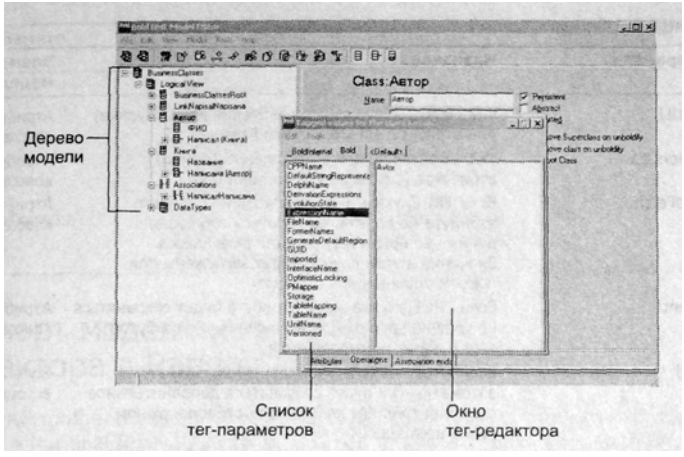


Рис. 4.21. Редактирование тег-параметров во встроенном UML-редакторе

Как уже говорилось ранее, наборы тег-параметров привязаны к уровням иерархической структуры модели, и для каждого уровня этой структуры имеется собственный набор таких параметров. Далее в табл. 4.1 представлено краткое описание функционального назначения некоторых из них. Ряд параметров (DelphiName, ExpressionName и т. д.) был описан ранее и в таблице не приводится.

Таблица 4.1. Состав основных тег-параметров и их назначение

Тег-параметр	Назначение	Элементы модели
DefaultStringRepresentation	Определяет строковое представление для объектов класса, в частности при отображении на формах, автоформах, заголовках столбцов сеток (Grid). Задается OCL-выражением	Класс
FileName	При генерации кода определяет имя файла, содержащего код операций класса	Класс
InitalVaiue	Начальное значение, автоматически присваемое атрибуту при вызове конструктора объекта	Атрибут класса
Derived	Признак того, что значение данного объекта является «вычисляемым» по данным других объектов. Правила для вычисления задаются либо OCL-выражением, либо в коде программы	Атрибут класса
Visibility	Задает «видимость» объекта при генерации свойства класса в программном коде	Атрибут класса
AttributeKind	Вид атрибута. Если значение равно BOLD, то атрибут является Bold-атрибутом, если значение равно Delphi, то — Delphi-свойством. В последнем случае информация об объекте недоступна во время выполнения. По умолчанию равно BOLD	Атрибут класса
Length	Длина атрибута. Имеет значение при генерации строковых полей таблиц некоторых СУБД, имеющих ограничения на длину строки. По умолчанию 255	Атрибут класса

продолжение

Таблица 4.1 (*продолжение*)

Тег-параметр	Назначение	Элементы модели
AllowNULL	Указывает, допустимы ли значения NULL (пустые) для атрибута. По умолчанию False	Атрибут класса
DerivationOCL	OCL-выражение для вычисляемых (derived) атрибутов (см. выше в этой таблице)	Атрибут класса
DelayedFetch	Если TRUE, указывает, что значения данного атрибута не должны вызываться из уровня данных во время загрузки объекта класса. Значения в этом случае будут загружены при первом обращении к атрибуту	Атрибут класса
Persistent	Если TRUE, то значение атрибута будет сохраняться на уровне данных. Для вычисляемых атрибутов этот параметр игнорируется	Атрибут класса
Ordered	Если TRUE, то роль упорядоченная. При этом автоматически будет создаваться дополнительное поле для таблицы класса на противоположном конце ассоциации	Роль ассоциации
DeleteAction	Определяет тип действия при попытке удаления связанного объекта. Принимает значения: Allow (разрешить) — объект удаляется Prohibit (запретить) — генерируется программное исключение Cascade (каскадное удаление) — удаляются все связанные объекты	Роль ассоциации

Ряд тег-параметров предназначены для настройки самого верхнего уровня иерархии — уровня модели. Некоторые из них представлены в табл. 4.2.

Таблица 4.2. Состав основных тег-параметров модели

Тег-параметр	Назначение	Значение по умолчанию
ModelName	Определяет имя модели	'BusinessClasses'
RootClass	Определяет имя суперкласса-родоначальника всех классов модели. Если значение не присвоено, то в качестве суперкласса используется BusinessClassesRoot	пустая строка
GUID	Определяет глобальный уникальный идентификатор библиотеки типов (Type Library) при генерации интерфейсов. Если не назначен, генерируется случайный GUID	пустая строка
TypeLibVersion	Определяет версию библиотеки типов при генерации интерфейсов	пустая строка
UnitName	Определяет имя программного модуля, содержащего код для классов модели. Используется при генерации кода	'BusinessClasses'
ImplemenationUses	Задает список модулей, разделенных запятой, для включения в IMPLEMENTATION — раздел генерируемого программного модуля	пустая строка
InterfaceUses	Задает список модулей, разделенных запятой, для включения в INTERFACE — раздел генерируемого программного модуля	пустая строка

Тег-параметр	Назначение	Значение по умолчанию
UseGlobaLId	Определяет необходимость генерации GUID для каждого объекта, сохраняемого в БД	TRUE
UseXFiles	Определяет необходимость генерации дополнительной таблицы, сохраняющей информацию обо всех объектах, когда-либо содержащихся в БД	TRUE
UseModelVersion	Позволяет задать номер версии модели. Используется дополнительными средствами поддержки версий	0

Экспорт модели из встроенного редактора в Rational Rose

После настройки модели во встроенном редакторе есть возможность экспортировать ее в Rational Rose. Для этого, как и при импорте, необходимо настроить компонент BoldUMLRoseLink на имя файла модели и экспортировать модель, нажав на крайнюю левую кнопку со стрелкой на панели инструментов (рис. 4.22) или выбрав команду File • Export via Link.

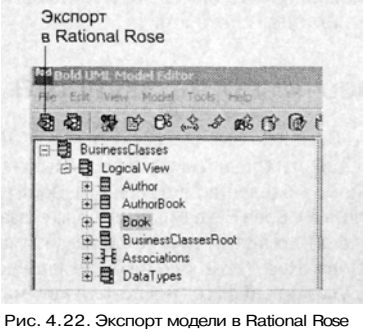


Рис. 4.22. Экспорт модели в Rational Rose

При этом если приложение Rational Rose не было заранее запущено, произойдет его автоматический запуск. При экспорте модели полностью сохраняются значения всех тег-параметров. Таким образом, после настройки тег-параметров в среде Bold for Delphi очень удобно снова экспортировать уточненную модель в Rational Rose и запомнить ее в файле. Можно также создать UML-модель во встроенном Bold-редакторе полностью, а затем экспортировать ее в редактор Rational Rose (например, в заранее созданную пустую модель). Однако при этом необходимо иметь в виду, что после такого экспорта окно диаграммы классов в Rational Rose будет выглядеть пустым. Это и понятно, поскольку в формате модели встроенного текстового Bold-редактора отсутствует информация о геометрическом расположении классов, их размерах, цвете и т. д., в то время как в формате модели Rational Rose вся эта информация сохраняется. Но выход в таком случае есть, и достаточно про-

стои. После такого экспорта в дереве модели Rational Rose (рис. 4.23) содержатся все элементы модели (классы и ассоциации).

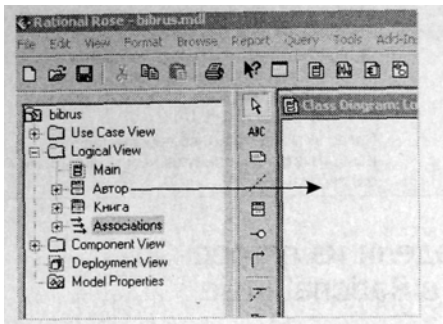


Рис. 4.23. «Перетаскивание» элементов модели на диаграмму классов

Достаточно при помощи мыши «перетащить» эти элементы на пустую область диаграммы классов. При этом автоматически будут созданы графические изображения классов, а для связанных классов также автоматически появятся линии-ассоциации. После сохранения такой модели в Rational Rose в дальнейшем с ней можно будет работать обычным способом.

Другие способы импорта-экспорта моделей

Для взаимодействия с Rational Rose существует и другой способ импорта и экспорта UML-моделей. Для этих целей можно воспользоваться средствами открытия и сохранения файлов моделей во встроенном Bold-редакторе. Если выбрать команду главного меню File > Open File и в окне выбора указать файл модели Rational Rose (с расширением .mdl), то в этом случае также автоматически произойдет запуск приложения Rational Rose (если оно не было запущено ранее). В это приложение будет загружен указанный файл, после чего автоматически произойдет импорт в среду Borland MDA. Преимуществом такого способа импорта является отсутствие необходимости использования и настройки компонента BoldUMLRoseLink.

ВНИМАНИЕ

Все без исключения операции импорта и экспорта, реализующие обмен моделями с CASE-системой Rational Rose, требуют обязательной предварительной инсталляции и настройки приложения Rational Rose.

Аналогично, создав или отредактировав модель во встроенном Bold-редакторе, можно выбрать команду главного меню File ▸ Save File As..., указать имя файла модели (обязательно существующего!) и осуществить тем самым экспорт модели в Rational Rose. При этом существует только одно, но существенное отличие от экспорта «штатным» способом (с использованием компонента BoldUMLRoseLink). Перед таким «экспортом через сохранение» обязательно нужно сделать операцию Tools ▸ Unboldify Model, то есть убрать из модели дополнительные элементы, авто-

матически созданные средой Bold. В противном случае, если сохранять болдифицированную модель, то все «промежуточные» классы будут также экспортированы в Rational Rose.

Bold может сохранять и загружать информацию о модели не только в файлах формата Rational Rose (.mdl), но и в собственном формате (.bld), в формате XMI, а также в формате ModelMaker (.mpb). Для использования этих возможностей необходимо из меню встроенного редактора выбрать File • Open File или File ▸ Save File as....

Инструменты встроенного редактора моделей

В этом разделе будут рассмотрены дополнительные инструментальные средства встроенного редактора моделей.

Средства настройки свойств модели

Панель настройки общих свойств модели отображается при выборе в дереве моделей самого верхнего элемента (рис. 4.24). Эта панель представляет собой набор флажков-переключателей, задающих основные свойства модели. Кратко рассмотрим назначение некоторых из них.

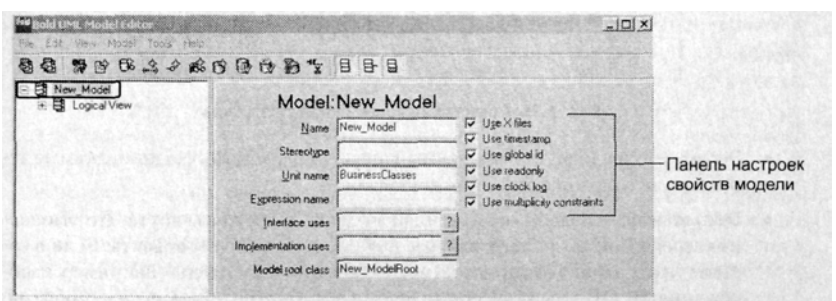


Рис. **4.24.** Панель настроек модели встроенного редактора

Use X files — этот параметр задает необходимость генерации средой дополнительной таблицы, предназначенной для хранения информации обо всех объектах, когда-либо существовавших в объектном пространстве. Необходимо включить при использовании отдельно поставляемого продукта-расширения «Object Lending Library», обеспечивающего обмен данными между несколькими базами данных. Также необходимо включить при задействовании параметров Use timestamp и Use global id (см. ниже).

Use timestamp — необходим для генерации дополнительного столбца в таблице, наличие которой определяется предыдущим параметром.

Use global id — задает необходимость генерации 128-битных глобальных уникальных идентификаторов (GUID) для каждого элемента в базе данных. Доступ к таким GUID возможен из программы приложения.

- Use readonly — задает необходимость генерации дополнительного столбца BOLD_READONLY для главной таблицы БД каждого класса модели.

Средства настройки атрибутов класса

Для настройки свойств атрибутов используется набор параметров, отображающихся во встроенном редакторе при выборе в дереве модели конкретного атрибута класса (рис. 4.25). При этом доступны следующие параметры-свойства.

- Persistent — задает, будет ли данный атрибут сохраняться в базе данных; если данный флажок не установлен, атрибут будет существовать только во время выполнения приложения.
- Allow null — показывает, допустимо ли Null-значение данного атрибута.

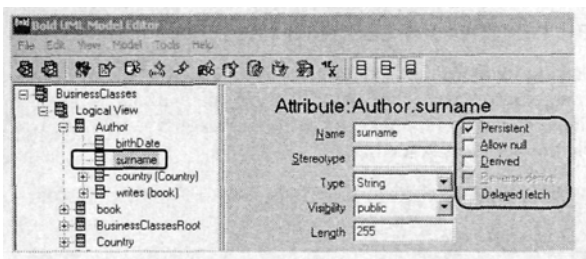


Рис. **4.25.** Средства настройки атрибутов класса

- Derived — определяет вычисляемый атрибут (см. главу 5). Вычисляемые атрибуты не сохраняются в базе данных.
- Delayed fetch — задает «отложенный» вызов данного атрибута. По умолчанию среда Bold загружает из базы данных все атрибуты объекта. Если в составе таких атрибутов присутствуют «большие» элементы (например, изображения, BLOB-объекты), то для повышения быстродействия целесообразно установить для таких атрибутов данный флажок. При этом их загрузка будет осуществляться при первом обращении к данному атрибуту (например, при активизации окна, в котором должна отображаться картинка).

Настройка ролей ассоциаций

При выборе в дереве модели встроенного редактора роли какой-нибудь ассоциации, справа появляется набор параметров для настройки ее свойств (рис. 4.26).

- Navigable — означает, что элементы данной роли могут быть доступны для просмотра. Для изображенной на рис. 4.26 роли writes установленный флажок означает, что при навигации по модели мы можем, начав с контекста класса Author, получить список написанных им книг. Если данный флажок снять, то мы получим однонаправленную ассоциацию (рис. 4.27). То есть в данном случае мы сможем просматривать авторов конкретной книги, но не сможем получить список книг конкретного автора.

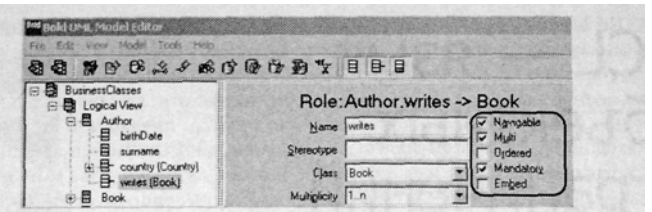


Рис. 4.26. Средства настройки ассоциаций

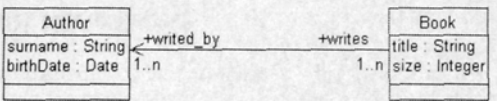


Рис. 4.27. Однонаправленная ассоциация

Отметим, что в случае, если разработчик убрал флажки с обеих ролей ассоциации, среда Bold будет рассматривать эту ассоциацию как ненаправленную, с обеспечением двустороннего доступа к элементам ролей.

Multi — устанавливается автоматически в случае кратности роли, большей 1.

Ordered — задает упорядоченную роль, при этом в соответствующей таблице базы данных будет автоматически сгенерирован дополнительный столбец, сохраняющий информацию о порядке элементов данной роли.

Mandatory — задает «обязательность» роли и устанавливается автоматически при кратности данной роли, большей чем 0.

Embed — задает «встраивание» роли в данный класс (аналогично внешним ключам в таблицах реляционных БД). Параметр может быть установлен только для ролей единичной кратности. Если обе роли ассоциации — множественные, то Bold сгенерирует дополнительную связующую таблицу в базе данных (см. также главу 3).

Резюме

В этой главе мы познакомились с основами создания модели приложения в графическом UML-редакторе Rational Rose и во встроенном редакторе моделей. Вполне очевидно, что при создании больших моделей наглядность графического представления безусловно будет иметь преимущество перед текстовым описанием. Кроме того, как мы убедились, при импорте модели из Rational Rose автоматически происходит ее болдификация, что также позволяет не задумываться при проектировании о второстепенных элементах. И, наконец, простота доступа к тег-параметрам Borland MDA из Rational Rose обеспечивает необходимую гибкость при настройках модели приложения. Показано, что наличие тег-параметров обусловлено необходимостью тонкой настройки модели и ее адаптации к среде разработки Delphi. С этой точки зрения тег-параметры играют роль элементов платформенно-зависимой модели (PSM).

OCL — язык объектных ограничений

Общие сведения

Появление Object Constraint Language (OCL) было вызвано необходимостью формализации описания условий и ограничений, накладываемых на элементы диаграммы классов UML. Такие условия могут быть сформулированы и на естественном языке, который, однако, не является строгим и формальным и допускает неоднозначные трактовки, вследствие чего не может быть использован при создании UML-моделей. Язык OCL создавался как формальный текстовый язык, дополняющий графические возможности языка UML. В этом смысле OCL является частью UML. При этом средствами языка OCL принципиально невозможно изменить сами элементы модели (классы, атрибуты, отношения). OCL также не является языком программирования, то есть не позволяет создать программу из своих операторов или описать логику выполнения каких-либо действий. Язык OCL представляет собой формальный язык, основанный на выражениях. Любое выражение OCL возвращает некоторое значение.

OCL был разработан в корпорации IBM, в 1997 году вышла спецификация языка версии 1.1, в разработке и согласовании которой приняли участие такие компании, как Rational, Microsoft, Oracle, Hewlett-Packard и ряд других. В настоящее время разработку новых спецификаций OCL координирует консорциум OMG.

Роль OCL в Borland MDA

В Borland MDA язык OCL играет чрезвычайно важную роль, выполняя следующие основные функции:

- навигация по элементам модели (классам, атрибутам, ассоциациям);
- задание условий и ограничений на элементы модели;
- задание выражений для вычисляемых (derived) атрибутов.

Навигация по модели, обеспечиваемая посредством OCL в Borland MDA, позволяет осуществить гибкий и мощный механизм «запросов» к *объектному пространству* приложения. Понятие объектного пространства будет описано позднее, сейчас достаточно сказать, что оно охватывает реализацию всех элементов модели во время работы приложения. Такие «OCL-запросы» в принципе способны полностью заменить привычный разработчикам приложений баз данных язык SQL, обладая при этом наглядностью, лаконичностью и мощностью. Кроме того, учитывая платформенную независимость OCL, эти запросы являются универсальными и абсолютно не привязаны к конкретной СУБД, используемой в приложении.

Модель для изучения

Для изучения возможностей OCL рассмотрим следующую UML-модель (рис. 5.1). Будем считать, что эта модель описывает условное приложение для работы с библиотечным каталогом. Модель содержит следующие классы:

- Country (страна) — имеет текстовый атрибут name (название страны);
- Author (автор) — имеет текстовый атрибут surname (фамилия) и атрибут типа дата — **birthDate** (дата рождения);

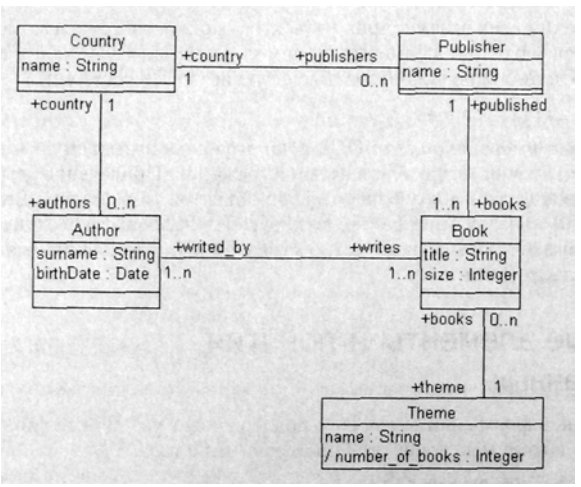


Рис. 5.1. Рассматриваемая UML-модель

- Publisher (издательство) — имеет текстовый атрибут name (название);
- Book (книга) — имеет текстовый атрибут title (название) и целый атрибут size (количество страниц);
- Theme (тематика) — имеет текстовый атрибут name и вычисляемый целый атрибут number_of_books.

Содержащиеся в модели отношения-ассоциации позволяют сформулировать следующие основные бизнес-правила, определяемые нашей моделью:

- каждый автор может написать одну или несколько книг,
- каждая книга может быть написана одним или несколькими авторами;
- каждый автор является гражданином одной страны;
- в каждой стране может проживать несколько авторов или ни одного;
- в каждой стране может существовать несколько издательств или ни одного;
- каждое издательство принадлежит только одной стране;
- каждое издательство может издать одну или несколько книг;
- каждая книга может быть издана только одним издательством;
- каждая книга может относиться только к одной тематике;
- может существовать несколько книг, принадлежащих одной тематике, или не существовать ни одной книги по данной тематике.

Доступ к классам и атрибутам из OCL

Для получения всех экземпляров (объектов) конкретного класса модели в OCL применяется операция allInstances; таким образом, для получения списка стран в нашей модели можно использовать следующее OCL-выражение:

```
Country.allInstances
```

На этом примере видно, что OCL использует так называемую «dot-нотацию», то есть симол точки между элементами выражения. Приведенное выражение для класса Country (страна) возвращает набор объектов типа Country. Если мы хотим получить набор названий стран, то должны воспользоваться следующим выражением Country.allInstances.name то есть указать в OCL-выражении имя конкретного атрибута класса (name).

Базовые элементы и понятия

Типы данных

OCL, являясь формальным языком, поддерживает различные типы данных. Их список и необходимые пояснения представлены в табл. 5.1.

Таблица 5.1. Типы данных в OCL

Тип	Описание типа
Целый	Любое целое число
Действительный	Любое действительное число
Логический	True (Истина) или False (Ложь)
Строка	Любое количество символов

Операции сравнения в OCL

OCL поддерживает следующие операции сравнения, используемые в формировании различных условий:

- = (равно);
- o (не равно);
- < (меньше);
- > (больше);
- <= (меньше или равно);
- >= (больше или равно);
- or (или);
- and (и);
- not (не);
- xor (исключающее ИЛИ);
- implies (одностороннее И).

На последней операции implies остановимся поподробнее, так как она не является широко известной. Implies выступает как своеобразная операция типа and, но действует «в одну сторону» (табл. 5.2). Смысл ее в OCL примерно следующий: «если результат левой части выражения — истина, то результат правой части также должен быть истиной. Если левая часть выражения — ложь, то результат — всегда истина».

Таблица 5.2. Варианты значений операции implies

	A=True B=True	A=True B=False	A=False B=True	A=False B=False
A implies B	True	False	True	True
B implies A	True	True	False	True

Типы возвращаемых выражений

В результате применения некоторого OCL-выражения могут быть получены результаты следующих типов:

- метаданные — информация о типах, OCL также обеспечивает некоторые возможности по приведению типов;
- объекты — экземпляры классов модели;
- типы атрибутов модели (целые, строковые и т. д.).

Кроме этого, результатом может быть также и набор из элементов каждого из вышеперечисленных типов — *коллекция*. При этом внутри каждой коллекции могут содержаться элементы только одного типа.

Арифметические операторы

OCL подержиивает общепринятые арифметические операторы, состав которых и примеры использования приведены в табл. 5.3.

Таблица 5.3. Арифметические операторы в OCL

Оператор	Пример	Результат	Описание
+	2+9 '2'+'3'	11'23'	Сложение
_	19-2 [17]		Вычитание
*	Book.allInstances–>size*2	Удвоенное количество книг	Умножение
/	6/2	3	Деление
0	(3 - 2)*4	4	Взятие в скобки
.round	(Book.allInstances–>size/ Author.allInstances–>size).round	Округленное среднее количество книг, написанных одним автором	Округление до ближайшего целого
.floor	375.8.floor	375	Взятие целой части
.abs	- 23.abs	23	Взятие модуля

Коллекции

Понятие «коллекция» встречается в OCL довольно часто. Коллекция — это тип данных, определяющий некоторый набор элементов одного типа. В OCLсуществует три основных вида коллекций:

1. Set (множество) — определяет неупорядоченный набор элементов без повторяющихся элементов.
2. Bag (мультимножество) — определяет неупорядоченный набор элементов, возможно, с повторяющимися элементами.
3. Sequence (последовательность) — определяет упорядоченный набор элементов без повторяющихся элементов.

Для применения операций к коллекциям в OCL используется специальный оператор взятия коллекции, обозначаемый «->».

Например, выражение OCL Book.allInstances->asSet вернет множество-коллекцию объектов типа Book, не содержащую одинаковых книг. А выражение Author. allInstances–>size вернет количество всех авторов, возможно, с повторяющимися элементами в списке. Здесь операция size выполняет функцию получения количества элементов коллекции, к которой она применена.

Навигация по модели

Используя язык OCL, достаточно просто осуществлять доступ к элементам модели при помощи отношений-ассоциаций между классами. При этом в качестве *контекста* OCL-выражения может выступать один класс, а в качестве результата — элементы других классов. Отметим, что понятие «контекст» в данном случае означает элемент модели, выступающий в качестве источника информации для OCL-выражения. Для обозначения текущего контекста в OCL используется зарезервированное слово self.

Рассмотрим следующее выражение Country.allInstances.authors. В данном случае в качестве контекста-источника выступает класс Country, однако далее рассматриваемое OCL-выражение включает в себя название роли authors ассоциации, связывающей класс Country с классом Author. Поэтому результатом выражения в данном случае будет являться коллекция объектов — экземпляров типа Author. Аналогично можно построить такие OCL-выражения как Country.allInstances.publishers.books или Book.allInstances.writed_by.country.

Таким образом, наглядно видно, как именно происходит навигация по UML-модели — начиная с конкретного класса и последовательно дописывая в OCL-выражение названия ролей противоположных концов ассоциаций — отношений, связывающих очередной класс со следующим в цепочке, в принципе можно получать результаты, относящиеся к любому классу модели. Необходимым для этого условием является наличие ассоциаций между классами в построенной таким образом цепочке. И опять мы наблюдаем здесь активное использование «dot-нотации». При этом может возникнуть закономерный вопрос — а зачем в таком случае используется оператор взятия коллекции «–>», рассмотренный в предыдущем подразделе, почему нельзя просто воспользоваться «dot-нотацией»? Для ответа на этот вопрос рассмотрим следующее OCL-выражение Book.allInstances.size. В этом выражении присутствует неоднозначность, вызванная совпадением имен атрибута size и операции взятия количества элементов. Поэтому в данном случае не вполне понятно, что имеется в виду — количество всех книг или набор-коллекция, элементы которой содержат число страниц каждой книги. Операция взятия коллекции устраняет такого рода неоднозначности, и выражение Book.allInstances–>size имеет единственную трактовку — количество всех книг.

Из рассмотренных выше примеров вполне очевидно, что возможности навигации по модели посредством OCL-выражений определяются в основном правильным построением структуры самой модели. Для отдельного изолированного класса модели, не имеющего отношений-ассоциаций с другими классами, такая навигация невозможна.

Во всех рассмотренных выше случаях результатами OCL-выражений являются коллекции, которые могут быть получены и более простыми способами. В самом деле, если мы рассмотрим формально вполне допустимое OCL-выражение Theme.allInstances.books.published, то из его содержания очевидно, что результатом такого выражения является коллекция, содержащая все издательства. Понятно, что этот же результат может быть получен просто как Publisher.allInstances. Поэтому в таком виде возможности OCL-навигации выглядят не совсем понятными с точки зрения практики. Чтобы такая навигация имела практический смысл, целесообразно использовать специальные операции над коллекциями, которые мы сейчас и рассмотрим.

Операции над коллекциями

Навигация по коллекции

Данные операции предназначены для перемещения по набору элементов, образующих коллекцию.

Операция First

Используется для выбора первого элемента коллекции. Например, OCL-выражение

```
Book.allInstances->first.name
```

возвращает в качестве результата название первой книги.

Операция Last

Данная операция используется для выбора последнего элемента коллекции. Например, OCL-выражение

```
Book.allInstances->last.authors
```

вернет в качестве результата список авторов последней книги, а выражение

```
Book.allInstances->last.authors->first.name
```

вернет имя первого автора последней книги.

Операция At

Операция At(K — целый параметр) используется для выбора конкретного элемента коллекции, который располагается на *K*-м месте в наборе. Например, OCL-выражение

```
Author.allInstances->at(3).name
```

возвращает имя третьего автора из списка всех авторов.

Выборка (фильтрация) и исключение

Операция Select

Операция select(<условие>) используется в OCL для осуществления фильтрации коллекций, то есть выборки подмножества элементов по какому-либо условию.

Например, для выбора всех объектов типа Country (страна) за исключением страны с названием «Мозамбик» можно сформировать следующее OCL-выражение:

```
Country.allInstances->select(name<>'Мозамбик')
```

В результате мы получим коллекцию стран, не содержащую указанной страны. Привлекая возможности OCL-навигации по модели, можно написать следующее выражение:

```
Country.allInstances->select(name='Мозамбик').authors.writes
```

- и получить список всех книг, авторы которых проживают в Мозамбике. В этом примере внутри скобок операции select стоят атрибуты, принадлежащие классу, объекты которого фильтруются (в приведенных выражениях использовался атрибут name — название страны). Можно объединять операции выборки в цепочки и реализовывать более сложные OCL-запросы. Рассмотрим следующее OCL-выражение:

```
Book.allInstances
->select(published.country.name<>'Япония')
->select(theme.name='учебник')->size
```

Его результатом является количество книг, изданных всеми издательствами, за исключением находящихся в Японии, и при этом выбираются только книги, относящиеся к учебной тематике. В этом выражении интересным является использование в условии выборки «навигационных» выражений — published.country.name и theme.name, содержащих как роли, так и атрибуты «конечных» классов — Country и Theme. На этом примере можно оценить гибкость подхода, реализуемого языком ОСЬ. Если провести сравнение с языком SQL, то оно будет явно не в пользу последнего. Для получения даже более простого результата пришлось бы сформировать довольно громоздкий SQL-оператор примерно такого вида (в достаточно условных обозначениях):

```
select b.name,b.idpub,b.idtheme,p.idcountry
from book b, theme t, publisher p, country c
where b.idpub=pub.id and b.idtheme=theme.id
and p.idcountry=country.id and c.name<>'Япония'
and t.name='учебник'
```

Отметим, что в рассмотренном OCL-выражении не использовались отношения «многие-ко-многим» между классами UML-модели, в противном случае аналогичный SQL-запрос сформировать было бы вообще невозможно в силу ограничений реляционной модели (в этом случае пришлось бы искусственно добавлять связующую таблицу). Кроме продемонстрированной лаконичности записи, OCL-выражение, безусловно, выигрывает также и в наглядности.

Операция Reject

Операция исключения Reject(<условие>) является операцией, обратной по отношению к Select, то есть все записи, удовлетворяющие какому-либо условию, исключаются из результирующего набора элементов. Синтаксис операций Reject и Select и правила их использования в OCL-выражениях идентичны. Например, эти OCL-выражения полностью эквивалентны:

```
Country.allInstances->select(name<>'Мозамбик')
Country.allInstances->reject(name='Мозамбик')
```

Операции с несколькими множествами

Операция Difference

Смысл операции Difference(<набор>) заключается в получении результирующего множества значений как разницы между исходным набором и некоторым заданным внешним набором. Данная операция работает с двумя коллекциями и требует, чтобы до ее выполнения был сформирован некоторый внешний набор элементов.

Рассмотрим следующие OCL-выражения:

```
Book.allInstances->select(theme.name='художественная')
Book.allInstances->select(published.country.name='Россия')
->difference(набор)
```

Первое выражение возвращает набор художественных произведений, а второе берет этот набор в качестве параметра, сравнивает его поэлементно со всеми книгами, изданными в России, и возвращает разницу этих двух наборов книг, то есть

в результате получится список книг российских издательств, исключая художественные произведения.

Операция SymmetricDifference

Операция SymmetricDifference(<набор>) также работает с двумя наборами элементов. Ее результатом является множество-коллекция, содержащая элементы, каждый из которых присутствует в одной и только в одной коллекции. Применяя OCL-выражения, аналогичные предыдущему примеру

```
Book.allInstances->select(theme.name='художественная')
Book.allInstances->select(published.country.name='Россия')
->symmetricdifference(набор)
```

получим в результате набор книг, включающий либо художественные произведения, изданные где угодно кроме России, либо нехудожественные российские произведения.

Операция Intersection

Операция Intersection(<набор>) позволяет получить пересечение двух множеств-наборов, то есть коллекцию элементов, одновременно входящих в обе заданные коллекции.

Для аналогичных выражений:

```
Book.allInstances->select(theme.name='художественная')
Book.allInstances->select(published.country.name='Россия')
->intersection(набор)
```

получим в результате список изданных российскими издательствами художественных книг.

Операция Union

Как легко догадаться по названию, операция Union(<набор>) возвращает объединение двух заданных множеств-коллекций. Например, для выражений:

```
Book.allInstances->select(theme.name='художественная')
Book.allInstances->select(published.country.name='Россия')
->union(Набор)->asSet
```

в результате мы получим список, объединяющий все художественные произведения плюс все книги российских издательств. Для исключения повторяющихся значений во втором выражении применена уже знакомая нам операцию взятия множества asSet.

Сортировка элементов коллекции

Операция OrderBy

Операция OrderBy(<OCL-выражение>) обеспечивает сортировку коллекции по условию, задаваемому выражением-параметром. В качестве параметра может выступать любое допустимое OCL-выражение, то есть результат этого выражения должен удовлетворять контексту.

Например, OCL-выражение

```
Book.allInstances->orderby(name)
```

вернет в качестве результата коллекцию книг, упорядоченных по их названиям.

А выражение

```
Author.allInstances->orderby(country.name)
```

вернет список авторов, упорядоченный по названиям стран. В приведенных примерах выражения-параметры удовлетворяют контексту, так как построены либо из атрибутов класса, над которым производится операция, либо из роли связанной с ним ассоциации и атрибута связанного класса. А вот следующее OCL-выражение:

```
Author.allInstances->orderby(publishers.title)
```

является некорректным, поскольку классы Author 11 Publisher не связаны отношением ассоциации.

Операция OrderDesending

Операция OrderDesending(<OCL-выражение>) аналогична предыдущей, с тем только отличием, что порядок сортировки коллекции-результата — обратный. Результатами следующих двух OCL-выражений:

```
Author.allInstances->orderby(country.name)
Author.allInstances->orderdescending(country.name)
```

будут являться одинаковые по составу, но упорядоченные взаимно-противоположным способом коллекции авторов.

Логические операции над коллекциями

Операция Includes

Операция Includes(<элемент>) возвращает логическое значение True, если элемент, заданный в качестве параметра, присутствует в коллекции, и False — в противном случае.

Например, OCL-выражение

```
Country.allInstanses.name->includes('Россия')
```

будет истинным, если Россия присутствует в списке стран, и ложным — в противном случае.

Операция forAll

Операция forAll(<условие>) проверяет заданное условие-параметр для каждого элемента коллекции, и возвращает логическое значение True только в том случае, если это условие выполняется для всех элементов. Если хотя бы для одного элемента коллекции оно не выполнено, результатом данной операции будет логическое значение False. Например, следующее OCL-выражение

```
Author.allInstances->forAll(country.name='США')
```

вернет в качестве результата True только в том случае, если все авторы проживают в США.

Операция IncludesAll

Операция IncludesAll(<набор>) использует коллекцию-параметр и осуществляет проверку, входит ли каждый элемент коллекции-параметра в коллекцию, над которой производится операция. Результатом операции является True только в том случае, если все без исключения элементы коллекции-параметра являются членами обрабатываемой коллекции. Например, рассмотрим следующие OCL-выражения:

```
Author.allInstances->select(country.name='Франция').books
Book.allInstances->select(published.country.name='Франция')
->includesAll(набор)
```

Первое из них возвращает список — набор книг, написанных авторами, проживающими во Франции. Второе — выбирает список книг, изданных французскими издательствами, и проверяет, все ли книги из первого набора входят в этот список. Если хотя бы одна книга автора-француза была издана не во Франции, то результатом данной операции будет False.

Операция isEmpty

Проверяет, является ли коллекция пустой (не содержащей ни одного элемента), и возвращает в этом случае значение True. Если коллекция содержит хотя бы один элемент, результатом будет False.

Операция notEmpty

Данная операция является обратной к операции isEmpty, и возвращает True, если коллекция не пуста. Например, OCL-выражение

```
Author.allInstances->select(name='Иванов').books
->select(theme.name='Учебник')->notEmpty
```

будет истинным, если автор с фамилией Иванов написал хотя бы один учебник.

Операция Collect

Операция Collect является довольно мощным инструментом языка OCL. В результате применения данной операции создается новая коллекция, вид которой зависит от параметра-условия данной операции. При этом коллекция-результат будет представлять собой набор элементов, каждый из которых является результатом своеобразной группировки исходной коллекции по задаваемому параметру. Например, следующее OCL-выражение

```
Publisher.allInstances->collect(books->size)
```

даст список (коллекцию целых чисел), каждый элемент которого равен числу книг, выпущенных конкретным издательством, а количество элементов в этом списке равно количеству всех издательств. Легко видеть, что без применения операции Collect результат будет совсем иным:

```
Publisher.allInstances.books->size
```

В последнем случае результатом выражения будет являться общее количество книг, хотя результатом выражения Publisher.allInstances.books также будет являться коллекция, то есть это выражение аналогично выражению

```
Publisher.allInstances->collect(books)
```

Таким образом, OCL трактует применение «dot-нотации» по отношению к свойству как применение операции Collect.

Рассмотрим более сложный пример. Предположим, что нам необходимо получить общую статистику по всем странам, включающую количество всех изданных в каждой стране книг. Заметим, что в рассматриваемой модели (см. рис. 5.1) отсутствует непосредственная ассоциация, связывающая классы Country (страна) и Book (книги). Алгоритм решения на естественном языке можно сформулировать примерно так: для конкретной страны необходимо выбрать все ее издательства, для каждого получить суммарное количество выпущенных им книг и просуммировать все полученные величины. Повторить этот алгоритм для всех стран. Используя операцию Collect, мы «транслируем» естественное описание алгоритма на язык OCL, формируя OCL-выражение, решающее поставленную задачу:

```
Country.allInstances->collect(publishers->collect(books->size)
->sum)
```

Проведем подробный разбор полученного OCL-выражения. Рассмотрим его в обратном порядке. Выражение books->size в контексте publishers дает количество книг, изданных конкретным издательством. Выражение

```
publishers->collect(books->size)
```

возвращает набор элементов, каждый из которых представляет собой суммарное количество книг, выпущенных конкретным издательством, а количество элементов этого набора равно количеству издательств. OCL-выражение

```
publishers->collect(books->size)->sum
```

обеспечивает суммирование всех элементов предыдущего выражения, то есть дает общее количество книг, выпущенных всеми издательствами, участвующими в этом выражении (здесь использована OCL-операция суммирования элементов коллекции sum, и ее смысл достаточно очевиден).

А для того, чтобы эти издательства-участники каждый раз принадлежали одной стране, применяется еще одна операция Collect, уже не для издательств, а для стран, и эта операция обеспечивает «группировку» по издательствам.

Необходимо четко осознавать, что при формировании полученного OCL-выражения мы существенным образом опирались на заданную UML-модель. Так, например, если бы у класса Country отсутствовала ассоциация с ролью publishers, то мы бы не смогли написать выражение параметра для первой операции Collect. Другими словами, при формировании OCL-выражений очень часто полезно и необходимо применять навигацию по UML-модели. На этом примере видно, как тесно интегрированы языки OCL и UML. И при этом они эффективно дополняют друг друга — если UML играет роль «архитектора» системы в целом, то OCL — использует знание этой архитектуры (UML-модели) для решения конкретных задач.

Рассматривая полученное OCL-выражение

```
Country.allInstances->collect(publishers->collect(books->size)
```

нельзя очередной раз не отметить присущие языку OCL мощность и элегантность. Также можно констатировать, что выражения OCL, в отличие от многих других

формальных языков, выглядят достаточно естественно и при этом легко читаются.

Вычисления над коллекциями

Операция Count

Count(<элемент>) возвращает количество элементов коллекции, удовлетворяющих заданному параметру. При этом тип параметра должен совпадать с типом базового класса обрабатываемой коллекции. Например, OCL-выражение:

```
Author.allInstances->count(Author.allInstances-
>select(name='Иванов'))
```

возвращает в качестве результата количество авторов с фамилией Иванов. Отметим, что этого результата можно добиться и более простым способом:

```
Author.allInstances->select(name='Иванов')->size
```

Операция Sum

Операция суммирования элементов коллекции, возвращающая численное значение. Использование данной операции было продемонстрировано в предыдущем разделе.

Операция MinValue

Применяется для получения минимального значения из коллекции.

Например, OCL-выражение:

```
Author.allInstances->collect(books->size)->minValue
```

возвращает в качестве результата минимальное количество книг, написанное одним автором (при этом неизвестно, какой именно из авторов написал наименьшее количество книг, впрочем, таких авторов может быть и несколько).

Операция MaxValue

Эта операция аналогична предыдущей, за исключением того, что возвращается максимальное значение из коллекции. На базе предыдущего примера выражения для операции MinValue, построим следующее OCL-выражение:

```
Author.allInstances->select(books->size=Author.allInstances
->collect(books->size)->maxValue).name
```

Результатом будет являться список авторов, написавших максимальное количество книг. Отметим, что в данном случае мы не знаем фактический тип результирующего выражения — это будет либо строка-имя автора, либо список строк-имен таких авторов. В самом деле, пусть максимальное количество написанных каким-либо автором книг равно 5, тогда вполне может быть, что сразу несколько авторов написали по 5 книг, и в этом случае все они будут включены в результат OCL-выражения. Подобные ситуации, когда заранее неизвестен тип результата (значение или массив-коллекция), довольно часто имеют место при работе с OCL. При этом OCL всегда «считает» такие результаты именно коллекциями, независимо от количества элементов в результате. Для приведения типа результата, на-

пример, к строковому типу, можно воспользоваться описанными выше операциями first, last и at, возвращающими единственный элемент коллекции. Так, следующее OCL-выражение будет иметь строковый тип и вернет имя первого автора, написавшего максимальное количество книг:

```
Author.allInstances->select(books->size=Author.allInstances
->collect(books->size)->maxValue).name->first
```

Работа с типами в OCL

Обработка типов для элементов

Операции OclType и TypeName

Данные операции возвращают тип операнда. Отличие их в том, что операция TypeName возвращает не сам тип, а его название в виде строки.

Например, OCL-выражение:

```
Author.typeName
```

вернет в качестве результата строку 'Author'.

Операция oclIsTypeOf

Операция oclIsTypeOf(<метатип>) возвращает логическое значение True, если тип операнда совпадает с типом параметра. В противном случае возвращается логическое значение False.

Операция ocAsType

Операция ocAsType(<метатип>) осуществляет приведение операнда к типу, заданному параметром. Если такое преобразование типов невозможно, то вырабатывается исключение. Эту операцию, как правило, целесообразно использовать только для классов-наследников некоторого суперкласса.

Операция safeCast

Операция safeCast(<метатип>) имеет тот же смысл, что и предыдущая, но не вырабатывает исключения при невозможности преобразования типов. Вместо этого в такой ситуации результатом операции будет nil.

Операция superTypes

В качестве результата возвращается коллекция ближайших суперклассов (непосредственных классов-родителей) операнда. Для сред разработки, не поддерживающих множественное наследование (например, для Borland Delphi), результатом операции будет коллекция с единственным элементом.

Операция allSuperTypes

Данная операция по функциональному назначению аналогична предыдущей, за исключением того, что в качестве результата возвращаются все суперклассы, а не только непосредственные классы-родители. Такая ситуация возникает при наличии в модели цепочек наследуемых классов.

Операция allSubClasses

Данная операция возвращает коллекцию всех дочерних классов.

Операция attributes

Возвращает набор атрибутов данного класса. Например, OCL-выражение

```
Book.attributes
```

в качестве результата вернет список атрибутов класса Book — title, size.

Операция associationEnds

Возвращает список концов ассоциаций (ролей) данного класса. Например, для класса Book эта операция вернет список ролей — published, writed_by, theme.

Обработка типов для коллекций

Операция filterOnType

Операция filterOnType(<метатип>) вернет подмножество, тип элементов которого удовлетворяет типу входного параметра.

Операции логического выбора

В языке OCL существует операция логического выбора, позволяющая формировать результат OCL-выражения в зависимости от задаваемых разработчиком логических условий. По синтаксису эта операция близка аналогичным операторам известных языков программирования и имеет следующий вид:

```
if <условие>
then <выражение 1>
else <выражение 2>
endif
```

Результатом данной операции является OCL-выражение, которое выбирается из двух возможных выражений, исходя из истинности или ложности задаваемого в операции условия.

Существенным отличием синтаксиса данной операции от аналогов является обязательность наличия выражения 2, то есть ветки «else..». Рассмотрим два OCL-выражения:

```
Book.allInstances
if (publisher.country.name='Россия') then 'Российская'
else 'Иностранная'
endif
```

Предположим, что второе выражение работает с контекстом первого (то есть с контекстом класса Book). В качестве результата работы второго выражения в этом случае мы получим список строк, содержащий слова «Российская» или «Иностранная» в зависимости от места издания книги. Размер полученного списка будет совпадать с общим количеством книг.

Необходимо иметь в виду: кажущаяся схожесть рассматриваемой операции с аналогичными операторами языков программирования не означает, что OCL мо-

жет содержать фрагменты программной реализации. Каким бы сложным ни было содержание рассматриваемой операции, в результате ее мы всегда имеем просто конкретное OCL-выражение, а не изменение хода выполнения программы.

Прочие операции

Операции над строками

OCL имеет развитые средства работы со строками, обеспечивающие гибкие возможности по обработке строковых элементов.

Операция Concat

Операция Concat(<строка>) «склеивает» операнд со строкой-параметром. Например, следующее OCL-выражение:

```
Author.allInstances.surname.concat(' автор')
```

вернет коллекцию строк, каждая из которых образована фамилией автора и добавленной далее строкой ' автор'. Отметим, что эту операцию полностью заменяет оператор сложения (+), так что следующее выражение полностью эквивалентно рассмотренному выше:

```
Author.allInstances.surname+' автор'
```

Операция pad

Операция pad(<целое>,<символ>) изменяет длину строки (количество символов) путем добавления символа-параметра в начало строки, с тем чтобы общая получившаяся длина строки была не меньше заданного параметра-целого. Если исходная длина строки равна или превышает целое-параметр, то никаких изменений не производится.

Например, следующее OCL-выражение

```
'имя'.pad(5,'-')
```

вернет строку '--имя', длина которой равна 5.

Операция postPad

Операция postPad(<целое>,<символ>) аналогична предыдущей, только символы-параметры добавляются к концу строки.

Например, OCL-выражение

```
'возраст'.postPad(8,'_')
```

вернет строку 'возраст_', длина которой равна 8.

Операции sqlLike и sqlLikeCaseInsensitive

Операции sqlLike(<строка>) и sqlLikeCaseInsensitive(<СТроКа>), как правило, используются совместно с операцией выборки Select, и обеспечивают поиск записей, содержащих заданную строку-параметр. При этом часто применяется символ маски поиска (%). Например, следующее OCL-выражение

```
Author.allInstances->select(surname.sqlLike('%ac%')
```

оставит в результирующей коллекции только тех авторов, в фамилии которых встречается подстрока 'ac'. В этом смысле данные операции соответствуют ANSI SQL оператору Like. Отличие операции sqlLikeCaseInsensitive в том, что отбор осуществляется без учета регистра символов.

Операция substring

subString(<целое>,<целое>) возвращает подстроку, образованную символами исходной строки, при этом позиции начального и конечного символов подстроки задаются входными параметрами. Например, OCL-выражение

```
'паровоз'.subString(3,5)
```

вернет строку 'ров'.

Если заданный номер конечного символа превышает длину исходной строки, то он игнорируется, и возвращается последний символ исходной строки, то есть выражение

```
'паровоз'.subString(5,19)
```

вернет строку 'воз'.

Операции toLower и toUpper

Эти операции приводят все символы строк-операндов к нижнему (toLower) или верхнему (toUpper) регистру.

Операции преобразования строк к другим типам

Данные операции (табл. 5.4) предназначены для преобразования типов. При невозможности преобразования вырабатывается исключение.

Таблица 5.4. Операции преобразования типов

Операция	Тип возвращаемого результата
strToInt	Целое
strToTime	Время
strToDate	Дата
strToDateTime	Шаблон Дата-Время

Операции с датами и временем

Форматы дат и времени

В OCL используется ISO-стандарт для представления дат и времени. Для даты применяется нотация <#год-месяц-день>, например, #2003-11-26. Для времени используется нотация <#часы:минуты:секунды>, например, #02:23:45. Для дат и времени допустимы описанные выше операции сравнения например OCL-выражение

```
Author.allInstances->select(birthDate<#1980-01-01)
```

вернет в качестве результата коллекцию авторов, родившихся после 1 января 1980 года.

Операция inDateRange

Операция inDateRange(<дата 1>,<дата 2>) возвращает логическое значение True, если входная дата находится в диапазоне дат между <дата 1> и <дата 2>, в противном случае результатом операции будет являться логическое значение False.

Операция sumTime

Данная операция, примененная к коллекции, все элементы которой имеют тип Время, возвращает суммарное количество времени.

Использование OCL

Задание выражений для вычисляемых атрибутов

Язык OCL часто используется при формировании выражений для вычисляемых (derived) атрибутов. Например, в классе Theme рассматриваемой UML-модели существует вычисляемый атрибут number_of_books (количество всех книг по данной тематике). В этом случае при формировании модели для этого атрибута может быть записано следующее OCL-выражение:

```
books->size
```

При первом обращении к данному атрибуту (например, при отображении его значения) будет оперативно рассчитано его значение.

Формирование вычисляемых атрибутов в Rational Rose

Для формирования вычисляемых атрибутов в UML-редакторе CASE-системы Rational Rose необходимо сначала создать новый атрибут, а затем перейти на вкладку Detail в окне спецификации атрибута (рис. 5.2), где установить флажок Derived.

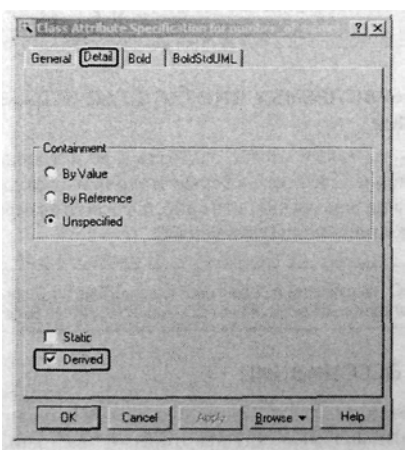


Рис. 5.2. Установка признака вычисляемого атрибута

Тем самым мы сообщили среде моделирования, что данный атрибут — вычисляемый.

Кроме этого, мы должны сформировать OCL-выражение, в соответствии с которым будет вычисляться значение данного атрибута. Для этого необходимо перейти на вкладку Bold и в открывшемся списке тег-параметров выбрать параметр DerivationOCL (рис. 5.3). После двойного щелчка по данной строке откроется окно редактирования для ввода OCL-выражения. Для рассматриваемого атрибута number_of_books введем выражение books->size, то есть общее количество книг. После закрытия окна спецификации информация о вычисляемом атрибуте будет сохранена в UML-модели.

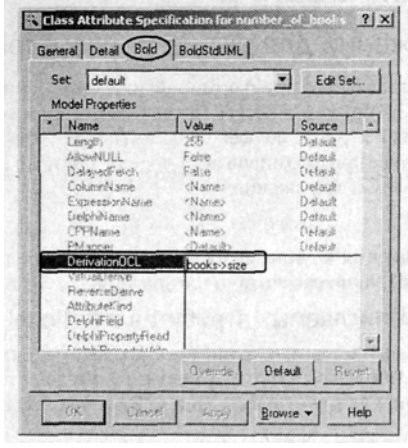


Рис. 5.3. Задание OCL-выражения для вычисляемого атрибута

Формирование вычисляемых атрибутов во встроенном редакторе моделей

Вычисляемые атрибуты также можно задавать и во встроенном редакторе моделей среды Bold for Delphi. Для этого в дереве модели необходимо выбрать нужный атрибут, затем справа на панели свойств (рис. 5.4.) активизировать флажок Derived и ввести OCL-выражение в поле Derivation OCL.

ПРИМЕЧАНИЕ

Для формирования OCL-выражений в редакторе моделей Bold for Delphi вместо «ручного» ввода можно (и удобнее) пользоваться встроенным OCL-редактором. Он будет рассмотрен в главе 8.

Вычисляемые ассоциации

В главе 1, когда мы рассматривали основы языка UML, было описано понятие ассоциации как вида отношений между некоторыми классами. При этом были рассмотрены примеры «обычных» ассоциаций, роли которых фиксированы и заданы. В этом разделе мы познакомимся с *вычисляемыми ассоциациями,* имеющими

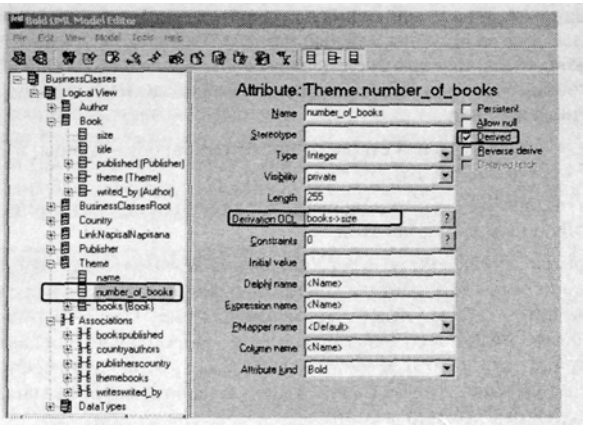


Рис. 5.4. Формирование вычисляемого атрибута средствами Bold

несколько другой характер связи элементов модели. Вычисляемая ассоциация не имеет заранее фиксированных ролей на своих концах, а вместо этого такие роли задаются специальными OCL-выражениями, и их значения «вычисляются» при выполнении приложения.

ПРИМЕЧАНИЕ

Описание понятия вычисляемых ассоциаций требует понимания основ языка OCL. Именно по этой причине это понятие не рассматривалось ранее, в частности в главе 1.

Для лучшего понимания вычисляемых ассоциаций рассмотрим небольшой пример.

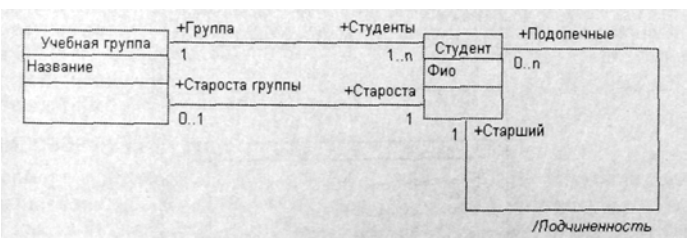


Рис. 5.5. Модель с вычисляемой ассоциацией

Модель (рис. 5.5) включает два класса — Учебная группа и Студент и описывается следующими основными бизнес-правилами:

- в группу входят несколько студентов;
- каждый студент принадлежит только одной группе;
- у каждой группы существует один староста из числа студентов.

Кроме того, в классе Студент присутствует вычисляемая ассоциация Подчиненность. Рассмотрим ее подробнее. Эта ассоциация говорит о том, что у каждого сту-

дента есть один студент, являющийся старшим (роль Старший у вычисляемой ассоциации). С другой стороны, каждый студент может либо не иметь вообще, либо иметь несколько «подопечных» студентов (роль Подопечные вычисляемой ассоциации). С точки зрения описываемой предметной области можно сформулировать два очевидных правила.

1. Для каждого студента старшим является староста группы, в которой данный студент обучается.
2. Для старшего студента подопечными являются студенты той группы, в которой он является старостой.

Эти положения позволяют сформировать OCL-выражения для ролей вычисляемой ассоциации следующим образом. Из правила 1 следует, что для роли Старший можно написать следующее выражение на языке OCL — «группа.староста», где «группа» представляет учебную группу, в которой обучается данный студент, а «.староста» представляет собой студента-старосту этой группы. Стоит подчеркнуть, что здесь мы опять эффективно используем возможности навигации по модели посредством OCL-выражений.

ВНИМАНИЕ

В данном случае мы используем русскоязычные названия элементов модели только для наглядности. При создании реального приложения они должны быть представлены английскими идентификаторами.

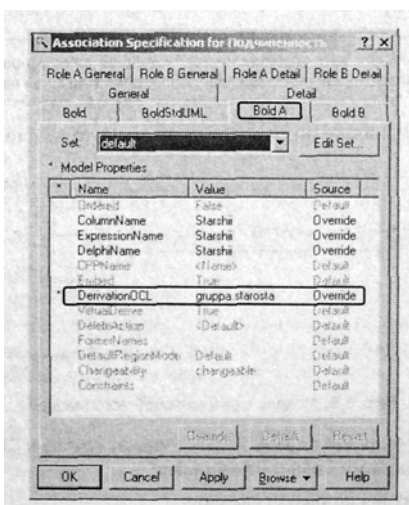


Рис. 5.6. Задание роли Старший в Rational Rose

Аналогично, пользуясь правилом 2, можно написать OCL-выражение для второй роли вычисляемой ассоциации — Подопечные. Оно будет выглядеть следующим образом — «староста группы.студенты», задавая коллекцию студентов той учебной груп-

ны, в которой старший студент является старостой. Таким образом, вычисляемые ассоциации позволяют динамически связывать элементы модели, исходя из уже имеющихся других ассоциаций. Это, безусловно, удобно с точки зрения автоматизации и повышения гибкости разработки. Формирование OCL-выражений для ролей вычисляемых ассоциаций производится аналогично описанным выше правилам для формирования вычисляемых атрибутов. Эти операции можно производить как в Rational Rose (рис. 5.6), так и во встроенном редакторе моделей (рис. 5.7).

При этом необходимо предварительно установить соответствующие флажки Derived, указав, что данная ассоциация является вычисляемой.

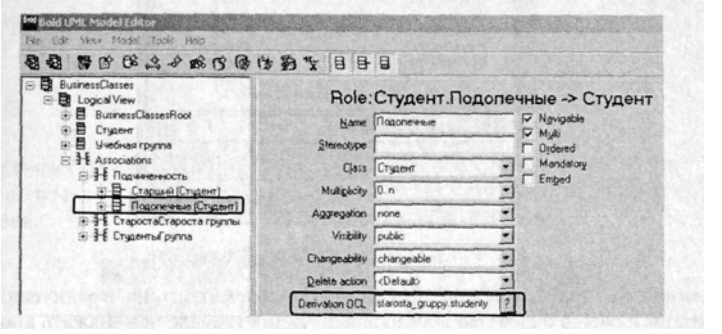


Рис. 5.7. Задание роли Подопечные в Bold

Формирование ограничений (Constraints)

OCL также широко применяется для создания условий-ограничений для элементов модели (собственно, для этой цели OCL и был первоначально разработан). В этом случае для задания каждого ограничения на уровне модели также используются OCL-выражения. Например, для класса Author мы можем написать следующее ограничение surname<>", то есть запретить включения нового автора с «пустой» строкой-фамилией в состав авторов.

Формирование OCL-ограничений в Rational Rose

Для формирования ограничений в Rational Rose необходимо перейти в окно спецификации класса (напомним, что оно вызывается двойным щелчком по изображению класса или из контекстного меню). Затем на вкладке Bold необходимо выбрать тег-параметр Constraints (рис. 5.8) и после двойного щелчка по этой строке ввести в окне редактирования необходимое OCL-выражения. В данном случае для класса Book мы ввели ограничение size>50, означающее, что в книге должно быть больше 50 страниц (напомним, во избежание возможной путаницы, что в данном случае «size» — это название атрибута в классе, а не оператор OCL).

Формирование ограничений во встроенном редакторе моделей Bold for Delphi

Для создания ограничений во встроенном редакторе предназначено специальное окно редактирования (см. рис. 5.6), в котором отображается общее количество

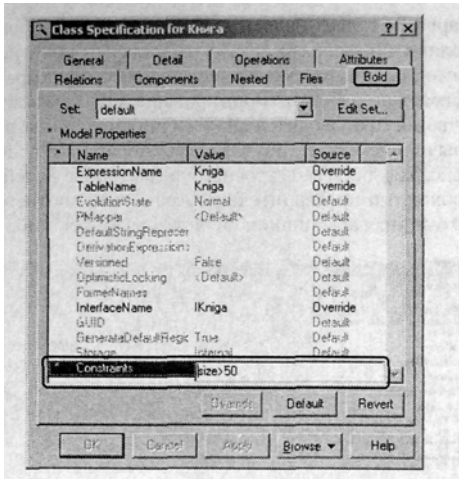


Рис. 5.8. Задание ограничения в Rational Rose

ограничений у текущего элемента модели. Для формирования ограничений необходимо выбрать в дереве модели нужный элемент, после чего нажать кнопку со знаком вопроса, расположенную справа от окошка Constraints (рис. 5.9). В результате откроется специальный редактор ограничений (рис. 5.10). Этот редактор очень прост в использовании. Он содержит две кнопки — для добавления и удаления ограничений, а также поля name и body для ввода разработчиком имени ограничения и OCL-выражения соответственно.

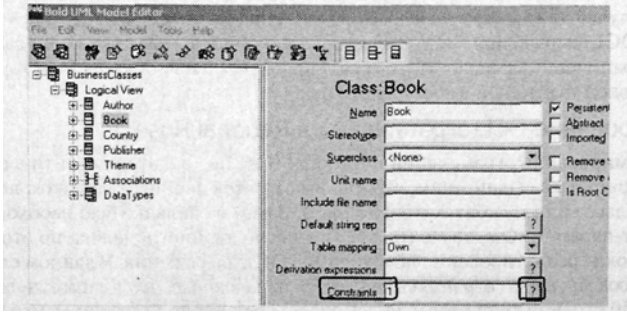


Рис. 5.9. Элементы управления ограничениями

Выделим в дереве модели класс Book и вызовем редактор ограничений. Добавим одно ограничение, назвав его PageAmount (количество страниц), и введем в поле body OCL-выражение size>50 (см. рис. 5.10).

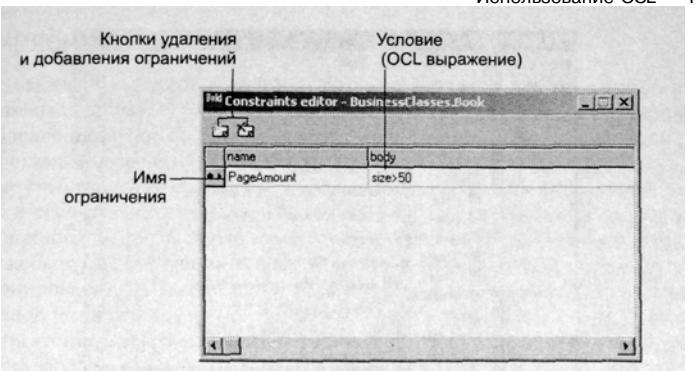


Рис. 5.10. Редактор ограничений Bold for Delphi

Добавим еще одно ограничение Title_not_null, для которого введем OCL-выражение `title<>''` (рис. 5.11). Тем самым мы «запретили» книге иметь пустое название.

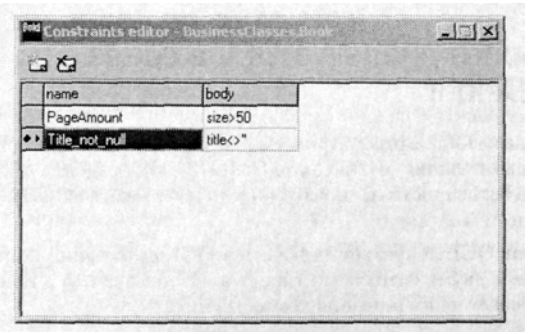


Рис. 5.11. Вид редактора с двумя ограничениями

Таким образом, для любого элемента модели можно при необходимости задать несколько ограничений. При этом возможны ситуации, когда в результате неправильного или ошибочного ввода OCL-выражений (например, при нарушении синтаксиса OCL) они будут «неработоспособны» и приведут к ошибкам выполнения приложения. Для предотвращения таких ситуаций в составе средств встроенного редактора имеется инструмент для проверки корректности (validation) всех OCL-выражений, содержащихся в модели. Для его вызова необходимо выбрать в главном меню редактора Tools ▸ Validate OCL in model (рис. 5.12).

ПРИМЕЧАНИЕ

Такую операцию рекомендуется производить и для проверки корректности OCL-выражений, используемых при формировании вычисляемых атрибутов.

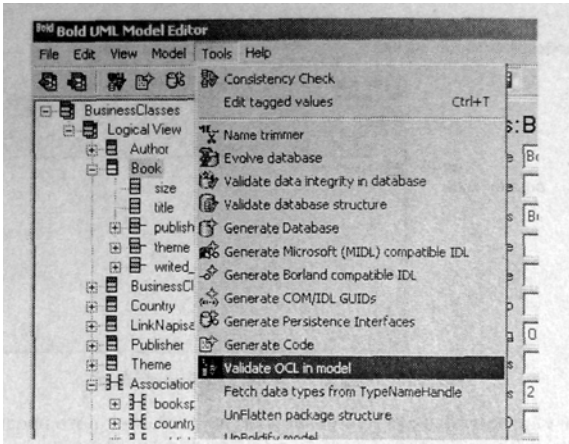


Рис. 5.12. Запуск проверки корректности выражений OCL

Особенности диалекта OCL в среде Bold for Delphi

Реализация языка OCL в рассматриваемой версии Borland MDA (Bold for Delphi) имеет некоторые отличия от стандарта OMG. В этом разделе мы кратко перечислим основные особенности используемого диалекта языка OCL и его отличия от спецификации.

- Стандарт OCL допускает включать в OCL-выражения комментарии, обозначаемые двумя символами тире (--). Используемый в Bold диалект OCL не допускает использования комментариев.
- В составе используемого диалекта OCL присутствует операция allInstances-AtTime(t-параметр), позволяющая получить состояние всех элементов объектного пространства на заданный момент времени, передаваемый в качестве параметра.

ПРИМЕЧАНИЕ

Для использования операции allInstancesAtTime необходимо установить отдельно приобретаемый модуль расширения среды Bold for Delphi, называемый «Object Versioning Extension», который поддерживает «историю» изменений объектного пространства во времени.

- Стандарт OCL использует оператор .size для получения длины строкового атрибута. Bold-диалект OCL для этой цели использует оператор .length.
- Для задания «пустого» начального значения атрибута Bold-диалект OCL допускает использование NULL.

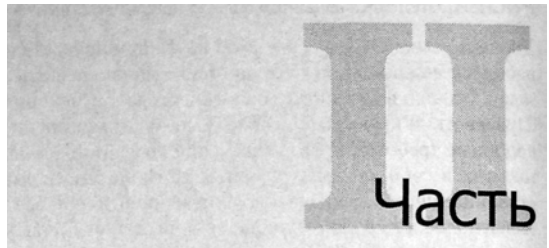
Расширяемость диалекта OCL в Bold

Диалект языка OCL в среде Bold for Delphi предоставляет разработчику богатый набор возможностей, и с его помощью можно решить подавляющее большинство задач. Однако в результате практического применения Bold разработчики в ряде случаев сталкиваются с необходимостью расширения этого набора. Например, подобные требования возникают при создании «смешанных» приложений, сочетающих в себе как инструменты Borland MDA, так и элементы реализации «традиционного» подхода разработки приложений баз данных. По-видимому, учитывая подобного рода ситуации, с одной стороны, а также с целью повышения удобства формирования OCL-выражений, с другой стороны, некоторые компании по собственной инициативе (и совершенно бесплатно, по крайней мере, на момент написания этой книги) разработали и начали предоставлять расширенные наборы OCL-операторов для их использования в среде Bold for Delphi. Такие наборы OCL-расширений поставляются в виде специальных программных модулей. В настоящее время существует возможность свободного расширения диалекта OCL, причем количество дополнительных операторов OCL в этих наборах весьма велико — более 80. Подробнее информация о продуктах сторонних производителей приведена в главе 15.

Резюме

В этой главе мы познакомились с OCL — языком объектных ограничений. Несмотря на свое название, отражающее лишь одну из функций OCL — задание ограничений на элементы модели, очевидно, что средства этого языка охватывают гораздо больший спектр возможностей, предоставляемых разработчику. Имея развитые средства работы с коллекциями, OCL предоставляет очень удобные и гибкие возможности навигации по UML-модели, которые практически делают ненужным использование языка SQL. OCL также обладает необходимыми средствами работы с метаинформацией, то есть с информацией о типах элементов модели. Синтаксис OCL прост и понятен, удобные и прозрачные правила формирования OCL-выражений позволяют получить легко читаемые, лаконичные и одновременно мощные выражения, играющие роль своеобразных «OCL-запросов» к элементам объектного пространства. Показана роль OCL в формировании вычисляемых атрибутов и задания ограничений на элементы UML-модели. Рассмотрено понятие вычисляемых ассоциаций и формирование ролей таких ассоциаций посредством OCL-выражений.

Конечно, мы в этой главе рассмотрели только базовые возможности языка, поскольку полное его описание заняло бы гораздо больше места. Но и сказанного достаточно для понимания важности OCL как одного из основных инструментов MDA вообще и Borland MDA в частности. В последующих главах книги при описании практического использования OCL в ходе разработки MDA-приложений мы не раз будем ссылаться на данную главу, которая в этом смысле является своего рода справочником по языку OCL.



Создание MDA-приложений

Объектное пространство

Понятие объектного пространства (Object Space) является важнейшим элементом в архитектуре Borland MDA. Без понимания основ работы с объектным пространством многие возможности, предоставляемые этой архитектурой, остаются недоступными.

Понятие Object Space

Что же собой представляет объектное пространство (ОП) с точки зрения разработчика MDA-приложений? С одной стороны, его можно рассматривать как своеобразный контейнер, в котором во время выполнения приложения размещаются объекты, представляющие собой реализацию элементов UML-модели. С этой точки зрения, объектное пространство, по сути, является экземпляром модели приложения, аналогично тому, как объект является экземпляром класса в ООП. Это означает, что в объектном пространстве «воплощаются» и наполняются конкретным содержанием все сущности и все связи, присутствующие в модели (классы, атрибуты, ассоциации, роли). Каждый объект, содержащийся в объектном пространстве, таким образом, содержит полную информацию о своих свойствах (атрибутах, операциях и т. д.) и об отношениях-связях с другими объектами.

С другой стороны, объектное пространство представляет собой программный буфер (кэш), который накапливает происходящие в системе изменения, возникающие, например, при удалении, изменении или добавлении объектов, в том числе при реализации транзакционных операций. При этом специальные механизмы отслеживают накопление таких изменений и вычисляют «разницу» между состоянием элементов-объектов в памяти и состоянием уровня данных (базы данных). При вызове операции обновления уровня данных (UpdateDatabase) механизмы объектного пространства обеспечивают синхронизацию содержимого памяти и базы данных. В этом контексте объектное пространство реализует аналог известных

механизмов дотируемого отложенного обновления (CashedUpdates), используемых при работе с СУБД.

Кроме того, объектное пространство содержит специальный механизм отображения данных, называемый «persistence mapper», который служит своеобразным каналом между уровнем данных (реализуемым, например, посредством СУБД) и объектами, содержащимися в Object Space. Таким образом, в архитектуре Borland MDA-приложений объектное пространство занимает центральную часть (рис. 6.1), реализуя следующие основные функции.

- Хранение и представление информации о реализации элементов модели во время работы приложения. Обеспечение целостной логической структуры приложения. Обработка внутренних событий системы.
- Отслеживание происходящих изменений в системе и формирование дельта-информации о накопленных различиях с уровнем данных.
- Взаимодействие с графическим интерфейсом, предоставление информации о состоянии объектов.
- Взаимодействие с уровнем данных, синхронизация состояний памяти и базы данных.

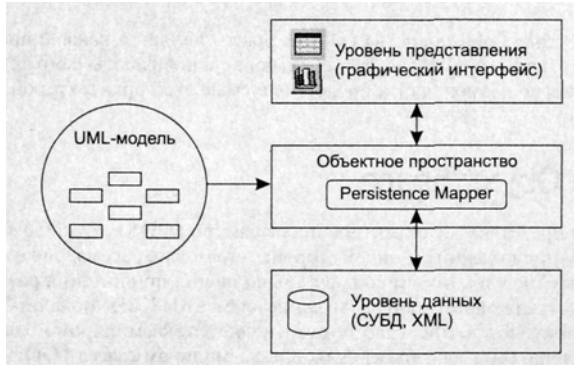


Рис. 6.1. Взаимодействие компонентов Borland MDA

Для обеспечения таких функций в Borland MDA включены специальные классы-носители информации модели, а также классы, реализующие управление элементами объектного пространства, и осуществляющие обработку событий, происходящих в системе. Для иллюстрации решаемых на этом уровне системных задач рассмотрим вопрос контроля и управления жизненным циклом объектов во время работы приложения. При запуске MDA-приложения часть объектов загружается с уровня данных в объектное пространство (необходимость загрузки того или иного объекта может задавать разработчик на этапе создания приложения или пользователь во время его выполнения, эти вопросы будут освещены в следующих главах). При этом часть объектов остается не загруженной в память, однако система считает, что они, тем не менее, существуют, поскольку не были удалены во время прошлых сеансов работы. Таким образом формируется совокупность объектов в па-

мяти и объектов в БД, которая образует так называемое «концептуальное объектное пространство». Если происходит удаление объекта, он, тем не менее, остается в памяти и в БД, пока не будет вызван метод UpdateDatabase. До момента этого вызова такой объект помечается системой как удаленный, но будет продолжать существовать как в памяти, так и в БД, однако, он не будет существовать в концептуальном объектном пространстве, и станет недоступным для использования. Для управления такими ситуациями Borland MDA имеет специальные средства, так называемые *машины состояния (State Machines)*.

Состав и структура ОП

Структура иерархии классов, обеспечивающая функционирование объектного пространства, довольно сложна и объемна, и включает сотни классов. Мы рассмотрим только несколько ее наиболее важных фрагментов. Все классы условно можно разбить на две основные категории — классы внутренние и внешние. Внутренние классы имеют своим предком класс TObject, а внешние — класс TComponent. Фрагмент иерархии внутренних классов представлен на рис. 6.2.

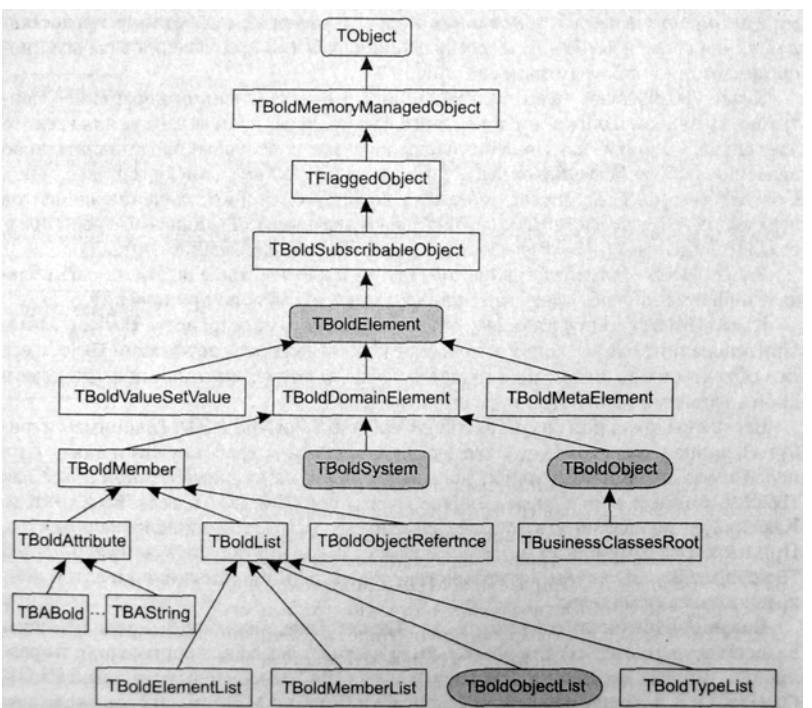


Рис. 6.2. Иерархия внутренних классов в Borland MDA

Кратко опишем некоторые из них.

TBoldMemoryManagedObject является классом-родоначальником этой иерархии классов. Его появление обусловлено тем, что Borland MDA имеет собственный менеджер памяти.

Класс TBoldFlaggedObject предназначен для внутреннего использования и содержит набор битовых значений-флагов, а также набор значений перечисляемого типа, упакованных в целочисленный формат. Флаги в основном используются для сохранения информации о различных состояниях элементов объектного пространства.

Класс TBoldSubscribableObject представляет очень важный для Borland MDA **механизм подписки на события (subscribing).** Он будет рассмотрен позднее, сейчас лишь отметим, что этот класс позволяет реализовать мощную и гибкую поддержку реакций на различные события, происходящие в системе. Практически любые компоненты Borland MDA, в том числе и визуальные компоненты, способны выступать в качестве источников определенных событий, на которые можно «подписаться», и с их наступлением автоматически выполнить заранее заданные реакции — обработчики событий. Механизм подписки эффективно используется самой средой Borland MDA, в частности при работе с вычисляемыми (derived) атрибутами и ассоциациями, но этот механизм также доступен и разработчику, предоставляя уникальные возможности по управлению работой приложения в событийно-организованной программной системе.

Класс TBoldElement является важнейшим в представленной иерархии — центральным звеном в описании объектного пространства, суперклассом для всех его элементов. Он является универсальным классом, способным представлять либо значение, либо собственно объект, либо *метаинформацию* (информацию о типе). Соответственно, TBoldElement имеет трех потомков для представления элементов этих видов — TBoldValueSetValue (представляет значения), TBoldDomainElement (представляет объекты) и TBoldMetaObject (представляет информацию о типах).

Класс TBoldSystem представляет объектное пространство в целом, то есть обладает информацией обо всех экземплярах элементов модели приложения.

Класс TBoldObject представляет объект объектного пространства Borland MDA. При генерации кода все пользовательские классы являются потомками TBoldObject. Все объекты в объектном пространстве также являются потомками этого класса либо дочерними по отношению к его потомкам.

Все члены, входящие в состав любого объекта Borland MDA (например, атрибуты), принадлежат к классу TBoldMember. Однако этот класс служит и для других целей и имеет самостоятельное значение, порождая, например, такой класс как TBoldList, который, в свою очередь, является родителем важного класса TBoldObjectList. Как следует из названия, этот класс служит для представления списка объектов. При этом каждому классу модели соответствует свой объект-экземпляр класса TBoldObjectList объектного пространства, содержащий коллекцию объектов конкретного класса модели.

Важное значение имеет также класс TBoldAttribute, являющийся родительским классом (суперклассом) для нескольких классов, представляющих атрибуты различных типов — строковых (TBAString), даты (TBADate), битовых массивов BLOB (TBABlob) и т. д. Таким образом, мы видим, что Borland MDA имеет свои типы данных—аналоги стандартных Delphi-типов. И это не случайно, поскольку Borland

MDA представляет собой своего рода «систему в системе», то есть самостоятельную программную систему, функционирующую в рамках определенной операционной системы (в настоящее время Windows) и обладающую собственным менеджером памяти, собственным развитым механизмом генерации и обработки событий и другими подобными свойствами. При желании разработчик может создать и зарегистрировать новый MDA-тип данных, воспользовавшись мастером создания атрибутов (в меню Delphi выбрать Bold ▸ Attribute Wizard...).

Теперь кратко рассмотрим иерархию внешних классов (рис. 6.3).

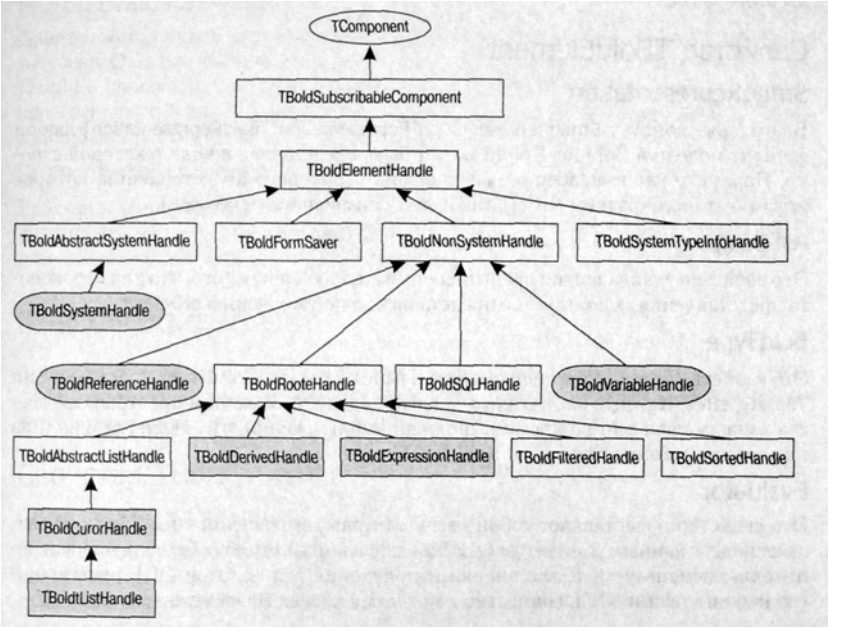


Рис. 6.3. Иерархия внешних классов в Borland MDA

Все внешние классы имеют в качестве родителя класс TComponent. Это, естественно, означает, что большинство этих классов мы увидим на палитре компонентов среды разработки Delphi. Если внутренние классы можно рассматривать как своего рода «скелет», поддерживающий сложный «организм» объектного пространства, то внешние классы скорее являются своеобразными «адаптерами-переходниками» к уровню представления и уровню данных. Эти классы являются также основными инструментами разработчика приложений в Borland MDA. Уровни представления и данных мы рассмотрим в дальнейшем более детально, а сейчас ооратим внимание на то, что подавляющее большинство внешних классов содержит в своем названии термин «Handle» — описатель, или *дескриптор*. Дескрипторам и работе с ними будет посвящена следующая глава.

Класс TBoldElement

Данный класс, безусловно, можно считать основным в иерархии внутренних классов, формирующих объектное пространство. Он является суперклассом для всех элементов модели, обладая способностью представлять не только конкретное значение, но также объекты и метаинформацию (информацию о типах). Основные методы, включенные в класс TBoldElement, также имеют важное значение во всех его дочерних классах. Далее мы рассмотрим некоторые основные свойства и методы этого класса.

Свойства TBoldElement

StringRepresentation

Благодаря свойству StringRepresentation[Representation:TBoldRepresentation] любой элемент модели в Bold for Delphi может быть представлен в виде текстовой строки. Параметр Representation определяет вид строкового представления, которое может быть как кратким (brief), так и многосимвольным (verbose).

AsString

Это свойство эквивалентно предыдущему за исключением того, что для строкового представления используется представление по умолчанию brDefault.

BoldType

Это свойство возвращает информацию о типе элемента и само при этом имеет тип TBoldTypeInfo. Каждый экземпляр элемента модели в объектном пространстве всегда «знает» свой тип во время выполнения приложения, что является одной из ключевых особенностей среды Bold for Delphi.

Evaluator

Это свойство представляет собой «вычислитель» выражений типа TBoldEvaluator, связанных с данным элементом ОП. Каждый элемент имеет собственный «вычислитель», используемый для оценки и получения результатов OCL-выражений (об использовании OCL совместно с методами класса TBoldElement см. главу 8).

Mutable

Логическое свойство mutable отвечает за возможность модификации данного элемента. Оно используется для проверки возможности изменения элементов ОП. Примером элементов, для которых это свойство, как правило, принимает значение True, являются атрибуты объекта класса. А у элементов, представляющих метаинформацию, это свойство всегда имеет значение False, так как во время работы приложения запрещено изменять собственно модель или ее типы. Любой элемент можно сделать «неизменяемым» путем вызова метода MakeImmutable. Однако обратную операцию после этого реализовать невозможно, поскольку среда Bold for Delphi отслеживает появление таких «неизменяемых» элементов, и, например, с целью оптимизации времени выполнения, может «по своему усмотрению» поместить их в специальный кэш, копировать в другую область памяти и т. д.

Методы TBoldElement

Метод Assign

Метод Assign(Source: TBoldElement) используется, по аналогии с известными классами Delphi, для копирования значения элемента-источника ОП в текущий элемент. При копировании осуществляется проверка на идентичность типов обоих элементов.

Метод EvaluateExpression

Данный метод очень часто применяется при программировании операций с элементами ОП. Его наличие отражает еще одну ключевую особенность Bold for Delphi — способность каждого элемента ОП вычислять OCL-выражения и возвращать результат в виде элемента ОП. Использование этого и аналогичных ему методов будет рассмотрено более подробно в главе 8.

SubscribeToExpression

Еще один очень важный метод класса TBoldElement. Назначение этого метода — «подписаться» на изменение элемента ОП, заданного OCL-выражением-параметром. Механизм «подписки» (subscribing) будет более подробно описан в главе 14.

Класс TBoldSystem

Здесь мы подробнее познакомимся с некоторыми свойствами и методами класса TBoldSystem, поскольку именно в нем сосредоточена реализация и управление объектным пространством в целом.

Свойства TBoldSystem

Свойство DirtyObjects

Свойство DirtyObjects: TList определяет список всех объектов, которые были изменены в памяти, но еще не сохранены в базе данных. При попытке завершения приложения с непустым состоянием указанного списка будет выработано соответствующее исключение.

Свойство PersistenceController

Свойство PersistenceController: TBoldPersistenceController определяет контроллер уровня данных для объектного пространства. Если указанное свойство пусто, то все объектное пространство является временным (transient), то есть его состояние не будет сохраняться в уровне данных.

Свойство Classes

Classes[index:Integer]: TBoldObjectList определяет список всех объектов класса с заданным индексом. Все классы модели пронумерованы начиная с 0.

Свойство ClassByExpressionName

Свойство ClassByExpressionName[constExpressionName:string]: TBoldObjectList определяет список всех объектов класса с заданным именем.

Методы TBoldSystem

Метод MakeDefault

Приложение может включать несколько объектных пространств (ОП), одно из которых является ОП по умолчанию. Данный метод, примененный к конкретному экземпляру, переводит текущее объектное пространство в статус «по умолчанию».

Метод StartTransaction

Метод StartTransaction(Mode: TBoldSystemTransactionMode = stmNormal) инициализирует последовательность действий, составляющих транзакцию. Режим транзакции определяется параметром Mode. Кроме значения по умолчанию (stmNormal) можно использовать значение stmUnSafe. При этом скорость обработки данных возрастает за счет исключения действий по проверке состояния объектов и ассоциаций, однако повышается вероятность нарушения функционирования. Транзакции могут быть вложены в другие транзакции. В этом случае внешняя транзакция будет считаться завершенной только в случае завершения всех вложенных транзакций. Для завершения транзакции используется метод CommitTransaction (см. далее в этом разделе), а для отката изменений — метод RollBackTransaction.

Метод CommitTransaction

CommitTransaction (Mode: TBoldSystemTransactionMode = stmNormal) завершает транзакцию, начатую методом StartTransaction. Если по какой-то причине транзакция не может быть завершена, то вырабатывается исключение. Это происходит в двух случаях. Во-первых, если данная транзакция включает в себя вложенные транзакции и хотя бы одна из них не была завершена, то есть был произведен откат изменений. Во-вторых, существуют специальные виртуальные методы MayCommit и связанные с ними события bqMayCommit, которые вызываются при изменении любого элемента, участвующего в транзакции. Если хотя бы один из вызовов этих методов вернет False, то при попытке завершения транзакции также будет выработано исключение. Для удобства обработки подобных ситуаций можно воспользоваться специальным методом TryCommitTransaction, который возвращает True, если данная транзакция может быть завершена, или False, если завершение невозможно. В последнем случае данный метод обеспечивает автоматический откат изменений (вызывает RollBackTransaction).

Метод UpdateDatabase

Производит синхронизацию ОП с уровнем данных (БД). Все объекты, составляющие рассмотренный выше список DirtyObjects, то есть добавленные, удаленные или измененные с момента прошлого вызова данного метода, сохраняются в базе данных или документе XML.

Работа с объектным пространством

Чтобы описанная выше иерархия классов Borland MDA не выглядела чисто абстрактной схемой, оторванной от реальности, полезно рассмотреть на конкретных примерах, как на практике происходит работа с этими классами. Во всех примерах

этой главы будут продемонстрированы методы работы с ОП, не требующие использования языка OCL. Расширенные методики работы с внутренними и внешними классами Bold for Delphi, применяющие OCL, будут рассмотрены в главе 8. В качестве приложения-основы для иллюстрации методов работы с элементами ОП мы будем использовать созданное ранее простое приложение, описывающее библиотечный каталог (см. главу 3).

Программное управление атрибутами объектами ОП

ПРИМЕЧАНИЕ

Среда Bold for Delphi поддерживает разработку приложений в двух режимах — без использования кодогенератора и с генерацией кода классов модели. Особенности работы с последним режимом будут рассмотрены в главе 12. Во всех остальных главах, включая данную, описываются приемы работы без использования генерации кода.

Поставим следующую задачу — программно, во время работы приложения, получить фамилию первого автора в каталоге. Для достижения этой цели лучше всего подойдет класс TBoldObjectList, обеспечивающий доступ к коллекции объектов конкретного класса. Каждый экземпляр класса TBoldObjectList представляет собой список объектов (экземпляров) определенного класса модели приложения. Здесь можно провести некоторую грубую аналогию с базой данных — то есть список объектов класса TBoldObjectList (на самом деле — указателей на объекты) можно рассматривать как таблицу записей БД. Продолжая аналогию, мы можем условно представить, что каждый объект в списке имеет набор атрибутов, так же как каждая запись в таблице БД имеет набор полей. Приступим к решению поставленной задачи. В качестве класса-источника информации будем использовать класс с названием (Expression Name) Author. Для доступа к списку всех объектов класса Author применяем выражение:

```
BoldSystemHandle1.System.ClassByExpressionName['Author'];
```

В этом выражении использованы следующие классы и их атрибуты:

- BoldSystemHandle1 — компонент Borland MDA, отвечающий за управление объектным пространством в целом;
- System — свойство компонента BoldSystemHandle1 типа TBoldSystem (см. рис. 6.2);
- ClassByExpressionName — метод класса TBoldSystem, возвращающий выражение типа TBoldObjectList, в данном случае — список всех объектов-экземпляров класса Author нашей модели.

Класс TBoldObjectList имеет свойство BoldObjects[i] типа TBoldObject, указывающее на объект в списке с порядковым номером i (при этом нумерация объектов в списке начинается с 0). В свою очередь, класс TBoldObject обладает свойством BoldMemberByExpressionName['имя_члена'], позволяющим получить доступ к заданному атрибуту (члену класса). Теперь мы можем сформировать полное выражение-оператор для решения нашей задачи:

```
BoldSystemHandle1.System.ClassByExpressionName['author'].BoldObjects[0]
.BoldMemberByExpressionName['aname'].AsString;
```

Для проверки работоспособности полученного оператора добавим на форму одну кнопку Button1 и обычную метку Label1, после чего напишем следующий обработчик события нажатия кнопки:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
label1.Caption:=BoldSystemHandle1.System.ClassByExpressionName['author']
.BoldObjects[0].BoldMemberByExpressionName['aname']
.AsString;
end;
```

Добавим несколько авторов, пользуясь Bold-навигатором (см. главу 3), и, нажав на кнопку, убедимся, что наш оператор решает поставленную задачу (рис. 6.4).

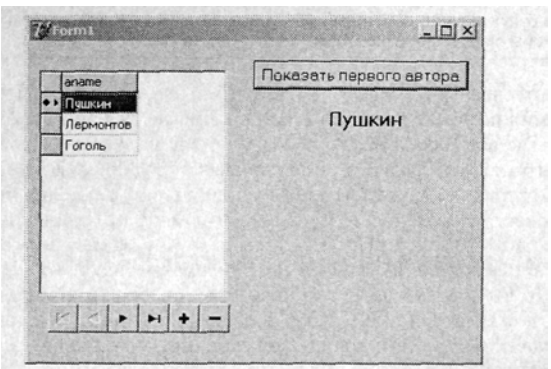


Рис. 6.4. Программное отображение атрибута объекта

Теперь усложним задачу — попробуем программно изменить фамилию первого автора. Для этого добавим на форму поле редактирования Editl и вторую кнопку Button2 с названием Изменить. Напишем следующий обработчик нажатия для второй кнопки:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
 BoldSystemHandle1.System.ClassByExpressionName['author']
.BoldObjects[0].BoldMemberByExpressionName['aname']
.AsString:=Edit1.Text;
end;
```

Легко видеть, что мы используем для редактирования то же самое выражение, но в данном случае присваиваем ему строку текста из окна редактирования. После запуска приложения введем в поле Editl фамилию Носов и, нажав новую кнопку, увидим, что первый автор изменился. Снова нажав первую кнопку, так же убедимся, что из объектного пространства мы получаем уже изменившегося автора (рис. 6.5).

Обратите внимание, что мы не использовали генерацию кода приложения, и по этой причине не имеем описания классов модели на уровне программного кода, однако получили доступ к необходимому атрибуту конкретного объекта по имени во время выполнения приложения. Это еще раз показывает, что объектное про-

странство является экземпляром модели и во время работы приложения обладает информацией обо всех ее элементах. На основе простого разобранного примера можно решать и более серьезные задачи, так как мы по сути уже сейчас получили возможность манипулирования атрибутами объектов-элементов коллекции авторов.

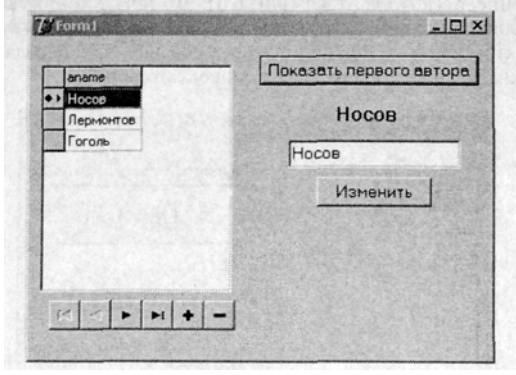


Рис. 6.5. Программное изменение атрибута объекта

Программное управление объектами ОП

Можно пойти дальше и, например, попытаться программно добавлять авторов в коллекцию. Для этого используем метод InsertNew (i:integer) класса TBoldObjectList, где *i* — порядковый номер элемента коллекции (напомним, что нумерация производится от 0). Поставим задачу — по нажатию кнопки добавить нового автора в конец списка и присвоить ему имя, введенное пользователем. Для этого поместим на форму еще одно окно редактирования Edit2 для ввода фамилии добавляемого автора и третью кнопку Button3 с надписью Добавить. Напишем следующий обработчик нажатия новой кнопки:

Листинг 6.1. Добавление нового элемента в коллекцию авторов

```
procedure TForm1.Button3Click(Sender: TObject);
var author_count : integer;
begin
author_count:=boldsystemhandle1.
System.ClassByExpressionName['author'].Count;
boldsystemhandle1.System.
ClassByExpressionName['author'].InsertNew(author_count);
BoldSystemHandle1.System.ClassByExpressionName['author']
.BoldObjects[author_count].BoldMemberByExpressionName['aname']
.AsString:=Edit2.Text;
end;
```

Рассмотрим приведенный код. Чтобы поместить нового автора на последнее место в коллекции, необходимо получить информацию о количестве ее элементов. Для этой цели используется целая переменная author_count и свойство Count класса TboldObjectList, возвращающее количество элементов списка. Далее производится

вызов метода InsertNew с параметром author_count, благодаря чему новый элемент создается в конце списка. И наконец, как и в предыдущем случае, содержимое поля редактирования Edit2 присваивается элементу коллекции авторов, однако не первому ее элементу с индексом 0, а последнему, с индексом author_count. Запустив наше приложение, легко убедиться (рис. 6.6), что новая кнопка Добавить теперь с успехом заменяет функциональную кнопку Bold-навигатора, отвечающую за добавление объектов, и при этом дополнительно обеспечивает передачу имени нового автора из окна редактирования Edit2 в список авторов.

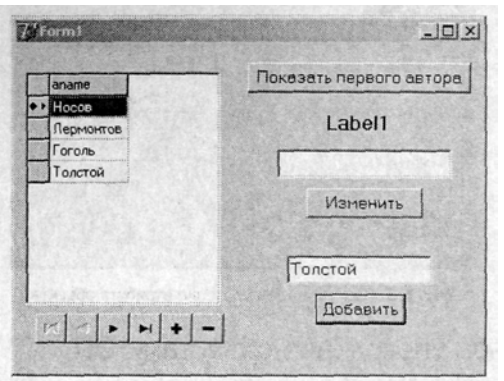


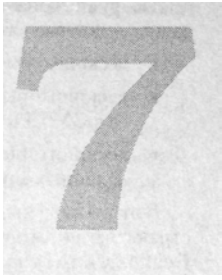
Рис. 6.6. Программное добавление объекта

Рассмотренные примеры можно легко модифицировать и оформить в виде процедур, которые в дальнейшем могут быть использованы для самых разных целей, например для автоматического программного заполнения базы данных какой-либо информацией или замены функций стандартного компонента BoldNavigator собственными или сторонними компонентами и т. п.

Резюме

В этой главе мы познакомились с основами построения объектного пространства в среде Bold for Delphi. Объектное пространство базируется на наборах классов, которые можно условно разделить на внутренние и внешние классы (компоненты). Описаны основные свойства и методы двух базовых внутренних классов Bold — TboldElement и TboldSystem. Также продемонстрированы приемы работы с объектами классов и их атрибутами из программного кода, без задействования специфических Bold-компонентов. Такие методы работы основаны на возможностях класса TBoldObjectList. Рассмотренные простые примеры программного кода призваны помочь начинающим разработчикам «не бояться» работать на программном уровне с инструментами Bold for Delphi и демонстрируют, что, несмотря на большое количество классов, программная система Bold for Delphi базируется на сравнительно небольшом «корневом» наборе основных элементов, к которому относятся и классы, рассмотренные в данной главе.

Дескрипторы (Handles)



Роль дескрипторов

При практической разработке MDA-приложений в среде Delphi 7 дескрипторы занимают, пожалуй, центральное место. Какие бы функции разработчик не реализовывал, будь то вывод информации из объектов в табличном виде, сортировка или фильтрация данных, специальные запросы к СУБД и так далее — все эти функции реализуются посредством дескрипторов. Когда мы в главе 3 разрабатывали простое MDA-приложение, мы тоже использовали дескрипторы списков TBoldListHandle. Все дескрипторы, представленные на палитре компонентов Bold for Delphi, являются невизуальными компонентами. Их основная функция — обеспечение разнообразных информационных связей между объектным пространством, с одной стороны, и графическим интерфейсом и уровнем данных, с другой стороны. Дескрипторы образуют большинство основных элементов структуры внутренних классов, рассмотренных в предыдущей главе, и все они являются потомками класса TBoldElementHandle, который, в свою очередь, происходит от класса TComponent.

Классификация дескрипторов

Дескрипторы BMDA подразделяются на два основных типа— *корневые (root)* и *производные (rooted)*. Корневые дескрипторы являются, как легко догадаться по их названию, первичными источниками информации об объектном пространстве, в то время как любой производный дескриптор обязательно обладает свойством RootHandle (корневой дескриптор). В качестве значения этого свойства не обязательно должен выступать корневой дескриптор, то есть производные дескрипторы могут объединяться в цепочки, при этом свойство RootHandle последующего члена цепочки указывает на предыдущий член. Однако у первого члена цепочки это свойство должно указывать на корневой дескриптор.

По функциональному назначению все дескрипторы можно разделить на следующие основные категории:

- дескрипторы ОП — обеспечивают взаимодействие с объектным пространством;
- дескрипторы уровня данных — обеспечивают функционирование интерфейса с СУБД и XML;
- дескрипторы удаленного ОП — обеспечивают функционирование распределенных многозвенных приложений.

В этой главе мы сосредоточим свое внимание на дескрипторах объектного пространства, которые представлены на вкладке BoldHandles палитры компонентов Delphi. К ним относятся следующие компоненты.

Корневые дескрипторы:

- BoldSystemHandle — представляет объектное пространство в целом;
- BoldReferenceHandle — представляет ссылку на объект ОП;
- BoldVariableHandle — обеспечивает хранение информации о переменной ОП.

Производные дескрипторы:

- BoldSystemTypeInfoHandle — представляет информацию об UML-модели;
- BoldExpressionHandle — представляет результат OCL-запроса, примененного к корневому дескриптору;
- BoldDerivedHandle — представляет результат обработки информации от корневого дескриптора;
- BoldListHandle — представляет список объектов, возвращая текущий элемент;
- BoldSQLHandle — обеспечивает реализацию SQL-запросов к уровню данных;
- BoldCursorHandle — преобразует значение в коллекцию элементов;
- BoldOCLVariables — вспомогательный компонент, обеспечивающий «внедрение» ОП-переменных в OCL-контекст;
- BoldUnloaderHandle — обеспечивает автоматическую выгрузку неиспользуемых объектов из ОП.

Далее в этой главе мы рассмотрим все перечисленные компоненты, за исключением BoldSQLHandle, описание которого приведем в главе 10, посвященной работе с уровнем данных. Последовательность дальнейшего описания компонентов-дескрипторов ОП соответствует порядку их расположения на палитре компонентов Delphi.

BoldSystemHandle

Этот компонент присутствует в каждом приложении, созданном с использованием Borland MDA для Delphi 7. Он фактически представляет в IDE (интегрированной среде разработки Delphi) описанный в предыдущей главе класс TBoldSystem. Данный компонент, таким образом, «отвечает» за объектное пространство в целом. Внешние интерфейсы компонента представлены свойствами-компонентами

(рис. 7.1) SystemTypeInfoHandle и PersistenceHandle. При этом значения этих свойств являются ссылками на соответствующие компоненты BMDA.

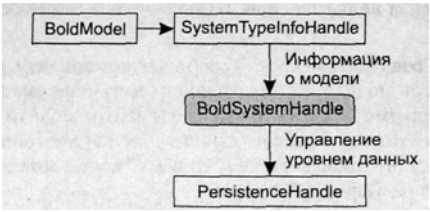


Рис. 7.1. Структура информационных связей компонента BoldSystemHandle

Компонент, указанный в свойстве SystemTypeInfoHandle, поставляет информацию о типах элементов UML-модели. Компонент, указанный в свойстве PersistenceHandle обеспечивает связь с уровнем данных. Если такой компонент не указан, то ОП является полностью транзиентным и не будет сохранять свое состояние между сеансами работы. Таким образом, BoldSystemHandle, обладая информацией о модели на этапе выполнения приложения, обеспечивает взаимодействие ОП с уровнем данных (СУБД), занимая тем самым центральное место в организации функционирования объектного пространства в целом. Приложение может использовать несколько таких компонентов, при этом формируется соответственно несколько ОП. Компонент BoldSystemHandle имеет следующие свойства:

- isDefault (логическое) — задает статус «по умолчанию» для ОП, представленного данным компонентом;
- Active (логическое) — задает режим активности или пассивности ОП; при активизации ОП, в частности, происходит активизация уровня данных, подключение к данным и их загрузка; как правило, на этапе разработки этому свойству рекомендуется присвоить значение False, а для автоматической активизации ОП использовать следующее свойство;
- AutoActivate (логическое) — определяет, будет ли ОП автоматически активизироваться «по первому требованию». Например, если какой-то визуальный компонент подключен к данному ОП, и создается форма, содержащая такой компонент, то будет предпринята попытка активизировать ОП, если данное свойство имеет значение True;

Данный компонент, как правило, является основным источником (RootHandle) для других производных дескрипторов ОП, используемых в приложении.

BoldSystemTypeInfoHandle

Данный компонент-дескриптор предназначен для реализации одной главной функции — поставки информации об UML-модели дескриптору BoldSystemHandle, рассмотренному в предыдущем разделе (см. рис. 7.1). Информация о модели содержит в основном метаинформацию, то есть описание типов элементов модели. При этом данный компонент получает метаинформацию из компонента BoldModel, пред-

ставляющего модель приложения, и преобразует ее в специальный формат, оптимизированный с точки зрения скорости обработки. Единственным внешним интерфейсом компонента является свойство-компонент BoldModel. Другие свойства перечислены ниже.

- UseGeneratedCode (логическое) — сообщает компоненту, был ли сгенерирован код классов модели, от этого зависит метаинформация о модели. Если генерация кода имела место, то имена элементов модели эквивалентны сгенерированным именам классов (с учетом тег-параметров, см. главу 4). Если генерация кода не производилась, то все объекты модели будут формироваться как типы TObject и TObjectList;
- CheckCodeCheckSum — указывает компоненту, нужно ли при старте сравнивать контрольную сумму кода с ее исходным значением, сформированным при генерации кода. Такая проверка служит целям дополнительного обеспечения надежности. Это свойство игнорируется, если генерация кода не производилась.

BoldExpressionHandle

Предназначен для обработки информации от корневого дескриптора посредством OCL-выражения. Может функционировать в составе цепочки дескрипторов. Для иллюстрации работы с этим компонентом обратимся к примеру, рассмотренному при создании простого выражения (см. главу 3).

Поместим на форму компонент BoldExpressionHandlel с вкладки BoldHandles и визуальный компонент BoldLabell с вкладки BoldControls (рис. 7.2).

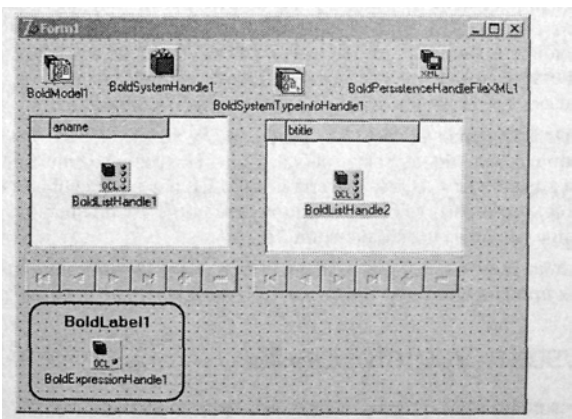


Рис. 7.2. Вид формы с новыми компонентами

Поставим следующую задачу — отображать на форме посредством метки BoldLabell общее количество авторов. Для решения этой задачи подсоединим компо-

нент BoldExpressionHandle1 к корневому дескриптору, установив свойству RootHandle значение BoldSystemHandle1 (рис. 7.3) — этим мы укажем новому компоненту-дескриптору, что источником информации для него будет являться системный дескриптор. Введем следующее OCL-выражение:

```
Author.allInstances->size.asString
```

в свойство Expression нашего компонента (см. рис. 7.3). Это выражение, как мы уже знаем (см. главу 5), возвращает общее количество авторов, преобразованное к строковому типу.

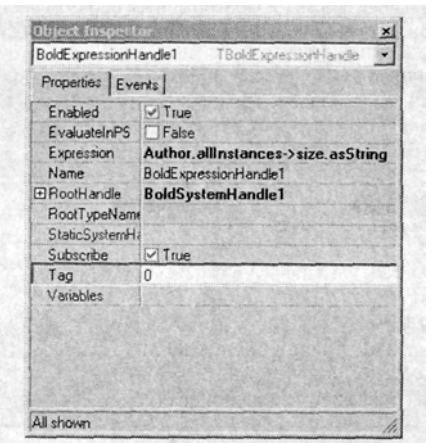


Рис. 7.3. Настройка дескриптора BoldExpressionHandle

Для того чтобы метка BoldLabell отображала нужную информацию, подключим ее к нашему дескриптору с помощью задания ее свойству BoldHandle значения BoldExpressionHandlel (рис. 7.4).

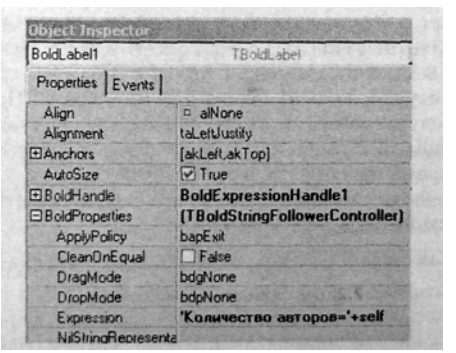


Рис. 7.4. Настройка свойств метки BoldLabel

Далее, в подсвойство Expression свойства BoldProperties введем следующее OCL-выражение:

```
'Количество авторов='+self
```

В данном случае результат будет суммой двух строк, последняя из которых — self - возвращает контекст OCL-выражения (см. главу 3). Поскольку в качестве источника информации выбран BoldExpressionHandle1, постольку вместо self будет подставлен результат, возвращаемый выражением, заданным для этого дескриптора, то есть количество авторов.

Кроме этого, для нашей формы можно обычным образом настроить цвет и шрифт. После запуска приложения мы убедимся, что поставленная задача решена (рис. 7.5).

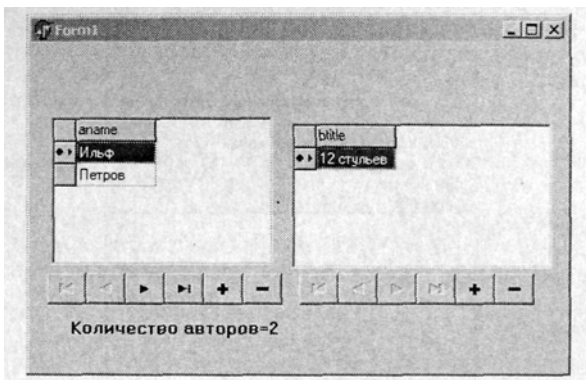


Рис. **7.5.** Вид работающего приложения

Более того, добавляя или удаляя в процессе работы авторов с помощью навигатора, легко заметить, что наша метка «отслеживает» все изменения, отображая корректную информацию.

Может возникнуть закономерный вопрос: а почему мы не можем обойтись без нового дескриптора выражения BoldExpressionHandle, подключив нашу метку к уже имеющемуся дескриптору списка авторов BoldListHandle1? Ответ прост: специфика описателя списка BoldListHandle такова, что в качестве контекста он возвращает всегда только один текущий элемент списка, а не весь список. Поэтому OCL-операция ->size в этом случае возвратит значение 1, а не количество элементов списка. Однако в описанном простом примере все же имеется возможность обойтись без нового дескриптора. Дело в том, что нашу метку BoldLabel1 можно просто подключить непосредственно к системному дескриптору BoldSystemHandle1, а OCL-выражение для метки записать в следующем виде:

```
'Количество авторов='+Author.allInstances->size.asString
```

Легко проверить, что в этом случае поставленная задача также решается. Но такая простая ситуация, как в нашем примере, встречается редко. В следующей главе, посвященной использованию OCL, мы рассмотрим более сложный пример и дополнительно продемонстрируем, в том числе и использование дескриптора

выражения. Сейчас лишь отметим, что непосредственное подключение визуальных элементов (меток и т. д.) к системному дескриптору не является удачной практикой. Интуитивно ясно, что системный дескриптор, представляющий объектное пространство в целом, не стоит использовать для таких «мелких» задач, как формирование меток. Для этих целей предназначены другие дескрипторы, использование которых позволяет структурировать приложение, введя определенные уровни представления информации по принципу «от общего — к частному».

Остальные свойства описываемого компонента относятся к механизмам взаимодействия с уровнем данных и механизмам «подписки» (subscribing), которые являются общими для большинства дескрипторов ОП и будут рассмотрены в следующих главах.

BoldDerivedHandle

Этот компонент также предназначен для решения задач обработки информации, полученной от корневого дескриптора. Его отличие от дескриптора выражений BoldExpressionHandle в том, что обработка информации производится без использования OCL. Несмотря на мощность и гибкость языка OCL, на практике хоть и редко, но встречаются ситуации, когда возможностей OCL «не хватает». Тогда приходится использовать программную обработку на базовом языке (Object Pascal в Delphi). Для таких редких случаев и предназначен дескриптор обработки информации BoldDerivedHandle.

Для иллюстрации использования дескриптора поместим на нашу форму из предыдущего примера компонент BoldDerivedHandle1 и еще одну метку BoldLabel2.

Настроим дескриптор BoldDerivedHandle1 следующим образом (рис. 7.6):

1. Свойству RootHandle присвоим BoldListHandle1 (список авторов).
2. Для задания свойства ValueTypeName нажмем на кнопку с многоточием и в окне селектора типов последовательно выберем Lists • Class lists • Collection (Author). Этим мы укажем, что тип выходного результата нашего дескриптора будет представлять собой коллекцию авторов.

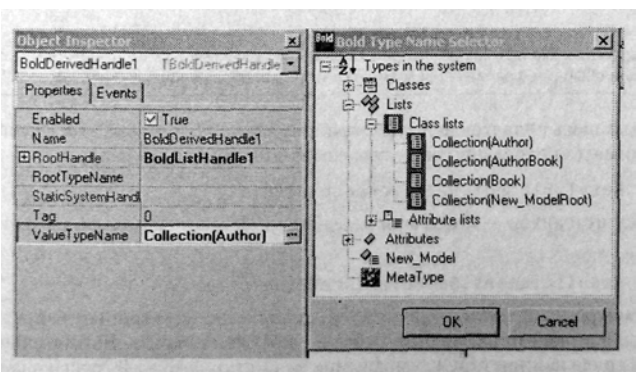


Рис. 7.6. Настройка свойств дескриптора BoldDerivedHandle

Для метки BoldLabel2 в качестве дескриптора **BoldHandle** укажем наш новый дескриптор BoldDerivedHandlel, а в окно свойства Expression введем OCL-выражение aname, то есть имя автора.

Поставим следующую, достаточно бессмысленную с точки зрения практики, задачу — пусть на метке BoldLabel2 отображается имя автора только в том случае, если выбран автор с фамилией Ильф. Для использования возможностей дескриптора BoldDerivedHandle необходимо написать обработчик события OnDeriveAndSubscribe. Щелкнем дважды в инспекторе объектов на раскрывающемся списке и введем код (листинг 7.1).

Листинг 7.1. Пример процедуры обработки события для BoldDerivedHandle

```
procedure TForm1.BoldDerivedHandle1DeriveAndSubscribe(Sender:
TComponent;
   RootValue: TBoldElement; ResultElement: TBoldIndirectElement;
   Subscriber: TBoldSubscriber);
var st:string;
     El: TBoldElement;
begin
     El:=BoldListHandle1.CurrentElement;
     st:=(El as
TBoldObject).BoldMemberByExpressionName['aname'].AsString;
     if st = 'Ильф' then     ResultElement.SetReferenceValue(El)
     else ResultElement.SetReferenceValue(nil);
end;
```

Выходная информация дескриптора обработки передается через параметр ResultElement. Тип этого параметра — TBoldIndirectElement. Этот тип специально введен для обеспечения хранения либо информации, либо ссылки на уже присутствующий элемент объектного пространства. Для нашего простого случая, как ясно видно из приведенного кода, достаточно проверить фамилию текущего автора, использовав уже знакомое (см. главу 6) выражение:

```
BoldMemberByExpressionName['aname'].AsString
```

для доступа к атрибуту объекта класса по имени.

ВНИМАНИЕ

Необходимо четко осознавать, что это выражение не является OCL-выражением, а представляет собой обычный фрагмент программного кода для обращения к свойству класса TBoldObject.

Новым здесь является использование переменной EI типа TBoldElement, которая необходима для передачи в качестве ссылки на выходе процедуры

```
ResultElement.SetReferenceValue(El)
```

если текущий автор — Ильф, или передачи пустого указателя в противном случае:

```
ResultElement.SetReferenceValue(nil)
```

Рассмотренный пример является несколько искусственным и предназначен только для общей иллюстрации работы с BoldDerivedHandle. Внимательный читатель без труда напишет OCL-выражение для метки BoldLabel2, решающее ту же задачу и без использования описываемого дескриптора.

BoldVariableHandle и BoldOCLVariables

Компонент-дескриптор переменной BoldVariableHandle является интересным с точки зрения того, что он предоставляет возможность своеобразного «шлюза», через который «обычные» переменные Delphi могут «проникать» в объектное пространство и использоваться там для различных операций наравне с «родными» элементами. Такая задача очень часто может возникнуть при практической разработке MDA-приложений. Напомним, что по умолчанию объектное пространство содержит только те элементы, которые представлены в UML-модели. Рассмотрим следующий пример. Предположим, мы хотим осуществлять фильтрацию авторов в зависимости от количества написанных книг. А именно, при вводе пользователем какого-то числа в поле формы в списке авторов должны остаться только те, количество написанных книг у которых совпадает с введенным числом. Новым в этой задаче является «смешение» сред и средств программирования. Поместим на нашу форму VCL-компонент TEditl для ввода количества книг, компонент BoldOCLVariablesl и дескриптор переменной BoldVariableHandlel, которому присвоим имя vhNumBooks (рис. 7.7). Компонент Edit1 принадлежит к «обычным» VCL-компонентам, в то время как сетка BoldGridl принадлежит к MDA-компонентам. Объектное пространство «ничего не знает» ни о компоненте Editl, ни о вводимых в его окошко значениях.

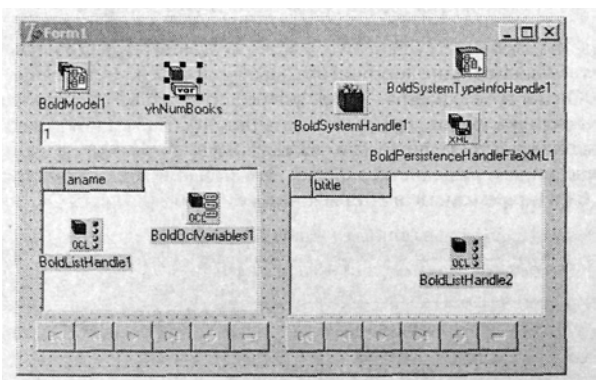


Рис. 7.7. Форма, иллюстрирующая работу с BoldVariableHandle

Для организации такой связи мы и используем дескриптор переменной. Настроим компонент vhNumBooks следующим образом (рис. 7.8).

1. Свойству StaticSystemHandle присвоим значение (из раскрывающегося списка) системного дескриптора BoldSystemHandle1. Свойство StaticSystemHandle указывает на источник информации о типах элементов ОП.
2. Для задания значения свойства ValueTypeName откроем уже известное нам окно селектора типов и выберем тип Integer. Данное свойство определяет тип ОП-переменной, который в данном случае должен быть целым (количество книг).

3. Зададим начальное значение нашей ОП-переменной, раскрыв свойство InitialValues и введя в открывшемся редакторе 1 (одну строку).

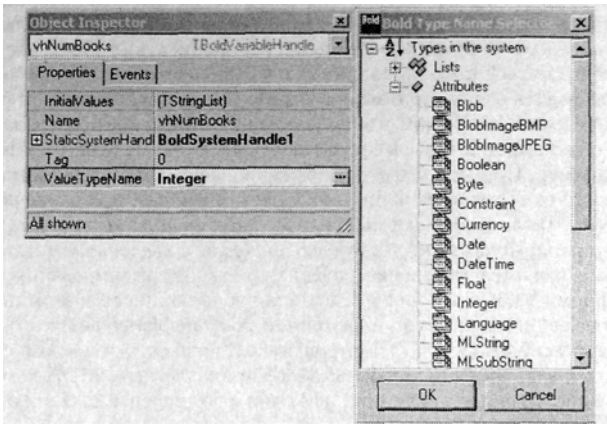


Рис. 7.8. Настройка компонента BoldVariableHandle

Перейдем к настройке компонента BoldOCLVariables. Раскрыв в инспекторе объектов свойство Variables этого компонента, мы увидим редактор набора переменных (рис. 7.9). Он построен аналогично другим редакторам Delphi. Добавим новый элемент в список переменных редактора и настроим его свойства. В качестве BoldHandle зададим наш дескриптор переменных vhNumBooks, а в качестве имени переменной VariableName введем название NumBooks. Это название будет являться псевдонимом нашей ОП-переменной, и именно оно может применяться в OCL-выражениях.

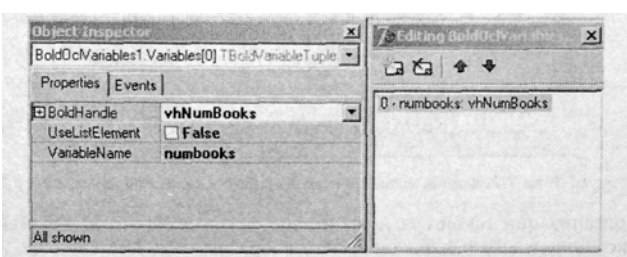


Рис. 7.9. Настройка компонента BoldOCLVariables

Для чего предназначен компонент BoldOCLVariables? Он представляет собой контейнер-интерфейс для нескольких ОП-переменных (переменных, доступных в объектном пространстве), причем все переменные, входящие в этот контейнер, становится возможным использовать не только в ОП, но и в выражениях OCL. Это мы сейчас и сделаем. Обратимся к дескриптору списка авторов BoldListHandle1. Во-пер-

вых, мы должны указать этому компоненту, что кроме объектов модели, он имеет доступ и к нашим ОП-переменным. Для этого в раскрывающемся списке свойства Variables этого компонента зададим значение BoldOCLVariables1. Во-вторых, дважды щелкнув по свойству Expression, откроем встроенный OCL-редактор и вместо имеющегося выражения введем следующее (рис. 7.10):

```
if numbooks=0
then Author.allInstances
else Author.allInstances->select(writes->size=numbooks)
endif
```

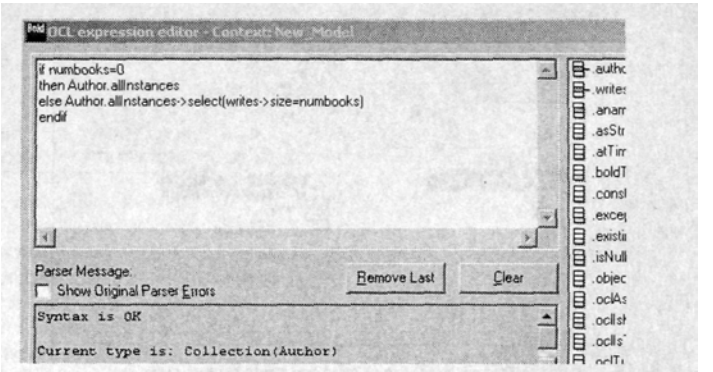


Рис. 7.10. Встроенный OCL-редактор

Это OCL-выражение содержит оператор условия. Если значение ОП-переменной numbooks равно 0, то будет использовано старое выражение (то есть будут выбираться все авторы), в противном случае будет произведена необходимая выборка по авторам (основы OCL изложены в главе 5). Для чего в OCL-выражении используется проверка на ноль? Это сделано для обеспечения возможности редактирования списка авторов. Как мы скоро увидим, отфильтрованный список авторов не допускает изменений.

Итак, мы настроили дескриптор ОП-переменной и встроили ее в контейнер OCL-переменных. Осталось написать программный код, передающий содержимое окна редактирования в ОП-переменную. Для этого используем событие OnChange компонента Edit1, и напишем процедуру, представленную в листинге 7.2.

Листинг 7.2. Пример передачи значения в ОП-переменную

```
procedure TForm1.Edit1Change(Sender: TObject);
var st:string;
begin
   st:=Edit1.Text;
   if st<>'' then vhNumBooks.Value.SetAsVariant(strtoint(st));
end;
```

Из приведенного фрагмента кода ясно, что значение введенного в редактор Edit1 числа присваивается свойству Value нашей ОП-переменной.

ВНИМАНИЕ

Стоит еще раз обратить внимание на то, что в тексте программы используется имя переменной vhNumBooks, в то время как в тексте OCL-выражения используется псевдоним этой переменной NumBooks, заданный в контейнере BoldOCLVariables1.

Запустим приложение на выполнение (рис. 7.11). Поскольку наша ОП-переменная по умолчанию имеет значение 1, то мы получим уже отфильтрованный список авторов, написавших по одной книге. При этом кнопки навигатора, отвечающие за редактирование списка авторов, стали недоступными.

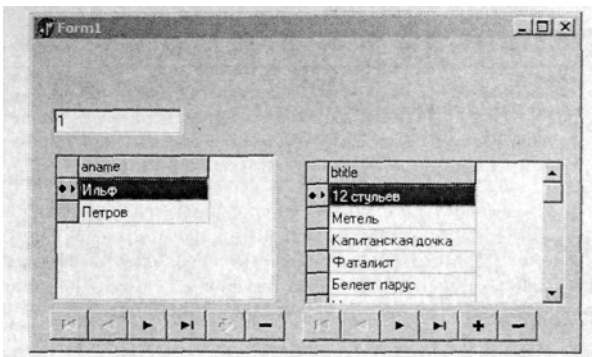


Рис. 7.11. Приложение во время выполнения

Для редактирования списка авторов введем в окошко редактора значение О, после чего можно добавлять новых авторов и новые книги, чтобы на практике убедиться в корректности функционирования созданного приложения.

BoldListHandle

Этот компонент очень часто используется — в основном, в качестве дескриптора для таких визуальных компонентов, как BoldGrid или BoldListBox, то есть там, где нужно получать список элементов. Примеры использования BoldListHandle не раз уже приводились на страницах этой книги, поэтому здесь мы остановимся только на особенностях его применения.

Во-первых, необходимо еще раз повторить, что этот дескриптор отличается от других рассматриваемых дескрипторов тем, что в качестве результата всегда возвращает единственное значение, а не коллекцию элементов. Это значение определяется текущим внутренним указателем дескриптора списка, положением которого можно программно управлять с помощью методов навигации, приведенных в табл. 7.1.

Легко видеть, что указанные методы схожи с соответствующими методами компонента DataSet.

Кроме того, дескриптор списка обладает рядом специфических свойств, использование которых облегчает работу с коллекциями элементов. Некоторые из этих свойств приведены в табл. 7.2.

Таблица 7.1. Методы навигации дескриптора списка

Метод	Описание
First	Перемещение указателя на первый элемент списка
Last	Перемещение указателя на последний элемент списка
Next	Перемещение указателя на следующий элемент списка
Prior	Перемещение указателя на предыдущий элемент списка

Таблица 7.2. Свойства дескриптора списка

Свойство	Тип	Описание
Count	Integer	Количество элементов списка
CurrentIndex	Integer	Текущее значение указателя
HasNext	Boolean	Показывает, имеется ли следующий элемент
HasPrior	Boolean	Показывает, имеется ли предыдущий элемент
CurrentElement	TBoldElement	Возвращает текущий элемент
CurrentBoldObject	TBoldObject	Возвращает текущий объект

Кроме того, BoldListHandle имеет интересное свойство MutableListExpression, на котором стоит остановиться подробнее. Как мы видели в предыдущем примере, в «фильтрованном» состоянии дескриптор списка не позволяет редактировать сам список, а только допускает навигацию по нему (см. рис. 7.11). Такое состояние списка возникает при наложении какого-нибудь ограничения (фильтра) на коллекцию элементов. Для того чтобы избежать такого ограничения, и используется указанное свойство MutableListExpression. Оно представляет собой OCL-выражение, не содержащее ограничений. В этом случае возможно редактирование состава списка, однако для просмотра таким образом модифицированного списка необходимо использовать другое специальное свойство — MutableList. Почему в состоянии фильтрации список не позволяет редактирования? Это можно пояснить на предыдущем примере. Допустим, мы добавляем нового автора в список. Но непосредственно при добавлении неизвестно, удовлетворяет ли он условиям фильтрации, то есть сколько он написал книг. Поэтому система «не знает», отображать его или нет в сетке. С точки зрения целостности ОП лучше в этом случае вообще запретить непосредственную модификацию, что мы и наблюдаем на практике.

BoldCursorHandle

Этот компонент предназначен в основном для преобразования единичных значений в коллекцию элементов для дальнейшего представления этой коллекции в визуальных компонентах типа BoldGrid или BoldListBox. Он может эффективно применяться совместно с дескрипторами ОП-переменных BoldVariableHandle. Для иллюстрации применения дескриптора курсора видоизменим приложение, которое мы рассматривали при описании дескрипторов ОП-переменных. Удалим из формы редактор Edit1 и добавим дескриптор курсора BoldCursorHandle1 и еще одну сетку BoldGrid3 (рис. 7.12).

Для этой сетки в качестве свойства BoldHandle укажем новый дескриптор BoldCursorHandle1 и зададим вид столбцов по умолчанию. Далее, в качестве свойства

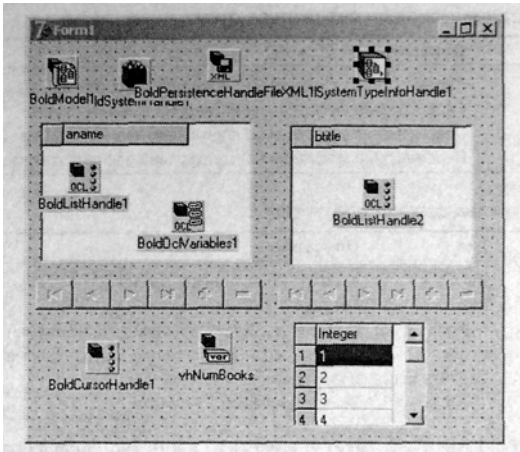


Рис. 7.12. Форма приложения с дескриптором курсора

BoldHandle для нового компонента BoldCursorHandle1 зададим созданный ранее дескриптор ОП-переменной vhNumBooks. Немного изменим свойства этой ОП-переменной. Во-первых, в качестве начальных значений (свойство InitialValues) зададим не одно значение, как в прошлом случае, а пять строк со значениями от 1 до 5 (рис. 7.13). Во-вторых, изменим тип ОП-переменной с Integer (целого) на Collection (Integer) (коллекцию целых) (см. рис. 7.14).

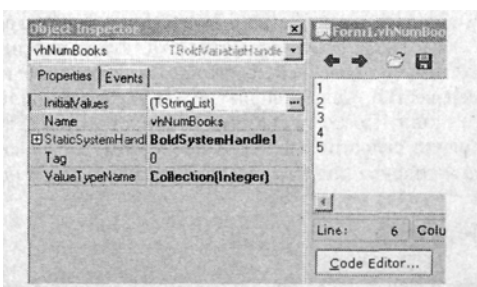


Рис. 7.13. Задание начальных значений для ОП-переменной

И, наконец, добавим вновь созданный дескриптор курсора в контейне OCL-переменных BoldOCLVariables (рис. 7.15), присвоив ему псевдоним curnum.

Теперь нам осталось изменить OCL-выражение для дескриптора списка авторов BoldListHandle1 на следующее:

```
if curnum=0
then Author.allInstances
else Author.allInstances->select(writes->size=curnum)
endif
```

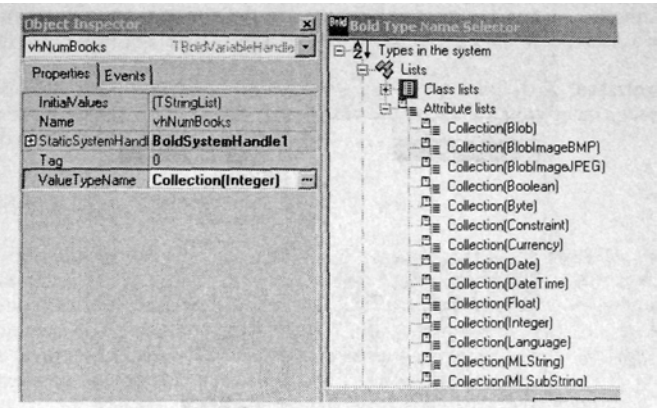


Рис. 7.14. Задание нового типа для ОП-переменной

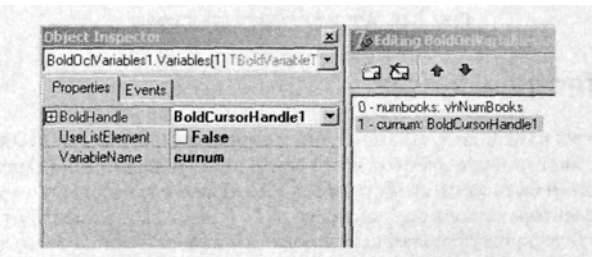


Рис. 7.15. Добавление дескриптора курсора в контейнер OCL-переменных

Запустив новое приложение на выполнение (рис. 7.16), увидим, для чего мы все это делали. При выборе на новой сетке конкретной цифры (от 1 до 5) наш список авторов автоматически фильтруется. При этом происходит следующее: информация из ОП-переменной, содержащая введенный нами набор начальных значений (1...5) благодаря подключению этой переменной к новому дескриптору курсора BoldCursorHandle1 стала «доступна» для непосредственного отображения на сетке. Дело в том, что сетка требует в качестве дескриптора только список и не может подключиться к ОП-переменной. Именно такое преобразование в список и обеспечивает дескриптор курсора. При выборе на сетке строки с конкретным значением оно автоматически присваивается нашей ОП-переменной, и происходит обработка OCL-выражения с этим новым значением. В результате мы получаем фильтр, управляемый из сетки.

Как и описанный в предыдущем разделе дескриптор списка BoldListHandle, дескриптор курсора BoldCursorHandle также всегда возвращает только единственное значение. Это обстоятельство подчеркивает общность этих типов дескрипторов. И, конечно, дескриптор курсора, так же как и BoldListHandle, обладает всеми специфичными свойствами и методами для работы с коллекциями (см. табл. 7.1 и 7.2).

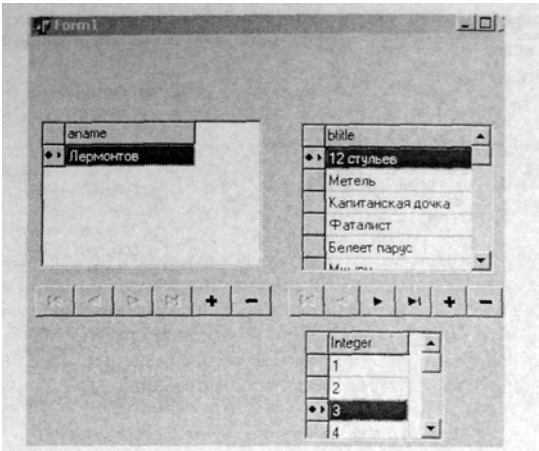


Рис. 7.16. Вид работающего приложения

BoldReferenceHandle

Как следует из названия, этот компонент-дескриптор предназначен для хранения ссылки на какой-нибудь элемент объектного пространства (BoldElement). Этот элемент должен быть задан свойством StaticValueTypeName посредством уже знакомого нам селектора типов (см., например, рис. 7.14). Как рекомендуют разработчики, данный дескриптор может использоваться в качестве стартового дескриптора формы приложения. Как и в предыдущем случае, дескриптор ссылки можно использовать в контейнерах OCL-переменных, присваивая ему OCL-псевдоним для дальнейшего использования этого псевдонима в OCL-выражениях. Мы не будем сейчас приводить подробные примеры использования этого дескриптора, поскольку общие принципы и подходы работы с дескрипторами должны быть понятны из предыдущих разделов.

BoldUnloaderHandle

И наконец, кратко рассмотрим последний из дескрипторов объектного пространства — дескриптор выгрузки объектов. Его назначение — обеспечить автоматическую выгрузку неиспользуемых объектов из ОП по истечении некоторого задаваемого интервала времени. Такая функциональность может потребоваться в ситуациях недостатка оперативной памяти. Критерием разрешения выгрузки объектов служит временной интервал, задаваемый (в миллисекундах) в свойстве дескриптора MinAgeForUnload. Этот временной интервал задает период, в течение которого к объекту не было обращения. Количество одновременно инспектируемых на предмет необходимости выгрузки объектов задается свойством ScanPerTick. Установка чрезмерно большого значения этого свойства не рекомендуется, так как

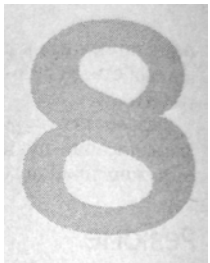
будет приводить к увеличению пауз при проведении очередной инспекции. Если это свойство установлено в 0, то одновременно будут инспектироваться все объекты ОП.

Разработчик может воспользоваться событиями OnMayUnload и OnMayInvalidate для написания процедур их обработки, в которых будет дополнительно проверяться, нужно ли выгружать конкретные объекты.

Резюме

Дескрипторы объектного пространства представляют собой специальные невизуальные компоненты, реализующие бизнес-уровень приложения. Они также обеспечивают взаимодействие с уровнем данных и графическим интерфейсом пользователя. Дескрипторы ОП играют центральную роль при разработке MDA-приложения, любое такое приложение обязательно содержит по крайней мере два дескриптора — системный дескриптор и дескриптор типов модели.

Использование OCL



Роль Object Constraint Language в Borland MDA

В предыдущей главе мы остановились на обсуждении дескрипторов объектного пространства. При практическом использовании дескрипторов ОП активно задействуется язык OCL.

В Borland MDA язык OCL играет чрезвычайно важную роль, выполняя следующие основные функции:

- навигация по элементам модели (классам, атрибутам, ассоциациям);
- формирование OCL-запросов к объектному пространству;
- создание вычисляемых атрибутов и ассоциаций.

Навигация по модели, обеспечиваемая посредством OCL в Borland MDA, позволяет осуществить гибкий и мощный механизм запросов к объектному пространству приложения. Такие «OCL-запросы», как мы увидим в этой главе, в принципе способны полностью заменить привычный разработчикам приложений баз данных язык SQL (см. также главу 5). Кроме того, учитывая платформенную независимость OCL, эти запросы являются универсальными и абсолютно не привязаны к конкретной СУБД, используемой в приложении.

Условная модель

Для иллюстрации использования OCL при работе с дескрипторами ОП усовершенствуем нашу модель приложения, добавив в нее дополнительные классы и ассоциации (рис. 8.1). В качестве основы используем условную модель, описанную в главе 5 при изучении языка OCL. Напомним, что наше приложение предназначено для работы с библиотечным каталогом, и в новую модель включены классы Страна, Издательство и Тематика.

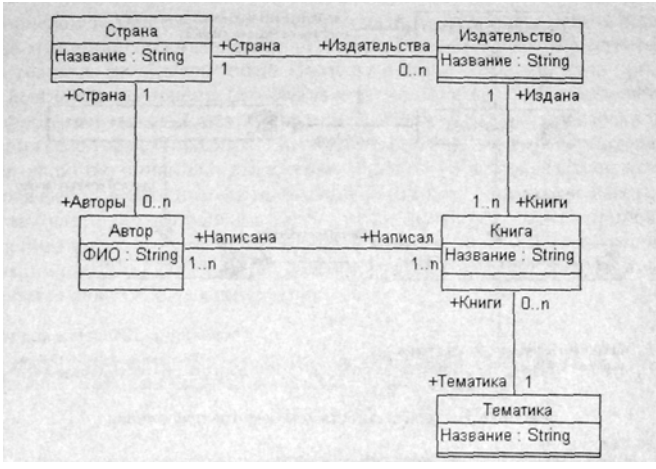


Рис. 8.1. Рассматриваемая модель для изучения OCL

Создадим модель в редакторе Rational Rose, преобразуем русскоязычные названия классов и атрибутов в англоязычные путем транслитерации и сохраним ее в файле lib.mdl.

ВНИМАНИЕ

Далее везде в книге, если это особо не оговорено, при использовании русскоязычных идентификаторов элементов UML-модели подразумевается их последующая транслитерация для применения в OCL-выражениях.

Создание приложения

Создадим новый проект в Delphi, состоящий из одной формы и модуля данных. С палитры компонентов поместим на модуль данных компоненты и настроим их свойства в соответствии с приведенной на рис. 8.2 диаграммой.

Теперь приступим к созданию графического интерфейса формы (рис. 8.3). Поместим на форму 7 визуальных компонентов BoldGridl, BoldGrid2,.. BoldGrid7 (на рисунке их номера представлены крупными цифрами), далее над каждым из них поместим по одной обычной метке (Label). Для удобства привязки присвоим меткам названия — Все авторы, Все книги и т. д. Под компонентами BoldGridl,3,5,6,7 поместим по одному компоненту BoldNavigator. Затем поместим три компонента BoidLabel (на рис. 8.3 они обведены контуром), пока не настраивая их свойства (на рисунке эти компоненты показаны в настроенном виде). Мы создали графический интерфейс, однако он еще не привязан к объектному пространству. Для осуществления такой привязки используем невизуальные компоненты — дескрипторы списков BoldListHandles с палитры компонентов BoldHandles, и проиллюстрируем на нашем примере, как используется язык OCL для взаимодействия графического интерфейса (уровня представления) с объектным пространством (бизнес-уровнем).

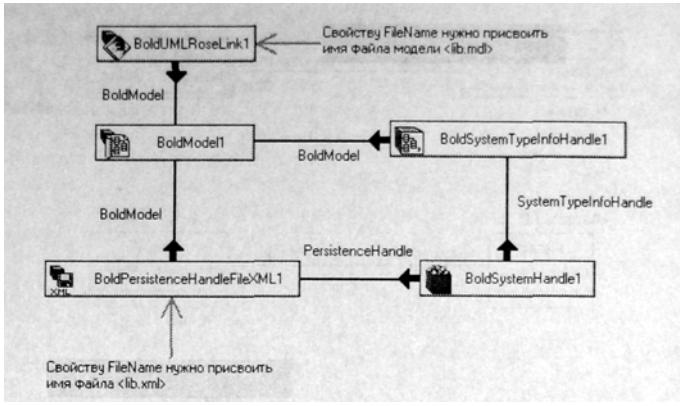


Рис. 8.2. Настройка свойств компонентов приложения

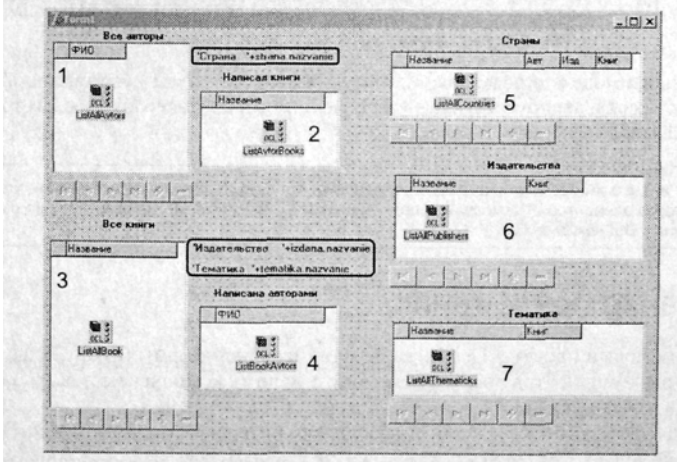


Рис. 8.3. Вид формы демонстрационного приложения

Встроенный OCL-редактор

Поместим дескриптор списка BoldListHandle1 на форму и присвоим ему название ListAllAvtors. Задачей данного дескриптора является получение из объектного пространства списка всех имеющихся авторов и передача этого списка визуальному компоненту BoldGrid1 для отображения. Дескриптор списка относится к производным дескрипторам (см. предыдущую главу), то есть он обладает свойством Root-

Handle (корневой дескриптор), указывающим на источник информации. В данном случае таким источником является компонент-системный дескриптор всего объектного пространства BoldSystemHandle1. Поэтому в инспекторе объектов присвоим свойству RootHandle компонента ListAllAvtors значение DataModule1.BoldSystemHandle1. Теперь сформулируем OCL-запрос, который обеспечит получение нужной нам информации об авторах. Обратимся к свойству Expression нашего дескриптора списка. Данное свойство описывает выражение на языке OCL, посредством которого реализуется выбор необходимой информации. Это выражение можно ввести вручную, как мы часто делали раньше, рассматривая пример простого приложения, однако удобнее воспользоваться встроенным в Borland MDA OCL-редактором. Для его активации щелкнем дважды в инспекторе объектов на свойстве Expression, при этом откроется окно OCL-редактора (рис. 8.4).

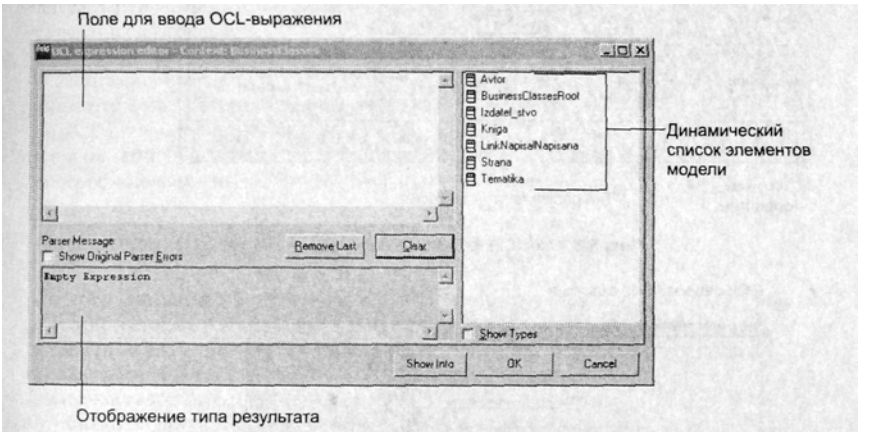


Рис. 8.4. Общий вид встроенного OCL-редактора

Интерфейс OCL-редактора устроен довольно просто, в верхней части он имеет окно для ввода выражений на языке OCL, внизу под ним располагается окошко для отображения информации о корректности выражения и типе возвращаемого результата, а справа имеется многофункциональное навигационное окно. В данном случае, пока мы не сформулировали OCL-выражение, навигационное окно отображает доступные элементы нашей модели (в среде разработки Delphi — значки с красной полоской). Поскольку нас сейчас интересуют авторы, щелкнем дважды в навигационном окне по элементу Avtor и обнаружим (рис. 8.5), что окно OCL-редактора изменилось.

Во-первых, в верхнем окне появилось выражение Avtor. Нижнее окошко сообщает нам, что синтаксис OCL-выражения корректен, а тип возвращаемого результата — метатип (то есть класс модели). И наконец, радикально изменилось содержимое навигационного окна, теперь в нем отображается список возможных OCL-операторов, применимых к нашему выражению (значки с желтой полоской). В самом верху этого списка находится нужный нам оператор .allInstances — «все

сущности» (в данном случае все объекты типа Автор, поскольку мы уже частично задали наше OCL-выражение). Щелкнем дважды по этому элементу и увидим картину, представленную на рис. 8.6.

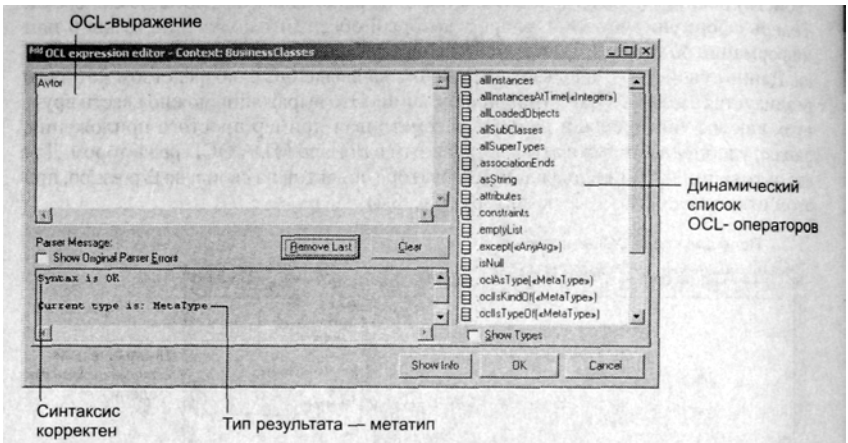


Рис. 8.5. Окно OCL-редактора при вводе имени класса

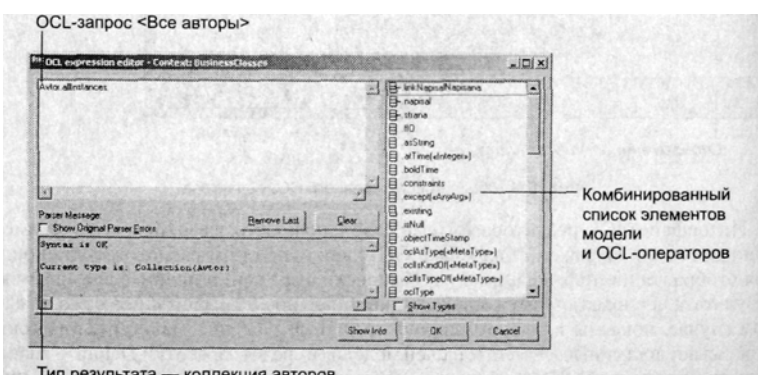


Рис. 8.6. Окно OCL-редакторэ при вводе OCL-выражения

Возвращаемый теперь тип результата — нужная нам коллекция объектов типа **Автор.**

Если посмотреть внимательно на навигационное окно справа, то можно увидеть характерное разнообразие отображаемых в нем элементов. Здесь присутствуют ассоциации и их роли (значки с зеленой полоской), причем не все, а только связанные с классом **Автор.** Также присутствуют атрибуты класса (значки с синей

полоской), в данном случае единственный атрибут ПО класса Автор. И наконец, по-прежнему присутствуют доступные OCL-операторы, которыми можно «продолжить» наше выражение для большей детализации. Мы чуть позже посмотрим, как использовать эти богатые возможности, а сейчас отметим, что, несмотря на кажущуюся простоту, OCL-редактор является весьма мощным средством формирования OCL-запросов. Обладая «интеллектом» и «знаниями» о модели, он динамически формирует возможные варианты последовательной детализации OCL-выражения, отображая их в навигационном окне. Выйдем из OCL-редактора и настроим компоненты BoldGridl и BoldNavigatorl, присвоив их свойству BoldHandle значение ListAllAvtors. Далее, щелкнем правой кнопкой мыши на компоненте BoldGridl, из контекстного меню выберем пункт Create Default Columns и убедимся, что появился заголовок столбца ФИО. Таким образом, мы подключили визуальный компонент BoldGridl к объектному пространству посредством дескриптора списка ListAllAvtors.

Продолжим создание нашего приложения. По аналогии с настройкой списка всех авторов создадим дескриптор списка ListAllBook для получения списка всех книг. Это читатель легко сделает самостоятельно, единственное отличие будет только в OCL-выражении, которое в данном случае будет выглядеть как Kniga.allInstances. Список книг будем отображать в BoldGrid3, для чего свяжем уже известным нам способом компоненты BoldGrid3 и BoldNavigator2 с дескриптором списка ListAllBook.

Формирование цепочек дескрипторов

До этого момента мы не сделали ничего принципиально нового по сравнению с рассмотренным в предыдущих частях примером простого приложения, хотя и начали применять возможности OCL-редактора. В этом разделе мы познакомимся с технологией использования цепочек дескрипторов списков и продемонстрируем гибкость и мощность OCL-запросов. Поставим следующую задачу — в BoldGrid2 отобразить книги, но не все книги, а только написанные конкретным автором

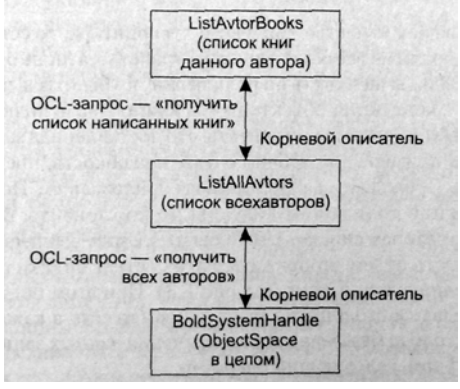


Рис. 8.7. Схема функционирования цепочки дескрипторов

который выбран в **BoldGridl.** Для решения этой задачи поместим новый дескриптор списка и назовем его ListAvtorBooks. Принципиальным моментом в данном случае является выбор корневого дескриптора — в качестве его логично выбрать не используемый выше системный дескриптор объектного пространства BoldSystemHandlel, а созданный нами дескриптор списка авторов ListAllAvtors. Тем самым мы формируем цепочку дескрипторов (рис. 8.7), в которой каждый последующий элемент ссылается на предыдущий своим свойством Root Handle.

Присвоим в инспекторе объектов свойству RootHandle дескриптора ListAvtorBooks значение ListAllAvtors и запустим OCL-редактор (рис. 8.8). Хотя мы не ввели пока никакого OCL-выражения, в навигационном окне, как легко видеть, отображаются теперь не все элементы нашей модели, а лишь те, которые относятся к классу **Автор.**

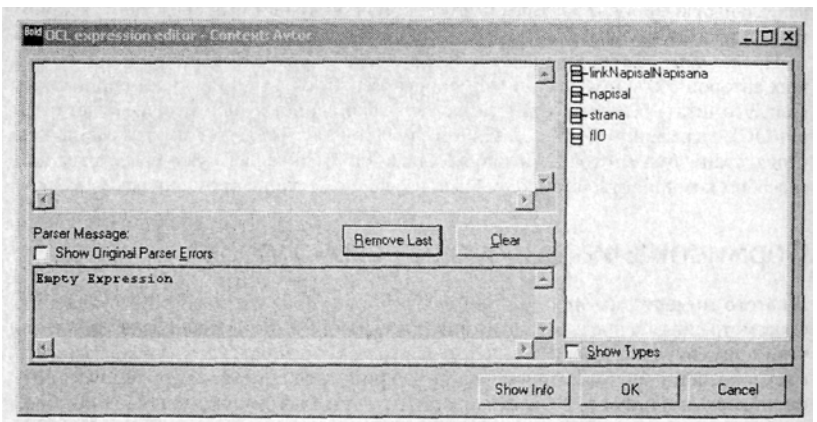


Рис. 8.8. Вид окна OCL-редактора для второго дескриптора цепочки

Это понятно, ведь в качестве корневого дескриптора, то есть поставщика информации, мы выбрали не все объектное пространство, а лишь один его класс. Нам осталось теперь дважды щелкнуть по роли napisal, и убедиться, что тип возвращаемого результата — коллекция объектов типа Книга. После подключения BoldGrid2 к дескриптору ListAvtorBooks можно считать, что поставленная задача решена. Даже на таком простом примере достаточно очевидна гибкость, предоставляемая средой Borland MDA. Продолжим работу с нашим приложением. По аналогии со списками авторов и книг подключим визуальные компоненты BoldGrid5, BoldGrid6 и BoldGrid7 к дескрипторам списков ListAllCountries, ListAllPublishers, ListAllThematics соответственно, то есть будем отображать в этих сетках списки стран, издательств и тематических направлений книг (см. рис. 8.3). При этом пока в каждой из указанных сеток создадим столбцы по умолчанию (то есть в каждой будет присутствовать только столбец Название). Для сохранения данных запишем в обработчик события OnClose формы следующий оператор:

```
DataModule1.BoldSystemHandle1.UpdateDatabase;
```

Запустим наше приложение, внесем данные на авторов, книги, страны, издательства и тематики и убедимся, что они сохраняются при выходе из приложения. Теперь можно более подробно познакомиться с организацией взаимодействия бизнес-уровня и графического интерфейса.

Связь OCL и графического интерфейса

Поставим следующую задачу — отображать на метке BoldLabel1 название страны, в которой проживает текущий автор, выбранный в сетке авторов (BoldGridl). Метка BoldLabell, как и любой другой визуальный компонент Borland MDA, имеет свойство BoldHandle — то есть Bold-дескриптор. Мы уже знаем, что в качестве источника данных для этого дескриптора целесообразно выбрать дескриптор списка авторов ListAllAvtors, что мы и сделаем. А в качестве OCL-выражения напишем вручную или введем во встроенном OCL-редакторе (напомним, что OCL-редактор вызывается при двойном щелчке на свойстве Expression компонента ListAIIAvtors типа BoldListHandle) следующее выражение 'Страна '+strana.nazvanie. Здесь 'Страна ' — это текстовая константа, а strana.nazvanie — это и есть OCL-запрос для выбора страны текущего автора и выбора атрибута nazvanie класса Страна для отображения его на нашей форме. Встроенный в Borland MDA механизм обеспечит при этом автоматическое изменение текста метки при изменении текущего автора в сетке BoldGridl. На этом моменте стоит остановиться подробнее — как уже упоминалось ранее, одним из важных механизмов, встроенных в Borland MDA, является механизм так называемой «подписки на события» (subscribing). Как работает данный механизм?

В нашем случае, связав свойство BoldHandle метки с дескриптором списка авторов ListAIIAvtors, мы «сообщили» среде Borland MDA, что необходимо «подписать» нашу метку на события, возникающие при любых изменениях списка авторов (в том числе и на события, возникающие при переходах от одного объекта к другому при перемещении по списку авторов в сетке). Кроме того, мы сформировали OCL-запрос, который должен быть выполнен при наступлении такого события (изменении списка авторов), с указанием конкретной необходимой нам информации (названия страны, в которой проживает автор). Остальное среда Borland MDA сделает самостоятельно — то есть по наступлении события проверит, есть ли элементы, подписавшиеся на изменение этого списка, и если таковые имеются, выполнит соответствующий OCL-запрос и вернет полученные данные подписанному элементу для дальнейшего использования (в данном случае — для отображения на форме). Отметим, что в данном случае «подписку» незаметно для нас «оформила» сама среда, но это совсем необязательно. Разработчик может и самостоятельно «подписать» любой Bold-элемент на необходимое событие, воспользовавшись специальными методами класса TBoldElement.

Читатель вправе спросить, ну и что здесь нового — в традиционной разработке приложений баз данных тоже имеется, например, визуальный компонент TDBLabel — метка, которая автоматически отслеживает и отображает на форме содержимое указанного поля таблицы БД при перемещении по сетке. Однако новое качество есть — обратите внимание, что в нашей «таблице»-классе Автор отсутствует «поле»-атрибут Страна. Для того, чтобы решить эту задачу традиционными методами, нужно было бы сформировать SQL-запрос, который бы выбирал страну из другой таб-

лицы для конкретного автора, и связать метку с этим запросом. Кроме того, необходимо было бы специально «подключать» запрос к сетке либо к набору данных, чтобы он выполнялся при переходах между авторами. Мы, как легко видеть, обошлись без подобных операций.

Аналогично проделаем операции по подключению Bold-меток BoldLabel2 (для отображения названия издательства) и BoldLabel3 (для отображения тематики книги). В качестве источника информации для этих меток выступает дескриптор списка книг ListAllBooks. После запуска приложения убеждаемся, что тексты меток отслеживают перемещение по списку книг, отображаемых в сетке BoldGrid3.

OCL и вычисляемые данные

Использование OCL в Borland MDA позволяет обеспечить гибкие механизмы получения разнообразных данных. На примере нашего приложения рассмотрим некоторые из таких возможностей. Обратимся для этого к сетке BoldGrid7, пока в ней присутствует всего один пользовательский столбец, отображающий тематику книг. Поставим дополнительную задачу — показывать в этой же сетке суммарное количество книг по данной тематике. Для этого необходимо добавить еще один столбец

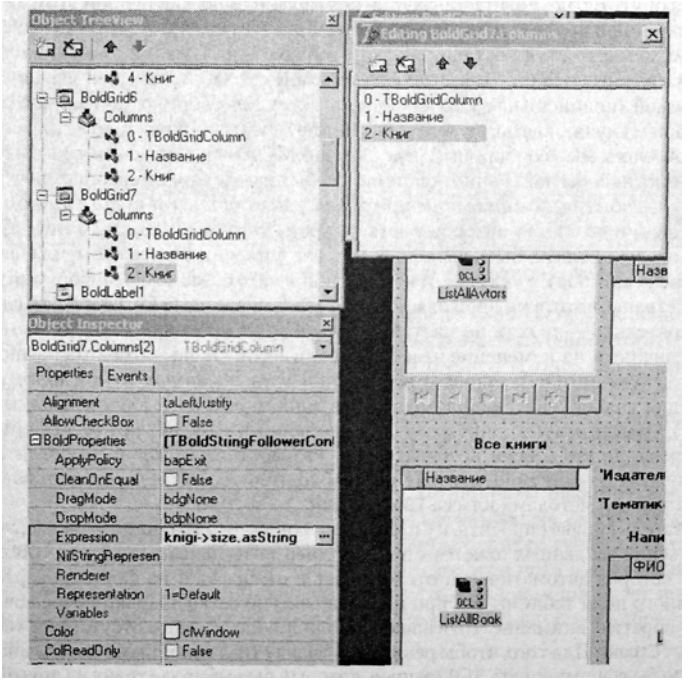


Рис. 8.9. Настройка компонента BoldGrid

в сетку — дважды щелкнем по ней и появится редактор столбцов (рис. 8.9, вверху справа). Добавление столбца осуществляется также как при использовании традиционного компонента Grid. Присвоим новому столбцу название Книг (оно будет отображаться в заголовке столбца). Однако на этом аналогия с традиционным компонентом заканчивается — как мы видим (см. рис. 8.9, слева), в инспекторе объектов снова присутствует уже знакомое нам свойство Expression. Введем в него вручную или воспользовавшись OCL-редактором выражение knigi->size.asString. В данном случае оно означает, что необходимо взять значение роли Книги ассоциации класса Тематика (см. модель на рис. 8.1), получить количество элементов этой роли — оператор ->size и, наконец, преобразовать полученное выражение в строку — .asString. Тем самым мы решили поставленную задачу.

Аналогично, в сетке BoldGrid6 добавим столбец для отображения количества книг, изданного конкретным издательством. В этом случае OCL-выражение, как легко убедиться, будет тем же самым, поскольку название роли такое же, как и в предыдущем случае.

И, наконец, обратимся к сетке, отображающей названия стран (BoldGrid5). До этого момента в ней отображался всего один столбец, созданный по умолчанию, содержащий название страны. Предположим, что мы хотим получить дополнительную статистическую информацию по каждой стране, а именно — общее количество авторов, проживающих в ней, общее количество издательств, а также общее количество всех изданных в ней книг. Для этого добавим в сетку BoldGrid5 три новых столбца с названиями Авт, Изд, Книг. Для столбца Авт выражение OCL примет

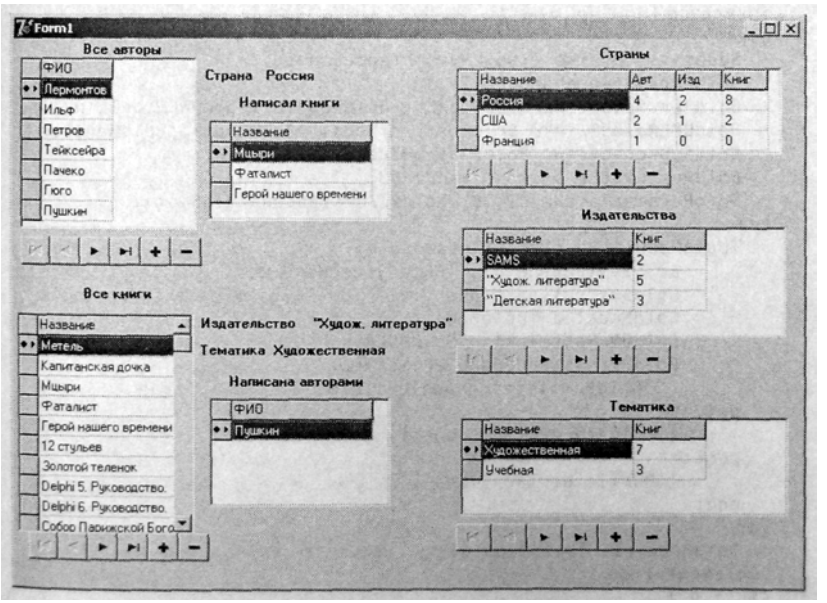


Рис. 8.10. Вид работающего приложения

следующий вид: avtory->size.asString, для столбца Изд — izdatel_stva->size.asString Однако для последнего столбца подобное простое выражение не будет являться решением, поскольку алгоритм расчета суммарного количества книг является более сложным. В самом деле, в нашей модели отсутствует ассоциация, непосредственно связывающая классы Страна и Книга, и это не является недостатком модели, так как вся необходимая информация для решения поставленной задачи у нас имеется (см. главу 3, где рассматривался аналогичный пример).

Алгоритм решения на естественном языке можно сформулировать для данного случая примерно так — для каждой страны необходимо выбрать все ее издательства, подсчитать количество книг, изданных каждым, и просуммировать все полученные величины. На языке OCL данный алгоритм можно представить так:

```
izdatel_stva->collect(knigi->size)->sum
```

Здесь мы снова встречаемся с оператором взятия коллекции ->collect и оператором суммирования ->sum. Их смысл был раскрыт в главе 5 и вполне ясен из рассмотренного примера. После запуска приложения можно убедиться, что оно приобрело новые, необходимые нам качества (см. рис. 8.10).

Полные исходные тексты созданного приложения приведены в листингах 8.1-8.3.

Листинг 8.1. Модуль данных приложения

```
unit UDataModule;
interface
uses
  SysUtils, Classes. BoldPersistenceHandle,
  BoldPersistenceHandleFile,
  BoldPersistenceHandleFileXML, BoldHandle, BoldUMLModelLink,
  BoldUMLRose98Link, BoldAbstractModel, BoldModel, BoldHandles,
  BoldSubscription, BoldSystemHandle,
  BoldAbstractPersistenceHandleDB,
  BoldPersistenceHandleDB, BoldPersistenceHandleSystem;
type
  TDataModule1 = class(TDataModule)
      BoldSystemHandle1: TBoldSystemHandle;
      BoldSystemTypeInfoHandle1: TBoldSystemTypeInfoHandle;
      BoldModel1: TBoldModel;
      BoldUMLRoseLink1: TBoldUMLRoseLink;
      BoldPersistenceHandleFileXML1:
        TBoldPersistenceHandleFileXML;
  private
      { Private declarations }
  public
      { Public declarations }
  end;
var
  DataModule1: TDataModule1;
implementation
{$R *.dfm}
end.
```

Листинг 8.2. Главный модуль приложения

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls,
  Forms, Dialogs, BoldSubscription, BoldHandles,
  BoldRootedHandles,
  BoldAbstractListHandle, BoldCursorHandle, BoldListHandle,
  Grids,
  BoldGrid, StdCtrls, ExtCtrls, BoldNavigatorDefs,
  BoldNavigator,
  BoldLabel, BoldAFPDefault;
type
  TForm1 = class(TForm)
      BoldGrid1: TBoldGrid;
      BoldGrid2: TBoldGrid;
      BoldGrid3: TBoldGrid;
      BoldGrid4: TBoldGrid;
      ListAllAvtors: TBoldListHandle;
      ListAllBook: TBoldListHandle;
      ListAvtorBooks: TBoldListHandle;
      Label1: TLabel;
      Label2: TLabel;
      Label3: TLabel;
      Label4: TLabel;
      BoldNavigator1: TBoldNavigator;
      BoldNavigator2: TBoldNavigator;
      BoldLabel1; TBoldLabel;
      BoldLabel2: TBoldLabel;
      ListBookAvtors: TBoldListHandle;
      BoldGrid5: TBoldGrid;
      BoldGrid6: TBoldGrid;
      BoldGrid7: TBoldGrid;
      Label5: TLabel;
      Label6: TLabel;
      Label7: TLabel;
      BoldLabel3: TBoldLabel;
      BoldNavigator3: TBoldNavigator;
      BoldNavigator4: TBoldNavigator;
      BoldNavigator5: TBoldNavigator;
      ListAllCountries: TBoldListHandle;
      ListAllPublishers: TBoldListHandle;
      ListAllThematicks: TBoldListHandle;
      procedure FormClose(Sender: TObject; var Action:
        TCloseAction);
  private
      { Private declarations }
  public
      { Public declarations }
  end;
```

```
var
  Form1: TForm1;
implementation
uses UDataModule;
{$R *.dfm>
procedure TForm1.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
  DataModule1.BoldSystemHandle1.UpdateDatabase;
end;
end.
```

Листинг 8.3. Текст главной программы

```
program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  UDataModule in 'UDataModule.pas' {DataModule1: TDataModule};
{$R *.res}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TDataModule1, DataModule1);
  Application.Run;
end.
```

Легко заметить, что исходные тексты в основном состоят из объявлений переменных и вызовов внешних модулей. Учитывая полученную функциональность приложения, нельзя не отметить такую исключительную «лаконичность» программных текстов. При создании приложения мы практически не писали кода на языке Object Pascal и не использовали язык SQL, но, тем не менее, смогли «связать» все элементы нашей модели в одно целое. Вся основная функциональность заключена не в приведенных выше текстах, а в вызываемых внешних модулях среды Bold for Delphi, которые и обеспечивают в результате компиляции получение готового MDA-приложения. Напомним здесь, что созданное приложение обладает всеми возможностями по редактированию данных (рис. 8.11), для доступа к которым достаточно дважды щелкнуть по любому элементу (имя автора, название книги и т. д.), после чего автоматически откроются созданные средой Bold окна редактирования (автоформы, подробнее см. главу 9).

На примере созданного приложения были продемонстрированы основные подходы к реализации взаимодействия бизнес-уровня и графического интерфейса пользователя. Мы убедились на практике, что при использовании Borland MDA эта среда обеспечивает принципиально новые возможности по формированию такого взаимодействия благодаря встроенной поддержке языка OCL. Так, любой визуальный элемент приобретает способность отображать любые данные, содержащиеся в объектах объектного пространства. В рассмотренном примере мы «заставили» метки и сетки отображать информацию из объектов, принадлежащих разным классам нашей модели, и даже получать новую информацию, которая в модели непосредственно не содержится (статистические данные по странам, издательствам и т. д.). Необходимо отметить, что формирование такого рода взаимодействий не

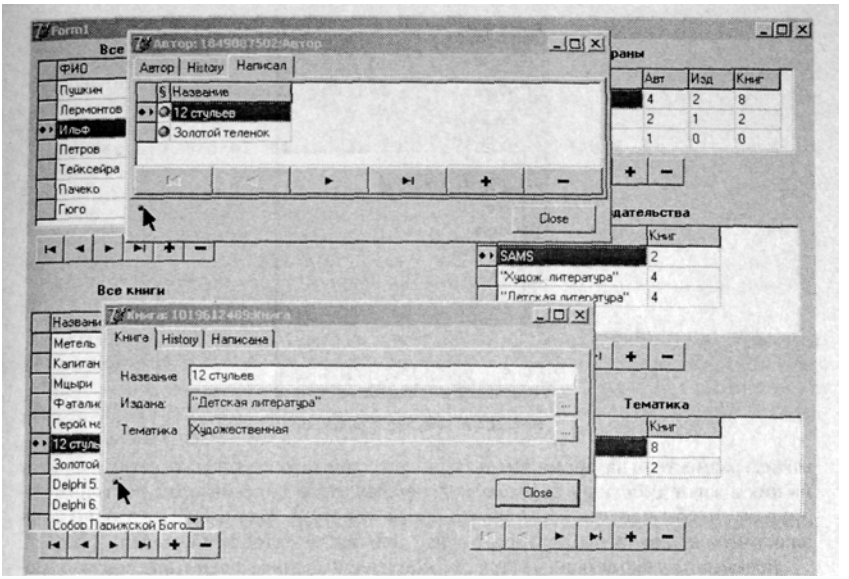


Рис. 8.11. Вид приложения с окнами редактирования

ограничивается этапом разработки приложения. Любое OCL-выражение можно формировать и присваивать и во время выполнения программы. И такой OCL-запрос будет исполнен средой Borland MDA, поскольку OCL-интерпретатор включается в состав исполняемого exe-файла приложения, так же как и вся информация о модели.

Использование OCL позволяет сделать приложение более «платформенно-независимым», так как язык OCL сам является платформенно-независимым. На практике это означает, что бизнес-логика приложения, будучи один раз создана на языке OCL без кодирования на языках программирования, при переносе разработки на другую платформу полностью сохранится и может быть повторно использована. И это является еще одним преимуществом технологии Borland MDA.

Использование OCL в программе

Формирование OCL-выражений возможно не только в свойствах компонентов Bold for Delphi. Бывают ситуации, когда это необходимо и может быть достаточно просто реализовано непосредственно в коде приложения. Рассмотрим простой пример.

Пусть нам необходимо задавать конкретный тип информации в сетке формы в зависимости от положения переключателя (рис. 8.12).

То есть если выбрано Авторы, в сетке должен отображаться список авторов, если выбрано Книги — список книг и т. д. Как это сделать наиболее просто? Мы, конечно,

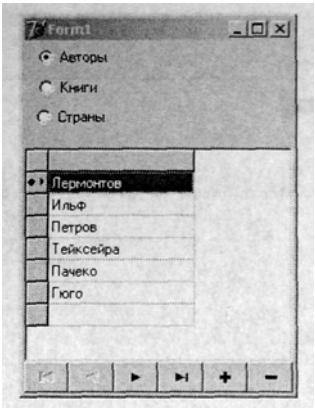


Рис. 8.12. Простое приложение

можем разместить на форме три дескриптора списка и подключать сетку к одному из них в зависимости от положения переключателя. Но сейчас мы поступим по-другому, чтобы продемонстрировать взаимодействие кода и OCL, а заодно и познакомиться с некоторыми нюансами настройки элементов UML-модели в BMDA.

Возьмем в качестве основы приложение, рассмотренное ранее (датамодуль и модель), и создадим новую простую форму (рис. 8.13), содержащую BoldGrid, BoldNavigator, дескриптор списка List и три переключателя, верхний из которых (Авторы) сделаем выбранным (checked). Чтобы по умолчанию отображался список авторов в качестве OCL-выражения для дескриптора списка List введем:

```
Avtor.allInstances
```

Напишем обработчики событий для переключателей (листинг 8.4).

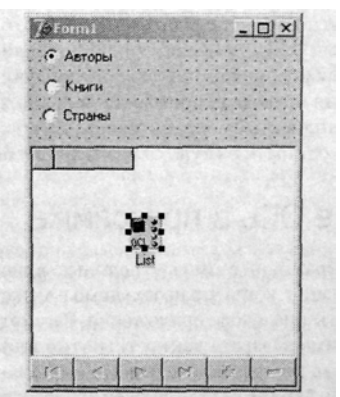


Рис. 8.13. Форма приложения

Листинг 8.4. Формирование OCL-выражений в коде программы

```
procedure TForm1.RadioButton1Click(Sender: TObject);
begin
   List.Expression:='Avtor.allInstances';
end;
procedure TForm1.RadioButton2Click(Sender: TObject);
begin
   List.Expression:='Kniga.allInstances';
end;
procedure TForm1.RadioButton3Click(Sender: TObject);
begin
   List.Expression:='Strana.allInstances';
end;
```

Из приведенного кода понятно, что свойство Expression является обычным текстом, который можно формировать программно. Если мы сейчас запустим наше приложение, то оно послушно будет менять классы при выборе того или иного переключателя, однако при этом отображение информации в сетке будет несколько странным (рис. 8.14).

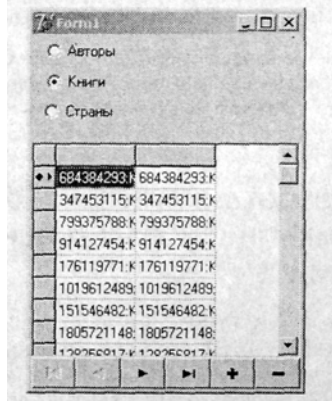


Рис. 8.14. Вид сетки без настройки представления класса

Объясняется это просто. В данном случае мы не настраивали сетку для отображения «по умолчанию», как в предыдущих примерах. Поэтому она «не знает», что именно требуется отображать, и показывает внутреннюю информацию системы об идентификаторах объектов. Для того чтобы любой визуальный элемент без дополнительной настройки «знал», что именно отображать для конкретного класса, существует специальный параметр для каждого класса модели, называемый «Default String Representation» — то есть текстовое представление по умолчанию. Для настройки этого параметра откроем встроенный редактор модели, выберем класс Автор в дереве модели (рис. 8.15) и введем в поле Default string rep OCL-выражение fIO (атрибут фамилия автора). Для классов Книга и Страна аналогичным образом введем выражение nazvanie. Можно даже не вводить эти выражения

вручную, а воспользоваться встроенным OCL-редактором, который в данном случае вызывается нажатием на расположенную рядом кнопку со знаком вопроса. Для классов, содержащих множество атрибутов, применение указанного механизма позволяет задать наиболее характерный атрибут по умолчанию, который и будет отображаться всеми визуальными компонентами BMDA.

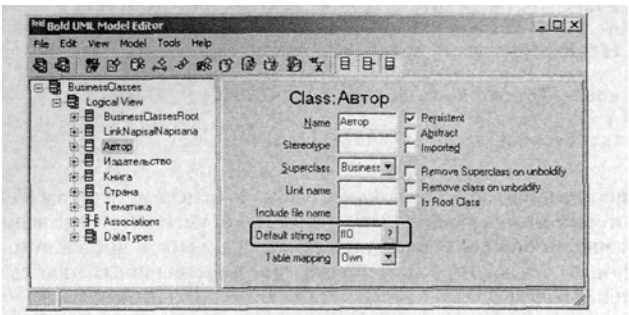


Рис. 8.15. Настройка параметра представления по умолчанию

После описанных настроек наше приложение будет функционировать штатно (см. рис. 8.11). Таким образом, на очень простом примере мы убедились, что формировать OCL-выражения можно при необходимости непосредственно в коде приложения.

Программная работа с классом TBoldElement и его наследниками с использованием OCL-вычислений

В главе 6 рассматривался класс TBoldElement, являющийся родительским классом для всех элементов модели. В составе его многочисленных методов присутствуют и те, что эффективно используют язык OCL. Здесь мы на простых примерах покажем, как практически использовать некоторые возможности этого класса. Возвращаясь к созданному ранее приложению (см. листинги 8.1-8.3), поставим следующую задачу — программно, без использования дескрипторов, рассчитать общее количество авторов в каталоге. Мы уже знаем, что для получения такой информации необходимо вычислить OCL-выражение вида:

```
avtor->allInstances->size
```

При этом встраиваемый в исполняемый файл приложения интерпретатор OCL-выражений обеспечит необходимые вычисления без использования программного кодирования.

Покажем, как программно воспользоваться предоставляемыми OCL-интерпретатором возможностями. Для этого используем метод EvaluateExpressionAsString класса TboldObjectList, который является наследником класса TBoldElement (см. главу 6). Напишем следующее OCL-выражение:

```
i:=strtoint(datamodule1.BoldSystemHandle1.System
.ClassByExpressionName['avtor']
.EvaluateExpressionAsString('self->size',1));
```

где *i* — целая переменная, в которую передается искомое общее количество авторов. Несмотря на несколько громоздкий вид представленного выражения, смысл его довольно прост и частично рассматривался в главе 6. «Новое» в данном выражении — это вызов метода EvaluateExpressionAsString с текстовым параметром — OCL-выражением вида self->size. В данном случае self обозначает текущий контекст (см. главу 5), определяемый выражением ...ClassByExpressionName['avtor'], то есть коллекцию объектов типа Автор. Особенностью подобного применения метода EvaluateExpressionAsString является то, что он не привязан ни к каким дескрипторам или визуальным Bold-компонентам, а зависит только от знания модели средой Bold и разработчиком программного кода. Естественно, что в качестве OCL-выражения можно использовать гораздо более сложные операторы OCL, получая при этом возможность расчета сложных выражений без программирования на языке Object Pascal. Подобные возможности могут с успехом применяться, например, при проведении массовых расчетов сводных аналитических данных для вывода отчетной информации. Другое возможное применение — расчет «псевдовычисляемых» атрибутов с последующим сохранением их в базе данных (напомним, что вычисляемые атрибуты в базе данных не могут быть сохранены обычным образом).

OCL-репозиторий

Формирование OCL в тексте программы, подобно описанному в предыдущем разделе, предоставляет разработчику достаточно гибкие возможности, например при динамическом формировании OCL-выражений или даже для их генерации «на лету». Однако такой практикой не стоит злоупотреблять, так как в результате нарушается целостность приложения, и часть бизнес-логики, содержащаяся в OCL, «размазывается» по коду. Это может приводить к плохой читаемости программы и к сложностям переноса на другую языковую платформу. Видимо, имея ввиду подобные ограничения, разработчики BMDA включили в состав своих компонентов так называемый OCL-репозиторий, который, по сути, является хранилищем для разнообразных OCL-выражений. Продемонстрируем работу с этим компонентом на примере из предыдущего раздела. Поместим на форму компонент OCLRepositoryl с вкладки BoldMisc и присвоим ему имя OCL (рис. 8.16).

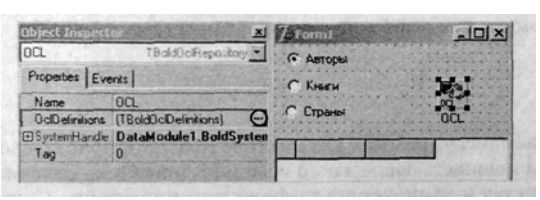


Рис. 8.16. Форма с OCL-репозиторием и свойства компонента

В инспекторе объектов установим для него в качестве свойства SystemHandle системный дескриптор DataModule1.BoldSystemHandle1, после чего раскроем свойства OclDefinitions, нажав кнопку с многоточием (см. рис. 8.16, 8.17).

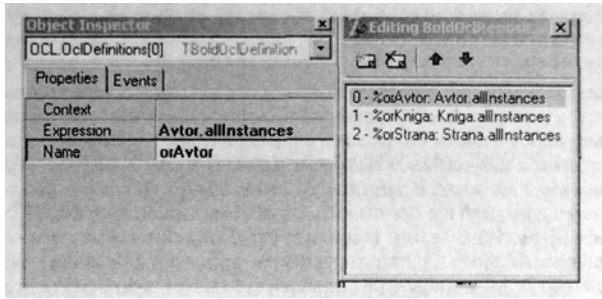


Рис. 8.17. Редактирование состава OCL-элементов

В открывшемся окне редактирования состава репозитория добавим три элемента, каждому в инспекторе объектов присвоим псевдонимы (свойство Name) orAvtor, orKniga, orStrana и введем те OCL-выражения, которые в предыдущем примере присваивали в коде (см. рис. 8.16). Перепишем обработчики переключателя (листинг 8.5).

Листинг 8.5. Задание OCL-выражений из репозитория

```
procedure TForm1.RadioButton1Click(Sender: TObject);
begin
   List.Expression:=OCL.LookUpOclDefinition('orAvtor');
end;
procedure TForm1.RadioButton2Click(Sender: TObject);
begin
   List.Expression:=OCL.LookUpOclDefinition('orKniga');
end;
procedure TForm1.RadioButton3Click(Sender: TObject);
begin
   List.Expression:=OCL.LookUpOclDefinition('orStrana');
end;
```

В приведенном примере кода доступ к конкретному OCL-выражению, содержащемуся в репозитарии, осуществляется по его псевдониму с использованием свойства LookUpOclDefinition. Также можно применять свойство OclDefinitions. Items[index] (где index — номер выражения) для доступа к нужному OCL-выражению по его номеру в репозитории.

Резюме

В данной главе описана работа с OCL в среде Bold for Delphi. Продемонстрированы гибкость и мощность совместного использования OCL, дескрипторов объектного пространства и компонентов графического интерфейса. С другой стороны, показано на конкретных примерах, как можно использовать возможности OCL непосредственно из программного кода, достигая при этом нового качества, а именно — привлечения функциональных вычислительных возможностей встроенного OCL-интерпретатора без программирования сложных выражений на уровне кода.

Графический интерфейс

Программная среда разработки Bold for Delphi включает собственные визуальные компоненты, предназначенные для формирования фафического пользовательского интерфейса, а также специальные средства управления отображением визуальной информации — *рендереры*. Общая структура взаимодействия бизнес-уровня приложения с графическим интерфейсом может быть описана следующим образом. Информация из бизнес уровня, доступ к которой обеспечивается посредством дескрипторов объектного пространства (см. главу 7), передается на уровень представления (графический интерфейс) посредством рендереров (рис. 9.1). Рендереры управляют процессом отображения информации, формируя конкретное визуальное представление данных на основе задаваемых разработчиком правил.

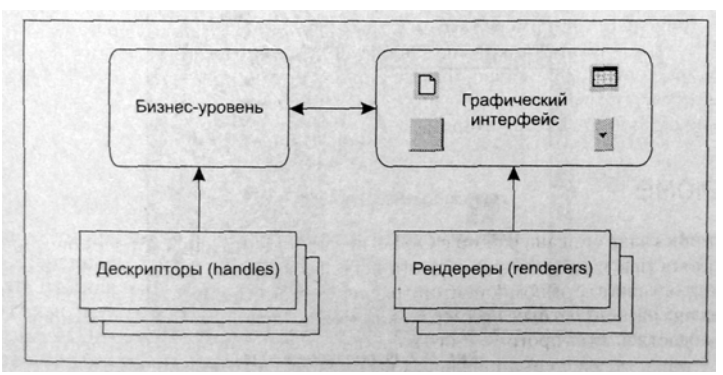


Рис. 9.1. Общая схема взаимодействия бизнес-уровня и графического интерфейса

В данной главе дается описание визуальных компонентов Bold for Delphi и рассматриваются вопросы управления отображением визуальной информации посредством рендереров.

Особенности визуальных MDA-компонентов

В рассматриваемой версии Borland MDA для Delphi 7 визуальные компоненты, предназначенные для использования в MDA-приложениях, имеют ряд специфических особенностей. Это объясняется необходимостью подключения этих компонентов к объектному пространству приложения, которое, как мы уже знаем, имеет собственные механизмы управления поведением объектов и обработки различных событий, возникающих в системе.

Наиболее общим свойством каждого визуального MDA-компонента является свойство BoldHandle, представляющее собой ОП-дескриптор (дескриптор объектного пространства). Назначением такого дескриптора является поставка информации для отображения в визуальном компоненте. Для указания конкретного состава информации, получаемой от ОП-дескриптора, каждый визуальный компонент имеет также текстовое свойство Expression, предназначенное для задания OCL-выражения. Далее в этом разделе мы рассмотрим основные графические MDA-компоненты и особенности их практического применения.

BoldLabel

Этот компонент является самым простым из графических MDA-компонентов. Его задачей является отображение информации из свойства Caption, получающейся в результате вычисления (оценки) заданного OCL-выражения. Использование данного компонента было продемонстрировано в предыдущих примерах и не вызывает трудностей. Здесь мы остановимся только на одной интересной особенности

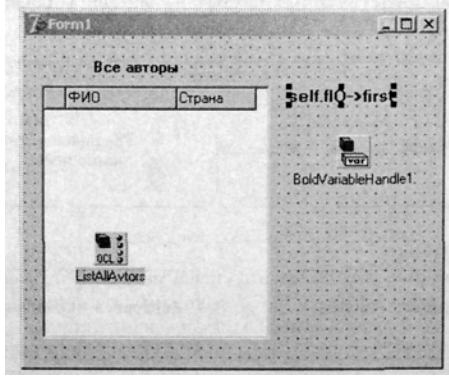


Рис. 9.2. Форма приложения

BoldLabel, а именно на принципиальной возможности применения интерфейса drag-and-drop. Для этого на базе приведенных ранее примеров (см. главу 3) создадим простое приложение, состоящее из ранее построенного модуля данных и одной формы, на которой оставим BoldGrid для отображения списка авторов, и дополнительно поместим компоненты BoldLabell и дескриптор ОП-переменной BoldVariableHandlel (рис. 9.2).

Для дескриптора ОП-переменной зададим тип переменной Collection(Avtor), как показано на рис. 9.3.

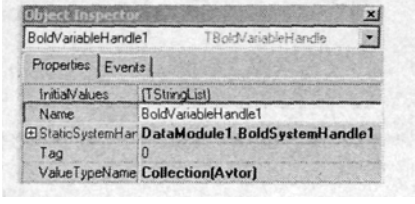


Рис. 9.3. Настройка дескриптора переменнои

Для метки BoldLabel установим следующие свойства (рис. 9.4):

- BoldHandle = BoldVariableHandle1;
- Expression = self.fio->first;
- DropMode = bdpInsert;
- NilStringrepresentation = 'Нет автора'.

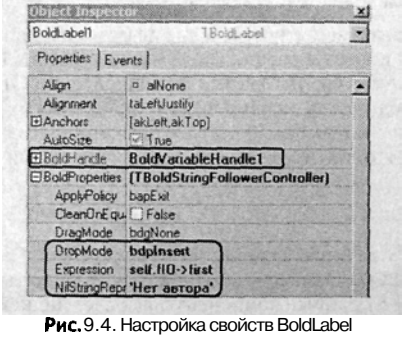


Рис. 9.4. Настройка свойств BoldLabel

Запустим приложение. По умолчанию наша метка отображает текст «Нет автора». Перетащим автора Пачеко на метку и увидим, что теперь ее текст стал отображать фамилию этого автора (рис. 9.5). Приведенный пример показывает, что даже самые простые MDA-компоненты способны активно взаимодействовать с другими MDA-компонентами и при этом поддерживать интерфейс drag-and-drop практически без участия программиста.

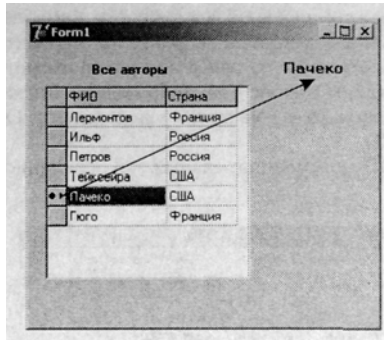


Рис. 9.5. Присваивание нового значения метки посредством drag-and-drop

BoldEdit

Этот компонент является простейшим MDA-аналогом обычного VCL-компонента TEdit. Его назначение — ввод и редактирование текста. BoldEdit обладает новым свойством ApplyPolicy, которое имеется у многих MDA-компонентов. Это свойство может принимать три различных значения.

1. bapExit — значение по умолчанию, указывает, что все изменения, сделанные в окошке редактирования, будут восприняты средой только после потери компонентом фокуса.
2. bapChange — означает, что любое изменение сразу воспринимается средой.
3. bapDemand — означает, что изменение не будет воспринято до тех нор, пока явно не будут вызваны специальные методы BMDA TBoldQueueable.ApplyAll или TBoldQueueable.ApplyAllMatching для обновления ОП. Эти методы полезны при необходимости «кэширования» проделанных изменений, и фиксации их в ОП по назначенному разработчиком событию, например при нажатии кнопки Сохранить изменения на форме приложения.

BoldGrid

Этот компонент мы уже не раз использовали в примерах. Он является MDA-аналогом компонента DBGrid. Однако возможности BoldGrid гораздо шире. Как мы уже убедились (см. главу 8), данный компонент позволяет индивидуально настраивать отображаемую в каждом конкретном столбце информацию, эффективно используя при этом OCL-навигацию по UML-модели. Поэтому, «привязав» BoldGrid к одному классу модели, мы получаем возможность отображать в различных его столбцах информацию из совершенно других классов, связанных с «исходным» классом через ассоциации-связи. Эти возможности позволяют, не применяя SQL-запросы, «собрать» в столбцах единственного компонента BoldGrid всю необходимую информацию. Не повторяя подробно уже разобранные примеры (см. главу 8), проиллюстрируем сказанное следующим. Привязав BoldGrid посредством дескриптора списка к классу Автор, который связан в нашей UML-модели ассоциацией с классом Страна, мы сможем во втором столбце BoldGrid отобразить название страны для каждого

автора (рис. 9.6). Для этого достаточно в свойстве Expression второго столбца задать навигационное OCL-выражение strana.nazvanie. А если мы захотим еще в одном столбце этой же сетки отобразить количество книг, написанных каждым автором, то можем ввести для этого столбца OCL-выражение napisal->size. И так далее. Эта возможность компонента BoldGrid чрезвычайно удобна, так как разработчик получает возможность без программирования кода и SQL-запросов выводить практически любую доступную информацию в одном компоненте BoldGrid. Ниже мы остановимся на некоторых его особенностях, не рассмотренных ранее.

Свойство BoldShowConstraints

Данное логическое свойство предоставляет разработчику удобную возможность визуально контролировать нарушения ограничений (constraints) модели. Если значение этого свойства True, то в сетке BoldGrid появляется дополнительный столбец (см. рис. 9.6), в котором отображаются специальные разноцветные индикаторы. Если для данного элемента (строки) нарушаются заданные в модели ограничения, то цвет индикатора — красный. На представленном рисунке добавленный автор Иванов нарушает сразу два ограничения — он не написал ни одной книги, и не проживает ни в какой стране, хотя в модели присутствуют размерности ролей 1..n и 1 для этих ассоциаций соответственно (рис. 9.7).

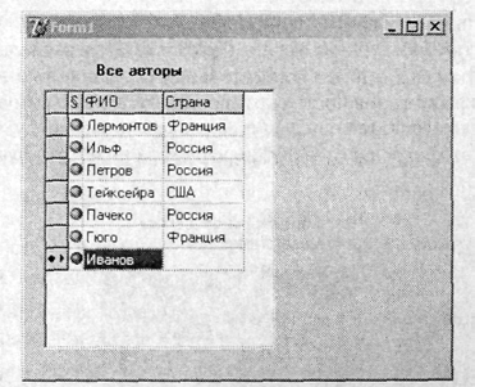


Рис. 9.6. Отображение индикаторов нарушений ограничений модели

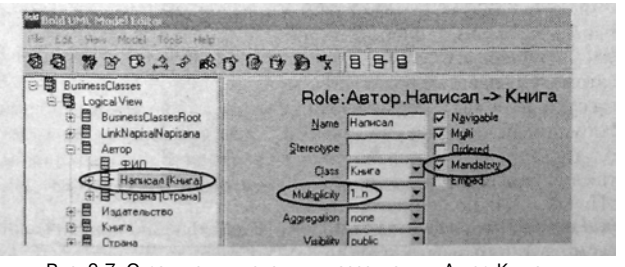


Рис. 9.7. Ограничения модели для ассоциации «Автор-Книга»

Свойство BoldHandleIndexLock

Это логическое свойство управляет взаимодействием визуального компонента и дескриптора списка, к которому данный компонент подключен. Если значение этого свойства True, то перемещение курсора по строкам сетки будет автоматически вызывать перемещение указателя в дескрипторе списка. Это бывает весьма удобно для синхронизации всех остальных визуальных MDA-компонентов, подключенных к тому же дескриптору или к другим дескрипторам, для которых данный дескриптор является корневым (о цепочках дескрипторов см. главу 7). При этом не важно, на этой форме расположены синхронизируемые визуальные компоненты или на других формах приложения.

Свойство DefaultDblClick

Это логическое свойство управляет отображением автоформ BMDA. Если его значение False, то автоформы появляться не будут. Напомним, что для отображения автоформ также необходимо добавить в список используемых модулей (Uses) модуль BoldAFPDefault.

BoldSortingGrid

В составе визуальных компонентов присутствует «усовершенствованный» эквивалент компонента BoldGrid. Он называется BoldSortingGrid и располагается на вкладке BoldMisc палитры компонентов. Отличительной особенностью этого компонента является встроенная возможность сортировки записей по возрастанию и убыванию, которая осуществляется при щелчке мышкой по заголовку столбца (рис. 9.8). Сортировка при этом осуществляется по набору записей, отображаемых в данном столбце.

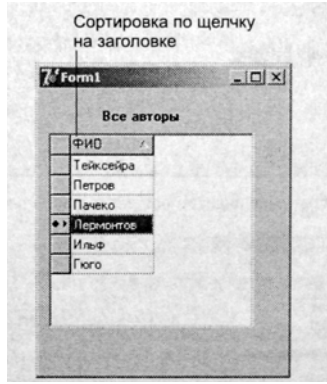


Рис. 9.8. Сортировка записей в компоненте BoldSortingGrid

В остальном использование компонента BoldSortingGrid практически не отличается от рассмотренного выше «обычного» BoldGrid.

BoldComboBox

Данный компонент предназначен для выбора одного элемента из некоторого раскрывающегося списка и присвоения его значения элементу другого списка. В этом смысле можно сказать, что BoldComboBox является в некоторой степени аналогом известного компонента DBLookupComboBox. Основными свойствами компонента являются две ссылки на ОП-дескрипторы. Свойство BoldHandle указывает на дескриптор списка, элемент которого будет изменяться при выборе конкретного значения из раскрывающегося списка. Свойство BoldListHandle указывает на дескриптор списка, элементы которого будут отображаться в выпадающем списке. Рассмотрим для иллюстрации простой пример. Используя созданное ранее приложение, оставим на форме сетку BoldGrid для отображения списка авторов и добавим компонент BildComboBox1 (рис. 9.9).

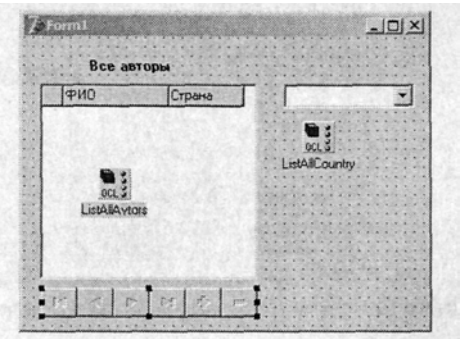


Рис. 9.9. Вид формы с компонентом BoldComboBox

Настроим свойства компонента следующим образом (рис. 9.10):

- BoldHandle = ListAllAvtors (список всех авторов);
- BoldListHandle = ListAllCountry (список стран; названия стран будут отображаться в раскрывающемся списке компонента);
- BoldProperties ▸ Expression = strana.nazvanie — это OCL-выражение определяет, какая информация будет отображаться в BoldComboBox при перемещении по списку авторов в BoldGrid;
- BoldRowProperties • Expression = nazvanie — это OCL-выражение определяет, какая информация будет отображаться в раскрывающемся списке компонента;
- BoldSelectChangeAction = bdcsSetValue — это свойство задает тип информации, передаваемой из раскрывающегося списка, в данном случае — само значение (можно передавать также индекс в списке, ссылку или текст);
- BoldSetValueExpression = strana — это OCL-выражение определяет значение, которое будет передано в качестве элемента для присвоения, в данном случае — сам объект типа Страна.

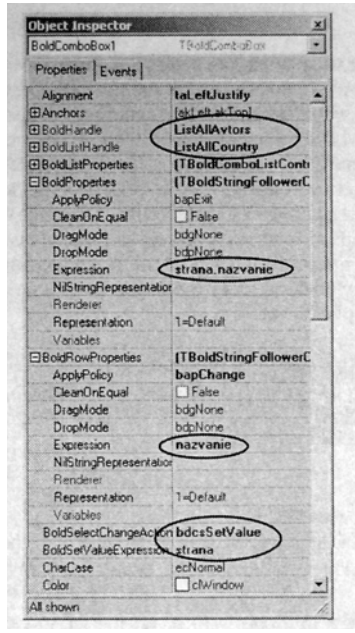


Рис. 9.10. Настройка свойств компонента BoldComboBox

Запустим приложение. Во-первых, мы увидим, что при перемещении по сетке и выборе конкретного автора в компоненте BoldComboBox в соответствии с этим меняется название страны. И во-вторых, выделив конкретного автора в сетке, мы можем открыть раскрывающийся список, выбрать любую из стран и тем самым «присвоить» ее текущему автору (рис. 9.11).

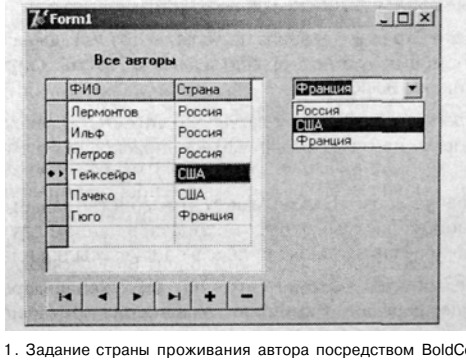


Рис. 9.11. Задание страны проживания автора посредством BoldComboBox

BoldListBox

Данный компонент является MDA-аналогом стандартного компонента TListBox. Использование его достаточно тривиально. Необходимо только помнить, что, как и предыдущий компонент, BoldListBox имеет свойство BoldRowProperties. Именно в этом свойстве задается OCL-выражение, определяющее, что именно отображается в списке. Как и обычный Delphi-компонент ListBox, BoldListBox также способен отображать данные, располагая их в нескольких столбцах. И, естественно, после двойного щелчка мышью на какой-нибудь записи в списке, так же как и в случае компонента BoldGrid, откроется автоформа для просмотра и редактирования данных (рис. 9.12), относящихся к этой записи.

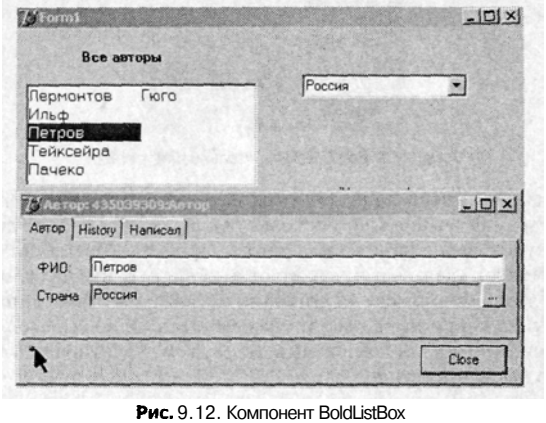


Рис. 9.12. Компонент BoldListBox

ПРИМЕЧАНИЕ

Для окрытия автоформы необходимо, чтобы свойство DefaultDBLClick имело значение True. Кроме того, в состав подключаемых модулей (раздел Uses) должен быть добавлен модуль BoldAFPDefault.

BoldCheckBox

Этот компонент предназначен для отображения и редактирования логических значений типа Boolean. При использовании BoldCheckBox необходимо различать ситуации, когда появляется возможность редактирования отображаемого им значения, от случаев, когда этот компонент способен только отобразить логический результат. Здесь можно руководствоваться следующим простым правилом — визуальное редактирование логического значения возможно только в том случае, когда данный компонент непосредственно отображает состояние логического атрибута какого-либо класса. В противном случае компонент BoldCheckBox будет только отображать «справедливость» текущего OCL-выражения, задаваемого свойством Expression данного компонента. Например, если в приведенном ранее примере простого приложения поместить на форму компонент BoldCheckBox, связать его с деск-

риптором списка авторов и задать OCL-выражение fio<>'Ильф' (то есть фамилия автора — не Ильф) в качестве его свойства Expression, то мы получим компонент «только для чтения», который будет отображать флажок только при выборе автора Ильф в списке авторов (рис. 9.13).

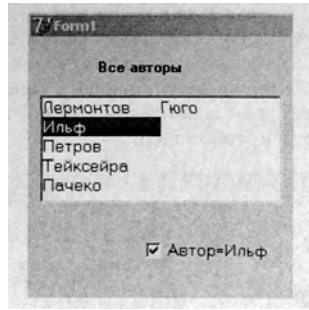


Рис. **9.13.** BoldCheckBox «только для чтения»

Попытка изменить состояние компонента (снять флажок) не приведет в такой ситуации к успеху. Это и понятно, поскольку в данном случае такое действие означало бы принудительное изменение другого атрибута — фамилии автора, к которому BoldCheckBox «не имеет никакого отношения». И наоборот, если мы добавим в состав атрибутов класса Автор новый логический атрибут, например, пол (мужчина=True, женщина=False) и укажем его в OCL-выражении свойства Expression рассматриваемого компонента, то сможем редактировать состояние такого атрибута непосредственно.

BoldPageControl

Компонент BoldPageControl является довольно интересным и несколько «экзотическим» представителем визуальных Bold-компонентов. Этот компонент в полной мере реализует основной принцип визуальных Bold-компонентов — автоматизация поведения графического интерфейса, и, если можно так выразиться, доводит этот принцип до своего «логического завершения». Смысл предоставляемой разработчику функциональности компонента BoldPageControl — автоматически открывать ту вкладку, чье имя совпадает со значением, возвращаемым OCL-выражением, которое задано в его свойстве Expression. Рассмотрим простой пример. Поместим на форму компонент BoldPageControl и обычным образом добавим в него две вкладки. В свойство Name этих вкладок введем Ilf и Petrov (английские транскрипции авторов Ильф и Петров), а в свойство Caption этих же вкладок введем Ильф и Петров. Сам компонент BoldPageControl подключим (свойство BoldHandle) к дескриптору списка авторов, а для свойства Expression введем выражение fio (фамилия автора). Запустим приложение и увидим, что ничего не происходит, то есть вкладки «не реагируют» на перемещение по списку авторов. Теперь добавим авторов Ilf и Petrov в список с помощью Bold-навигатора. После этого легко убедиться, что при выборе строки с фамилией Ilf или Petrov автоматически открывается соответствующая вкладка компонента BoldPageControl (рис. 9.14).

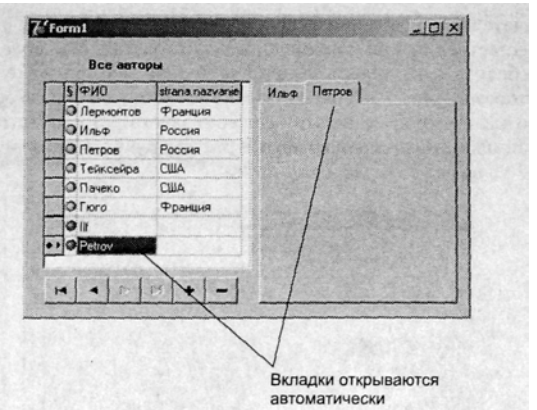


Рис. 9.14. Использование BoldPageControl

Внимательный читатель уже, видимо, понял, что для русскоязычных интерфейсов данный компонент не вполне удобен, так как мы, к сожалению, не можем назначить именам компонентов (в данном случае свойству Name вкладок) русскоязычные идентификаторы. Однако при желании эту проблему можно решить, хотя и несколько искусственным способом. Мы можем добавить (в UML-модели) новый текстовый атрибут в класс Автор, где будем хранить англоязычные транскрипции фамилий авторов (их значения при этом могут образовываться автоматически при вводе каждой фамилии), и для наглядности настроить второй столбец сетки BoldGrid на отображение значения этого атрибута. Пусть, например, наш новый атрибут будет называться alias (псевдоним). Тогда свойству Expression компонента BoldPageControl мы вместо OCL-выражения — фамилии fio, присвоим OCL-выражение alias. Теперь при перемещении по списку авторов в сетке, вкладки открываются корректно по значению псевдонима (рис. 9.15), без необходимости ввода дополнительных «англоязычных» фамилий авторов.

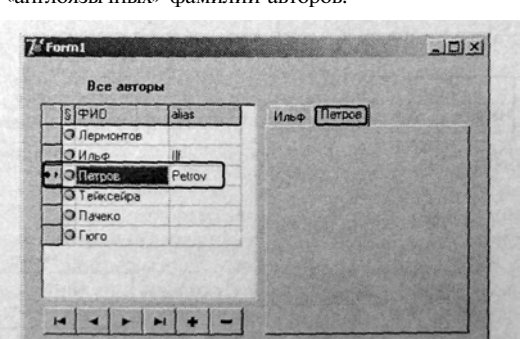


Рис. 9.15. Настройка BoldPageControl на атрибут alias

Для окончательного решения задачи удалим из сетки BoldGrid второй столбец-псевдоним (естественно, при этом он должен остаться в UML-модели и в свойстве Expression). Убедимся, что теперь мы вполне можем работать и с русскоязычными идентификаторами (рис. 9.16). Использование компонента BoldPageControl может быть полезно для синхронизации отображения некоторой сопутствующей детальной информации со списком (например, в рассмотренном примере на полях вкладок можно отображать биографии авторов).

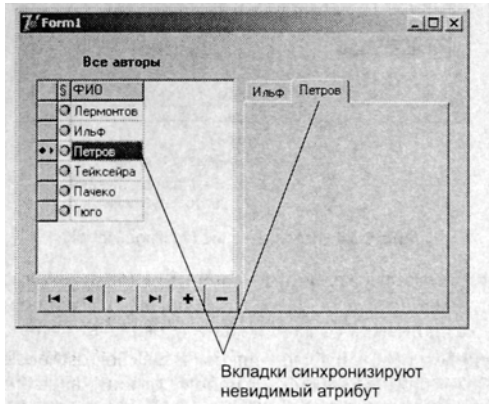


Рис. 9.16. Синхронизация вкладок по невидимому атрибуту

BoldTreeView

Компонент BoldTreeView является самым сложным из состава визуальных Bold-компонентов. Его назначение, как легко догадаться по названию, — отображение набора некоторых элементов в виде «дерева», то есть иерархической структуры. Проще всего понять принципы работы этого компонента на конкретном примере. Для этого будем использовать созданную ранее UML-модель (см. главу 8), фрагмент которой представлен на рис. 9.17.

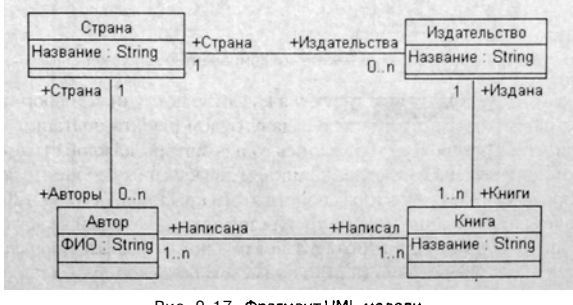


Рис. 9.17. Фрагмент UML-модели

Из набора элементов модели мы в нашем примере будем использовать классы Страна, Автор и Книга. Поставим следующую задачу — отображать в виде «дерева» состав авторов, проживающих в выбранной стране, и написанные каждым автором книги, причем книги должны быть связаны с соответствующим автором в иерархическом дереве (рис. 9.18).

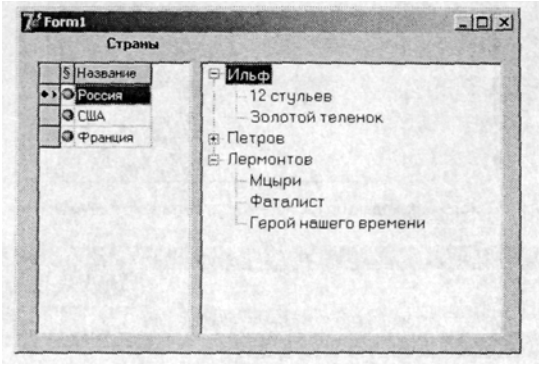


Рис. 9.18. Конечный вид приложения

Поместим на форму компонент дескриптор списка BoldListHandle, присвоим ему название ListCountry и подключим его к набору стран, введя OCL-выражение Strana.allInstances в качестве его свойства Expression. Поместим на форму компонент DBGrid и подключим его к дескриптору списка стран ListCountry. И, наконец, добавим на форму компонент BoldTreeView1 и также подключим его (свойство BoldHandle) к дескриптору списка стран ListCountry (рис. 9.19).

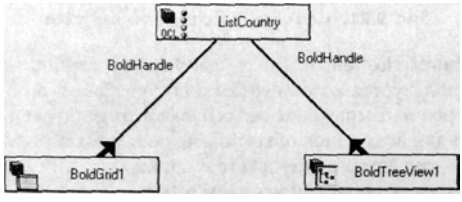


Рис. 9.19. Диаграмма настройки компонентов приложения

Таким образом, компонент BoldTreeView в качестве источника информации будет иметь коллекцию стран. Поставленную задачу будем решать поэтапно. На первом этапе добьемся, чтобы в дереве отображались только авторы каждой страны, без книг. Для настройки свойств нашего «дерева» кликнем дважды по компоненту BoldTreeView, при этом отобразится окно редактора свойств этого компонента (рис. 9.20).

Добавим элемент-описание нового уровня иерархии, который будет отвечать за отображение авторов книг. Для этого нажмем на кнопку Add List Fragment, расположенную справа в редакторе свойств, и увидим, что иод «корнем» нашего «дерева» появился новый элемент. Настроим его свойства следующим образом (рис. 9.21).

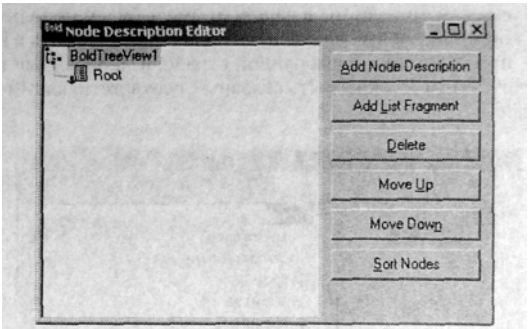


Рис. 9.20. Редактор свойств компонента BoldTreeView

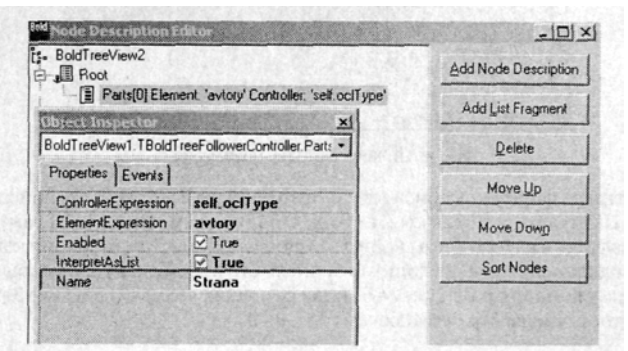


Рис. 9.21. Настройка свойств уровня иерархии

Свойству ControllerExpression присвоим OCL-выражение self.oclType. Данное свойство используется компонентам для поиска описаний узлов иерархии на каждом уровне. Выражение self.oclType устанавливает текущий контекст (см. главу 5) для всех типов объектов, включаемых в данный уровень, в нашем случае — тип Автор, в результате компонент будет искать для отображения на этом уровне иерархии все элементы данного типа.

Свойству ElementExpression присвоим значение роли ассоциации, связывающей классы Страна и Авторы (см. рис. 9.17) — avtory, то есть элемент этого уровня будет содержать всех авторов.

Свойству InterpretAsList присвоим значение True, поскольку мы в дальнейшем хотим добавить еще один уровень иерархии, включающий книги каждого автора.

Свойству Name присвоим сторону Strana, так как контекст данного уровня должен совпадать с контекстом описания элемента уровня (свойство ElementExpression).

Теперь добавим собственно узел иерархии, отображающей авторов. Для этого нажмем на кнопку с названием AddNodeDescription и настроим новый узел следующим образом (рис. 9.22).

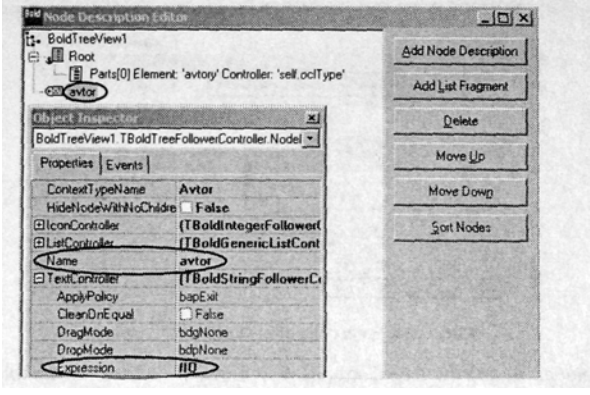


Рис. 9.22. Настройка свойств узла

- Свойству ContextTypeName присвоим значение Avtor, это свойство указывает тип элемента.
- Свойству Expression зададим OCL-выражение Fn, то есть мы хотим отображать фамилии авторов.

Запустим приложение и убедимся, что мы справились с первой частью поставленной задачи — отображать авторов, проживающих в данной стране (рис. 9.23).

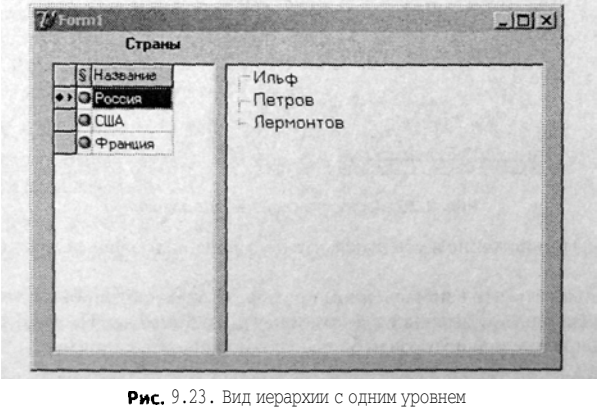


Рис. 9.23. Вид иерархии с одним уровнем

Для создания второго уровня иерархии повторим вышеописанные действия. но по отношению не к корневому элементу, а к новому узлу Avtor. Сначала создадим описание второго уровня иерархии и настроим его свойства (рис. 9.24.).

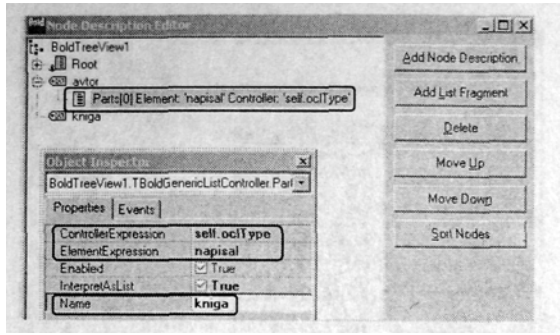


Рис. 9.24. Настройка описания второго уровня

Далее добавим новый узел в «дерево» для отображения списка книг и настроим его свойства (рис. 9.25).

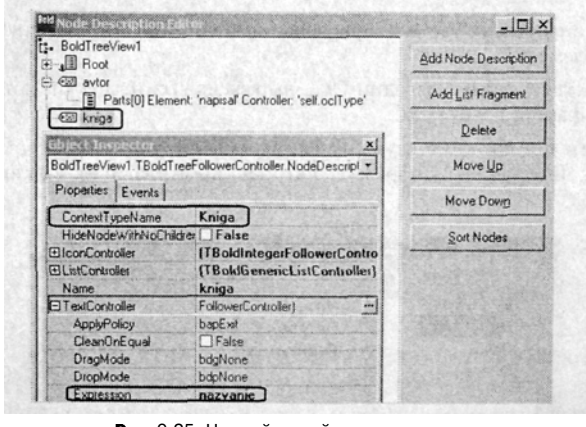


Рис. 9.25. Настройка свойств узла для книг

Запустим приложение и убедимся, что поставленная задача решена полностью (рис. 9.26).

Стоит отметить, что в приведенном примере продемонстрирована только весьма небольшая часть возможностей компонента BoldTreeView. На самом деле этот компонент предоставляет гораздо больше возможностей, например:

- отображение графических элементов (значков) для элементов иерархии;

отображение элементов, не входящих в объектное пространство (создаваемых во время выполнения приложения);

возможность описания нескольких уровней для одного узла;

возможность создания большого количества уровней иерархии.

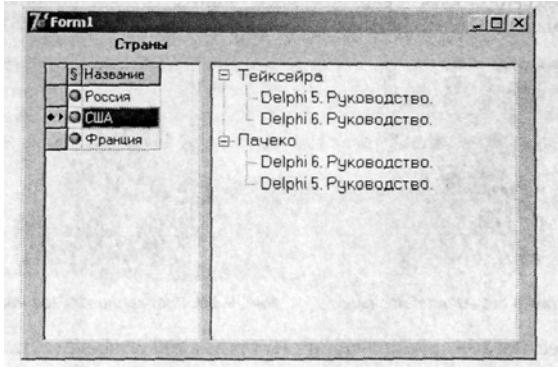


Рис. 9.26. Вид приложения в работе

Для описания всех возможностей этого компонента потребовалась бы отдельная большая глава. Более подробная информация о них приведена в сопроводительной документации к продукту Bold for Delphi и проиллюстрирована в двух прилагаемых к нему демонстрационных программах, предоставляемых вместе с исходными текстами.

BoldImage

Данный компонент предназначен для отображения графической информации. Такая информация может содержаться в атрибутах типа BLOB, TypedBLOB, BlobImageBMP, BlobImageJpeg и т. д. Использование данного компонента особых трудностей не представляет, отметим здесь некоторые его особенности.

Загрузка изображения из файла

Для загрузки в компонент BoldImage изображения из файла во время выполнения программы можно использовать его свойство LoadFromFile следующим образом:

```
BoldImage1.LoadFromFile(filename);
```

где filename — имя файла с изображением.

Использование формата JPEG

Для поддержки работы с JPEG-файлами необходимо добавить в секцию Uses два модуля:

```
Uses .... JPEG,BoldImageJpeg;
```

Масштабирование изображения

Компонент имеет свойство StretchMode, которое может принимать следующие значения:

- bsmNoStretch — картинка отображается без масштабирования (рис. 9.27);
- bsmStretchProportional — картинка масштабируется пропорционально, вписываясь максимальным размером в размер компонента (рис. 9.28):

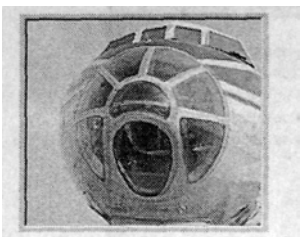


Рис. 9.27. Картинка без масштабирования

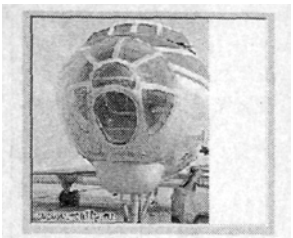


Рис. 9.28. Пропорциональное масштабирование

- bsmStretchToFit — картинка принимает размеры компонента (рис. 9.29);
- bsmStretchToScale — картинка масштабируется пропорционально с учетом свойства Scale (рис. 9.30).

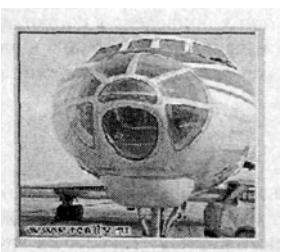


Рис. 9.29. Масштабирование по размеру компонента

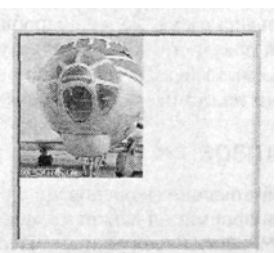


Рис. 9.30. Масштабирование с заданной степенью (Scale=50)

Рендереры

При описании ряда визуальных MDA-компонентов мы сознательно не акцентировали внимание на одном общем свойстве, присущем им, — свойстве Renderer. Оноподключает к визуальному компоненту специальный MDA-компонент с одноименным названием, трудно переводимым на русский язык — Renderer («прорисовщик» или «представитель»). Понятие рендеринга включает обработку некоторой информации и ее отображение (прорисовку), и широко применяется, например,

в графических 3D-пакетах. В BMDA это понятие во многом схоже, только под прорисовкой понимается установка некоторых свойств подключенного визуального компонента — например, задание цвета, размера и вида шрифта и т. д. Другими словами, рендереры служат для представления информации. Их использование позволяет гибко управлять указанными свойствами визуальных компонентов в зависимости от результата обработки некоторой информации. Причем имеется в виду управление оперативное, то есть свойства отображения задаются на этапе выполнения приложения. В качестве примера рассмотрим следующую задачу. Допустим, мы хотим, чтобы в сетке BoldGrid все российские авторы были окрашены в красный цвет. Как добиться этого? Поместим на форму строковый рендерер BoldAsStringRendererl с вкладки BoldControls палитры компонентов Delphi (рис. 9.31).

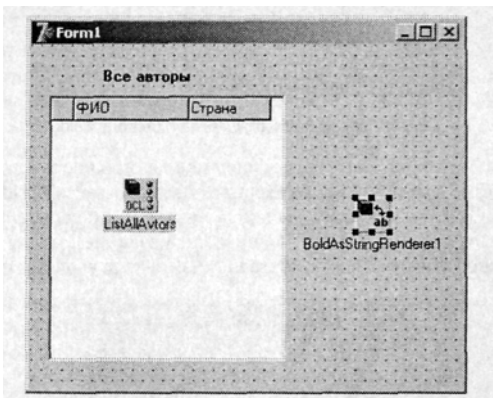


Рис. 9.31. Вид формы с компонентом строкового рендерера

Особенностью рендереров является то, что они не подключаются к визуальным компонентам. Наоборот, визуальные MDA-компоненты подключаются к рендерерам. В данном случае мы будем подключать к рендереру компонент BoldGridl, точнее, его столбцы. Для этого дважды щелкнем на BoldGridl и в вызванном редакторе столбцов выберем столбец ФИО. В инспекторе объектов установим свойству Renderer для этого столбца значение BolsAsStringRendererl (рис. 9.32).

Теперь осталось указать рендереру, какую именно информацию он должен обрабатывать и какими свойствами управлять. Поскольку нашей задачей является изменение цвета, то воспользуемся событием OnSetColor, для которого напишем код обработки (листинг 9.1).

Листинг 9.1. Обработка события рендерера для управления цветом

```
procedure TForm1.BoldAsStringRenderer1SetColor(Element:
    TBoldElement;
  var AColor: TColor; Representation: Integer; Expression:
    String);
```

```
var st:string;
begin
  st:=element.EvaluateExpressionAsString('strana.nazvanie',1);
  if st='Россия' then AColor:=clRed;
end;
```

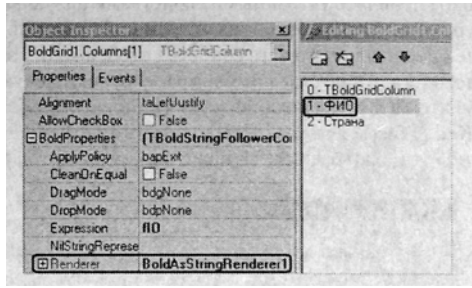


Рис. 9.32. Подключение рендерера к столбцу BoldGrid

В представленном коде мы воспользовались возможностью оценки OCL-выражения посредством метода EvaluateExpressionAsString класса TBoldElement. В качестве собственно OCL-выражения здесь используется строка strana.nazvanie, которая в контексте выбранного автора возвращает название страны. После запуска приложения можно убедиться, что поставленная задача выполнена (рис. 9.33).

Рис. 9.33. Приложение в работе

Аналогичным образом можно использовать и другие события рендерера, например OnSetFont — для установки параметров шрифта, OnGetAsString — для замены текста, отображаемого в компоненте, на любой другой текст, в зависимости от задаваемых условий, и т. д.

Таким образом, использование рендереров позволяет осуществлять так называемый mapping (отображение), то есть задавать правила отображения содержи-

мого объектного пространства на графический интерфейс приложения. Учитывая, что в обработчики событий рендереров просто включаются методы оценки OCL-выражений, эти компоненты в принципе позволяют разработчику полностью управлять внешним видом приложения, задавая чрезвычайно гибкие правила для каждого визуального элемента графического интерфейса. Отметим, что один рендерер может «обслуживать» одновременно несколько различных визуальных компонентов, это позволяет создать в том числе и определенное стилевое оформление.

Автоформы

Общие сведения. Структура автоформы

Автоформы — это автоматически генерируемые средой BMDA специальные формы для просмотра и редактирования данных, поддерживающие интерфейс drag-and-drop. Мы уже имели дело с ними, когда рассматривали создание простого приложения (см. главу 3). Автоформы инициализируются по умолчанию при двойном щелчке на некоторых визуальных MDA-компонентах. Для использования автоформ необходимо в разделе Uses модуля формы прописать модуль BoldAFPDefault. Использование автоформ позволяет легко осуществлять навигацию по модели. Рассмотрим более подробно структуру генерируемых автоформ на примере, приведенном в главе 9. При двойном щелчке мышью на строке сетки (отображающей список стран) со словом Россия появляется соответствующая автоформа. По умолчанию она имеет вид, представленный на рис. 9.34. Заголовок-название автоформы определяется тег-параметром DefaultStringRepresentation (см. главу 4). Если он не установлен, то отображается внутренняя индексная информация BMDA.

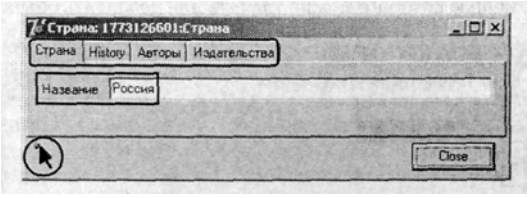


Рис. 9.34. Вид автоформы по умолчанию

В средней части автоформы располагается область атрибутов (обведена прямоугольником на рис. 9.34). В ней приведены названия атрибутов класса (в данном случае у класса Страна имеется единственный атрибут Название), рядом с каждым названием атрибута располагается окно редактирования, отображающее текущее содержимое данного атрибута (в нашем случае — название страны Россия).

Непосредственно на автоформе в указанном окне редактора можно изменить значение любого атрибута.

В верхней части автоформы располагается область мультиассоциаций (она обведена скругленным прямоугольником). Эта область представляет собой набор вкладок, количество которых соответствует количеству ассоциаций данного класса-

плюс одна главная вкладка с названием класса (в данном случае — Страна). При этом все эти ассоциации имеют кратность больше 1 (мультиассоциации).

Если мы посмотрим на фрагмент UML-модели (рис. 9.35), то убедимся, что у класса Страна имеются две ассоциации, связывающие его с классами Автор и Издательство. Названия вкладок на автоформе соответствуют названиям концов этих ассоциаций — Авторы и Издательства. Вкладка History предназначена для целей отслеживания изменений системы, и сейчас мы обсуждать ее не будем. Отметим, что для ее скрытия достаточно установить глобальной переменной BoldShowHistoryInAutoForms значение False. Это можно сделать, например, в секции инициализации модуля.

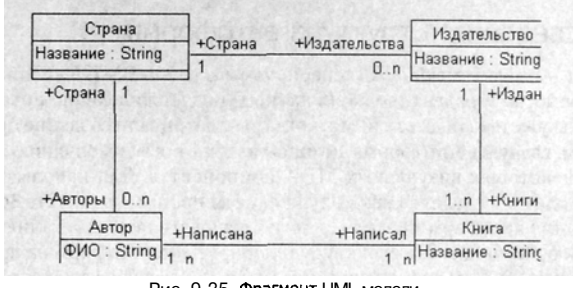


Рис. 9.35. Фрагмент UML-модели

Если мы выберем вкладку Авторы, то на автоформе откроется окно соответствующей ассоциации (рис. 9.36), в котором будут отображены все объекты связанного класса Авторы в сетке типа BoldGrid, доступной для редактирования, а также Bold-навигатор, с помощью которого можно удалять или добавлять объекты типа Автор.

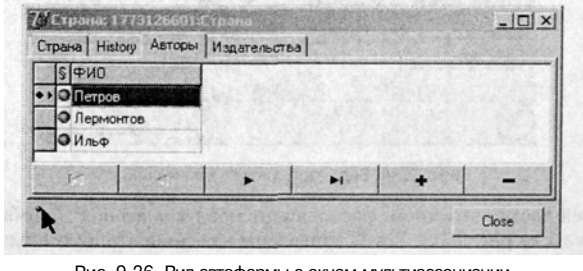


Рис. 9.36. Вид автоформы с окном мультиассоциации

Автоформы обладают свойством автоматически генерировать производные автоформы при двойном щелчке мышью. Так, если мы щелкнем по строке сетки автоформы с фамилией автора Ильф, то отобразится производная автоформа (рис. 9.37), которая содержит информацию об этом авторе. Мы могли бы получить ее и другим способом — дважды щелкнув по соответствующей строке сетки главной формы приложения, отображающей список всех авторов. Отметим, что авто-

форма автора отличается от автоформы страны наличием серого поля с названием страны Россия (см. рис. 9.37), которое не допускает непосредственного редактирования. Такие серые поля отображают однократную ассоциацию данного класса (в модели заложено правило — автор может жить только в одной стране).

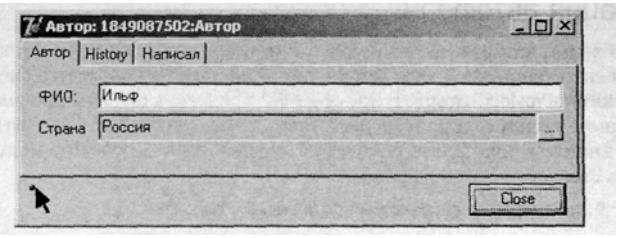


Рис. **9.37.** Вид производной автоформы с однократной ассоциацией

Для редактирования ассоциаций в автоформах активно используется интерфейс Drag-and-Drop. Любая автоформа содержит «точку связи» (она обведена кружком на рис. 9.34), отображаемую в виде стрелки с красной точкой. Если «потащить» автоформу, изображенную на рис. 9.34, за эту точку мышью на автоформу, показанную на рис. 9.37, и отпустить кнопку мыши над серым полем с названием страны, то таким образом можно соединить конкретную страну с конкретным автором, при этом в сером поле отобразится название новой страны. Точно так же можно перетаскивать автоформы на сетки внутри других автоформ, реализуя мультиасоциации. Напомним, что можно и непосредственно перетащить строку сетки BoldGrid обычной формы на автоформу, как это делалось в главе 3.

Все эти возможности обеспечивают кроме удобства еще и непротиворечивость данных, поскольку непосредственное редактирование подобных серых полей для однократных ассоциаций запрещено, то есть ролям ассоциаций можно присваивать только существующие объекты. Напротив, для мультиассоциаций можно добавлять объекты связанного класса непосредственно на автоформе. Так, если мы, используя навигатор на автоформе, изображенной на рис. 9.36, добавим новых, не существующих до этого момента, авторов (хотя сама эта автоформа привязана не к классу Автор, а к классу Страна!), то мы беспрепятственно проделаем эту операцию, и при этом в класс Автор будут автоматически добавлены новые объекты.

Ограничения

Автоформы имеют ряд ограничений. Они не имеют визуальных элементов для отображения атрибутов типа Memo или атрибутов типа BLOB. Автоформы не настраиваются обычным образом, то есть мы не можем поместить на них раскрывающиеся списки, кнопки и т. п.

ПРИМЕЧАНИЕ

Индивидуальная настройка автоформ в принципе возможна только на уровне исходного текста. Для этой цели можно воспользоваться поставляемым исходным текстом программного модуля BoldAFPDefault.pas.

Поэтому, с точки зрения практики, автоформы целесообразно использовать для просмотра содержимого объектного пространства с целью отладки либо для редактирования простых текстовых или числовых атрибутов.

Генерация автоформ

При необходимости разработчик может инициировать генерацию автоформы самостоятельно, например нажатием кнопки. Для этого используется специальный класс AutoFormProviderRegistry. В листинге 9.2 приведена простая процедура, которую можно использовать для генерации автоформы непосредственно из программного кода приложения. Для ее использования необходимо добавить модуль BoldAFP в секцию Uses.

Листинг 9.2. Генерация автоформы программным способом

```
procedure ShowAutoFormForClass(ClassName:String);
var F : TForm;
F:=AutoFormProviderRegistry.FormForElement(BoldSystemHandle
.System.ClassByExpressionName[(ClassName)];
if Assigned(F) then F.Show;
end;
```

Здесь BoldSystemhandle — это системный дескриптор, а ClassName — имя класса модели, для которого необходимо сгенерировать автоформу.

Управление отображением атрибутов

Как было сказано выше, автоформа не способна корректно отображать атрибуты определенных типов. Если не предпринимать никаких действий, то такие атрибуты на автоформе будут отображены в обычных однострочных текстовых полях редактирования, При этом, например, для атрибута, содержащего графическое изображение или RTF-текст (BLOB-атрибуты), содержимое указанных окошек редактирования может выглядеть довольно странно и быть совершенно нечитаемым. С целью управления составом отображаемых на автоформах атрибутов можно воспользоваться компонентом BoldPlaceableAFP с вкладки BoldMisc. Этот компонент поддерживает событие OnMemberShouldBeDisplayed, в обработчик которого можно поместить программный код для формирования условия отображения конкретного атрибута. Пример такого кода приведен в листинге 9.3.

Если мы поместим указанный компонент на главную форму и введем приведенный код обработки, то увидим, что фамилия автора на автоформах отображаться не будет (рис. 9.38). В данном случае мы программно проверяем, совпадает ли название текущего атрибута с выражением Avtor.fIO, и если это так, то присваиваем признаку отображения ShowMember значение False (см. листинг 9.3).

Листинг 9.3. Управление отображением атрибута

```
procedure TForm1.BoldPlaceableAFP1MemberShouldBeDisplayed(
  Member: TBoldMemberRTInfo; var ShowMember: Boolean);
var st:string;
begin
st:=member.EvaluateExpressionAsString('self',1);
```

```
if st='Avtor.fIO' then ShowMember:=false;
end;
```

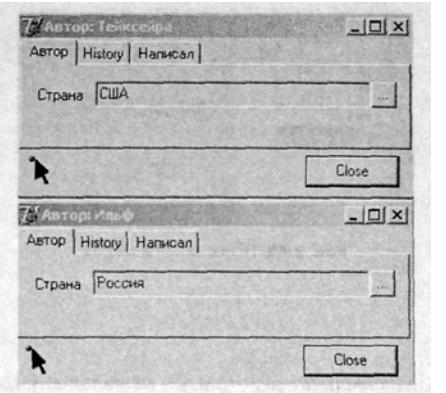


Рис. 9.38. Вид автоформ с запрещенным для отображения атрибутом

Дополнительные инструменты

BoldCaptionController

Данный компонент предназначен для автоматического формирования визуального наименования любого VCL-компонента (свойства Caption) в зависимости от значения OCL-выражения, задаваемого в его свойстве Expression. Поместим на форму компонент BoldCaptionController и подключим его к дескриптору списка стран из разобранного ранее примера (рис. 9.39). Также добавим на форму обычный VCL-компонент Panell и укажем его в свойстве TrackControl компонента BoldCaption-Controller.

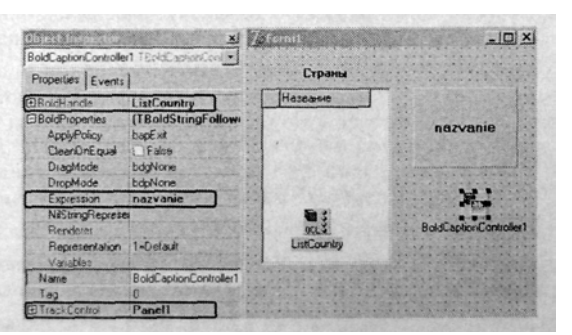


Рис. 9.39. Настройка BoldCaptionController

Запустим приложение и убедимся, что при переходе по списку стран на панели отображается название текущей страны (рис. 9.40).

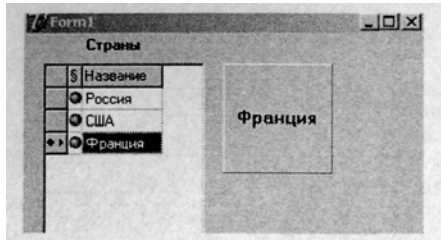


Рис. 9.40. Приложение в работе

Резюме

В этой главе мы познакомились с некоторыми визуальными компонентами среды Bold for Delphi. Из рассмотренных примеров становятся ясными основные принципы функционирования таких компонентов, а именно — максимальная автоматизация поведения графического интерфейса приложения и его синхронизация с состоянием бизнес-уровня, задаваемым дескрипторами объектного пространства.

Работа с уровнем данных

Возможность эффективной работы с данными является одной из ключевых функциональных возможностей, предоставляемых средой Bold for Delphi. При этом Bold не ограничивается только взаимодействием с СУБД, а предоставляет удобные средства для автоматической генерации баз данных по UML-модели приложения. Кроме того, в состав арсенала инструментов Bold for Delphi входят эффективные средства для сохранения объектов в XML-документах. В этой главе мы ознакомимся с основными компонентами Bold for Delphi, предназначенными для работы с данными, а также на многочисленных примерах рассмотрим вопросы практического применения основных возможностей этой среды для работы с реляционными СУБД и XML-документами.

Функции уровня данных

Уровень данных, носящий в Borland MDA название Persistence Layer, предназначен для обеспечения реализации следующих основных функций.

- Сохранение элементов (объектов и ассоциаций) объектного пространства в долговременной памяти (как правило, на жестком диске, хотя для этой цели могут использоваться и любые другие устройства хранения — дискеты, flash-диски, CD/DVD-RW и т. д. при наличии стандартного доступа к ним из приложения). При этом сохранены могут быть только те объекты и связи, для которых в модели приложения задано свойство Persistent.
- Загрузка элементов объектного пространства из долговременной памяти.
- Поддержка механизма объектно-реляционного отображения, то есть преобразования объектной UML-модели в структуру реляционной базы данных.
- Генерация схемы реляционной базы данных по имеющейся объектной модели.

- Преобразование OCL-выражений в операторы SQL (так называемый механизм «OCL2SQL»).

Структура и состав компонентов

На рисунке 10.1 представлена схема взаимодействия объектного пространства и уровня данных. Центральным элементом, организующим это взаимодействие, является дескриптор уровня данных. В качестве такого дескриптора может выступать как компонент BoldPersistenceHandleFileXML, предназначенный для сохранения ОП в XML-файле (мы уже использовали этот компонент при описании создания простого приложения (см. главу 3), так и компонент BoldPersistenceHandleDB, предназначенный для организации взаимодействия с реляционными СУБД.

Для обеспечения работы с конкретными типами СУБД в состав Bold for Delphi включены специальные компоненты-адаптеры баз данных.

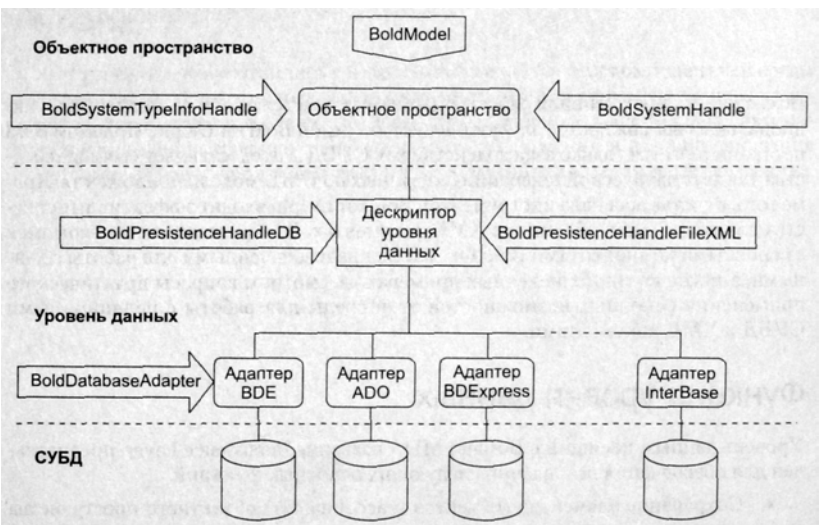


Рис. 10.1. Состав уровня данных и структура его связей

В рассматриваемой версии Borland MDA имеются следующие основные адаптеры баз данных:

- BoldDataBaseAdapterBDE — обеспечивает подключение к СУБД через Borland Database Engine;
- BoldDataBaseAdapterADO — обеспечивает подключение к СУБД через интерфейс ActiveX Data Objects (ADO);
- BoldDataBaseAdapterIB — обеспечивает подключение к СУБД Interbase;

- BoldDataBaseAdapterDBX — обеспечивает подключение к СУБД через интерфейс DBEXpress.

Кроме вышеперечисленных, имеются адаптеры для организации взаимодействия с данными посредством SOAP (Simple Object Access Protocol), а также для СУБД DBISAM и для пакета SQLDirect. Впрочем, как будет показано ниже, разработчик при необходимости имеет возможность создать собственный адаптер СУБД.

Работа с СУБД

Подключение уровня данных

Рассмотрим, как на практике происходит подключение к СУБД. Для этого используем ранее созданное приложение (см. главу 8), состоящее из одной формы и модуля данных. Модель приложения (рис. 10.2) оставим без изменений.

Рис. 10.2. Используемая модель

ВНИМАНИЕ

Еще раз обратим внимание, что при работе в среде Bold for Delphi все русскоязычные идентификаторы элементов UML-модели должны быть заменены на англоязычные. Далее везде будет предполагаться, что представленная UML-модель обработана специальным скриптом (см. главу 4) для транслитерации идентификаторов элементов модели. При этом все символы кириллицы заменяются на латинские символы (в соответствии с используемой в тексте скрипта таблицей), символ пробела заменяется символом подчеркивания, а к названию ассоциаций в начале добавляется строка Link (так, например, для ассоциации Автор–Книга будет использовано выражение LinkNapisalNapisana). Полный исходный текст используемого скрипта и инструкции по его применению приведены в конце книги (см. приложение А).

Удалим из модуля данных ранее используемый компонент BoldPersistenceHandle-FileXML, а вместо него добавим компоненты BoldPersistenceHandleDBI, BoldDatabase-AdapterIB1 с вкладки BoldPersistence, а также компонент, представляющий базу данных Interbase — IBDatabase1 (рис. 10.3).

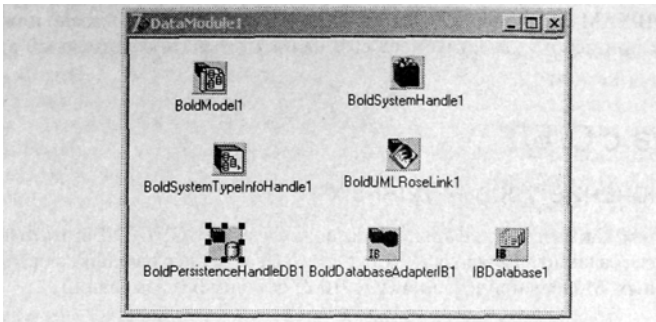


Рис. 10.3. Состав компонентов модуля данных

Настроим добавленные компоненты следующим образом (рис. 10.4).

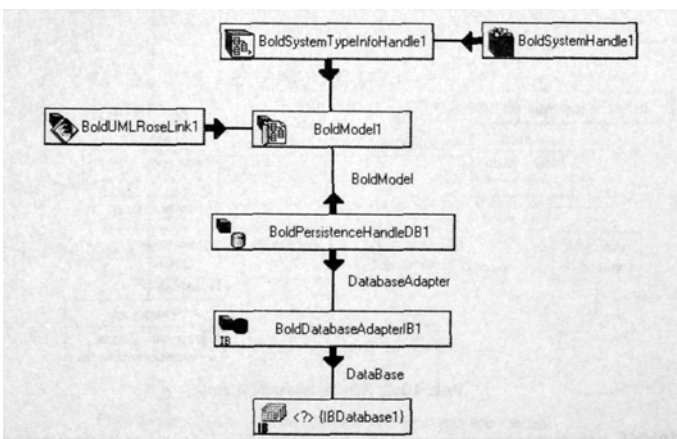


Рис. 10.4. Диаграмма настройки свойств компонентов модуля данных

Создадим пустую базу данных Interbase с помощью стандартного приложения IBConsole. Присвоим ей имя LIB.GDB и укажем путь размещения, например: C:\LIB.GDB (рис. 10.5).

Подключим компонент IBDatabase1 к созданной базе данных, а его свойству SQLDialect присвоим значение 1. Проверим соединение с БД (рис. 10.6).

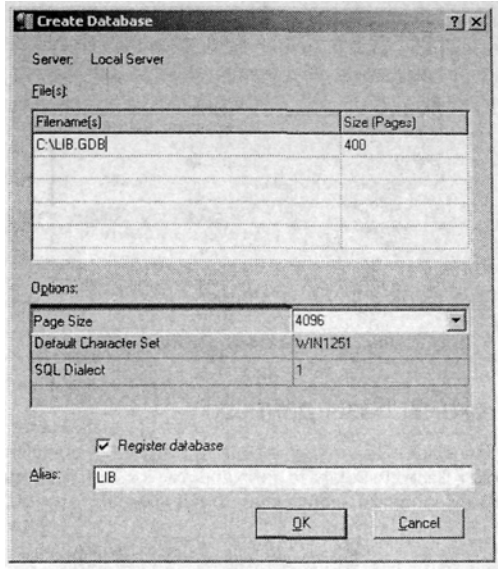


Рис. 10.5. Создание пустой базы данных

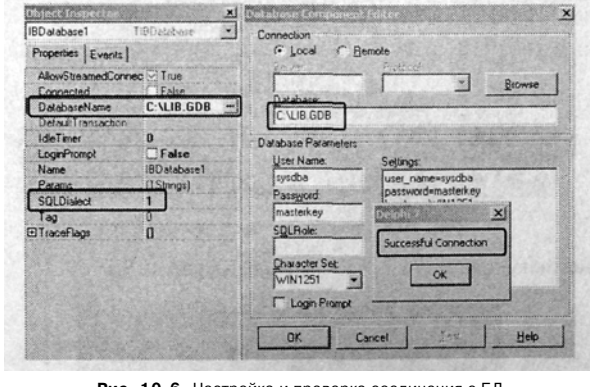


Рис. 10.6. Настройка и проверка соединения с БД

Далее в инспекторе объектов настроим адаптер базы данных, в данном случае одновременно указав тип СУБД и используемый SQL-диалект, выбрав из раскрывающегося списка свойства DatabaseEngine значение dbeInterbaseSQLDialect1 (рис. 10.7).

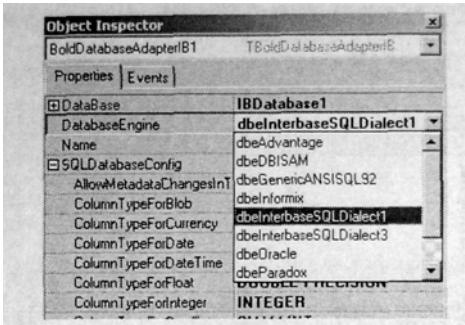


Рис. 10.7. Настройка адаптера СУБД

Генерация схемы базы данных

Для генерации базы данных запустим редактор моделей и в главном меню выберем команду Tools • Generate Database либо просто нажмем на кнопку верхней панели инструментов с условным изображением БД (рис. 10.8).

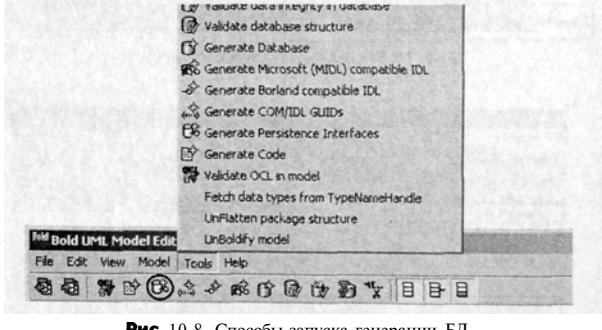


Рис. 10.8. Способы запуска генерации БД

Появится окно с запросом, представленное на рис. 10.9.

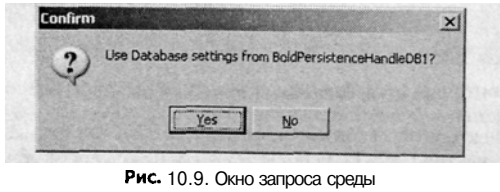


Рис. 10.9. Окно запроса среды

Данное окно запрашивает подтверждение использования компонента BoldPersistenceHandleDB1 для генерации схемы БД. Если бы наше приложение содержало

несколько таких компонентов, то после ответа No мы бы последовательно получили такие же окна с запросами об использования других аналогичных компонентов. В данном случае мы просто нажмем кнопку Yes. Далее поведение среды зависит от того, пустая наша база данных или уже заполнена какой-то информацией. Если база данных уже содержит информацию, то будет выведено окно-предупреждение (рис. 10.10), сообщающее о том, что все данные в БД будут потеряны в результате генерации структуры.

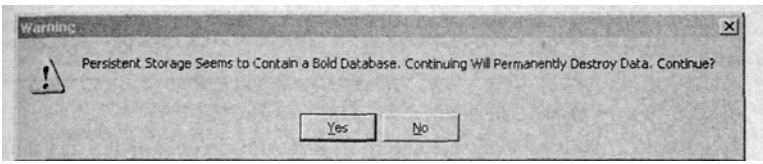


Рис. 10.10. Предупреждение о возможной потере данных

В нашем случае база данных пока пуста, поэтому среда Bold сразу начнет генерировать схему БД в соответствии с нашей UML-моделью. Ход генерации схемы можно отслеживать по информации в строке состояния редактора. Время, затрачиваемое на процедуру генерации структуры БД, зависит от количества элементов модели и от быстродействия компьютера. Для нашей модели оно составляет порядка 2—3 секунд. После окончания генерации БД мы можем снова воспользоваться утилитой IBConsole и посмотреть, какие таблицы базы данных создал Bold.

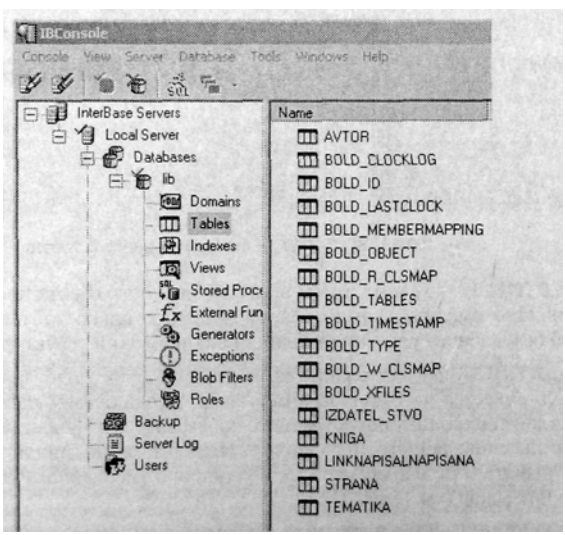


Рис. 10.11. Состав сгенерированных таблиц БД

Состав системных таблиц

Нетрудно заметить, что в составе таблиц (рис. 10.11), автоматически генерируемых средой Bold for Delphi, присутствует и те, которые мы «не заказывали». Их создала среда для собственного использования. Имена таких *системных* таблиц начинаются с префикса «BOLD_».

ПРИМЕЧАНИЕ

При желании разработчик может самостоятельно указать префикс, добавляемый средой Bold for Delphi к названиям системных таблиц. Для этого достаточно задать необходимый префикс в строковом свойстве компонента-адаптера СУБД (в рассматриваемом примере — BoldDatabaseAdapterIB).

Рассмотрим состав и назначение системных таблиц.

- BOLD_ID — данная таблица всегда содержит единственное поле — целое значение ID, используемое средой для присвоения новым объектам при их первом сохранении в базе данных. Начальное значение этого поля равно 1.
- BOLD_TYPE — эта таблица задает и хранит соответствие между конкретным классом модели и целым значением, определяющим его тип. В рассматриваемом примере данная таблица содержит правила отображения типов показанные на рис. 10.12.

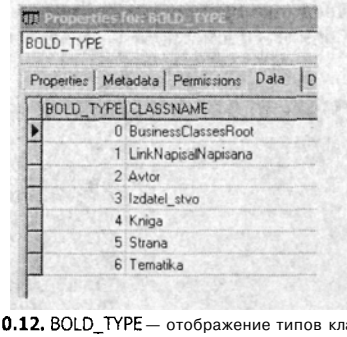


Рис. 10.12. BOLD_TYPE — отображение типов классов

BOLD_TABLES — хранит имена всех таблиц (включая системные) базы данных. Информация из этой таблицы используется средой для анализа условий безопасного удаления таблиц при обновлении структуры базы данных.

BOLD_TIMESTAMP — содержит единственное целое поле, значение которого увеличивается на 1 при каждом вызове метода UpdateDatabase (то есть при каждой синхронизации объектного пространства и уровня данных). Начальное значение данного поля равно 0. Использование данной таблицы можно разрешать или запрещать путем установки глобального свойства модели UseTimeStamp (см. главу 4).

BOLD_XFILES — данная таблица сохраняет всю историю базы данных. При создании любого объекта его глобальный уникальный идентификатор (GUID)

заносится в данную таблицу и никогда из нее не удаляется, даже если сам объект был удален из базы данных. Таблица используется при синхронизации информации с внешними базами данных. Кроме GUID каждого объекта, данная таблица хранит временную метку, соответствующую моменту его последнего изменения. Использование данной таблицы можно разрешать или запрещать путем установки глобального свойства модели UseXFiles (см. главу 4). Использование временной метки в данной таблице можно разрешать или запрещать путем установки глобального свойства модели UseTimeStamp (см. главу 4).

- BOLD_CLOCKLOG — предназначена для привязки целочисленных значений меток времени TimeStamp к физическим значениям даты и времени. При большом объеме изменений базы данных можно настроить параметр ClockLogGranularity, управляющий временной детализацией соседних записей. Использование данной таблицы можно разрешать или запрещать путем установки глобального свойства модели UseClockLog (см. главу 4).
- BOLD_LASTCLOCK — используется совместно с предыдущей таблицей для хранения времени занесения в нее последней записи.

Остальные системные таблицы предназначены для реализации механизма «эволюции базы данных» и будут рассмотрены позднее.

Структура генерируемых таблиц

Рассмотрим вкратце, как среда Bold for Delphi преобразует элементы UML-модели (классы, атрибуты, ассоциации) в таблицы и поля базы данных. Воспользовавшись утилитой IBConsole, получим структуру таблицы Avtor (рис. 10.13).

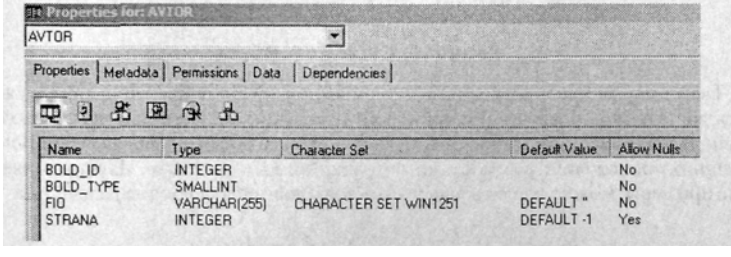


Рис. 10.13. Структура таблицы Avtor

Рассмотрим присутствующие в данной таблице поля. Поле BOLDJD — это первичный автоинкрементный ключ таблицы. Поле BOLD_TYPE определяет тип записи, используя для отображения типов описанную ранее системную таблицу BOLD_TYPE.

ВНИМАНИЕ

В рассматриваемую модель специально не вводились классы-наследники и отношения наследования с целью упрощения изложения материала. Для таких простых случаев поле BOLD_TYPE будет всегда содержать одинаковые значения в рамках одной таблицы. Для более сложных ситуаций, когда данная таблица отражает класс-родитель (суперкласс модели), это поле будет содержать различные значения, которые будут совпадать с типами потомков отображаемого суперкласса.

Поле FIO — обычный строковый атрибут с заданной по умолчанию пустой строкой. Поле STRANA в данном случае является ключом, указывающим на записи одноименной мастер-таблицы STRANA, поскольку классы модели Страна и Автор связаны отношением «один-ко-многим».

Далее, рассмотрим еще одну таблицу LINKNAPISALNAPISANA (рис. 10.14).

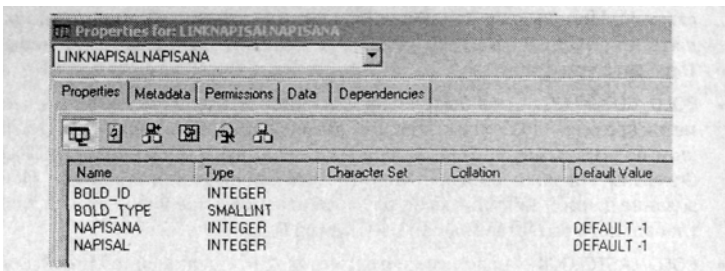


Рис. 10.14. Структура связующей таблицы

Ее структура также достаточно проста. Данная таблица является связующей и предназначена для хранения информации о связи типа «многие-ко-многим», которая отражает наличие ассоциации в UML-модели между классами Автор и Книга (рис. 10.15). Очевидно, что названия полей (NAPISANA и NAPISAL), используемые для хранения связей, являются именами ролей этой ассоциации.

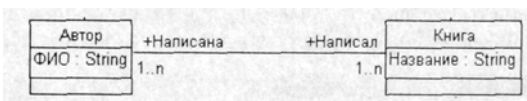


Рис. 10.15. Фрагмент UML-модели

Таким образом, структура генерируемых средой Bold for Delphi таблиц, является достаточно прозрачной и понятной. Возможны ситуации, когда необходимо воспользоваться базой данных, схема которой была сгенерирована средой Bold, из внешних приложений, разработанных традиционным способом. Изучив приведенные примеры, можно сделать вывод, что подобная задача вполне разрешима.

Использование адаптеров СУБД

В рассмотренном выше примере мы использовали компонент BoldDataBaseAdapterIB, обеспечивающий работу с СУБД Interbase. В арсенале Bold for Delphi имеется набор подобных компонентов, предназначенных для подключения к другим типам баз данных (BoldDataBaseAdpapterBDE, BoldDatabaseAdapterADO и т. д.). Все эти компоненты находятся на вкладке BoldPersistence палитры компонентов Delphi. Их используют подобно рассмотренному адаптеру СУБД Interbase, при этом необходимо только заменить компонент IBDatabase на соответствующий аналог. Например, при использовании СУБД Microsoft SQL Server логично использовать компонент TDataBaseAdapterAdo совместно с ADOConnection. Все адаптеры СУБД имеют настра-

ираемый интерфейс диалекта SQL (рис. 10.16), определяемый свойством SQLDatabaseConfig. Данное свойство является комплексным, и содержит множество конкретных подсвойств-настроек, позволяющих адаптировать компонент-адаптер к конкретной СУБД или ее клону. Например, некоторые СУБД имеют ограничения на длину идентификатора строкового поля, для задания такого ограничения можно воспользоваться свойством DefaultStringLength. Использование свойства UseSQL92Joins в ряде случаев позволяет повысить производительность работы с СУБД (естественно, если данная СУБД поддерживает такие запросы-соединения). Эти запросы имеют следующий синтаксис:

```
SELECT <Имя_поля> FROM <Имя_таблицы> <Псевдоним_таблицы>
LEFT JOIN <Имя_таблицы> <Псевдоним_таблицы> ON <Первичный_ключ>
WHERE <Условие_выборки>
```

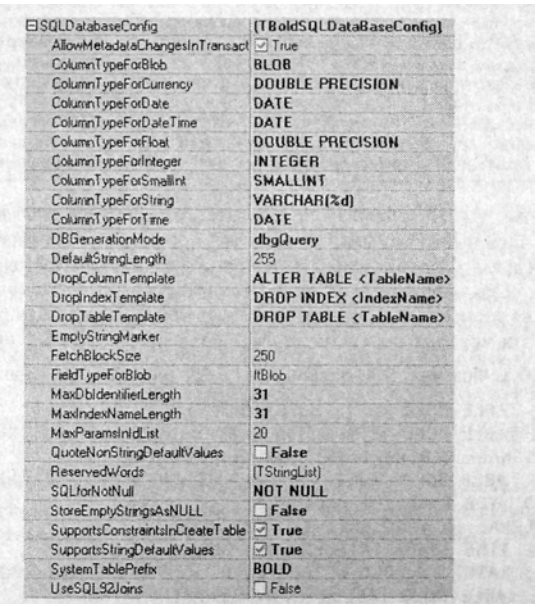


Рис. 10.16. Настройка диалекта SQL

Смысл остальных настроек вполне ясен из их названий, и подробно на них мы останавливаться не будем.

Создание собственных адаптеров СУБД

Среда Bold for Delphi содержит широкий спектр интерфейсов взаимодействия с СУБД (BDE, ADO, DBExpress, IBExpress и т. п.), что позволяет работать с подавляющим большинством существующих на данный момент SQL-серверов и локальных СУБД. Тем не менее, может возникнуть потребность обеспечить работу при-

ложения с некоторой «уникальной» базой данных, не поддерживаемой указанными интерфейсами. Такую задачу можно решить, создав для работы с базой собственный компонент-адаптер. Однако перед этим полезно познакомиться с требованиями и ограничениями, которые «предъявляет» среда Bold for Delphi к системам управления базами данных.

Требования к СУБД

Не всякая СУБД может быть использована совместно с Bold for Delphi. Для такого взаимодействия необходимо выполнение трех основных условий. Во-первых, СУБД должна быть реляционной.

ПРИМЕЧАНИЕ

Разработчики Bold for Delphi, вероятно, поступили абсолютно правильно, разработав механизм объектно-реляционного отображения для преобразования по сути объектно-ориентированной структуры (UML-модели) диаграммы классов в «чужеродную» реляционную модель данных, хотя им пришлось преодолеть немало трудностей на этом пути. Скорее всего, еще в течение длительного времени реляционные СУБД будут доминировать над объектно-ориентированными базами данных (ООБД) по распространенности и доступности. Количество отработанных промышленных ООБД в настоящее время очень невелико и ограничивается буквально единицами наименований. Хотя, безусловно, при использовании ООБД объем и сложность (а значит, и стоимость) такого программного продукта, как Bold for Delphi, кардинально бы уменьшились.

Во-вторых, СУБД должна поддерживать язык определения данных DDL (Data Definition Language) для манипулирования объектами базы данных (создания, изменения и т. д.). Дело в том, что генерацию схемы БД Bold всегда производит с помощью создаваемого им специального SQL-скрипта, использующего DDL-операторы. Для иллюстрации приведем небольшой фрагмент (листинг 10.1) такого SQL-скрипта, который был сгенерирован средой Bold в нашем примере.

Листинг 10.1. Фрагмент SQL-скрипта для генерации схемы базы данных

```
CREATE TABLE BOLD_ID ( BOLD_ID INTEGER NOT NULL)
CREATE TABLE BOLD_TYPE ( BOLD_TYPE SMALLINT NOT NULL,
  CLASSNAME VARCHAR(255) NOT NULL)
CREATE TABLE BOLD_XFILES ( BOLD_ID INTEGER NOT NULL,
  BOLD_TYPE SMALLINT NOT NULL,
  EXTERNAL_ID VARCHAR(255) NOT NULL,
  BOLD_TIME_STAMP INTEGER NOT NULL,
  CONSTRAINT IX_BOLD_XFILES_BOLD_ID PRIMARY KEY (BOLD_ID))
CREATE TABLE BOLD_TABLES ( TABLENAME VARCHAR(255) NOT NULL)
CREATE TABLE BOLD_TIMESTAMP ( BOLD_TIME_STAMP INTEGER NOT NULL)
CREATE TABLE BOLD_LASTCLOCK ( LastTimestamp INTEGER NOT NULL,
  LastClockTime DATE NOT NULL)
CREATE TABLE BOLD_CLOCKLOG ( LastTimestamp INTEGER NOT NULL,
  ThisTimestamp INTEGER NOT NULL,
  LastClockTime DATE NOT NULL,
  ThisClockTime DATE NOT NULL)
CREATE TABLE BOLD_MEMBERMAPPING ( CLASSNAME VARCHAR(60) NOT NULL,
  MEMBERNAME VARCHAR(60) NOT NULL.
  TABLENAME VARCHAR(60) NOT NULL,
```

```
    COLUMNS VARCHAR(60)  NOT NULL,
    MAPPERNAME VARCHAR(60)  NOT NULL)
CREATE TABLE BOLD_R_CLSMAP (  CLASSNAME VARCHAR(60)  NOT NULL,
    TABLENAME VARCHAR(60)  NOT NULL,
```

Получить текст подобного скрипта можно с помощью оператора, приведенного в листинге 10.2.

Листинг 10.2. Оператор получения SQL-скрипта для генерации схемы БД

```
datamodule1.BoldPersistenceHandleDB1.
PersistenceControllerDefault.PersistenceMapper.GenerateDatabaseScript(
memo1.Lines,'');
```

В данном листинге текст скрипта помещается в строковый массив компонента Memo1 типа TMemo, хотя для этой цели можно использовать любой класс типа TStrings.

И, в-третьих, СУБД должна поддерживать транзакции и управление ими (команды StartTransaction, CommitTransaction, RollbackTransaction), так как исполняющая система Bold должна обеспечивать контроль и управление процессом внесения изменений в уровень данных.

Состав программных модулей адаптера СУБД

Любой компонент-адаптер СУБД, используемый средой Bold, состоит из следующих основных модулей (на примере адаптера ADO):

- BoldDatabaseAdapterAdo.pas — обеспечивает соединение с компонентом, представляющим СУБД (в данном случае — с ADOConnection). Формирует внутреннюю переменную Bold, описывающую базу данных — InternalDatabase;
- BoldPersistenceHandleADO.pas — устанавливает соединение с базой данных;
- BoldPersistenceHandleADOReg.pas — регистрирует компоненты в IDE;
- BoldADOInterfaces.pas — реализует все операции с СУБД, то есть формирование запросов, управление транзакциями и т. п.

Исходные тексты перечисленных программных модулей — составных частей адаптеров СУБД поставляются в составе продукта Bold for Delphi. Воспользовавшись ими как примерами, разработчик имеет возможность создать собственный адаптер СУБД.

Преимущества использования адаптеров

Описанная методика подключения к СУБД и возможность генерации схемы БД обеспечивают качественное преимущество при создании приложении баз данных. Если после разработки приложения возникает необходимость его «перевода», например, с локальной СУБД MS Access на клиент-серверную СУБД Oracle, то при традиционной разработке придется, скорее всего, создавать базу данных в Oracle практически заново. В случае использования Bold for Delphi достаточно подключить другой адаптер СУБД и автоматически сгенерировать структуру новой базы данных.

Использование BoldActions для работы с БД

В состав Borland MDA входят достаточно удобные средства автоматизации создания баз данных и подключения к уровню данных на этапе выполнения приложения — **BoldActions.**

ПРИМЕЧАНИЕ

В рассматриваемой и существующих в настоящее время версиях продукта Bold for Delphi использование этих средств возможно только при работе с СУБД Interbase.

Использование этих инструментов осуществляется посредством стандартного Delphi-компонента ActionList. Поместим его на нашу форму, вызовем правой кнопкой мыши контекстное меню и выберем пункт NewStandartAction. В открывшемся окне (рис. 10.17) выберем действие TBoldIBDatabaseAction.

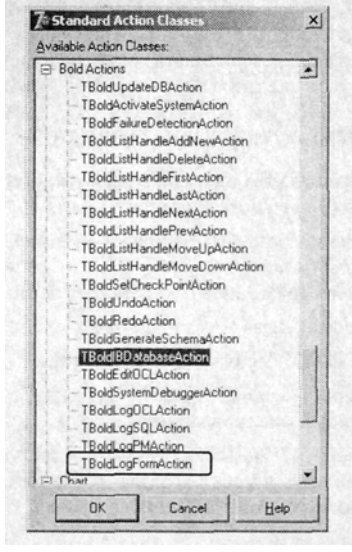


Рис. 10.17. Выбор действия из списка

Это действие предназначено для генерации схемы базы данных СУБД Interbase. В инспекторе объектов можно настроить его параметры, указав свойства генерируемой базы данных (рис. 10.18).

Кроме этого, добавим еще два действия — TBoldActivateSystemAction (активизация объектного пространства) и TBoldUpdateDBAction (обновление базы данных). Поместим на форму три кнопки (рис. 10.19), каждую из которых свяжем с одним из указанных действий (рис. 10.20).

Запустив приложение, убедимся, что при нажатии кнопки:

- Создать БД — генерируется новая база данных;

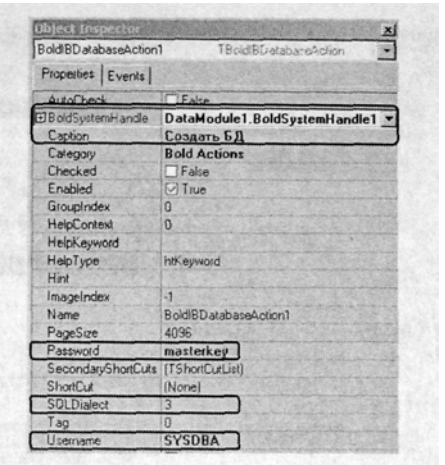


Рис. 10.18. Настройка свойств действия

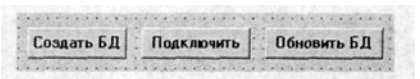


Рис. **10.19.** Кнопки для инициализации действий

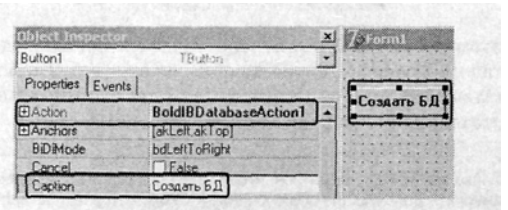


Рис. 10.20. Привязка кнопки к действию

- Подключить — активизируется объектное пространство (отображается информация из БД и обеспечивается возможность ее редактирования);
- Обновить БД — отредактированная или добавленная пользователем информация запоминается в базе данных (рис. 10.21).

Отметим, что при использовании BoldActions нет необходимости заранее создавать пустую базу данных, как мы это делали в начале главы. Единственное, что необходимо сделать, — это задать имя файла (пусть даже не существующего) базы данных в свойстве DatabaseName компонента IBDatabase.

ПРИМЕЧАНИЕ

Естественно, если файла базы данных физически не существует, то бессмысленно и производить операции по проверке соединения с этой БД, как это делалось нами ранее.

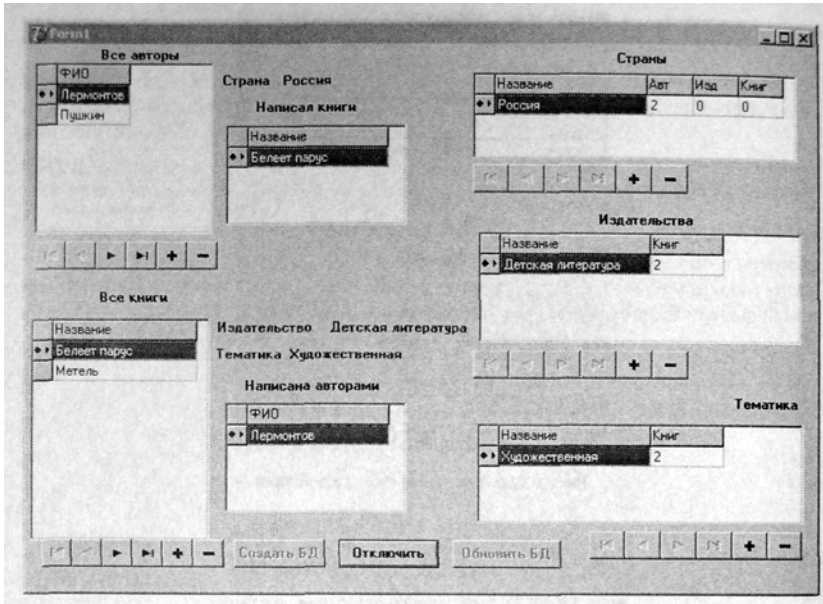


Рис. 10.21. Вид приложения в работе

Если при этом путь к файлу не будет задан, база данных будет сгенерирована в текущем каталоге. Такие возможности объясняются тем, что Bold при генерации базы данных взаимодействует непосредственно с программным интерфейсом (API) InterBase для создания файла БД, то есть «в обход сервера».

ПРИМЕЧАНИЕ

На рассмотренных возможностях компонентов BoldActions основаны все демонстрационные приложения-примеры, использующие подключение к СУБД, которые поставляются в составе продукта Bold for Delphi.

Язык SQL в MDA-приложениях

Идеология применения SQL

Концепция MDA-архитектуры вообще и ее реализация в рассматриваемом продукте Bold for Delphi предполагают, по крайней мере с точки зрения идеологии, что все действия с объектами должны реализовываться на бизнес-уровне. Такие развитые возможности, как OCL-запросы, навигация по модели, «умные» компоненты графического интерфейса и т. п., предоставляют в распоряжение разработчика мощные и гибкие средства управления объектами, как на этапе проектирования приложения, так и непосредственно на этапе его исполнения. При этом уровень

данных (Persistence Layer) с точки зрения объектного бизнес-уровня является вспомогательным и расположен «ниже». Поэтому, теоретически, уровень данных предназначен только для хранения данных между сеансами работы MDA-приложения.

Подобная привлекательная схема существует в теории, однако на практике мы получаем другую ситуацию. При создании «больших» и сложных приложений оказывается невозможным обойтись без использования инструментов, традиционно относимых не к бизнес-уровню, а к уровню данных (СУБД). Причин такой ситуации несколько, но, по-видимому, главной является **использование двух принципиально различных моделей — объектно-ориентированной модели элементов бизнес-уровня и реляционной структуры данных СУБД.** Остановимся на этом моменте чуть подробнее. Реляционная модель данных разработана давно, и за те несколько десятилетий, которые прошли с момента создания первых реляционных СУБД, их инструментарий был полностью отработан, существенным образом усовершенствован и оптимизирован. Тысячи специалистов в десятках компаний годами разрабатывали способы повышения производительности SQL-запросов, и в настоящее время этому инструменту (языку SQL) нет равных по эффективности доступа к реляционным данным. Для оптимизации выборок данных реляционные СУБД создают специальные элементы структуры, такие как индексы и индексные таблицы. Причем MDA-приложение «ничего не знает» об этих сущностях и «не умеет» ими пользоваться.

С другой стороны, методы доступа к объектно-ориентированным данным, по крайней мере в настоящее время, не настолько глубоко и тщательно проработаны и оптимизированы. Пока еще не существует формализованных общепринятых стандартов построения объектно-ориентированных баз данных, правил и методов доступа к объектам и атрибутам. Поэтому на практике нередки ситуации, когда внешне «объектно-ориентированная» БД базируется на реляционном ядре, используя его возможности для эффективной обработки информации.

Из сказанного вполне очевидно, что при использовании реляционных СУБД в качестве уровня данных MDA-приложения «вынуждены» использовать такое эффективное и отлаженное средство, как язык SQL. В противном случае резко бы упала производительность при работе с данными, особенно при больших объемах их. С другой стороны, если рассматривать клиент-серверную архитектуру приложений баз данных, то мы обнаружим еще одну причину для использования языка SQL. В самом деле, если мы представим клиентское MDA-приложение, работающее с серверной базой данных большого объема, то на практике обязательно возникнет следующая проблема — как и где осуществлять выборки из базы данных. Если это делать в MDA-клиенте, то получится, что необходимо сперва получить на клиентской стороне копию базы данных, и только потом обрабатывать эти данные с помощью OCL-запросов. Такое требование возникает в связи с тем, что **объектное пространство всегда располагается в оперативной памяти.** Описанная схема функционирования явно противоречит клиент-серверному подходу, это во-первых, а во-вторых, она абсолютно бессмысленна с точки зрения требуемых ресурсов клиентской рабочей станции. Отсюда вывод: в клиент-серверной архитектуре целесообразно и практически необходимо воспользоваться возможностями SQL-сервера для обработки данных, а следовательно, MDA-клиент должен иметь возможность формирования необходимых SQL-запросов.

ПРИМЕЧАНИЕ

Стоит подчеркнуть тот факт, что уровень данных (Persistence Layer) в идеологии Bold for Delphi включает в себя не только сами данные или базы данных, но и инструменты управления этими данными, то есть собственно SQL-серверы (СУБД).

Однако необходимо четко понимать, что после получения результата такого SQL-запроса среда Bold for Delphi обеспечивает «интеграцию» полученных данных в объектное пространство (для чего существуют специальные компоненты и механизмы, рассматриваемые ниже в этой главе). И после такой интеграции полученные «напрямую» данные могут использоваться приложением так же, как и все другие элементы объектного пространства.

Отметим, что во многих случаях (если объем данных в БД не очень велик или в случае применения локальных БД) можно при создании MDA-приложений баз данных вполне обойтись и без использования языка SQL. В каждом конкретном случае разработчик может самостоятельно определить необходимость применения такого «низкоуровневого» (с точки зрения объектной архитектуры бизнес-уровня) инструмента, как язык SQL. Далее в этом разделе мы рассмотрим основные инструменты и возможности среды Bold for Delphi для работы с SQL.

Использование дескриптора BoldSQLHandle

В главе 7, при обзоре дескрипторов объектного пространства, мы не остановились на компоненте BoldSQLHandle, отложив его рассмотрение до данной главы. В этом разделе мы изучим возможности этого компонента. Компонент BoldSQLHandle находится на вкладке BoldHandles палитры компонентов Delphi. Его назначение - прямое обращение к СУ БД для быстрого получения выборок данных посредством SQL-запросов. Для иллюстрации использования данного компонента создадим на базе рассмотренного ранее примера простой проект, при этом в UML-модели оставим только один класс Avtor. Также для удобства будем использовать уже рассмотренные компоненты BoldActions и связанные с ними кнопки. Добавим на форму компонент BoldSQLHandle, свойства которого настроим следующим образом (рис. 10.22):

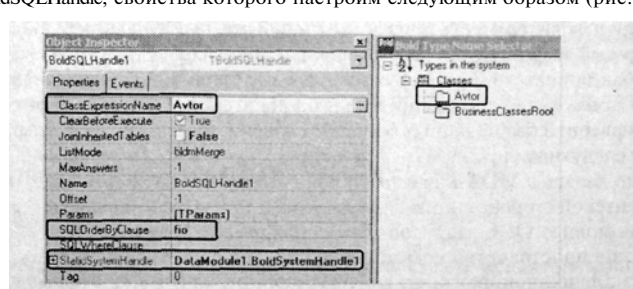


Рис. 10.22. Настройка компонента BoldSQLHandle

Свойству ClassExpressionName присвоим значение класса Avtor (это свойство можно задать вручную или после двойного щелчка по свойству выбрать нужный класс из списка доступных классов модели). Данное свойство опреде-

ляет элемент модели, таблица-аналог которого будет являться источником информации SQL-запроса.

- Свойству SQLOrderByClause присвоим значение fio. Данное свойство определяет условие Order By для SQL-запроса. В нашем случае мы собираемся упорядочивать авторов по фамилии.
- Свойству StaticSystemHandle присвоим значение системного дескриптора DataModule1.BoldSystemHandle1. Данное свойство «подключает» дескриптор SQL к объектному пространству, используемому по умолчанию. Если это свойство не задано, попытка активизировать SQL-запрос приведет к программному исключению.

Для того чтобы сравнить поведение обычного дескриптора списка ОП и дескриптора SQL, поместим на форму второй компонент BoldGrid и второй дескриптор списка. Необходимо сказать, что сам компонент BoldSQLHandle не способен возвращать наборы данных, по этой причине возвращаемые им результаты невозможно непосредственно отобразить, например в BoldGrid, и необходим «компонент-переходник», в качестве которого могут использоваться дескрипторы списка или дескрипторы курсора (см. главу 7). Мы для этой цели используем второй дескриптор списка, в качестве корневого дескриптора которого (свойство RootHandle) зададим SQL-дескриптор. Вторую сетку BoldGrid свяжем с добавленным вторым дескриптором списка. После создания столбцов по умолчанию эта сетка отобразит те же столбцы, что и первая (рис. 10.23).

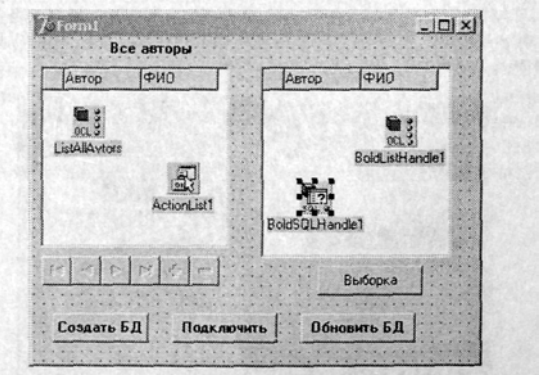


Рис. **10.23**.Видформы с двумя сетками

Таким образом, первая сетка BoldGrid будет отображать данные стандартным для Bold способом (то есть путем получения данных из ОП), а вторая сетка, подключенная через цепочку дескрипторов «дескриптор списка — дескриптор SQL» — будет получать данные непосредственно из уровня данных (СУБД). Для активизации SQL-запроса компонента BoldSQLHandle необходимо вызвать его метод ExecuteSQL. Подключим этот вызов к кнопке Выборка, в обработчик нажатия которой введем строку кода:

```
BoldSQLHandle1.ExecuteSQL;
```

Запустим приложение, и после активизации БД нажмем кнопку Выборка. Мы увидим, что в правой сетке отобразились те же авторы, только упорядоченные по фамилиям (рис. 10.24).

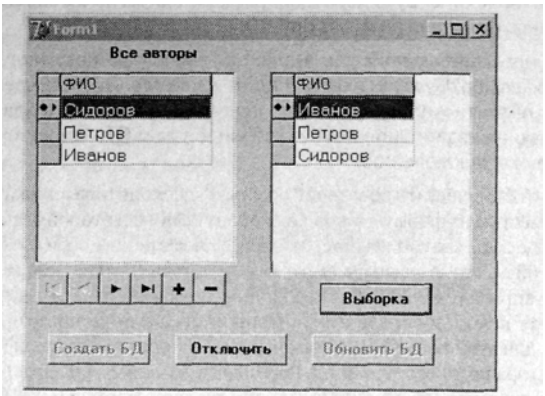


Рис. **10.24.** Приложение в работе

Добавим с помощью Bold-навигатора нового автора Антонов в левую сетку и снова нажмем кнопку Выборка. При этом ничего не произойдет (рис. 10.25), то есть содержимое правой сетки не изменится

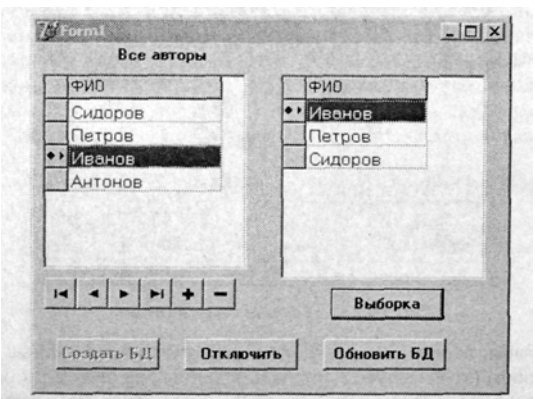


Рис. 10.25. Новые данные не зафиксированы в БД

Объясняется это простым обстоятельством — правая сетка отображает данные непосредственно из БД, а мы после добавления нового автора в ОП не произвели операцию обновления базы данных. После нажатия кнопки Обновить БД и повторного запуска SQL-запроса можно убедиться, что в правой сетке данные обновились, а новый автор Антонов занял верхнюю позицию в списке авторов (рис. 10.26).

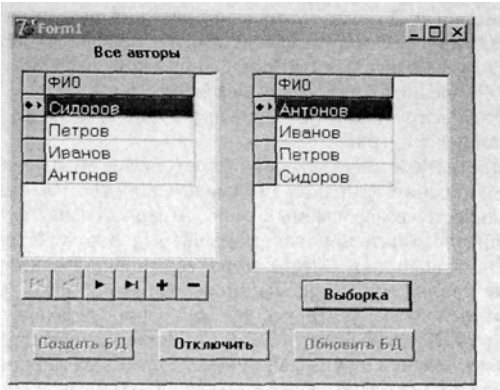


Рис. 10.26. БД синхронизирована с ОП

Из рассмотренного простого примера очевидна разница «штатного» функционирования приложения Bold (работа на уровне объектного пространства) и непосредственной «низкоуровневой» работы с уровнем данных. При взаимодействии с ОП все происходящие в нем изменения автоматически «отслеживаются» графическим интерфейсом приложения. При непосредственной работе с уровнем данных такие изменения необходимо отслеживать разработчику. Нужно подчеркнуть, что рассматриваемый компонент — SQL-дескриптор, как следует из состава его свойств, предназначен для реализации выборок данных из единственной таблицы, которая в БД «отображает» класс, задаваемый свойством ClassExpression. Рассмотрим другие свойства данного компонента. Логическое свойство ClearBeforeExecute задает необходимость предварительной очистки результатов предыдущего запроса

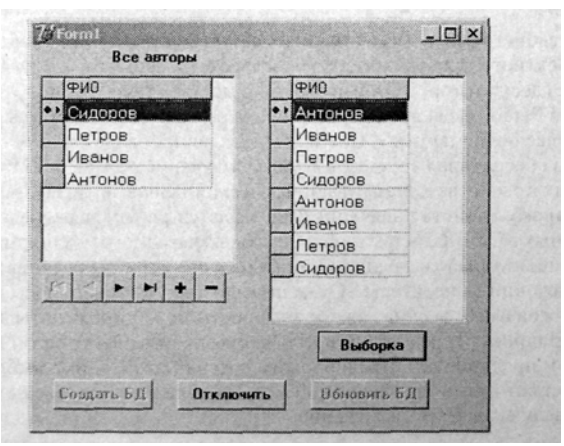


Рис. 10.27. Добавление данных при втором запросе

перед его активизацией. Правила объединения результатов последовательных SQL-запросов при задании рассматриваемому свойству значения False определяются другим свойством компонента — ListMode. Например, если свойству ListMode задать значение Allow (разрешить), а свойству ClearBefore Execute — False, то после выполнения двух последовательных SQL-запросов результаты второго запроса добавятся к результатам первого (рис. 10.27).

Если свойство ListMode имеет значение Merge (объединить), то результаты последовательных запросов будут объединяться. Свойство MaxAnswers позволяет ограничить количество возвращаемых запросом записей (по умолчанию это количество не ограничено, и значение свойства равно –1). Это свойство удобно использовать совместно со свойством Offset, которое задает количество пропускаемых записей, которые не будут возвращены на выходе компонента. Свойство Params позволяет формировать параметрический SQL-запрос, и его использование аналогично обычному DB-компоненту TQuery. Свойство JoinInheritedTables определяет необходимость включения в SQL-запрос унаследованных таблиц, В случае, когда SQL-запрос использует только собственные атрибуты класса, этот параметр необходимо оставить присвоенным по умолчанию (False). И, наконец, последнее свойство SQLWhereClause позволяет добавить в SQL-запрос дополнительное условие выборки, соответствующее SQL-оператору WHERE. Отметим, что BoldSQLHandle возвращает данные, которые можно редактировать. В этом легко убедиться, добавив на форму компонент BoldNavigator, связанный через дескриптор списка с дескриптором SQL (аналогично второй сетке BoldGrid), и запустив приложение. После активизации SQL-запроса данные в правой сетке можно редактировать (изменять, удалять, и после обновления БД сделанные изменения сохранятся), однако добавить нового автора мы не сможем (запрос SELECT с условиями). Рассмотренный компонент может применяться при необходимости обеспечить максимально быстрые выборки данных, особенно при работе в архитектуре «клиент-сервер».

Механизм OCL2SQL

В составе свойств ранее рассмотренных нами (см. главу 7) компонентов-дескрипторов объектного пространства BoldListHandle (дескриптор списка) и BoldExpressionHandle (дескриптор OCL-выражения) присутствует свойство EvaluateInPS — Evaluate In Persistence Layer, то есть, если перевести на русский язык, — «получить результат на уровне данных». По умолчанию данное свойство имеет значение False, тем самым обеспечивая «оценку», то есть получение результата OCL-выраженин, задаваемых в свойстве Expression данных дескрипторов, на бизнес-уровне. Присвоением данному свойству значения True задается режим непосредственной обработки данных в СУБД с использованием так называемого механизма «OCL2SQL». Этот механизм объектно-реляционного отображения обеспечивает трансляцию OCL-выражений в операторы SQL-запросов. Как уже говорилось, использование SQL-запросов имеет весьма важное значение и может применяться во многих ситуациях. Например, представим приложение, работающее с базой данных кадров на крупном предприятии. Предположим, что нам необходимо выбрать сотрудников, возраст которых не превышает 30 лет. Для решения поставленной задачи сформируем следующее OCL-выражение:

```
Sotrudnik.allInstances->select(vozrast<=30)
```

Но можно ли после этого считать, что поставленная задача решена? Только в теории. А на практике, вспомним, что оценка OCL-выражений реализуется по умолчанию на бизнес-уровне, то есть в объектном пространстве — в оперативной памяти рабочей станции. А это означает, что в рассматриваемой ситуации в оперативную память сперва должны быть «загружены» все сотрудники фирмы, и только после этого будет проведена необходимая выборка. Естественно, что для больших объемов данных это недопустимо, ни по скорости (сетевой доступ), ни по сетевому трафику, ни по требуемым ресурсам (оперативная память). Гораздо логичнее в данной ситуации возложить операции по выборке данных на удаленный SQL-сервер СУБД. Однако, такое преобразование-трансляция имеет ряд ограничений, с которыми полезно ознакомиться разработчику. Они могут привести в некоторых случаях даже к необходимости изменения UML-модели. Во всяком случае, наличие таких ограничений целесообразно учитывать при создании новой модели. Причина имеющихся ограничений отображения OCL2SQL проста и обсуждалась ранее — существенное различие объектной и реляционной концепций представления данных. Ниже перечислены конструкции языка OCL, которые поддерживаются механизмом объектно-реляционного отображения и **могут быть транслированы** в SQL-запросы:

- все инструменты OCL-навигации по UML-модели для доступа к ролям и атрибутам, **за исключением вычисляемых атрибутов и ассоциаций**;
- операции для работы с коллекциями: select, reject, allInstances, size, orderBy, minValue, maxValue, average, sum, exists, forall, notEmpty, isEmpty, union;
- логические операторы: =, <, >, <=, >=, o, and, or, not, xor, sqlLike, sqlCaseInsensitiveLike;
- арифметические операторы: +, *, /, -, div, mod;
- операции для работы с типами: oclIsKindOf, oclIsTypeOf, oclAsType;
- оператор isNULL

В следующем списке приведены ограничения OCL2SQL, то есть выражения OCL, которые **не могут быть транслированы** в SQL-запросы:

- операции преобразования типов и получения метаданных модели: TypeName, attributes, associationEnds, superTypes, allSuperTypes, allSubClasses, oclType;
- операции преобразований простых типов: substring, pad, postPad, formatNumeric, formatDateTime, strToDate, strToTime, strToDateTime;
- операции для поддержки «истории» ОП: atTime, allInstancesAtTime, existing;
- операции с коллекциями: count, includesAll, difference, including, excluding, symmetricDifference, asSequence, asBag, asSet, append, prepend, subsequence, at, first, last, orderDescending, sumTime;
- другие конструкции: length, min, max, asString, allLoadedObjects, regExpMatch, inDateRange, inTimeRange, constraints, collect, if, concat.

Несмотря на немалое количество ограничений, механизм OCL2SQL позволяет во многих случаях кардинально повысить производительность функционирования программной системы и, безусловно, должен активно применяться разработ-

чиком. Этот механизм функционирует полностью в автоматическом режиме, незаметно для разработчика. Поэтому применение OCL2SQL не вызывает трудностей — достаточно просто установить свойству EvaluateInPS используемого дескриптора значение True. Заботу обо всех дальнейших действиях системы по формированию SQL-запросов возьмет на себя среда Bold for Delphi.

Оптимизация работы в клиент-серверной архитектуре

Кроме использования ранее описанных компонентов и механизмов, для оптимизации функционирования системы в клиент-серверной конфигурации целесообразно дополнительно учесть следующие рекомендации.

1. Избегать OCL-выражений типа Avtor.allInstances. Исполнение такого OCL-запроса приведет к загрузке всех объектов данного типа в ОП со всеми вытекающими последствиями (ресурсы, производительность, трафик). В данном случае «не поможет» и механизм OCL2SQL, поскольку это OCL-выражение будет отображено в SQL-запрос вида SELECT * FROM Avtor с теми же последствиями. Как и при традиционной разработке клиент-серверных приложений, необходимо выбирать из БД только те данные, которые нужны в данный момент.
2. Для повышения скорости загрузки набора объектов целесообразно использовать метод EnsureObjects класса TBoldObjectList. При этом все объекты будут загружаться за одну сессию работы с БД. В противном случае Bold загружает каждый объект в отдельном сеансе.
3. Использовать «интеллектуальные» возможности компонента TBoldGrid. Этот компонент управляет загрузкой из уровня данных таким образом, чтобы количество объектов соответствовало количеству видимых строк сетки. При вертикальной прокрутке сетка «подгружает» следующую порцию данных из БД.
4. Минимизировать количество активных дескрипторов ОП на момент запуска приложения и подключать их по мере необходимости.
5. Использовать дескриптор BoldUnloaderHandle (см. главу 7) для автоматического освобождения оперативной памяти рабочей станции.

Bold и «тонкие базы данных»

Bold for Delphi — это мощное средство для быстрой разработки сложных приложений баз данных и информационных систем. Однако не следует думать, что этот продукт может быть полезен только при решении больших и «серьезных» задач корпоративного уровня. Использование Bold для создания небольших локальных приложений также весьма эффективно, так как кардинально повышает производительность и автоматизирует решение таких задач, как создание форм для ввода данных. Особенно привлекательно выглядит использование Bold for Delphi совместно с так называемыми «тонкими базами данных» [4]. Вкратце поясним, что «тонкие базы данных» (ТнБД) отличаются от других локальных БД тем, что ядро СУБД встраивается при компиляции проекта непосредственно в исполняемый файл

приложения, поэтому для функционирования таких приложений ничего не требуется — ни BDE, ни ADO, ни дополнительные библиотеки DLL и т. п. Приложения ТнБД легко могут быть запущены безо всякой инсталляции и с любого носителя (Flash, CD-ROM и т. д.), даже без копирования на жесткий диск. Разработчики Bold for Delphi, вероятно, учитывая подобные возможности, включили в состав адаптеров СУБД компонент TBoldDataBaseAdapterDBISAM, который обеспечивает взаимодействие с ТнБД DBISAM [4]. При использовании DBISAM совместно с Bold достаточно сложные приложения, предназначенные для использования в небольших компаниях, можно создавать буквально за считанные дни. Кроме того, такие небольшие «тонкие» приложения можно использовать для отработки и моделирования будущих «сложных» приложений (например, отлаживать дизайн и функциональные возможности графического интерфейса), а также для создания работающих макетов и презентационных вариантов поставки, в целях рекламы и т. д.

ПРИМЕЧАНИЕ

Недавно появился аналогичный по функциональности свободно распространяемый продукт, являющийся клоном СУБД FireBird, который получил название FireBird Embedded (встраиваемый FireBird). Его также можно отнести к «тонким базам данных» со всеми вытекающими преимуществами, рассмотренными выше. Описываемая версия продукта Bold for Delphi полностью совместима с этой СУБД. Для использования FireBird Embedded достаточно подключиться к ней с использованием стандартных компонентов IBExpress (см. начало данной главы). При этом можно задействовать рассмотренные в этой главе компоненты BoldActions.

Работа с несколькими СУБД

В практике разработки приложений нередко возникают ситуации, когда необходимо преобразовать ранее созданное приложение для работы с другой СУБД. При этом, естественно, крайне желательно не потерять уже введенную в базу данных информацию. Разберем на простом примере, как решить эту задачу с использованием возможностей Bold for Delphi. За основу возьмем ранее рассмотренное в этой главе приложение, функционирующее с СУБД Interbase. Предположим, что возникла необходимость перевода всего приложения на СУБД Microsoft Access. Какие действия в этом случае должен совершать разработчик, чтобы не потерять данные? Во-первых, необходимо построить «копию» набора компонентов для работы с другой СУБД. Эта копия будет являться временной, обеспечивая возможность импорта из старой БД ранее введенных данных. Добавим в модуль данных еще один системный дескриптор, дескриптор СУБД и адаптер ADO, подключив последний к компоненту ADOConnection (рис. 10.28). Создадим пустую базу данных MS Access и настроим соответствующие свойства соединения с нею. Далее, на этапе разработки в среде Delphi временно подключим к компоненту BoldModel новый дескриптор уровня данных BoldPersistenceHandleDB2 и сгенерируем схему базы данных MS Access. Таким образом, наше приложение получило принципиальную возможность работать с двумя СУБД. В этом легко убедиться, добавив на форму приложения две сетки и два дескриптора списка, причем эти дескрипторы должны в качестве корневых использовать различные системные дескрипторы bsh1 и bsh2.

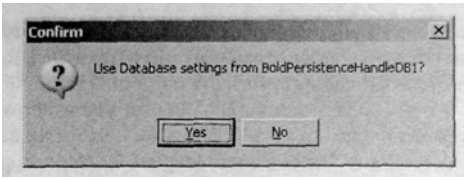


Рис. 10.28. Диаграмма компонентов приложения для двух СУБД

Осталось обеспечить перенос информации. Это можно сделать несколькими способами; выберем из них наиболее наглядный (листинг 10.3).

Листинг 10.3. Перенос информации из одной БД в другую

```
procedure TForm1.Button6Click(Sender: TObject);
var i,j :word;
   ob1,ob2:TBoldObject;
begin
   for i:=0 to
dm.bs1.System.ClassByExpressionName['Avtor'].Count-1
   do begin
ob1:=dm.bs1.System.ClassByExpressionName['Avtor'].BoldObjects[i];
ob2:=TBoldObject.Create(dm.bs2.System);
for j:=0 to ob1.BoldMemberCount-1 do
ob2.BoldMembers[j].Assign(ob1.BoldMembers[j]);
dm.bs2.System.ClassByExpressionName['Avtor'].Add(ob2);
end;
dm.bs2.UpdateDatabase;
end;
```

Смысл программных действий довольно прост и нагляден — используется пообъектное копирование между разными ОП (старым и новым) и добавление объекта в список, соответствующий новой базе данных. После этого достаточно вызвать метод UpdateDatabase, и вся информация будет зафиксирована на уровне данных новой СУБД. Теперь можно убрать из приложения компоненты, отвечающие за «старую» БД InterBase, и продолжать работать уже с MS Access. На основе этого подхода достаточно легко сделать простые процедуры, автоматизирующие перенос данных, и для более сложных приложений. Таким образом, главным преимуществом программной системы Bold for Delphi при реализации подобных операций является то, что работа ведется на бизнес-уровне (уровне объектного пространства), и при создании процедур переноса информации (см. листинг 10.3) мы опять можем обойтись без знания конкретной структуры базы данных, поскольку задачу преобразования структуры ОП в реляционную структуру данных БД берет на себя Bold.

Использование XML-документов

Сохранение данных в XML-файлах

В этой книге мы уже не раз использовали компонент TBoldPersistenceHandleFileXML (см. главу 3). Именно этот компонент-дескритор XML-файлов обеспечивает со-

хранение состояния объектного пространства в файле формата XML. Причем, без ограничений на вид информации, то есть мы можем сохранять не только текстовые данные, но и фотографии, документы Word, PDF-документы Adobe Acrobat и т. д.

ПРИМЕЧАНИЕ

Естественно, для перечисленных типов документов необходимо сформировать в UML-модели специальные атрибуты типа BLOB или его подтипов.

Способ хранения данных в файле XML имеет свои преимущества и недостатки. К основным преимуществам можно отнести следующие.

- Простота разработки. Нет необходимости применять посторонние продукты (СУБД), настраивать подключение к БД, генерировать схему БД и т. д.
- Простота использования. Обеспечивается максимальная переносимость приложения, отсутствует необходимость инсталляции и настройки, возможен запуск приложения с любого носителя на любом компьютере.

В качестве недостатков использования XML можно признать:

- низкую эффективность работы при большом объеме данных;
- невозможность использования SQL, невозможность использования механизма OCL2SQL;
- невозможность работы в клиент-серверной конфигурации.

Однако имеющиеся ограничения нисколько не мешают возможному широкому применению XML для разработки небольших офисных или домашних приложений, справочников, презентационных приложений и т. д. В этом смысле перечисленные преимущества практически совпадают с преимуществами использования «тонких баз данных», рассмотренных в предыдущем разделе данной главы.

ПРИМЕЧАНИЕ

Однако следует иметь в виду, что «тонкие базы данных» поддерживают язык SQL и поэтому имеют значительное преимущество по быстродействию.

Практический пример использования XML

Проиллюстрируем практическое использование файлов XML на простом, но несколько неожиданном примере. Пусть необходимо создать программную библиотеку (модуль), набор процедур которой обеспечивал бы сохранение положения и размера форм приложения между сеансами работы. Ограничимся схемой решения, иллюстрирующей основные подходы, однако ее вполне достаточно для самостоятельного полного решения задачи.

В данном случае UML-модель будет состоять из единственного класса, атрибуты которого описывают положение и размеры формы (рис. 10.29).

Назовем класс Form, а атрибуты — аналогично моделируемым свойствам Delphi-формы, то есть Height, Width, Left, Top. Кроме того, для привязки к конкретной форме приложения добавим атрибут Name, то есть имя формы. Создадим для отладки

нашей библиотеки новый простой Bold-проект, содержащий пустую форму и модуль данных. Добавим на него необходимые компоненты (системный дескриптор, модель, дескриптор типов и дескриптор XML-файлов) и настроим их, как это делалось при создании простого приложения (см. главу 3). Импортируем созданную модель, содержащую единственный класс, из Rational Rose в Bold for Delphi.

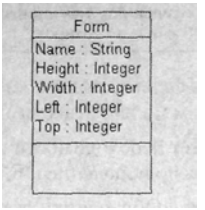


Рис. 10.29. Класс, обеспечивающий сохранение состояния формы

ПРИМЕЧАНИЕ

В данном случае проще создать такую модель во встроенном редакторе Bold, поэтому этот шаг приведен только ради полноты схемы решения задачи

Теперь приступим к созданию процедур. Их достаточно иметь всего две — одна процедура будет обеспечивать сохранение свойств формы, а другая — загрузку сохраненных параметров из файла XML. Программный код первой процедуры приведен в листинге 10.4.

Листинг 10.4. Процедура сохранения состояния формы

```
Procedure SaveFormState (F:TForm);
var O:TBoldObject;
begin
  O:=ObjectLocate('Form','name',f.name);
  if O<>nil
  then EditObject(O,['name','left','top','height','width'],
            [f.name,f.left,f.top,f.height,f.width])
  else NewObject('Form',['name','left','top','height','width'],
            [f.Name,f.left,f.top,f.height,f.width]);
end;
```

ПРИМЕЧАНИЕ

Приводимые в этом разделе процедуры используют ряд вспомогательных универсальных функций и процедур для работы с объектным пространством, описание которых и исходные тексты приведены в приложении В данной книги.

Последовательность действий в данной процедуре следующая:

1. Проверяется, существует ли запись о данной форме. Для этого вызывается функция ObjectLocate, возвращающая искомый объект, если он найден, или значение NIL, если таковой объект отсутствует.
2. Если объект-форма найден, то редактируются его атрибуты, в качестве которых задаются текущие параметры формы. Для этого применяется процедура EditObject.

3. Если запись о данной форме заносится впервые, то создается новый объект класса Form, атрибутам которого присваиваются текущие параметры формы.

Ниже (листинг 10.5) приведен программный код второй процедуры, обеспечивающей загрузку состояния формы из XML-документа.

Листинг 10.5. Процедура восстановления состояния формы

```
Procedure LoadFormState (F:TForm);
var O:TBoldObject;
begin
   O:=ObjectLocate('Form','name',f.name);
   if O<>nil then with F do
   begin
        left:=Attr(O,'left');
        top:=Attr(O,'top');
        height:=Attr(O,'height');
        width:=Attr(O,'width');
   end;
end;
```

Последовательность действий в данной процедуре такова.

1) Проверяется, существует ли запись о данной форме. Для этого вызывается функция `ObjectLocate`, возвращающая искомый объект, если он найден, или значение `NIL`, если таковой объект отсутствует.

2) Если объект-форма найден, то текущим параметрам состояния формы присваиваются значения атрибутов найденного объекта. Для этого применяется функция `Attr`, возвращающая значение атрибута объекта по его имени.

Для использования созданных процедур в других приложениях, необходимо выделить их вместе с остальными задействованными компонентами в отдельный программный модуль (Unit) и обеспечить доступ к ним путем выноса объявлений этих процедур в интерфейсную часть полученного модуля. Теперь достаточно подключить этот модуль на этапе разработки к любому приложению, и вызвать созданные процедуры в обработчиках событий (например, событий создания и закрытия формы) для тех форм, состояние которых требуется запоминать между сеансами работы с приложением.

ПРИМЕЧАНИЕ

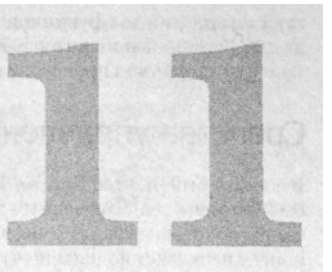
При желании состав сохраняемых параметров формы можно существенно расширить, добавив, например, атрибуты цвета, шрифтов и т. д., а в предельном случае — сохраняя состояние и всех дочерних компонентов формы. В связи с этим рассмотренный пример не является чисто абстрактным. В настоящее время ведутся активные разработки в области использования языка XAML (XML Application Language), который планируется использовать в качестве универсального средства описания графического интерфейса приложений в некоторых будущих операционных системах.

Как видно из приведенного примера, область применения XML-документов в качестве «хранилища» данных весьма широка.

Резюме

В данной главе описана работа с уровнем данных (Persistence Layer). На конкретных примерах продемонстрирован широкий спектр возможностей среды Bold for Delphi в части использования реляционных СУБД. Продемонстрирована «прозрачность» структуры баз данных, генерируемой средой Bold. Показана важность языка SQL при разработке клиент-серверных приложений, и описаны основные компоненты для работы с этим языком. Описан второй способ хранения данных, применяемый Bold, — использование XML-документов, который является в ряде случаев полезным инструментом создания приложений.

Использование сторонних визуальных компонентов



Входящие в состав Bold for Delphi визуальные компоненты для создания графического интерфейса (см. главу 9), предоставляют разработчику широкий спектр возможностей по созданию интерфейса пользователя, обеспечивая при этом автоматическую синхронизацию с объектным пространством. Однако при практической разработке приложений часто возникает необходимость использования «сторонних» компонентов, например для повышения функциональности и удобства интерфейса, в целях улучшения его дизайна или обеспечения единого стилевого оформления.

ПРИМЕЧАНИЕ

К настоящему моменту разработаны десятки пакетов компонентов для Delphi, обеспечивающих формирование весьма выразительных, привлекательных и многофункциональных графических интерфейсов. В качестве примеров можно привести программные продукты компаний DeveloperExpress, LMD, TMS Software и др.

При этом возникает закономерный вопрос — каким образом обеспечить функционирование подобных сторонних компонентов в приложениях, созданных с использованием Bold for Delphi? В самом деле, в отличие от визуальных Bold-компонентов, сторонние компоненты «ничего не знают» ни об объектном пространстве, ни о языке OCL. Эти компоненты не имеют свойства типа Expression и не могут получать информацию от дескрипторов ОП. Поставленные вопросы непосредственно касаются и использования стандартных VCL-компонентов, входящих в поставку Delphi. Используя возможности Bold for Delphi, разработчик может очень быстро создать работающее приложение баз данных, но как он после этого сможет обеспечить формирование красивых и наглядных круговых диаграмм посредством стандартного компонента DBChart? В этой главе будут даны ответы на поставленные вопросы. Сразу скажем, что никаких принципиальных трудностей при использовании сторонних пакетов компонентов в MDA-приложениях не возникает. Разработчики Bold for Delphi, естественно, предусмотрели варианты использования

этой технологии для функционирования в такой «смешанной» конфигурации, когда для формирования интерфейса пользователя необходимо задействовать и компоненты сторонних производителей.

Средства управления внешними компонентами

В состав компонентов Bold for Delphi специально включен компонент TBoldPropertiesController, предназначенный для управления свойствами внешних компонентов. Он расположен на вкладке BoldControls палитры компонентов Delphi. Данный компонент позволяет подключить к себе набор внешних VCL-компонентов и для каждого подключенного компонента выбрать конкретное свойство, которым необходимо управлять. В процессе работы приложения компонент TBoldPropertiesController автоматически устанавливает значения выбранных свойств указанных внешних компонентов, присваивая им результат OCL-выражения (это выражение задается в свойстве Expression рассматриваемого компонента). Проиллюстрируем на примере, как работает данный компонент. Для этого возьмем за основу созданное в предыдущей главе простое приложение, на главной форме которого оставим только сетку BoldGridl с навигатором и кнопки управления компонентами BoldActions (см. главу 10). Добавим на форму стандартный компонент Label2. Затем добавим на форму компонент BoldPropertiesControllerl и настроим его свойства следующим образом (рис. 11.1):

- В качестве дескриптора (свойство BoldHandle) назначим дескриптор списка авторов ListAllAvtors;
- В качестве OCL-выражения (свойство Expression) зададим фамилию автора fio.

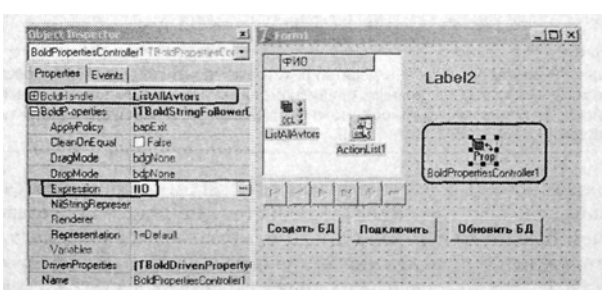


Рис. 11.1. Настройка компонента управления свойствами

После этого дважды щелкнем в инспекторе объектов на свойстве DrivenProperties (или на самом компоненте BoldPropertiesController) и войдем в редактор элементов управления свойствами. Этот редактор устроен стандартным образом. Он имеет кнопки для добавления и удаления элементов. Добавим новый элемент управления (рис. 11.2).

Далее перейдем в редактор свойств и раскроем свойство VCLComponent (рис. 11.3) для привязки внешнего компонента к новому элементу. Мы увидим, что в списке

компонентов присутствуют все компоненты всех форм приложения, что весьма удобно (не требуется ручного ввода). Выберем в качестве «субъекта управления» компонент Label2.

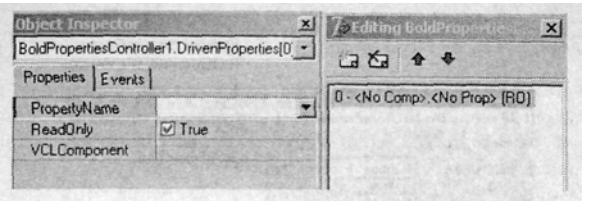


Рис. 11.2. Редактор элементов управления свойствами

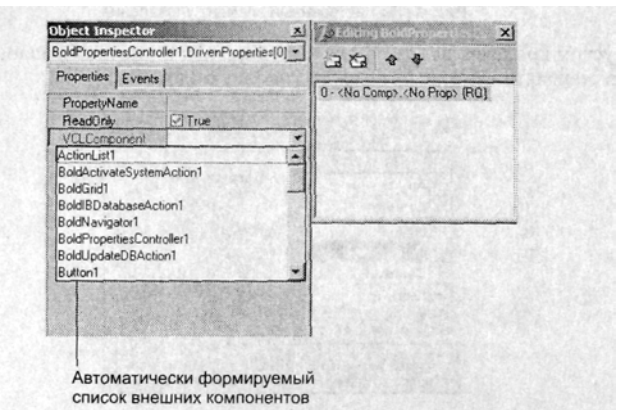


Рис. 11.3. Название рисунка

Теперь назначим конкретное свойство метки Label2, которым должен управлять компонент BoldPropertiessController1. Для этого раскроем свойство PropertyName и из списка возможных свойств выбранного компонента (этот список также формируется автоматически) выберем свойство Caption (рис. 11.4).

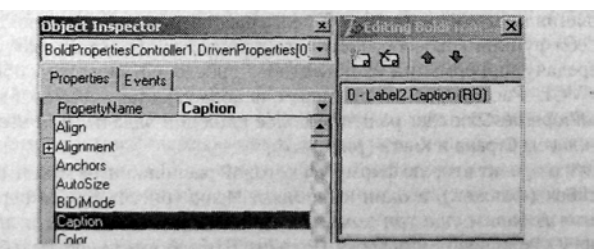


Рис. 11.4. Выбор свойства для управления

После этого настройку единственного элемента управления нашего компонента можно считать законченной (рис. 11.5). Цель проделанных операций — автоматическое отображение фамилии текущего автора в качестве текста стандартной метки.

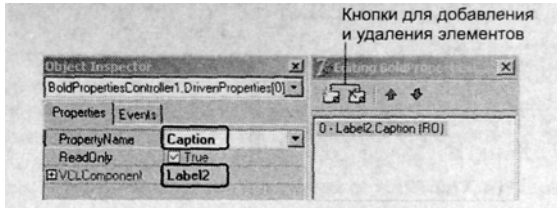


Рис. 11.5. Настроенный элемент управления

Запустим приложение (рис. 11.6) и убедимся, что при перемещении по списку авторов стандартная метка отображает текущую фамилию.

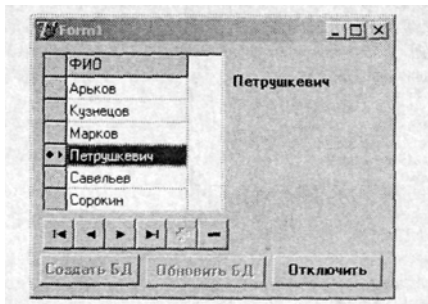


Рис. 11.6. Управление свойством стандартной метки

Мы рассмотрели простейший пример использования компонента BoldPropertiesController. На самом деле его возможности гораздо шире. Используя данный компонент, можно, например, обеспечить автоматическую синхронизацию выбранных визуальных свойств различных компонентов, расположенных на разных формах приложения. Важно, что при этом все подобные задачи могут быть решены без использования программного кода. То есть рассматриваемый компонент, по сути, берет на себя функции своеобразного интеллектуального «адаптера», обеспечивающего передачу информации из объектного пространства в среду обычных компонентов VCL. Рассмотрим на примере с использованием возможностей компонента BoldPropertiesController решение более сложной задачи. Для этого добавим в модель классы Страна и Книга (рис. 11.7).

Добавим в проект вторую форму, на которой расположим 4 метки, один компонент CheckBox (флажок), и один компонент Memo (рис. 11.8). На первую форму приложения добавим еще три компонента BoldPropertiesController и присвоим им имена PropFIO, PropCountry, PropBooks и PropRus. В обработчике события OnShow главной формы напишем оператор `Form2.Show` для отображения второй формы.

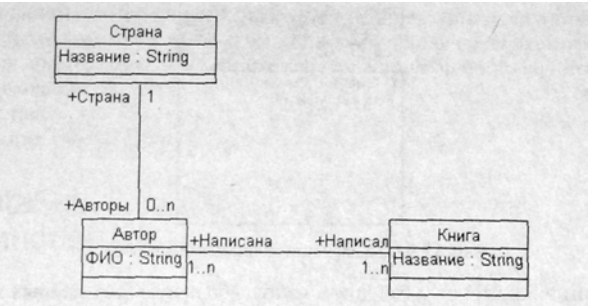


Рис. 11.7. Расширенная модель

Рис. 11.8. Модификация приложения

Поставим следующую задачу — автоматически отображать во второй форме текущую информацию о фамилии автора, количестве написанных им книг, стране проживания. Кроме того, если автор — российский, то на второй форме должен активизироваться флажок, и отображаться поле Memo для ввода дополнительной информации. Для этого уже известным нам способом настроим элементы управления в контроллерах свойств. Сначала все контроллеры подключим к одному и тому же дескриптору списка авторов ListAllAvtors. Для компонента PropFIO оставим настройки, сделанные в предыдущем примере, только немного изменим OCL-выражение — заменим его для удобства на строку 'Фамилия '+fIO. Контроллеру PropCountry зададим OCL-выражение 'Страна '+strana.nazvanie и добавим одно свойство управления меткой с названием Страна. Контролеру PropBooks зададим OCL-выражение 'Написал книг '+napisal->size.asString и подключим его к метке, отображающей количество книг. Для контроллера PropRus введем следующее OCL-выражение strana.nazvanie='Россия', возвращающее логическое значение. Для этого контроллера зададим два элемента управления — управление свойством Checked компонента CheckBox (флажок) и управление свойством Visible компонента Memo1 (рис. 11.9).

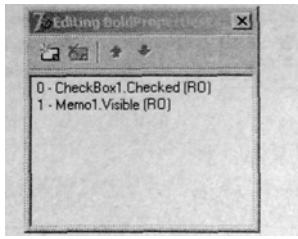


Рис. 11.9. Два элемента в контроллере свойств

Запустим приложение и заполним нашу небольшую базу данных произвольной информацией, используя автоформы для добавления и редактирования данных. Теперь мы можем проверить работоспособность приложения и убедиться, что поставленная задача решена. Приложение корректно отображает информацию как для «российских» авторов (рис. 11.10), активизируя флажок (CheckBox) и отображая поле Memo, так и для «зарубежных» (см. рис. 11.11), когда компонент обеспечивает снятие флажка и скрытие окна редактирования Memo.

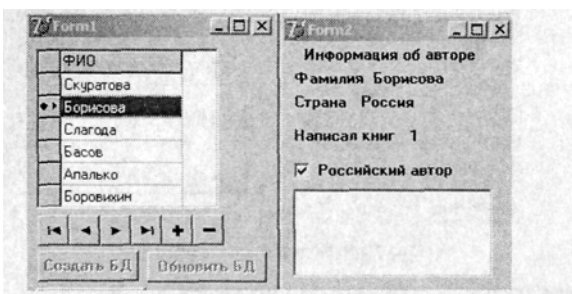


Рис. 11.10. Текущий автор — российский

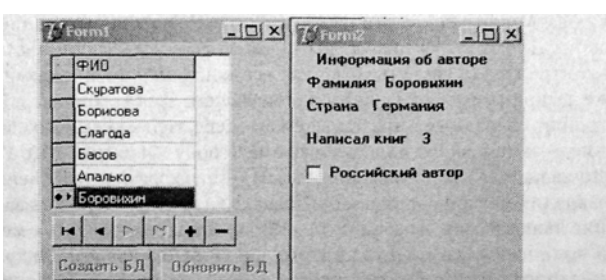


Рис. 11.11. Текущий автор — не российский

Как мы видим, применение рассмотренного компонента-контроллера свойств не вызывает трудностей и позволяет не прибегать к программированию. В рассмотренных примерах мы ограничились использованием стандартных VCL-компонентов, входящих в состав поставки Delphi. Однако компонент BoldProperties-Controller с таким же успехом может применяться и для создания приложений, использующих компоненты сторонних производителей.

BoldDataSet — шлюз для использования DB-компонентов

Еще одним важным компонентом, который обеспечивает взаимодействие со сторонними компонентами, является BoldDataset. Он играет роль своеобразного шлюза к любым внешним визуальным компонентам, предназначенным для работы с базами данных в Delphi, поскольку является наследником известного класса TDataset. Поэтому BoldDataset способен заменить такие компоненты, как TTable, TQuery, TAdoTable и т. д. Компонент BoldDataset находится на вкладке BoldMisc палитры компонентов Delphi. Рассмотрим, как он применяется на практике. Вернемся к одному из предыдущих примеров и внесем некоторые изменения. Уберем с формы BoldGrid, BoldNavigator, а вместо них добавим стандартные компоненты DBGrid и DBNavigator. Кроме того, добавим на форму компонент DataSource и новый компонент BoldDataset. Компонент BoldDataset имеет привычное свойство BoldHandle, которому мы присвоим значение дескриптора списка авторов ListAllAvtors. Чтобы обеспечить активизацию этого компонента по первому требованию, присвоим его свойству AutoOpen значение True. Остальные компоненты свяжем обычным способом (рис. 11.12).

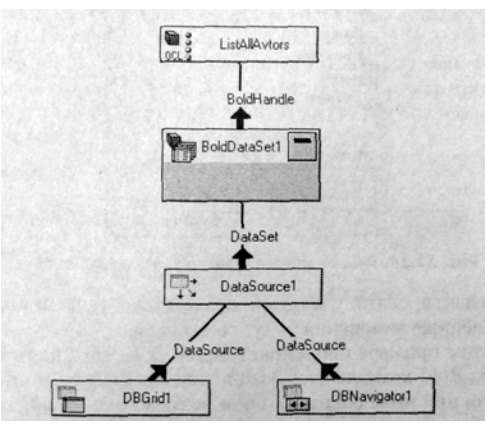


Рис. 11.12. Диаграмма связей компонентов

Теперь настроим поля компонента BoldDataset. После двойного щелчка по нему откроется стандартный редактор полей (рис. 11.13). Добавим новое поле и зададим его свойства в инспекторе объектов. Каждое поле компонента BoldDataset имеет индивидуальное свойство Expression для ввода OCL-выражения. Введем выражение fIO, то есть в данном поле будет содержаться информация о фамилии автора. Дадим новому полю имя (свойство FieldName) Avtor.

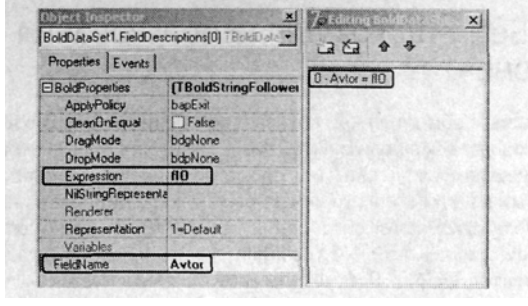


Рис. 11.13. Настройка свойств поля компонента BoldDataSet

После двойного щелчка по сетке DBGrid1 мы уже традиционным образом способом добавим один столбец и свяжем его с полем Avtor. Запустим приложение и убедимся, что оно вполне работоспособно (рис. 11.14).

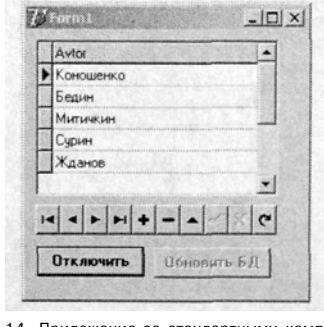


Рис. 11.14. Приложение со стандартными компонентами

Мы можем редактировать, удалять и добавлять авторов, и после обновления базы данных введенные изменения будут сохранены.

На этом простом примере продемонстрирована возможность полного исключения визуальных Bold-компонентов из приложения с сохранением функциональности. Более того, при этом сохраняются и возможности синхронизации графического интерфейса. Для того чтобы в этом убедиться, просто добавим на форму метку BoldLabell и подключим ее к тому же дескриптору списка авторов ListAllHandles (при этом задав OCL-выражение fio для ее свойства Expression). После запуска приложения увидим, что метка отслеживает переходы по списку авторов (рис. 11.15).

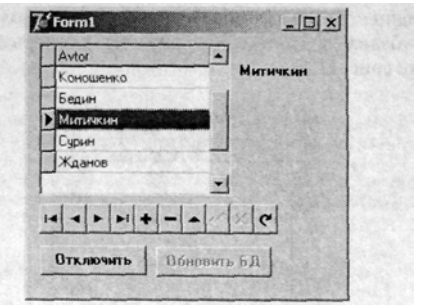


Рис. 11.15. Синхронизация компонентов

ПРИМЕЧАНИЕ

Некоторая потеря функциональности все-таки неизбежна при использовании сторонних (и стандартных) компонентов. Инициация автоформ, поддержка Drag-and-Drop и ряд других возможностей будут потеряны. Впрочем, их можно реализовать и программным образом.

Использование BoldDataset имеет и другие преимущества. Как и для рассматриваемого ранее (см. главу 9) компонента BoldGrid, принципиальным и качественным отличием данного компонента от стандартных аналогов является эффективное использование механизма навигации по модели. Это позволяет, даже используя сторонние визуальные компоненты типа DBGrid, обеспечить отображение в одной сетке практически любой нужной информации, из любых классов модели (естественно, эти классы должны быть связаны ассоциациями). В нашем примере мы отображаем информацию об авторах, хранящуюся в таблице базы данных Avtor. Если бы мы использовали стандартный компонент типа TTable, то такой возможностью все бы и ограничилось. Но поскольку мы используем Bold-компонент BoldDataset в качестве источника информации, постольку можем отобразить любую информацию и из других (связанных) классов, например информацию о стране. Для этого просто добавим новое поле Country и введем в его свойство (рис. 11.16) Expression навигационное OCL-выражение вида

```
strana.nazvanie
```

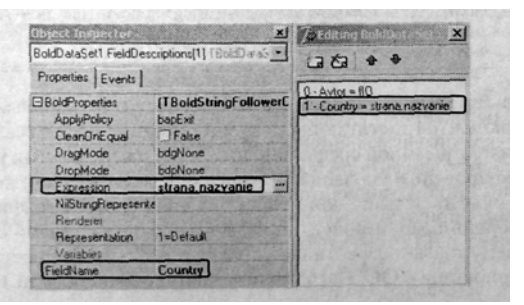


Рис. 11.16. Добавление и настройка второго поля

Добавим еще один столбец в DBGrid и свяжем его с новым полем Country. Запустив приложение, убедимся, что и стандартная сетка теперь отображает данные из нескольких таблиц (рис. 11.17).

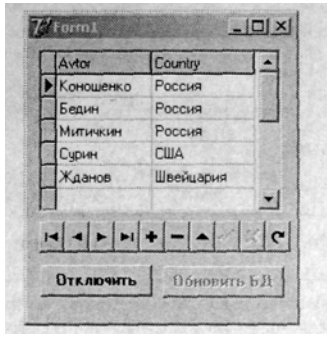


Рис. 11.17. Отображение данных из разных таблиц

ПРИМЕЧАНИЕ

В самом факте отображения в одном компоненте DBGrid данных из нескольких таблиц ничего принципиально нового нет, этого же эффекта можно добиться, например, путем формирования SQL-запроса и подключения этой сетки к соответствующему компоненту TQuery. Другое дело, что в приведенном примере эта задача решается гораздо проще и элегантнее.

Таким образом, можно сделать следующий вывод — использование компонента BoldDataset позволяет применять любые обычные DB-компоненты для построения графического интерфейса приложения, сохраняя при этом значительную часть преимуществ рассматриваемой MDA-технологии. На практике этот компонент очень удобно использовать, например, совместно с визуальным компонентом TDBChart для построения графических диаграмм.

ВНИМАНИЕ

В рассматриваемой версии Bold for Delphi в реализации компонента BoldDataset разработчиками допущена ошибка. Она проявляется, когда свойству AutoOpen устанавлено значение True, и выражается в сбоях при запуске приложения (потери информации о поле). Поэтому рекомендуется на этапе разработки устанавливать значение False для обоих свойств — Active и AutoOpen, и инициализировать BoldDataset программно после запуска приложения путем установки свойству Active значения True.

Заметим, что такие функциональные возможности при использовании описанных ранее компонентов достигаются за счет принципиального разделения составных элементов программной среды Bold for Delphi на три разных уровня: уровень данных, бизнес-уровень и уровень представления (графический интерфейс). При этом каждый уровень решает свои задачи. Заменяя уровень графического интерфейса Bold сторонними компонентами, мы никак «не задеваем» бизнес-уровень, и он по-прежнему так же эффективно управляет объектным пространством, обеспечивает интерпретацию OCL-выражений, навигацию по модели, взаимодействие

с СУБД и т. п. При этом посредством ранее рассмотренных компонентов-«адаптеров» бизнес-уровень также обеспечивает и передачу необходимой информации для функционирования сторонних компонентов.

ПРИМЕЧАНИЕ

Будет ошибкой считать, что использование DBGrid приводит к непосредственному взаимодействию графического интерфейса с таблицей базы данных. На самом деле замена BoldGrid на DBGrid влияет только на «взаимоотношения» графического интерфейса и бизнес-уровня. При этом все механизмы обмена информацией между бизнес-уровнем Bold и уровнем данных (СУБД) такой замены даже «не почувствуют», и будут продолжать функционировать в обычном режиме.

Использование механизма подписки и программного кода

В качестве еще одного способа реализации взаимодействия бизнес-уровня и сторонних визуальных компонентов могут применяться компоненты, использующие механизм подписки (subscribing), встроенный в Bold for Delphi. На самом механизме мы подробнее остановимся несколько позже, а здесь продемонстрируем некоторые возможности такого подхода. Для этой цели рассмотрим компонент BoldPlaceableSubscriber с вкладки палитры компонентов BoldMisc. Вообще говоря, данный компонент предназначен для реализации несколько других функций, касающихся непосредственно механизма подписки, но здесь мы покажем, что его с успехом можно применять и для других целей, а именно для синхронизации графического интерфейса при использовании сторонних компонентов. Поставим следующую задачу — отображать в стандартном компоненте TImage фотографию текущего автора, при этом фотографии должны иметь формат jpg. Для ее решения сперва изменим нашу модель — введем в класс Автор новый атрибут pic типа TypedBLOB, в котором будем хранить фотографии авторов. Эту операцию можно произвести непосредственно во встроенном редакторе моделей (рис. 11.18).

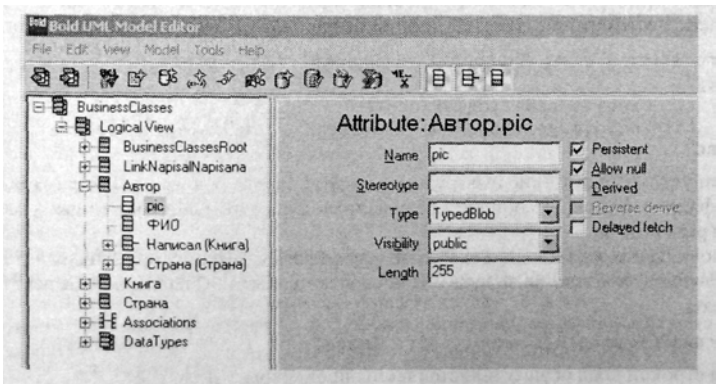


Рис. 11.18. Добавление нового атрибута

Добавим на форму приложения два компонента для отображения изображений — один стандартный компонент Image1 и один Bold-компонент BoldImage1 (он нам пригодится для занесения в базу данных фотографий авторов наиболее простым способом). Добавим на форму компонент BoldPlaceableSubscriber1. Кроме того, добавим стандартный компонент-диалог открытия файлов изображений OpenPictureDialog1 и одну кнопку Button1 с надписью Изменить фото (рис. 11.19).

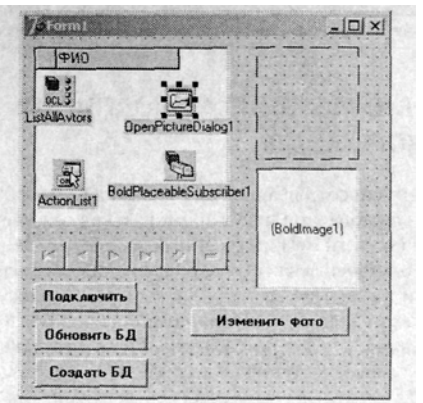


Рис. 11.19. Размещение компонентов на форме

Поскольку мы будем сохранять фотографии авторов в формате JPG, добавим в секцию Uses нашего модуля вызов модулей JPEG и BoldImageJPEG (см. главу 9). Подключим компонент BoldImage1 к дескриптору списка авторов ListAllAvtors, и зададим для его свойства Expression OCL-выражение pic. Для занесения новых фотографий в атрибут pic используем удобный метод класса TBoldImage — LoadFromFile. для чего в обработчике событий кнопки Изменить фото введем следующий код:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
   if OpenPictureDialog1.Execute
   then BoldImage1.LoadFromFile(OpenPictureDialog1.FileName);
end;
```

Запустим приложение и загрузим фотографии авторов из заранее заготовленных файлов, после чего обновим базу данных для сохранения внесенных изменений (рис. 11.20).

Подключим компонент BoldPlaceableSubscriber1 к дескриптору списка авторов ListAllAvtors (свойство BoldHandle). В обработку события OnSubscribeToElement введем код

```
DrawJPGFromBold(element,'pic',Image1);
```

Этой строкой кода осуществляется вызов процедуры

```
DrawJPGFromBold(El:TBoldElement; attr:string;Im:TImage)
```

которая имеет следующие параметры:

El — параметр типа TBoldElement, задающий элемент объектного пространства, из которого необходимо загрузить изображение;

attr — название атрибута, в котором содержится изображение;

Im — имя компонента типа TImage, которому надо присвоить полученное изображение.

Рис. 11.20. Фотографии загружены в БД

Программный код используемой процедуры (листинг 11.1) приведен для иллюстрации методов работы с BLOB-атрибутами. Процедура использует типы TBABlob (BLOB-атрибут Bold), TBoldBlobStream (Bold-поток, обеспечивающий возможность хранения BLOB-атрибутов), а также известный нам класс TBoldObject (см. главу 6). Основные принципы работы процедуры можно понять, анализируя приведенный листинг; необходимые пояснения содержатся в комментариях.

Листинг 11.1, Процедура загрузки JPG-картинки из Bold-атрибута в Image

```
procedure DrawJPGFromBold(El:TBoldElement;
attr:string;Im:TImage);
var bs : TBoldBlobStream;
   sm : TBoldBlobStreamMode;
   Ы : TBABlob;
   jpg : TJpegImage;
begin
if Assigned(El) then // проверка, существует ли элемент ОП
try
   // присвоение переменной Ы значения атрибута
   Ы:=(El as TBoldObject).BoldMemberByExpressionName[attr] as
     TBABlob;
   sm:=bmReadWrite;
   if not (bl.IsNull) then // проверка, не пустая ли картинка
   begin
        bs:=bl.CreateBlobStream(sm); // создание BLOB-потока Bold
```

```
            jpg:=TJpegImage.Create;
            // создание временного экземпляра картинки
            jpg.LoadFromStream(bs); // передача картинки
            im.Picture.Graphic:=jpg;
            // присвоение картинки компоненту
            jpg.Free; // уничтожение временного экземпляра
        end;
    finally
    bs.Free; // уничтожение BLOB-потока
    end; // try
    end: // proc
```

Запустим приложение и убедимся, что поставленная задача решена (рис. 11.21). Теперь при перемещении по сетке BoldGridl стандартный компонент Imagel отображает фотографию текущего автора.

Рассмотренный пример применения компонента BoldPlaceableSubscriber опять демонстрирует нам, что поведение бизнес-уровня приложения не зависит от используемых компонентов графического интерфейса. Достаточно подключиться к этому уровню — и обеспечивается такая же событийная синхронизация элементов графического интерфейса, какая присутствует при использовании Bold-компонентов.

Рис. 11.21. Синхрониация изображений и сетки

Преимущество применения компонента BoldPlaceableSubscriber по сравнению с рассмотренным ранее компонентом BoldPropertiesController (см. начало данной главы) в том, что в обработчике события компонента BoldPlaceableSubscriber разработчик может написать любой программный код, который обеспечит необходимую гибкость и функциональные возможности. Для иллюстрации заменим написанный ранее обработчик следующим кодом:

```
procedure TForm1.BoldPlaceableSubscriber1SubscribeToElement(
   element: TBoldElement; Subscriber: TBoldSubscriber);
var Ob:TBoldObject;
   st:string;
begin
   st:=(element as
     TBoldObject).BoldMemberByExpressionName['fio'].asString;
   if st<>'Лермонтов' then  DrawJPGFromBold(element,'pic',Image1)
   else ShowMessage('Вы выбрали документ, доступ к которому
     ограничен');
end;
```

В приведенном фрагменте кода проверяется фамилия текущего автора. Если выбран автор Лермонтов, выводится простое сообщение (рис. 11.22).

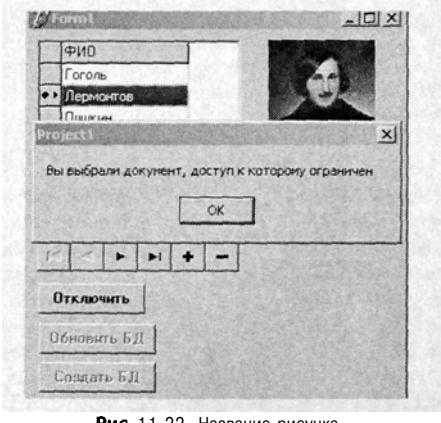


Рис. 11.22. Название рисунка

Подобным программным способом можно легко сформировать любую требуемую логику функционирования графического интерфейса приложения, и в этом проявляется гибкость рассмотренного компонента BoldPlaceableSubscriber.

ПРИМЕЧАНИЕ

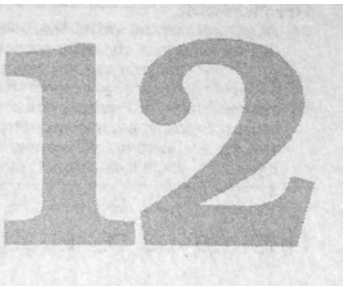
В данном примере вместо BoldGrid с таким же успехом можно использовать и стандартный компонент DBGrid, как было показано ранее в этой главе при описании работы с компонентом BoldDataset.

Резюме

Программная система Bold for Delphi имеет в арсенале своих средств эффективные и многофункциональные компоненты для работы со сторонними (внешними) пакетами компонентов, предназначенными для формирования графического интер-

фейса пользователя. Для автоматического управления свойствами внешних компонентов целесообразно использовать специальный компонент BoldPropertiesController. С целью обеспечения возможности использования внешних компонентов, предназначенных для отображения информации из баз данных, в составе Bold for Delphi присутствует компонент BoldDataset, способный заменить любой стандартный или сторонний компонент-потомок класса TDataset. И, наконец, для реализации наиболее гибкого программного управления поведением внешних компонентов можно воспользоваться компонентом BoldPlaceableSubscriber. Использование внешних компонентов графического интерфейса не влияет на функциональность бизнес-уровня Bold, при этом сохраняются практически все возможности управления приложением.

Генерация кода



При изучении возможностей среды Bold for Delphi, ранее в этой книге рассматривались примеры, не использующие генерацию кода классов модели. Если внимательно посмотреть на создаваемые в этих примерах программные модули Delphi (Units), то обнаруживается, что в них практически не содержится программного кода, за исключением специально создаваемых нами обработчиков событий или процедур. Основная функциональность приложения при таком варианте разработки (без генерации кода классов модели) обеспечивается встроенными программными механизмами Bold for Delphi. Таким образом, в рассмотренных ранее примерах программная система Bold выступала в качестве «интерпретатора» UML-модели (см. главу 1). При этом, обладая информацией об UML-модели на этапе исполнения приложения, среда Bold «на лету» формировала необходимые связи и методы для доступа к необходимой информации, обеспечивала OCL-навигацию и реализацию других полезных функций. Во многих случаях такой подход «интерпретатора» вполне оправдан, и можно создавать достаточно сложные приложения, не используя при этом генерацию классов модели. Получаемые программные модули Delphi будут весьма компактными, достаточно легкими в сопровождении, и основная логика работы не будет «размазана» по различным фрагментам программного кода. Однако весь арсенал возможностей Bold становится доступным только при генерации кода классов. Гибкая работа с механизмами подписки (subscribing), использование операций в классах модели, применение «обратно-вычисляемых» (reverse-derived) атрибутов — эти и другие возможности появляются только с привлечением кодогенератора Bold. Интуитивно ясно, что программирование с использованием программных описаний всех элементов модели, обеспечивает максимальную гибкость при разработке приложений.

Далее в этой главе будут рассмотрены вопросы использования генерации кода при разработке приложений в Bold for Delphi.

ПРИМЕЧАНИЕ

Может показаться, что применение кодогенератора для получения программного кода классов модели несколько сближает рассматриваемый продукт Bold for Delphi с CASE-системами. Однако, как мы увидим в дальнейшем, даже при работе с кодом классов модели все рассмотренные ранее возможности «интерпретатора» сохраняются в полном объеме. Это объясняется тем обстоятельством, что генерируемый программный код представляет собой по сути программную «обертку» вокруг внутренних методов и классов Bold, составляющих основу функционирования этой среды. Поэтому генерируемый код не заменяет собой реализацию функциональности «интерпретатора», а лишь предоставляет удобный программный интерфейс для разработчика.

Процедура генерации

Возьмем за основу рассмотренный в предыдущей главе пример простого приложения и преобразуем его для варианта использования программного кода классов модели. UML-модель приложения состоит из трех классов и двух ассоциаций (рис. 12.1).

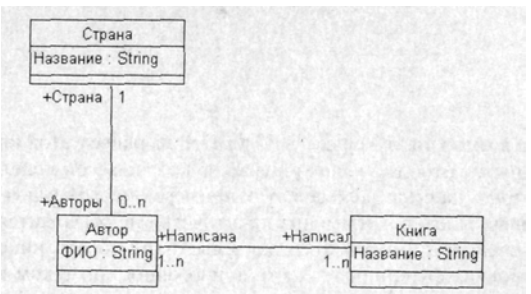


Рис. 12.1. UML-модель

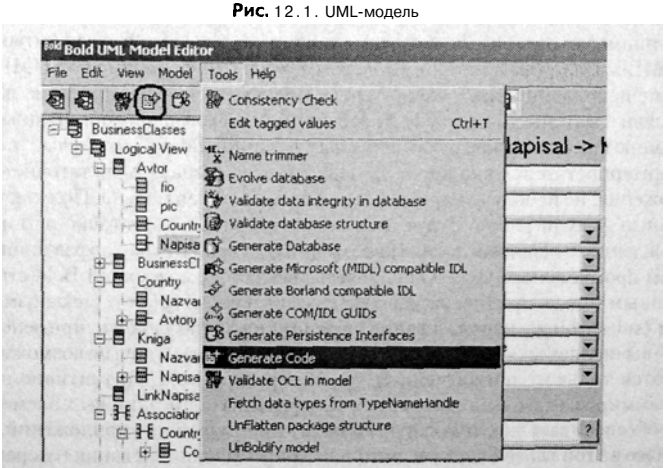


Рис. 12.2. Запуск генерации кода из встроенного редактора

Для генерации кода достаточно вызвать встроенный редактор моделей и выбрать в главном меню Tools • Generate Cod либо нажать кнопку с изображением листа и желтой звездочки на панели инструментов (рис. 12.2).

Появятся последовательно два окна с предложением выбрать имена генерируемых файлов интерфейса (рис. 12.3) и файла программы. Оставим предлагаемые имена файлов без изменений.

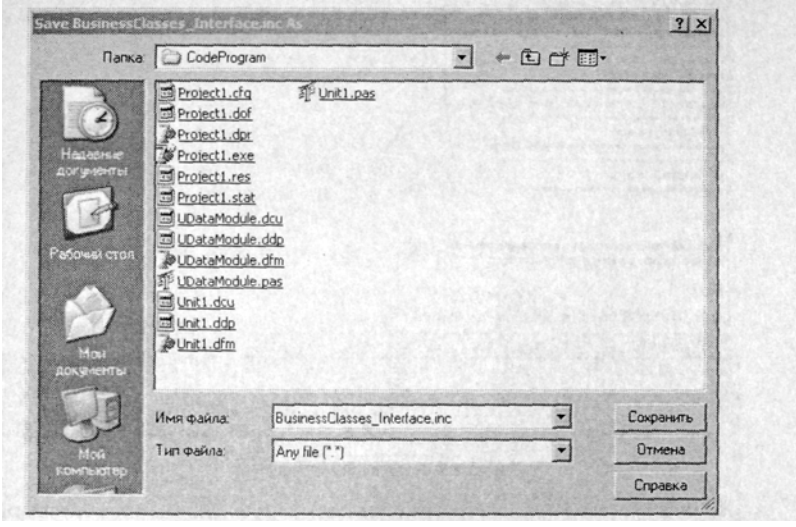


Рис. 12.3. Задание имени генерируемого файла

Собственно генерация кода происходит очень быстро, после чего появится предупреждение в нижней части встроенного редактора (рис. 12.4) о необходимости ознакомиться с протоколом генерации. Для этого нужно нажать на изображение желтого кружка.

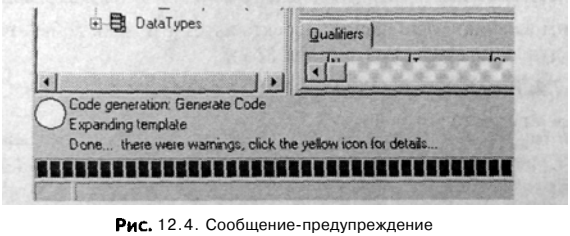


Рис. 12.4. Сообщение-предупреждение

В результате откроется окно протокола генерации кода, содержащее информацию о дате и времени генерации, папке для размещения выходных файлов, а также собственно записи о произведенных действиях (рис. 12.5).

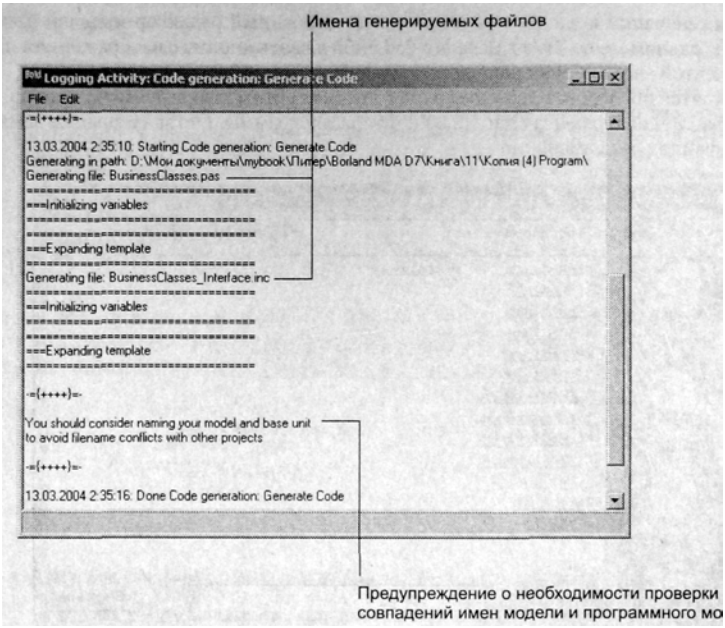


Рис. 12.5. Протокол генерации кода

В конце протокола содержится напоминание о необходимости сравнения имен, используемых в модели, с именем главного программного модуля. Эти имена не должны совпадать во избежание конфликтов. В результате проведенной операции на диске образуются два файла:

- BusinessClasses_Interface.inc — файл описания внешних программных интерфейсов;
- BusinessClasses.pas — файл, содержащий код классов UML-модели.

При этом в секцию Uses нашего проекта автоматически добавится сгенерированный модуль BusinessClasses (листинг 12.1).

Листинг 12.1. Программный код проекта после генерации

```
program Project1;
{%File 'BusinessClasses_Interface.inc'}
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  UDataModule in 'UDataModule.pas' {DataModule1: TDataModule},
  BusinessClasses in 'BusinessClasses.pas';
{$R *.res}
begin
  Application.Initialize;
```

```
    Application.CreateForm(TForm1, Form1);
    Application.CreateForm(TDataModule1, DataModule1);
    Application.Run;
end.
```

Теперь нам осталось указать используемому системному дескриптору типов модели BoldSystemTypeInfoHandle1 на необходимость использования сгенерированных программных модулей для получения информации о типах, для чего его свойству UseGeneratedCod необходимо установить значение True (рис. 12.6).

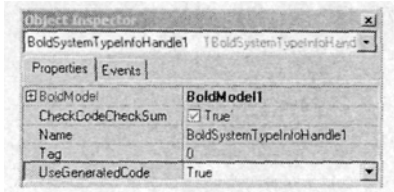


Рис. 12.6. Задание признака использования кода

Сразу отметим, что если установить данный признак до генерации кода, то попытка запуска приложения вызовет программное исключение (рис. 12.7)

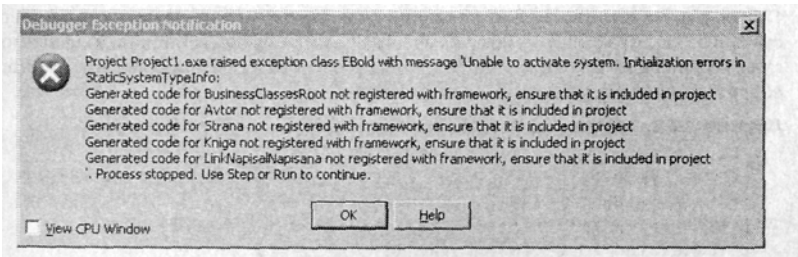


Рис. 12.7. Исключение при недоступности кода модели

Попробуем запустить наше приложение, добавить несколько новых авторов в список — и убедимся, что приложение по-прежнему функционирует корректно (рис. 12.8).

Структура и состав генерируемых модулей

Рассмотрим подробнее программный код классов модели, который сгенерировал Bold. Начнем с файла описания программных интерфейсов BusinessClasses_Interface.inc. Фрагмент кода из этого файла приведен в листинге 12.2. В начале кода располагается «шапка», содержащая информацию о дате создания файла, и предупреждение о том, что все внесенные в этот файл «ручные» изменения могут быть утеряны. Сгенерированная структура описания интерфейсов достаточно понятна, и здесь мы обратим внимание лишь на некоторые важные моменты. Во-первых, из листинга видно, что для каждого класса UML-модели генерируется два программных

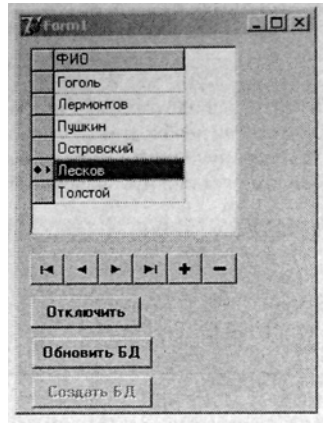


Рис. 12.8. Приложение в работе

класса — один класс (например, TAvtor) предназначен для описания собственно класса модели, а другой — для описания соответствующей коллекции-массива (TAvtorList). Во-вторых, роли ассоциаций класса модели входят в состав соответствующего программного класса в качестве классов-свойств, причем, в зависимости от размерности конкретной роли, она может быть представлена как классом объекта, так и классом коллекции объектов.

Листинг 12.2. Фрагмент кода описания интерфейсов

```
(*****************************************)
(*      This file is autogenerated       *)
(*   Any manual changes will be LOST!    *)
(*****************************************)
(* Generated 29.01.2004 15:40:13         *)
(*****************************************)
(• This file should be stored in the     •)
(• same directory as the form/datamodule •)
(• with the corresponding model          *)
(*****************************************)
(* Copyright notice:                     •)
(*                                       •)
(*****************************************)
{SIFNDEF BusinessClasses_Interface.inc}
{$DEFINE BusinessClasses_Interface.inc}
{SIFNDEF BusinessClasses_unitheader}
unit BusinessClasses;
{$ENDIF}
interface
uses
   // interface uses
   // interface dependencies
```

```
   // attribute classes
   BoldAttributes,
   // other
   Classes,
   Controls, // for TDate
   SysUtils,
   BoldDefs,
   BoldSubscription,
   BoldDeriver,
   BoldElements,
   BoldDomainElement,
   BoldSystemRT,
   BoldSystem;
type
{ forward declarations of all classes }
TBusinessClassesRoot = class;
TBusinessClassesRootList = class;
TAvtor = class;
TAvtorList = class;
TStrana = class;
TStranaList = class;
TKniga = class;
TKnigaList = class;
TLinkNapisalNapisana = class;
TLinkNapisalNapisanaList = class;
TBusinessClassesRoot = class(TBoldObject)
private
protected
public
end;
TAvtor = class(TBusinessClassesRoot)
private
   function _Get_M_FIO: TBAString;
   function _GetFIO: String;
   procedure _SetFIO(const NewValue: String);
   function _Get_M_tpic: TBATypedBlob;
   function _Gettpic: String;
   procedure _Settpic(const NewValue: String);
   function _GetNapisal: TKnigaList;
   function _GetLinkNapisalNapisana: TLinkNapisalNapisanaList;
   function _GetStrana: TStrana;
   function _Get_M_Strana: TBoldObjectReference;
   procedure _SetStrana(const value: TStrana);
protected
public
   property M_FIO: TBAString read _Get_M_FIO;
   property M_tpic: TBATypedBlob read _Get_M_tpic;
   property M_Napisal: TKnigaList read _GetNapisal;
   property M_LinkNapisalNapisana: TlinkNapisalNapisanaList
    read _GetLinkNapisalNapisana;
   property M_Strana: TBoldObjectReference read _Get_M_Strana;
   property FIO: String read _GetFIO write _SetFIO;
```

```
    property tpic: String read _Gettpic write _Settpic;
    property Napisal: TKnigaList read _GetNapisal;
    property LinkNapisalNapisana: TlinkNapisalNapisanaList
     read _GetLinkNapisalNapisana;
    property Strana: TStrana read _GetStrana write _Set5trana;
end;
```

Так, например, в описании класса TAvtor присутствует свойство Strana типа **TStrana,** и свойство Napisal типа TKnigaList. Наличие этих двух свойств объясняется тем обстоятельством, что в соответствии с UML-моделью (см. рис. 12.1) автор может проживать только в одной стране (кратность роли равна 1, программный класс TSstrana), а с другой стороны, автор может написать много книг (кратность роли больше 1, программный класс TKnigaList). Таким способом при генерации кода среда Bold обеспечивает исчерпывающее описание бизнес-правил, заложенных в UML-модель, на уровне генерируемых программных конструкций. И, наконец, в приведенном фрагменте кода присутствует еще одна особенность. В описании атрибутов (свойств программных классов) легко заметить «двойные» объявления свойств. Например, для атрибута FIO класса Avtor присутствуют два похожих программных объявления:

```
property FIO: String read _GetFIO write _SetFIO
property M_FIO: TBAString read _Get_M_FIO;
```

С первой строкой все понятно — описывается строковое свойство FIO и соответствующие методы (процедуры) доступа к нему — _GetFIO для чтения, и _SetFIO для записи. А вот для чего Bold сгенерировал вторую строку с объявлением похожего свойства M_FIO? Чтобы ответить на этот вопрос, обратим внимание на тип этого свойства — TBAString. Это свойство не является текстовым атрибутом, а по сути играет роль описателя метаинформации, то есть информации о типах (отсюда и префикс «М» в идентификаторе). Подобные «метаатрибуты» представляют собой специальные объекты, которые выполняют следующие важные основные функции:

- управление загрузкой значений атрибутов;
- кэширование изменений и временное хранение предыдущих значений атрибута;
- поддержка механизма подписки (subscribing);
- вычислениеderived-атрибутов.

Для простоты во многих случаях можно руководствоваться следующим правилом — свойство «А» представляет собой **значение** свойства «M_A». Например, для рассматриваемого свойства атрибута FIO будет справедлив следующий метод доступа к значению атрибута (фамилии автора).

```
FIO:=M_FIO.AsString
```

В корректности сформулированного правила мы убедимся далее, при изучении содержимого другого сгенерированного файла — BoldClasses.pas. Рассмотрим фрагмент (листинг 12.3), включающий начало этого файла и программную реализацию класса **Avtor.**

Листинг 12.3. Фрагмент сгенерированного кода модуля BusinessClasses.pas

```
(*****************************************)
(*      This file is autogenerated       *)
(*   Any manual changes will be LOST!    *)
(*****************************************)
(• Generated 29.01.2004 15:40:13          •)

(*****************************************)
(* This file should be stored in the     *)
(* same directory as the form/datamodule •)
(* with the corresponding model          *)
(*****************************************)
(*****************************************)
(* Copyright notice:                     *)
C                                         *)

(*****************************************)
unit BusinessClasses;
{$DEFINE BusinessClasses_unitheader}
{$INCLUDE BusinessClasses_Interface.inc}
{ Includefile for methodimplementations }
const
BoldMemberAssertInvalidObjectType: string =
'Object of singlelink (%s.%s) is of wrong type (is %s, should be
  %s)';
{ TBusinessClassesRoot }
procedure TBusinessClassesRootList.Add(NewObject:
TBusinessClassesRoot);
begin
   if Assigned(NewObject) then
        AddElement(NewObject);
end;
function TBusinessClassesRootList.IndexOf
(anObject: TBusinessClassesRoot): Integer;
begin
   result := IndexOfElement(anObject);
end;
function TBusinessClassesRootList.Includes
(anObject: TBusinessClassesRoot) : Boolean;
begin
   result := IncludesElement(anObject);
end;
function TBusinessClassesRootList.AddNew: TBusinessClassesRoot;
begin
   result := TBusinessClassesRoot(InternalAddNew);
end;
procedure TBusinessClassesRootList.Insert
(index: Integer; NewObject: TBusinessClassesRoot);
begin
   if assigned(NewObject) then
        InsertElement(index, NewObject);
end;
function TBusinessClassesRootList.GetBoldObject
(index: Integer): TBusinessClassesRoot;
```

```
begin
   result := TBusinessClassesRoot(GetElement(index));
end;
procedure TBusinessClassesRootList.SetBoldObject
(index: Integer; NewObject: TBusinessClassesRoot);
begin;
   SetElement(index, NewObject);
end;
{ TAvtor }
function TAvtor._Get_M_FIO: TBAString;
begin
   assert(ValidateMember('TAvtor', 'FIO', 0, TBAString));
   Result := TBAString(BoldMembers[0]);
end;
function TAvtor._GetFIO: String;
begin
   Result := M_FIO.AsString;
end;
procedure TAvtor._SetFIO(const NewValue: String);
begin
   M_FIO.AsString := NewValue;
end;
function TAvtor._Get_M_tpic: TBATypedBlob;
begin
   assert(ValidateMember('TAvtor', 'tpic', 1. TBATypedBlob));
   Result := TBATypedBlob(BoldMembers[1]);
end;
function TAvtor._Gettpic: String;
begin
   Result := M_tpic.AsString;
end;
procedure TAvtor._Settpic(const NewValue: String);
begin
   M_tpic.AsString := NewValue;
end;
function TAvtor._GetNapisal: TKnigaList;
begin
   assert(ValidateMember('TAvtor', 'Napisal', 2, TKnigaList));
   Result := TKnigaList(BoldMembers[2]);
end;
function TAvtor._GetLinkNapisalNapisana:
TLinkNapisalNapisanaList;
begin
   assert(ValidateMember('TAvtor', 'LinkNapisalNapisana', 3,
    TLinkNapisalNapisanaList));
   Result := TLinkNapisalNapisanaList(BoldMembers[3]);
end;
function TAvtor._Get_M_Strana: TBoldObjectReference;
begin
   assert(ValidateMember('TAvtor', 'Strana', 4,
TBoldObjectReference));
   Result := TBoldObjectReference(BoldMembers[4]);
```

```
end;
function TAvtor._GetStrana: TStrana;
begin
assert(not assigned(M_Strana.BoldObject) or
(M_Strana.BoldObject is TStrana),SysUtils.format
(BoldMemberAssertInvalidObjectType,
[ClassName, 'Strana', M_Strana.BoldObject.ClassName,
  'TStrana']));
Result := TStrana(M_Strana.BoldObject);
end;
procedure TAvtor._SetStrana(const value: TStrana);
begin
  M_Strana.BoldObject := value;
end;
procedure TAvtorList.Add(NewObject: TAvtor);
begin
  if Assigned(NewObject) then
       AddElement(NewObject);
end;
function TAvtorList.IndexOf(anObject: TAvtor): Integer;
begin
  result := IndexOfElement(anObject);
end;
function TAvtorList.Includes(anObject: TAvtor) : Boolean:
begin
  result := IncludesElement(anObject);
end;
function TAvtorList.AddNew: TAvtor;
begin
  result := TAvtor(InternalAddNew);
end;
procedure TAvtorList.Insert(index: Integer: NewObject: TAvtor);
begin
  if assigned(NewObject) then
       InsertElement(index, NewObject);
end;
function TAvtorList.GetBoldObject(index: Integer): TAvtor;
begin
  result := TAvtor(GetElement(index));
end;
procedure TAvtorList.SetBoldObject(index: Integer; NewObject:
TAvtor);
begin;
  SetElement(index, NewObject);
end;
```

Этот фрагмент также содержит «шапку» с информацией, содержащей дату и время генерации кода, и предупреждение о возможной потере «постороннего» кода. В начале кода присутствуют общие процедуры-методы класса TbusinessClassesRootList, который в данном случае выступает в качестве суперкласса элементов модели. Эти методы обеспечивают операции добавления объектов, получения индекса текущего объекта и ряд других (см. листинг 12.3). Далее содержится код

реализации для классов Avtor и AvtorList. Сразу можно отметить использование одинаковых программных конструкций вида

```
function TAvtor._GetFIO: String;
begin
   Result := M_FIO.AsString;
end;
```

для доступа к значению атрибутов (программных свойств) класса, о чем говорилось выше при рассмотрении свойств типа M_FIO. Похожим образом осуществляются и операции записи в свойства-атрибуты:

```
procedure TAvtor._SetFIO(const NewValue: String);
begin
   M_FIO.AsString := NewValue;
end;
```

Представленные фрагменты (см. листинги 12.2, 12.3) могут быть полезны для самостоятельного изучения принципов программирования работы с объектами в среде Bold for Delphi. Подробно разбирать все используемые в приведенном коде программные конструкции мы не станем в силу достаточно большого объема кода. Гораздо полезнее будет познакомиться с практическим применением сгенерированного программного кода при решении конкретных задач, к чему мы и приступим в следующем разделе.

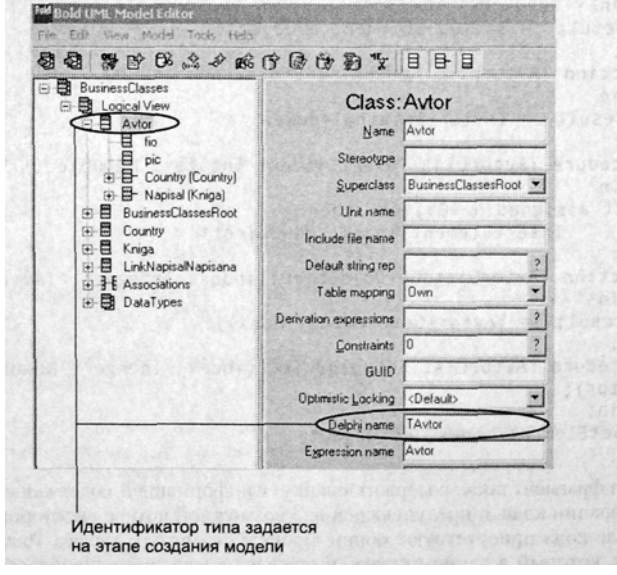


Рис. 12.9. Задание идентификатора типа

Практическое использование кода модели

Работа с классами и атрибутами

Для начала поставим простую задачу — добавление новых авторов в список. Для этого добавим на форму кнопку Buttonl, присвоив ей заголовок Добавить авторов, а в процедуру обработки события OnClick этой кнопки введем одну строку кода

```
TAvtor.Create(nil);
```

Этот оператор просто создает новый экземпляр объекта типа TAvtor. Напомним, что идентификатор этого типа задается тег-параметром DelphiName (см. главу 4) и по умолчанию формируется добавлением префикса «T» к названию класса (рис. 12.9).

Запустим приложение и убедимся, что при нажатии кнопки в список авторов добавляются новые пустые строки (рис. 12.10).

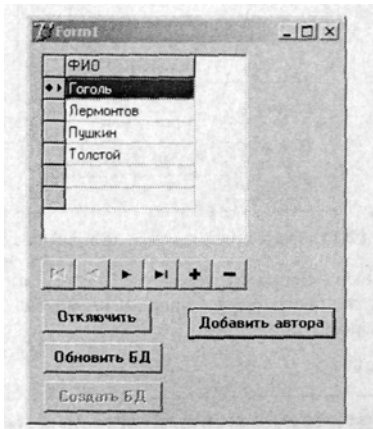


Рис. 12.10. Программное добавление «пустых» авторов

Для обеспечения программного формирования атрибута-фамилии автора добавим на форму окно ввода Editl, в которое будем вводить фамилию нового автора. Программную реакцию на нажатие кнопки модифицируем следующим образом:

```
procedure TForm1.Button4Click(Sender: TObject);
var NewAvtor:TAvtor;
begin
   if Edit1.Text<>'' then
   begin
        NewAvtor:=TAvtor.Create(nil);
        NewAvtor.FIO:=Edit1.Text;
   end;
end;
```

В данном случае наглядно видно, как просто и естественно осуществляется доступ к атрибутам класса. Используется обычная dot-нотация, принятая в языке Object Pascal для доступа к атрибутам класса — NewAvtor.FIO. Запустим модифицированное приложение и убедимся, что фамилии добавляемых авторов действительно совпадают с введенным в окно ввода текстом (рис. 12.11). Если окно ввода оставить пустым, то добавления нового автора не произойдет, поскольку соответствующая проверка присутствует в написанном нами обработчике события.

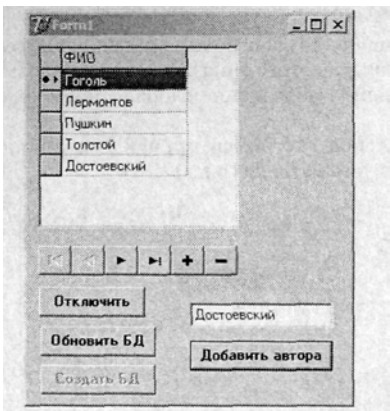


Рис. 12.11. Программное присвоение атрибута-фамилии

Для удаления объектов используется метод Delete. Например, мы можем сразу после создания удалить нового автора, воспользовавшись для этой цели следующим простым оператором:

```
NewAvtor.Delete;
```

ВНИМАНИЕ

Для операции удаления объектов классов модели нельзя применять обычный метод Free. Вызов данного метода приведет к инициации программного исключения.

Работа с ассоциациями

Немного усложним наше приложение для решения следующей задачи — отображать страну, в которой проживает текущий автор, программным способом. Для этого на форму приложения добавим метку Labell для отображения названия страны и еще одну кнопку, в процедуру-обработчик нажатия которой введем следующий программный код:

```
procedure TForm1.Button5Click(Sender: TObject);
var CurAvtor:TAvtor;
begin
   CurAvtor:=ListAllAvtors.CurrentBoldObject as TAvtor;
   label1.Caption:=CurAvtor.Strana.Nazvanie;
end;
```

Опять можно наблюдать, как просто и наглядно осуществляется программный доступ к ассоциации и ее роли из программного кода. Оператор CurAvtor.Strana.Nazvanie полностью «прозрачен» и выглядит очень просто, однако при этом он реализует довольно сложную операцию доступа к другому классу (Strana) и получения из него значения-атрибута Nazvanie. Первый оператор в приведенном выше фрагменте кода имеет также простой смысл — присвоить объекту TAvtor значение текущего автора в списке. После запуска приложения убеждаемся, что поставленная задача решена (рис. 12.12).

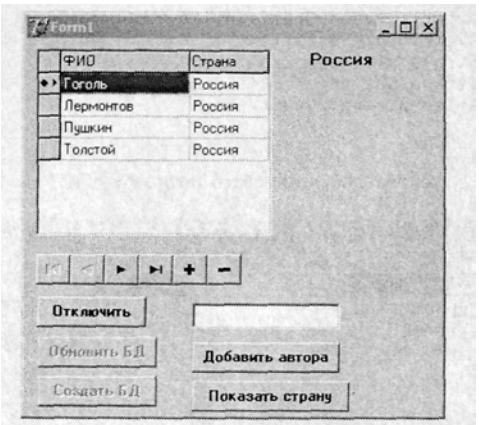


Рис. 12.12. Доступ к значению роли ассоциации

Попробуем теперь таким же простым способом при вводе нового автора присваивать ему заодно и страну проживания. Добавим в обработчик события кнопки Добавить автора следующий оператор:

```
NewAvtor.Strana.Nazvanie:='Россия';
```

В результате после попытки ввести нового автора получим программное исключение (рис. 12.13), окно сообщения которого предупреждает, что класс TStrana не существует.

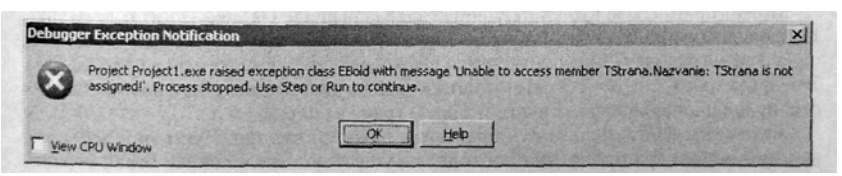


Рис. 12.13. Попытка доступа к несуществующему объекту

Это вполне очевидный результат, поскольку при добавлении нового автора у него еще не существует связи ни с одной страной. Попробуем одновременно с автором заносить и страну проживания. Название страны будем вносить во второе окно

ввода Edit2. Для лучшего понимания происходящего поместим на форму дескриптор списка стран ListAllCountry и вторую сетку BoldGrid2 для отображения списка стран. Для внесения страны изменим реакцию на нажатие кнопки Добавить автора:

```
procedure TForm1.Button4Click(Sender: TObject);
var NewAvtor:TAvtor;
NewStrana :TStrana;
begin
   if Edit1.Text<>'' then
   begin
        NewAvtor:=TAvtor.Create(nil);
        NewAvtor.FIO:=Edit1.Text;
        NewStrana:=TStrana.Create(nil);
        NewStrana.Nazvanie:=Edit2.Text;
        NewAvtor.Strana:=NewStrana;
   end;
end;
```

Запустим приложение и добавим нового автора (рис. 12.14).

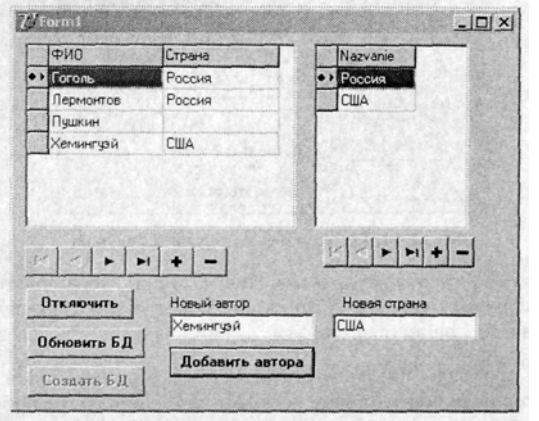


Рис. 12.14. Одновременное создание двух объектов и связи между ними

Может показаться, что поставленная задача решена. Однако это не совсем так. После ввода второго автора, проживающего в той же стране, мы обнаружим, что одна и та же страна (США) повторяется дважды в общем списке стран (рис. 12.15). Это происходит, потому что мы автоматически каждый раз добавляем новый объект-страну, не проверяя, существует ли такая страна в списке.

Усложним нашу процедуру-обработчик нажатия кнопки. Пусть при добавлении нового автора процедура проверяет, существует ли в списке стран страна с названием, введенным в окно ввода Edit2. Если страна существует, то она связывается с новым автором. В противном случае создается новый объект-страна и также связывается с новым автором. Код процедуры снабжен подробными комментариями (листинг 12.4).

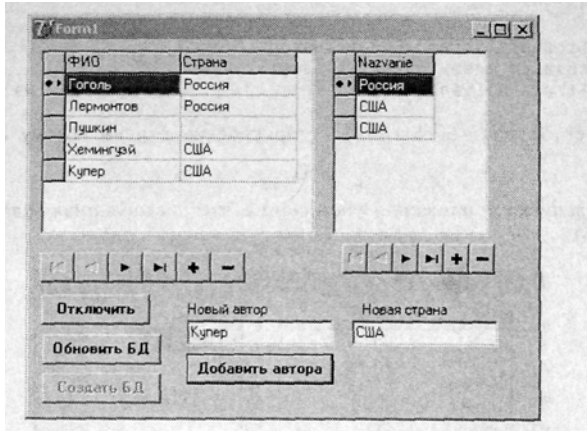


Рис. 12.15. Некорректное дублирование объектов-стран

Листинг 12.4. Процедура, добавляющая нового автора и, при необходимости, новую страну проживания

```
procedure TForm1.Button4Click(Sender: TObject);
var NewAvtor   :TAvtor;
   CurStrana   :TStrana;
   i: integer;
   bStranaExists :boolean; // признак существования страны
begin
if Edit1.Text<>'' then
begin
   NewAvtor:=TAvtor.Create(nil); // создаем нового автора
   NewAvtor.FIO:=Edit1.Text;     // фамилию автора берем из Edit1
   bStranaExists:=False;     // считаем, что страны не существует
   // цикл по всем элементам списка стран
   for i:=0 to ListAllCountry.Count-1 do
   begin
   CurStrana:=ListAllCountry.ObjectList.Elements[i] as TStrana;
   //текущий
   //элемент-страна
   if CurStrana.Nazvanie=Edit2.Text then
   // проверка совпадения названия
   //текущей страны и текста, введенного в Edit2
   begin
   bStranaExists:=true;
   // устанавливаем признак существования страны
   break; // выход из цикла, если нашли страну в списке
end;
end;
if (not bStranaExists) then
// если страна не найдена, создаем новую
```

```
begin
   CurStrana:=TStrana.Create(nil);
   // создаем новый объект-страну
   CurStrana.Nazvanie:=Edit2.Text; // название страны из Edit2
end;
NewAvtor.Strana:=CurStrana; // присвоение страны новому автору
end;
end;
```

После запуска приложения убеждаемся, что поставленная задача решена (рис. 12.16).

Рис. 12.16. Отлаженное приложение в работе

Операции

Генерация кода позволяет использовать *операции,* включаемые в классы модели. Операции, по сути, представляют собой аналоги методов в объектно-ориентированном программировании. По этой причине формируемые в UML-модели операции при генерации кода «отображаются» в обычные методы классов. Каждая операция может иметь входные параметры и возвращать определенный тип, то есть исполнять роль функций или процедур. Рассмотрим на простом примере, как происходит работа с операциями. Создадим операцию **BookCount** для подсчета общего количества книг, написанных автором. Для полноты картины создадим ее в Rational Rose. Для этого загрузим нашу модель в редактор диаграмм классов Rational Rose (см. главу 4) и, вызвав спецификацию класса **Avtor,** перейдем на вкладку Operations (рис. 12.17).

Вызвав контекстное меню, добавим операцию и после двойного щелчка по ней перейдем в окно спецификации новой операции. Присвоим операции имя BookCount и зададим целый тип возвращаемого результата (см. рис. 12.17). После этого добавленная операция появится на изображении класса в UML-модели (рис. 12.18).

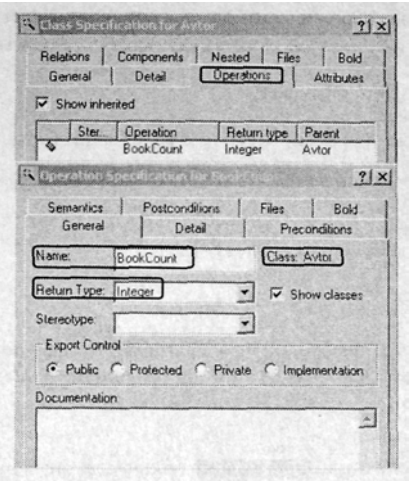


Рис. 12.17. Настройка свойств операции в Rational Rose

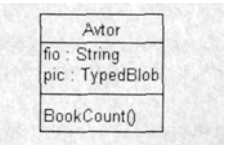


Рис. 12.18. Класс с операцией BookCount

Импортируем усовершенствованную модель в среду Bold for Delphi и увидим, что операция отобразилась в дереве элементов модели (рис. 12.19).

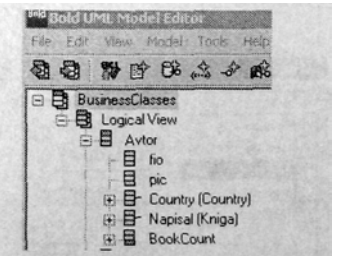


Рис. 12.19. Дерево модели с добавленной операцией

Теперь нам необходимо повторно сгенерировать код классов модели, для того чтобы в нем нашел отражение добавленный новый элемент UML-модели. Сделаем это и обнаружим, что при генерации появился новый файл BusinessClasses.inc, содержащий программную заготовку для операции **BookCount** (листинг 12.5).

Листинг 12.5. Заготовка программного кода для новой операции

```
(*                                                    *)
(*  Bold for Delphi Stub File                         *)
C                                                     *)
(*  Autogenerated file for method implementations     *)
(*                                                    *)


{$INCLUDE BusinessClasses_Interface.inc}
function TAvtor.BookCount: Integer;
begin

end;
```

Нам осталось вписать необходимые операторы в тело функции. В данном случае для подсчета общего количества книг достаточно написать всего одну строку программного кода:

```
Result:=Napisal.Count;
```

Этот оператор обеспечит возврат общего количества значений роли Napisal, что и соответствует количеству книг автора. Нельзя опять не отметить простоту и элегантность программных выражений, обеспечиваемую средой Bold при использовании генерации кода.

Для проверки работоспособности операции добавим на форму стандартную метку Label1 и кнопку Операция, в обработчик события нажатия которой введем следующий код:

```
procedure TForm1.Button4Click(Sender: TObject);
var CurAvtor:TAvtor;
begin
   CurAvtor:=ListAllAvtors.CurrentBoldObject as TAvtor;
   Label1.Caption:='Количество книг
     +IntToStr(CurAvtor.BookCount);
end;
```

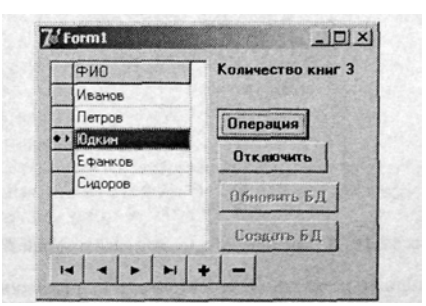


Рис. 12.20. Операция в работе

Мы видим, что обращение к операции производится точно так же, как и к любому другому методу класса — CurAvtor.BookCount. После запуска приложения можно убедиться, что операция исправно производит подсчет книг для любого выбранного автора (рис. 12.20).

Естественно, подобную простую задачу можно было бы решить гораздо проще с помощью OCL и одного вычисляемого атрибута. Но не стоит по этой причине недооценивать операции, поскольку возможности языка OCL все-таки ограничены, а возможности программного кода — практически нет. Поэтому с помощью операций на практике можно инкапсулировать в классы модели и программно реализовывать очень сложные методы управления поведением объектов и обработки информации.

Резюме

Генерация кода является удобным средством для создания программных объектов, отражающих элементы UML-модели. С этими программными объектами можно работать как с обычными классами в ООП, при этом очень легко и наглядно реализуются методы доступа к атрибутам, ассоциациям и ролям, присутствующим в модели. Дополнительным преимуществом применения кодогенератора является возможность создания и использования операций, которые в генерируемом программном коде отображаются в обычные методы классов. В следующей главе будут продемонстрированы и другие преимущества использования генерации кода.

Механизм «подписки»

В этой главе будет рассматриваться так называемый механизм «подписки» (subscribing). Данный механизм, без преувеличения, играет центральную роль в организации взаимодействия составных элементов и уровней приложения во время его работы. Синхронизация графического интерфейса с бизнес-уровнем, вычисляемые атрибуты и ассоциации и реализация многих других функций обеспечивается механизмами подписки.

Описание и реализация

События и подписка

В одной из предыдущих глав упоминалось, что программная система Bold for Delphi по сути представляет собой «систему в системе». Обладая собственным менеджером памяти, собственными типами — расширениями стандартных типов, Bold for Delphi обладает также и собственными средствами для обмена сообщениями. Такие сообщения возникают, например, когда изменяется состояние какого-либо атрибута объекта или происходит перемещение по списку объектов и т. д. Таким образом, сообщения генерируются средой Bold при наступлении определенных событий. Все события можно разделить на внутренние и пользовательские. Внутренние события использует сама программная среда. Например, синхронизация элементов графического интерфейса обеспечивается благодаря многочисленным внутренним событиям, возникающим на бизнес-уровне при изменении состояния каких-либо элементов объектного пространства. Пользовательские события предназначены для применения разработчиком. Однако очевидно, что наличия одних только событий недостаточно для решения задач взаимодействия между элементами системы. Необходимо присутствие некоторого механизма, который обеспечивает не только генерацию событий, но и их отслеживание, и поддержку реализа-

ции ответных реакций. Именно в качестве такого инструмента и выступает механизм подписки. Этот механизм дает возможность «подписать» выбранный элемент на некоторое событие, происходящее в системе. Причем подписаться можно на совершенно определенное событие, например на изменение состояния конкретного атрибута объекта конкретного класса. И при изменении значения такого атрибута пользователем элемент-«подпнсчик» получит уведомление от системы и выполнит какие-то ответные действия. Если посмотреть на функционирование Bold-приложения во время его работы, то можно образно представить некую «живую» систему из множества элементов, которые постоянно «общаются» между собой, вырабатывая сообщения-события, и, в свою очередь, «отвечая» на эти сообщения. Простой щелчок мышью на сетке вызывает резкое «оживление» такого «сообщества», однако не хаотическое, а полностью упорядоченное и контролируемое ядром программной системы. Еще раз подчеркнем, что события в Bold for Delphi и обычные события в Windows — вещи разные, и механизм подписки реализуется программной системой Bold tor Delphi совершенно независимо.

ПРИМЕЧАНИЕ

Возможно, одной из причин такой «изолированности» событийной среды в Bold for Delphi явилось намерение разработчиков в дальнейшем создать версии этого продукта и для других операционных систем. К сожалению, на настоящий момент это пока не реализовано.

Основные классы и реализация

Возможность использования механизма подписки «встроена» разработчиками Bold for Delphi в важный внутренний класс TBoldElement (см. главу 6), который является суперклассом для любых типов элементов объектного пространства. Непосредственным «родителем» класса TBoldElement является класс TBoldSubscibableObject, из названия которого понятна его роль — он является абстрактным классом для всех тех дочерних типов, которые «нуждаются» в использовании подписки. Непосредственными «реализаторами» механизма подписки являются классы TBoldPublisher (входящий в качестве свойства в класс TBoldSubscribableObject) и TBoldSubscriber. Класс TBoldPublisher обеспечивает добавление подписчиков методами AddSubscription и AddSmallSubscription. «Малая» (Small) (или лучше сказать — стандартная) подписка использует стандартные события, зарезервированные разработчиками Bold. Каждое событие описывается простым целым значением-номером, и для стандартных событий выделены номера от 0 до 31 (табл. 13.1). Из них события с номерами от 24 до 31 предоставлены под пользовательские «малые» события, которые также могут быть использованы в стандартных подписках.

Таблица 13.1. Перечень основных стандартных событий

Номер (значение)	Название	Когда вырабатывается
О	beDestroying	Перед уничтожением объекта
2	beMemberChanged	Генерирует объект при изменении своего члена (атрибута)
3	beObjectDeleted	После удаления объекта
4	beItemAdded	При добавлении объекта в список
5	beItemDeleted	При удалении объекта из списка

Таблица 13.1 (*продолжение*)

Номер (значение)	Название	Когда вырабатывается
6	beItemReplaced	При замене объекта в списке на другой объект
7	beOrderChanged	После изменения порядка элементов в списке
8	beValueChanged	Генерирует BoldElement (например, атрибут) после изменения своего значения
11	beObjectCreated	При создании нового объекта
12	beValueInvalid	При недопустимом значении элемента
17	beDeactivating	При деактивации уровня данных
18	beRolledBack	Генерирует BoldSystem при откате транзакции
21	beObjectFetched	Посылает объект при его вызове

Преимущество использования стандартной подписки (метод AddSmallSubscription) в том, что в одной заявке на подписку можно подписаться сразу на несколько событий (хотя это достигается использованием ограниченного набора стандартных событий). Функционирование механизма подписки происходит следующим образом. Элемент, желающий оформить подписку на какое-нибудь событие, вызывает методы класса TboldPublisher (этот класс присутствует во многих стандартных классах, поскольку большинство используемых классов являются наследниками TboldElement): AddSmallSubscription или AddSubscription, используя в качестве параметра объект типа TboldSubscriber. Кроме этого, в параметрах запроса передаются номер события (или сразу несколько номеров для стандартной подписки методом AddSmallSubscription) и обязательный идентификатор события. Идентификатор используется механизмом для привязки к подписчику. Далее, если данное событие наступило, объект BoldPublisher проверяет, присутствуют ли подписчики на это событие, и для всех найденных подписчиков вызывает процедуру — метод класса TBoldSubscriber.Receive (объект класса TBoldSubscriber передается как параметр при «оформлении» подписки), в которую разработчик может поместить реакцию на данное событие. Далее в этой главе будет рассмотрено на конкретном примере, как все вышесказанное осуществляется на практике. Это — очень упрощенное описание механизма, в который на самом деле заложено гораздо больше возможностей. И хотя механизм подписки весьма объемен и довольно сложен для восприятия, ситуация существенно облегчается тем, что на практике необходимость его «ручного» использования возникает редко. Причина этого в том, что в основном механизм подписки используется самой программной средой Bold for Delphi, которая предоставляет разработчику уже значительно адаптированные высокоуровневые средства для реализации подавляющего большинства требуемых возможностей. В конце этой главы будут рассмотрены примеры работы с подобными средствами — программная работа с вычисляемыми атрибутами, а также использование так называемых «обратно-вычисляемых» (reverse-derived) атрибутов, наличие которых является уникальной особенностью продукта Bold for Delphi.

Программное использование подписки

В этом разделе на конкретном примере будут проиллюстрированы основные приемы работы с механизмом подписки при использовании «ручного» программиро-

вания. В качестве основы будем использовать созданные в предыдущих главах простые приложения, отображающие список авторов. Пользователь при работе с этим списком может совершать различные действия — добавление, удаление, изменение записей и т. д. При этом автоматически вырабатываются соответствующие события в системе. Покажем на простом примере, как «подписаться» на такого рода события и обрабатывать соответствующие реакции при получении уведомлений о наступлении этих событий. Для подписки необходимо наличие, по крайней мере, двух составляющих компонентов — того, кто подписывается, и того, «на кого подписываются». Подписываться мы будем на список авторов, который представлен дескриптором списка ListAllAvtors. А собственно «подписчика» создадим сами. В качестве такого элемента создадим класс-наследник класса TboldSubscriber. Для этого добавим в программный модуль описание нового класса, присвоив ему имя TnewSubscriber:

```
type
   TNewSubscriber=class(TBoldSubscriber)
end;
```

Если считать что в дальнейшем мы правильно «оформим» подписку, то при наступлении события (какого — пока непонятно, так как мы его еще не выбрали), среда Bold автоматически вызовет метод Receive нашего подписчика. В родительском классе TboldSubscriber эта процедура не наполнена содержанием, так как этот класс — абстрактный. Поэтому необходимо переопределить стандартную процедуру Receive, что делается простым добавлением строки (код процедуры будет написан позже):

```
type
TNewSubscriber=class(TBoldSubscriber)
procedure Receive(Originator: TObject;
OriginalEvent: TBoldEvent; RequestedEvent:
TBoldRequestedEvent);override;
end;
```

Рассмотрим подробнее состав параметров процедуры Receive:

- Originator — указывает на источник события;
- OriginalEvent — тип наступившего события (его номер), по этому параметру можно определить, какое именно событие наступило;
- RequestedEvent — идентификатор события, необходимый для привязки подписчиков к конкретной подписке.

Все перечисленные параметры процедуры Receive являются входными и формируются автоматически программной средой (объектом типа TboldPublisher того элемента, на который осуществляется подписка). Далее для использования в программе мы должны описать переменную-объект созданного нами типа. Назовем ее NewSub.

```
Var NewSub : TNewSubscriber;
```

Кроме того, зададим для подписчика какой-нибудь идентификатор, например MyEvent.

```
Const MyEvent=58;
```

Приступим к оформлению нашей подписки. Поместим на форму кнопку Оформить подписку и введем следующий код реакции на ее нажатие:

```
procedure TForm1.Button5Click(Sender: TObject);
begin
   NewSub:=TNewSubscriber.Create;
listallavtors.List.AddSmallSubscription(NewSub,[beItemAdded],MyEvent);
end;
```

Первый оператор создает экземпляр подписчика NewSub (который необходимо удалить впоследствии обычным методом NewSub.Free). Второй оператор формирует стандартную подписку, указывая в качестве параметров: подписчика NewSub, набор стандартных событий (состоящий из одного события beltemAdded) и идентификатор MyEvent. В данном случае мы воспользовались тем, что в классе-дескрипторе списка ListAllAvtors уже существует свойство типа TboldPublisher, позволяющее нам использовать метод AddSmallSubscrption. Используемое в подписке событие beltemAdded возникает при добавлении элемента в список. Теперь нам осталось реализовать программную реакцию на данное событие. Для этого создадим код процедуры Receive для класса-подписчика TnewSubscriber:

```
Procedure TNewSubscriber.receive;
begin
   ShowMessage('Внимание! Автор добавлен в список!');
end;
```

То есть при получении сообщения данная процедура должна вызвать появление стандартного окна диалога. Запустим приложение, нажмем кнопку Оформить подписку и попробуем добавить нового автора (рис. 13.1).

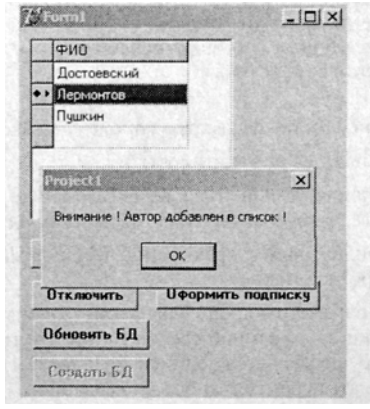


Рис. 13.1. Реакция на событие

Мы убедимся, что механизм подписки работает. Если же мы попробуем удалять или редактировать авторов, ничего происходить не будет. Теперь немного модифицируем наше приложение. Во-первых, «подпишемся» одновременно на два события — добавление (beltemAdded) и удаление (beItemDeleted) авторов:

```
procedure TForm1.Button5Click(Sender: TObject);
begin
   NewSub:=TNewSubscriber.Create;
   listallavtors.List
.AddSmallSubscription(NewSub,[beItemAdded,beItemDeleted],MyEvent);
end;
```

Во-вторых, попытаемся различать, какое именно событие произошло, в процедуре Receive:

```
Procedure TNewSubscriber.receive;
begin
   case OriginalEvent of
   beltemDeleted:   ShowMessage('Внимание! Автор удален из
списка!');
   beltemAdded:     ShowMessage('Внимание! Автор добавлен в
список!');
   end; // case
end;
```

После запуска приложения убедимся, что в результате добавления или удаления авторов на экран выводятся различные сообщения (рис. 13.2).

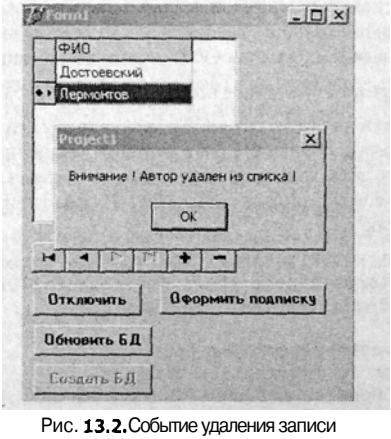


Рис. **13.2.** Событие удаления записи

На базе рассмотренного примера можно создавать при необходимости и более сложные варианты подписки.

Использование BoldPlaceableSubscriber

В предыдущей главе мы уже применяли компонент BoldPlaceableSubscriber, правда, не совсем по прямому назначению. На самом деле этот компонент предназначен для формирования пользовательских подписок, значительно облегчая их реализа-

цию и устраняя необходимость «ручного» создания классов и процедур, как мы это делали в предыдущем примере. Рассмотрим работу с этим компонентом. Для этого поместим компонент BoldPlaceableSubscriber на форму и обратим внимание на его свойства-события. Их всего два — OnReceive и OnSubscribeToElement. Назначение обработчика события OnReceive эквивалентно описанной выше процедуре Receive, то есть в этот обработчик помещается процедура реакции на подписанное событие. Второй обработчик, для события OnSubscribeToElement, позволяет «оформить» собственно подписку. Поставим следующую задачу — отслеживать работу с базой данных в части добавления и удаления книг у конкретных авторов. Сначала подключим компонент BoldPlaceableSubscriber к списку авторов (свойство BoldHandle). Далее, оформим нашу подписку, введя в обработчик события OnSubscribeToElement следующий код:

```
procedure TForm1.BoldPlaceableSubscriber1SubscribeToElement(
element: TBoldElement; Subscriber: TBoldSubscriber);
begin
   Element.SubscribeToExpression
   ('Avtor.allInstances->select(fio=''Пушкин'')
   ->first.napisal', Subscriber, true);
end;
```

В этом обработчике используется присущее всем элементам ОП (наследникам класса TBoldElement) свойство — возможность подписаться на результат OCL-выражения. В данном случае мы подписываемся на количество книг, написанных конкретным автором (Пушкин), применяя OCL-навигацию по модели

```
'Avtor.allInstances->select(fio=''Пушкин'')->first. napisal'
```

Что такое «подписка на результат OCL-выражения»? Это значит, что при любом изменении результата оценки OCL-выражения подписчик автоматически получит уведомление, то есть, как мы уже знаем, будет вызван метод Receive (в данном случае будет вызван обработчик OnReceive). В обработчик события OnReceive введем следующий код:

```
procedure TForm1.BoldPlaceableSubscriber1Receive(
sender: TBoldPlaceableSubscriber; Originator: TObject;
OriginalEvent, RequestedEvent: Integer);
begin
   if OriginalEvent=beItemAdded then
   if (Originator is TKnigaList) then
   ShowMessage('Попытка добавить книгу к автору Пушкин!');
   if OriginalEvent=beItemDeleted then
   if (Originator is TKnigaList) then
   ShowMessage('Попытка удалить книгу у автора Пушкин!');
end;
```

В этом обработчике проверяется, какое именно событие наступило и кто является его источником. В качестве источника события выступает объект Originator, значение которого сравнивается с классом TKnigaList. Эта проверка необходима, чтобы исключить из обработки события, возникающие при удалении или добавлении авторов, поскольку нас в данном случае интересуют не авторы, а книги. Если теперь запустить приложение, то легко убедиться, что реакции на события воз-

никают только в тех случаях, когда производятся манипуляции (добавление или удаление) с книгами конкретного автора Пушкин. Во всех остальных случаях (просто добавление или удаление авторов; добавление и удаление книг других авторов) программа реагировать не будет. Таким образом, мы можем еще раз убедиться, насколько гибкие возможности по «перехвату» конкретных событий предоставляет нам программная среда Bold for Delphi.

Программирование вычисляемых атрибутов

Ранее в этой книге мы уже знакомились с вычисляемыми (derived) атрибутами (см. главу 5). Они являются аналогами вычисляемых полей таблиц в традиционной схеме разработки приложений баз данных. При работе с вычисляемыми атрибутами в среде Bold for Delphi очень удобно использовать возможности языка OCL для задания правил вычисления их значений. Однако на практике могут встретиться ситуации, когда возможностей OCL для этих целей будет не хватать. И тогда единственным способом решения подобных задач будет использование программного кода. К счастью, разработчики Bold for Delphi заранее предусмотрели такой вариант работы с вычисляемыми атрибутами. Далее мы рассмотрим, как практически реализуется программирование значений вычисляемых атрибутов. Почему эта тема рассматривается именно в этой главе? Как будет показано ниже, методы программирования вычисляемых атрибутов непосредственно связаны с рассматриваемыми в данной главе механизмами подписки. Необходимо также отметить, что использование программного кода для формирования значений вычисляемых атрибутов требует обязательной генерации кода (см. предыдущую главу).

Рассмотрим следующую задачу. Пусть для учета некоторых товаров в базе данных необходимо осуществлять автоматическую привязку момента ввода записи о каждом новом товаре к текущей дате и моменту времени. Для решения поставленной задачи создадим простой класс, содержащий наименование товара, и два вычисляемых атрибута для автоматического описания даты и времени (рис. 13.3). Как в данном случае сформировать выражения для вычисляемых атрибутов? Ведь в языке OCL отсутствуют операции для получения даты и времени.

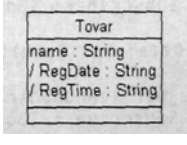


Рис. 13.3. Класс с вычисляемыми атрибутами

В этой ситуации есть выход — воспользоваться программным расчетом для получения значений вычисляемых атрибутов. Для этого сначала необходимо сгенерировать код модели (см. главу 12). При этом дополнительно формируется файл BusinessClasses.inc, содержащий программные заготовки процедур для вычисления значений текущих даты и времени (листинг 13.1).

Листинг 13.1. Код, генерируемый для вычисляемых атрибутов

```
(*******************************************************)
C                                                      •)
(•  Bold for Delphi Stub File                          *)
C                                                      •)
(*  Autogenerated file for method implementations      *)
(*                                                     *)
(*******************************************************)
//
{$INCLUDE BusinessClasses_Interface.inc}
procedure TTovar._RegDate_DeriveAndSubscribe
(DerivedObject: TObject; Subscriber: TBoldSubscriber);
//var
//  Result: String;
begin
   // Calculate value into Result and place the required
subscriptions
   // Result := <<formula>>
   // M_RegDate.AsString := Result;
end;
procedure TTovar._RegTime_DeriveAndSubscribe
(DerivedObject: TObject; Subscriber: TBoldSubscriber);
//var
//  Result: String;
begin
   // Calculate value into Result and place the required
subscriptions
   // Result := <<formula>>
   // M_RegTime.AsString := Result;
end;
```

Листинг даже снабжен необходимыми комментариями, помогающими разработчику написать программные выражения. В данном случае эти выражения будут выглядеть достаточно просто (листинг 13.2).

Листинг 13.2. Вычисление derived-атрибутов в программе

```
procedure TTovar._RegDate_DeriveAndSubscribe
(DerivedObject: TObject; Subscriber: TBoldSubscriber);
begin
   M_RegDate.AsString := DateToStr(Date);
end;
procedure TTovar._RegTime_DeriveAndSubscribe
(DerivedObject: TObject; Subscriber: TBoldSubscriber);
begin
   M_RegTime.AsString := TimeTostr(Time);
end;
```

Создадим простой графический интерфейс для проверки работоспособности приложения, и убедимся, что при вводе нового товара автоматически формируются дата и время ввода (рис. 13.4).

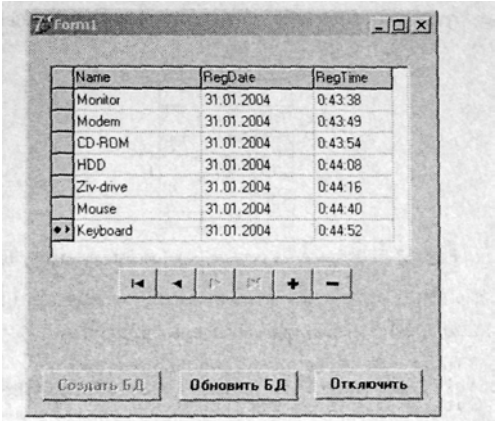


Рис. 13.4. Автоматическая регистрация момента ввода записи

Однако мы пока не полностью решили задачу, поскольку значения вычисляемых атрибутов не сохраняются на уровне данных. Для того чтобы в этом убедиться, просто закроем приложение и запустим его вновь. Мы увидим, что все ранее сформированные значения автоматически заменятся на текущие дату и момент времени запуска программы (рис. 13.5).

Рис. 13.5. Вычисляемые атрибуты не сохраняются в БД

Нетрудно усовершенствовать рассмотренное приложение с целью сохранения вычисленных значений. Покажем, как это сделать. Добавим в класс Tovar два обыч-

ных текстовых атрибута, которые будем использовать для хранения вычисляемых значений derived-атрибутов (рис. 13.6).

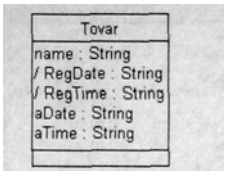


Рис. 13.6. Добавление постоянных (persistent) атрибутов

Немного изменим текст процедур вычисления полей (листинг 13.3).

Листинг 13.3. Сохранение значений вычисляемых атрибутов

```
procedure TTovar._RegDate_DeriveAndSubscribe
(DerivedObject: TObject; Subscriber: TBoldSubscriber);
begin
   M_RegDate.AsString := DateToStr(Date);
   if (aDate='') then M_aDate.AsString:=RegDate;
end;
procedure TTovar._RegTime_DeriveAndSubscribe
(DerivedObject: TObject; Subscriber: TBoldSubscriber);
begin
   M_RegTime.AsString := TimeTostr(Time);
   if (aTime='') then    M_aTime.AsString:=RegTime;
end;
```

Значения вычисляемых атрибутов присваиваются обычным атрибутам, если последние пусты (то есть принадлежат новому объекту). Если такой проверки не сделать, то, очевидно, предыдущая ситуация повторится. Отобразим для наглядности в сетке все атрибуты объекта и введем несколько значений (рис. 13.7).

Рис. 13.7. Ввод новых данных

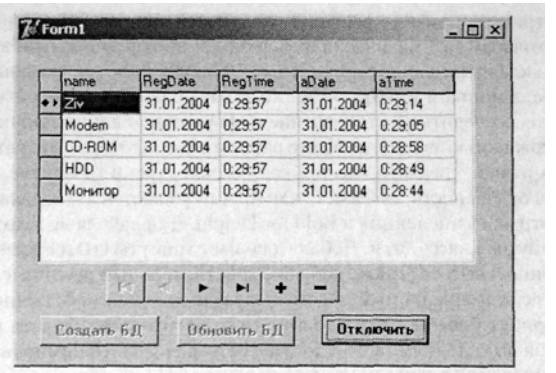


Рис. 13.8. Копии вычисляемых значений сохранились в БД

После этого обновим базу данных и перезапустим приложение (рис. 13.8).

Наглядно видно, что при повторном запуске вычисляемые атрибуты переписались заново, но прежняя информация сохранилась в их атрибутах-«двойниках». Таким образом, поставленную задачу можно считать решенной.

ПРИМЕЧАНИЕ

Описанный прием сохранения значений вычисляемых атрибутов в базе данных можно расширить и применять во многих случаях на практике, так как подобные задачи возникают довольно часто в реальных разработках.

Подводя итог, можно констатировать, что использование программного кода для формирования вычисляемых атрибутов существенно расширяет возможности их применения, позволяя реализовать функциональность, недоступную для OCL-выражений.

Обратно-вычисляемые атрибуты

Обратно-вычисляемые (reverse-derived) атрибуты являются уникальной особенностью среды Bold for Delphi. Обычные вычисляемые атрибуты формируют значения, недоступные для редактирования. И это вполне понятно. Например, можно использовать вычисляемый атрибут для расчета суммарной зарплаты сотрудников отдела. В этом случае «ручное» редактирование значения этого атрибута привело бы к неоднозначности — в самом деле, как из полученной общей суммы зарплат сотрудников можно «обратно вычислить» зарплату каждого? Ведь данный атрибут должен отображать именно сумму зарплат. Однако существуют ситуации, когда такие «обратные вычисления» все-таки возможны. Например, из полной строки «Иванов Иван Петрович», получаемой в результате «склейки» фамилии, имени и отчества, достаточно просто восстановить эти параметры. Таким обстоятельством и пользуется среда Bold for Delphi, предоставляя разработчику, где это возможно, использовать «обратные вычисления». Эту интересную возможность

мы рассмотрим на примере решения следующей задачи. Предположим, что база данных по товарам автоматически импортирует информацию о габаритах каждого товара из накладных, в виде строки типа «100x200x300», где указаны габариты товара (длина, ширина и высота в миллиметрах) с разделителем «x» между ними. Необходимо обеспечить восстановление информации о длине, ширине и высоте товара, с возможностью автоматического расчета объема товара в литрах, а также с возможностью ручного редактирования как полной строки габаритов, так и каждого параметра в отдельности. Покажем, как просто решается эта задача с использованием «обратных вычислений» в Bold for Delphi. Для начала необходимо уточнить состав атрибутов класса Tovar. Добавим целые атрибуты L, D и H, соответствующие длине, ширине и высоте. Также добавим действительный вычисляемый атрибут V (автоматически вычисляемый объем товара) и, наконец, собственно строку габаритов — атрибут Gabarits. Этот атрибут тоже необходимо сделать вычисляемым, поскольку он по условиям задачи должен автоматически формироваться при ручном редактировании длины, ширины или высоты. Полученную модель, состоящую из одного класса (рис. 13.9), импортируем в среду Bold for Delphi и запустим встроенный редактор моделей.

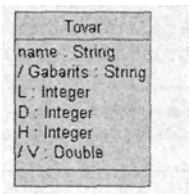


Рис. 13.9. Модифицированный класс

В редакторе моделей зададим признак Reverse derive у атрибута Gabarits (рис. 13.10).

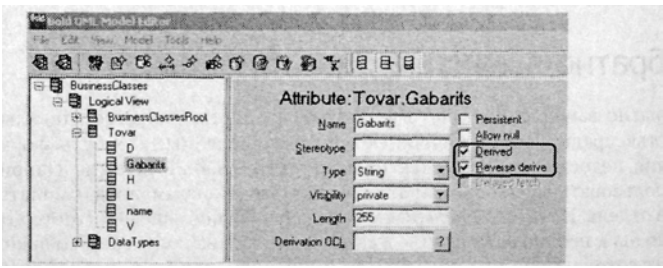


Рис. 13.10. Задание признака обратно-вычисляемого атрибута

Это позволит нам восстанавливать длину, ширину и высоту из полной строки габаритов, представленной эти атрибутом. Далее произведем операцию генерации кода. Образовавшийся в результате файл BusinessClasses.inc, как и ранее, будет содержать заготовки для реализации программных методов (листинг 13.4).

Листинг 13.4. Сгенерированный код с поддержкой обратно-вычисляемого атрибута

```
procedure TTovar._Gabarits_DeriveAndSubscribe(DerivedObject:
TObject;
 Subscriber: TBoldSubscriber);
//var
//  Result: String;
begin
   // Calculate value into Result and place the required
subscriptions
   // Result := <<formula>>
   // M_Gabarits.AsString := Result:
end;
procedure TTovar._Gabarits_ReverseDerive(DerivedObject: TObject);
begin
end;
procedure TTovar._V_DeriveAndSubscribe
(DerivedObject: TObject: Subscriber: TBoldSubscriber);
//var
//  Result: double;
begin
   // Calculate value into Result and place the required
subscriptions
   // Result := <<formula>>
   // M_V.AsFloat := Result;
end;
```

Только, в отличие от вышерассмотренных примеров, этот код также содержит и пустую программную заготовку для реализации обратных вычислений

```
procedure TTovar._Gabarits_ReverseDerive(DerivedObject: TObject);
```

Программирование вычислений начнем с обычных derived-атрибутов. Для процедуры расчета объема напишем следующий код:

```
procedure TTovar._V_DeriveAndSubscribe
(DerivedObject: TObject; Subscriber: TBoldSubscriber) ;
begin
   M_V.AsFloat := L/100.0*D/100.0*H/100.0;
   M_L.DefaultSubscribe(Subscriber);
   M_D.DefaultSubscribe(Subscriber);
   M_H.DefaultSubscribe(Subscriber);
end;
```

В первой строке кода вычисляется объем товара в литрах. Следующие строки иллюстрируют использование механизма подписки, в данном случае вызывается метод DefaultSubscribe для каждого из атрибутов, участвующих в вычислении объема. Это делается для того, чтобы при любом изменении длины, ширины или высоты подписчик (Subscriber) «узнал» об этом событии и пересчитал объем. Похожую процедуру напишем и для «склеивания» габаритов с разделителем «x»:

```
procedure TTovar._Gabarits_DeriveAndSubscribe
(DerivedObject: TObject; Subscriber: TBoldSubscriber);
begin
```

```
    M_Gabarits.AsString:=IntToStr(L)+'x'+IntToStr(D)+'x'+
      IntToStr(H);
    M_L.DefaultSubscribe(Subscriber);
    M_D.DefaultSubscribe(Subscriber);
    M_H.DefaultSubscribe(Subscriber);
end;
```

В этой процедуре мы снова «подписываемся» на изменение параметров L, D, H для обновления, при необходимости, склеенной строки-атрибута Gabarits. И, наконец, напишем код для «обратного вычисления» длины, ширины и высоты по строке габаритов. Этот код выделен в процедуру DecodeGabarits, получающую на входе строку sth и возвращающую три целых восстановленных параметра — oL, oD, oH, которые и являются длиной, шириной и высотой. Отметим, что работа этой процедуры не зависит от используемого разделителя между габаритами (главное, чтобы он не являлся цифрой). Текст процедуры вполне прозрачен и приведен ниже:

```
Procedure DecodeGabarits(sth :string; var oL,oD,oH : Integer);
var st:string;
    j : word;
begin
    if sth='' then exit;
    st:=trim(sth);
    for j:=1to length(st) do if NOT (st[j] in ['0'..'9']) then break;
    oL:=strtoint(Copy(st,1,j-1));
    Delete(st,1,j);
    for j:=I to length(st) do if NOT (st[j] in ['0'..'9']) then break:
    oD:=strtoint(Copy(st,1,j-1));
    Delete(st,1,j);
    oH:=strtoint(st);
end;
```

После этого нам осталось вызвать данную процедуру в методе обратно-вычисляемого атрибута и присвоить возвращаемые значения атрибутам L, D, H:

```
procedure TTovar._Gabarits_ReverseDerive(DerivedObject: TObject);
var oL.oD.oH : integer;
begin
    DecodeGabarits(Gabarits,oL,oD,oH);
    L:=oL;
    D:=oD;
    H:=oH;
end;
```

Запустим приложение (рис. 13.11). При добавлении новой записи автоматически отображается «склеенная» строка габаритов с нулевыми начальными значениями. Теперь приложение предоставляет широкие возможности для работы. Можно вводить по отдельности длину, ширину или высоту, и после такого ввода автоматически будет пересчитан объем, а также «склеится» и отобразится новая строка габаритов. Это — функционирование в стиле обычных вычисляемых атрибутов. А можно поступить наоборот — непосредственно редактировать строку габаритов либо прямо в сетке BoldGrid, либо вызвав автоформу (рис. 13.12). После редактирования автоматически обновятся длина, ширина и высота и будет пересчитан объем товара.

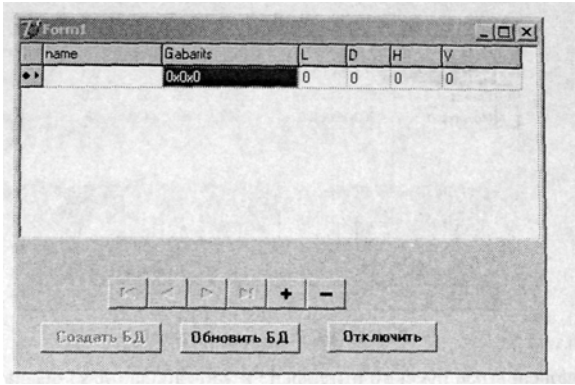


Рис. 13.11. Добавление новой записи

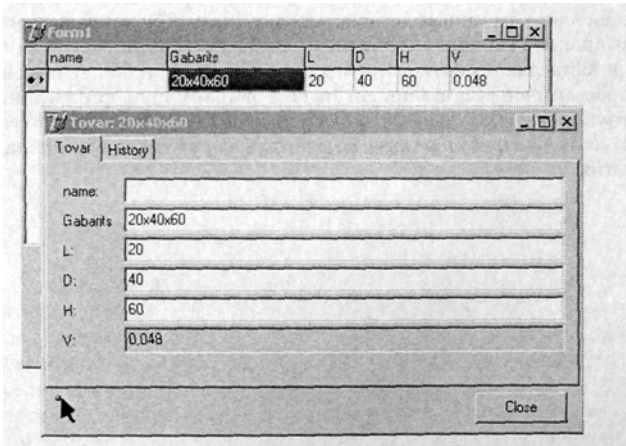


Рис. 13.12. Редактирование обратно-вычисляемого атрибута

Такой способ редактирования является качественно новым, и возможностью своей реализации он «обязан» обратно-вычисляемым атрибутам. В результате их использования мы полностью решили поставленную задачу (рис. 13.13), получив при этом приложение с очень гибким интерфейсом.

Резюме

Механизм подписки на события (subscribing) встроен в программную систему Bold for Delphi и активно используется самой средой для управления взаимодействием элементов в процессе работы приложения. Синхронизация уровней приложения

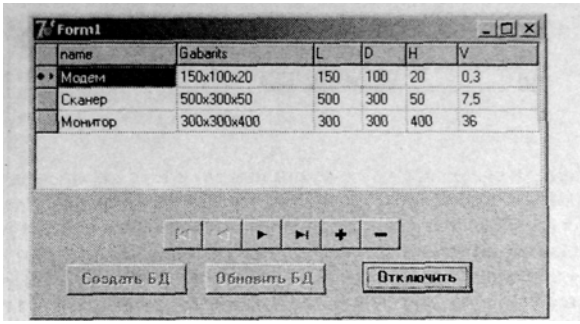


Рис. 13.13. Приложение в работе

(бизнес-уровень, графический интерфейс и уровень данных) обеспечивается механизмами подписки. Вычисление derived-атрибутов, оценка OCL-выражений также во многом обеспечиваются этими механизмами. Среда Bold for Delphi — активная событийно-организованная система, использующая собственный, изолированный от операционной системы механизм обмена сообщениями-событиями. Разработчику предоставляются возможности для формирования собственных видов событий и программной реализации подписки. Программное формирование вычисляемых атрибутов и уникальная возможность использования обратно-вычисляемых (reverse-derived) атрибутов также во многом базируются на механизмах подписки на события.

14 Обзор дополнительных возможностей

В этой главе представлены в виде краткого обзора дополнительные инструменты и возможности программной системы Bold for Delphi, не нашедшие отражения в предыдущих главах. При необходимости получения более подробной информации можно воспользоваться документацией и демонстрационными примерами, входящими в состав поставки продукта.

Регионы

Регионы (regions) используются для реализации блокировок на уровне элементов модели при многопользовательской работе. В практике разработки приложений баз данных нередко встречаются случаи, когда необходимо обеспечить защиту некоторой совокупности данных от одновременного изменения разными пользователями. Рассмотрим, например, ситуацию, когда в модели присутствуют моменты времени старта и окончания какого-нибудь процесса. Естественно, время окончания при этом должно превышать время старта. Но при многопользовательской работе возможны ситуации, когда один пользователь отредактирует момент старта, а другой, независимо от первого, — момент окончания таким образом, что время окончания станет меньше времени старта и логика данных нарушится. Для защиты от подобных ситуаций оба момента времени необходимо поместить в один регион. При этом действует следующее правило — элементы, принадлежащие одному региону, не могут одновременно изменяться несколькими пользователями. То есть все содержимое региона защищается (блокируется) от действий других пользователей, пока первый пользователь не прекратит сеанс работы с элементами данного региона. Элемент может входить и в несколько регионов. В этом случае действует следующее правило — для защиты элемента от изменения блокируются все регионы, в которые он входит.

Для реализации регионов Bold tor Delphi обеспечивает поддержку специального языка определения регионов RDL (Region Definition Language). Возможности RDL обеспечивают создание регионов, включение в состав региона классов и атрибутов, а также формирование *подрегионов* (несколько подрегионов могут входить в один регион). Использование подрегионов удобно для включения в них классов, соединенных ассоциациями с основным классом. В этом случае будет автоматически обеспечена защита ассоциации и значений ее ролей. Bold for Delphi по умолчанию создает регионы для классов модели при значении True тег-параметра модели GenerateDefaultRegions. Создаваемые при этом регионы по умолчанию формируются по следующим правилам:

1. Каждый класс формирует регион с именем Default, в который входят сам класс, все однократные (кратность роли не больше 1) связи с другими классами, а также все агрегатные мультиассоциации (кратность роли больше 1).
2. Каждый класс формирует регион с названием Exists, в который не входят члены класса (атрибуты или ассоциации). Данный регион обеспечивает защиту от удаления объектов данного класса.
3. Для каждой агрегатной связи с другим классом основной регион с именем Default включает аналогичный регион агрегируемого класса в качестве подрегиона.
4. Для каждой неагрегатной однократной связи формируется еще один регион с названием Default, включающий только саму ассоциацию, при этом аналогичный регион связанного класса входит в этот регион в качестве подрегиона.

Задание регионов и формирование их описаний на языке RDL осуществляется путем ввода тестовых RDL-объявлений в значение тег-параметра RegionDefinitions. Этот тег-параметр определен для модели в целом. Напомним (см. главу 4), что доступ к индивидуальному набору тег-параметров для конкретного элемента модели осуществляется командой меню встроенного редактора Tools > Edit tagged values. Настройку тег-параметров также можно выполнить и в Rational Rose (см. главу 4).

Жизненный цикл связанных объектов

Bold tor Delphi позволяет задать на уровне модели правила управления жизненным циклом существования объектов. Для этой цели используются свойства роли ассоциации Delete action и Aggregation. Эти свойства определяют, что будет происходить с ассоциированным («подчиненным») объектом при удалении главного объекта. Правила применения указанных свойств приведены в табл. 14.1.

Таблица 14.1. Свойства управления жизненным циклом

№	Значение свойства Delete action	Значение свойства Aggregation	Описание
1	Allow	Любое	Разрешено удалять главный объект без удаления «подчиненных» (нарушение целостности)
2	Prohibit	Любое	Запрещено удалять главный объект (будет выработано программное исключение)

№	Значение свойства Delete action	Значение свойства Aggregation	Описание
3	Cascade	Любое	Все связанные объекты автоматически уничтожатся при удалении главного
4	<Default>	None	Как в п. 1
5	<Default>	Aggregate	Как в п. 2
6	<Default>	Composite	Как в п. 3

Трехзвенная архитектура

Bold for Delphi позволяет реализовать приложения с многозвенной архитектурой. В этом случае каждое «звено» архитектуры может представлять соответствующий уровень программной системы — графический интерфейс, бизнес-уровень и уровень данных. На практике два последних уровня чаще всего размещаются на одном компьютере, в то время как графический интерфейс выносится на клиентские рабочие места, образуя так называемые «тонкие клиенты». Для организации таких приложений в Bold for Delphi присутствуют специальные компоненты TBoldComConnectionHandle и TBoldSystemandleCom. Последний из них является «представителем» системного дескриптора на стороне клиента, получая всю необходимую информацию от бизнес-уровня через DCOM-соединение. Особенностью создания многозвенных DCOM-приложений является использование специальных визуальных компонентов для формирования графического интерфейса а также специального набора дескрипторов объектного пространства клиента. При создании клиентского графического интерфейса таких приложений отсутствуют некоторые удобные возможности, например ограничено использование встроенного OCL-редактора для формирования навигационных OCL-выражений, и их приходится формировать вручную. Эти неудобства можно отчасти компенсировать, если воспользоваться поставляемой в составе продукта утилитой BoldClientifier. Она обеспечивает на уровне модификации исходных текстов проекта автоматическое преобразование приложения для функционирования в качестве тонкого клиента. Данную утилиту можно найти в папке ..\BoldSoft\BFDR40D7Arch\Tools\BoldClientifier.

Удаленное подключение к БД посредством SOAP

В рассматриваемой версии Bold for Delphi существует возможность создания удаленного уровня данных (remote persistence). Это означает физическое размещение бизнес-уровня вместе с графическим интерфейсом на одном компьютере, а базы данных вместе с обеспечивающими механизмами Persistence Layer — на удаленном компьютере. Для взаимодействия этих составных частей системы в данном случае применяется протокол SOAP (Simple Object Access Protocol), использующий в качестве «транспорта» обычный протокол HTTP. При этом удаленный уровень данных функционирует как обычный веб-сервер, и доступ к данным из БД может быть получен через интранет-сети и через глобальную сеть Интернет. Для организации описанной схемы на стороне клиента применяются специальные компоненты TBoldHTTPClientPersistenceHandle и TBoldWebConnection, а на стороне сервера -

TBoldHTTPServerPersistenceHandlePassthrough. Надо сказать, что программирование серверной части рассматриваемой схемы требует довольно большого объема работы. Однако это не является принципиальной помехой для реализации, и некоторые компании (см. главу 15) успешно используют данный механизм в своих программных продуктах.

Синхронизация объектных пространств

При работе многопользовательской системы в оперативной памяти каждого клиента формируется собственное объектное пространство. При этом вносимые отдельным пользователем в «свое» объектное пространство изменения по умолчанию никак не отражаются на объектных пространствах других пользователей. Такая ситуация может являться причиной возникновения многих сложностей и неудобств при эксплуатации информационных систем. В составе Bold for Delphi имеется специальное расширение среды — так называемый пропагатор (propagator) — инструмент для синхронизации ОП (OSS — Object Space Synchronization). Пропагатор устанавливается на выделенном компьютере и играет роль «сервера синхронизации», «прослушивая» состояние ОП каждого компьютера, «подписываясь» на изменения ОП, и осуществляя принудительную рассылку специальных оповещений в клиентские ОП с целью их синхронизации. Для задействования таких возможностей Bold for Delphi содержит набор специальных компонентов, которые можно найти на вкладке Bold OSS/CMS палитры компонентов Delphi.

Сервер управления блокировками

Вышеописанный механизм синхронизации ОП функционирует в неразрывной связи с механизмом управления блокировками (CMS — Concurrency Management Server). Последний основан на использовании регионов (см. начало этой главы), и обеспечивает реализацию «пессимистичных» блокировок на уровне множества одновременно функционирующих объектных пространств. Блокировки осуществляются автоматически при попытке изменить значение элемента ОП, причем отслеживаются как изменения пользователем, так и обращение к элементу из программного кода. Для использования сервера управления блокировками предназначены специальные компоненты, располагаемые на той же вкладке Bold OSS/CMS.

Object Lending Library (OLLE)

Еще одним расширением программной системы Bold for Delphi является библиотека OLLE, которая предназначена для репликации данных, содержащихся в нескольких БД, созданных из среды Bold for Delphi. При этом речь идет не столько о непосредственном копировании объектов, а скорее о предоставлении (Lending) прав доступа к «своему» объекту со стороны «чужой» базы данных. Такой предоставленный объект может использоваться базой данных только в режиме чтения, без возможности изменения его значения. Таким образом, несмотря на передачу

объекта в распоряжение другой БД, «хозяином» этого объекта, способным его изменить, остается его «родная» база данных. При изменении данного объекта уведомление об этом автоматически будет передано в те базы данных, которым он был предоставлен. Для реализации такой функциональности используются компоненты, которые находятся на вкладке BoldOLLE.

Эволюция модели и БД

На практике часто могут возникать ситуации, когда необходимо внести изменения в UML-модель (уточнение задачи, исправление ошибок моделирования, создание новых расширенных версий приложения и т. д.). Само по себе изменение модели реализуется просто, однако, как это часто бывает, к моменту возникновения такой потребности «старое» приложение уже эксплуатировалось в течение какого-то промежутка времени, и, соответственно, база данных, с которой работало это приложение, уже заполнена определенной информацией. При простой генерации новой структуры базы данных в соответствии с усовершенствованной моделью вся введенная ранее информация будет безвозвратно утеряна. Поэтому разработчики Bold создали специальные механизмы, использование которых позволяет во многих случаях относительно безболезненно решить подобные проблемы. Эти механизмы носят названия Model Evolution и Database Evolution. Использование механизма эволюции (развития или изменения) модели основано на следующем принципиальном подходе — разные версии модели могут функционировать с одной и той же базой данных. Этот подход реализуется благодаря способности Bold for Delphi автоматически генерировать схемы базы данных по UML-модели. В механизм эволюции модели входят специальные средства для преобразования старой модели и базы данных в их варианты, с которыми может работать как новое приложение, так и «старое». Для управления процессом преобразования модели и базы данных используется набор специальных тег-параметров. Например, тег-параметр FormerNames способен хранить несколько версий имен для конкретного элемента модели (класса, ассоциации и т. д.). Тег-параметр EvolutionState управляет использованием элементов в разных версиях. Так, например, при присвоении этому параметру какого-то класса значения Removed среда Bold for Delphi будет «игнорировать» существование этого класса, как будто он отсутствует в модели. Но при этом генерация в БД таблицы для этого класса не отменяется, что позволяет использовать такую базу старым приложениям. Применение тег-параметра ModelVersion, когда тег-параметр UseModelVersion (использовать версии модели) имеет значении True, приведет к автоматической генерации нового столбца таблицы БД, в котором каждому элементу будет автоматически поставлен в соответствие номер версии модели. Собственно эволюция базы данных после создания новой модели и ее импорта в среду Bold реализуется из встроенного редактора моделей с помощью команды Tools • Evolve Database. В результате начнется процесс перегенерации структуры базы данных с автоматической проверкой возможности сохранения информации в БД. При этом разработчику будет представлен подробный протокол всех изменений, вносимых в базу данных. Описанные механизмы, безусловно, весьма полезны при практической разработке.

Многоязыковая поддержка

Bold for Delphi обладает развитыми возможностями формирования многоязычных интерфейсов пользователя. Для представления языка предназначен класс TBALanguage, содержащий его описание. При этом каждый экземпляр данного класса представляет отдельный язык. В каждый момент времени система работает с единственным (текущим) языком, определяемым значением переменной CurrentLanguage. Метод BoldSetCurrentLanguageByName позволяет задать любой из имеющихся языков в качестве текущего. Тип TBAMLString служит для представления строковых элементов модели на разных языках. Он позволяет «подписаться» (см. главу 13) на замену текущего языка, чтобы автоматизировать соответствующее изменение пользовательского интерфейса. Компонент-рендерер (см. главу 9) TBoldAsMLStringRenderer также обеспечивает автоматизацию отображения многоязычных строковых элементов.

Средства отладки

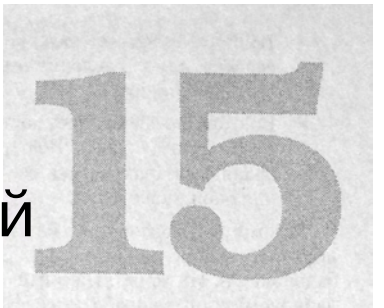
Отладка MDA-приложений, созданных в среде Bold for Delphi, имеет определенную специфику. Во-первых, интегрированный отладчик Delphi не способен реализовать отладку выражений OCL (напомним, что такие выражения можно формировать и активизировать непосредственно из программного кода приложения). Во-вторых, использование механизма OCL2SQL (см. главу 10) приводит к автоматической генерации множества SQL-запросов. И, в-третьих, в среде Bold генерируются специальные события (см. главу13). Разработчики Bold предоставили для целей отладки приложений специальную копию всех программных dcu-модулей, расположенную в папке ..\SLIB. По умолчанию при разработке задействуются программные модули из папки ..\LIB. Для получения возможности расширенной отладки разработчик должен заменить в настройках Delphi путь к используемой при компоновке программной библиотеке, указав вместо папки LIB папку SLIB. Кроме того, необходимо в свойствах проекта добавить признак BOLD_SLIB (командой меню Delphi: Delphi • Project ▸ Options • Directories/Conditionals). После этого необходимо перекомпилировать проект. В результате разработчик получает доступ к следующим инструментам:

- отладчик времени исполнения (класс TboldSystemDebuggerFrm);
- трассировщик OCL-выражений;
- трассировщик SQL-запросов;
- трассировщик обращений к БД (более высокоуровневый инструмент, чем трассировщик SQL), позволяет отслеживать конкретные выборки данных.

Резюме

Приведенный перечень дополнительных возможностей, предоставляемых разработчику средой Bold for Delphi и ее расширениями, позволяет существенно расширить область практического использования данного продукта. А использование развитых средств отладки обеспечивает повышение качества и эффективности разработки приложений.

Продукты сторонних производителей



Появление программного продукта Bold for Delphi (его первая версия вышла в 1998 году), обладающего множеством качественно новых и даже уникальных возможностей, не могло пройти незамеченным для фирм-производителей программного обеспечения. К настоящему моменту ряд компаний уже выпустило несколько версий программных продуктов, специально созданных для разработки в программной среде Bold for Delphi. В этой главе представлен краткий обзор таких продуктов.

BoldExpressStudio

Производитель — Neosight Technologies Limited (http://www.neosight.com).

Вид лицензии — коммерческий продукт. Существует пробная (trial) версия.

BoldExpress Studio является весьма интересным и многофункциональным программным продуктом. В действительности в его состав включено несколько самостоятельных разработок.

- BoldExpress XMLSerialisation — Model Driven XML Applications — обеспечивает трансляцию информации, содержащейся в UML-модели, в описание на языке XML. Кроме того, данный продукт содержит инструменты для использования предлагаемого разработчиком продукта языка OCL2XML, который позволяет реализовать «XML-запросы». Такие запросы довольно удобны для использования в интранете и Интернете.
- BoldExpress SOAP — Model Driven Web Services — предоставляет удобные возможности для использования XML веб-сервисов. Он может использоваться для автоматического преобразования обычных Bold-приложений в набор веб-сервисов.
- BoldExpress Security — обеспечивает защиту, назначение прав доступа пользователям и шифрование информации.

- Bold Express Server — реализует функциональность, подобную обычному веб-серверу, для управления и развертывания созданных приложений в сетях Интернет-интранет.
- BoldRetina — позволяет достаточно простым способом полностью заменить графический уровень Bold, преобразовав его в веб-интерфейс, при этом сохраняются практически все возможности по редактированию, формирования окон, меню и т. д.

Резюмируя, можно утверждать, что BoldExpress является очень удачным расширением и дополнением Bold для использования в корпоративных сетях и глобальной сети Интернет. На сайте компании-разработчика есть возможность в онлайновом режиме поработать с веб-приложением непосредственно через браузер (рис. 15.1).

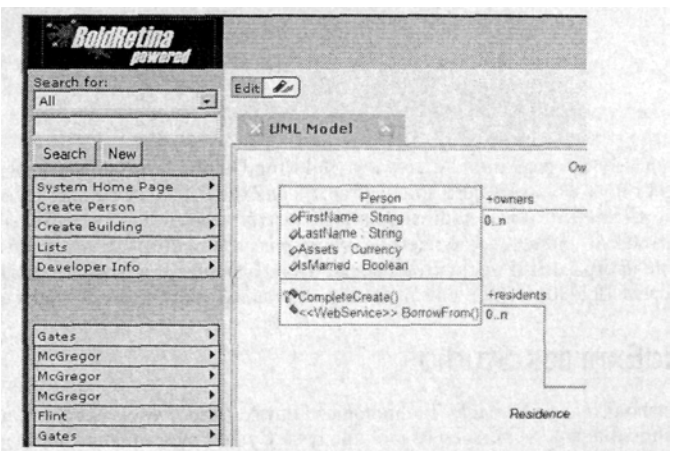


Рис. 15.1. Фрагмент демонстрационного онлайн-приложения

При этом можно добавлять и удалять объекты, изменять значения атрибутов и т. д.

BoldGridPro

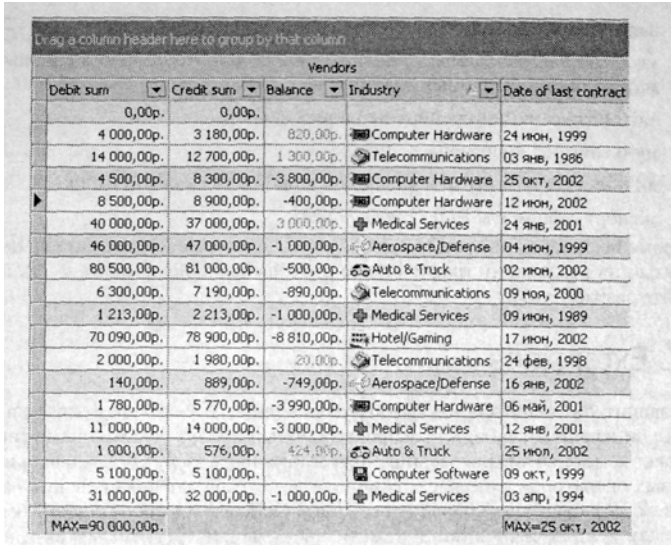
Производитель — Vipper Software (http://www.rad-studio.com).

Вид лицензии — коммерческий продукт. Существует пробная (trial) версия.

Компонент BoldGridPro представляет собой сетку для отображения данных с очень развитыми возможностями интерфейса (рис. 15.2).

BoldGridPro обеспечивает следующие возможности:

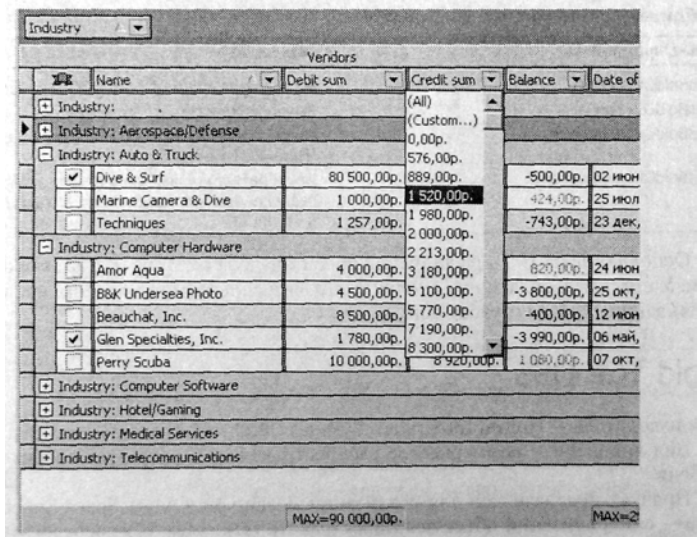
- многострочные заголовки, формируемые как при разработке, так и во время выполнения приложения;
- группировку, сортировку и фильтрацию записей (рис. 15.3);


Drag a column header here to group by that column

Vendors

Debit sum	Credit sum	Balance	Industry	Date of last contract
0,00р.	0,00р.			
4 000,00р.	3 180,00р.	820,00р.	Computer Hardware	24 июн, 1999
14 000,00р.	12 700,00р.	1 300,00р.	Telecommunications	03 янв, 1986
4 500,00р.	8 300,00р.	-3 800,00р.	Computer Hardware	25 окт, 2002
8 500,00р.	8 900,00р.	-400,00р.	Computer Hardware	12 июн, 2002
40 000,00р.	37 000,00р.	3 000,00р.	Medical Services	24 янв, 2001
46 000,00р.	47 000,00р.	-1 000,00р.	Aerospace/Defense	04 июн, 1999
80 500,00р.	81 000,00р.	-500,00р.	Auto & Truck	02 июн, 2002
6 300,00р.	7 190,00р.	-890,00р.	Telecommunications	09 ноя, 2000
1 213,00р.	2 213,00р.	-1 000,00р.	Medical Services	09 июн, 1989
70 090,00р.	78 900,00р.	-8 810,00р.	Hotel/Gaming	17 июн, 2002
2 000,00р.	1 980,00р.	20,00р.	Telecommunications	24 фев, 1998
140,00р.	889,00р.	-749,00р.	Aerospace/Defense	16 янв, 2002
1 780,00р.	5 770,00р.	-3 990,00р.	Computer Hardware	06 май, 2001
11 000,00р.	14 000,00р.	-3 000,00р.	Medical Services	12 янв, 2001
1 000,00р.	576,00р.	424,00р.	Auto & Truck	25 июл, 2002
5 100,00р.	5 100,00р.		Computer Software	09 окт, 1999
31 000,00р.	32 000,00р.	-1 000,00р.	Medical Services	03 апр, 1994
MAX=90 000,00р.				MAX=25 окт, 2002

Рис. 15.2. Сетка BoldGridPro


Industry

Vendors

Name	Debit sum	Credit sum	Balance	Date of
Industry:				
Industry: Aerospace/Defense				
Industry: Auto & Truck				
Dive & Surf	80 500,00р.		-500,00р.	02 июн
Marine Camera & Dive	1 000,00р.		424,00р.	25 июл
Techniques	1 257,00р.		-743,00р.	23 дек,
Industry: Computer Hardware				
Amor Aqua	4 000,00р.		820,00р.	24 июн
B&K Undersea Photo	4 500,00р.		-3 800,00р.	25 окт,
Beauchat, Inc.	8 500,00р.		-400,00р.	12 июн
Glen Specialties, Inc.	1 780,00р.		-3 990,00р.	06 май,
Perry Scuba	10 000,00р.		1 080,00р.	07 окт,
Industry: Computer Software				
Industry: Hotel/Gaming				
Industry: Medical Services				
Industry: Telecommunications				
MAX=90 000,00р.				MAX=2

(All)
(Custom...)
0,00р.
576,00р.
889,00р.
1 520,00р.
1 980,00р.
2 000,00р.
2 213,00р.
3 180,00р.
5 100,00р.
5 770,00р.
7 190,00р.
8 300,00р.

Рис. 15.3. Группировка и фильтрация

- запоминание настроек во время исполнения приложения;
- удобные настраиваемые панели — footers, для отображения сводных автоматически вычисляемых величин;
- автоматический инкрементный поиск записи;
- мультивыбор диапазона записей;
- индивидуальная настройка цвета и шрифта для каждой ячейки;
- экспорт в форматы PDF, Excel, HNML.

Кроме BoldGridPro, компания предлагает и некоторые другие визуальные компоненты для реализации графического интерфейса. Информация о них имеется на сайте разработчика.

OCL Extensions

Производитель — Holton Integration Systems (http://www.holtonsystems.com).

Вид лицензии — свободно-распространяемый продукт с исходными текстами.

Пакет компонентов для расширения состава операторов OCL. Содержит около 80 новых операторов, которые можно использовать для формирования OCL-выражений. После установки пакета все новые OCL-выражения становятся доступными для использования как в режиме выполнения программы, так и на этапе разработки во встроенном OCL-редакторе Bold for Delphi. Некоторые из предоставляемых дополнительных возможностей описаны ниже (табл. 15.1).

Таблица 15.1. Некоторые дополнительные операторы OCL

OCL-выражение	Описание
THisBoldOCLBoldID	Возвращает идентификатор объекта BoldID
THisBoldOCLSqrt	Вычисление квадратного корня
THisBoldOCLasInteger	Преобразует логическое значение в целое (False=0, True=1)
THisBoldOCLasCommaList	Преобразует коллекцию в список значений, разделенных «;». Удобно для экспорта в StringList

Основная часть OCL-расширений предназначена для работы с датами и временем. Учитывая бесплатность продукта, его, безусловно, можно рекомендовать для скачивания с сайта разработчика.

Bold TCP OSS

Производитель — Holton Integration Systems (http://www.holtonsystems.com).

Вид лицензии — коммерческое распространение. Существует пробная (trial) версия.

Продукт предназначен для реализации механизма Object Space Synchronization — синхронизации объектных пространств (см. главу 14) посредством TCP-протокола.

Bold SOAP Server (BSS)

Производитель — sourceforge (http://sourceforge.net/projects/boldsoapserver/default.htm).

Вид лицензии — свободно распространямый продукт с исходными текстами.

BSS — программная оболочка (wrapper) вокруг BoldSystem, обеспечивающая веб-доступ к Bold-приложению (см. также BoldExpress Studio в начале главы).

BoldRave

Производитель — Intrastinq (http://intrasting.com/boldrave/).

Вид лицензии — свободно распространяемый продукт с исходными текстами.

Пакет компонентов обеспечивает непосредственную генерацию отчетов Nevrona Rave из BoBI-приложений, без необходимости использования программных переходников типа TBoldDataSet (см. главу 11). Поддерживает дескрипторы переменных ОП, обладает богатыми возможностями настройки вида и содержания отчета, в том числе и во время работы приложения, а также обеспечивает предварительный просмотр перед печатью (рис. 15.4).

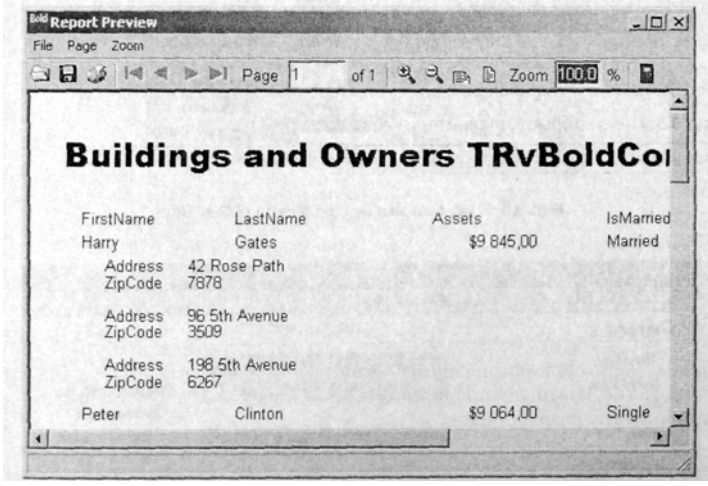


Рис. 15.4. Окно предварительного просмотра отчета (BoldRave)

deBold

Производитель — DroopyEyes (http://www.droopyeyes.com).

Вид лицензии — свободно распространяемый продукт с исходными текстами.

Пакет из нескольких полезных компонентов, предназначенных для проверки корректности состояния объектов и дескрипторов ОП, ограничении (constraints) для объектов и форм приложения в целом. Также включает довольно интересные компоненты TdeBoidAutoIncManager и TdeBoldAutoIncProvider для реализации пользовательских автоинкрементных атрибутов.

pHGantTimePackage и phGrid_BA

Производитель — PlexityHide (http://www.plexityhide.com).

Вид лицензии — коммерческое распространение. Существует пробная (trial) версия.

phGantTimePackage — пакет компонентов для автоматизированного построения диаграмм, временных шкал, расписаний и графиков. Богатые возможности настройки интерфейса (рис. 15.5).

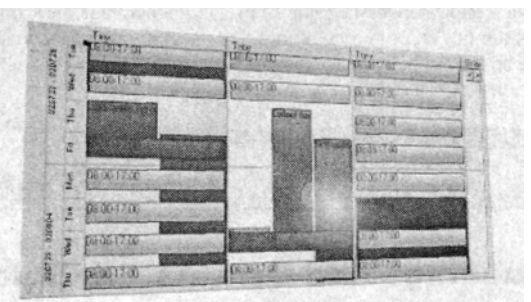


Рис. 15.5. Использование компонента phGantTime

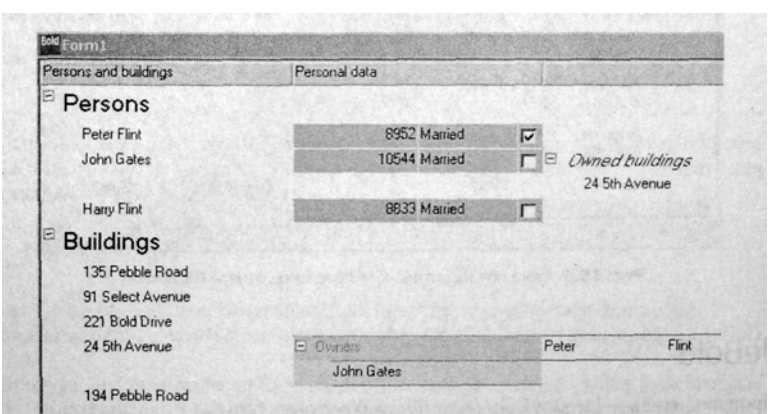


Рис. 15.6. Сетка phGrid_BA

phGridBA — сетка с развитыми возможностями отображения данных (рис. 15.6). Например, позволяет включать вложенные сетки в ячейки основной сетки.

Резюме

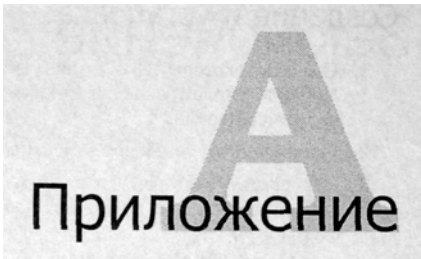
Из представленного краткого обзора можно сделать вывод, что разработанные сторонними производителями программные расширения и дополнения для Bold for Delphi довольно многочисленны и реализуют весьма разнообразные и полезные дополнительные возможности для создания приложений в среде Borland MDA.

Заключение

В этой книге читатель начал знакомство с MDA-архитектурой разработки приложений XXI века на примере ее замечательного представителя — программного продукта Bold for Delphi компании Borland. Именно только начал знакомство, потому что в одной книге совершенно нереально описать все возможности и внутреннее устройство такого сложного и объемного инструмента. Многие интересные вопросы остались «за кадром» (регионы, синхронизация объектных пространств, DCOM-приложения и др.), часть вопросов хотелось бы обсудить более детально, например, «внутренний мир» механизмов подписки, отладку Bold-приложений. Также совершенно незаслуженно обойден вниманием и инструмент моделирования ModelMaker. Но увы: как сказал известный наш соотечественник, «нельзя объять необъятное». И на этом автору пока приходится остановиться. Однако это не означает, что читателю тоже надо останавливаться, наоборот, автор рассчитывает, что данная книга сыграет роль «стартовой площадки» для многих разработчиков, поможет им на начальном этапе освоения качественно новой технологии разработки приложений баз данных в Delphi. Автор надеется, что хоть в какой-то степени ему удалось заинтересовать опытных читателей-разработчиков, чуть-чуть убедить сомневающихся и немного помочь начинающим. И всем вам, дорогие читатели этой книги, автор искренне желает успехов, радости преодоления трудностей и того особенного творческого настроя, который возникает при работе с замечательным программным продуктом Bold for Delphi.

С уважением,

Константин Грибачев



Скрипт для транслитерации имен UML-модели

В этом приложении приводится описание и исходный текст скрипта транслитерации. Он обеспечивает преобразование тег-параметров русскоязычных идентификаторов UML-модели, разрабатываемой в редакторе диаграммы классов Rational Rose, для дальнейшего использования этой модели в программной среде Bold for Delphi.

Особенности

Отметим некоторые особенности этого скрипта.

- Скрипт не изменяет UML-идентификаторы (названия классов, атрибутов, ролей), модифицируются только соответствующие тег-параметры для Bold. Это позволяет после применения скрипта продолжать работать с внешне «русскоязычной» UML-моделью в Rational Rose.
- Все пробелы в именах заменяются на символ подчеркивания.
- Англоязычные имена не изменяются.
- Имена классов ассоциаций формируются как сумма строк «Link»+<имя роли 1>+<имя роли 2>. Например, если в какой-то ассоциации используются названия ролей «авторы» и «книги», то после работы скрипта будут сформированы тег-параметры вида «Linkavtoryknigi» для использования идентификатора ассоциации в среде Bold for Delphi.

Создание и установка:

1. Создать текстовый файл и ввести текст скрипта (листинг А.1), используя любой текстовой редактор. Сохранить полученный файл с расширением .ebs, например translit.ebs.
2. Файл скрипта translit.ebs поместить в папку ..\Rational\Rose\Scripts.
3. Отредактировать файл главного меню Rational Rose ..\Rational\Rose\rose.mnu, добавив следующие строки в конце этого файла (перед последней фигурной скобкой «}»):

```
option "Преобразование для Bold..."
{
enable %model:writeable
RoseScript $SCRIPT_PATH\SCRIPTS\translit.ebs
```

Использование

Для запуска скрипта, перед сохранением модели необходимо в Rational Rose вызвать команду главного меню Tools ▸ Преобразование для Bold... (рис. АЛ).

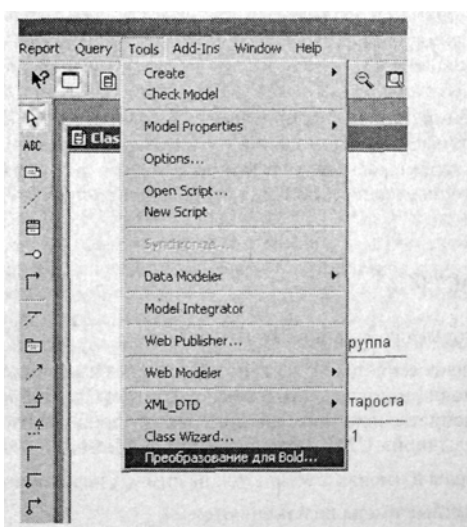


Рис. А.1. Вызов скрипта для транслитерации

После окончания работы скрипта на экран будет выведено окно (рис. А.2). Необходимо нажать кнопку OK для выхода из скрипта.

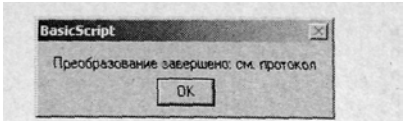


Рис. А.2. ОКНО сообщения о завершении транслитерации

Протокол работы скрипта отобразится в окне протокола Rational Rose (рис. А.3).

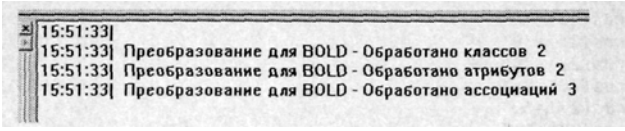


Рис. А.3. Протокол работы скрипта

ВНИМАНИЕ

Еще раз стоит напомнить, что в процессе транслитерации скрипт модифицирует **только тег-параметры** модели, отвечающие за имена типов Delphi и т. п., что необходимо для использования модели в Bold for Delphi. Имена классов, атрибутов и ролей самой UML-модели не модифицируются, поэтому после работы скрипта внешний вид самой модели не изменится. Но при этом с такой моделью сможет работать Bold.

После окончания работы скрипта модель можно импортировать в среду Bold for Delphi.

Исходный текст скрипта

В листинге А. 1 приведен исходный текст скрипта на языке BasicScript. При желании его можно свободно модифицировать, оптимизировать и адаптировать под собственные требования.

Листинг АЛ. Пример скрипта для транслитерации

```
Function transl (srus As String) As String
Dim brus(63),beng(63),s,seng As String
   brus(0)="а"
   brus(1)="б"
   brus(2)="в"
   brus(3)="г"
   brus(4)="д"
   brus(5)="е"
   brus(6)="ж"
   brus(7)="3"
   brus(8)="и"
   brus(9)="ii"
   brus(11)="к"
   brus(12)="л"
   brus(13)="м"
```

```
brus(14)="н"
brus(15)="о"
brus(16)="п"
brus(17)="р"
brus(18)="с"
brus(19)="т"
brus(20)="у"
brus(21)="ф"
brus(22)="х"
brus(23)="ц"
brus(24)="ч"
brus(25)="ь"
brus(26) = 'V
brus(27)="ы"
brus(28)="ш"
brus(29)="щ"
brus(30)="э"
brus(31)="ю"
brus(32) = 'V
brus(33)="А"
brus(34)="Б"
brus(35)="В"
brus(36)="Г"
brus(37)="Д"
brus(38)="Е"
brus(39)="Ж"
brus(40)="З"
brus(41)="И"
brus(42)="К"
brus(43)="П"
brus(44)="М"
brus(45)="Н"
brus(46)="О"
brus(47)="П"
brus(48)="Р"
brus(49)="С"
brus(50)="Т"
brus(51)="У"
brus(52)="Ф"
brus(53)="Х"
brus(54)="Ц"
brus(55)="Ч"
brus(56)="Ш"
brus(57)="Щ"
brus(58)="Э"
brus(59)="Ю"
brus(60)="Я"
brus(61)="ё"
brus(62)="Ё"


beng(0)="a"
beng(1)="b"
```

```
beng(2)="v"
beng(3)="g"
beng(4)="d"
beng(5)="e"
beng(6)="g"
beng(7)="z"
beng(8)="i"
beng(9)="i"
beng(11)='k"
beng(12)="l"
beng(13)="k"
beng(14)="n"
beng(15)="o"
beng(16)="p"
beng(17)="r"
beng(18)="b"
beng(19)="t"
beng(20)="u"
beng(21)="f"
beng(22)="h"
beng(23)="c"
beng(24)="ch"
beng(25)="j"
beng(26)="j"
beng(27)="y"
beng(28)="sh"
beng(29)="sh"
beng(33)="e"
beng(31)="yu"
beng(32)="ya"
beng(33)="A"
beng(34)="B"
beng(35)="V"
beng(36)="G"
beng(37)="D"
beng(38)="E"
beng(39)-"G"
beng(40)="Z"
beng(4])="I"
beng(42)="K"
beng(43)="L"
beng(44)="M"
beng(45)="N"
beng(45)="O"
beng(47)="P"
beng(48)="R"
beng(49)="S"
beng(50)="T"
beng(51)="U"
beng(52)="F"
beng(53)=H"
beng(54)="C"
```

```
    beng(55)="Ch"
    beng(56)="Sh"
    beng(57)="Sh"
    beng(58)="E"
    beng(59)="Yu"
    beng(60)="Ya"
    beng(61)="yo"
    beng(62)="Yo"
    Dim bukva As String

seng=""                         'пустая заготовка для строки
For i=1 To Len(srus)
bukva=""
s=Mid(srus,i,1)      'символ с позиции i
For j=0 To 62
If s=brus(j) Then bukva=beng(j) ' найдена русская буква
Next j
If bukva="" Then bukva=s      ' оставляем английскую букву
If s=" "    Then bukva="_"  ' заменяем пробел
If s="-"    Then bukva="_"  ' заменяем тире
seng=seng+bukva             ' накапливаем строку
Next i
Transl=seng
End Function

Sub TranslitName(theModel As Model)
Dim Cl   As Class
Dim Attr  As Attribute
Dim Ass  As Association

nclas=0
nattr=0
nass=0
'Цикл по всем классам
For I=1 To theModel.GetAllClasses().Count
nclas=nclas+1
Set Cl=theModel.GetAllClasses().GetAt(i)
b=Cl.OverrideProperty("Bold", "Delphi Name",
 "T"+Transl(Cl.name))
b=Cl.OverrideProperty("Bold", "ExpressionName",
 Transl(Cl.name))
b=Cl.OverrideProperty("Bold", "TableName",
 Transl(Cl.name))
b=Cl.OverrideProperty("Bold", "InterfaceName",
"I"+ Transl(Cl.name))
'Цикл по всем атрибутам
For J = 1 To Cl.Attributes.count
nattr=nattr+1
Set Attr =Cl.Attributes.GetAt(J)
b=attr.overrideproperty("Bold","ColumnName",
Transl(Attr.name))
b=attr.OverrideProperty("Bold", "ExpressionName",
```

```
Transl(Attr.name))
b=attr.OverrideProperty("Bold", "DelphiName",
Transl(Attr.name))
Next J    'следующий атрибут
Next I    'следующий класс

'Цикл по всем ассоциациям
For K=1 To theModel.GetAllAssociations().Count
Set Ass=theModel.GetAllAssociations().GetAt(K)
nass=nass+1
' атрибуты ассоциации
if Ass.name<>"" then  b=Ass.OverrideProperty("Bold",
              "LinkClassName","Link"+Transl(Ass.name))
if Ass.name="" then  b=Ass.OverrideProperty
("Bold","LinkClassName","Link"+Transl(Ass.Role1.name)
+Transl(Ass.Role2.name)) ' роли ассоциации
b=Ass.Role1.OverrideProperty("Bold","ColumnName",
Transl(Ass.Role1.Name))
b=Ass.Role1.OverrideProperty
("Bold", ExpressionName",Transl(Ass.Role1.Name))
b=Ass.Role1.OverrideProperty("Bold", "DelphiName",
Transl(Ass.Role1.Name))
b=Ass.Role2.OverrideProperty("Bold","ColumnName",
Transl(Ass.Role2.Name))
b=Ass.Role2.OverrideProperty
("Bold","ExpressionName",Transl(Ass.Role2.Name))
b=Ass.Role2.OverrideProperty("Bold", "DelphiName",
Transl(Ass.Role2.Name))
Next K 'следующая ассоциация
roseapp.WriteErrorLog " "
RoseApp.WriteErrorLog "Преобразование для BOLD
- Обработано классов  " & nclas
RoseApp.WriteErrorLog "Преобразование для BOLD
- Обработано атрибутов  " & nattr
RoseApp.WriteErrorLog "Преобразование для BOLD
- Обработано ассоциаций  " & nass
End Sub
Sub Main
' транслитерация имен
TranslitName(RoseApp.CurrentModel)
MsgBox "Преобразование завершено: см. протокол"
End Sub
```

Приложение Б

ECO — развитие Borland MDA для платформы Microsoft .NET

В этом приложении представлен краткий обзор основных возможностей технологии ECO, являющейся развитием Borland MDA, на примере программного продукта Borland C# Builder Architect. По ходу обзора будут отмечаться особенности и различия ECO в сравнении с продуктом Bold for Delphi. В качестве иллюстрации возможностей ECO будет описан процесс создания простого приложения в C#Builder Architect.

О продукте C#Builder Architect

В конце сентября 2003 года вышла новая версия инструментальной среды разработки Borland для платформы Microsoft .NET — Borland C#BuiIder Architect. Принципиальным отличием версии Architect от ранее вышедших версий продукта Borland C#Builderявляется присутствие в ней средств разработки MDA-приложений. Важнейшим из них является интегрированная в Borland C#Builder Architect технология ECO — Enterprise Core Objects. Читатели этой книги, познакомившиеся с технологией Bold for Delphi, обнаружат в ECO очень много общего с Bold. И это не случайно, поскольку ECO базируется на технологии Bold, и можно сказать, что ECO — это своеобразный «Bold for .NET» (конечно, существенно переработанный). С целью полной интеграции MDA-инструментов в среду разработки компания Borland сделала еще один весьма логичный шаг — в состав Borland C#Builder Architect входит полнофункциональный редактор UML-моделей, созданный на базе продуктов компании TogetherSoft (см. также главу 2). Поэтому сейчас уже

нет необходимости привлекать сторонние графические UML-редакторы типа Rational Rose для разработки модели MDA-приложений. Благодаря этому Borland C#Builder Architect стал первым продуктом Borland, обеспечивающим полный жизненный цикл разработки приложений — от создания модели до генерации кода. Пробную 30-дневную версию продукта можно свободно скачать с сайта разработчика.

Основные возможности ECO

В полном соответствии с идеологией Borland MDA, ECO предоставляет в распоряжение разработчика широкий спектр инструментальных возможностей для создания MDA-приложений, включая:

1) разработку UML-модели во встроенном графическом UML-редакторе;
2) дополнительную настройку модели с использованием тег-параметров;
3) формирование бизнес-уровня приложения с использованием набора специальных компонентов-дескрипторов объектного пространства;
4) генерацию структуры базы данных в соответствии с UML-моделью;
5) создание графического интерфейса приложения, с привязкой интерфейсных элементов к бизнес-уровню;
6) использование языка OCL для навигации по модели, формирования OCL-запросов к объектному пространству и реализации вычисляемых атрибутов;
7) реализацию механизмов подписки (subscribing);
8) возможность использования XML-документов в качестве уровня данных, вместо СУБД;
9) импорт UML-моделей, созданных в других редакторах, из файлов в формате XML.

Создание простого ECO-приложения

Для демонстрации возможностей и специфики технологии ECO создадим простое приложение в среде C#Builder Architect. В процессе работы над ним будет легче понять специфику продукта а также его отличия от Bold, на некоторых таких моментах мы будем заострять внимание читателя.

Генерация шаблона

В начале необходимо создать заготовку-шаблон будущего ECO-приложения. выбрав в главном меню C#Builder Architect команду File ▸ New ▸ Other. В появившемся окне щелкнем по значку ECO Application. Будет предложено ввести имя проекта, назовем его Lib. После этого будут сгенерированы файлы, содержащие программные заготовки для классов нового приложения. Состав имеющихся файлов нового проекта отображается в менеджере проекта (рис. Б.1).

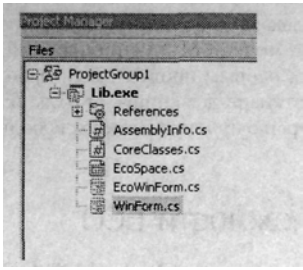


Рис. Б.1. Классы проекта в менеджере проектов

Рассмотрим назначение этих файлов.

- CoreClasses.cs содержит (пока пустой) набор программных реализаций UML-классов модели, образующих в совокупности UML-пакет. Таких пакетов в одном приложении может быть несколько.
- EcoSpace.cs представляет объектное пространство приложения, которое в ECO носит название «EcoSpace». В этом файле содержится реализация объектов, доступных во время выполнения приложения.
- EcoWinForms.cs представляет реализацию окон приложения (WinForm). В реализацию этих классов включены специальные компоненты, обеспечивающие возможности взаимодействия с EcoSpace.
- WinForms.cs — реализация обычных окон WinForm.

Разработка UML-модели

Для разработки UML-модели C#Builder Architect оснащен собственным графическим (в отличие от текстового встроенного редактора Bold) UML-редактором. Для его отображения выберем вкладку Model View и дважды щелкнем на строке CoreClassesPackage (рис. Б.2).

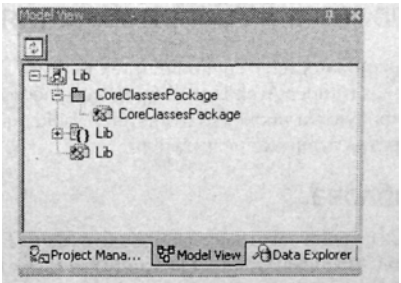


Рис. Б.2. Окно структуры модели

Откроется окно встроенного UML-редактора, и одновременно с этим отобразится панель UML-элементов (рис. Б.3), предназначенных для формирования мо-

дели. Мы видим, что в этой версии продукта их не так много, как, например, в Rational Rose. Например, поддерживается всего два вида связей — ассоциация и обобщение.

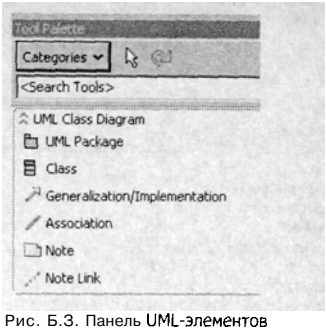


Рис. Б.3. Панель UML-элементов

Используя эти UML-элементы, создадим простую модель, состоящую из двух классов и одной ассоциации (рис. Б.4). В качестве моделируемой области возьмем уже не раз рассмотренный в этой книге библиотечный каталог (см. главу 3), содержащий авторов и книг. Принципы работы в UML-редакторе достаточно просты и не требуют подробных разъяснений.

Рис. Б.4. UML-модель

Присвоим первому классу название Avtor и настроим его свойства в инспекторе объектов, как показано на рис. Б.5. Видно, что многие из настраиваемых свойств совпадают по названию и назначению с соответствующими тег-параметрами Bold. Однако присутствуют и некоторые новые свойства, например Alias, позволяет назначить классу русскоязычный псевдоним, отображаемый в UML-редакторе (см. рис. Б.4). Название группы свойств Borland.Studio.Together.EcoClassTaggedValues говорит о том, что UML-редактор и включенные в него средства обеспечения интеграции являются частью разработки компании TogetherSoft.

Создадим второй класс Kniga с псевдонимом Книга. Зададим ассоциацию AvtorKniga и настроим ее роли napisal и napisana, как показано на рис. Б.6. Кратности обеих ролей установим как 1..* («многие-ко-многим»). Очевидна схожесть многих свойств ассоциации с аналогичными параметрами, применяемым в Bold (см. главу 4). Мы можем задавать «вычисляемость» (derived), «встраиваемость» (embed), упорядочение (ordered) и способ удаления связанных объектов (delete action).

ПРИМЕЧАНИЕ

Данная версия продукта на момент написания книги не поддерживала создание вычисляемых ассоциаций.

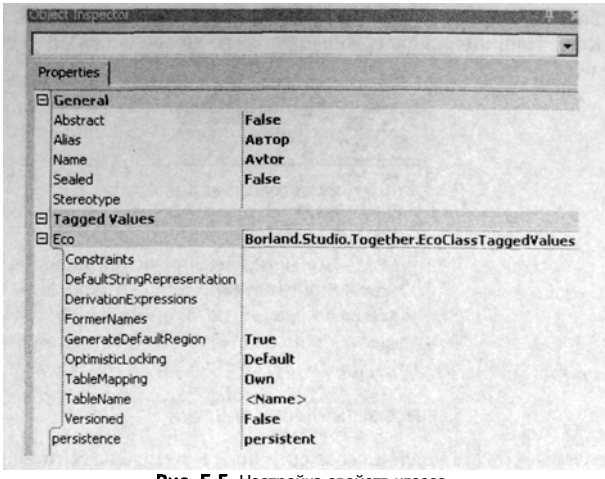


Рис. Б.5. Настройка свойств класса

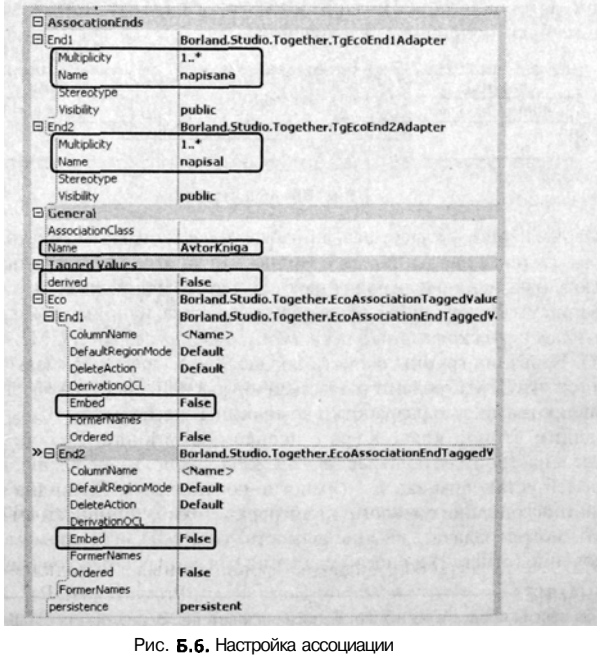


Рис. **Б.6.** Настройка ассоциации

Формирование бизнес-уровня

После формирования UML-модели необходимо перекомпилировать (build) проект. Это является отличительной особенностью продукта C#Builder Architect. Поскольку информация о модели, по крайней мере, ее большая часть, сохраняется в коде, то необходимо обеспечить синхронизацию кода с изменившейся моделью.

ПРИМЕЧАНИЕ

Напомним, что Bold for Delphi сохраняет информацию о модели не в коде, а в компоненте TBoldModel.

После перекомпиляции перейдем в менеджер проекта и щелкнем дважды по файлу EcoWinForms.cs. При этом откроется окно с пустой формой-заготовкой, в нижней части которой отображаются элементы, формирующие бизнес-уровень. Для этой цели нам предлагается набор компонентов ECO, расположенный на инструментальной панели (рис. Б.7) в группе Enterprise Core Objects.

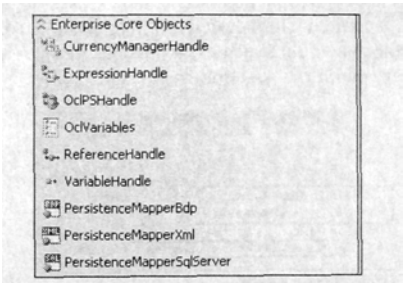


Рис. Б.7. Компоненты ECO

В набор входят следующие компоненты ECO.

- CurrencyManagerHandle — дескриптор-указатель на текущий объект другого дескриптора, к которому подключен этот компонент. Замена дескриптора списка в Bold (см. далее).
- ExpressionHandle - аналог дескриптора OCL-выражения (BoldExpressionHandle, см. главу 7).
- OclPSHandle — аналог дескриптора SQL(BoldSQLHandle, см. главу 10).
- OclVariables - контейнер ОП-переменных, аналогBoldOCLVariables (см. главу 7).
- ReferenceHandle — аналог дескриптора ссылки (BoldReferenceHandle, см. главу 7).
- VariableHandle - аналог дескриптора ОП-переменной (BoldVarableHandle, см. главу 7).
- PersistenceMapperBdp — адаптер уровня данных для подключения СУБД посредством инструментария BorlandDataProvider (BDP). В Bold аналогом является компонент BoldPersistenceHandleDB (см. главу 10) в совокупности с конкретным адаптером СУБД (например, BoldDataBaseAdapterBDE).

- PersistenceMapperXml — адаптер уровня данных для сохранения информации в XML-документах (см. главу 10).
- PersistenceMapperSqlServer — адаптер уровня данных для работы с СУБД Microsoft SQL Server.

Таким образом, мы имеем набор компонентов для формирования объектного пространства (в данном продукте — EcoSpace), по сути аналогичным используемому в Bold for Delphi. Появление «нового» дескриптора CurrencyManagerHandle «вынужденно», и оно компенсирует отсутствие в ECO дескриптора списка (BoldList-Handle, см. главу 7), который также возвращал значение текущего объекта. Обращает на себя внимание отсутствие системного дескриптора (аналога BoldSystemHandle). Вместо него при создании новой EcoWin-формы на эту форму по умолчанию добавляется дескриптор ссылки rhRoot (ReferenceHandleRoot), указывающий на объектное пространство в целом, в нашем случае на переменную Lib.LibEcoSpace. Все добавляемые разработчиком дескрипторы имеют возможность, взяв rhRoot в качестве корневого дескриптора, получить доступ к элементам модели. Для иллюстрации добавим с панели инструментов два дескриптора OCL-выражений (ExpressionHandle), описывающих классы Avtor и Kniga. Назовем первый дескриптор ehAvtors и настроим его свойства, как показано на рис. Б.8.

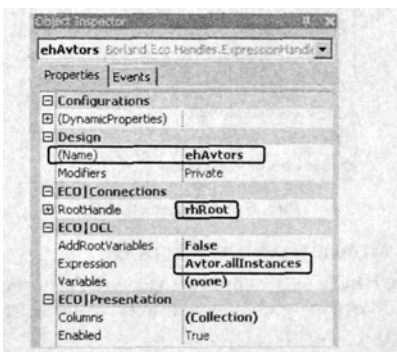


Рис. Б.8. Настройка дескриптора ehAvtors

Для ввода OCL-выражения можно, дважды щелкнув по полю Expression, войти во встроенный OCL-редактор (рис. Б.9). Вид этого редактора абсолютно идентичен уже известному нам OCL-редактору Bold for Delphi. Аналогичным образом настроим и второй дескриптор OCL-выражений ehKnigi для получения списка всех книг.

Теперь поставим следующую задачу — для каждого автора отображать только написанные им книги, а для каждой книги — только ее авторов. Для решения задачи необходимо, во-первых, иметь информацию о текущем авторе или книге. Вот здесь нам и пригодятся дескрипторы текущих объектов — компоненты CurrencyManagerHandle. Добавим два таких дескриптора и назовем их: cmhCurAvtor — для текущего автора и cmhCurKniga — для текущей книги. В свойстве RootHandle первого дескриптора зададим дескриптор авторов ehAvtors, а в аналогичном свойстве второго — дескриптор книг ehKnigi. И, наконец, для решения поставленной задачи не-

обходимо добавить еще два дескриптора ExpressionHandle. Первый из них будет «отбирать» книги, написанные конкретным автором, поэтому назовем его просто - NapisalKnigi. Настроим этот дескриптор следующим образом (рис. Б.10).

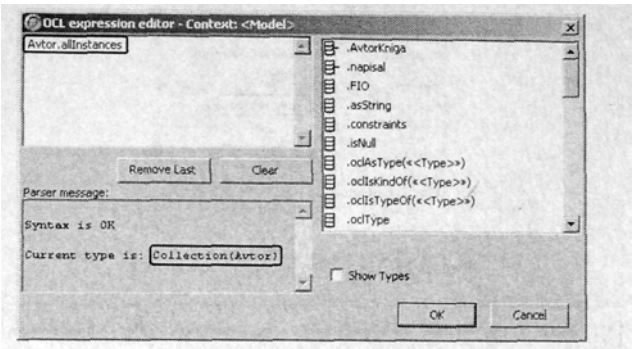


Рис. Б.9. Работа во встроенном OCL-редакторе

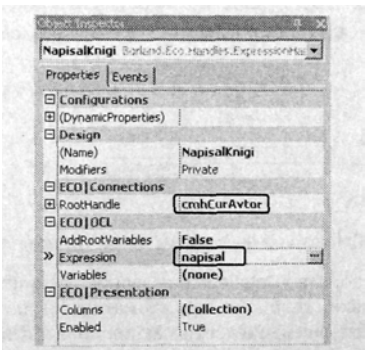


Рис. Б.10. Настройка первого дескриптора

Информацию этот компонент будет получать от дескриптора текущего автора, а возвращать — значения роли napisal, это и будет список книг текущего автора. По аналогии настроим и второй компонент-дескриптор, который назовем Napisana-Avtorom (рис. Б.11).

В результате мы получим набор компонентов ECO, достаточный для решения нашей задачи (рис. Б.12).

Создание графического интерфейса

Поместим на EcoWin-форму сетку dgAvtor типа DataGrid с панели инструментов (категория DataControl). Для привязки этой сетки к данным предназначена группа свойств DataBindings (рис. Б.13).

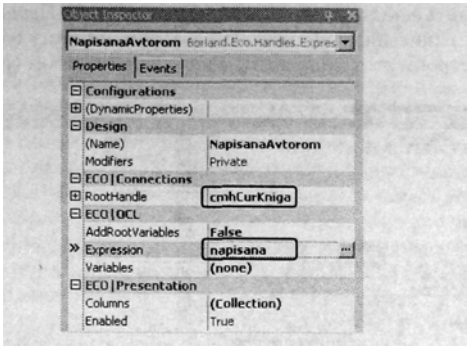


Рис. ь.и. Настройка второго дескриптора

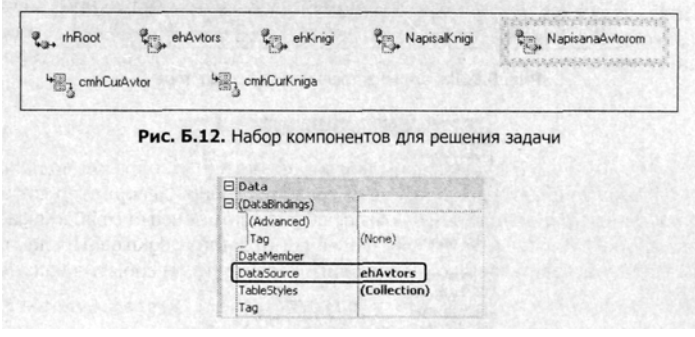


Рис. Б.12. Набор компонентов для решения задачи

Рис. Б.13. Привязка сетки непосредственно к дескриптору EcoSpace

Просто введем в свойство DataSource название дескриптора авторов ehAvtors. В результате в заголовке столбца сетки автоматически появится название атрибута FIO. Стоит обратить внимание на то, как просто в C#Builder Architect осуществляется привязка обычных визуальных компонентов к бизнес-уровню. Напомним, что в Bold для создания бизнес-уровня используются специализированные компоненты графического интерфейса (см. главу 9).

ПРИМЕЧАНИЕ

В Bold тоже можно применять стандартные визуальные компоненты (см. главу 11), однако это требует использования специальных средств.

Теперь, после появления сетки данных, необходимо сделать одну важную вещь. Дескриптор текущего автора cmhCurAvtor пока «не знает», какой именно компонент управляет перемещением указателя по списку авторов. Поскольку мы теперь имеем сетку для отображения всех авторов, то логично и использовать ее в качестве источника информации для cmhCurAvtor о переходах между авторами списка. Для этого в свойстве BindingContext компонента cmhCurAvtor из раскрывающегося списка выберем имя нашей сетки данных dgAvtor (рис. Б.14).

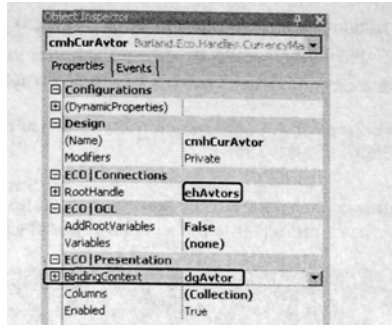


Рис. Б.14. Формирование свойств cmhCurAvtor

Поместим на форму кнопку Добавить автора, а в обработчик события ее нажатия введем строку кода

```
private void button1_Click1(object sender, System.EventArgs e)
    {
        new Avtor(EcoSpace);
    }
```

Добавим вторую сетку dgKniga на форму, аналогичным образом подключим ее к дескриптору книг ehKnigi. Не забудем сразу настроить дескриптор cmhCurKniga текущей книги, для чего в свойстве BindingContext компонента cmhCurKniga из раскрывающегося списка выберем имя нашей сетки данных dgKniga. После этого добавим кнопку Добавить книги со следующим обработчиком события нажатия:

```
private void button2_Click(object sender, System.EventArgs e)
    {
        new Kniga(EcoSpace);
    >
```

ПРИМЕЧАНИЕ

Похожие операторы мы использовали при изучении принципов работы с генерацией кода в Bold (см. главу 12).

Попробуем теперь запустить приложение, однако в результате мы увидим лишь пустую форму WinForm. Это происходит потому, что мы не обеспечили создания экземпляра окна EcoWinForm. Для «штатного» запуска поместим на форму WinForm две кнопки (рис. Б.15).

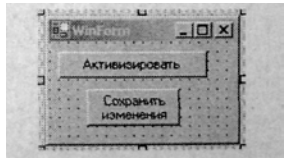


Рис. Б.15. Вид окна WinForm

Настроим пока только кнопку Активизировать, для чего введем следующий обработчик реакции ее нажатия:

```
private void button1_Click(object sender, System.EventArgs e)

        EcoWinForm EcoWinForm=new EcoWinForm(EcoSpace);
        EcoWinForm.Show();
```

В этом обработчике создается и отображается на экране экземпляр EcoWinForm. Теперь можно запустить приложение и ввести авторов и книги (рис. Б.16).

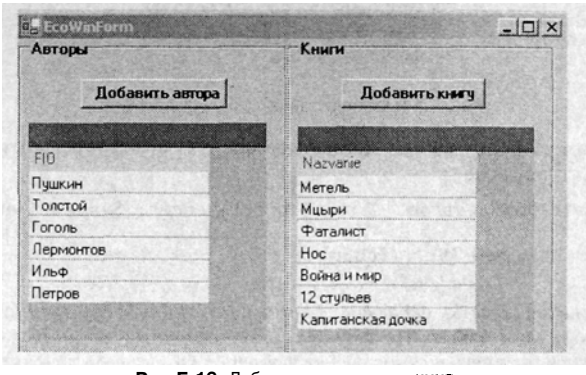


Рис Б.16. Добавление авторов и книг

В данном случае оба списка независимы, поскольку никаких привязок авторов к книгам мы еще не обеспечили. Оставим существующие настроенные сетки данных, пусть в них всегда отображаются полные списки авторов и книг. Для отображения книг конкретного автора добавим еще одну сетку данных и настроим ее, как показано на рис. Б.17.

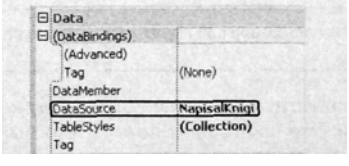


Рис. Б.17. Настройка сетки для отображения книг текущего автора

Понятно, что эта сетка будет получать информацию из «вторичного» дескриптора NapisalKnigi, который возвращает коллекцию книг текущего автора. Аналогичным образом добавим четвертую сетку данных и привяжем ее к дескриптору NapisanaAvtorom для отображения всех авторов текущей книги. После запуска убедимся, что последние две сетки пусты, так как авторы и книги не связаны друг с другом. Для реализации таких связей добавим на форму еще две кнопки, расположив их под третьей и четвертой сетками данных. Кнопка Связать с книгой будет

обеспечивать привязку текущего автора к текущей книге. Кнопка Связать с автором будет обеспечивать привязку текущей книги к текущему автору. Для реализации привязки книг к автору в обработчик нажатия кнопки Связать с книгой введем следующий текст

```
private void button3_ClickI(object sender, System.EventArgs e)

        Avtor CurAvtor=(Avtor)cmhCurAvtor.Element.AsObject;
        Kniga CurKniga=(Kniga)cmhCurKniga.Element.AsObject;
        CurAvtor.napisal.Add(CurKniga);
}
```

Что делает приведенный код? Первая строка обеспечивает присвоение переменной CurAvtor типа Avtor значения (ссылки) текущего автора. Это значение берется из созданного нами дескриптора cmhCurAvtor. Вторая строка аналогичным образом формирует значение CurKniga для текущей книги из дескриптора cmhCurKniga. И, наконец, последняя строка добавляет текущую книгу как элемент роли napisal к текущему автору. Можно убедиться, что генерируемые классы обеспечивают простую навигацию по модели (Cur.Avtor.napisal). Похожим образом сформируем код для кнопки Связать с автором.

```
private void button4_Click(object sender, System.EventArgs e)
        {
            Avtor CurAvtor=(Avtor)cmhCurAvtor.Element.AsObject;
            Kniga CurKniga=(Kniga)cmhCurKniga.Element.AsObject;
            CurKniga.napisana.Add(CurAvtor);
        }
```

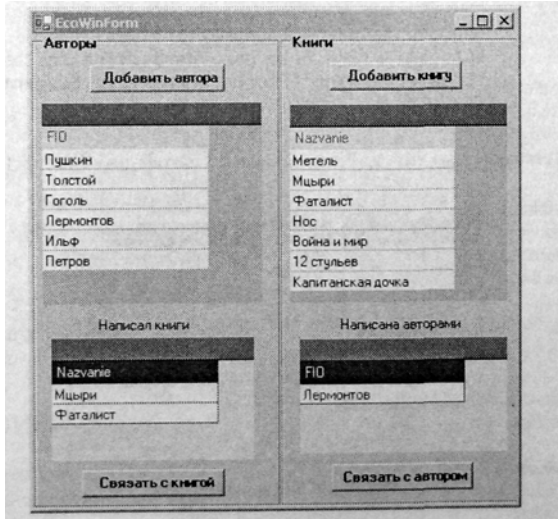


Рис. Б.18. Приложение в работе

Отличие только в том, что в этом случае «добавляется» не книга к автору, а наоборот, автор к текущей книге.

Запустим приложение, введем несколько авторов и книг. Выделим первого автора в сетке авторов и любую книгу в сетке книг. Нажмем кнопку Связать с книгой. Мы увидим, что автор «привязался» к книге, при этом соответствующие сетки данных в нижней части формы правильно отображают информацию о привязке. Добавим авторов Ильф и Петров и книгу 12 стульев. Поставим указатель в сетке авторов на фамилию Ильф, а в сетке книг выберем книгу 12 стульев. Нажмем кнопку Связать с автором. Выберем автора Петров и снова нажмем кнопку Связать с автором. Повторим в любой комбинации описанные действия и убедимся, что привязка работает и для случаев, когда у книги несколько авторов, а у автора несколько книг (рис. Б.18).

Автоформы

Для использования автоформ необходимо добавить строку

```
using Borland.Eco.AutoContainers;
```

в начало секции объявлений программы. Для вызова автоформы также необходимо запрограммировать ее создание по какому-нибудь событию. Например, обеспечим появление автоформы по двойному щелчку на сетке авторов. Для этого в обработчик этого события введем следующий код:

```
   private void dataGrid1_DoubleClick(object sender,
System.EventArgs e)
        {
             IAutoContainer AutoFormAvtor;
             AutoFormAvtor= AutoContainerService.Instance
             .CreateContainer(EcoSpace,ehAvtors.Element);
             (AutoFormAvtor as Form).Show();
        }
```

Попробуем запустить приложение и вызвать автоформу (рис. Б.19).

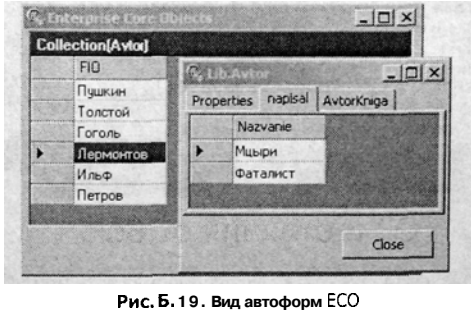


Рис. Б.19. Вид автоформ ECO

Внешний вид автоформ похож на имеющиеся в Bold for Delphi, они также позволяют редактировать данные, однако мы не видим средств добавления или удаления данных. Кроме того, автоматически не поддерживается drag-and-drop.

Создание уровня данных

Нам осталось создать уровень данных для сохранения информации между сеансами работы. Для этого можно воспользоваться как СУБД, так и сохранением данных в XML-документах. Воспользуемся последним способом, как наиболее простым для рассматриваемой задачи. Для этого перейдем на вкладку Design элемента EcoSpace.cs и перетащим на нее компонент persistenceMapperXML1 с панели инструментов. Для настройки данного компонента в инспекторе объектов зададим свойству FileName имя файла, например 1.xml. Щелкнем на пустом пространстве и в свойство PersistenceMapper компонента LibEcoSpace введем имя компонента persistenceMapperXML1 (рис. Б.20).

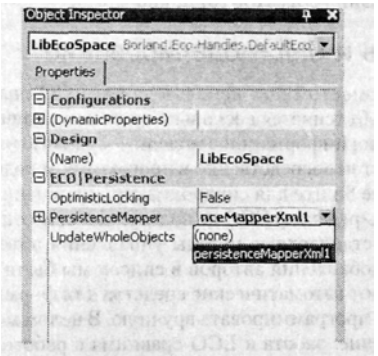


Рис. Б.20. Привязка EcoSpace к уровню данных

Кроме этого, перейдем на WinForm и введем следующий код для реакции на нажатие кнопки Сохранить изменения

```
private void button2_Click(object sender, System.EventArgs e)
        {
             EcoSpace.UpdateDatabase();
        }
```

После запуска приложения можно убедиться, что все введенные изменения сохраняются.

Специфика ECO и отличия от Bold

В результате проведенной работы по созданию простого приложения можно сделать небольшой сравнительный анализ.

Графический интерфейс

ECO не требует наличия специальных визуальных компонентов для построения графического интерфейса пользователя. Таким образом, все стандартные (входящие в поставку продукта C#BuiIder) или любые визуальные компоненты сторонних производителей можно беспрепятственно использовать для создания ECO-приложений. Однако степень автоматической синхронизации графического интерфейса с бизнес-уровнем в ECO несколько ниже, чем в Bold for Delphi. Так, мы были вынуждены вручную задавать свойства источника информации для дескриптора текущего автора и дескриптора текущей книги. По умолчанию эти дескрипторы «не чувствуют» перемещений по спискам авторов и книг. Возможно, что это — своеобразная «плата» за возможность работы со стандартными компонентами. Использование автоформ в ECO также существенно ограничено по сравнению с Bold. Их вызов необходимо программировать вручную, автоформы не позволяют добавлять или удалять авторов, не поддерживается автоматический механизм перетаскивания объектов (drag-and-drop).

Бизнес-уровень и UML-моделирование

Принципиальным моментом при использовании ECO является необходимость генерации кода. Это объясняется несколько другой по сравнению с Bold for Delphi идеологией сохранения информации о модели — существенную часть этой информации ECO сохраняет непосредственно в программном коде, используя так называемый механизм Live Source для синхронизации изменений UML-модели и программного кода. UML-редактор ECO не поддерживает ассоциации типа агрегаций. В ECO отсутствуют стандартные методы управления объектами, даже простейшие операции типа добавления авторов в список мы были вынуждены программировать. Отсутствуют автоматические средства для формирования ассоциаций, их также необходимо программировать вручную. В целом можно сформулировать следующее утверждение: работа в ECO сравнима с работой в Bold при условии генерации кода (см. главу 12). Возможности Bold функционировать без генерации кода классов модели в ECO не поддерживаются.

Уровень данных

В настоящей версии доступны следующие BDP-провайдеры для доступа к СУБД (рис. Б.21).

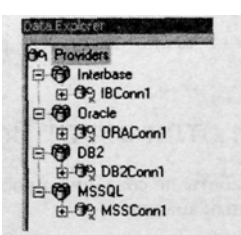
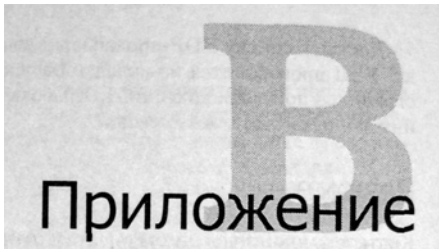


Рис. Б.21. BDP-провайдеры

Доступ к списку BDP-провайдеров для последующего выбора и подключения к СУБД производится на вкладке DataExplorer. Состав СУБД пока существенно ограничен по сравнению с BOLD. Также имеется удобная возможность сохранения данных в XML-документах.

Резюме

Краткое знакомство с инструментом Borland C#Builder Architect позволяет сказать, что технология Borland MDA (называемая в этом продукте ECO) претерпела существенные изменения. Качественно новым свойством является полная независимость от других программных средств в результате интеграции собственного графического UML-редактора, а также интеграция с платформой Microsoft .NET. Из главных специфических особенностей ECO можно выделить гораздо большую «привязанность» к коду программы, что вызвано необходимостью хранения большей части модели непосредственно в коде. Другая принципиальная особенность — возможность использования стандартных графических компонентов, что является существенным преимуществом. Однако это достигается ценой некоторой потери «автоматизации» поведения приложения и необходимости практически всегда прибегать к программированию. Несмотря на довольно сильные отличия, в ECO присутствует и очень большая степень «наследования» качеств своего «родителя» — Bold for Delphi. Практически идентичный набор дескрипторов объектного пространства, использование языка OCL и встроенный OCL-редактор — эти и другие подобные свойства позволяют существенным образом облегчить знакомство со следующим поколением технологии Borland MDA — Enterprise Core Objects.



Библиотека для работы с Object Space

В этом приложении приводится полный исходный текст демонстрационного примера программной реализации библиотеки для работы с объектным пространством при использовании Bold for Delphi.

Назначение

Назначение данной библиотеки — обеспечение реализации минимально необходимого набора наиболее часто употребляемых операций с объектами ОП, то есть добавления, удаления, редактирования объектов, поиска объектов, программное связывание объектов (для случаев однократных и многократных ассоциаций между классами). Другими словами, эта библиотека позволяет программно выполнить операции, которые обеспечивают автоформы или любые другие средства графического интерфейса Bold for Delphi,

Причины и цели создания

При практической разработке приложений в Bold for Delphi часто возникает одна весьма серьезная проблема. Она заключается в следующем. Приступая к созданию приложений базы данных с использованием этой новой технологии, разработчик сталкивается с необходимостью импорта информации, уже накопленной к этому моменту в некоторой базе данных, существовавшей ранее. Причем эта «старая» база данных была создана традиционными способами. Как, не прибегая к ручному вводу информации, обеспечить такой импорт? Ведь Bold for Delphi, как мы уже знаем, сгенерирует новую структуру БД. При этом стандартными методами ввода информации в объектное пространство является использование графического интерфейса (автоформы, сетки и т. д.). С этой точки зрения представленная

ниже программная библиотека может выступать в качестве основы создания подобного инструмента импорта. Вторая цель демонстрации этой библиотеки — более детально показать принципы работы с объектами ОП, частично рассмотренные на страницах данной книги.

Описание

Основные процедуры и функции

В таблице В.1 приведен состав основных процедур и функций библиотеки, их назначение и описание входных параметров и выходных величин.

Таблица В.1. Описание процедур и функций библиотеки

№	Тип	Имя	Назначение	Параметры
1	Function	Attr	Возвращает значение атрибута объекта	L — объект; attrname — имя атрибута. Возвращает вариантное значение
2	Function	ObjectLocate	Ищет объект по значению атрибута	ClassName — имя класса объекта; Attr — имя атрибута; Searchvalue — значение атрибута. Возвращает найденный объект или NIL
3	Procedure	AddObject	Добавляет объект в ОП	ClassName — имя класса объекта; AttrName — массив имен атрибутов; AttrValue — массив значений атрибутов
4	Procedure	AddObject-WithBlobs	То же, но с BLOB-атрибутами	То же, плюс: BlobAttrName — массив имен BLOB-атрибутов; Fname — массив имен файлов, данные из которых будут помещены в BLOB
5	Function	NewObject	Создает новый объект в ОП	ClassName — имя класса объекта; AttrName — массив имен атрибутов; AttrValue — массив значений атрибутов. Возвращает новый объект
6	Function	NewObject-WithBlobs	То же, но с BLOB-атрибутами	То же, плюс: BlobAttrName — массив имен BLOB-атрибутов; Fname — массив имен файлов, данные из которых будут помещены в BLOB
7	Procedure	EditObject	Изменяет атрибуты объекта	ClassName — имя класса объекта; AttrName — массив имен атрибутов; AttrValue — массив новых значений атрибутов
8	Procedure	EditObject-WithBlobs	То же, включая BLOB-атрибуты	То же, плюс: BlobAttrName — массив имен BLOB-атрибутов; Fname — массив имен файлов, данные из которых будут помещены в BLOB
9	Procedure	SetObject-ToLink	Связывает два объекта (однократная ассоциация)	S — объект; Rolename — имя роли противоположного конца ассоциации; LinkObject — объект класса на противоположном конце ассоциации

Таблица B.1 (*продолжение*)

№	Тип	Имя	Назначение	Параметры
10	Procedure	AddObject-ToMultiLink	Добавляет объект к связи (многократная ассоциация)	O — объект; Rolename — имя роли на противоположном конце ассоциации; AdOb — добавляемый объект класса на противоположном конце ассоциации
11	Procedure	LocateAnd-SetLinkTo-Object	Находит объект и связывает с ним заданный объект (однократная ассоциация)	S — объект Rolename — имя роли на противоположном конце ассоциации; attrlink — имя атрибута искомого объекта; searchvalue — значение атрибута, по которому будет произведен поиск
12	Procedure	LocateAnd-SetMultiLink-ToObject	То же для многократной ассоциации	S — объект; Role — имя роли на противоположном конце ассоциации; attr — имя атрибута искомого объекта; LinkObject — имя класса искомого объекта; Searchvalue — массив значений атрибута, по которому будет произведен поиск

Вспомогательные процедуры и функции

Для удобства работы с русскоязычными названиями классов, атрибутов и ролей непосредственно в программе, в состав данной библиотеки включена функция транслитерации TL. Нет необходимости вызывать ее непосредственно, поскольку она используется внутри программного кода описанных процедур и функций. Кроме того, для использования библиотеки необходимо в секции инициализации главной программы (или где-то еще, перед первым обращением к данной билиотеке) вызвать процедуру

```
RegisterObjectSpace(bsystem:TBoldSystemHandle)
```

где в качестве bsystem необходимо задать системный дескриптор ОП.

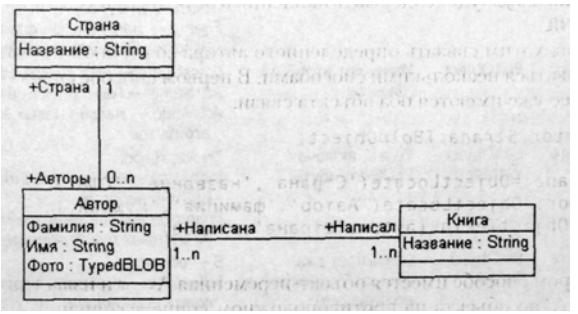


Рис В.1. Используемая UML-модель

Примеры использования

Здесь для наглядности будут рассмотрены несколько примеров применения функций и процедур из данной библиотеки. В качестве UML-модели принимается фрагмент уже известной по книге модели (рис. В.1).

Добавление и изменение объектов ОП

Оператор для создания (добавления) одного автора:

```
NewObject('Автор',['фамилия','имя'],['Пушкин','Александр']);
```

Оператор для добавления автора вместе с фотографией:

```
NewObjectWithBlobs('Автор',['фио'],['Пушкин'],
['фото'],['C:\Photo\push.jpg']);
NewObject('Автор',['фамилия','имя'],
['Пушкин','Александр'],['фото'],['C:\Photo\push.jpg']);
```

Оператор для изменения названия страны:

```
EditObject('Страна',['фио'],['Пушкин']);
```

Благодаря транслитерации можно использовать русскоязычные имена классов и атрибутов.

Поиск и связывание объектов

Для поиска книги по атрибуту достаточно написать

```
var  Kniga:TBoldObject;
   st  :string;
begin
   st:='12  стульев';
   Kniga:=ObjectLocate('Книга','название',st);
или
   Kniga:=ObjectLocate('Книга','название', '12 стульев');
```

Переменная Kniga будет содержать найденный объект (при успешном поиске) либо значение NIL.

Если мы хотим связать определенного автора со страной проживания, можно воспользоваться несколькими способами. В первом способе сперва отыскиваются или заранее уже имеются оба объекта связи.

```
var  Avtor,Strana:TBoldObject;
begin
   Strana:=ObjectLocate('Страна','название','США');
   Avtor:=ObjectLocate('Автор','фамилия','Пушкин');
   SetObjectToLink(avtor,'страна',strana);
end;
```

Во втором способе имеется объект-переменная Avtor и известно, что автор живет в США, но объекта на противоположном конце ассоциации (страна) еще не существует.

```
LocateAndSetLinktoObject(Avtor,'страна','название', 'США');
```

И, наконец, работа с многократными ассоциациями. Одним оператором мы «сообщим» объекту типа страна, что к нему нужно привязать многих авторов

```
LocateAndSetMultilinkToObject
(Strana,'авторы','Автор','фио',['Пушкин','Толстой',
'Гоголь','Лесков'])
```

Или привяжем к одной книге нескольких авторов

```
Kniga:=ObjectLocate('Книга','название', '12 стульев');
LocateAndSetMultilinkToObject
(Kniga,'написана','Автор','фио',['Ильф','Петров'])
```

Таким образом, при использовании процедур данной библиотеки можно относительно простым способом добавить в ОП новые объекты, при необходимости их отредактировать, а также связать друг с другом посредством ассоциаций.

Предупреждения

Программный текст библиотеки (листинг В.1) необходимо рассматривать в качестве демонстрационного примера, не предназначенного для непосредственного промышленного применения. Автор сознательно не оптимизировал код ради простоты и наглядности, кроме того, в ряде случаев необходимо перед использованием представленного кода предусмотреть защиту типа блоков «try..finally», при работе с BLOB-потоками и файлами. Также для использования библиотеки без изменений кода необходимо учесть, что по умолчанию форматом графических файлов принимается JPG. Внимательный разработчик самостоятельно может «довести» приведенный пример реализации библиотеки до необходимого уровня надежности, быстродействия и универсальности.

ВНИМАНИЕ

Автор не несет ответственности за возможные нарушения работоспособности программных или аппаратных средств, прямо или опосредованно связанных с использованием представленного в листинге В.1 программного кода.

Листинг В.1. Демонстрационный пример программной реализации библиотеки **для** работы с объектным пространством Bold **for Delphi**

```
unit ObjectSpace;

                                        INTERFACE

uses
SysUtils,
Variants,
Boldattributes,
BoldSystem,
BoldSystemHandle,
BoldElements,
BoldMeta;
var
```

```
BSys : TBoldSystem;
CurObj,ObjSave,ObjLocate,NewObj,LocObj,AddObj : TBoldObject;
ObjList : TBoldObjectList;

//-----------------------------------------------------------------------------
Function TL(mylangstring:string) : String;
//-----------------------------------------------------------------------------
Procedure RegisterObjectSpace(bsystem:TBoldSystemHandle)
//-----------------------------------------------------------------------------
Function Attr(
   L:TBoldObject;
   attrname:string)  :variant;
//-----------------------------------------------------------------------------
Function ObjectLocate(
   ClassName:string;
   attr:string;
   searchvalue:variant)  :TBoldObject;
//-----------------------------------------------------------------------------
Procedure AddObject(
                        ClassName:string;
                        AttrName: array of string;
                        AttrValue:array of variant);
//-----------------------------------------------------------------------------
Procedure AddObjectWithBlobs(
                        ClassName:string;
                        AttrName: array of string;
                        AttrValue:array of variant;
                        BlobAttrName:array of string;
                        FName:array of string);
//-----------------------------------------------------------------------------
Function NewObject(
                        ClassName:string;
                        AttrName: array of string;
                        AttrValue:array of variant):
TBoldObject;
//-----------------------------------------------------------------------------
Function NewObjectWithBlobs(
                        ClassName:string;
                        AttrName: array of string;
                        AttrValue:array of variant;
                        BlobAttrName:array of string;
                        FName:array of string):      TBoldObject;
//-----------------------------------------------------------------------------
Procedure EditObject(
                        L:TBoldObject;
                        AttrName: array of string;
                        AttrValue:array of variant);
//-----------------------------------------------------------------------------
Procedure EditObjectWithBlobs(
L:TBoldObject;
                        AttrName: array of string;
                        AttrValue:array of variant:
```

```
                        BlobAttrName:array of string;
                        FName:array of string);
//-------------------------------------------------------------------------------
Procedure SetObjectToLink(
                        S:TBoldObject;
                        rolename:string;
                        LinkObject:TBoldObject);
//-------------------------------------------------------------------------------
Procedure AddObjectToMultiLink(
                        O:TBoldObject;
                    rolename:string;
                        AdOb.TBoldObject);
//-------------------------------------------------------------------------------
Procedure LocateAndSetLinkToObject(
                        S:TBoldObject;
                        rolename:string;
                        attrlink:string;
                        searchvalue:variant);
//-------------------------------------------------------------------------------
Procedure LocateAndSetMultHinkToObject(
                        S:TBoldObject;
                        role:string;
                        LinkObject:string;
                        attr:string;
                        searchvalue:Array of string);
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
                                    IMPLEMENTATION

const ABCLength=64;
const bLang: array[1..ABCLength] of string=
('а','б','в','г','д','е','ж','з','и','й','к','л','м','н','о',
 'п','р','с','т','у','ф','х','ц','ч','ш','щ','ь','ъ','ы','э','ю','я',
 'А','Б','В','Г','Д','Е','Ж','З','И','Й','К','Л','М','Н','О',
 'П','Р','С','Т','У','Ф','Х','Ц','Ч','Ш','Щ','ь','ъ','ы','Э','Ю','Я'
 );
const bEnglish: array[1..ABCLength] of string =
('a','b','v','g','d','e','g','z','i','i','k','l','m','n','o',
 'p','r','s','t','u','f','h','c','ch','sh','sh','j','j','y','e','yu','ya',
 'A','B','V','G','D','E','G','Z','I','I','K','L','M','N','O',
 'P','R','S','T','U','F','H','C','Ch','Sh','Sh','j','j','Y','E','Yu','Ya'
 );
var InnerObj : TBoldObject;



//-------------------------------------------------------------------------------
Function TL(mylangstring:string) : String;
var i,ib,j:word;
   s,seng:string;
   pr:boolean;
begin
ib:=length(mylangstring);
```

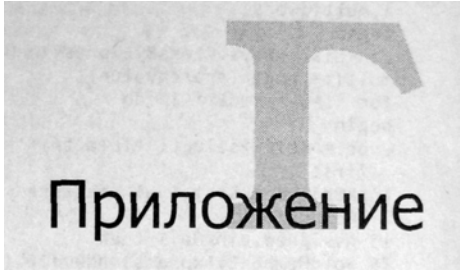
```
seng:='';
for i:=1 to ib do
begin
pr:=false;
s:=mylangstring[i];
if s='' then continue
else for j:=1 to ABClength do
if s=bLang[j] then begin
pr:=true;
seng:=seng+bEnglish[j];
break;
end;
if s=' ' then seng:=seng+'_' else
if s='.' then seng:=seng+'.' else if (not pr) then
seng:=seng+s;
end;
Result:=seng;
end;
//=================================================================

Procedure RegisterObjectSpace(bsystem:TBoldSystemHandle);
begin
BSys:=bsystem.System;
end;


Function Attr(
             L:TBoldObject;
             attrname:string)  {variant;
begin
result:=L.BoldMemberByExpressionName[TL(attrname)].AsString;
end;


//=================================================================
Function ObjectLocate(
                        ClassName:string;
                        attr:string;
                        searchvalue;variant)  :TBoldObject;
var expr:string:
SearchObj:TBoldObject;
L:TBoldObjectList;
begin
if searchvalue=null then begin Result:=nil; exit;end;
L:=BSYS.ClassByExpressionName[TL(ClassName)];
expr:='self->select('+TL(attr)+'='+QuotedStr(searchvalue)+')->
  first';
SearchObj:=L.EvaluateExpressionAsDirectElement(expr) as
  TBoldObject;
if Assigned(SearchObj) then Result:=SearchObj else Result:=nil;
end;
```

```
Procedure AddObject(
                        ClassName:string;
                        AttrName: array of string;
                        AttrValue:array of variant);
var
   L  : TBoldObjectList;
   i  : word;
   nattr  :integer;
begin
L:=BSys.ClassByExpressionName[TL(ClassName)];
InnerObj:=(L.AddNew as TBoldObject);
nattr:=High(AttrName);
if nattr<0 then exit;
for i;=0 to High(Attrname)  do
if string(AttrValue[i])<>'' then
InnerObj.BoldMemberByExpressionName[TL(AttrName[i])]
.SetAsVariant(AttrValue[i]);
AddObj:=InnerObj;
end;

//================================================================
Procedure AddObjectWithBlobs(
                        ClassName:string;
                        AttrName: array of string;
                        AttrValue:array of variant;
                        BlobAttrName:array of string;
                        FName:array of string);
var
   L  : TBoldObjectList;
   bs : TBoldBlobStream;
   sm : TBoldBlobStreamMode;
   i : word;
   blobnum:integer;
   filext : string;
begin
L:=BSys.ClassByExpressionName[TL(ClassName)];
InnerObj:=(L.AddNew as TBoldObject);
for i:=0 to High(Attrname)  do
if string(AttrValue[i])<>'' then
InnerObj.BoldMemberByExpressionName[TL(AttrName[i])]
.SetAsVariant(AttrValue[i]);
blobnum:=Length(BlobAttrName);
sm:=bmReadWrite;
if (blobnum<>0) then  for i:=0 to blobnum-1 do
if FileExists(FName[i]) then
begin        // здесь необходимо использовать блок try..finally !
filext:=ExtractFileExt(FName[i]);
bs:=(InnerObj.BoldMemberByExpressionName[TL(BlobAttrName[i])] as
  TBABlob)
.CreateBlobStream(sm);
bs.LoadFromFile(FName[i]);
if (filext='.jpg') then
```

```
begin
(innerobj.BoldMemberByExpressionName[TL(BlobAttrName[i])] as
  TBATypedBlob)
.ContentType:='image/jpeg';
// сюда можно добавить типы контекстов для других графических форматов
end;
bs.Free;
end;
AddObj:=InnerObj;
end;

//=================================================================
Function NewObject(
                        ClassName:string;
                        AttrName: array of string;
                        AttrValue:array of variant):
TBoldObject;
begin
AddObject(ClassName,AttrName,AttrValue);
Result:=InnerObj;
CurObj:=InnerObj;
NewObj:=InnerObj;
end;
//=================================================================
Function NewObjectWithBlobs(
                        ClassName:string;
                        AttrName; array of string;
                        AttrValue:array of variant;
                        BlobAttrName:array of string;
                        FName:array of string):      TBoldObject;
begin
AddObjectWithBlobs(ClassName,AttrName,AttrValue,BlobAttrName,FName);
Result:=InnerObj;
CurObj:=InnerObj;
NewObj:=InnerObj;
end;
//=================================================================
Procedure EditObject(
                        L:TBoldObject;
                        AttrName: array of string;
                        AttrValue:array of variant);
var
   i: word;
begin
if L=nil then Exit;
if Length(AttrName)<>0 then
for i:=0 to High(Attrname) do
L.BoldMemberByExpressionName[TL(AttrName[i])].SetAsVariant(AttrValue[i]);
end;
Procedure EditObjectWithBlobs(
                        L:TBoldObject;
                        AttrName: array of string;
```

```
                        AttrValue:array of variant;
                        BlobAttrName:array of string;
                        FName:array of string);
var bs : TBoldBlobStream;
sm : TBoldBlobStreamMode;
i  : word;
blobnum:integer;
filext:string;
begin
if L=nil then exit;
if length(AttrName)<>0 then
for i:=0 to High(Attrname)  do
L.BoldMemberByExpressionName[TL(AttrName[i])].SetAsVariant(AttrValue[i]);
sm:=bmReadWrite;
blobnum:=Length(BlobAttrName);
if (blobnum<>0) then for i:=0 to blobnum-1 do if
FileExists(FName[i]) then
begin  // здесь необходимо использовать блок try..finally !
filext:=ExtractFileExt(FName[i]);
bs:=(L.BoldMemberByExpressionName[TL(BlobAttrName[i])] as
  TBABlob)
.CreateBlobStream(sm);
bs.LoadFromFile(FName[i]);
if (filext='.jpg') then
begin
(L.BoldMemberByExpressionName[TL(BlobAttrName[i])] as
  TBATypedBlob)
.ContentType:='image/jpeg';
end;
bs.Free;
end;
end;

//===============================================================
Procedure SetObjectToLink(
                        S:TBoldObject;
                        rolename:string;
                        LinkObject:TBoldObject);
begin
S.BoldMemberByExpressionName[TL(rolename)].assign(LinkObject);
end;


Procedure LocateAndSetMultiLinkToObject(
                        S:TBoldObject;
                        role:string;
                        LinkObject:string;
                        attr:string;
                        searchvalue:Array of string);
var expr:string;
LinkObj:TBoldObject;
LinkList:TBoldObjectList;
```

```
i,multi:word;
begin
LinkList:=bsys.ClassByExpressionName[TL(LinkObject)];
multi:=Length(searchvalue);
for i:=0 to multi-1 do
begin
expr:='self->select('+TL(attr)+'='+QuotedStr(searchvalue[i])+')->
  first';
LinkObj:=LinkList.EvaluateExpressionAsDirectElement(expr) as
  TBoldObject;
if Assigned(LinkObj) then
(S.BoldMemberByExpressionName[TL(role)] as
TBoldObjectList).Add(LinkObj);
end;
LocObj:=LinkObj;
end;


Procedure AddObjectToMultiLink(
                        O:TBoldObject;
                        rolename:string;
                        AdOb:TBoldObject);
begin
(O.BoldMemberByExpressionName[TL(rolename)] as
  TBoldObjectList).Add(AdOb);
end;


Procedure LocateAndSetLinkToObject(
                        S:TBoldObject;
                        rolename:string;
                        attrlink:string;
                        searchvalue:variant);
var expr:string;
LinkObj:TBoldObject;
L:TBoldObjectList;
LinkClass :string;
begin
if searchvalue=NULL then exit;
LinkClass:=S.BoldMemberByExpressionName[TL(rolename)].BoldType.ModelName;
L:=BSys.ClassByExpressionName[TL(LinkClass)];
expr:='self->select('+TL(attrlink)+'.asstring='
  +QuotedStr(searchvalue)+')->first'; // найти объект по атрибуту
LinkObj:=L.EvaluateExpressionAsDirectElement(expr) as
  TBoldObject;
if Assigned(LinkObj) then
S.BoldMemberByExpressionName[TL(rolename)].assign(LinkObj);
LocObj:=LinkObj;
end;
//*****************************************************************
end.
```



Часто задаваемые вопросы

В этом приложении приводятся ответы на вопросы, которые часто возникают у разработчиков, впервые сталкивающихся с технологией Bold for Delphi. Часть вопросов взята из интернет-конференций и форумов разработчиков.

Общие вопросы

Вопрос: Какие версии Rational Rose можно использовать с Bold for Delphi?
Ответ: Версии Rational Rose 98 и выше.

Вопрос: Можно ли создавать Bold-приложения без графического интерфейса?
Ответ: Да. Вариант такого применения демонстрировался в главе 10 при описании примера сохранения размеров и положения формы в XML-документе.

Вопрос: Какие программные модули необходимо дополнительно поставлять при распространении приложений, созданных с применением Bold for Delphi?
Ответ: Никакие (в части Bold).

Вопрос: Как импортировать в Bold-приложение информацию из текстовых файлов или файлов Excel?
Ответ: Для этой цели можно использовать в качестве основы пример программной библиотеки (см. приложение В).

Вопрос: Как получить значение тег-параметра во время выполнения программы?
Ответ: Можно использовать следующий оператор.

```
BoldSystemHandle.System.BoldSystemTypeInfo.
ClassTypeInfoByExpressionName['MyClass'].TaggedValues['persistent']
```

Вопрос: Можно ли во время выполнения приложения изменять значения тег-параметров?

Ответ: Нет, поскольку тег-параметр является частью модели приложения, изменение которой во время выполнения невозможно.

Вопрос: Как получить значение BoldID?

Ответ: Проще всего для этой цели использовать свободно распространяемые расширения OCL (см. главу 15).

Вопрос: Как отобразить полный список всех элементов ОП, например в BoldGrid?

Ответ: Подключить **BoldGrid** к дескриптору списка TBoldListHandle, который в свою очередь подключить к системному дескриптору. Оставить свойство Expression дескриптора списка пустым. В результате возвращаемая им информация будет содержать все элементы ОП.

СУБД

Вопрос: Можно ли использовать уже созданные «обычные» базы данных?

Ответ: Да, при условии написания специального программного адаптера для перевода «старой» БД в базу данных Bold. Для этой цели можно использовать в качестве основы пример программной библиотеки (см. приложение **В).**

Вопрос: Как сгенерировать структуру базы данных в run-time?

Ответ: Можно использовать следующий код в качестве основы

```
BodSystemHandle.PersistenceHandleDB.CreateDataBaseSchema(true);
```

Логический параметр указывает, игнорировать или нет «посторонние» таблицы (которые не созданы автоматически при генерации БД, например, если SQL-сервер использует свои системные таблицы).

Вопрос: Как узнать, синхронизировано ли объектное пространство с базой данных? Как закончить работу без сохранения данных в БД?

Ответ: Для этой цели можно использовать свойство BoldDirty. Ниже приведен пример такой проверки при попытке закрытия главной формы. Для выхода без синхронизации можно использовать метод DisCard (см. тот же листинг ниже).

```
procedure TfrmMain.FormCloseQuery(Sender: TObject; var CanClose:
Boolean);
begin
CanClose := true;
if (BoldSystemHandle1.Active and
BoldSystemHandle1.system.BoldDirty) then
begin
if MessageDlg('Имеются несохраненные объекты. Сохранить их в базе
  данных?',
mtConfirmation, [mbYes, mbNo], 0) = mrYes then
BoldSystemHandle1.UpdateDatabase   // синхронизация ОП и БД
else
BoldSystemHandle1.system.Discard   // выход без синхронизации ОП
end
end;
```

Компоненты и OCL

Вопрос: Как обеспечить заполнение стандартного комбинированного списка (ComboBox) значениями из какого-то дескриптора?

Ответ: Можно использовать приведенную ниже процедуру

```
procedure FillComboFromBold(classname:string;attr:string;
                                    Comb : TComponent);
var I :word;
st:string;
begin
TComboBox(Comb).Clear;
n:=dm.sys.system.ClassByExpressionName[TL(cname)].Count;
for i:=0 to n-1 do
begin
   st:=dm.sys.system.ClassByExpressionName[TL(cname)]
   .BoldObjects[i].BoldMemberByExpressionName[TL(attr)].AsString;
   TComboBox(Comb).Items.Add(st);
end;
end;
```

Вопрос: Как осуществить инкрементный поиск в дескрипторе списка?

Ответ: Можно использовать следующий код в качестве основы

```
procedure TMyForm.EditSearchChange(Sender: TObject);
var st,expr :string;
begin
st:='%'+EditSearch.Text+'%';
Expr:='self->orderby(attrsearch)'+'
->select(attrsearch.sqlLikeCaseInsensitive('+
QuotedStr(st)+'))';
MyListHandle.Expression:=Expr;
end;
```

Вопрос: Можно ли настроить внешний вид автоформы (цвет, шрифт и т. д.)?

Ответ: Да. Для этого можно воспользоваться исходными текстами, поставляемыми с Bold for Delphi, или сделать частичные настройки программно (пример ниже) во время выполнения приложения.

```
procedure ShowAutoFormForClass(ClassName:String);
var AF : TForm;
nc,i : word;
grid:TBoldGrid;
im:TImage;

begin
AF:=AutoFormProviderRegistry.FormForElement(dm.Sys.System
.ClassByExpressionName[TL(ClassName)]);
if Assigned(AF) then AF.Show;
AF.Ctl3D:=False;
AF.Caption:='Это форма для класса '+ClassName;
AF.left:=0;
AF.Top:=0;
```

```
AF.AutoSize:=false;
af.Width:=400;
grid:=(AF.FindComponent('BoldGrid') as TBoldGrid);
if assigned(grid)  then
begin
   grid.Align:=alClient;
   grid.Color:=$00E1FFFF;
   grid.DefaultColWidth:=150;
end;
end;
```

Вопрос: Как заполнить стандартный компонент TRichEdit информацией из BLOB-атрибута?

Ответ: Можно использовать следующий код в качестве основы (в реальном коде необходимо задействовать try...finally)

```
Procedure  RichEditFromList(R:TRichEdit;H:TBoldListHandle;
AttrBlob:string);
begin
R.Clear;
R.Lines.LoadFromStream((H.CurrentBoldObject
.BoldMemberByExpressionName[AttrBlob] as
  TBABlob).CreateBlobStream(bmRead));
(H.CurrentBoldObject.BoldMemberByExpressionName[AttrBlob]
  as TBABlob).CreateBlobStream(bmRead).Free;
end;
```

Вопрос: Как пользоваться компонентом BoIdFilter?

Ответ: Этот компонент является несколько устаревшим и введен для совместимости с предыдущими версиями Bold for Delphi. Для целей фильтрации гораздо удобнее пользоваться OCL. Например, для фильтрации авторов по странам можно использовать следующее OCL-выражение в дескрипторе списка

```
Avtor.allInstances->select(strana.nazvanie='Россия');
```

Вопрос: Для чего нужен компонент BoldComparer?

Ответ: Этот компонент также введен для совместимости с предыдущими версиями Bold for Delphi. Он предназначен для сортировки объектов в списке. Вместо этого компонента гораздо удобнее пользоваться OCL. Например, для сортировки авторов по фамилии можно использовать следующее OCL-выражение в дескрипторе списка

```
Avtor.allInstances->orderby(fio);
```

Гибкость здесь в том, что OCL позволяет сортировать авторов, например, по количеству написанных книг (при этом само это количество вычисляется «на лету»), то есть эффективно используя навигацию по модели.

```
Avtor.allInstances->orderby(napisal->size);
```

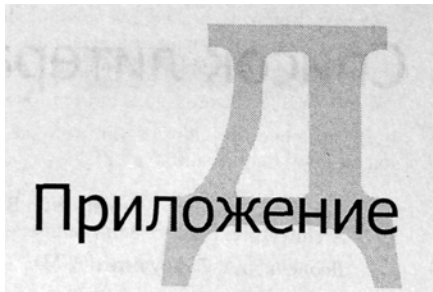
Вопрос: Как сделать фильтрацию по нескольким условиям?

Ответ: Для этого в качестве условия фильтра можно использовать любое составное логическое выражение. Например, следующее OCL-выражение выберет всех авторов, кроме автора «Иванов», написавших более двух **книг**

```
Avtor.allInstances->select((fio<>'Иванов' ) and (writes->size>2))
```

Другим способом достижения того же результата является последовательная (цепочечная) фильтрация

```
Avtor.allInstances->select(fio<>'Иванов' )->select(writes->
size>2)
```



Интернет-источники

Ниже приведен перечень основных интернет-источников информации о продукте Bold for Delphi с краткими аннотациями. Описания продуктов сторонних организаций приведены в главе 15.

1. http://info.borland.com/techpubs/delphi/boldfordelphi/ — официальная страница сайта компании-разработчика Borland, посвященная Bold for Delphi.
2. news://forums.borland.com — новостной сайт Borland, содержащий следующие полезные группы новостей по продукту Bold for Delphi: borland.public.delphi.modeldrivenarchitecture.general (общие технические вопросы) и borland.public.delphi.modeldrivenarchitecture.thirdparty (технические вопросы по продуктам сторонних компаний).
3. http://www.boldbox.com — сайт-портал по Bold for Delphi, содержащий множество ссылок на другие источники.
4. http://www.howtodothings.com — сайт, содержащий страницу «Bold», содержащую примеры решений и обзорные статьи.
5. http://www.viewpointsa.com/bold_resources/ - сайт, содержащий статьи для начинающих.
6. http://intrasting.com/boldrave — сайт компании-разработчика продукта BoldRave.
7. http://www.DroopyEyes.com — сайт компании-разработчика продукта deBold.
8. http://www.plexityhide.com — сайт компании-разработчика продуктов ph_grid, ph_gant.
9. http://www.rad-studio.com - сайт компании-разработчика продукта BoldGridPro.
10. http://www.holtonsystems.com — сайт компании-разработчика OCL-расшнрений и TCP OSS.
11. http://www.sf.net/projects/boldsoapserver/— сайт компании-разработчика BoldSoapServer.
12. http://www.neosight.com/ — сайт компании-разработчика BoldExpress.

Список литературы

1. Anneke Юеppe, Jos Warmer, Wim Bast. MDA Explained. The Model Driven Architecture: Practice and Promise. — Addison Wesley, 2003.
2. Леоненков А. Самоучитель UML - СПб.: BHV, 2001.
3. Елманова Н., Трепалин С, Тендер А. Delphi 6 и технология COM. — СПб: Питер, 2002.
4. Грибачев К. Тонкие базы данных и инструменты для их разработки в Delphi и C++Builder. - КомпьютерПресс, 2003, № 7, 8.

Алфавитный указатель

A

ALM, 50

B

BoldActions, 218
BoldCaptionController, 204
BoldCheckBox, 187
BoldComboBox, 185
BoldCursorHandle, 144, 155
BoIdDataset, 241
BoldDerivedHandle, 144, 149
BoldEdit, 182
BoldExpressionHandle, 144, 146
BoldGrid, 182
BoldImage, 195
BoldLabel, 181
BoldListBox, 187
BoldListHandle, 144, 154
BoldOCLVariables, 144, 151, 152
BoldPageControl, 188
BoldReferenceHandle, 144
BoldSortingGrid, 184
BoldSQLHandle, 144, 222
BoldSystemHandle, 144, 145
BoldSystemTypeInfoHandle, 144, 145
BoldTreeView, 190
BoldUnloaderHandle, 144
BoldVariableHandle, 144, 151
BolsAsStringRenderer, 198
Borland MDA, 47, 54
  Bold for Delphi, 50, 55
  C#Builder Architect, 51
  ECO, 51
  возможности, 51
  инсталляция, 56
  история, 49
  обзор компонентов, 57
  преимущества, 52
  состав, 47
  трехуровневая структура, 51

C

CASE, 52
CMS ,292
COM, 84

D

DCOM, 291

E

ECO, 54
MDA, 46
  история, 21
  концепции, 28
  перспективы, 27
  преимущества, 26
  структура и состав, 21
  этапы разработки, 24

M

ModelMaker, 45, 50
mutable, 136

O

OCL, 51, 52, 53, 104
  арифметические операторы, 108
  встроенный редактор, 163
  выражение, 66
  вычисляемые атрибуты, 121
  доступ к классам, 106
  запрос, 67
  коллекции, 108, 109
  контекст, 108
  навигация по модели, 105, 108
  ограничения (constraints), 125
  операции сравнения, 107
  особенности диалекта, 128
  программное использование, 173
  работа с множествами, 111
  расширения, 298

OCL (*продолжение*)
репозиторий, 177
роль в Borland MDA, 104
типы данных, 107
фильтрация, ПО
OLLE, 292
OSS, 292
TCP OSS, 298

R

Rational Rose, 39
задание вычисляемых атрибутов, 121
задание ограничений, 125
настройка, 84
Renderer, 197

S

SOAP, 291
Bold SOAP Server, 299
SQL
BoldSQLHandle, 222
OCL2SQL, 226
идеология применения, 220

T

TBoldElement, 134, 136, 137, 176
TBoldObjectList, 134, 139
TBoldSystem, 134, 137, 138

U

UML, 24, 29, 30
встроенный редактор, 61
диаграмма классов, 30
пакет, 36
редакторы, 39
UpdateDatabase, 138

X

XMI, 45
XML, 51, 68, 230
BoldExpress Studio, 295
использование, 231
преимущества, 231

А

автоформа, 69, 199
генерация, 202
автоформа (*продолжение*)
ограничения, 202
структура и состав, 199
точка связи, 201
управление, 202
адаптер СУБД, 206, 214
создание собственных, 215
состав модулей, 217
ассоциация
вычисляемая (derived), 122
класс-ассоциация, 35
концы, 65
кратность роли, 32
направленная, 32
настройка, 65
настройка в Rational Rose, 43, 87
работа в коде, 264
роли, 32
атрибут, 64
вычисляемый, 32
настройка в Rational Rose, 86
обратно-вычисляемый, 283

Б

бизнес-правила, 30, 33, 62, 65, 106
бизнес-уровень, 60, 61, 66

Г

генерация
автоформ, 202
кода, 82, 251
схемы БД, 82, 210
графический интерфейс, 179
BoldGrid, 67
контроллер свойств, 236
сторонние компоненты, 235

Д

дескриптор, 66
выгрузки из памяти, 158
выражения, 146
вычислений, 149
классификация, 143
корневой, 143
курсора, 155
описания типов модели, 145
переменной, 151

дескриптор *(продолжение)*
производный, 143
системный, 144
списка, 66, 154
ссылки, 158
цепочки, 165

К

класс, 31
абстрактный, 32
временный (transient), 43
постоянный (persistent), 43
работа в коде, 263
суперкласс, 34

М

многозвенные приложения, 291
многоязыковая поддержка, 294
модель
OCL-навигация, 108
PIM, 24
PSM, 24
болдификация, 73, 90
болдифицированная, 91
встроенный редактор, 61
дерево элементов, 73
импорт, 84
интерпретация, 82
проверка (validation), 89
роль в Borland MDA, 81
создание в Rational Rose, 83
типы, 23
эволюция, 293
экспорт, 84, 99

О

объектно-реляционное отображение, 52
объектное пространство, 82, 131
иерархия классов, 133
понятие, 132
программное управление, 141
синхронизация, 292
функции, 132
оптимизация, 228
отладка приложений, 294
отношения, 32
агрегация, 33, 34
ассоциация, 32
генерализация, 34

П

подписка, 245, 272
BoldPlaceableSubscriber, 278
механизм, 272
применение в reverse-derived атрибутах, 283
применение в вычисляемых атрибутах, 279
программная реализация, 275
реализация, 273
стандартные события, 274

Р

регионы, 289

С

сторонние производители, 295
СУБД, 207, 221
OCL2SQL, 227
перенос данных, 229
репликация данных, 292
тонкие БД, 228

Т

таблицы
генерируемые, 213
системные, 212
тег-параметры, 83
настройка в Bold, 96
настройка в Rational Rose, 92
привязка к иерархии, 83
состав и назначение, 97

У

уровень данных, 68, 205
BoldDataSet, 241
OCL2SQL, 226
подключение к СУБД, 207
состав, 206
функции, 205