АНДРЕЙ БОРОВСКИЙ

bhv®

Qt4.7+ Практическое программирование на C++

СОЗДАНИЕ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ ДЛЯ WINDOWS И LINUX

ПРОГРАММИРОВАНИЕ ГРАФИКИ, БАЗ ДАННЫХ, МНОГОПОТОЧНЫХ ПРИЛОЖЕНИЙ, ВИДЖЕТОВ И ДР.

РАБОТА С XML

НОВЫЕ ПОДСИСТЕМЫ Qt DECLARATIVE И Qt3/D

БОЛЕЕ 40 ДЕМОНСТРАЦИОННЫХ ПРОГРАММ

ПРОФЕССИОНАЛЬНЫЕ ИНСТРУМЕНТЫ ДЛЯ РАБОТЫ С БИБЛИОТЕКОЙ Qt

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Андрей Боровский

Qt4.7+ Практическое программирование на C++

Санкт-Петербург
«БХВ-Петербург»
2012

УДК 681.3.06
ББК 32.973.26-018.2
Б83

Боровский А. Н.

Б83 Qt4.7+. Практическое программирование на C++. — СПб.: БХВ-Петербург, 2012. — 496 с.: ил. — (Профессиональное программирование)

ISBN 978-5-9775-0757-8

Книга посвящена разработке приложений для Windows и Linux с использованием библиотеки Qt версий 4.7.x и 4.8. Подробно рассмотрено программирование трехмерной и интерактивной графики, баз данных, многопоточных приложений, создание собственных виджетов, описание принципов работы с XML, а также использование новейших подсистем Qt Declarative и Qt3/D. Дано описание классов Qt применительно к решению конкретных задач. Значительное внимание уделено основным принципам разработки сложных приложений. Рассмотрено применение различных профессиональных инструментов разработчика при работе с библиотекой Qt. На авторской странице поддержки книги расположены исходные тексты демонстрационных примеров (более 40).

Для программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Ирина Иноземцева*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Марины Дамбиевой*
Зав. производством	*Николай Тверских*

Подписано в печать 31.10.11.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 39,99.
Тираж 2000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0757-8

© Боровский А. Н., 2011
© Оформление, издательство "БХВ-Петербург", 2011

Оглавление

Введение ... **7**
Что такое Qt? ... 7
Что вы найдете в этой книге? ... 8
Для кого предназначена эта книга? ... 10

ЧАСТЬ I. НАЧАЛЬНЫЙ ЭТАП РАЗРАБОТКИ ПРОЕКТОВ ... **11**

Глава 1. Краткое введение в Qt 4 ... **13**
Средства разработки для библиотеки Qt library ... 13
Qt SDK ... 14
Qt для Microsoft Visual Studio ... 15
Qt Framework ... 16
Qt и Eclipse ... 17
Qt и CMake ... 18
Общая структура приложений Qt ... 20
Инструменты Qt ... 21
Утилита qmake ... 22
Инструменты интернационализации и локализации ... 24
Разделение контекстов ... 29
Склонение существительных, следующих после числительных ... 29
Qt Designer ... 31
Компоновка виджетов ... 31
Визуальное программирование ... 36
Другие возможности Qt Designer ... 49
Редактирование партнеров ... 54
Редактирование порядка перехода между виджетами ... 54
Qt Designer и таблицы стилей ... 55
Контейнеры Qt library ... 57
Сборка приложения Qt 4 под управлением CMake ... 57

Глава 2. Взаимодействие объектов Qt между собой ... **60**
События Qt ... 60
Обработка событий, более подробное обсуждение ... 69
События Qt и многопоточность ... 72
Удаление объекта после выхода из его метода ... 73

Фильтрация событий ... 73
Сигналы и слоты Qt library ... 75
Исследование сигналов и слотов различных типов ... 77
Полезные трюки при работе с сигналами и слотами ... 81
Класс *QSignalMapper* и программа-калькулятор ... 81

Глава 3. Работа над серьезными проектами ... 86
Правило модульности ... 86
Правило разделения движка и интерфейса ... 87
Правило разделения движка и политики ... 88
Правило "ноль-один-бесконечность" ... 88
Простота переноса на другие платформы ... 89
Интроспекция ... 91
Повторное использование кода ... 96
Применение принципа модульности в Qt ... 96
Принцип модульности и простое приложение баз данных ... 98
Интеллектуальные указатели ... 105
Интеллектуальные указатели – "за и против" ... 108
Паттерны и анти-паттерны ... 109
Модульное тестирование ... 114
Qt и модульное тестирование ... 116

Глава 4. Инструментарий профессионального разработчика ... 125
Отладчик GDB ... 125
Точки останова ... 129
Обзор данных ... 130
Стек вызовов ... 132
Статический анализ кода C++ ... 132
Инструменты рефакторинга ... 134
Рефакторинг в Microsoft Visual Studio ... 135
Рефакторинг в среде Eclipse ... 138
Рефакторинг "вручную" ... 140
Рефакторинг и производительность ... 142
Системы контроля версий ... 143
Какую систему контроля версий выбрать? ... 144
Subversion ... 145
Структура директорий проекта Subversion ... 146
Создание резервной копии репозитория ... 148
Клиенты Subversion ... 148
Программа svn ... 148
Qt Creator и Subversion ... 150
Eclipse и Subversion ... 153
Microsoft Visual Studio и Subversion ... 156
TortoiseSVN ... 159

ЧАСТЬ II. РАЗВИВАЕМ ПРОЕКТЫ ... 165

Глава 5. Qt и многопоточность ... 167
Основы теории многопоточности ... 172
Критические области ... 173

Реентерабельность ... 173
Потоковая безопасность ... 174
Потоки без цикла обработки событий ... 175
Атомарные операции и порядок доступа к памяти ... 183
Пример с использованием класса *ExtThread* ... 183
Локальная память потоков ... 187
Передача данных межу потоками ... 188
Класс *QSharedMemory* ... 189
Мьютексы и семафоры ... 189
Кольцевой буфер без блокировок ... 193
Очередь без блокировок ... 205

Глава 6. Высокоуровневый интерфейс потоков ... 218
Классы *QRunnable* и *QThreadPool* ... 218
Что выбрать: *QRunnable* или *ExtThread*? ... 220
Программа поиска файлов по содержимому ... 221
Быстрый доступ к содержимому файла ... 225
Быстрый поиск строк ... 229
Функция *QtConcurrent::run()* ... 232

Глава 7. Возвращаемся к Interview Framework ... 235
Класс *QSqlDatabase* ... 235
Схема работы Interview Framework ... 238
Утилита sqlite3 ... 241
Отношения, допускающие значение *NULL* ... 242
Класс *WeakRelationalTable* ... 243
Класс *WeakRelation* ... 249
Класс *WeakRelationalDelegate* ... 250
Творческое использование делегатов ... 256
Создание стиля заголовков таблиц ... 261
Класс *QDataWidgetMapper* ... 262
Классы *QDataWidgetMapper* и *QComboBox* ... 269

Глава 8. Библиотека Qt и ваша видеокарта ... 271
Графическая система Arthur ... 271
Класс *QGLContext* ... 276
Шейдеры OpenGL в Qt ... 280
Взгляд в будущее ... 286
Подсистема Qt/3D ... 286
Поддержка OpenCL в Qt 4.8 ... 292

Глава 9. Растровая графика и текст ... 298
Вывод на экран больших изображений ... 298
Классы *QImage*, *QPicture* и *QPixmap* ... 299
Виджет для вывода больших изображений ... 299
Текст с элементами форматирования ... 316
Классы *QTextDocument* и *QTextEdit* ... 316
Установка aspell под Windows ... 318
Подготовка программы ... 319

Сохранение документа *QTextDocument* в различных форматах ... 328
Добавление изображений в документ *QTextDocument* ... 329
Новшества в *QTextDocument* и сопутствующих классах ... 332
Класс *QTextLayout* ... 332
Класс *QStaticText* ... 335

Глава 10. Система Graphics View Framework ... 340
Знакомство с системой ... 342
Пишем свою игру ... 344
Формат файла данных ... 344
Переходим к графике ... 353
Встраивание виджетов ... 360
Использование встроенных виджетов в качестве элементов управления ... 362
Виджет для выбора фрагментов изображений ... 371
Graphics View Framework и OpenGL ... 386
Виджет в стиле браузера Opera ... 388
Создаем графический виджет ... 395

ЧАСТЬ III. ДОПОЛНИТЕЛЬНЫЙ МАТЕРИАЛ ... 401

Глава 11. Конечные автоматы и анимация ... 403
Конечные автоматы и минимизация повторяющегося кода ... 408
Конечные автоматы и анимация ... 414

Глава 12. Сценарии для программ Qt ... 418
Передача ссылок на объекты Qt в сценарии ... 420
Обработка сигналов в сценарии ... 426
Использование функций приложения в сценарии ... 428
Создание объектов в сценарии ... 429
Создание новых типов данных в сценарии ... 430
Новшества в системе сценариев Qt 4.7 ... 436

Глава 13. Динамические расширения программ Qt ... 437
Класс *QLibrary* ... 449

Глава 14. Консольные приложения Qt ... 450
Обработка событий в консольной программе ... 450
Ввод и вывод данных на консоль ... 451
Службы и демоны ... 457

Глава 15. Язык QML ... 461
Виджет QML в программе Qt ... 469
Самостоятельная программа на языке QML ... 475
Изменения и дополнения ... 477
Программа clocks ... 478
Утилита qmlviewer ... 481

Заключение ... 483

Список литературы ... 484

Предметный указатель ... 485

Введение

Написание хорошей книги о библиотеке Qt — задача непростая. И не только потому, что написать хорошую книгу вообще непросто. Библиотека Qt давно уже стала популярным средством разработки программ на платформах Windows и Linux, а также для некоторых мобильных устройств. Это значит, что о программировании с использованием Qt уже написаны хорошие книги (некоторые из них выдержали не одно издание). Имена Марка Саммерфилда (Mark Summerfield), Джасмина Бланшетта (Jasmin Blanchette) и Макса Шлее (Max Schlee) должны быть знакомы всем, кто имеет систематический опыт программирования с использованием Qt. Вдобавок к этому библиотека Qt снабжена великолепной документацией с подробным описанием всех классов и множеством примеров. Частично эта документация переведена на русский язык.

Что такое Qt?

Самый простой ответ на этот вопрос выглядит так: Qt — кросс-платформенная библиотека классов C++. Сначала это была библиотека для создания программ с графическим интерфейсом пользователя, которые можно было бы собирать на разных системах без изменения исходных текстов. Со временем библиотека разрослась вширь и вглубь, и теперь, помимо разработки собственно графических интерфейсов, включает в себя сотни классов, охватывающих самые разные аспекты программирования — от разработки приложений баз данных до создания мультимедийных программ и от работы с динамически загружаемыми модулями до конечных автоматов и собственного интерпретатора языков сценариев. И все это работает практически одинаково на таких разных платформах как Microsoft Windows, Linux, Mac OS, многочисленные коммерческие версии UNIX, а также WinCE и Symbian. Библиотека Qt работает на 32- и 64-битных процессорах Intel, процессорах ARM старших моделей и некоторых других. Библиотеку очень легко локализовать, т. е. перевести ее текстовые ресурсы на новый язык, что уже сделано (и постоянно делается) для большинства языков мира, так что после установки в русскоязычной системе многие компоненты Qt сразу же "заговорят" по-русски. Инструментарий, необходимый для локализации приложений, как и многие другие вспомогательные инструменты, входит в состав самой Qt.

Но и это еще не все. Сравнительно недавно библиотека Qt перестала быть только библиотекой классов C++ и вспомогательных элементов и обзавелась собственной кросс-платформенной интегрированной средой разработки Qt Creator (что, однако, не мешает работать с библиотекой Qt в других популярных средах, таких как Microsoft Visual Studio и Eclipse).

Библиотека Qt используется для разработки приложений крупнейшими поставщиками программного продукта. Достаточно назвать такие компании как Nokia (которая в данный момент и владеет библиотекой Qt), Oracle и Google. Кроме того, на основе Qt разрабатывается одна из самых популярных графических оболочек для Linux — KDE, и огромное количество открытых приложений для Linux и других UNIX-систем.

Что вы найдете в этой книге?

В данной книге я старался, насколько это возможно, не повторять то, что уже написано про Qt другими авторами, и не дублировать документацию к библиотеке. Основная цель этой книги — помочь читателю в решении практических задач, в том числе задачи, возникающей перед каждым программистом, пишущим серьезные приложения, — адаптации Qt под свои нужды. Библиотека Qt весьма обширна и охватывает самые разные аспекты разработки программного обеспечения, но ни одна библиотека в мире не содержит в точности те инструменты, которые необходимы именно вам. Особенно, если вы пишете приложение, которое должно быть в чем-то уникальным и непохожим на другие. В этой книге рассмотрены основные вопросы практического программирования на Qt, но в ней также показано, как модифицировать и адаптировать различные компоненты Qt, не теряя таких преимуществ как переносимость приложения между разными платформами и способность модифицированных компонентов взаимодействовать со стандартными компонентами Qt.

Практический характер книги наложил на нее свой отпечаток. Библиотека Qt существует не в вакууме. В книге показано, как Qt взаимодействует с популярными средами разработки и инструментами программирования, как эти инструменты взаимодействуют между собой при использовании Qt. В книге также продемонстрировано, как с помощью Qt реализовать популярные концепции программирования, такие как кросс-платформенность, принцип модульности и паттерны проектирования.

Но и это еще не все. Как программисту с большим стажем работы с Qt, мне приходилось видеть немало ошибок, допущенных менее опытными программистами. На страницах этой книги я стремился предупредить читателя о наиболее распространенных ошибках и помочь ему избежать малоэффективных решений. Для этой цели использованы примеры, представляющие собой фрагменты программ, написанных профессиональными программистами (не только мной). Поскольку Qt широко применяется для разработки программного обеспечения с открытыми исходными текстами, найти исходные тексты надежных, быстрых и эффективных приложений, написанных с помощью Qt программистами высокого уровня, не так уж и трудно.

Следует сказать и о том, чем не является эта книга. Эта книга не является энциклопедией по Qt. Библиотека Qt настолько обширна, что охватить ее целиком в одной книге не представляется возможным. Некоторые аспекты программирования в Qt, например, разработка мультимедийных приложений, остались за рамками этой книги. Книга не является энциклопедией еще и в том смысле, что материал не систематизирован так, как это принято в энциклопедиях (любая попытка систематизировать материал подобным образом неизбежно привела бы к дублированию документации, которая, в любом случае, является самой лучшей энциклопедией по Qt). Из сказанного следует, что книгу лучше читать не выборочно, как справочник, а целиком.

Хотя книга охватывает Qt 4 в целом, особое внимание уделено новинкам последних на момент написания книги версий Qt — 4.7, 4.7.1, 4.7.2, 4.7.3.

Книга состоит из трех частей. Первая часть (*главы 1–4*) является своего рода "расширенным введением" в Qt. Я не стал останавливаться на самых простых понятиях Qt, поскольку они очень хорошо освещены в документации, в том числе на русском языке. Вместо этого я сразу перешел к более углубленному рассмотрению таких концепций как сигналы, слоты и события Qt, механизмы обработки событий, нетривиальные методы работы с Qt Creator и Qt Designer и автоматическое тестирование. Помимо этого первая часть содержит описание методов интеграции Qt с различными популярными инструментами разработки.

Во второй части книги (*главы 5–10*) обсуждается практика разработки серьезных проектов. В *главах 5 и 6* рассматриваются вопросы создания многопоточных приложений Qt. В связи с триумфальным шествием многоядерных процессоров эта тема весьма актуальна, но, на мой взгляд, недостаточно подробно представлена в литературе по Qt. *Глава 7* посвящена разработке приложений баз данных, причем упор сделан на адаптацию существующих компонентов Qt для решения различных нетривиальных задач, которые возникают при работе с базами данных. В *главах 8–10* подробно рассматривается то, для чего Qt изначально была создана, — работа с графикой и отформатированным текстом. Помимо прочего, обсуждается создание простой двухмерной компьютерной игры и программы, которая интерактивно проверяет орфографию, выделяя слова с ошибками, как это делает Microsoft Word, а также способы представления в программах больших изображений (например, географических карт) с минимальными затратами памяти. Кроме того, рассматриваются такие передовые технологии, как использование ресурсов вашей видеокарты для расчетов в приложениях Qt, использование аппаратной поддержки OpenGL для ускорения работы с двухмерной графикой и некоторые другие.

Третья часть (*главы 11–15*) содержит материал, который я классифицировал как дополнительный. Здесь вы узнаете, как работать с конечными автоматами Qt, как задействовать в своих программах систему сценариев Qt, как создавать и использовать динамические модули расширений Qt, а также, как применять новинку Qt — язык QML.

Все главы книги снабжены полнофункциональными примерами программ. Исходные тексты этих примеров вы найдете на моем сайте по адресу **http://symmetrica.net/qt47book/**.

Для кого предназначена эта книга?

С учетом уровня изложения материала и концепции книги от читателя требуется наличие определенных предварительных знаний. Если пробелы в знании Qt можно заполнить с помощью документации непосредственно в процессе чтения книги, то хорошее знание языка программирования C++ требуется с самого начала. Желательно хорошее понимание концепций объектно-ориентированного программирования, принципа модульности и знакомство с паттернами проектирования приложений.

Книга будет полезна тем программистам, которые обладают обширным опытом разработки приложений в таких средах как, например, Borland C++ Builder и Microsoft Visual Studio MFC, и желают перейти на Qt (желание, которое я всячески приветствую), а также тем, кто знаком с предыдущими версиями Qt и хочет обновить свои знания.

Если вы хотите глубже разобраться в таких сложных вопросах как многопоточное программирование, применение языка сценариев для расширения возможностей приложений Qt или использование ресурсов современных высокопроизводительных видеокарт для решения неграфических задач, эта книга тоже может быть вам полезна.

ЧАСТЬ I

Начальный этап разработки проектов

Глава 1. Краткое введение в Qt 4

Глава 2. Взаимодействие объектов Qt между собой

Глава 3. Работа над серьезными проектами

Глава 4. Инструментарий профессионального разработчика

ГЛАВА 1

Краткое введение в Qt 4

Наверняка вы уже знаете, что Qt library — это кросс-платформенная библиотека классов C++ и вспомогательных инструментов, предназначенная для разработки самого широкого круга приложений как для персональных компьютеров, так и для мобильных устройств. Хотя я предполагаю, что вы не только хорошо знаете C++, но и знакомы, по крайней мере, с основами Qt library, в этой главе будет дан краткий обзор Qt 4.7, который окажется особенно полезен тем из вас, кто не имел серьезного опыта разработки на Qt, но пользовался другими средствами разработки, основанными на C++, например Borland C++ Builder или Microsoft MFC. Если же вы вообще незнакомы ни с чем из вышеперечисленного, то вам рекомендуется начать с одной из книг, указанных в *Списке литературы*.

Средства разработки для библиотеки Qt library

Программиста, который только начинает знакомиться с Qt, ожидает некоторое удивление: вместо одного варианта библиотеки ему предлагают несколько коммерческих и открытых вариантов, и не совсем понятно, какой из них лучше выбрать. Возможно, вы не до конца разобрались во всем этом многообразии, даже если у вас уже есть опыт программирования в Qt, тем более что количество вариантов этой библиотеки постоянно меняется. Во время написания этой книги были доступны, как минимум, три среды разработки и два вида лицензирования.

Разработчики Qt распространяют свой продукт на условиях двойного лицензирования, дающего право на бесплатное использование библиотеки в некоммерческих продуктах и более широкую свободу выбора лицензии при покупке коммерческой версии Qt.

Открытый вариант Qt распространяется на условиях GPL 2.0. Сами представители Qt именуют свой подход "Quid Pro Quo", что в вольном переводе с древней латыни означает "баш на баш". Если вы хотите пользоваться средствами Qt бесплатно, взамен вы должны предоставить сообществу свой код (открытая модель). Если вы не желаете делиться кодом, вы должны заплатить деньги (своего рода "выкуп", который, естественно, будет потрачен на дальнейшее совершенствование Qt). Интерес-

но отметить, что лицензионная политика Qt не позволяет лицензиатам применять тот же принцип двойного лицензирования к своим продуктам. Если вы ведете разработку с помощью Qt, ваш проект должен быть либо открытым на условиях GPL, либо коммерческим. То есть вам, конечно, никто не мешает раздавать ваш код бесплатно, но если это делается не на основе GPL, то для разработки этого кода необходимо использовать (и, естественно, оплатить) коммерческую версию Qt.

Коммерческий дистрибутив Qt распространяется в трех вариантах: Qt Console, который позволяет создавать только консольные приложения, Qt Desktop Lite, которая позволяет создавать приложения с графическим интерфейсом, но не включает в себя некоторые важные компоненты, которые присутствуют в Qt Console, например библиотеки для работы с сетью, и Qt Desktop — наиболее полный вариант.

Коммерческая версия Qt поддерживает ОС Windows, Mac OS X, Linux, Embedded Linux, Windows CE. Что касается открытого варианта библиотек, то набор поддерживаемых ОС здесь примерно тот же, но условия работы несколько другие. Некоммерческая версия Qt содержит все те же компоненты, что и наиболее полная коммерческая версия.

Примечание

Во время написания этой книги компания Nokia, которая ведет разработку Qt, фактически перешла под крыло Microsoft. Компания Microsoft интересуется компанией Nokia, прежде всего, как производителем смартфонов, на которые Microsoft планирует устанавливать собственную ОС Windows Phone. При этом создание версии Qt для Windows Phone в планы Nokia не входит, т. к. у Microsoft для этой платформы есть свое проприетарное средство разработки. В результате всего этого будущее Qt для смартфонов и других мобильных устройств выглядит несколько туманным. Что касается Qt для больших компьютеров, то разработчики клянутся, что с ней все будет попрежнему, и мы им верим.

Рассмотрим теперь различные варианты Qt library с точки зрения средств разработки на платформах Windows и Linux, на которых мы, в основном, и сосредоточим наше внимание.

Qt SDK

Этот дистрибутив содержит все необходимое для того, чтобы сразу приступить к разработке приложений Qt. В него, помимо уже собранных библиотек Qt, входит интегрированная среда разработки Qt Creator (рис. 1.1), полная справочная документация и файлы примеров. Для сборки приложений Qt SDK использует инструментарий GNU. На платформе Linux используется тот инструментарий, который поставляется вместе с Linux, а на платформе Windows — инструментарий MinGW, который входит в дистрибутив.

Примечание

Следует помнить, что объектные файлы, которые генерирует MinGW, несовместимы с объектными файлами, которые создает Microsoft Visual Studio. То же относится и к разделяемым библиотекам, экспортирующим классы C++. Если вы создаете на MinGW разделяемую библиотеку, которую потом можно будет использовать в среде Microsoft Visual Studio, позаботьтесь о том, чтобы все функции экспортировались этой библиотекой в формате C (мы еще скажем об этом позже).

Рис. 1.1. Интегрированная среда Qt Creator

Qt для Microsoft Visual Studio

Существуют библиотеки Qt, специально собранные для Microsoft Visual Studio. Сама по себе среда Visual Studio не приспособлена для работы с Qt (создать заготовку проекта приложения Qt в Visual Studio по умолчанию нельзя). Тут можно пойти двумя путями. Первый путь заключается в том, чтобы для генерации специальных файлов Qt (файлов проектов, ресурсов и т. д.) использовать напрямую соответствующие инструменты Qt (qmake, designer, rcc и т. д.). Затем править исходные тексты, отлаживать и собирать проект в Visual Studio. Можно и вовсе не пользоваться средой Visual Studio, а собирать проект с помощью сопутствующей ей утилиты nmake. Тем более, что файлы, которые генерирует qmake в этом варианте Qt library, предназначены именно для nmake.

Более удобное решение — использовать средство интеграции Qt и Visual Studio, которое позволяет работать с инструментами Qt так, как если бы они были частью интегрированной среды разработки Microsoft (рис. 1.2). Этот инструмент нужно скачать и установить отдельно.

Для компиляции приложения в этом случае используется компилятор Microsoft, который, как считается, генерирует несколько более компактный и быстродействующий код, нежели компилятор GNU. Разумеется, этот вариант доступен только под ОС Windows.

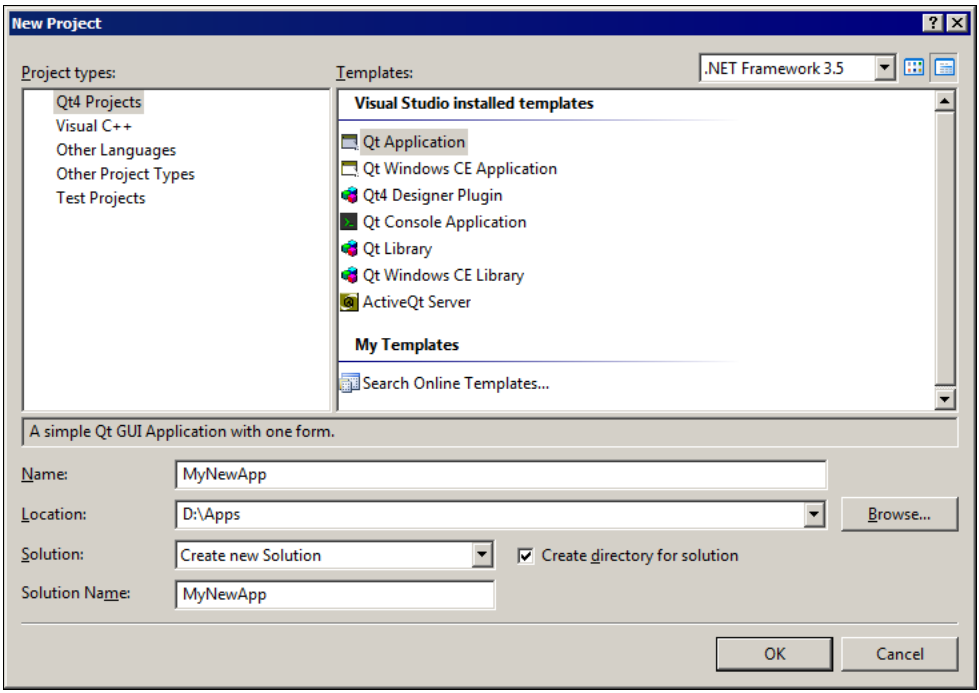


Рис. 1.2. Интеграция Qt и Microsoft Visual Studio (окно выбора типа проекта)

Примечание

При работе с Qt для Microsoft Visual Studio необходимо, чтобы переменные среды окружения были настроены на работу с Qt. Неправильная настройка переменных среды окружения может стать источником неочевидных ошибок в процессе сборки приложения. При установке пакета создаются два ярлыка. Один открывает окно командной строки Qt Command Prompt с уже настроенными переменными среды, второй запускает Microsoft Visual Studio с теми же настройками.

Qt Framework

Этот вариант не содержит никаких компиляторов и интегрированных сред. В него входят только сами библиотеки Qt, вспомогательные инструменты Qt, примеры и документация. Этот вариант Qt можно скачать не только в виде собранных библиотек для различных платформ, но и в виде исходных текстов для самостоятельной сборки (причем собирается все, включая такие программы как qmake). Вариант со сборкой из исходных текстов лучше выбрать тогда, когда вы собираетесь использовать Qt в такой конфигурации, для которой официальной сборки не существует. Кроме того, новейшие релизы Qt зачастую доступны только в таком варианте. Обычно номер доступной версии Qt Framework опережает номер версии Qt в доступной версии Qt SDK. Тем не менее Qt SDK можно настроить на работу с новейшей версией Qt Framework. Чтобы настроить Qt Creator на работу с версией Qt, отличной от той, что входит в состав SDK, необходимо перейти в группу **Проекты** Qt Creator и открыть вкладку **Настройки сборки** (рис. 1.3). Грамотный выбор пара-

метров на этой вкладке позволит настроить Qt Creator на работу с любым вариантом библиотек Qt, который поддерживает ваша платформа, и с любым компилятором. Но подробное описание этого процесса выходит за рамки данной книги. Лично я предпочитаю сочетание Qt Creator, компилятора GNU и новейшей версии Qt Framework. Кстати, если вам нужен только Qt Creator, не обязательно скачивать весь SDK.

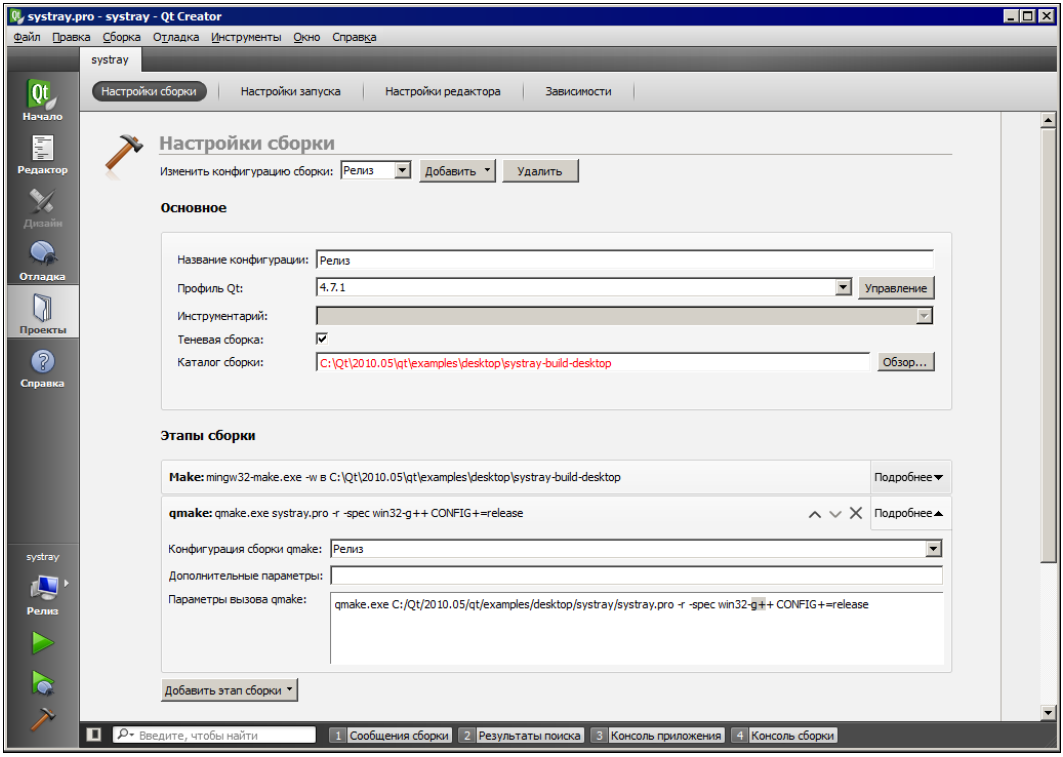


Рис. 1.3. Настройка Qt Creator для работы с Qt Framework другой версии

Qt и Eclipse

Если для разработки вы выбрали версию Qt, использующую инструментарий GNU, но не хотите пользоваться Qt Creator, имеет смысл установить среду интегрированной разработки Eclipse. Для того чтобы среда Eclipse могла работать с C++ (изначально она ориентирована на Java), необходимо установить пакет расширений Eclipse CDT и пакет интеграции Qt и Eclipse (его можно скачать с сайта Qt). Установка пакета интеграции Qt и Eclipse выполняется очень просто: все, что вам придется сделать, — это распаковать архив пакета в ту же директорию, которая содержит директорию Eclipse. Уровень интеграции Qt и Eclipse — примерно такой же, как и уровень интеграции Qt с Microsoft Visual Studio (рис. 1.4). Перед сборкой первого приложения Qt в Eclipse вам потребуется указать версию библиотеки Qt и директорию, в которой расположены такие инструменты Qt как qmake, moc и т. д.

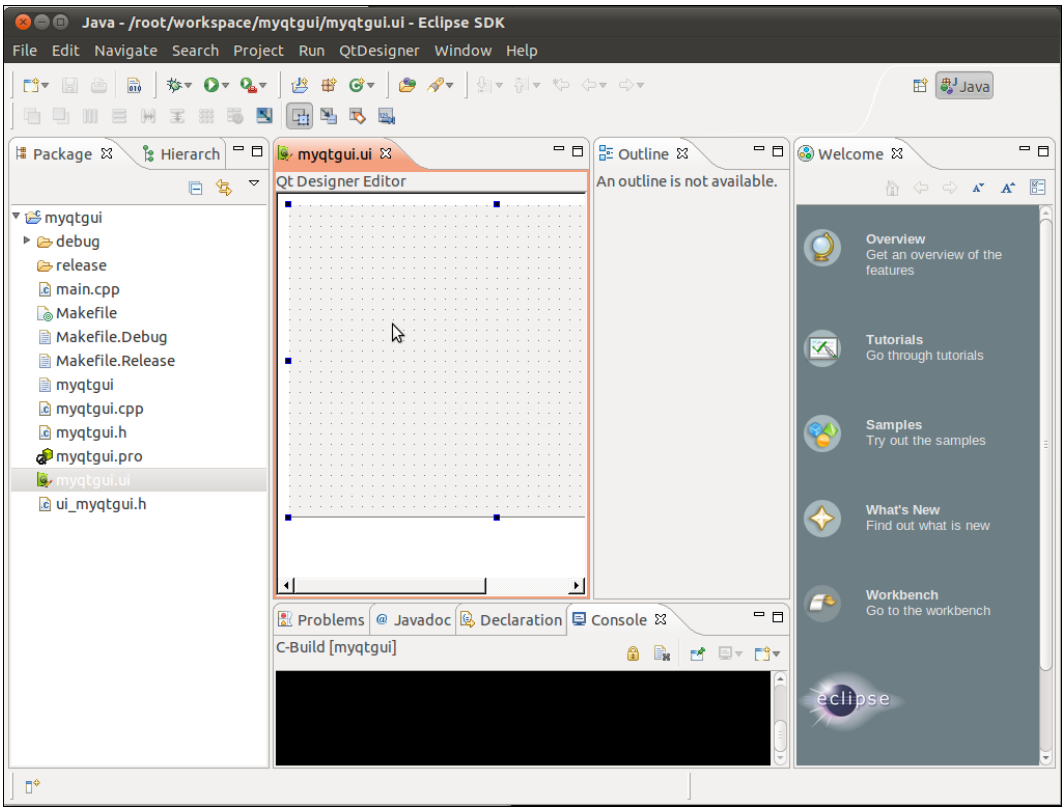


Рис. 1.4. Интеграция Qt и Eclipse

Примечание

Пусть вас не смущает, что пакет интеграции Qt и Eclipse собран не с той версией Qt, которую вы собираетесь использовать. По моим наблюдениям, это никак не влияет на работу с Qt. Например, в моей системе пакет интеграции Qt и Eclipse собран с Qt 4.6.2, а для сборки приложений с его помощью я использую новейшую на данный момент версию Qt 4.7.2. Впрочем, вы так же можете скачать исходные тексты пакета интеграции и собрать его самостоятельно под свою версию Qt.

Qt и CMake

Традиционно программисты, работающие в Linux (и других UNIX-системах), используют для сборки программ инструментарий GNU build system (состоящий из утилит autoconf, automake, libtool, gnulib). Именно с помощью GNU build system создаются знакомые всем нам файлы configure, с помощью которых мы генерируем make-файлы для сборки приложения. Система GNU build system настолько тесно связана с историей Linux и других открытых проектов, что своеобразное заклинание "configure–make–make install" рассматривается некоторыми UNIX-программистами как единственно верный способ установки ПО. К недостаткам GNU build system можно отнести ее ориентацию на инструментарий разработчика GNU (GNU

make, GCC и т. п.), который очень популярен на открытых UNIX-платформах, но мало распространен в других средах.

Стремясь заполнить этот пробел, разработчики из компании Kitware создали свой собственный вариант кросс-платформенного генератора сборочных файлов — CMake (cross-platform make).

Каковы же достоинства CMake? Прежде всего, CMake является по-настоящему кросс-платформенным генератором проектов, позволяющим создавать единые описания проектов для Linux и других UNIX-систем (включая Mac OS X) и Windows. Остановимся на этом подробнее. Важное отличие Windows от Linux (с точки зрения разработчика) заключается в том, что на платформе Windows нет единого стандарта сборочных файлов. MS Visual Studio использует собственные файлы проектов, у C++ Builder они другие, а у MinGW — третьи. Преимущество CMake в том, что эта система способна генерировать "родные" сборочные файлы для всех перечисленных средств разработки (как и для многих других). Кроме того, CMake стремится максимально использовать собственные средства генерации сборочных файлов — например, для генерации сборочных файлов проекта Qt используется qmake. И вообще, разработчики CMake уделили Qt много внимания.

Помимо прочего, CMake обладает интеллектуальной системой поиска инструментов сборки и библиотек на конкретной платформе (интроспекцией) и автоматического конфигурирования. Благодаря этому система CMake сама устанавливает многие параметры сборочных файлов, которые в других системах управления сборкой приходится устанавливать вручную.

Примечание

Для тех, кто совсем незнаком с CMake, дам некоторые пояснения. Схематично работу CMake можно описать следующим образом: для сборки приложения создается файл CMakeLists.txt, в котором описываются параметры сборки (расположение файлов исходных текстов, требуемые внешние модули, цели сборки). Далее этот файл передается утилите cmake. Результатом работы cmake является файл, содержащий инструкции сборки приложения для конкретной платформы (make-файл GNU make, файл проекта Visual Studio и т. д.). Суть идеи заключается в том, что описание процесса сборки в файле CMakeLists.txt абстрагировано как от конкретных особенностей отдельных систем (расположение файлов, возможности компиляторов), так и целых платформ. Читая общее описание процесса сборки из файла CMakeLists.txt, программа cmake создает файл инструкций сборки, учитывающий специфику конкретной системы.

CMake — консольная утилита (для которой, правда, есть графическая оболочка). Если проект Qt поставляется с файлом CMakeLists.txt, то перед сборкой этого проекта необходимо перейти в его директорию и выдать команду:

```
Cmake ./
```

В результате будет создан файл сборки проекта, соответствующий вашей платформе и используемому средству разработки. Дальше, в зависимости от того, какой именно файл был сгенерирован, нужно либо открыть его в интегрированной среде разработки, либо запустить утилиту make или nmake.

Qt Creator умеет работать с проектами, использующими в процессе сборки CMake. Если проект Qt поставляется с файлом CMakeLists.txt, то всю информацию о входящих в проект файлах, параметрах их сборки и компоновки Qt Creator экспортирует из файла CMakeLists.txt. Вы можете редактировать файл CMakeLists.txt в окне редактора Qt Creator, при этом, когда вы захотите сохранить измененный файл, Qt Creator предложит вам выполнить утилиту cmake. Соглашайтесь. Во-первых, это поможет вам сразу убедиться, что отредактированный файл CMakeLists.txt не содержит ошибок (или устранить обнаруженные), а во-вторых, в результате выполнения cmake будут созданы файлы проекта и make-файлы, на основе которых Qt Creator и выполнит сборку приложения в соответствии с вашим замыслом. Microsoft Visual Studio игнорирует файлы CMake, зато CMake не игнорирует Visual Studio и всегда готов сгенерировать файл проекта для этой среды разработки.

Что касается Eclipse, то у него есть даже специальный пакет расширений для работы с CMake, и эта среда интегрируется с CMake так же хорошо, как и Qt Creator.

Если вы намереваетесь распространять свое приложение в виде исходных текстов, которые пользователь должен скомпилировать самостоятельно, использование CMake в проекте Qt является, на мой взгляд, наилучшим выбором.

Общая структура приложений Qt

Для того чтобы понять, что делают различные инструменты Qt, необходимо представлять себе общую структуру приложения Qt. Как и все программы на C++, приложение Qt должно содержать главную функцию `main()`. Обычно функция `main()` приложения Qt делает немного: создает объект класса `QApplication`, создает объекты, управляющие локализацией приложения, создает главный визуальный элемент приложения (обычно это главное окно программы) и запускает цикл обработки сообщений.

Вот, например, типичная функция `main()` настоящего, а не учебного приложения Qt (листинг 1.1).

Листинг 1.1. Функция `main()` типичной программы Qt

```
#include <QApplication>
#include <QTranslator>
#include <QLocale>
#include "mainwindow.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTranslator translator;
        if (!translator.load("yagf_" + QLocale::system().name(),
                "/usr/local/share/yagf/translations"))
            translator.load("yagf_" + QLocale::system().name(),
                "/usr/share/yagf/translations");
```

```
    app.installTranslator(&translator);
        QTranslator translator2;
        translator2.load("qt_" + QLocale::system().name(),
            "/usr/share/qt4/translations");
        app.installTranslator(&translator2);
    MainWindow mainWindow;
    Window.show();
    return app.exec();
}
```

Класс `QApplication` играет центральную роль в работе приложения Qt. У класса `QApplication` несколько конструкторов, мы выбираем из них конструктор с двумя параметрами, аналогичными параметрам функции `main`. Получив переменные `argc` и `argv`, конструктор извлекает из них ключи, которые относятся к конфигурации приложения на данной платформе. Например, если приложение создано для платформы Linux/X Window, то в качестве дополнительных ключей ему могут быть переданы дисплей для вывода и геометрия окна. Таким образом, используя объект класса `QApplication`, мы "бесплатно" получаем приложение, которое ведет себя в соответствии со стандартами X Window. Далее следует блок, связанный с локализацией приложения. *См. разд. "Инструменты интернационализации и локализации"* далее в этой главе.

Затем мы создаем объект `mainWindow` класса `MainWindow`. Это и есть главное окно нашей программы. Соответствующий класс объявлен в файле mainwindow.h, а определен в файле mainwindow.cpp. Таким образом, данное приложение включает в себя, как минимум, три файла: main.cpp, в котором реализована функция `main()`, mainwindow.h и mainwindow.cpp, в которых объявлен и реализован класс `MainWindow`. На самом деле, в проекте этого приложения гораздо больше файлов, но перечисленное выше составляет необходимый минимум. Обратите внимание, `mainWindow` объявлена как локальная переменная функции `main()`. Это логично, поскольку выход из функции `main()` в любом случае приведет к завершению программы.

Последнее, что мы делаем в нашей программе, — запускаем цикл обработки сообщений с помощью метода `exec()` объекта `app`. Цикл будет выполняться до тех пор, пока приложение не получит команду завершить работу. Цикл обработки сообщений запускается с помощью метода `exec()` объекта `app` класса `Application`. Когда приложение завершится, метод `exec()` вернет значение, которое будет передано оператору `return` функции `main()`. Нетрудно догадаться, что это значение является кодом завершения программы.

Инструменты Qt

В этом разделе мы перечислим те инструменты, которые распространяются со всеми дистрибутивами Qt и без которых нельзя серьезно заниматься программированием с помощью Qt.

Утилита qmake

Если говорить коротко, утилита qmake предназначена для генерации файлов проектов и файлов, управляющих сборкой приложений. Допустим, что у вас в некоторой директории есть набор исходных файлов программы Qt (файлы с расширениями cpp, h, ui, qrc и т. д.). Важно, чтобы это действительно были файлы программы Qt, а не чего-то другого. Перейдите в эту директорию и выдайте команду:

```
qmake -project
```

В результате в директории появится файл с расширением pro. Это заготовка файла проекта вашего приложения, автоматически созданная утилитой qmake. Если вы откроете этот файл, то увидите в нем набор переменных, которым присваиваются некие значения. Как минимум, файл проекта содержит переменные `CONFIG`, `HEADERS`, `SOURCES`, `FORMS` и `TARGET`. Переменная `CONFIG` содержит информацию, необходимую для системы сборки, ее значениями могут быть наименования конфигураций сборки: `debug`, `release` и др. Переменные `HEADERS` и `SOURCES` содержат, соответственно, перечни заголовочных файлов приложения и файлов с расширением cpp. Переменная `FORMS` содержит перечень файлов с расширением ui (подробнее об этих файлах будет рассказано в разделе, посвященном Qt Designer). Переменная `TARGET` определяет, что именно является результатом сборки приложения. Файл проекта, созданный программой qmake, почти всегда включает в себя и другие переменные и даже более сложные конструкции. Рассмотрим типичный файл проекта, созданный программой qmake с нуля (листинг 1.2).

Листинг 1.2. Пример файла проекта

```
TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += mainwindow.h
FORMS += mainwindow.ui
SOURCES += main.cpp mainwindow.cpp
```

Откуда qmake "узнал", что исходные тексты программы содержатся в файлах main.cpp и mainwindow.cpp, что описание внешнего вида главного окна содержится в файле mainwindow.ui, а объявление класса — в файле mainwindow.h? Никакой магии здесь нет. Программа просто рассортировала файлы, которые она нашла в той директории, где была запущена. Сортировка выполнялась по именам файлов, а не по их содержимому, так что qmake могла и ошибиться.

Если теперь мы просто выдадим команду:

```
qmake
```

то в нашей директории появятся один или несколько make-файлов, с помощью которых уже можно собрать приложение. Какие это будут make-файлы, зависит от

конфигурации Qt, которую мы используем. Если наши библиотеки ориентированы на Microsoft Visual Studio, то это будут файлы для утилиты nmake. Если наша версия Qt использует инструментарий GNU, то это будут make-файлы для GNU make.

Утилита qmake умеет создавать не только make-файлы. Если при работе с версией Qt, предназначенной для Microsoft Visual Studio, мы выдадим команду:

```
qmake -tp vc -o untitled.vcproj
```

то результатом станет файл проекта Microsoft Visual Studio.

Будет ли сборка приложения успешной, если мы выполним созданные таким образом make-файлы или попробуем собрать проект Microsoft Visual Studio? В случае простейшего приложения, скорее всего, да. В случае более сложном, скорее всего, нет. Дело в том, что, как отмечалось ранее, утилита qmake создает заготовку проекта приложения по формальным признакам. В случае сложного приложения эта заготовка, скорее всего, будет включать в себя далеко не все, что нужно приложению. Соответственно и созданные с помощью qmake файлы, управляющие сборкой, будут неполноценными. Интегрированные среды разработки с подключенными модулями интеграции Qt стараются помочь вам сформировать правильный файл проекта путем подсказок. Например, вы можете выбрать, какие именно подсистемы Qt (OpenGL, сетевую, связи с базами данных) будет использовать ваше приложение. Все эти сведения попадут в файл проекта, который интегрированная среда разработки создаст с помощью той же утилиты qmake. Однако и этого может оказаться недостаточно, и тогда файл проекта придется редактировать вручную.

Не забывайте, что вызов

```
qmake -project
```

всегда создает новый файл проекта (и затирает все, что вы внесли туда вручную), тогда как вызовы qmake без ключа `-project` не меняют содержимое файла с расширением pro. Они, как правило, генерируют файлы, непосредственно управляющие сборкой приложения в данной системе.

Если вы собираетесь выполнять сборку приложения на нескольких разных платформах (или с использованием разных компиляторов на одной платформе), вы можете создать один кросс-платформенный файл с расширением pro и использовать его на всех целевых платформах. Правда, при создании такого файла вам не удастся избежать его ручного редактирования (и потребуется хорошее знание синтаксиса файлов с расширением pro). В дальнейших примерах к этой книге мы будем использовать кросс-платформенные файлы проектов, но подробные инструкции по их составлению опустим. Все это хорошо описано в документации по Qt, доступной, в том числе, и на русском языке. Вы можете спросить, зачем нам нужны специальные кросс-платформенные файлы проектов, если сама библиотека Qt уже является кросс-платформенной. Вопрос справедливый. Для многих приложений, в том числе довольно сложных, файл с расширением pro будет выглядеть одинаково на всех платформах именно благодаря кросс-платформенности Qt. Однако в некоторых случаях в файлах проектов приходится описывать механизмы взаимодействия приложения с системой, которые зависят от системы, а не от Qt. И тогда в файле про-

екта приходится принимать специальные меры для обеспечения кросс-платформенности.

После всего этого у читателя может возникнуть вопрос, зачем нам описанная выше система CMake, если у нас есть qmake. Ответ прост. Хотя утилиты cmake и qmake очень похожи (и та, и другая используют файлы мета-проектов для генерации инструкций для конкретной системы сборки), qmake более ориентирована на разработчиков конкретного приложения (а не на пользователей, которым придется собирать это приложение из исходных текстов). Вспомните, что неправильный вызов qmake может привести к потере всего содержимого файла с расширением pro. Кроме того, CMake — универсальная система управления сборкой, которая используется не только для Qt. У сборщика вашей программы может не быть опыта работы с qmake, но наверняка есть опыт работы с CMake.

Если же вы не собираетесь распространять исходные тексты программы, то система CMake вам, скорее всего, не нужна. Впрочем, не исключено, что нужна, ведь она, помимо прочего, является прекрасным средством обеспечения кросс-платформенности вашего проекта.

Инструменты интернационализации и локализации

Когда величайшие мудрецы человечества учили свои народы грамоте, они, к сожалению, не предвидели, что в далеком будущем текст будут вводить с клавиатуры. Мало того, что в одних языках пишут слева направо, а в других — справа налево, начертание символов в некоторых языках (например, арабском) зависит от соседних символов. Все это нужно учитывать, если вы хотите выполнить грамотную интернационализацию своей программы. Для демонстрации возможностей интернационализации мы напишем максимально простую программу (листинг 1.3).

Листинг 1.3. Программа, подготовленная для интернационализации

```
#include <QApplication>
#include <QPushButton>
#include <QTranslator>
#include <QLocale>
int main(int argc, char *argv[])
{
QApplication I18nApp(argc, argv);
    QTranslator Translator(0);
    Translator.load(QString(QApplication::applicationDirPath() +
        QString("/I18nApp_") + QLocale::system().name()) , ".");
    I18nApp.installTranslator(&Translator);
    QPushButton Button(QObject::tr("Push me!"), NULL);
    Button.show();
    return I18nApp.exec();
}
```

Исходные тексты этой программы вы найдете в папке Ch1/I18n на сайте **http://symmetrica.net/qt47book/**). Пояснять все, что происходит в этой программе, я не буду. Если вы хотя бы немного знакомы с Qt и читали предыдущие разделы, вы все поймете. Программа выводит на экран кнопку с надписью "Push me!" ("Нажми меня!"). Наша задача — сделать так, чтобы текст интерфейса программы (состоящий из одной строчки!) можно было перевести на любой язык, не вмешиваясь в исходные тексты самой программы.

Откройте файл I18nApp.pro (листинг 1.4).

Листинг 1.4. Файл I18nApp.pro

```
SOURCES += \
    main.cpp
TRANSLATIONS += I18nApp_ru.ts
```

Этот файл проекта даже проще, чем файл проекта, рассмотренный нами ранее. Но в нем появилась переменная, с которой мы раньше не встречались. Переменная `TRANSLATIONS` содержит перечень имен файлов, в которых находятся исходные тексты для перевода интерфейса программы. Собственно говоря, расширение файла ts как раз и является сокращением от translation source. Имена файлов, содержащих исходные тексты перевода, могут быть любыми, но далее мы убедимся, что очень удобно, когда они содержат в качестве суффикса или префикса название соответствующей локали (ru — для русской, fr — французской, de — немецкой и т. д.). Если вы захотите перевести интерфейс программы и на другие языки, вам понадобятся дополнительные файлы с именами, соответствующими этой же схеме.

Файла I18nApp_ru.ts в директории I18nApp нет. Вы должны создать его сами, это несложно. Просто в окне консоли выдайте команду:

```
lupdate I18nApp.pro
```

При правильном выполнении команды в окне консоли вы увидите примерно следующее:

```
Updating 'I18nApp_ru.ts'...
    Found 1 source text(s) (1 new and 0 already existing)
```

Утилита lupdate является частью инструментария Qt (поэтому приведенная выше команда сработает только в том случае, если среде окружения известно, где она находится). В рассматриваемом примере программа lupdate делает следующее: из файла I18nApp.pro она извлекает имена файлов исходных текстов (в нашем примере — файл main.cpp), а из этих файлов извлекает все текстовые строки, помеченные особым образом. Далее, из того же файла I18nApp.pro извлекает имена файлов исходных текстов перевода (в нашем примере — файл I18nApp_ru.ts) и записывает в них строки, которые нужно перевести.

Теперь, когда у нас появились заготовки файлов перевода, мы можем приступить к самому переводу. Для этой цели нам понадобится графическая утилита Qt Linguist (рис. 1.5). Интерфейс этой утилиты очевиден. Она показывает нам строку кода, из

которой был извлечен текст для перевода, сам текст и статус перевода. Запустив Qt Linguist, мы последовательно открываем созданные нами файлы с расширением ts, выбираем в левом окне объект создаваемой программы, предназначенный для перевода. При этом в главном окне утилиты отображаются все связанные с объектом строковые ресурсы. Обратите внимание, что Qt Linguist "знает", что мы переводим именно на русский. Все дело в суффиксе ru, который мы добавили к имени файла, содержащего исходный текст перевода. Переведя очередную фразу, необходимо пометить перевод как законченный. При этом слева от исходной фразы появится зеленая галочка.

Между прочим, файлы с расширением ts — это текстовые файлы в формате XML, так что если вы разберетесь в их структуре, то сможете добавлять переводы в

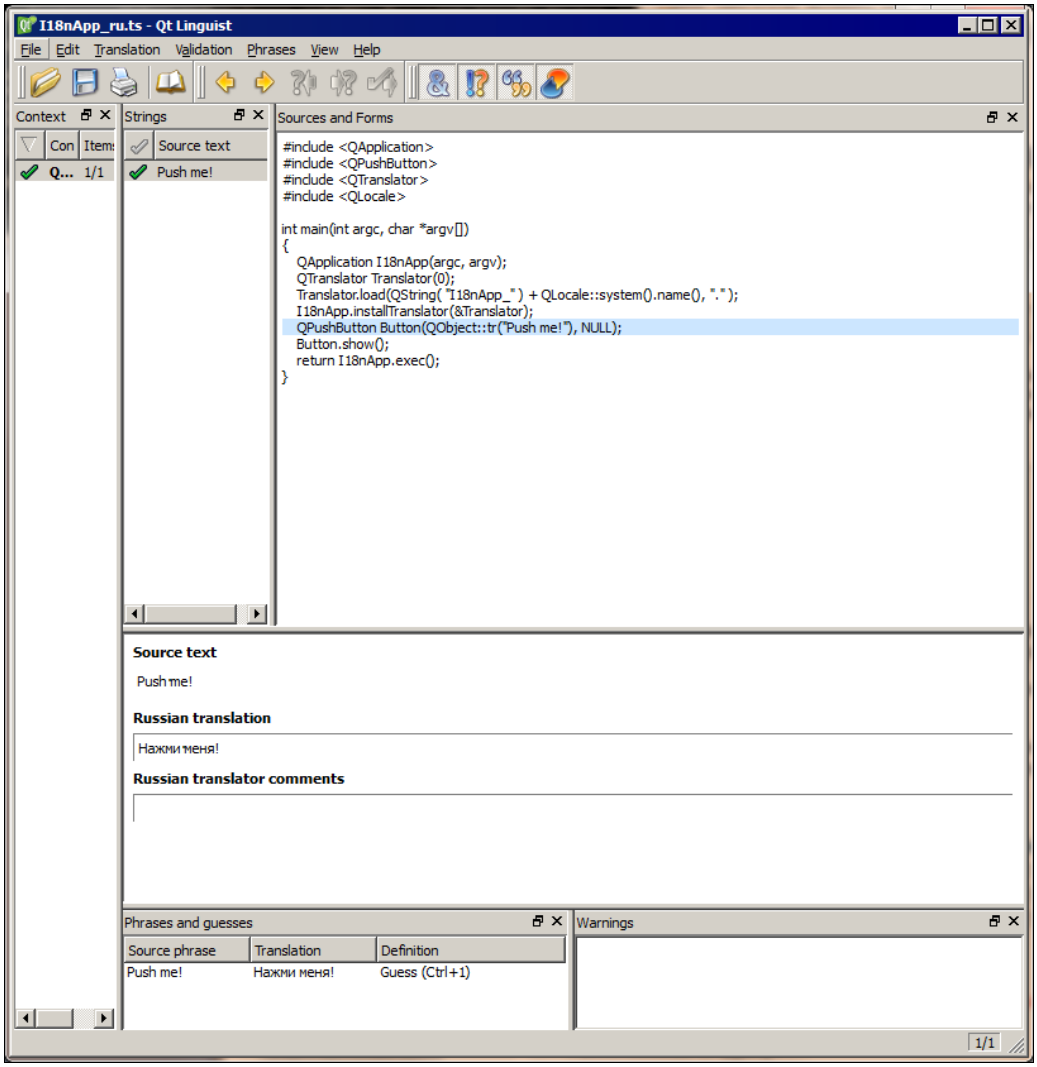


Рис. 1.5. Qt Linguist

обычном текстовом редакторе, не пользуясь Qt Linguist, хотя, на мой взгляд, это неудобно.

Итак, у нас есть файл, содержащий перевод интерфейса нашей программы на русский язык, но работа на этом далеко не закончена. Созданные файлы можно рассматривать как исходные тексты файлов ресурсов перевода. Их еще нужно скомпилировать с помощью утилиты lrelease. В окне консоли выдадим команду:

```
lrelease I18nApp.pro
```

В результате появится файл I18nApp_ru.qm. Это и есть готовый ресурс для перевода интерфейса программы на русский язык. Скопируйте его в ту же директорию, где находится исполнимый файл программы (если его там еще нет). Если вы теперь запустите исполнимый файл программы, то увидите, что она "заговорила" по-русски (рис. 1.6).

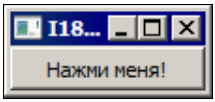


Рис. 1.6. Русифицированная программа

Примечание

Теперь у меня есть еще один довод в пользу применения CMake. Посмотрите, сколько инструментов нам пришлось использовать для того, чтобы собрать файлы перевода в готовый ресурс перевода. CMake бы сделал все это автоматически.

Теперь вернемся к исходному тексту нашей программы. Класс `QTranslator` загружает файлы ресурсов интернационализации во время выполнения программы и управляет переводом интерфейса на соответствующий язык. Как определить, какой файл ресурсов должен быть загружен? Метод `QLocale::system().name()` возвращает имя текущей локали. Мы добавляем это имя в качестве суффикса к имени файла ресурса. Здесь возникает одна интересная ситуация. Метод `QLocale::system().name()` возвращает строку наподобие `"ru_RU"` (`"язык_страна"`), так что сгенерированное нами имя файла будет выглядеть как I18nApp_ru_RU. Наш модуль ресурсов локализации называется I18nApp_ru.qm. Как же объекту класса `QTranslator` удается загрузить правильный файл? Дело в том, что этот объект учитывает, что переданный ему шаблон имени файла — всего лишь подсказка. Сначала он пытается найти файл с точно таким именем и расширением qm. Затем, если такой файл не найден, он отрезает расширение qm и снова пытается найти файл. В случае неудачи снова добавляет расширение qm, но "откусывает" от имени шаблона суффикс. В качестве разделителя суффиксов по умолчанию используется символ "_". То есть объект отрежет суффикс `"_RU"` и с третьей попытки загрузит "правильный" файл I18nApp_ru.qm. Обратите внимание также на то, что Qt не делает разницы между прямым и обратным слэшами при разделении директорий, так что их можно даже смешивать в одной ссылке на файл.

Еще интереснее обстоит дело с директориями. В нашем примере файл ресурса локализации находится в той же директории, что и исполнимый файл программы. Это

логично, но невсегда соответствует правилам. Например, в UNIX-системах файлы программ и файлы ресурсов, как правило, хранятся раздельно. Например, исполнимый файл приложения может храниться в директории /usr/bin/ (это только один из возможных вариантов), а файлы ресурсов перевода — в директории /usr/shre/*имя_приложения*/translations/.

> ***Примечание***
>
> Для UNIX-систем существует также спецификация XDG Base Directory Specification, которая определяет, в каких именно директориях следует хранить различные файлы приложения. Если вы планируете разрабатывать приложения для Unix-систем, я рекомендую вам ознакомиться с этим документом (он доступен в Интернете).

В Windows традиционным местом хранения исполнимого файла программы и файлов ресурсов является директория C:\Program Files*имя_приложения*\, причем внутри этой директории приложение может создавать любые иерархии. Папка C:\Program Files*имя_приложения*\ не подходит для хранения локальных ресурсов отдельных пользователей системы, т. к. не у всех пользователей может быть разрешение на запись в эту папку (в грамотно настроенной системе так и должно быть). Мы подробнее поговорим об этом позже.

Вернемся к нашему коду. Метод `installTranslator()` класса `QApplication` устанавливает в приложении объект, выполняющий перевод интерфейса. Теперь для перевода любой строки, содержащей эквивалент в файлах ресурсов, можно воспользоваться статическим методом `tr()` класса `QObject`. Метод `tr()` указывает, что переданная ему строка должна быть переведена. Этот же метод ищет утилита lupdate, когда извлекает из файла исходных текстов строки для перевода. У метода `tr()` есть аналог — метод `trUtf8()`, он делает то же, что и `tr()`, только предполагает, что текст передан ему в кодировке UTF-8. Каким методом пользоваться? Это, прежде всего, зависит от того, в какой кодировке вы пишете исходные тексты программы. Ведь исходная строка для перевода берется именно из файла исходного текста. Под Windows вы, скорее всего, будете использовать локальную восьмибитную кодировку, и для нее подойдет метод `tr()`. Под Linux сейчас практически везде используется кодировка UTF-8, и здесь понадобится метод `trUtf8()`. Вы так же можете явным образом определить, в какой кодировке представлены строки, предназначенные для перевода. Это можно сделать в файле проекта вашего приложения с помощью переменной `CODECFORTR`, например:

```
CODECFORTR = UTF-8
```

или программно, в функции `main()`, с помощью класса `QTextCodec` (листинг 1.5).

Листинг 1.5. Файл I18nApp.pro

```
QTextCodec::setCodecForLocale(QTextCodec::codecForName("UTF-8"));
QTextCodec::setCodecForTr(QTextCodec::codecForName("UTF-8"));
QTextCodec::setCodecForCStrings(TextCodec::codecForName("UTF-8"));
```

В этом листинге мы указали, что для всех строк нашей программы используется кодировка UTF-8.

Разделение контекстов

Иногда одно и то же английское слово необходимо перевести по-разному. Например, слово build может, в зависимости от контекста, означать либо "собрать", либо "сборка", либо еще что-нибудь. Система перевода Qt предусматривает различные переводы одного и того же слова для разных контекстов. Например, вы можете указать:

```
fileMenu = menuBar()->addMenu(tr("&Build"), "action");
```

Здесь строка `"action"` указывает контекст для перевода. Указание контекста попадет в файл в расширением ts и будет видно вам в программе Qt Linguist.

Склонение существительных, следующих после числительных

Думаю, при работе за компьютером всем приходилось видеть сообщения типа "сохранено 23 записей". Как писать такие строки правильно? Я имею в виду, как писать такие строки так, чтобы при переводе на другие языки имена существительные, следующие за числительными или цифрами, склонялись так, как этого требуют правила соответствующего языка? Можно, конечно, прибегнуть к сухому складскому стилю: "сохраненных записей: 23" Но можно научить программу говорить и более человеческим языком.

Для примера добавьте в функцию `main()`, перед вызовом метода `exec()`, такие строки:

```
int n =3;
QMessageBox::information(0, QObject::tr("info"),
    QObject::tr("%n plane(s) in the sky", "", n));
```

И не забудьте включить в файл main.cpp заголовочный файл `<QMessageBox>`.

Далее вызовите утилиту lupdate, как было описано ранее. Запустите Qt Linguist. Теперь поле для перевода выглядит иначе, чем в предыдущем случае (рис. 1.7). Когда утилите Qt Linguist попадается для перевода конструкция `tr("%n plane(s) in the sky", "", n)`, утилита предлагает нам три варианта перевода: единственное число, "двойное" число и множественное число. Qt Linguist "знает", что мы переводим на русский (ранее я объяснил, почему). Она также "знает", что в русском языке существует три формы склонения существительных, следующих после числительных. Одна форма — для числительных, заканчивающихся на 1 (эта форма названа единственным числом, хотя она применима к любому числу, заканчивающемуся на 1), другая форма для числительных, заканчивающихся на 2, 3, 4 (это то, что мы назвали "двойственным числом"), и третья форма — для числительных, заканчивающихся на 5, 6, 7, 8, 9, 0. Заполните в Qt Linguist все три формы, как это сделал я, скомпилируйте ресурс перевода с помощью lrelease и скопируйте его в директорию с исполнимым файлом. Экспериментируя с различными значениями `n`, вы можете убедиться, что перевод всегда выполняется в соответствии с правилами русского языка (4 самолета в небе, 5 самолетов в небе, 21 самолет в небе и т. д.).

Примечание

Как бы "умна" не была система перевода Qt, она не подскажет нам, как склоняется существительное кочерга с числительным 5, 6 и другими. Впрочем, я надеюсь, что вам не придется писать программу для учета этих самых... объектов "кочерга".

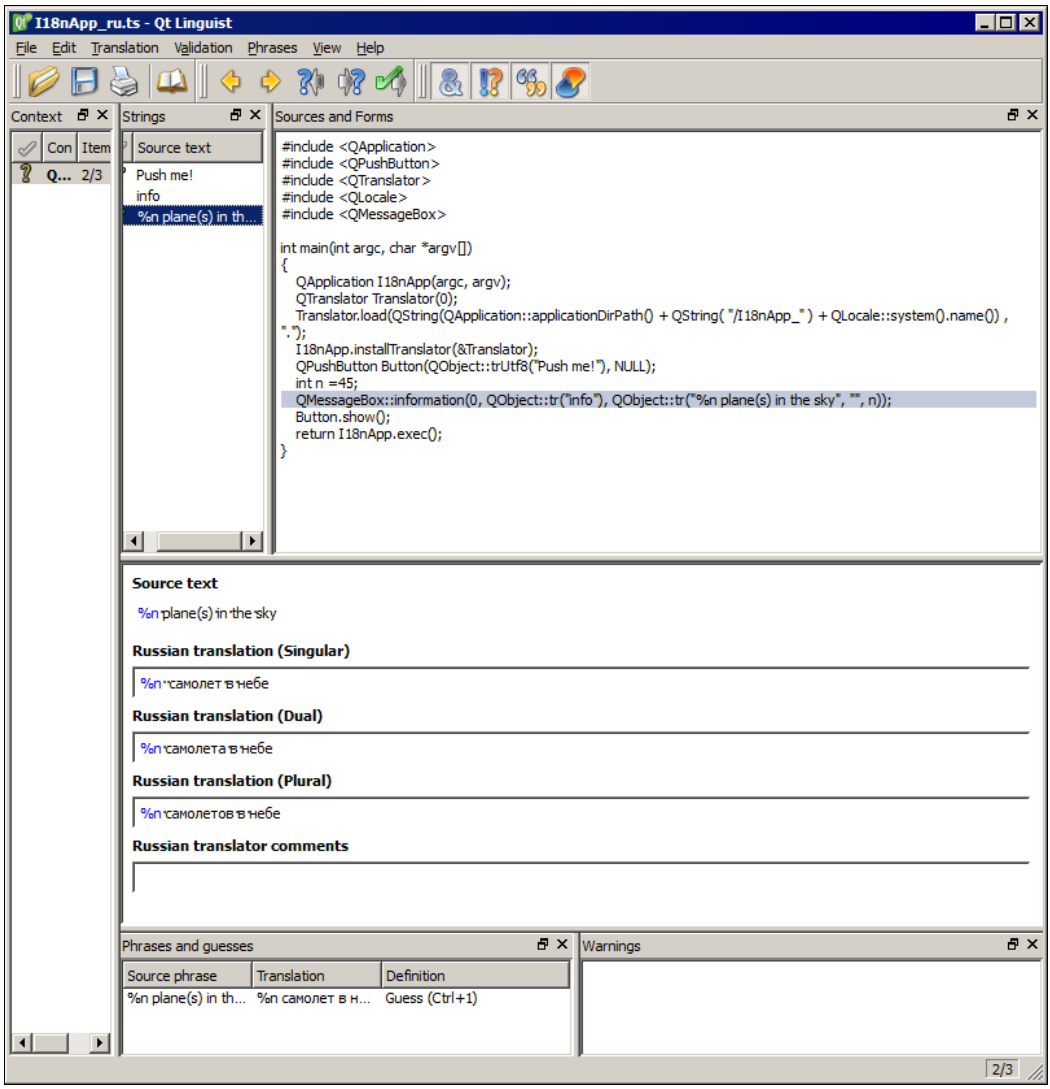


Рис. 1.7. Qt Linguist "умеет" обращаться с числительными в зависимости от языка перевода

Между прочим, строка `"%n plane(s) in the sky"` нарушает правила любого языка, в том числе и английского. Использование числительных — это тот редкий случай, когда нам придется делать перевод интерфейса и на английский. Вы, возможно, спросите меня, почему исходные строки для перевода нужно писать именно на английском. На самом деле это вовсе не обязательно. Но подумайте об этом так: если вы вообще никогда не собираетесь переводить интерфейс программы на другие языки, то пишите сразу на русском. При этом, правда, нужно учесть, что Qt использует для вывода текста кодировку UTF-16, а ваш текстовый редактор, скорее всего, какую-то другую. Так что позаботьтесь о том, чтобы все текстовые строки тексте вашей программы были конвертированы из "родной" кодировки в UTF-16 (далее я

покажу, как это делается). Если же вы все же планируете перевод интерфейса на другие языки, то вам наверняка потребуется в этом деле помощь носителей соответствующих языков (особенно, если вы планируете распространять программу с открытыми исходными текстами). И тут английский лучше всего подходит в качестве базового языка текстовых сообщений.

Qt Designer

Прежде чем перейти к описанию инструмента визуального программирования Qt Designer, сделаем небольшое отступление. Его могут пропустить те, кто уже знаком с методами компоновки визуальных элементов управления (виджетов) Qt.

Компоновка виджетов

Любое приложение из реального мира содержит множество визуальных элементов, расположенных в главном и вспомогательных окнах программы наиболее удобным для пользователя образом. Расположение визуальных элементов — не такое простое дело, как может показаться изначально. Как должны вести себя виджеты, когда меняется размер окна? Как должна вести себя текстовая метка при изменении размера шрифта, заданного по умолчанию? Наконец, что делать, если все виджеты просто не помещаются на экране при данном разрешении? Грамотные проектировщики интерфейсов должны заранее ответить на все эти вопросы.

Расположением виджетов в окнах программ Qt управляют менеджеры компоновки (layout managers). К стандартным менеджерам компоновки относятся классы:

- `QVBox`, который располагает дочерние элементы вертикально друг под другом;
- `QHBox`, который располагает элементы горизонтально;
- `QGrid`, который располагает дочерние элементы в виде таблицы;
- `QFormLayout`, который специально предназначен для создания форм ввода (фактически это таблица из двух столбцов, в первом из которых располагаются текстовые метки с пояснениями, а во втором — виджеты для ввода соответствующих данных).

В принципе, мы не обязаны пользоваться менеджерами компоновки. Можно просто "накидать" виджетов в главное окно приложения. Но в таком случае нам будет гораздо труднее решать те задачи, с перечисления которых я начал этот раздел.

Хотя это может быть сразу и не очевидно, указанные выше классы могут служить основой самого сложного интерфейса. Наглядный пример — редактор Microsoft Word, в котором я пишу этот текст. В первом приближении интерфейса редактор содержит пять элементов, расположенных вертикально: строку главного меню, две панели инструментов (их может быть больше или меньше), окно ввода текста и строку состояния в нижней части окна. Фактически основой интерфейса для такой сложной программы мог бы стать элемент `QVBox`! Конечно, в реальности все не так просто, и `QVBox` не обладает всей необходимой функциональностью. Однако этот

пример демонстрирует основную идею — построение сложных интерфейсов начинается с создания простых элементов управления.

Для демонстрации этого мы напишем приложение просмотра шрифтов, обладающее довольно сложным интерфейсом, и начнем с простого элемента — главного окна, которое станет контейнером для всех графических элементов приложения. Такое окно нам предоставляет класс `QDialog`. Вопреки своему названию этот класс создает простое окно общего назначения, а не диалоговое окно. Разработка приложений Qt немыслима без объектно-ориентированного программирования, к которому теперь прибегнем и мы. Весь интерфейс и логика работы нашего приложения будут содержаться в классе `MainForm`, потомке класса `QDialog`. Для того чтобы было понятно, что мы делаем, лучше всего сразу взглянуть на скриншот конечной программы (рис. 1.8).

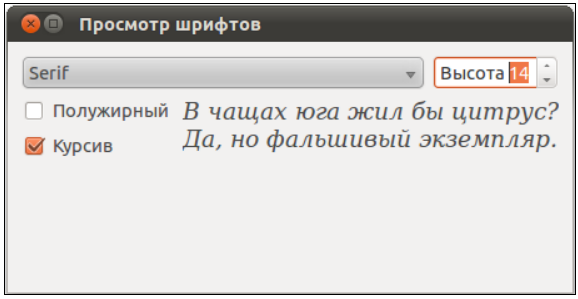


Рис. 1.8. Программа просмотра шрифтов

Прежде чем представить исходный текст программы (листинг 1.6), дам небольшое пояснение относительно кодировок. В этой программе мы не собираемся заниматься интернационализацией и все пишем по-русски. При этом исходные тексты приложения были набраны в Linux, а по умолчанию в современных дистрибутивах используется кодировка UTF-8, и, значит, набранный в редакторе текст нашей программы, включая строки на русском языке, тоже будет записан в UTF-8. Как мы уже знаем, этот текст надо преобразовать в UTF-16, иначе он не будет отображаться правильно. В решении этой проблемы нам на помощь приходит метод `trUtf8()`. Как уже упоминалось, этот метод выполняет перевод интерфейса. Но если ресурс для перевода отсутствует, `trUtf8()` просто перекодирует строку из UTF-8 в UTF-16. Эту особенность метода `trUtf8()` мы и используем для перекодировки всего статического текста в нашей программе.

Примечание

А будет ли эта программа, со всеми преобразованиями из UTF-8 в UTF-16, правильно работать в Windows? Безусловно. Ведь от того, что мы перенесем файлы исходных текстов из Linux в Windows, их кодировка (UTF-8) не изменится (если конечно, вы не конвертируете ее явным образом). Да и в самой Windows ничто не мешает вам писать тексты программ в UTF-8. Пожалуй, это даже лучше, чем использовать восьмибитную кодировку Windows или Unicode. Поскольку для работы с кодировкой UTF-8 всегда приходится использовать методы, специально на нее рассчитанные, ваша программа станет независимой от "родной" кодировки той платформы, на которой вы ее собираете, и ее проще будет переносить между платформами.

Листинг 1.6. Объявление и определение класса `MainForm`

```
#ifndef MAINFORM_H
#define MAINFORM_H

#include <QDialog>
class QLayout;
class QHBoxLayout;
class QSpinBox;
class QComboBox;
class QVBoxLayout;
class QCheckBox;
class QLabel;

class MainForm : public QDialog
{
        Q_OBJECT
public:
        MainForm( QWidget* parent = 0, Qt::WindowFlags f = 0 );
        ~MainForm() {}
private:
        QVBoxLayout * MainFormLayout;
        QHBoxLayout * ChooseFontLayout;
        QHBoxLayout * ViewTextLayout;
        QComboBox * SelectFontBox;
        QSpinBox * SetSize;
        QVBoxLayout * SetFontGroup;
        QCheckBox * SetBold;
        QCheckBox * SetItalic;
        QLabel * TextLabel;
private slots:
        void setLabelFont();

};

#endif // MAINFORM_H

#include "mainform.h"
#include <QLayout>
#include <QSpinBox>
#include <QComboBox>
#include <QVBoxLayout>
#include <QCheckBox>
#include <QLabel>
#include <QFont>
#include <QFontDatabase>
```

```
MainForm::MainForm(QWidget* parent, Qt::WindowFlags f)
 : QDialog(parent, f)
{
    setWindowTitle(trUtf8("Просмотр шрифтов"));
    resize(300, 200);
    MainFormLayout = new QVBoxLayout(this);
    ChooseFontLayout = new QHBoxLayout();
    MainFormLayout->addLayout(ChooseFontLayout);
    SelectFontBox = new QComboBox(this);
    ChooseFontLayout->addWidget(SelectFontBox, 0, Qt::AlignTop);
    SetSize = new QSpinBox(this);
    SetSize->setMinimum(1);
    SetSize->setMaximum(72);
    SetSize->setValue(12);
    SetSize->setPrefix(trUtf8("Высота "));
    ChooseFontLayout->addWidget(SetSize, 0, Qt::AlignTop);
    QSizePolicy fixedSP = QSizePolicy(QSizePolicy::Fixed,
        QSizePolicy::Fixed, QSizePolicy::DefaultType);
    SetSize->setSizePolicy(fixedSP);
    ViewTextLayout = new QHBoxLayout();
    MainFormLayout->addLayout(ViewTextLayout);
    SetFontGroup = new QVBoxLayout();
    SetBold = new QCheckBox(this);
    SetFontGroup->addWidget(SetBold);
    SetBold->setText(trUtf8("Полужирный"));
    SetItalic = new QCheckBox(this);
    SetFontGroup->addWidget(SetItalic);
    SetItalic->setText(trUtf8("Курсив"));
    ViewTextLayout->addLayout(SetFontGroup);
    TextLabel = new QLabel(this);
    TextLabel->setAlignment(Qt::AlignLeft);
    TextLabel->setText(trUtf8("В чащах юга жил бы цитрус?\nДа,
        но фальшивый экземпляр."));
    ViewTextLayout->addWidget(TextLabel, 0, Qt::AlignTop);
    MainFormLayout->addStretch(1);

    connect(SelectFontBox, SIGNAL(activated(int)), this,
        SLOT(setLabelFont()));
    connect(SetSize, SIGNAL(valueChanged(int)), this,
        SLOT(setLabelFont()));
    connect(SetBold, SIGNAL(toggled(bool)), this, SLOT(setLabelFont()));
    connect(SetItalic, SIGNAL(toggled(bool)), this,
        SLOT(setLabelFont()));
    QFontDatabase fontDatabase;
    SelectFontBox->addItems(fontDatabase.families());
    setLabelFont();
}
```

```
void MainForm::setLabelFont()
{
        QString family = SelectFontBox->currentText();
        int size = SetSize->value();
        int weight = QFont::Normal;
        if(SetBold->isChecked()) weight = QFont::DemiBold;
        bool italic = SetItalic->isChecked();
        QFont font(family, size, weight, italic);
        TextLabel->setFont(font);
}
```

Объявление главного класса `MainForm` сохраним в файле mainform.h. Рассмотрим конструктор класса `MainForm`. Свои параметры конструктор `MainForm` передает конструктору базового класса. Далее мы создадим объект `MainFormLayout` класса `QVBoxLayout`. Класс `QVBoxLayout` сам по себе не представляет визуального элемента. Его задача заключается в управлении расположением дочерних визуальных компонентов на родительской форме. Название класса `QVBoxLayout` похоже на название класса `QVBox`, и это не случайно. Класс `QVBoxLayout` управляет дочерними элементами так же как `QVBox`, т. е. располагает их вертикально, один под другим. Класс `QHBoxLayout` выстраивает элементы горизонтально. Почему мы выбрали классы `QDialog` и `QVBoxLayout`, а не класс `QVBox`? Преимущество `QVBoxLayout` заключается в том, что мы можем использовать вместе с ним другие классы, управляющие расположением элементов. Далее мы увидим, как это делается. Конструктору объекта `MainFormLayout` передается ссылка на окно `MainForm` (с помощью указателя `this`). Таким образом, мы сообщаем классу, что он будет управлять расположением элементов этого окна. Теперь посмотрим на скриншот. На нем можно выделить две вертикально расположенные группы, в каждой из которых элементы расположены горизонтально: раскрывающийся список и наборный счетчик в верхней группе, флажки и текстовая метка — в нижней группе. Для того, чтобы получить эти группы, мы создаем два объекта класса `QHBoxLayout` для горизонтального расположения элементов и делаем их дочерними элементами `MainFormLayout`, чтобы группы располагались вертикально. Сразу после создания "вертикального менеджера" расположения `MainFormLayout` мы создаем дочерний по отношению к нему "горизонтальный" менеджер `ChooseFontLayout`.

Далее мы создаем объекты классов `QComboBox` (раскрывающийся список) и `QSpinBox`, и с помощью метода вызовов `addWidget()` объекта `ChooseFontLayout` передаем их "под управление" этого объекта. Обратите внимание на флаг `Qt::AlignTop`, который мы передаем методу. Этот флаг указывает на привязку визуального элемента к верхнему краю родительского окна. В результате при изменении размеров окна положение элемента относительно верхнего края не изменится. Другие флаги из этой группы таковы: `Qt::AlignLeft`, `Qt::AlignRight`, `Qt::AlignBottom` и `Qt::AlignCenter`. Далее мы выполняем еще одну операцию, связанную с расположением элементов. Наши раскрывающийся список и наборный счетчик будут выстроены горизонтально объектом класса `ChooseFontLayout`. Этот объект позаботится о том, чтобы у элементов, расположением которых он управляет, была одинаковая ширина, и чтобы

при изменении размеров окна эта ширина увеличивалась пропорционально. Но нас такое поведение не устраивает.

Ширина раскрывающегося списка может меняться вместе с шириной окна, но ширину наборного счетчика желательно оставить такой, какой она была сразу после его создания. Дело в том, что при создании визуальных объектов Qt автоматически выбирает минимально допустимую ширину (с учетом длины текста, полей ввода и т. п.), а уж затем "растягивает" элемент соответственно размерам окна. Как зафиксировать размеры визуального элемента? Набор принципов, определяющих конечные размеры элементов, называется "политикой изменения размера" (size policy). Если мы хотим, чтобы ширина объекта `SetSize` оставалась неизменной, нам нужно изменить для этого объекта "политику", заданную по умолчанию, на другую, которая будет указывать, что его ширина должна быть фиксированной. С этой целью мы создаем объект `fixedSP` специального класса `QSizePolicy`. В конструкторе объекта мы указываем, что ширина (первый параметр) и высота (второй параметр), определяемые создаваемой "политикой", должны быть фиксированными. Третий параметр конструктора — это подсказка, с какими именно виджетами создаваемая "политика" будет иметь дело. Разные виджеты при изменении размеров окна должны вести себя немного по-разному. Впрочем, сейчас это несущественно и мы используем значение `QSizePolicy::DefaultType`, которое указывает, что при компоновке виджетов следует выбирать способ, заданный по умолчанию.

С помощью метода `setSizePolicy()` мы указываем объекту `SetSize` новую "политику" (фиксация высоты в данном случае не имеет значения, поскольку она и так фиксирована). Теперь при изменении ширины окна ширина наборного счетчика меняться не будет, а раскрывающийся список `SelectFontBox` займет все пространство слева от наборного счетчика. Далее мы создаем второй объект класса `QHBoxLayout` и "упаковываем" в него группу флажков и текстовую метку. Группа флажков формируется объектом `SetFontGroup` класса `QVBoxLayout`. Этот класс группирует виджеты вертикально, а само добавление виджетов выполняется так же, как и в случае с объектом класса `QHBoxLayout`.

Из визуальных элементов нам осталось добавить только метку, что мы и делаем. Далее следует еще один трюк работы с интерфейсом. С помощью метода `addStretch()` мы добавляем пустое пространство, указав ему фактор растяжения (stretch factor) равным единице. Поскольку у двух объектов `QHBoxLayout` этот фактор равен 0 (значение по умолчанию), при изменении высоты окна вся нижняя часть будет занята пустым пространством, а элементы управления окажутся "прижатыми" к верху. Закомментируйте эту строку и посмотрите, что получится.

Остальную часть программы, связанную с сигналами и слотами, мы сейчас рассматривать не будем. Подробный разговор о сигналах и слотах ожидает нас впереди.

Визуальное программирование

Преимуществами ручного конструирования являются свобода обращения с визуальными компонентами и достижение глубокого понимания механизмов работы Qt.

Недостаток у ручного проектирования только один — высокая трудоемкость. Процесс проектирования графического интерфейса можно упростить и ускорить с помощью среды визуального проектирования Qt Designer. Следует сразу подчеркнуть, что Qt Designer не является полноценной IDE, такой как Borland C++ Builder или MS Visual C++ (а вот Qt Creator уже является). Qt Designer не управляет процессами компиляции или отладки, а только генерирует "заготовки" исходных текстов приложения.

В качестве примера проектирования с помощью Qt Designer мы напишем простую программу, которая позволяет просматривать графические файлы.

> ***Примечание***
>
> Вообще-то интерфейс Qt Designer полностью русифицирован. Но на тот случай, если вам все же придется работать с англоязычной версией, в скобках приводятся английские эквиваленты команд. Это также поможет вам ориентироваться в англоязычных руководствах по Qt Designer.

В главном окне Qt Designer выбираем команду **Файл** (File) | **Новый** (New). В открывшемся окне выбираем тип интерфейса. Для нашей задачи подойдет интерфейс **Dialog without buttons**[1] (напомню, что окно `QDialog` является основой интерфейса простых приложений Qt). В результате будет создана новая форма — объект `Dialog` класса `QDialog`.

Внешний вид Qt Designer в режиме редактирования формы напоминает другие визуальные редакторы, например C++ Builder. Для размещения визуального элемента в окне нужно "перетащить" мышью соответствующую пиктограмму с панели компонентов на форму. На специальной панели можно установить значения свойств объектов, представляющих элементы интерфейса. Подобная панель есть у каждого средства визуальной разработки, но панель редактора свойств Qt Designer организована, на мой взгляд, особенно эргономично. Свойства в ней сгруппированы по классам. Например, класс `QDialog` является потомком `QWidget` и `QObject`. Соответственно первым идет свойство `objectName` — единственное публичное свойство объекта `QObject` (с помощью этого свойства можно изменить имя переменной-объекта класса `QDialog`, чем мы и воспользуемся, заменив его на `MainWindow`). Ниже следуют публичные свойства классов `QWidget` и `QDialog` соответственно. Каждая группа свойств выделена своим цветом.

> ***Примечание***
>
> По-моему, такой принцип расположения свойств позволяет найти нужное свойство быстрее, чем любой другой. Если вам нужны свойства, общие для всех виджетов, ищите их в верхней части панели; если вы хотите отредактировать свойство, специфичное для данного компонента, ищите его в нижней части (для этого, конечно, нужно иметь представление об иерархии классов Qt, но это знание необходимо вам в любом случае).

Помимо режима редактирования виджетов, который открывается в Qt Designer по умолчанию, у программы есть еще три режима: режим редактирования сигналов и

---

[1] В программе этот пункт выглядит именно так. Его можно перевести как "Диалоговое окно без кнопок".

слотов (signals/slots editor), режим редактирования партнеров (buddies editor) и режим изменения порядка переключений (tab order editor).

Окно создаваемой нами программы (рис. 1.9) содержит два визуальных элемента: кнопку **Открыть файл...** (объект `fileOpenButton` класса `QPushButton`), с помощью которой загружается файл, и объект `label` класса `QLabel` для вывода изображения. Класс `QLabel` чаще применяется для вывода текста, но с его помощью можно отобразить и растровую графику.

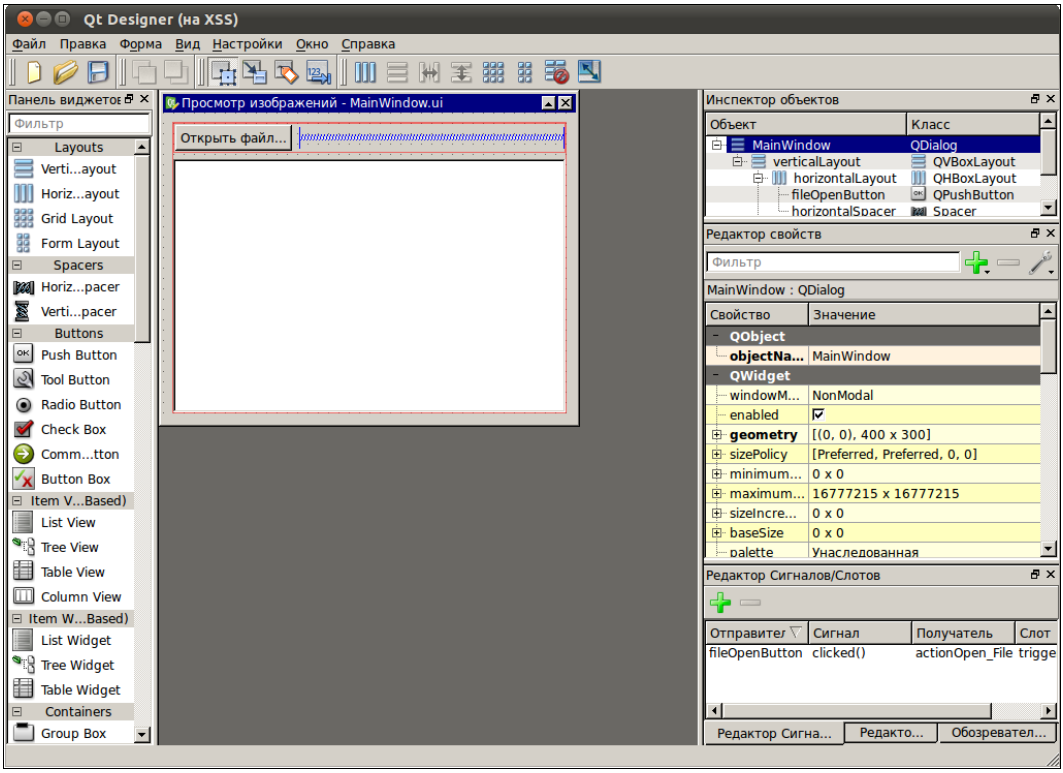


Рис. 1.9. Окно программы в режиме визуального редактирования

Примечание

Вы, возможно, хотите спросить, почему для вывода графики мы не использовали класс `QGraphicsView`. Этот класс действительно специально предназначен для вывода графики, и он позволяет сделать гораздо больше, чем нам нужно сейчас. Но за ним стоит целая система графического вывода, о которой мы поговорим отдельно.

Кнопка и объект вывода изображения объединены одним элементом `Layout` (менеджерами компоновки, с которыми мы уже познакомились). В режиме редактирования формы область действия менеджера компоновки обводится красной каймой. Для того чтобы связать одним менеджером компоновки несколько элементов интерфейса, необходимо выделить эти элементы (щелкая по ним мышью и удерживая нажатой клавишу <Shift>), а затем в контекстном меню выбрать одну из команд

дочернего меню **Компоновка** (Lay out). Чтобы скомпоновать кнопку и элемент вывода графики, мы щелкаем правой кнопкой мыши по свободному месту в окне формы и в контекстном меню выбираем команду **Скомпоновать по вертикали** (Lay Out Vertically). Будучи применена к форме, эта команда связывает все расположенные на ней группы элементов единым менеджером компоновки. Если теперь вы попробуете изменить размер окна, то увидите, что полученный нами результат далек от совершенства. Ширина кнопки меняется вместе с шириной окна, а нам бы хотелось, чтобы кнопка была фиксирована. Самый простой способ добиться этого — добавить справа от кнопки специальный элемент-"распорку" (`Spacer`, в палитре компонентов Qt Disigner распорки отображаются пиктограммами со значком пружины). Мы используем горизонтальную распорку, которая "прижмет" кнопку к левому краю окна. Но для этого нам придется разрушить созданную нами компоновку. В том же контекстном меню выбираем команду **Удалить компоновщик** (Break Layout). Теперь добавляем в горизонтальную форму распорку и связываем кнопку и распорку горизонтальным компоновщиком: выделяем мышью оба элемента в окне редактора форм и в контекстном меню выбираем команду **Скомпоновать по горизонтали** (Lay Out Horizontally). Теперь у нас есть два нескомпонованных элемента: группа из горизонтально скомпонованных кнопки и распорки и компонент `GraphicsView`. Мы компонуем их вертикально, как было описано выше. Однако результат опять оставляет желать лучшего. Дело в том, что по умолчанию менеджер вертикальной компоновки (как и любой другой менеджер компоновки) разделит доступное ему пространство поровну между компонентами. То есть кнопке и объекту вывода графики будет выделена одинаковая площадь пространства окна. Мы же хотим, чтобы кнопка занимала минимальное пространство, которое ей необходимо, а все остальное пространство занимал объект вывода. Для того чтобы исправить распределение пространства между виджетами, выделите объект `label` и в разделе свойств `QWidget` редактора свойств раскройте группу `SizePolicy`. Эта группа позволяет нам устанавливать политику размеров (которая нас в данном случае не интересует) и указать растяжение (уже знакомый нам stretch factor). Мы задаем значение фактора растяжения по вертикали равным 1. Вот теперь кнопка будет прижата не только к левому краю окна, но и к верхнему, а все пространство ниже кнопки будет занято объектом вывода изображения.

Вы можете заметить, что манипуляции с компоновками очень похожи на действия в примере с программой просмотра шрифтов, только в режиме визуального редактирования работать гораздо удобнее. Между прочим, если вы объединили менеджером компоновки несколько виджетов или скомпонованных групп, а потом обнаружили, что случайно перепутали местами некоторые виджеты или забыли что-то добавить, вам во многих случаях необязательно разрушать уже созданную компоновку и начинать все заново. Внутри рамок, определяющих границы компоновки, виджеты можно перетаскивать мышью, меняя их местами. Можно также перетащить внутрь рамки новый виджет с панели визуальных компонентов. Он будет вставлен туда, куда вы его перетащите, при этом другим виджетам в той же компоновке придется "подвинуться". Как именно они "подвинутся", зависит от выбранных для них политик изменения размера и фактора растяжения. Самым "бесправ-

ным" компонентом является распорка. Она всегда уступит новому виджету столько места, сколько тот потребует.

Наконец, если вы обнаружили, что вместо требуемого компонента `QPushButton` добавили в окно `QRadioButton`, вы можете исправить эту ошибку с минимальными затратами и потерями. Щелкните по виджету правой кнопкой мыши и выберите пункт контекстного меню **Преобразовать в** (Morph into). При этом откроется дочернее меню, в котором можно выбрать различные варианты кнопок для замены. Команду **Преобразовать в** не следует путать с командой **Преобразовать в...** (Promote to...) того же контекстного меню (и спасибо русским переводчикам Qt Designer за такое "разнообразие"). Эта команда выполняет другие, не менее полезные действия.

> ***Примечание***
>
> У меня создалось впечатление, что мои статьи по программированию для Qt 3.x, опубликованные некогда в одном журнале, посвященном Linux, повлияли на выбор терминологии переводчиками интерфейса Qt Designer на русский язык. Если кто-то из команды локализации Qt Designer вдруг прочтет и эту книгу, то вот мои предложения: в данном контексте **Morph into** лучше всего переводить как **Заменить на**. А вот перевод **Promote to...** можно оставить как **Преобразовать в...**.

Вероятно, вы помните, что в примере программы просмотра шрифтов мы немало внимания уделили политике изменения размера визуального компонента. В этом примере нам стоит беспокоиться о политике изменения размера кнопки и метки. Метку мы уже настроили. Что касается кнопки, необходимо учесть следующее. Для вывода надписи на кнопке мы используем шрифт определенного семейства и высоты, но в другой системе такого шрифта может просто не быть, а ширина шрифта, который система подберет в качестве замены, может отличаться от ширины нашего шрифта. Соответственно и ширина надписи изменится. Нам желательно, чтобы ширина кнопки учитывала ширину надписи (тем более это желательно, если мы планируем переносить интерфейс программы на другие платформы или переводить его на другие языки). Мы можем управлять политикой изменения размера с помощью редактора свойств компонента Qt Designer. В заготовке главного окна выделите мышью кнопку, а в окне редактора свойств, в разделе свойств `QWidget`, раскройте группу `SizePolicy`. Установите политику изменения размеров по горизонтали (здесь подойдет `Minimum` — политика, заставляющая кнопку принимать минимальную ширину с учетом ширины надписи на ней) и по вертикали (здесь подойдет `Fixed`). Фактор растяжения мы оставляем равным нулю, т. к. нашей кнопке не нужно бороться за пространство окна.

Итак, мы описали внешний вид главного окна нашей программы. Теперь неплохо добавить немного кода. В предыдущих версиях Qt это можно было сделать, не выходя из Qt Designer. Но теперь так делать нельзя и, на мой взгляд, это правильно.

Впрочем, в некоторых случаях можно обойтись и вовсе без кода. В режиме редактирования сигналов и слотов Qt Designer вы можете визуально связать сигнал одного виджета со слотом другого. Подробно эта операция описана в документации по Qt Designer. На мой взгляд, от такого визуального связывания мало пользы. Как

правило, нам нужно связать сигнал визуального элемента со слотом невизуального, или наоборот. А этого визуальный редактор сделать как раз и не позволяет. Единственный случай, в котором это может быть полезно, — связывание сигнала `clicked()` кнопки **Отмена** со слотом `reject()` диалогового окна (в этом случае щелчок по кнопке будет приводить к закрытию окна). Однако, по иронии судьбы, именно в этом случае программисты, использующие Qt 4.7.0, столкнулись с досадной ошибкой в библиотеке Qt. Автоматическое связывание сигнала виджета нижнего уровня (кнопки) со слотом виджета самого верхнего уровня (окна) могло внезапно перестать работать после очередной повторной сборки приложения, так что именно эти сигнал и слот приходилось связывать "вручную", с помощью метода `connect()`.

Гораздо интереснее другая возможность, связанная с редактором действий (action editor). Концепция единых действий должна быть хорошо знакома тем, кто программировал в Borland Delphi и C++ Builder. В приложениях с графическим интерфейсом, как правило, одну и ту же команду можно вызвать несколькими способами (из основного меню, с панели быстрого доступа, из контекстного меню). При этом обычно во всех формах вызова команды используются одни и те же элементы — пиктограмма, всплывающая подсказка и, конечно, функция-обработчик команды. До появления концепции единых действий программистам приходилось выполнять большой объем однообразной работы: для каждого элемента интерфейса нужно было явным образом указывать пиктограмму, подсказку и обработчик команды. Далее сложности нарастали: если в определенном состоянии программы команду требовалось заблокировать, необходимо было отслеживать состояния всех элементов интерфейса, связанных с этой командой. Как вы уже догадались, концепция единых действий существенно упрощает работу программиста (и является частью более важного принципа, о котором мы будем много говорить в этой книге и с которым подробнее познакомимся далее в этой главе). В рамках этой концепции все общие элементы команды объединяются в один объект (в случае Qt — объект класса `QAction`). Создав объект `QAction` для определенной команды, мы указываем объекту, в каких элементах интерфейса должен присутствовать вызов этой команды. Об остальном объект `QAction` позаботится сам. Если мы свяжем этот объект с каким-либо меню, будет создана строка меню, если с панелью быстрого доступа — будет создана кнопка и т. п. Если в какой-то момент команда должна быть заблокирована, нам достаточно вызвать метод `setEnabled(FALSE)` для объекта `QAction` и все визуальные представления команды будут заблокированы. Аналогично выполняется и разблокировка команды. Можно сказать, что объект `QAction` — это центр управления всеми элементами интерфейса, связанными с соответствующей командой.

Примечание

Из сказанного выше должно быть ясно, что в полную силу объект `QAction` может развернуться в приложениях со сложным интерфейсом, со строкой меню и панелями быстрого доступа, но и в нашем простеньком приложении мы используем его "не ради красного словца". Далее мы увидим, почему это полезно.

Для того чтобы создать новое действие, в редакторе действий щелкните кнопку **Новое**. Откроется диалоговое окно, в котором можно задать имя нового действия, текст всплывающей подсказки, пиктограмму, тип действия (триггерное или обычное), сочетание горячих клавиш для вызова действия. Большая часть этих параметров нам сейчас не нужна. Достаточно выбрать имя нового действия (мы назовем его `actionOpen_File`).

Теперь переходим на вкладку **Редактор сигналов и слотов** редактора действий. В поле **Отправитель** выберем кнопку `fileOpenButton`. В поле **Сигнал** — значение `clicked()`, в поле **Получатель** — объект `actionOpen_File`, а в поле **Слот** — значение `trigger()`. Вот, собственно и все. Теперь мы связали кнопку и действие.

Нельзя не отметить еще одну полезную особенность Qt Designer — возможность создания классов-потомков классов виджетов непосредственно в процессе визуального программирования. Все, кому приходилось заниматься серьезной разработкой в Delphi или Borland C++ Builder, наверняка были вынуждены создавать собственные визуальные элементы управления на основе стандартных. Это естественно, поскольку ни одна библиотека визуальных компонентов не может отвечать всем возможным потребностям программиста. Однако в Delphi и Borland C++ Builder процесс создания нового компонента довольно трудоемок и включает в себя установку этого компонента в среде разработки (а если потом вы переносите разработку вашей программы в другой экземпляр среды, всю процедуру установки придется выполнять и там). Qt Designer тоже позволяет вам создавать собственные компоненты и добавлять их в палитру, но сейчас мы рассмотрим более простой вариант. Допустим, стандартная кнопка `QPushButton` нас чем-то не устраивает, и мы хотим создать собственный элемент управления на основе `QPushButton`. В контекстном меню выбираем команду **Преобразовать в...** (не путайте ее с **Преобразовать в**!). В результате откроется окно (рис. 1.10), в котором можно задать имя нового класса-потомка `QPushButton` и имя заголовочного файла, в котором будет храниться объявление этого класса.

Что мы получаем в результате? У нас появляется класс, в котором мы можем переопределить поведение класса `QPushButton`, используя, в том числе, методы `QPushButton`, объявленные как `protected`. При этом объект нового класса остается частью визуального описания интерфейса нашей программы. Фактически мы создали новый визуальный компонент, который, однако, является частью только нашей программы, а не среды Qt Designer и не требует специальной установки для сборки программы. Все это хорошо, можете сказать вы, но очень часто мы создаем специальные компоненты только для того, чтобы не повторять один и тот же код многократно. Можно ли использовать повторно визуальный компонент, созданный описанным выше способом? В рамках данной программы — безусловно. Если вы внимательно посмотрите на диалоговое окно создания класса визуального компонента, то увидите, что оно сильно напоминает репозиторий. Это и есть локальный репозиторий визуальных компонентов. Для того чтобы использовать этот же компонент в другом месте, просто добавьте новый компонент `QPushButton`, выберите команду **Преобразовать в...**, и вы одним щелчком сможете превратить `QPushButton` в созданную ранее кнопку.

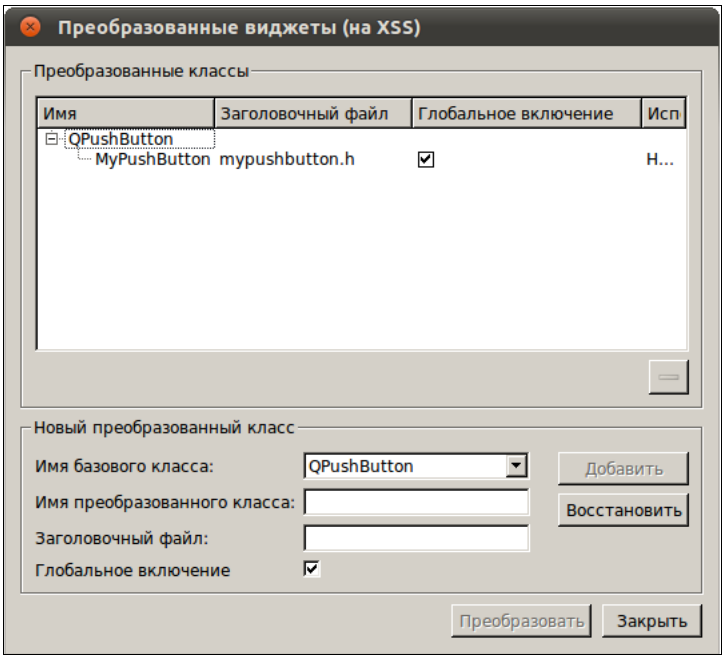


Рис. 1.10. Создание дочернего класса в визуальном редакторе

Как же все-таки связать окно с исходным кодом на C++? В этом нам поможет утилита под названием uic, что является сокращением от user interface compiler (компилятор пользовательского интерфейса). Если вы напишете команду:

```
uic MainForm.ui
```

то экран консоли заполнится исходным текстом C++. По умолчанию утилита uic выводит результаты своей работы (преобразование описания интерфейса в текст C++) в стандартный поток вывода. Это не должно вас удивлять, ведь зачастую uic вызывается другими программами для автоматической генерации кода C++. Этим программам удобнее читать результат работы uic из стандартного потока. Но нам нужен файл, поэтому мы напишем так:

```
uic MainForm.ui > ui_MainForm.h
```

Давайте посмотрим, что приготовил для нас uic (листинг 1.7).

Листинг 1.7. Код, сгенерированный утилитой uic

```
#ifndef UI_MAINWINDOW_H
#define UI_MAINWINDOW_H

#include <QtCore/QVariant>
#include <QtGui/QAction>
#include <QtGui/QApplication>
#include <QtGui/QButtonGroup>
#include <QtGui/QDialog>
```

```
#include <QtGui/QHBoxLayout>
#include <QtGui/QHeaderView>
#include <QtGui/QLabel>
#include <QtGui/QPushButton>
#include <QtGui/QSpacerItem>
#include <QtGui/QVBoxLayout>

QT_BEGIN_NAMESPACE

class Ui_MainWindow
{
public:
    QAction *actionOpen_File;
    QVBoxLayout *verticalLayout_2;
    QVBoxLayout *verticalLayout;
    QHBoxLayout *horizontalLayout;
    QPushButton *fileOpenButton;
    QSpacerItem *horizontalSpacer;
    QLabel *label;

    void setupUi(QDialog *MainWindow)
    {
        if (MainWindow->objectName().isEmpty())
            MainWindow->setObjectName(QString::fromUtf8("MainWindow"));
        MainWindow->resize(400, 300);
        actionOpen_File = new QAction(MainWindow);
        actionOpen_File->setObjectName(
QString::fromUtf8("actionOpen_File"));
        verticalLayout_2 = new QVBoxLayout(MainWindow);
        verticalLayout_2->setObjectName(
QString::fromUtf8("verticalLayout_2"));
        verticalLayout = new QVBoxLayout();
        verticalLayout->setObjectName(
QString::fromUtf8("verticalLayout"));
        horizontalLayout = new QHBoxLayout();
        horizontalLayout->setObjectName(
QString::fromUtf8("horizontalLayout"));
        fileOpenButton = new QPushButton(MainWindow);
        fileOpenButton->setObjectName(
QString::fromUtf8("fileOpenButton"));

        horizontalLayout->addWidget(fileOpenButton);

        horizontalSpacer = new QSpacerItem(40, 20,
 QSizePolicy::Expanding, QSizePolicy::Minimum);

        horizontalLayout->addItem(horizontalSpacer);
```

```
        verticalLayout->addLayout(horizontalLayout);

        label = new QLabel(MainWindow);
        label->setObjectName(QString::fromUtf8("label"));
        QSizePolicy sizePolicy(QSizePolicy::Preferred,
 QSizePolicy::Preferred);
        sizePolicy.setHorizontalStretch(0);
        sizePolicy.setVerticalStretch(1);
        sizePolicy.setHeightForWidth(
label->sizePolicy().hasHeightForWidth());
        label->setSizePolicy(sizePolicy);

        verticalLayout->addWidget(label);

        verticalLayout_2->addLayout(verticalLayout);

        retranslateUi(MainWindow);
        QObject::connect(fileOpenButton, SIGNAL(clicked()),
 actionOpen_File, SLOT(trigger()));

        QMetaObject::connectSlotsByName(MainWindow);
    } // setupUi

    void retranslateUi(QDialog *MainWindow)
    {
...
 } // retranslateUi

};

namespace Ui {
    class MainWindow: public Ui_MainWindow {};
} // namespace Ui

QT_END_NAMESPACE

#endif // UI_MAINWINDOW_H
```

В первую очередь здесь определен класс `Ui_MainWindow`, у которого есть метод `setupUi()`. Этот метод создает все визуальные компоненты нашей программы, а также все заданные нами менеджеры компоновки. Кроме того, он создает объект `actionOpen_File` и выполняет связывание сигнала и слота.

Метод `retranslateUi()` используется внутри программы для перевода интерфейса на другие языки (при наличии, естественно, соответствующих ресурсов перевода).

Далее объявляется пустой класс `MainWindow`, который является потомком `Ui_MainWindow`. Класс `MainWindow` объявляется в специальном пространстве имен

`Ui`. Это сделано для того, чтобы мы могли использовать удобное имя `MainWindow` в своей программе, не вступая в конфликт с автоматически сгенерированным классом. Обратите внимание на то, как uic работает с русскими буквами, заданными в кодировке UTF-8. Проблема перекодировок решена радикально: вместо символов используются их численные коды. Хотя созданный с помощью uic файл является заголовочным, он не требует файла с расширением cpp. Мы сразу можем использовать его в своей программе.

Использовать сгенерированный код описания интерфейса можно двумя способами. Но для начала напишем свой класс `MainWindow`, в котором реализуем логику работы приложения (листинг 1.8).

Листинг 1.8. Использование автоматически сгенерированного кода в программе

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include "ui_MainWindow.h"
#include <QDialog>

class MainWindow : public QDialog
{
    Q_OBJECT
 private:

    Ui::MainWindow * ui;

public:
    explicit MainWindow(QWidget *parent = 0);

signals:

private slots:
    void openFile();

};

#endif // MAINWINDOW_H


#include "mainwindow.h"
#include <QMessageBox>
#include <QFileDialog>
#include <QPixmap>

MainWindow::MainWindow(QWidget *parent) :
    QDialog(parent)
```

```
{
    ui = new Ui::MainWindow();
    ui->setupUi(this);
    connect(ui->actionOpen_File, SIGNAL(triggered()), this,
        SLOT(openFile()));
}

void MainWindow::openFile()
{
    QPixmap pm;
    QString fn = QFileDialog::getOpenFileName(this,
        trUtf8("Открыть изображение"));
    if (fn != "") {
        if (!pm.load(fn))
            QMessageBox::critical(this, trUtf8("Ошибка"),trUtf8(
            "Невозможно загрузить изображение из выбранного файла."));
        else {
            ui->label->setPixmap(pm);
            setWindowTitle(trUtf8("Просмотр изображений — ") + fn);
        }
    }
}
```

Исходные тексты этой программы вы найдете в папке Ch1/designrdemo на сайте **http://symmetrica.net/qt47book/**.

В конструкторе `MainWindow` мы создаем объект класса `Ui::MainWindow`, сгенерированного uic, и вызываем метод `setupUi()` этого класса, передавая ему указатель `this` в качестве параметра. Обратите внимание, что класс `MainWindow` является потомком класса `QDialog`. Мы могли бы использовать и другой базовый класс, но тогда при вызове `setupUi()` нам пришлось бы выполнять приведение типов, поскольку параметр этого метода имеет тип "указатель на `QDialog`". Теперь все элементы графического интерфейса нашего окна доступны нам через переменную `ui`.

Нам остается создать слот, который бы обрабатывал сигнал `triggered()` объекта `actionOpen_File` (слот `openFile`). В этом слоте мы сначала получаем имя открываемого файла с помощью статического метода `getOpenFileName()` класса `QFileDialog` (этот метод создает модальное диалоговое окно для выбора имени файла), а затем пробуем загрузить выбранный файл в объект `QPixmap`. Мы, естественно, делаем это только в том случае, если метод `getOpenFileName()` не вернул пустую строку (что происходит при отказе пользователя от выбора файла). Если объект `QPixmap` может загрузить выбранный файл (в этом случае метод `load()` возвращает значение `true`), то мы присваиваем загруженное изображение объекту `ui->label` с помощью метода `setPixmap()`. Затем мы изменяем заголовок окна так, чтобы он отображал имя загруженного файла (рис. 1.11).

Так чего же мы добились всеми этими манипуляциями? Очень многого. Мы на практике реализовали один из важнейших принципов профессионального про-

граммирования — принцип формального разделения различных компонентов программы. Этот принцип выступает во многих формах: разделение движка и пользовательского интерфейса, настраиваемой части программы и ненастраиваемой, кросс-платформенной части и части, зависящей от конкретной платформы, и т. д.

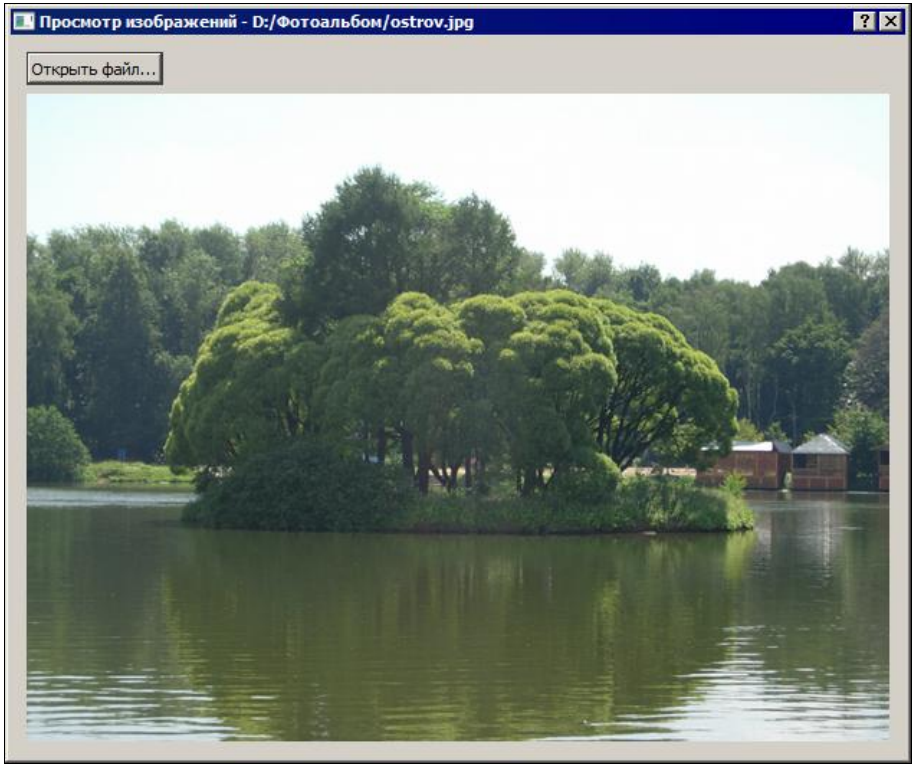


Рис. 1.11. Программа просмотра изображений, вариант 1

Как вы уже поняли, сейчас мы реализовали принцип разделения движка программы (класс `MainWindow`) и пользовательского интерфейса (класс `Ui::MainWindow`). Класс `MainWindow` взаимодействует с пользовательским интерфейсом, используя очень простой набор правил: класс получает сигнал от события `actionOpen_File`, выводит изображение в объект `label`. Все, что требуется "знать" классу `MainWindow` об интерфейсе программы, — это то, что он включает событие `actionOpen_File` и объект `label`. В остальном интерфейс программы можно изменить как угодно (что и будет продемонстрировано далее), вообще не касаясь текста класса `MainWindow` (точнее, почти не касаясь). Сравните этот подход с традиционным программированием в Delphi или Borland C++ Builder, где класс формы представляет собой мешанину из кода, реализующего логику работы программы, и кода, управляющего пользовательским интерфейсом. О принципе разделения движка и интерфейса (как и о других формах разделения) в этой книге будет сказано еще немало. Сейчас просто отметим тот факт, что у нас в программе есть два раздельных модуля, которые взаимодействуют между собой по четко зафиксированным правилам.

Примечание

Разумеется, и в Delphi можно реализовать разделение движка и интерфейса, точно так же, как в нашей программе мы могли бы соединить слот `openFile` напрямую с сигналом `clicked()` кнопки (как будет показано далее, это было бы ошибкой, из-за которой нам пришлось бы выполнять лишнюю работу впоследствии). Ни одна среда разработки не может сама научить вас грамотному программированию или запретить ошибаться. Но некоторые средства сильнее подталкивают вас на правильный путь, и Qt Designer является одним из них.

Теперь вернемся к созданию собственных виджетов (с помощью команды **Преобразовать в...**). Тут не все так просто и легко, как вам могло показаться. Код для нового виджета вам придется писать вручную, автоматический генератор кода uic вам тут ничем не поможет. Зато, если в своем новом виджете вы объявите новый слот, а затем объявите этот же слот в редакторе сигналов и слотов Qt Designer, в том, который вызывается по команде контекстного меню **Изменить сигналы и слоты** (Change signals/slots), то вы сможете воспользоваться методом визуального программирования для связывания какого-нибудь сигнала со своим слотом. Это тот самый случай, когда от визуального связывания сигналов и слотов может быть определенная польза. Правда, учитывая объем невизуальной работы, которую придется при этом проделать, выигрыш не так уж и велик.

В рассмотренном примере мы получали доступ к графическому интерфейсу программы из класса `MainWindow` через переменную `ui`. Этот способ является более предпочтительным, т. к. лучше реализует разделение движка и интерфейса, и он применяется, например, в интегрированной среде Qt Creator. Но у нас есть и другая возможность: используя множественное наследование C++, мы можем определить класс `MainWindow` как потомка `QDialog` и `Ui::MainWubdow`:

```
class MainWindow : public QDialog, private Ui::MainWindow {
...
};
```

В этом случае все объекты пользовательского интерфейса станут доступны в самом классе `MainWindow` (напомню, что в классе `Ui::MainWindow` все они объявлены в разделе `public`). Иначе говоря, нам не придется использовать префикс `ui`, что, на мой взгляд, не является таким уж существенным облегчением.

Другие возможности Qt Designer

Продемонстрируем возможности разделения движка и интерфейса на практике и радикально сменим внешний вид приложения. Для этого в Qt Designer создадим новый проект главного окна, но теперь за основу возьмем не `QDialog`, а `QMainWindow`. Класс `QMainWindow` отличается от класса `QDialog` возможностью добавить меню и панель быстрого доступа "совсем, как у больших". Я предоставляю вам самостоятельно разобраться с тем, как добавить меню и панель быстрого доступа (на самом деле, конечно, вы обнаружите, что все уже сделано в директории designerdemo2). Мы же подробно рассмотрим еще одни важный инструмент Qt Designer — редактор ресурсов.

Программы с графическим интерфейсом нуждаются в ресурсах, например, в пиктограммах для кнопок быстрого доступа, аудиофайлах для звуковых эффектов и тому подобном. Все это можно хранить в отдельных файлах и загружать в программу по мере надобности. Другой вариант — загрузить пиктограмму из файла непосредственно в тот класс, который будет с ней работать (не уверен, что такое возможно для аудиоэффектов). Первый вариант хорошо соответствует принципу разделения, но он неудобен, т. к. вместе с программой придется распространять множество файлов, которые должны храниться в таком месте, откуда программа сможет их загрузить (требование кросс-платформенности приложения еще больше усложняет этот вопрос). Второй вариант делает ресурсы неотъемлемой частью приложения, но... Представьте себе, что вам захотелось радикально сменить стиль приложения и использовать другие пиктограммы. Если пиктограммы разбросаны по объектам пользовательского интерфейса, процесс замены станет весьма трудоемким. Ресурсы Qt решают эту проблему наилучшим образом. Все файлы ресурсов объединяются в один модуль, который может быть частью исполнимого файла программы, а может распространяться в виде отдельного файла (тогда, конечно, опять возникает вопрос, где хранить этот файл, но, по крайней мере, теперь речь идет только об одном файле, а не о множестве). В любом случае, с точки зрения приложения, сами ресурсы, объединенные в отдельный модуль, располагаются в некоем виртуальном файловом пространстве, не зависящем от файловой системы той ОС, на которой работает приложение. Если нам захочется сделать косметический ремонт приложения, нам будет гораздо проще заменить файлы ресурсов, собранные в одном модуле, причем, если этот модуль представляет собой отдельный файл, мы даже можем сменить ресурсы без повторной сборки самого приложения. Таким образом, использование модулей ресурсов решает и проблему разделения, и проблему переноса приложений между платформами.

Так же как в случае с переводом приложения на другие языки нам нужно сначала создать исходные тексты переводов, при работе с ресурсами нам потребуется исходный текст модуля ресурсов. Он создается в редакторе ресурсов Qt Designer, а хранится в файле с расширением qrc. Такой файл представляет собой полностью текстовый документ XML. Говоря "полностью текстовый", я подразумеваю, что он не содержит двоичных данных. Вместо этого он содержит ссылки на двоичные файлы ресурсов, которые и будут скомпонованы в модуль ресурсов. Для начала нам понадобится директория для хранения файлов ресурсов. В нашем примере это директория designerdemo2/Images. Она содержит один-единственный файл — пиктограмму для действия "открыть файл". Теперь мы можем создать файл с расширением qrc. Этот файл обычно хранится в той же директории, что и исходные тексты программы (при условии, что сами исходные тексты занимают только одну директорию).

В нижнем правом углу Qt Designer перейдите на вкладку **Обозреватель ресурсов** (Resource Browser) и щелкните кнопку **Изменить ресурсы** (Edit Resources). Откроется окно редактора файлов ресурсов (рис. 1.12). В этом окне щелкните кнопку **Новый файл ресурсов** (New Resource File) и укажите имя файла ресурсов (обычно, оно такое же, как и у всего проекта, за исключением, разумеется, расширения qrc). В правой половине окна редактора ресурсов вы можете добавлять файлы ресурсов.

При этом можно использовать префиксы, которые играют роль директорий виртуальной файловой системы ресурсов. Имена этих префиксов могут совпадать с именами директорий, в которых расположены файлы ресурсов (как в нашем примере), но это не обязательно. После сохранения файла ресурсов у нас появится файл с расширением qrc (листинг 1.9).

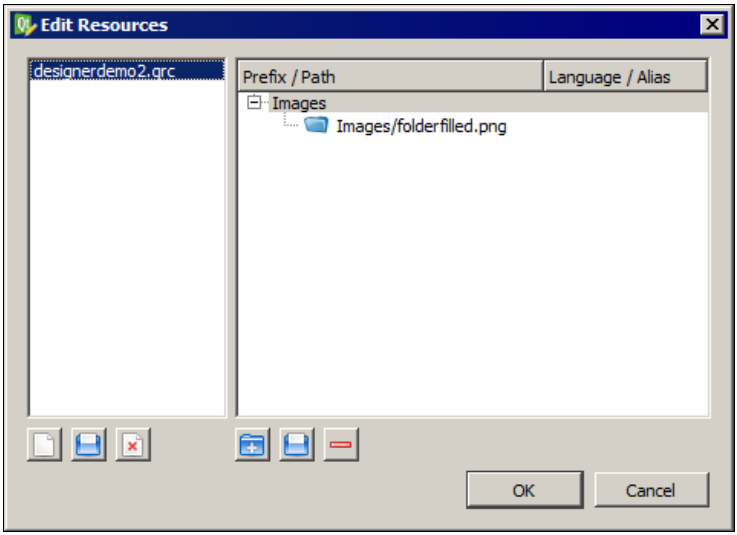


Рис. 1.12. Редактор файлов ресурсов

Листинг 1.9. Файл с расширением qrc

```
<RCC>
  <qresource prefix="Images">
    <file>Images/folderfilled.png</file>
  </qresource>
</RCC>
```

Как видите, ничего сложного. Такой файл можно создать в обычном текстовом редакторе.

В файл designerdemo2.pro нужно добавить строчку:

```
RESOURCES += designerdemo2.qrc
```

Зато теперь нам будет легко добавлять пиктограммы в наши объекты-действия (рис. 1.13). Такие компоненты как меню и панель быстрого доступа работают с действиями даже проще, чем обычные кнопки. Хотите добавить новый пункт в меню или на панель быстрого доступа — просто перетащите туда соответствующее действие из окна редактора действий. Не забудьте добавить в форму главного окна объект `QLabel` и настроить его с помощью менеджера компоновки таким образом, чтобы он занимал все свободное пространство окна.

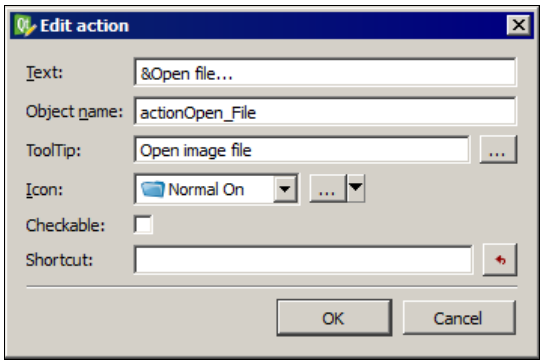


Рис. 1.13. Пиктограмма в редакторе действий

Теперь перейдем к классу `MainWindow`, созданному для примера demodesigner (файлы уже скопированы в папку demodesigner2). Поскольку теперь класс пользовательского интерфейса `Ui::MainWindow` происходит от класса `QMainWindow`, я заменил предка класса `MainWindow` с `QDialog` на `QMainWindow`. Это сделано для удобства вызова метода `setupUi()`. Проблему можно было бы решить и иначе, но мне такой способ представляется самым простым. В остальном код класса `MainWindow` никак не меняется. Обратите особое внимание на этот факт. Теперь программу можно собрать и получить совсем другой пользовательский интерфейс (рис. 1.14) с тем же движком, реализующим работу программы. Я нигде не упоминаю файл main.cpp, который содержит функцию `main()` нашей программы. Надеюсь, вы помните про него сами. Излишне говорить, что в этом файле тоже ничего не меняется.

В нашем примере модуль ресурсов стал частью исполнимого файла программы. Как уже было отмечено, есть и другой путь — скомпилировать модуль ресурсов в отдельный файл и загружать его в начале работы программы (в функции `main()`), так же, как мы загружаем скомпилированные файлы переводов. Для компиляции файла ресурсов следует использовать утилиту rcc, входящую в состав Qt. Результатом выполнения утилиты rcc станет файл с расширением rcc. Это и есть скомпилированный модуль ресурсов. Переменную `RESOURCES` из файла с расширением pro теперь можно убрать, а в файл main.cpp потребуется кое-что добавить.

Загрузкой ресурсов из файла rcc управляет класс `QResource`. Для регистрации файла внешних ресурсов в приложении воспользуемся статическим методом `QResource::registerResource()`. Если позднее вам понадобится выгрузить ресурсы (обычно это происходит автоматически вместе с завершением работы программы), вызовите статический метод `QResource::unregisterResource()`. Оба метода возвращают значение `true`, если соответствующая операция выполнена успешно, и `false` в противном случае.

> ***Примечание***
>
> В том действительно противном, мерзком, случае, когда ваше приложение не может найти директорию, в которой расположен файл *.rcc.

Я надеюсь, вы разобрались с тем, как загружать конкретные ресурсы в Qt Designer. А что делать, если вам потребуется загрузить файл ресурса, например, графический

Рис. 1.14. Программа просмотра изображений, вариант 2

файл, во время работы программы? Все очень просто. Как уже было упомянуто, благодаря системе ресурсов Qt, программа видит модули ресурсов как контейнер с виртуальной файловой системой внутри. Так что если вам потребуется загрузить файл folderfilled.png из модуля ресурсов в класс `QPixmap`, в программе это может выглядеть примерно так:

```
QPixmap pm;
pm.load(":/Images/folderfilled.png");
```

Напомню, что эта ссылка, начинающаяся с `":/"`, не зависит ни от платформы, ни от того, является ли модуль ресурсов частью файла приложения или отдельным файлом. Виртуальная файловая система ресурсов приложения Qt настолько соответствует настоящей, что в ней можно создавать дочерние директории и использовать

для работы с ней классы `QDir`, `QFileInfo` и другие классы Qt, предназначенные для манипулирования файлами и директориями.

Редактирование партнеров

Представьте себе, что вы добавляете в форму виджет `QLineEdit` и виджет `QLabel` с поясняющей надписью. В поясняющую надпись можно вставить символ `&`, который подчеркнет следующий за ним символ, так что у вас появится клавиша быстрого доступа. Но клавиша быстрого доступа для самой метки не имеет особого смысла, поскольку метка не является компонентом ввода. Режим редактирования партнеров позволяет связать метку с компонентом ввода, например с виджетом `QLineEdit`. В этом случае при нажатии клавиши быстрого доступа, определенной для метки, фокус ввода получит компонент ввода, связанный с этой меткой. Для редактирования партнеров в соответствующем режиме Qt Designer (в английском варианте он называется buddy editor) необходимо связать виджеты-партнеры, что делается примерно так же, как визуальное связывание сигналов и слотов.

Редактирование порядка перехода между виджетами

Клавиша табуляции — стандартное средство перехода между элементами ввода на одной и той же форме. Нажимая эту клавишу, пользователь вправе ожидать, что

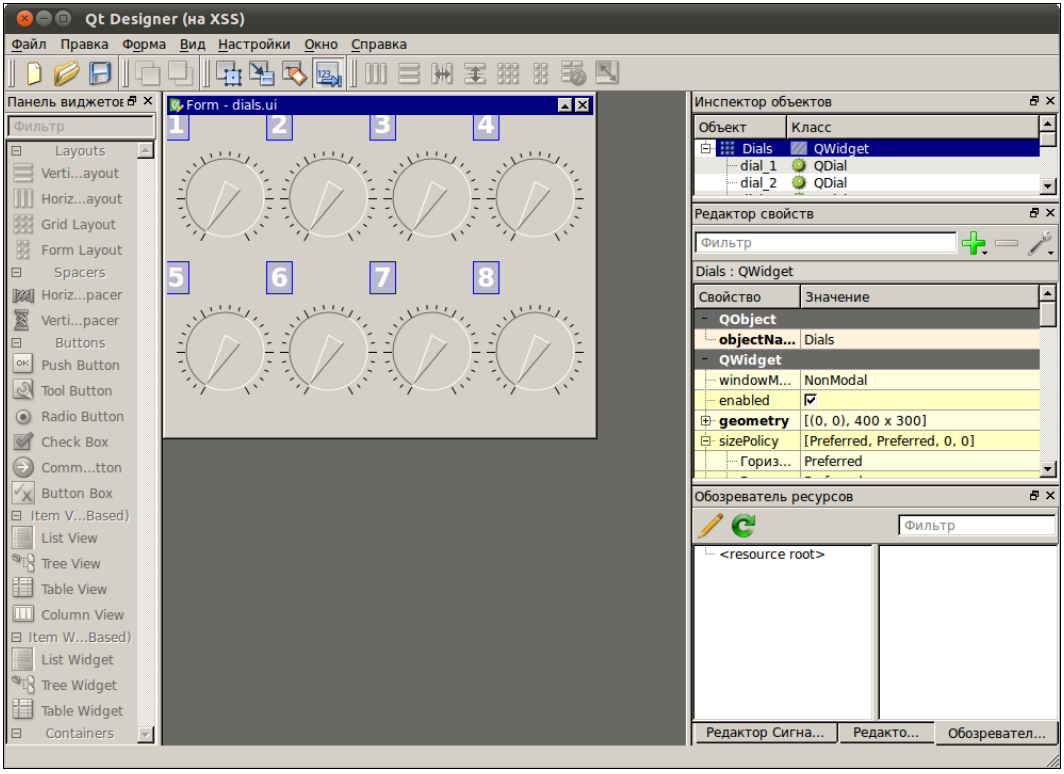


Рис. 1.15. Режим редактирования порядка обхода виджетов

переход от одного виджета к другому будет выполняться не случайным образом, а в каком-то разумном порядке, например, слева направо и сверху вниз, или в соответствии со смысловой группировкой виджетов. Режим редактирования порядка перехода между виджетами позволяет вам исправить возможные нарушения в этом порядке. Делается это очень просто и наглядно (рис. 1.15). Каждому виджету присваивается номер, соответствующий его порядковому номеру при обходе с помощью клавиши <Tab>. Щелкая указателем мыши по прямоугольникам с цифрами, вы можете изменить этот порядок.

Qt Designer и таблицы стилей

Иногда в программу неплохо добавить немного украшений. Предположим, вы хотите, чтобы все строки ввода в вашей форме имели скругленные края, чтобы текст был высотой 12 пунктов и полужирного начертания, и чтобы при получении фокуса ввода строка ввода окрашивалась в желтый цвет (а при потере фокуса возвращалась к исходному). Как ни странно, вам не придется для этого писать свой собственный компонент. Достаточно отредактировать таблицу стилей формы. Щелкните правой кнопкой мыши где-нибудь в свободном месте формы и в контекстном меню выберите пункт **Изменить styleSheet...** (Change styleSheet...)[1]. В открывшемся окне редактора стилей введите описание стиля для строк ввода (листинг 1.10).

Листинг 1.10. Стиль для класса `QLineEdit`

```
QLineEdit{border-style: outset;border-width: 2px;border-radius: 8px;
    font: bold 12px; min-height: 1em; min-width: 10em;padding: 6px;}
QLineEdit:focus{background-color: yellow}
```

Описание стиля начинается с указания класса, для которого он предназначен. За указанием класса могут следовать дополнительные спецификаторы. Например, спецификатор `:focus` означает, что этот стиль должен быть применен к элементу, получившему фокус ввода. Затем в фигурных скобках следует само описание стиля. Элементы `border-style`, `border-width`, `border-radius` описывают свойства границы виджета. Элемент `font` описывает различные состояния шрифта. Элементы `min-height` и `min-width` — минимальную высоту и ширину виджета. Поскольку мы выбрали шрифт нестандартной высоты, то должны позаботиться о том, чтобы виджет был достаточно высок для отображения этого шрифта.

Примечание

Как говорит документация Qt, спецификатор `:focus` указывает псевдосостояние виджета. Концепция псевдосостояний используется не только в таблицах стилей, но и, например, в QML. Она позволяет описать не только статический внешний вид виджета, но и изменение этого вида в процессе его работы. Обратите внимание, что, описывая псевдосостояние `focus`, мы указываем только то, что должно измениться по сравнению с состоянием виджета по умолчанию.

[1] И не стоит ругать переводчиков в этом случае, ведь `styleSheet` — это название свойства, естественно, что они оставили его "как есть", без перевода.

В результате каждый компонент `QLineEdit`, добавленный в форму, примет вид, описанный в таблице стилей (рис. 1.16). Вы можете проверить, как работает заданный вами стиль в динамике, не покидая Qt Designer. Воспользуйтесь для этого командой **Предварительный просмотр** (Preview). Ничто не мешает вам задать индивидуальные стили для разных виджетов, используя свойство `styleSheet` конкретного виджета. А как быть, если вы хотите разделить одно описание стилей между несколькими окнами приложения?

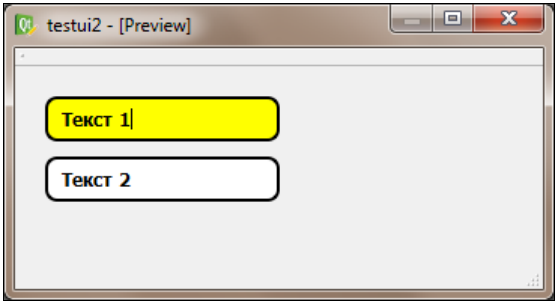


Рис. 1.16. Изменение стиля объектов `QLineEdit`

Один из способов — задать описание стиля с помощью метода `setStyleSheet()` объекта `QApplication`. В Qt Designer мы этого уже не увидим. Придется собрать приложение, чтобы посмотреть, как работает глобальный стиль.

Для многих виджетов удобно использовать растровую графику. Например, если вы хотите создать свой собственный стиль для флажка `QRadioButton`, то разумно нарисовать изображения флажка в установленном и сброшенном состоянии (можно также добавить изображения в неопределенном состоянии, в момент щелчка мышью и т. д.). Таблицы стилей позволяют использовать эти изображения в активной части виджета (листинг 1.11).

Листинг 1.11. Работа с растровой графикой в таблице стилей

```
QRadioButton::indicator::unchecked {
image: url(:/unchecked.png);
}
QRadioButton::indicator::checked {
image: url(:/checked.png);
}
```

Элемент `image` позволяет загрузить изображение, используя URL. Сам файл с изображением может храниться и на диске, но удобнее всего поместить его в модуле ресурсов, что мы и делаем, как вы можете видеть из листинга. Еще один важный вывод из этого примера: мы можем модифицировать отдельные элементы сложных виджетов Qt (в данном случае это элемент `QRadioButton::indicator`).

Контейнеры Qt library

В Qt 4 появилась система контейнеров Tulip (тюльпан), призванная заменить старые контейнеры Qt 3. Tulip предоставляет в наше распоряжение стандартный список контейнеров последовательного доступа (список, связный список, очередь, вектор, стек) и несколько ассоциативных контейнеров: отображения (классы `QMap` и `QMultiMap`), хеш-таблицы (классы `QHash` и `QMultiHash`) и набор (класс `QSet`). Теоретически разница между классами `QMap` и `QMultiMap` и классами `QHash` и `QMultiHash` заключается в том, что первый класс из каждой пары позволяет связать с ключом только одно значение, тогда как второй класс позволяет назначать по несколько значений одному ключу. Напомню, что в ассоциативных контейнерах хранимым значениям сопоставляются ключи, которые позволяют организовать произвольный доступ к данным контейнера. Разделение классов `QHash` и `QMultiHash` вызывает некоторые вопросы. При использовании хеш-таблиц нередко возникают коллизии (когда двум хранимым значениям соответствует один и тот же ключ). Придумать хеш-функцию, которая бы гарантированно не вызывала коллизий, очень трудно (а иногда и нереально). Фактически возможность связывать несколько значений с одним ключом является неотъемлемым свойством хеш-таблиц. Разработчики контейнеров Qt, разумеется, об этом знают, и в классе `QHash` реализована возможность добавления нескольких значений с одним и тем же ключом. В результате различие между контейнерами `QHash` и `QMultiHash` (второй, кстати, является потомком первого) выглядит, скорее, косметическим.

Сборка приложения Qt 4 под управлением CMake

Поскольку у нас уже появилось приложение Qt, обладающее всеми элементами "большого" приложения (я имею в виду пример designerdemo2), уместно рассмотреть процесс сборки этого приложения под управлением системы CMake. Хотя CMake и не является частью собственно библиотек Qt, эта система как нельзя лучше подходит для управления сборкой приложений Qt на разных платформах. Между прочим, CMake является официальной системой управления сборкой для оболочки KDE 4.x, которая основана на Qt 4.x. Для того чтобы CMake могла управлять сборкой нашей программы нужно, во-первых, установить CMake. При работе с ОС Linux лучше всего устанавливать CMake из онлайн-репозитория вашего дистрибутива Linux. Для установки CMake под ОС Windows вам потребуется скачать дистрибутив (например, отсюда: **http://www.cmake.org/cmake/resources/software.html**). Далее нам нужно написать файл CMakeLists.txt для приложения designerdemo2. На самом деле я его уже написал (листинг 1.12). Вы найдете его в папке Ch1/designerdemo2 на сайте **http://symmetrica.net/qt47book/**.

Листинг 1.12. Файл CMakeLists.txt для программы designerdemo2

```
project(designerdemo2 C CXX)
cmake_minimum_required(VERSION 2.6.0)
set (CMAKE_IN_SOURCE_BUILD TRUE)
```

```
set(SOURCES main.cpp mainwindow.cpp)
set(UIS MainWindow.ui)
set(MOC_HEADERS mainwindow.h)
set (designerdemo2_RCCS designerdemo2.qrc)
set(QM_FILES designerdemo2_ru.qm)
add_definitions(-Wall -g)
find_package(Qt4 REQUIRED)
include(${QT_USE_FILE})
qt4_wrap_ui(UI_HEADERS ${UIS})
qt4_wrap_cpp(MOC_SRCS ${MOC_HEADERS})
qt4_add_resources(designerdemo2_RCC_SRCS ${designerdemo2_RCCS})
include_directories( ${CMAKE_BINARY_DIR} )
add_executable(designerdemo2 ${SOURCES} ${UI_HEADERS} ${MOC_SRCS}
${designerdemo2_RCC_SRCS} ${QM_FILES})
target_link_libraries(designerdemo2 ${QT_LIBRARIES})
add_custom_target(translations DEPENDS ${QM_FILES})
qt4_add_translation(designerdemo2_ru.qm mainwindow.cpp ${MOC_SRCS}
${UI_HEADERS} designerdemo2_ru.ts)
```

Поскольку подробное описание CMake выходит за рамки данной книги, мы рассмотрим лишь основные моменты. Прежде чем разбирать пример, ознакомьтесь кратко со структурой языка сценариев CMake. Элементы языка сценариев CMake можно разделить на три категории: переменные, свойства и команды. Переменные играют в проектах CMake такую же важную роль, как и в файлах проектов. Настройка параметров сборки проекта выполняется, в основном, путем присвоения переменным различных значений. С помощью свойств проект CMake получает различные сведения о состоянии системы, для которой генерируются сборочные файлы. Свойства можно рассматривать как переменные, доступные только для чтения. Значения свойств устанавливаются средой CMake. Команды, как вы, конечно, догадались, выполняют различные действия с переменными и свойствами CMake и управляют процессом генерации файлов сборки. Важной особенностью языка CMake является его расширяемость. Модули CMake, о которых говорилось выше, позволяют определять новые переменные, свойства и команды. Именно благодаря расширяемости языка CMake этот пакет обрел свою популярность.

Язык CMake обладает частичной независимостью от регистра. Имена команд не зависят от регистра (`IF()`, `If()`, `if()` и `iF()` — одна и та же команда), в то время как имена свойств и переменных — зависят. В связи с этим возникает вопрос, как правильнее писать имена команд — с большой буквы, с маленькой или полностью заглавными буквами. Разные авторы пишут по-разному (в документации CMake имена команд пишутся полностью строчными буквами, а во многих файлах CMake, написанных самими разработчиками — заглавными). Лично я пишу имена команд с маленькой буквы и буду следовать этому принципу в листингах этой книги.

Команда `project()` задает имя проекта, а также позволяет указать (в качестве необязательного параметра) язык программирования. В нашем примере имя проекта `designerdemo2`, а `C` и `CXX` указывают на языки программирования C и C++ соответст-

венно. Хотя в нашей программе мы используем только C++, язык C тоже лучше указывать, т. к. он используется в служебных модулях CMake. Команда `cmake_minimum_required()` позволяет задать минимальный номер версии CMake, требуемой для сборки проекта.

Следующая команда, `set()`, является одной из наиболее часто используемых команд в файлах CMake. Команда `set()` позволяет присваивать значения переменным. Первым аргументом команды должно быть имя переменной (если переменной с этим именем ранее не существовало, она будет создана), далее следует список значений, которые присваиваются переменной (одной переменной можно присвоить сразу несколько значений). Например, строка

```
set(SOURCES main.cpp mainwindow.cpp)
```

присваивает переменной `SOURCES` значения `main.cpp` и `mainwindow.cpp` (нетрудно догадаться, что переменная `SOURCES` содержит список имен файлов исходных текстов программы). Обратите внимание, что элементы списка аргументов разделяются только пробелами. Стоит отметить еще одну особенность вызова переменных в CMake: переменная вызывается собственно по имени, только когда она является первым аргументом команды `set()`. В остальных случаях обращение к имени переменной имеет вид `${`*ИМЯ_ПЕРЕМЕННОЙ*`}`. Это сделано для того, чтобы не перепутать имя переменной, содержащей, например, имя файла, с самим именем файла.

Команда `add_definitions()` позволяет добавить дополнительные флаги компилятора.

Следующая команда

```
find_package(Qt4 REQUIRED)
```

приказывает CMake найти пакет для работы с Qt 4, причем спецификатор `REQUIRED` указывает на то, что без этого пакета сборка невозможна. Команды `qt4_wrap_ui()` и `qt4_wrap_cpp()` необходимы для работы со специфическими расширениями C++, которые использует Qt, наподобие макроса `Q_OBJECT`, сигналов и слотов. Команда `qt4_add_resources()` указывает, какие ресурсы следует добавить в приложение.

Команда `add_executable()` — одна из важнейших, поскольку она, собственно, задает конечную цель сборки — исполнимый файл приложения. Первый аргумент команды — имя исполнимого файла. Далее следует список переменных, которые содержат имена исходных файлов и промежуточных целей сборки. Команда `target_link_libraries()` определяет, какие разделяемые библиотеки следует подключить к создаваемому исполнимому файлу. Для Qt 4 перечень библиотек содержится в переменной `QT_LIBRARIES`, которая инициализируется пакетом расширений CMake `Qt4`. Остальные строки файла имеют отношение к переводу интерфейса, мы их сейчас рассматривать не будем. Теперь для сборки программы designerdemo2 достаточно перейти в директорию с исходными файлами и выдать команду:

```
cmake ./
```

Если результатом выполнения `cmake` станет make-файл (а это, напомню, зависит от платформы и выбранной системы сборки приложений), то в этом файле будет определено несколько целей. Помимо стандартных целей `all` и `clean`, будет определена цель `translations` для сборки ресурсов перевода интерфейса.

ГЛАВА 2

Взаимодействие объектов Qt между собой

В этой главе мы рассмотрим два механизма взаимодействия объектов Qt: события и связи между сигналами и слотами. Мы также обсудим особенности взаимодействия объектов в многопоточной программе и рассмотрим два весьма полезных алгоритма.

В языке C++ есть один серьезный пробел: не существует никакого стандартного механизма, с помощью которого объекты разных классов могли бы взаимодействовать друг с другом. Каким образом объект класса библиотеки, отвечающей за графический интерфейс, может передать информацию объекту класса, написанного программистом? Как сделать так, чтобы связи между объектами можно было изменять динамически во время работы программы? Спецификация C++ не отвечает на эти вопросы, так что разработчики разных библиотек на C++ сами придумывают соответствующие механизмы.

> ***Примечание***
>
> Такая ситуация существует далеко не во всех объектно-ориентированных языках программирования. Например, в Objective-C и C# средства взаимодействия между объектами встроены в сами языки.

Все эти методы взаимодействия могут синтаксически оформляться по-разному, но в их основе всегда лежат функции обратного вызова, потому что это единственный механизм, который предоставляет C++.

События Qt

Взаимодействие объектов Qt между собой и с внешним окружением происходит немного по-разному. Информацию о том, что происходит во внешнем окружении, программа получает, в основном, в виде событий. Обработчики событий — это обычные методы классов Qt, которым в качестве аргумента передается указатель на объект, описывающий событие. Эти объекты являются потомками объекта `QEvent`.

Методы-обработчики событий располагаются в разделах `protected`. Это означает, что вам придется напрямую иметь дело с событиями в тех случаях, когда вы создаете классы-потомки стандартных классов Qt.

Покажем это на простом, но полезном примере (одна из причин, по которой я в свое время перешел на Qt, заключалась именно в том, что в Qt можно без особого труда сделать что-то действительно полезное). Мы модифицируем компонент `QLineEdit` таким образом, чтобы в нем работало автоматическое завершение строки в стиле оболочки bash (для тех, кто не знаком с bash, поясню: вы набираете начало строки, нажимаете клавишу табуляции, и bash завершает строку, как умеет). Впрочем, последние выпуски консоли Windows демонстрируют аналогичное поведение.

Иначе говоря, наш компонент получает список из нескольких строк. Если пользователь набирает последовательность символов, которая является началом одной из строк автоматического завершения, а затем нажимает клавишу табуляции, то в строке ввода появляется вся строка целиком. Сразу предупреждаю, что мой компонент несколько недоработан: он не умеет правильно обрабатывать ситуацию, когда несколько строк для автозавершения начинаются с одного префикса (он всегда будет выводить первую строку). Вы можете сами доработать этот компонент.

Готовый текст программы, использующей этот компонент, вы можете найти в директории Ch2/testevents (исходные тексты к книге доступны на сайте **http://symmetrica.net/qt47book/**). Мы же рассмотрим процесс создания этой программы. В окне Qt Designer поместим компонент `QLineEdit`, а затем с помощью команды контекстного меню **Преобразовать в...** (*см. главу 1*) создадим его потомка — класс `AutoCompleteEdit` (рис. 2.1).

Теперь, как вы, конечно, помните, нам требуется объявление и реализация класса `AutoCompleteEdit`. Я совместил их в файле autocompleteedit.h (листинг 2.1).

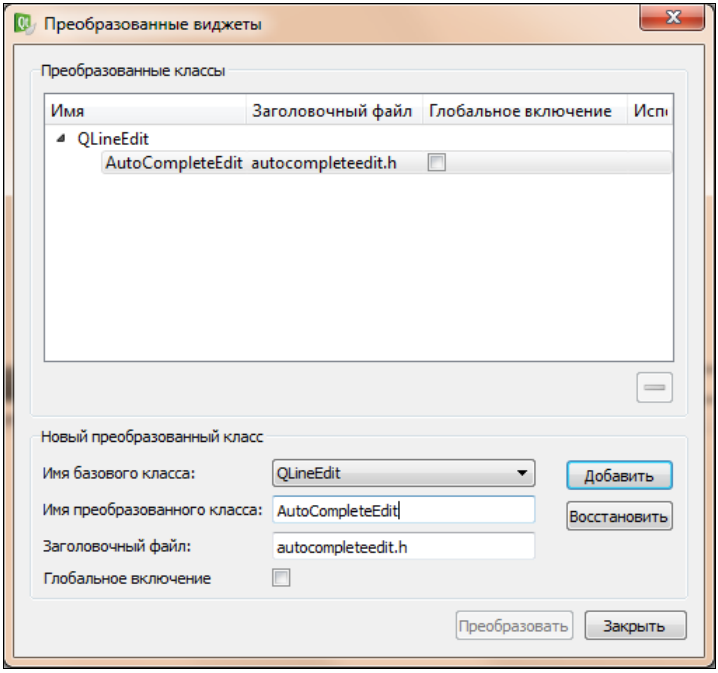


Рис. 2.1. Преобразование класса `QLineEdit` в `AutoCompleteEdit`

Листинг 2.1. Класс `AutoCompleteEdit`

```
#ifndef AUTOCOMPLETEEDIT_H
#define AUTOCOMPLETEEDIT_H
#include <QWidget>
#include <QLineEdit>
#include <QStringList>
#include <QKeyEvent>
class AutoCompleteEdit: public QLineEdit
{
    Q_OBJECT
public:
    explicit AutoCompleteEdit(QWidget* parent=0) : QLineEdit(parent)
    {
        autoCompleteList = new QStringList();
    }
    ~AutoCompleteEdit()
    {
        delete autoCompleteList;
    }
    void setAutoCompleteList(const QStringList &list)
    {
        (*autoCompleteList) = list;
    }
protected:
    void keyPressEvent(QKeyEvent * event)
    {
        if (event->key() == Qt::Key_Tab) {
            for (int i =  0; i < autoCompleteList->count(); i++)
                 if (autoCompleteList->at(i).startsWith(text())) {
                     setText(autoCompleteList->at(i));
                     break;
                 }
            event->accept();
            return;
        }
        QLineEdit::keyPressEvent(event);
    }
private:
        QStringList * autoCompleteList;
};
#endif // AUTOCOMPLETEEDIT_H
```

С точки зрения программиста, которому захочется использовать класс `AutoCompleteEdit` в своей программе, этот класс отличается от `QLineEdit` только наличием метода `setAutoCompleteList()`. Аргументом этого метода является ссылка на объект `QStringList`, который содержит список строк для автоматического завер-

шения. Внутри же нам, помимо реализации конструктора, деструктора и метода `setAutoCompleteList()`, потребуется перекрыть виртуальный метод `keyPressEvent()`, который обрабатывает события, вызванные нажатием клавиш, когда наш компонент обладает фокусом ввода.

> ***Примечание***
>
> Вы можете подумать, что операция присваивания, которая выполняется в методе `setAutoCompleteList()`, является либо слишком затратной, либо небезопасной, т. к. выполняется только поверхностное копирование. Первое утверждение неверно, второе верно лишь отчасти. Для работы со структурами данных, предназначенными для больших объемов данных (контейнеры входят в их число), Qt использует механизмы implicit sharing и copy-on-write. Это значит, что если две структуры ссылаются на один и тот же массив данных, этот массив не дублируется. Дублирование происходит тогда, когда одна из структур модифицирует данные. Таким образом, при копировании данных одного списка строк в другой затрат будет не больше, чем при неглубоком копировании.

Теперь рассмотрим гораздо более интересный метод `keyPressEvent()`. Информация о событии передается обработчику в параметре `event`, который указывает на экземпляр класса `QKeyEvent`. Этот объект содержит всю информацию о событии, связанном с нажатием клавиши на клавиатуре. В частности метод `event->key()` содержит код нажатой клавиши. Мы проверяем, не нажата ли клавиша табуляции. Если нажата именно она, мы ищем в списке `autoCompleteList` строки, которые начинаются с текста, введенного в компоненте (получить этот текст можно с помощью метода `text()`). Если такая строка найдена, мы присваиваем ее компоненту (методом `setText()`) и выходим из цикла.

Вызов метода `event->accept();` означает, что мы обработали событие (иначе оно может быть передано родительскому виджету, а нам это не требуется). Если нажата какая-то другая клавиша, мы вызываем метод `keyPressEvent()` базового класса `QLineEdit`, передавая ему по эстафете переменную `event`. В результате класс `AutoCompleteEdit` будет вести себя точно так же, как `QLineEdit`, за исключением обработки клавиши табуляции.

Дотошный читатель может вспомнить о том, что клавиша табуляции обычно используется для переключения фокуса ввода. Как наш виджет разрешает этот конфликт? Благодаря нашему методу обработки события и особенностям Qt library конфликт разрешается вполне разумно: если наш виджет — единственный потенциальный акцептор фокуса ввода в окне и переключаться с помощью клавиши табуляции просто некуда, то клавиша табуляции будет работать так, как описано ранее. Если кроме нашего виджета в окне есть и другие, которые могут получить фокус ввода, то простое нажатие клавиши табуляции приведет к переключению фокуса ввода (этим управляет другой механизм, а наш виджет в этом случае просто не получит информацию о нажатии клавиши). Но выполнять автоматическое завершение в строке ввода все равно можно, только для этого клавишу табуляции придется применять с модификатором (клавишей <Ctrl> или <Alt>).

Существует и другой способ разрешения конфликта. Допустим, что кроме строки ввода окно содержит еще один виджет-акцептор ввода — кнопку `QPushButton`

(в примере на сайте это так и есть). В этом случае мы можем в редакторе свойств объекта класса `QPushButton` изменить значение свойства `focusPolicy` со `StrongFocus` на `ClickFocus`. Теперь кнопка сможет получать фокус ввода только при щелчке мышью, но не при нажатии клавиши табуляции. В результате строка ввода снова останется единственным "переключаемым" элементом и снова сможет обрабатывать нажатие клавиши табуляции так, как было задумано.

Еще один способ разрешения конфликта, правда, более трудоемкий, предлагает документация по Qt. Мы бегло рассмотрим этот способ (подробное описание вы найдете в документации), главным образом потому, что он позволяет понять, как работает механизм событий Qt на низком уровне. Я уже писал о том, что "система" скрывает от нашего виджета нажатие клавиши табуляции, если это нажатие может быть использовано для переключения фокуса ввода. Сослаться на систему, конечно, удобно, но лучше разобраться, как она работает. У каждого объекта-потомка `QObject` есть метод-диспетчер событий. Этот метод называется просто `event()`. Его единственным аргументом является указатель на объект класса `QEvent` (или на объект любого потомка этого класса). Иначе говоря, функции `event()` можно передать указатель на объект, соответствующий любому событию Qt. Метод-диспетчер изучает тип переданного ему объекта, несущего информацию о событии, и решает, какая именно функция-обработчик должна быть вызвана для обработки данного события. Если функция `event()` смогла отправить событие "по назначению" и событие было корректно обработано, функция возвращает значение `true`, в противном случае — `false`. Рассмотрим возможный вариант функции `event()` для нашего примера (листинг 2.2).

Листинг 2.2. Перекрытая функция `event()`

```
bool event(QEvent *event)
{
if (event->type() == QEvent::KeyPress) {
QKeyEvent *ke = static_cast<QKeyEvent *>(event);
if (ke->key() == Qt::Key_Tab) {
    for (int i =  0; i < autoCompleteList->count(); i++)
        if (autoCompleteList->at(i).startsWith(text())) {
            setText(autoCompleteList->at(i));
            break;
        }
        return true;
    }
}
        return QLineEdit::event(event);
}
```

У каждого объекта-потомка `QEvent` есть метод `type()`, который возвращает идентификатор типа события. Мы вызываем этот метод для того, чтобы определить, име-

ем ли мы дело с событием `QKeyEvent` или с каким-то другим. Если это событие `QKeyEvent` и была нажата клавиша табуляции, мы выполняем уже знакомую вам последовательность действий, после чего возвращаем `true` (событие распознано и обработано). Если одно из перечисленных условий не выполняется, мы просто передаем параметр `event` методу `event()` базового класса. Теперь клавиша табуляции в строке ввода будет делать то, что нужно нам, независимо от внешних условий.

Думаю, вы понимаете, какую мощь дает нам в руки перекрытие метода `event()`. Мы можем блокировать события или заменять одно событие другим по своему усмотрению. Единственный недостаток такого подхода заключается в том, что для перекрытия стандартного метода `event()` того или иного виджета нам придется создавать класс-потомок этого виджета. Далее мы узнаем, как манипулировать событиями классов извне, не создавая их потомков.

Кроме того, функция `event()` дает нам возможность эмулировать события. Например, если мы хотим программно эмулировать щелчок мышью по виджету, нам требуется создать объект класса `QMouseEvent` и передать указатель на этот объект методу `event()` соответствующего виджета. Дальше все будет выглядеть так, как будто мы действительно выполнили некоторое действие с помощью мыши. После возвращения из метода `event()` объект-событие может быть уничтожен. Обратите внимание на то, что ни метод `event()`, ни методы-обработчики событий никогда не уничтожают объект, указатель на который им передан. Вы в своих реализациях методов `event()` и обработчиков событий тоже ни в коем случае не должны этого делать. В этом случае действует известное "правило хорошего тона" при программировании на C++: объект уничтожается на том уровне вложенности, на котором он был создан (или, по-другому: "не ты создавал — не ты уничтожаешь").

Примечание

Вы можете спросить: а не лучше ли использовать для автоматического завершения не клавишу табуляции, а какую-нибудь другую, чтобы избежать конфликта? Нет. Во-первых, клавиш, не нагруженных каким-либо значением, на клавиатуре не так уж и много. Для автоматического завершения можно использовать, например, одну из функциональных клавиш верхнего ряда клавиатуры, но это будет непривычно для пользователей, ведь именно клавиша табуляции используется для автоматического завершения строки во многих программах, в том числе обладающих графическим интерфейсом (пример — строки ввода некоторых диалоговых окон оболочки KDE).

Раз уж зашла речь о вызове `event()` для эмуляции событий, то стоит отметить, что это не единственная возможность эмуляции. К методу `event()` конкретного объекта мы обращаемся в том случае, если мы хотим, чтобы именно этот объект обработал именно это событие. Но такая эмуляция не всегда будет выглядеть правдоподобно. Некоторые события обрабатываются цепочками дочерних объектов. Очень часто, если объект не может сам обработать событие, он передает его родительскому объекту. Для того чтобы эмулировать стандартный процесс обработки события приложением, необходимо воспользоваться методом `QCoreApplication::notify()`. Напомню, что класс `QCoreApplication` является предком класса `QApplication`, с которым мы уже знакомы. Метод `notify()` этого класса имеет два параметра. В параметре `receiver` передается указатель на объект, который должен обработать событие.

В параметре `event` передается указатель на объект `QEvent` или его потомка. Метод возвращает значение типа `bool`, которое имеет тот же смысл, что и значение, возвращаемое методом `event()`. В простейшем случае метод `notify()` делает вызов:

```
return receiver->event(event);
```

В более сложных случаях может произойти передача события дочернему или родительскому объектам.

> ***Примечание***
>
> Перекрытие метода `QCoreApplication::notify()` предоставляет вам огромные возможности, но пользоваться ими надо осторожно. Ведь метод `QCoreApplication::notify()` управляет передачей событий всем объектам приложения и ошибка в его реализации может сделать все приложение неработоспособным, или, что еще хуже, породить ошибки, которые проявят себя не сразу и источник которых потом будет трудно обнаружить.

Нам осталось посмотреть на наш виджет в работе. Все, что нам требуется, — инициализировать его списком строк для автоматического завершения (листинг 2.3).

Листинг 2.3. Конструктор диалогового окна

```
Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
    ui->setupUi(this);
    QStringList sl;
    sl << "FreeBSD" << "Linux" << "MacOS X"
       << "QNX" << "Symbian" << "Windows";
    ui->lineEdit->setAutoCompleteList(sl);
}
```

Мы заполняем список автозавершения именами операционных систем, которые поддерживает Qt library. Так что, если вы наберете в строке ввода "Wi", а затем нажмете клавишу табуляции (возможно, с модификатором), в строке ввода появится имя нашей самой любимой операционной системы.

Еще один случай, когда вам придется иметь дело с событиями, — реализация механизма Drag and Drop (перетащить и бросить) в окне, которое по умолчанию для этого не предназначено. Для демонстрации того, как это работает, мы модифицируем программу designerdemo2. Этот модифицированный вариант вы найдете под именем dragndropdemo в папке Ch2. Суть нашей модификации заключается в том, чтобы открывать в программе графические файлы, просто перетаскивая их в окно программы мышью. Для того чтобы добавить такую полезную функциональность в нашу программу, нам потребуется внести совсем немного изменений. А именно, перекрыть в классе `MainWindow` функции `dragEnterEvent()` и `dropEvent()` сначала в заголовочном файле (листинг 2.4), а затем и в файле реализации (листинг 2.5).

Листинг 2.4. Класс окна с возможностью "перетащить и бросить"

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include "ui_MainWindow.h"
#include <QMainWindow>
class QDragEnterEvent;
class QDropEvent;
class MainWindow : public QMainWindow
{
    Q_OBJECT
 private:
    Ui::MainWindow * ui;
public:
    explicit MainWindow(QWidget *parent = 0);
protected:
    void dragEnterEvent(QDragEnterEvent *event);
    void dropEvent(QDropEvent *event);
signals:
private slots:
    void openFile();
};
#endif // MAINWINDOW_H
```

Листинг 2.5. Реализация класса окна с возможностью "перетащить и бросить"

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent)
{
    ui = new Ui::MainWindow();
    ui->setupUi(this);
    connect(ui->actionOpen_File, SIGNAL(triggered()), this,
        SLOT(openFile()));
    setAcceptDrops(true);
}
void MainWindow::openFile()
{
    QPixmap pm;
    QString fn = QFileDialog::getOpenFileName(this,
        trUtf8("Open image"));
    if (fn != "") {
        if (!pm.load(fn))
            QMessageBox::critical(this, trUtf8("Error"),
                trUtf8("Unable to load image from the selected file."));
        else {
            ui->label->setPixmap(pm);
```

```
            setWindowTitle(trUtf8("Image Viewer — ") + fn);
        }
    }
}
void MainWindow::dragEnterEvent(QDragEnterEvent *event)
{
    if (event->mimeData()->hasUrls())
        event->accept();
}
void MainWindow::dropEvent(QDropEvent *event)
{
    if (event->mimeData()->hasUrls()) {
        QPixmap pm;
        QString fn = event->mimeData()->urls().at(0).toLocalFile();
        if (pm.load(fn)) {
            ui->label->setPixmap(pm);
            setWindowTitle(trUtf8("Image Viewer — ") + fn);
        }
        event->accept();
    }
}
```

В конструкторе `MainWindow` мы вызываем метод `setAcceptDrops()` с параметром `true`. Тем самым мы объявляем системе, что окно `MainWindow` может служить приемником для перетаскиваемых мышью объектов. Событие `QDragEnterEvent` оповещает нас о том, что курсор мыши вошел в область виджета (в нашем примере — главного окна) в режиме перетаскивания. При этом нам, прежде всего, необходимо определить, что именно "тащит" мышь. У объектов `QDragEnterEvent` и `QDropEvent` есть метод `mimeData()`, который возвращает указатель на объект `QMimeData`. Этот объект позволяет определить формат перетаскиваемых данных и получить доступ к самим данным.

Наша программа принимает ссылки на файлы. Все подобные ссылки, независимо от их источника, преобразуются системой Qt Drag and Drop в универсальные ссылки URL. Метод `hasUrls()` объекта `QMimeData` позволяет нам определить, содержат ли перетаскиваемые данные ссылки URL. Если содержат, то мы вызываем метод `accept()` объекта `event`. В этом примере метод `accept()` играет более важную роль, чем в предыдущем. Он не только сообщает Qt, что мы готовы обработать событие, но и сообщает системе, что наша программа готова принять перетаскиваемый объект.

> ***Примечание***
>
> Конечно, не все ссылки URL подходят для нашей программы. Мы могли бы организовать проверку, является ли перетаскиваемая ссылка ссылкой на локальный графический файл или сетевым адресом.

Если метод `accept()` вызван, курсор мыши принимает соответствующий вид ("сюда можно бросать перетаскиваемый объект"), в противном случае курсор примет вид запрещающего знака.

Метод `dropEvent()` вызывается в том случае, если пользователь отпустил кнопку мыши в режиме перетаскивания объектов над нашим окном, и мы готовы принять соответствующий объект. Здесь мы снова проверяем, содержит ли перетаскиваемый объект ссылки URL. В принципе, в этом нет необходимости, т. к. если мы не вызвали метод `accept()` в обработчике `dragEnterEvent()`, то обработчик `dropEvent()` все равно не будет вызван, но дополнительная проверка лишней не будет. Метод `event->mimeData()->urls()` возвращает список перетаскиваемых ссылок (`QList<QUrl>`). Из этого списка мы берем самую первую ссылку (все равно наша программа может показать за один раз только одно изображение) и с помощью метода `toLocalFile()` преобразуем ее в ссылку на файл на диске. Дальнейшие действия должны быть вам уже знакомы.

У пытливого читателя может возникнуть вопрос: почему мы занимаемся обработкой событий Drag and Drop в классе `MainWindow`, а не в классе `QLabel`, и как `MainWindow` "узнает" об этих событиях, если все его активное пространство закрыто объектом `label`. Мы, конечно, могли бы создать потомка класса `QLabel` и организовать обработку событий Drag and Drop в этом классе, но это только усложнило бы наш пример без всякой на то необходимости. Дело в том, что встречая событие, которое он не может обработать, например, событие Drag and Drop, объект `QLabel` просто передаст его родительскому виджету, т. е. классу `MainWindow`.

Обработка событий, более подробное обсуждение

Вы, наверное, уже догадались, что главный цикл обработки событий прячется в недрах функции `exec()` объекта, представляющего приложение. Каким именно образом функция `exec()` обрабатывает события? Все события можно разделить на три группы: спонтанные, отложенные и немедленного выполнения. Источником спонтанных событий является операционная система. Эти события могут быть связаны с действиями пользователя или с какими-то происшествиями внутри самой системы (например, с появлением очередной порции данных в сетевом канале). Хотя нам кажется, что спонтанные события должны всегда обрабатываться в первую очередь, Qt отнюдь этого не гарантирует. Большинство ОС, с которыми работает Qt, не являются ОС реального времени (не следует путать ОС реального времени и интерактивную ОС) и в их спецификациях никак не отражено максимальное время реакции на события. Отложенные события — это те события, которые генерируются с помощью метода `QCoreApplication::postEvent()`. Список параметров у этого метода такой же, как и у `QCoreApplication::notify()`, но, в отличие от `notify()`, `postEvent()` возвращает управление сразу и ничего не сообщает о том, было событие обработано успешно или нет. Вызывая `postEvent()`, вы просто помещаете событие в очередь на обработку. В связи с этим возникает вопрос, кто уничтожает объект-потомок `QEvent`, который вы передаете методу `postEvent()`. Поскольку очередь обработки данного события может наступить уже после того, как метод `postEvent()` вернул управление программе, вы не должны сами уничтожать объект события,

который вы создали. Объект события для метода `postEvent()` не может также быть локальной переменной (поскольку в этом случае он автоматически уничтожится при выходе из области видимости переменной). Вы создаете объект события с помощью оператора `new`, передаете его методу `postEvent()` и дальше жизненный цикл события управляется средой Qt. Среда Qt сама удалит этот объект, когда он больше не будет нужен (так что правило не удалять объекты в функциях-перехватчиках и обработчиках справедливо и для отложенных событий). События немедленного выполнения обрабатываются непосредственно в процессе выполнения функции, которая генерирует их. Метод `QCoreApplication::sendEvent()` генерирует такие события, используя метод `notify()`. К моменту возврата из `sendEvent()` событие гарантированно обработано (или отвергнуто) системой и объект события больше не нужен. Поэтому объекты событий для `sendEvent()` обычно размещаются в стеке функции, вызывающей `sendEvent()`, как локальные переменные.

Хотя `sendEvent()` гарантирует, что событие будет тут же обработано, чрезмерное использование этого метода может замедлить выполнение программы. Дело в том, что вы, как правило, не можете знать заранее, с какой частотой будут генерироваться те или иные события (это зависит от многих факторов, которые просто невозможно предвидеть). Представьте себе, что в силу каких-то причин объект сгенерировал 10 событий `QPaintEvent` подряд (напомню, что событие `QPaintEvent` заставляет виджет перерисовать себя). Если для передачи этих событий виджету использовать метод `sendEvent()`, то виджет перерисует себя 10 раз подряд. Очевидно, что это приводит к ненужному расходованию ресурсов процессора. Если же эти 10 событий будут помещены в очередь обработки с помощью метода `postEvent()`, то "умная" система обработки событий Qt "увидит", что обработка этих 10 событий приведет к одному и тому же результату и вместо 10 событий обработает одно. Этот прием, получивший название "сжатие событий" (event compression), позволяет существенно сэкономить вычислительные ресурсы и ускорить работу программы. Разумеется, сжатие событий не всегда желательно. Например, в программе обработки мультимедийных данных вам может быть необходимо, чтобы виджет перерисовывался ровно столько раз, сколько вы этого потребовали. Именно для таких случаев и существуют события немедленного выполнения. Их, например, использует метод `QWidget::repaint()`. Следует помнить однако, что поскольку события немедленного выполнения обрабатываются, минуя общую очередь, их поток может полностью заблокировать обработку других событий, что также может быть нежелательно.

Схематически очередь обработки событий можно представить так, как показано в листинге 2.6.

Листинг 2.6. Схематическое представление очереди обработки событий Qt

```
while (!exit_was_called) {
   while (!posted_event_queue_is_empty) {
  process_next_posted_event();
   }
```

```
   while (!spontaneous_event_queue_is_empty) {
      process_next_spontaneous_event();
   }
   while (!posted_event_queue_is_empty) {
      process_next_posted_event();
   }
}
```

Сначала цикл обрабатывает все отложенные события, находящиеся в очереди. Затем обрабатываются все спонтанные события, затем обрабатываются все отложенные события, которые могли возникнуть в процессе обработки спонтанных событий, и все это происходит до тех пор, пока система не получит событие (обычно — спонтанное), приказывающее ей выйти из цикла (и завершить тем самым работу программы). На первый взгляд вам может показаться, что этот схематичный код написан новичком в программировании, и что один из блоков обработки отложенных событий можно безболезненно убрать, сделав код более компактным и элегантным. Это не так. Во-первых, данный код написан Джасмином Бланшеттом (Jasmin Blanchette), которого нельзя назвать новичком (код позаимствован из статьи, опубликованной указанным автором в Qt Quarterly), во-вторых, каждый элемент этой схемы необходим для правильного выполнения цикла обработки сообщений Qt. Предположим, что мы сократили последний блок обработки отложенных событий. Это будет означать, что каждый раз, переходя от обработки спонтанных событий к отложенным, система будет проверять, не приказано ли ей остановиться. Если система получила спонтанное событие, приказывающее завершить работу, работа будет завершена немедленно, и все отложенные события, находящиеся в очереди, просто будут потеряны. Если мы удалим первый блок обработки отложенных событий, то получится, что спонтанные события, источником которых обычно является операционная система, имеют более высокий приоритет, нежели отложенные события, источником которых является система Qt. А это противоречит идеологии Qt. Дело в том, что некоторые отложенные события могут сделать обработку некоторых спонтанных событий ненужной, тогда как обратное маловероятно. Вернемся к примеру с перерисовкой. Отложенное событие, приказывающее виджету стать невидимым (например, по причине закрытия окна) может сделать бессмысленным спонтанное событие, требующее перерисовки этого виджета. В общем, можно сказать, что в представленном варианте цикла проверка условия выхода из цикла выполняется между двумя внутренними циклами, каждый из которых опустошает очередь отложенных событий, и такой подход значительно увеличивает вероятность того, что отложенные события не будут потеряны, а "лишние" спонтанные события не будут выполнены.

Если вы хотите генерировать события для объектов Qt, метод `QCoreApplication::postEvent()` является предпочтительным способом, и не только из-за технологии сжатия событий. К преимуществам `postEvent()` можно отнести следующие:

- отсутствие зацикливания и взаимоблокировки. Поскольку отложенные события не выполняются сразу, а только ставятся в очередь, их использование практиче-

ски исключает бесконечные циклы и взаимные блокировки, которые могут появиться при использовании `sendEvent()` (об этом подробнее будет сказано далее);

- перегруженный вариант метода `postEvent()` позволяет указать численный приоритет события. События с более высоким приоритетом будут обработаны раньше, чем события с более низким.

Вообще, если ваша функция не вызывает метод `QCoreApplication::processEvents()`, и получатель события принадлежит тому же потоку, что и функция, вызывающая `postEvent()`, то событие, отправленное в очередь с помощью `postEvent()`, всегда будет выполняться после завершения функции, вызвавшей `postEvent()`. Используя эту особенность `postEvent()`, вы можете организовать контрактное программирование. Речь идет не о способе найма сотрудников, а о стиле программирования, в котором функция сообщает приложению, что ему делать после завершения функции.

Метод `QCoreApplication::processEvents()` позволяет выполнить цикл обработки сообщений, не покидая текущей функции Qt. Если функция Qt занята выполнением длительной операции, вызов этого метода позволит "оживить" интерфейс программы (хотя более грамотно располагать длительные операции в отдельных потоках, не блокирующих пользовательский интерфейс). Первый параметр метода указывает, какие именно события следует обработать. Можно, например, пропустить обработку событий, связанных с вводом пользователя, или событий, связанных с сокетами. Второй параметр задает максимальное время ожидания появления событий (вы можете приказать функции ждать появления события, если на момент ее вызова очередь обработки была пуста).

> ***Примечание***
>
> Метод `QCoreApplication::processEvents()` является статическим, и вызывать его надо именно как статический. У класса `QApplication`, который является потомком `QCoreApplication`, тоже есть статический метод `processEvents()`, который вызывается без параметров.

Интересен также метод `QCoreApplication::sendPostedEvents()`, который заставляет выполнить немедленную обработку отложенных событий определенного типа, адресованных определенному получателю. Этот метод игнорирует события, источником которых является операционная система.

События Qt и многопоточность

По умолчанию у приложения Qt есть только один цикл обработки событий, который обрабатывает события для всех потоков. Имеются в виду потоки, созданные с помощью класса `QThread`. Если вы создаете поток средствами операционной системы внутри приложения Qt, вы сами отвечаете за то, что делаете. Например, метод `QCoreApplication::notify()` рассылает события по всем потокам (в соответствии с тем, какому потоку принадлежит объект-получатель). Однако методы `QCoreApplication::notify()` и `QCoreApplication::sendEvent()` не являются потокобезопасными, поскольку требуют немедленной обработки события. Представьте себе, что произойдет, если мы пошлем из одного потока событие, адресованное

другому потоку, с помощью метода `sendEvent()`. Поскольку `sendEvent()` вызовет обработчик события напрямую (с помощью метода `notify()`), этот обработчик будет выполняться в потоке, вызвавшем `sendEvent()`, и при этом он будет оперировать данными из потока, которому принадлежит объект-получатель. Излишне, наверное, говорить о том, что поток, которому принадлежит объект-получатель, может в то же самое время обратиться к тем же данным, что вызовет конфликт доступа. И это еще одна причина использовать для отправки событий метод `postEvent()`.

Метод `QCoreApplication::postEvent()` тоже может отправлять события из одного потока другому потоку, но, в отличие от `sendEvent()`, этот метод является потокобезопасным. Отложенные события выполняются в контексте того потока, которому они адресованы, а не того потока, в котором они созданы. Это еще одна причина, по которой вы должны "забыть" об объекте события после передачи его методу `postEvent()`. Кроме того, цикл обработки сообщений следит за тем, чтобы обработчик событий корректно включился в механизм обработки событий потока-получателя, так что конфликт доступа не возникнет. Метод `processEvents()` во всех своих вариантах обрабатывает только события, находящиеся в очереди того потока, для которого он был вызван.

Удаление объекта после выхода из его метода

С помощью метода `deleteLater()` вы можете отдать приказ Qt удалить объект из одного из методов этого объекта. Разумеется, удаление произойдет после выхода из метода. Это еще один элемент контрактного программирования в Qt. Метод `deleteLater()` работает очень просто: он помещает в очередь обработки событий событие, требующее удалить данный метод. Нечто подобное можно сделать, например, с помощью сообщений Windows, используя функцию `PostMessage()` Windows API. Однако метод `deleteLater()` предлагает более простой и, к тому же, кросс-платформенный способ решения этой задачи.

Фильтрация событий

Для того чтобы управлять событиями, адресованными некоему объекту, необязательно создавать потомка класса и перекрывать в нем метод-обработчик события. С незапамятных времен в Qt существует и другой способ — фильтрация событий. Фильтр событий получает все события, адресованные тем объектам, для которого он установлен (один фильтр может быть установлен для нескольких объектов). Фильтр может заблокировать дальнейшее прохождение события (возможно, обработав его самостоятельно) или разрешить стандартную обработку. Фильтром событий может быть любой потомок `QObject`, у которого реализован метод `eventFilter()`. Для того чтобы установить фильтр, надо вызвать метод `installEventFilter()` того объекта, события которого мы хотим фильтровать, и передать ему ссылку на объект-фильтр событий. В качестве практической демонстрации мы рассмотрим все тот же многострадальный пример с автоматическим завер-

шением строки, но на этот раз мы не будем создавать потомка `QLineEdit`, а установим фильтрацию событий для самого объекта класса `QLineEdit` (листинг 2.7).

Листинг 2.7. Расширение возможностей `QLineEdit` с помощью фильтрации событий

```
class EventFilter : public QObject
{
public:
    bool eventFilter(QObject *object, QEvent *event)
    {
        if (event->type() == QEvent::KeyPress) {
            QKeyEvent *ke = static_cast<QKeyEvent *>(event);
            if (ke->key() == Qt::Key_Tab) {
                QLineEdit * le = static_cast<QLineEdit *>(object);
                QStringList sl;
                 sl << "FreeBSD" << "Linux" << "MacOS X"
                    << "QNX" << "Symbian" << "Windows";
                 for (int i =  0; i < sl.count(); i++)
                     if (sl.at(i).startsWith(le->text())) {
                        le->setText(sl.at(i));
                        break;
                     }
                return true;
            } else
                return false;
        }
        return false;
    }
};

Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
    ui->setupUi(this);
    eventFilter = new EventFilter();
    ui->lineEdit->installEventFilter(eventFilter);
}
Dialog::~Dialog()
{
    delete ui;
    delete eventFilter;
}
```

Полные исходные тексты этого примера вы найдете в папке Ch2/eventfilter. Наш фильтр событий определен в классе `EventFilter`. Метод `eventFilter()` получает два параметра: указатель на объект фильтрации, для которого он вызван, и указатель на

объект события, которое нужно "отфильтровать". Метод возвращает значение типа `bool`, от которого зависит, будет ли событие передано далее по цепочке обработки (значение `false`) или дальнейшая обработка события не нужна (значение `true`). Мы устанавливаем фильтр событий только для объекта `lineEdit` класса `QLineEdit`, так что можно не сомневаться, что всякий вызов метода `eventFilter()` делается именно для этого объекта. Как и функция `event()`, наш метод вызывается для всех событий, так что сначала мы проверяем тип события. Обратите внимание, что как и в случае с функцией `event()`, этот метод обработки нажатия клавиш получает полный контроль над клавишей табуляции, так что ее стандартная функция (переключение фокуса ввода) оказывается заблокированной.

Возможно, вы захотите перехватывать все события, адресованные объектам данного приложения. Для этого вам понадобится установить глобальный фильтр, который можно установить с помощью метода `QCoreApplication::InstallEventFilter()`. Как и у любого контроля, у полного контроля над событиями приложения есть своя обратная сторона. Использование глобальных фильтров может существенно замедлить работу программы, так что не пользуйтесь ими без крайней необходимости.

Кроме того, у вас есть возможность установить глобальный перехватчик сообщений системы. Устанавливается он с помощью метода `QCoreApplication::setEventFilter()` Глобальный перехватчик событий выглядит совсем не так, как локальный фильтр событий. Прежде всего, глобальный перехватчик — это не объект, а функция вида

```
bool myEventFilter(void *message, long *result);
```

Эта функция получает сообщения системы в "родном" формате и не является кросс-платформенной.

Сигналы и слоты Qt library

Сигналы Qt представляют собой более легкий и более гибкий способ обмена данными между объектами Qt. Прямое использование событий — случай довольно специфический, многие программы Qt вообще обходятся без собственных обработчиков событий, а вот программу, не использующую сигналы и слоты, представить себе довольно трудно. Я надеюсь, что вы уже знаете базовые приемы работы с сигналами и слотами, тем не менее, кратко повторю. Когда один объект Qt желает передать информацию другим объектам, он эмитирует нечто, называемое сигналом. Если этот сигнал связан со слотами других объектов, вызываются эти слоты. Слоты — это методы объектов Qt (и их можно вызывать как обычные методы), объявленные в одном из разделов `slots`. Слоты похожи на обработчики событий. Главных отличий несколько. Во-первых, слоты обладают гораздо большей свободой в том, что касается передачи параметров. Если обработчику события всегда передается указатель на объект-потомок `QEvent` и система управляет жизненным циклом этого объекта по определенным правилам, то для слотов списки передаваемых параметров не регламентированы. Многие слоты вызываются вообще без параметров, другим передаются параметры простых типов, но возможна и передача сложных

структур данных. С одним сигналом может быть связано несколько слотов и один слот может быть связан с несколькими разными сигналами.

Списки параметров связанных сигналов и слотов не обязаны совпадать в точности. Правило тут такое же, как при вызове функций C: список параметров слота должен совпадать с началом списка параметров сигнала, но при этом сигнал может передавать больше параметров. Например, сигнал с одним или несколькими параметрами может быть связан со слотом, который вообще не принимает параметров, или сигнал с двумя параметра типа `int` может быть связан со слотом с одним параметром этого типа (но сигнал с одним параметром не может быть связан со слотом, у которого два параметра). Объяснить это можно так: если некоторые параметры сигнала не будут востребованы слотом, ничего страшного не случится, но слот — это обычная функция и всем его параметрам (если они есть) должны быть присвоены значения.

Рассмотрим стандартную процедуру связывания сигнала и слота:

```
Connect(ui->actionOpen_File, SIGNAL(triggered()), this,
        SLOT(openFile()));
```

В этом методе сигнал `triggered()` объекта `actionOpen_File` связывается со слотом `openFile()` объекта `this`. Макросы `SIGNAL()` и `SLOT()` преобразуют переданные им аргументы в строки `QString`. Само связывание сигналов и слотов происходит во время выполнения программы и выглядит примерно так: функция `connect()` выполняет интроспекцию объекта источника сигнала и объекта получателя (используя для этого информацию о типах во время выполнения) и выясняет, имеются ли соответствующие сигнал и слот. Если сигнал и слот найдены, то функция выполняет связывание. Если сигнал или слот не найдены, либо их заголовки не соответствуют тем, что переданы функции `connect()`, связывание не выполняется, на консоль выводится предупреждающее сообщение, но работа программы на этом не прекращается. У такого подхода есть и плюсы, и минусы. Минус заключается в том, что ошибки, связанные с сигналами и слотами, можно просто не заметить. Плюс же заключается в том, что система сигналов и слотов обладает такой гибкостью в связывании разных объектов между собой, какую не может обеспечить стандартный C++. У метода `connect()` есть комплиментарный метод `disconnect()`, который отсоединяет сигнал от слота.

Сигналы и слоты не являются виртуальными. В связи с этим уместно обратить внимание на особенность связывания сигналов и слотов. Допустим, ваш объект получает указатель на объект-потомок `QWidget` в виде переменной `widget` типа `QWidget *`. У класса `QWidget` нет слота `setText()`, но вы знаете, что у того объекта, адрес которого содержится в указателе *на самом деле*, такой слот есть. В этой ситуации вы можете выполнить связывание следующим образом:

```
connect(this, SIGNAL(textChanged(const QString &), widget,
        SLOT(setText(const QString &)));
```

И оно сработает. Дело в том, что когда Qt во время выполнения программы ищет слоты объекта, на который указывает `widget`, система ориентируется не на тип ука-

зателя, заданный на этапе компиляции, а на информацию о типе, которую сам объект может предоставить во время выполнения. Так что, если слот `setText()` у объекта имеется, он будет связан. А если такой слот все же не будет обнаружен, система просто выдаст на консоль сообщение об этом (а метод `connect()` вернет значение `false`), но программа продолжит работать. Таким образом, механизм сигналов и слотов предоставляет вам большую гибкость, чем стандартный полиморфизм объектов C++.

Как и в случае с событиями, сигналы можно разделить на синхронные (выполняющиеся немедленно) и отложенные. Если ваш класс эмитирует синхронный сигнал, с которым связаны некоторые слоты, это эквивалентно тому, как если бы эти слоты были вызваны из вашего класса непосредственно. Что мешает вам именно так и сделать, т. е. вызвать методы другого объекта напрямую? В принципе, ничего. Только вы должны уже на этапе программирования знать, какие методы и какого объекта вы хотите вызвать (что невозможно, если вы пишете библиотеку объектов для использования другими программистами или просто хотите, чтобы ваш объект в разных случаях взаимодействовал с разными внешними объектами).

> ***Примечание***
>
> Как мы уже видели, при использовании сигналов и слотов типы объектов не имеют значения (за исключением того, что все они должны вести свою "родословную" от `QObject`), достаточно наличия у этих объектов методов с определенными именами и списками параметров.

А еще вам придется выполнять громоздкое приведение типов, причем разных типов, в зависимости от того, с какими именно объектами взаимодействует ваш объект. Механизм сигналов и слотов позволяет вам не беспокоиться обо всем этом, как будет показано в одном из примеров далее.

Отложенные сигналы и слоты предоставляют еще более интересные возможности. Если ваш объект эмитирует сигнал, связанный с некоторым слотом, как отложенный, функция-слот не вызывается немедленно. Вместо этого в очередь обработки событий помещается специальный объект, который будет отправлен на обработку, когда подойдет его время. Отложенные сигналы и слоты очень полезны, т. к. с их помощью можно посылать сигналы объектам, принадлежащим другому потоку (мы рассмотрим этот вариант позже). Но их польза заключается не только в этом.

Глядя на автоматически созданные слоты (например, в Qt Creator), имена которых начинаются с `on_`, вы тоже, скорее всего, захотите создавать слоты с такими именами. Но делать этого не следует. Имена слотов, начинающиеся с префикса `on_`, имеют особое значение для системы автоматического связывания сигналов и слотов Qt. Попытка связать такие слоты с сигналами обычным способом приведет к появлению странных сообщений об ошибках.

Исследование сигналов и слотов различных типов

В этом разделе я хотел привести пример показательной ошибки, которую я наблюдал однажды у программиста, работавшего с C++ Builder. Но оказалось, что в Qt создать аналогичную, заведомо ошибочную, ситуацию гораздо сложнее (т. к. Qt

защищает нас от многих ошибок). Все это вылилось в интересное исследование сигналов и слотов Qt, которым я с вами и поделюсь. Суть исходной ошибки заключалась в следующем: предположим, у нас есть класс-плеер (`Player`), который воспроизводит какой-нибудь мультимедийный ролик. У класса `Player` есть метод `setPosition()`, который позволяет установить текущую позицию для воспроизведения. У класса есть также средство, с помощью которого объекты этого класса могут сообщать другим объектам, что позиция воспроизведения изменилась.

Во многих современных мультимедийных программах для индикации текущей позиции и для изменения ее используется один и тот же элемент — ползунок. Когда плеер находится в режиме воспроизведения, ползунок движется сам, указывая текущую позицию воспроизведения. Если же вы хотите изменить ее, то просто устанавливаете этот ползунок в новое положение. То же самое хотел сделать и программист, работавший на Borland C++ Builder. Он связал событие плеера, указывающее на изменение позиции воспроизведения, с методом, устанавливающим положение ползунка. А событие, сигнализирующее об изменении положения ползунка, — с методом `setPosition()` объекта-плеера. Думаю, вы уже догадались, что произошло дальше. Изменение позиции воспроизведения плеера приводило к изменению положения ползунка, в результате чего вызывался метод `setPosition()` плеера. Но метод `setPosition()` снова требовал изменить положение ползунка, что снова приводило к вызову `setPosition()`... И так до тех пор, пока программа не завершалась аварийно. А происходило это очень быстро. Проблема заключалась в том, что программист создал замкнутый круг событий: изменение позиции воспроизведения — изменение положения ползунка, который уходил в бесконечную рекурсию (впрочем, эта бесконечная рекурсия заканчивалась переполнением стека).

Для имитации указанной ошибки я написал класс `Player` (листинг 2.8).

Листинг 2.8. Класс `Player`

```
class QTimer;
class Player : public QObject
{
    Q_OBJECT
public:
    Player();
    ~Player();
public slots:
    void setPosition(int pos);
signals:
    void positionChanged(int pos);
private:
    QTimer * timer;
    int position;
private slots:
    void timeout();
};
```

Этот класс, конечно, не воспроизводит никакого мультимедийного содержимого. Он просто посылает сигналы `positionChanged()` через определенные интервалы времени, используя объект класса `QTimer`. Обратите внимание на макрос `Q_OBJECT`. Этот макрос нужно включать в объявление класса всякий раз, когда мы хотим определить собственные сигналы и слоты. У нашего класса два слота: `setPosition()`, объявленный в разделе `public slots:`, и `timeout()`, объявленный в разделе `private slots:`. Слот `timeout()` нужен для взаимодействия с объектом `timer`, который посылает сигналы `timeout()` через регулярные промежутки времени. Получив сигнал от таймера, слот `timeout()` увеличивает значение переменной `position` и эмитирует сигнал `positionChanged()`. Рассмотрим подробнее слот `setPosition()` (листинг 2.9).

Листинг 2.9. Слот `setPosition()`

```
void Player::setPosition(int pos)
{
    position = pos;
    emit positionChanged(position%100);
}
```

Этот слот устанавливает значение переменной `position` в соответствии с переданным аргументом и эмитирует сигнал `positionChanged()`. Для эмитирования сигнала служит ключевое слово `emit`. Далее сам сигнал вызывается как простой метод. Если в программе сигнал будет связан с одним или несколькими слотами, то можно рассматривать все это так, как будто сигнал является псевдонимом этих методов-слотов в пространстве объекта класса `Player`. Если сигнал не связан ни с какими слотами, то ничего не произойдет, эмитирование сигнала сработает вхолостую.

Далее, в основной программе я попробовал создать замкнутый круг из вызовов между объектом класса `Player` и объектом класса `QSlider` (ползунком) (листинг 2.10). Полный текст программы вы найдете в папке Ch2/circledsignals.

Листинг 2.10. Замкнутый круг

```
connect(ui->horizontalSlider, SIGNAL(valueChanged(int)), player,
        SLOT(setPosition(int)));
connect(player, SIGNAL(positionChanged(int)), ui->horizontalSlider,
        SLOT(setValue(int)));
```

Получился классический замкнутый круг. Сигнал `valueChanged()` объекта `horizontalSlider` вызывает слот `setPosition()` объекта `player`, который в свою очередь эмитирует сигнал `positionChanged()`. Этот сигнал связан со слотом `setValue()` объекта `horizontalSlider`, а вызов `setValue()` приводит к вызову сигнала `valueChanged()`. Согласно моим ожиданиям программа должна была зациклиться и аварийно завершиться, но этого не произошло. Ползунок исправно отображал "позицию воспроизведения" моего псевдоплеера и при этом позволял в любой момент изменить ее. Это чудо произошло потому, что `QSlider`, как и многие другие классы

Qt library, "развязывает" слот `setValue()` и сигнал `valueChanged()`, так что зацикливание не произойдет. Но я не остановился в своих экспериментах, поскольку на тот момент мне не было понятно, выполняется ли развязка замкнутого круга виджетами или системой сигналов и слотов Qt, поэтому я написал свой собственный виджет — `Indicator`, основанный на `QLabel`. У этого виджета также есть слот `setValue()` и сигнал `valueChanged()` (листинг 2.11).

Листинг 2.11. Метод `setValue()` класса `Indicator`

```
void Indicator::setValue(int value)
{
   setText(QString::number(value));
   //if (!QCoreApplication::hasPendingEvents())
   emit valueChanged(value);
}
```

Для этого компонента я тоже создал замкнутый круг (листинг 2.12).

Листинг 2.12. Замкнутый круг, вторая версия

```
connect(ui->indicator, SIGNAL(valueChanged(int)), player,
        SLOT(setPosition(int)));
connect(player, SIGNAL(positionChanged(int)), ui->indicator,
        SLOT(setValue(int)));
```

Далее выяснилось, что если закомментировать строчку, которая закомментирована в листинге 2.11, то происходит то же, что происходило в программе, написанной на Borland C++ Builder. Эту проблему можно решить с помощью отложенных сигналов и слотов (листинг 2.13).

Листинг 2.13. Использование отложенных сигналов для исключения зацикливания

```
connect(ui->indicator, SIGNAL(valueChanged(int)), player,
        SLOT(setPosition(int)));
connect(player, SIGNAL(positionChanged(int)), ui->indicator,
        SLOT(setValue(int)), Qt::QueuedConnection);
        Indicator::setValue(int value)
```

Флаг `Qt::QueuedConnection` указывает на то, что данная связь сигнал–слот является отложенной. Теперь замкнутый круг разорван, и программа работает (рис. 2.2). Однако со временем ее работа замедляется, что, видимо, вызвано накоплением объектов событий в очереди. Для того чтобы избежать накопления объектов, я снял комментарий с той строки, которая закомментирована в листинге 2.11. Но после добавления этой строки программа, как мы уже знаем, работает нормально даже при синхронном связывании сигналов и слотов.

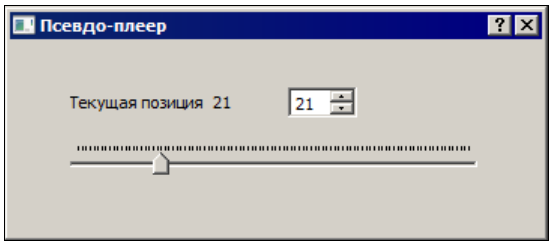


Рис. 2.2. Псевдоплеер, который не зацикливается

Если вы хотите быть уверенными, что ваши сигнал и слот связаны синхронной связью, используйте флаг `Qt::DirectConnection`. По умолчанию последним аргументом метода `connect()` является значение `Qt::AutoConnection`. В этом режиме, если источник сигнала и владелец слота расположены в одном потоке, для них выбирается синхронное соединение `Qt::DirectConnection`. Если слот и сигнал принадлежат разным потокам, выбирается значение `Qt::QueuedConnection`. Среди прочих флагов внимания заслуживает флаг `Qt::UniqueConnection`. Этот флаг работает так же, как и `Qt::AutoConnection`, за исключением того, что если указанные сигнал и слот уже связаны между собой, новое соединение не устанавливается.

Полезные трюки при работе с сигналами и слотами

Если сигнал обладает большим количеством параметров, чем слот, их все равно можно связать между собой. Главное, чтобы начало списка параметров сигнала совпадало со списком параметров слота.

Сигналы можно связывать по цепочке друг с другом, например:

```
connect(ui->pushButton_3, SIGNAL(clicked()), ui->pushButton_2,
        SIGNAL(clicked()));
```

В этом случае щелчок по кнопке `pushButton_3` вызовет сигнал `clicked()` кнопки `pushButton_2`, и, соответственно, тот слот, с которым связан этот сигнал.

Один (единственный, по-моему) недостаток сигналов по сравнению с событиями C++ Builder заключается в том, что мы не всегда можем быть уверены в том, какой объект является источником сигнала. У всех потомков `QObject` есть метод `sender()`, который возвращает (в виде указателя на `QObject`) адрес объекта, пославшего сигнал, который в данный момент обрабатывается. Метод `sender()` возвращает осмысленное значение, только если он вызван внутри слота, иначе метод возвращает 0. То же самое происходит, если слот, в котором вызывается метод `sender()`, вызывается из другого слота. Метод `receivers()`, также объявленный в разделе `protected`, возвращает количество слотов или сигналов, связанных с данным сигналом.

Класс *QSignalMapper* и программа-калькулятор

На самом деле, ситуаций, при которых нам необходимо знать что-либо об источнике сигнала, не так уж и много. Классический пример — группа кнопок, выполняющих сходные действия. Сигнал `clicked()` всех этих кнопок связывают с одним и

тем же слотом, и в этом случае слот должен "знать", какая кнопка его активировала. Для таких случаев разработчики Qt предусмотрели специальный класс `QSignalMapper`. Чтобы понять, что делает `QSignalMapper`, рассмотрим такой пример: нам нужно знать, какая именно кнопка инициировала сигнал `clicked()`. У кнопки `QPushButton` есть два сигнала `clicked()`. Один сигнал вообще не передает никаких параметров, другой сигнал передает параметр типа `bool`. Ни в том, ни в другом случае получить информацию о том, какая именно кнопка была нажата, из параметров сигнала нельзя. Можно, конечно, воспользоваться методом `sender()` в слоте, связанном с сигналом `clicked()`, привести значение, возвращенное методом `sender()`, к типу `QPushButton`, и затем изучить все параметры кнопки-источника сигнала. Но этот метод сработает только в том случае, если наш слот связан именно с кнопкой `QPushButton` или ее потомком. Иначе говоря, мы должны знать, объект какого класса является источником сигналов для нашего слота, а это противоречит идеологии Qt и принципу разделения, о котором мы говорили ранее.

Примечание

Сигналы и слоты могут быть перегружены так же, как и остальные методы.

Класс `QSignalMapper` создан как раз для того, чтобы решить указанную проблему в соответствии с идеологией Qt. Коротко говоря, этот класс позволяет преобразовать сигнал без параметров в сигнал с параметрами, причем значения этих параметров будут зависеть от того, какой объект является источником сигнала. Для демонстрации применения `QSignalMapper` мы напишем программу-калькулятор. Полные исходные тексты программы вы найдете в папке Ch2/calculator/ на сайте **http://symmetrica.net/qt47book/**).

Основу нашей программы составляет класс `Calculator`, который, согласно принципу разделения движка и интерфейса, ничего не знает о том графическом интерфейсе, с помощью которого пользователь будет взаимодействовать с классом (листинг 2.14).

Листинг 2.14. Объявление класса `Calculator`

```
class Calculator : public QObject
{
    Q_OBJECT
public:
    explicit Calculator(QObject *parent = 0);
signals:
    void valueChanged(int value);
public slots:
    void digitButtonPressed(int button);
    void operationButtonPressed(int button);
private:
    int operation;
    Modes mode;
    int bgValue, fgValue;
};
```

Разбираться во внутренней логике работы калькулятора мы не будем. Рассмотрим его интерфейс для связи с внешним миром. У калькулятора два слота и один сигнал. Слот `digitButtonPressed()` информирует калькулятор о том, что была нажата одна из кнопок с цифрами. Число, соответствующее цифре, передается слоту в качестве параметра. Слот `operationButtonPressed()` информирует калькулятор о том, что была нажата одна из кнопок выполнения операции ("плюс", "минус", "равно", "отменить"). Код операции также передается кнопке в качестве параметра. На самом деле кнопки могут быть и не кнопками, а любыми подходящими элементами ввода. Для класса `Calculator` это не имеет значения. С помощью сигнала `valueChanged()` калькулятор информирует графический интерфейс о том, что значение, которое должен отображать этот интерфейс, изменилось. Каким именно образом графический интерфейс отображает результат работы калькулятора, также не имеет значения для класса `Calculator`.

Графический интерфейс для калькулятора представляет собой окно (класс `Dialog`, производный от класса `QDialog`) с кнопками и виджетом `QLCDNumber` для отображения цифр. Все самое интересное происходит в конструкторе класса `Dialog` (листинг 2.15).

Листинг 2.15. Конструктор класса `Dialog`

```
Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
    ui->setupUi(this);
    QSignalMapper * mapper = new QSignalMapper(0);
    connect(ui->pushButton, SIGNAL(clicked()), mapper, SLOT(map()));
    mapper->setMapping(ui->pushButton, digit0);
    connect(ui->pushButton_2, SIGNAL(clicked()), mapper, SLOT(map()));
    mapper->setMapping(ui->pushButton_2, digit1);
    connect(ui->pushButton_3, SIGNAL(clicked()), mapper, SLOT(map()));
    mapper->setMapping(ui->pushButton_3, digit2);
    connect(ui->pushButton_5, SIGNAL(clicked()), mapper, SLOT(map()));
    mapper->setMapping(ui->pushButton_5, digit3);
    connect(ui->pushButton_4, SIGNAL(clicked()), mapper, SLOT(map()));
    mapper->setMapping(ui->pushButton_4, digit4);
    connect(ui->pushButton_6, SIGNAL(clicked()), mapper, SLOT(map()));
    mapper->setMapping(ui->pushButton_6, digit5);
    connect(ui->pushButton_8, SIGNAL(clicked()), mapper, SLOT(map()));
    mapper->setMapping(ui->pushButton_8, digit6);
    connect(ui->pushButton_7, SIGNAL(clicked()), mapper, SLOT(map()));
    mapper->setMapping(ui->pushButton_7, digit7);
    connect(ui->pushButton_9, SIGNAL(clicked()), mapper, SLOT(map()));
    mapper->setMapping(ui->pushButton_9, digit8);
    connect(ui->pushButton_10, SIGNAL(clicked()), mapper, SLOT(map()));
    mapper->setMapping(ui->pushButton_10, digit9);
```

```
    Calculator * calculator = new Calculator(0);
    connect(mapper, SIGNAL(mapped(int)), calculator,
            SLOT(digitButtonPressed(int)));
    connect(calculator, SIGNAL(valueChanged(int)), ui->lcdNumber,
            SLOT(display(int)));
    QSignalMapper * mapper2 = new QSignalMapper(0);
    connect(ui->pushButton_11, SIGNAL(clicked()), mapper2, SLOT(map()));
    mapper2->setMapping(ui->pushButton_11, opPlus);
    connect(ui->pushButton_12, SIGNAL(clicked()), mapper2, SLOT(map()));
    mapper2->setMapping(ui->pushButton_12, opMinus);
    connect(ui->pushButton_13, SIGNAL(clicked()), mapper2, SLOT(map()));
    mapper2->setMapping(ui->pushButton_13, opEqual);
    connect(ui->pushButton_14, SIGNAL(clicked()), mapper2, SLOT(map()));
    mapper2->setMapping(ui->pushButton_14, opCancel);
    connect(mapper2, SIGNAL(mapped(int)), calculator,
            SLOT(operationButtonPressed(int)));
}
```

Сначала мы создаем объект `mapper` класса `QSignalMappper`. Настройка преобразования сигнала кнопки в другой сигнал (с параметром, указывающим, какая именно кнопка является источником сигнала) выполняется в три этапа. Прежде всего, мы связываем сигнал `clicked()` объекта `QPushButton` со слотом `map()` объекта `QSignalMappper`.

Затем мы вызываем метод `setMapping()` объекта `QSignalMappper`. Этому методу передается указатель на объект `QPushButton` и некоторое значение, связанное по смыслу с этим конкретным объектом. Метод `setMapping()` существует в четырех перегруженных вариантах, которые отличаются типом значения, которое мы связываем с объектом. Класс `QSignalMappper` позволяет сопоставить объекту значения типов `int` (то, что мы используем в нашем случае), `QString`, `QWidget *`, `QObject *`.

На заключительном этапе мы связываем сигнал `mapped()` объекта `QSignalMappper` со слотом, который должен активироваться преобразованным сигналом. Причем заголовки сигнала и слота зависят от того, какой вариант метода `setMapping()` был выбран. Если мы использовали метод `setMapping(QObject * sender, int id)`, т. е. дополнительный параметр имеет тип `int`, то сигнал `mapped()` будет передавать значение типа `int` и слот должен принимать значение этого же типа:

```
connect(mapper, SIGNAL(mapped(int)), calculator,
        SLOT(digitButtonPressed(int)));
```

Иначе говоря, если мы связали с помощью `QSignalMappper` сигнал `clicked()` объекта `pushButton` класса `QPushButton` и цифровое значение 1, то при щелчке по кнопке `pushbutton` сигнал `clicked()` преобразуется в сигнал `mapped(int)` с параметром 1.

Примечание

Как объект класса `QSignalMappper` узнает, какой именно параметр должен быть передан в сигнале `mapped()`? Все очень просто. Внутри себя объект `QSignalMappper` хранит адреса объектов, события которых он должен транслировать, и связанные с этими

объектами значения одного из четырех допустимых типов. Сигнал `mapped()` также существует в четырех перегруженных вариантах. Когда один из объектов, связанных с объектом `QSignalMappper`, эмитирует сигнал без параметров, объект `QSignalMappper` эмитирует сигнал с параметром соответствующего типа и соответствующим значением.

Таким образом, любое нажатие кнопки с цифрой на нашем калькуляторе приведет к вызову слота `digitButtonPressed()`, которому в параметре `int` будет передано соответствующее значение. Для вызова слота `operationButtonPressed()` нам потребуется еще один объект класса `QSignalMappper`. Нам остается связать сигнал `valueChanged()` нашего класса-калькулятора со слотом `display()` виджета `QLCDNumber` и программа готова к работе (рис. 2.3). Осталось напомнить только, что в динамическом мире Qt любая связь может быть разорвана. Метод `removeMappings()` позволяет удалить ассоциации, созданные для определенного объекта.

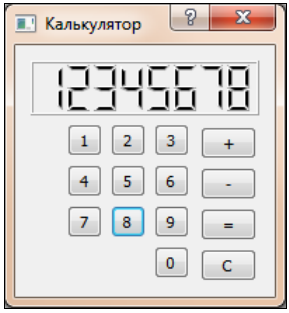


Рис. 2.3. Программа-калькулятор

Примечание

Разделение движка и интерфейса, которые мы продемонстрировали в этом примере, упрощает дальнейшую работу с проектом в еще одном аспекте. Если в будущем мы захотим расширить функциональность нашего калькулятора (например, добавить новые операции), мы сначала отдельно доработаем функциональность движка, а затем и пользовательского интерфейса. Более того, если нам при этом удастся сохранить обратную совместимость интерфейса и движка (например, просто добавить новые константы для передачи методу `operationButtonPressed()`), то мы сможем использовать старый интерфейс с новым движком (при этом новые функции калькулятора, конечно, не будут доступны, но этот вариант программы можно будет распространять как облегченную версию).

ГЛАВА 3

Работа над серьезными проектами

Начиная новый серьезный проект, вы должны быть готовы к тому, что он просуществует (в активном виде) дольше, чем вы предполагаете. Причем вам, или другому разработчику, придется поддерживать, развивать, совершенствовать этот проект в условиях, которые вы не можете описать заранее. В будущем кто-то может попытаться запустить вашу программу на операционной системе, которая еще не написана, пользователи могут потребовать дополнение вашей программы совершенно непредвиденной вами функциональностью, возможно, вашу программу потребуется переносить на совершенно другую платформу, например из-под Windows на Linux (или наоборот). К сожалению, не существует рецепта, который бы мог сделать приложение настолько адаптивным, что все перечисленные задачи можно было бы решить, не написав ни одной строчки дополнительного кода.

> ***Примечание***
>
> С другой стороны, если бы такие решения существовали, рынок труда для программистов заметно сократился бы...

Но вам, как программисту, конечно, хочется свести к минимуму работу, которую придется выполнять при адаптации программы к новым условиям. Разумеется, это не единственный критерий, по которому хороший проект можно отличить от плохого. Такие факторы, как безопасность, тоже играют существенную роль. Но сейчас мы сформулируем один важный принцип, о практической реализации которого и пойдет, в основном, речь в первой части этой главы: хороший проект должен быть задуман так, чтобы его адаптация к новым условиям и расширение функциональности требовали минимальных изменений в существующем коде. Как ни странно, этот принцип оказывается полезен не только для самих разработчиков проекта, но и для пользователей, и для тех, кому придется поддерживать работу приложения. Далее мы перечислим несколько правил, следование которым позволит реализовать грамотный проект.

Правило модульности

Это правило кратко формулируется так: независимые части системы должны быть отделены друг от друга, а зависимые — сгруппированы в модули. Модули должны взаимодействовать друг с другом посредством единообразных (в рамках каждого

модуля) интерфейсов и эти интерфейсы должны полностью скрывать детали реализации модуля.

Разделение независимых частей системы упростит в будущем ее отладку и добавление новых возможностей. Если модули грамотно разделены и взаимодействуют друг с другом по четко определенным правилам, мы можем быть уверены в том, что внесение изменений в один модуль не приведет к непредвиденным изменениям в работе другого модуля. Благодаря этому мы даже можем организовать независимое тестирование модулей (о чем будет подробнее рассказано далее). Кроме того, нам будет проще распределить работу в группе программистов.

Грамотное объединение зависимых частей в один модуль также упрощает отладку и развитие приложения и наглядно показывает, изменение каких частей системы неминуемо приведет к изменению других ее частей. Практический пример: заказчик требует добавить в систему новую функциональную возможность. Если точки, отвечающие за наращивание данного типа функциональности, разбросаны по всему проекту, найти их будет трудно, а забыть внести нужные изменения в одну из них, наоборот, легко. В результате будет потрачено много труда, а доработанная функциональность так и не станет полноценной.

Одно из правил построения модулей можно сформулировать так: группируйте в модуль те компоненты, которые изменяются месте. Если у вас есть несколько фрагментов кода, которые необходимо изменять всякий раз, когда вы вносите в программу определенные изменения (например, адаптируете программу для работы с новой СУБД, новым форматом данных, новой системой ввода/вывода), имеет смысл объединить эти фрагменты в один модуль. В этом случае, если вы начнете вносить изменения в код с определенной целью, то, поскольку весь подлежащий изменению код собран в одном месте, вероятность того, что вы забудете сделать необходимые изменения в одном из фрагментов кода, снизится.

Если модуль пишется с нуля, т. е. если его задача не заключается в том, чтобы инкапсулировать нечто, уже существующее, то многие рекомендуют писать модуль, начиная с внешнего интерфейса, с помощью которого он будет взаимодействовать с другими модулями. Тщательно продуманный внешний интерфейс модуля поможет четко сформулировать набор требований к его содержимому.

Примечание

Некоторые адепты визуального программирования рекомендуют начинать написание программы с интерфейса пользователя. По моему мнению, это неверно. Начинать все-таки надо с реализации модели работы программы.

Правило разделения движка и интерфейса

Это правило, о котором мы уже много говорили и даже демонстрировали его на примерах, можно рассматривать, как особо важный частный случай принципа модульности. Движок программы и интерфейс пользователя, как правило, представляют собой разные модули. Соответственно, их разделение и изоляция друг от друга при помощи четко определенных интерфейсов программирования приносят все те выгоды, которые были описаны выше.

Правило разделения движка и политики

Согласно этому правилу в код программы не должны быть "зашиты" численные параметры и другие характеристики, определяющие параметры работы программы в конкретных условиях (политику). Все, что может быть настроено в зависимости от конкретных условий, должно быть сделано настраиваемым. Существуют два вида настройки: автоматическая, выполняемая самой программой или вспомогательной утилитой, и ручная, выполняемая пользователем или администратором системы.

Иногда приходится слышать возражение, что обилие настроек может напугать пользователя. На это мы отвечаем, что пользователя не обязательно заставлять проходить через все настройки. Наилучшим является решение, при котором у всех настроек есть значения по умолчанию, выбранные наиболее разумным образом либо в процессе разработки программы, либо в процессе ее установки в конкретной системе. При таком подходе пользователю придется изменять только те настройки, значения которых его действительно не устраивают.

Правило разделения движка и политики относится не только к настройкам программы, но и к более сложным элементам. Если пользователь захочет изменить эти элементы, у него должна быть такая возможность (а прибегнуть к ней или нет, будет решать он сам). Примеры разделения движка и политики — это средства настройки форматов отчетов в приложениях баз данных, различные языки написания макросов, позволяющие автоматизировать рутинные действия пользователя, наконец, возможность настройки пользовательского интерфейса программы. Одно из интересных применений принципа разделения движка и политики — наделение программы возможностью работы над одной задачей совместно с другими программами, в том числе с теми, о которых заранее ничего неизвестно. Возможность написания сторонними разработчиками расширений ("плагинов") для вашей программы также можно рассматривать как реализацию этого принципа.

Правило "ноль-один-бесконечность"

Это правило гласит: в общем случае количество неких объектов в вашей программе должно быть либо равно 0, либо равно 1, либо потенциально бесконечно. Под объектами в данном случае подразумеваются такие элементы программы, как дочерние окна, сокеты, потоки и т. п. Некоторые объекты, например, главное окно, должны существовать не более чем в одном экземпляре. Но неразумно устанавливать максимальное количество сокетов, которые может открыть программа, равным, например, 100. Почему 100? Откуда вы знаете, что этого количества будет достаточно? Если лимит на максимальное количество сокетов должен быть установлен заранее, эту возможность следует предоставить тому, кто будет настраивать вашу программу на конкретной платформе (реализовав, таким образом, принцип разделения политики и движка). Наилучшим вариантом является такой: максимальное количество сокетов, которое, в принципе, способна поддерживать ваша программа, должно быть равно максимально допустимому количеству сокетов в системе, под управлением которой она работает.

А что означает "ноль" в названии правила? Ноль означает, что иногда некий элемент вообще лучше вынести за пределы системы (в другой модуль или даже другую программу), нежели пытаться интегрировать его в существующую систему.

Простота переноса на другие платформы

Много интересных проектов "умерло" из-за того, что их не удалось вовремя перенести на новую платформу. В литературе описано довольно много правил построения кросс-платформенных проектов, но не все они одинаково полезны для программистов, работающих с Qt. Дело в том, что сама библиотека Qt представляет собой шедевр кросс-платформенности, так что часто для переноса приложения Qt на новую платформу не требуется делать вообще ничего, кроме повторной сборки. Например, все программы, написанные нами до сих пор, могут собираться и работать и на 32-битной Windows, и на 64-битной Linux (что было проверено), хотя мы совершенно не заботились об этом.

> ***Примечание***
>
> Все примеры программ для этой книги являются кросс-платформенными, если в тексте явно не указано иное.

Рекомендуется, например, выносить код, отвечающий за графический интерфейс пользователя, в отдельные модули, которые пишутся для каждой платформы. Хотя сама идея вынесения кода, отвечающего за интерфейс, в отдельный модуль, является здравой, именно как средство, упрощающее перенос проекта Qt на другую платформу, оно не актуально. Одна из задач Qt заключается именно в том, чтобы предоставить разработчикам единые средства построения графического интерфейса на разных платформах.

Другая рекомендация заключается в том, чтобы вести разработку одновременно на всех целевых платформах, а не портировать приложение с одной платформы на другую. Эта рекомендация выглядит уже лучше, и далее мы рассмотрим принципы организации работы над проектом, при которых несколько человек могут разрабатывать одно приложение на разных платформах, не переписывая один и тот же код. Однако если мы хотим, чтобы наш проект жил долго, мы должны ориентироваться и на те платформы, которые нам в настоящее время недоступны, или даже не существуют еще в виде работоспособных систем. Поэтому, проектируя приложение с перспективой, мы должны исходить из предположения, что портировать его все-таки придется.

Третий совет очень важен. Он заключается в том, что весь код, связанный со спецификой конкретной платформы, должен быть вынесен в отдельные модули. И эти модули должны взаимодействовать с остальными модулями приложения при помощи не зависящих от платформы интерфейсов. Все специфичные для платформы функции и типы данных должны быть инкапсулированы внутри этих модулей. Собственно, этот принцип лежит в основе самой Qt library.

Один из вопросов, на которые также следует обратить внимание, — размер указателя в байтах. За долгие годы программирования для 32-битных систем мы при-

выкли к тому, что длина указателя равна четырем байтам и это может быть заложено где-то в коде, который не выглядит как платформо-зависимый. По-настоящему кросс-платформенный код не должен делать никаких предположений относительно длины указателя, так же как и о порядке байтов в многобайтовых перечислимых типах.

> ***Примечание***
>
> Определение фактической длины указателя на данной платформе — неплохой способ выяснить, с платформой какой разрядности мы имеем дело. Например, в системе управления сборкой CMake существует свойство `CMAKE_SIZEOF_VOID_P`, которое возвращает длину указателя в данной системе в байтах. Это свойство можно использовать для управления сборкой в зависимости от разрядности системы. Qt тоже пользуется этим способом, о чем вы можете узнать из документации. При этом надо учитывать, что длину указателя возвращает используемый компилятор, а не процессор или операционная система. Если вы запустите 32-битный компилятор в 64-битной системе, то получите длину указателя в 32 бита. Впрочем, именно эта длина нам и нужна, т. к. программа пользуется теми указателями, которые создает компилятор, а не предпочтительными указателями системы.

В кросс-платформенном коде также не следует делать никаких предположений о выравнивании элементов структур (принципы этого выравнивания могут зависеть не только от платформы, но и от компилятора).

При работе с текстом кросс-платформенным программам лучше всего использовать кодировку UTF-16, либо, в специальных случаях, Latin-1. Библиотека Qt фактически требует от нас, чтобы мы перевели текст в одну из этих кодировок, прежде чем она сможет работать с ним.

При этом необходимо обратить внимание на проблему строковых констант, используемых в исходном тексте программы. Их кодировка, естественно, совпадает с кодировкой, которую использует редактор исходных текстов. Хорошо, если исходный текст программы набирается в кодировке UTF-8 или UTF-16. Тогда при работе в Windows и Linux со строковыми константами не будет проблем. Но, допустим, мы набираем текст в программе Qt Creator на платформе Windows и не собираемся переводить интерфейс программы на другие языки. Тогда все строковые константы можно сразу писать по-русски. Редактор текста программы в Qt Creator под Windows по умолчанию использует локальную восьмибитную кодировку (CP-1251 в случае русифицированной Windows). В такой ситуации самый удобный способ перевести строковые константы в UTF-16 — использовать статический метод `QString::fromLocal8Bit()`, что я иногда делаю. Этот метод переводит текст в UTF-16 из локальной восьмибитной кодировки системы. Проблема в том, что на платформе Linux (и любой другой системы, отличной от Windows) локальная восьмибитная кодировка будет, скорее всего, совсем другая. Это значит, что если я попробую собрать программу под Linux, то мои строковые константы не конвертируются в UTF-16 правильно. Для решения данной проблемы можно воспользоваться следующим способом: добавьте в функцию `main()` программы, в какое-либо место до вызова метода `exec()` объекта приложения, строку:

```
QTextCodec::setCodecForLocale(QTextCodec::codecForName("Windows-1251"));
```

В текст программы нужно, конечно, включить заголовочный файл `<QTextCodec>`. Приведенная выше строка заставит программу думать, что локальная восьмибитная кодировка — CP-1251. В результате все текстовые константы конвертируются правильно. На ввод данных с помощью виджетов Qt, как и на надписи на виджетах, сделанные в визуальном редакторе, подмена кодировок никак не повлияет, поскольку там локальная восьмибитная кодировка не используется вообще.

Однако лучше всего сразу настроить текстовый редактор на использование более универсальной кодировки, например UTF-8. В Qt Creator это можно сделать в окне **Параметры** (команда меню **Инструменты | Параметры...**). На вкладке **Среда** меняем значение поля **Кодировка файла по умолчанию** с **System** на **UTF-8**. Если вы пишете тексты программ в Microsoft Visual Studio, лучше тоже сохранять их в UTF-8.

> ***Примечание***
>
> Почему стоит предпочесть UTF-8, а не "локальную восьмибитную кодировку", если Qt все равно использует UTF-16? Отвечу так: UTF-8 — это вполне конкретная кодировка, тогда как "локальная восьмибитная кодировка" — это просто ссылка на одну из кодировок, используемых системой. В разных системах эта ссылка будет указывать на разные кодировки.

Существует еще одно решение этой проблемы: перенос всех строковых констант в отдельный файл и присвоение им имен (например, с помощью оператора `const`). Тогда проблема кодировки текста заметно упрощается. В былые времена это также упрощало перевод интерфейса на другие языки (нужно было изменить содержимое только одного файла). Но Qt использует совсем другую модель перевода, в которой строки расположены "по месту". Кроме того, если для построения интерфейса вы используете визуальный редактор, задача вынесения всех строк в один файл еще более усложнится, ведь код, генерируемый редактором автоматически (этот код генерируется повторно всякий раз, когда вы что-то меняете) должен будет ссылаться на ваш файл.

Интроспекция

Когда речь заходит о кросс-платформенности, часто возникает путаница, связанная с двумя разными моментами определения платформы: на этапе сборки и на этапе выполнения программы. Эти два способа преследуют совершенно разные цели. Определение платформы на этапе сборки имеет своей целью выяснить, для какой платформы собирается программа. В результате в программу можно включить код, без которого она просто не будет работать на целевой платформе. Таким образом, у нас появляется новая версия программы, способная работать на целевой платформе. Отметим, что целевая платформа может совпадать, а может и не совпадать (в случае кросс-компиляции) с той платформой, на которой выполняется сборка.

Определение платформы во время выполнения преследует гораздо более узкие цели. Для того чтобы программа могла определить платформу во время выполнения, она должна, как минимум, запуститься на этой платформе. Иначе говоря, определение платформы во время выполнения возможно, когда мы имеем семейство

близкородственных платформ (например, различные версии Microsoft Windows, основанные на Windows NT) и хотим в случае запуска программы на определенной версии ОС использовать специфику этой ОС. В этом случае у нас есть одна версия нашей программы, способная адаптироваться к различным, но родственным, платформам.

Для определения платформы на этапе сборки Qt вводит ряд макросов. Значения этих макросов зависят от версии Qt и (в меньшей степени) от той платформы, на которой она развернута. Макросы определены в заголовочном файле QSysInfo (точнее, во включенном в него файле qglobal.h). Изучив содержимое этого файла, вы узнаете все, что Qt может рассказать программе о системе. Мы рассмотрим самые интересные.

Узнать версию библиотеки Qt можно с помощью `QT_VERSION`. Его значение является числом, например, 0x040700 соответствует версии Qt 4.7.

Выполнение следующих условий можно проверять с помощью директивы `#ifdef`:

- `Q_OS_WIN32` означает, что целевая платформа — 32-битная Windows;
- `Q_OS_WIN64` означает, что целевая платформа — 64-битная Windows. Стоит отметить, что вместе с этим условием определено и условие `Q_OS_WIN32`, вероятно потому, что на 64-битной Windows можно выполнять 32-битный код. В связи с этим рекомендуется сначала проверить, установлено ли условие `Q_OS_WIN64`, и только потом — `Q_OS_WIN32`;
- `Q_OS_WIN` — проверяйте, установлено ли это условие, если вам безразлично, какая именно версия Windows является целевой (32-битная, 64-битная, Windows CE), но важно знать, что это именно Windows;
- `Q_OS_LINUX` означает, что целевая платформа для сборки — Linux;
- `QT_LINUXBASE` означает, что целевая платформа соответствует спецификации Linux Standard Base. Версию спецификации можно узнать с помощью константы `__LSB_VERSION__`;
- `Q_OS_FREEBSD` означает, что целевая платформа для сборки — FreeBSD.

Для Linux и FreeBSD отсутствуют дополнительные константы, позволяющие узнать разрядность процессора. Тут может помочь система CMake и описанный ранее трюк с указателем в файле CMakeLists.txt (листинг 3.1).

Листинг 3.1. Определение размера указателя в файле CMakeLists.txt

```
if (CMAKE_SIZEOF_VOID_P EQUAL 4)
set (NUM_BITS 32)
endif(CMAKE_SIZEOF_VOID_P EQUAL 4)
if (CMAKE_SIZEOF_VOID_P EQUAL 8)
set (NUM_BITS 64)
endif(CMAKE_SIZEOF_VOID_P EQUAL 8)
configure_file(config.h.in config.h)
```

Обратите внимание на аргументы `configure_file()`. Файл config.h.in — это шаблон, на основе которого CMake создаст файл config.h (файл config.h.in останется неизменным). Нам следует создать файл config.h.in, который должен быть расположен в той же директории, что и CMakeLists.txt, и добавить в него строку:

```
#define NUM_BITS ${NUM_BITS}
```

В результате после выполнения команды `cmake ./` у нас появится файл config.h, который будет содержать строку:

```
#define NUM_BITS 32
```

или

```
#define NUM_BITS 64
```

в зависимости от того, что определит CMake.

Другой способ решения той же проблемы — включение в файл config.h константы C++, равной результату `sizeof(void*)`.

Преимущество метода с использованием CMake в том, что эта система может настроить такие аспекты сборки приложений, которые нельзя настроить из файлов исходных текстов программы, например пути поиска разделяемых библиотек для компоновки. Кроме того, если код, предназначенный для другой платформы, заблокирован константой, компилятор все равно попытается скомпилировать его, что может привести к ошибкам. В зависимости от настроек компилятора, этот код даже может остаться в приложении (хотя использоваться и не будет). Код, заблокированный директивой препроцессора, будет недоступен компилятору и в исполнимый файл не попадет.

Для определения параметров системы во время выполнения служит класс `QSysInfo`. Статические методы этого класса возвращают различные сведения о платформе, на которой выполняется программа. Лучше всего возможности этого класса продемонстрирует функция `getOSVersion()` (листинг 3.2).

Листинг 3.2. Функция `getOSVersion()`

```
QString getOSVersion()
{
    static QString osVersion;
    if(osVersion.isEmpty())
    {
#if defined(Q_OS_LINUX)
        utsname buf;
        if(uname(&buf) != -1)
        {
            osVersion.append(buf.release).append(QLatin1Char(' '));
            osVersion.append(buf.sysname).append(QLatin1Char(' '));
            osVersion.append(buf.machine).append(QLatin1Char(' '));
```

```
    osVersion.append(
QLatin1String(" (")).append(buf.machine)
.append(QLatin1Char(')'));
        }
        else
        {
            osVersion = QLatin1String("Linux(unknown)");
        }
...
#elif defined(Q_OS_WIN)
        switch(QSysInfo::WindowsVersion)
        {
            case QSysInfo::WV_CE_6:
                osVersion = QLatin1String("Windows CE 6.x");
                break;
            case QSysInfo::WV_CE_5:
                osVersion = QLatin1String("Windows CE 5.x");
                break;
            case QSysInfo::WV_CENET:
                osVersion = QLatin1String("Windows CE .NET");
                break;
            case QSysInfo::WV_CE:
                osVersion = QLatin1String("Windows CE");
                break;

            case QSysInfo::WV_WINDOWS7:
                osVersion = QLatin1String("Windows 7");
                break;
            case QSysInfo::WV_VISTA:
                osVersion = QLatin1String("Windows Vista");
                break;
            case QSysInfo::WV_2003:
                osVersion = QLatin1String("Windows Server 2003");
                break;
            case QSysInfo::WV_XP:
                osVersion = QLatin1String("Windows XP");
                break;
            case QSysInfo::WV_2000:
                osVersion = QLatin1String("Windows 2000");
                break;
            case QSysInfo::WV_NT:
                osVersion = QLatin1String("Windows NT");
                break;

            case QSysInfo::WV_Me:
                osVersion = QLatin1String("Windows Me");
                break;
```

```
            case QSysInfo::WV_98:
                osVersion = QLatin1String("Windows 98");
                break;
            case QSysInfo::WV_95:
                osVersion = QLatin1String("Windows 95");
                break;
            case QSysInfo::WV_32s:
                osVersion = QLatin1String("Windows 3.1 with Win32s");
                break;

            default:
                osVersion = QLatin1String("Windows(unknown)");
                break;
        }

        if(QSysInfo::WindowsVersion & QSysInfo::WV_CE_based)
            osVersion.append(QLatin1String(" (CE-based)"));
        else if(QSysInfo::WindowsVersion & QSysInfo::WV_NT_based)
            osVersion.append(QLatin1String(" (NT-based)"));
        else if(QSysInfo::WindowsVersion & QSysInfo::WV_DOS_based)
            osVersion.append(QLatin1String(" (MS-DOS-based)"));
#else
        return QLatin1String("Unknown");
#endif
    }
    if (QSysInfo::WordSize == 32)
        osVersion = osVersion + " 32 bit compiler";
        else
        if (QSysInfo::WordSize == 64)
        osVersion = osVersion + " 64 bit compiler";
    if (QSysInfo::ByteOrder == QSysInfo::LittleEndian)
        osVersion = osVersion + " little-endian";
    else
        osVersion = osVersion + " big-endian";
        return osVersion;
}
```

Полную версию функции вы найдете в программе в директории Ch3/getosversion (исходные тексты программ к книге доступны на сайте **http://symmetrica.net/qt47book/**). Функция `getOSVersion()` возвращает строку, описывающую текущую операционную систему и параметры сборки программы (рис. 3.1). На компьютере с ОС Linux программа выводит примерно такую строку:

```
2.6.34.7-0.5-desktop Linux x86_64  (x86_64) 64 bit compiler little-endian
```

где 2.6.34.7-0.5 — версия ядра. Поскольку некоторые параметры платформы, определяемые классом `QSysInfo` во время выполнения, имеют смысл только для опреде-

ленных платформ, функция `getOSVersion()` (и программа, которая ее использует) объединяет оба типа определения платформы: во время компиляции и во время выполнения.

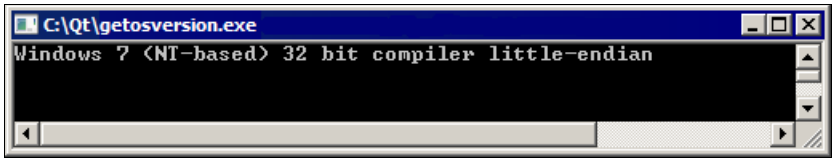


Рис. 3.1. Результат вызова функции `getOSVersion()`

Повторное использование кода

В любом проекте нужно стремиться к тому, чтобы одна и та же задача, независимо от того, где она возникает, решалась одним и тем же фрагментом программного кода. Думаю, не стоит терять время на рассуждения о том, насколько это упрощает отладку кода и внесение доработок.

Применение принципа модульности в Qt

Можно сказать, что библиотека Qt пронизана принципом модульности. Самые разные элементы Qt следуют этому принципу, а упоминание о нем часто встречается в документации. Одним из весьма наглядных примеров использования правила модульности в Qt является система Interview Framework. Система Interview Framework представляет собой вариант реализации парадигмы "модель–контроллер–вид". В основе парадигмы "модель–контроллер–вид" лежит принцип разделения движка программы и интерфейса, который, как мы помним, является частным случаем принципа модульности. В рамках парадигмы "модель–контроллер–вид" модель представляет собой, по сути, "движок" приложения. Именно модель определяет, что и как программа может делать. Термином "вид" (представление) фактически описывается все, что имеет непосредственное отношение к интерфейсу пользователя. Вид позволяет пользователю получать информацию о состоянии модели и передавать программе команды. Команды пользователя обрабатывает контроллер, который вносит соответствующие изменения в состояние модели или вида и, в частности, не позволяет пользователю нарушить целостность модели в результате введения неправильных команд. Как и многие другие парадигмы, призванные формализовать процесс создания программ, парадигма "модель–контроллер–вид" редко применяется на практике в чистом виде. В частности отдельные элементы парадигмы нередко объединяются друг с другом. Interview Framework превращает парадигму "модель–контроллер–вид" в парадигму "модель–вид", объединяя контроллер и вид в одно целое.

Переходя от абстрактного изложения принципов Interview Framework к изложению более конкретному, мы должны познакомиться (простите за каламбур) с тремя абстрактными классами `QAbstractItemModel`, `QAbstractItemView` и `QAbstractItemDelegate`.

Эти классы являются предками всех классов, реализующих, соответственно, модели, представления (виды) и делегаты. Классы `QTableView`, `QTreeView` и `QListView` реализуют три наиболее популярные формы представления данных: таблицу, дерево и простой список. Если вам требуется более сложный компонент отображения модели, вам придется создавать собственный класс, основанный на `QAbstractItemView`.

Класс `QStandardItemModel`, являющийся потомком `QAbstractItemModel`, представляет собой реализацию модели в самом общем смысле. Помимо прочего, этот класс реализует ряд методов, предназначенных для работы с индексами. Индексы используются в Interview Framework для указания элементов данных, с которыми работает модель. Класс `QDirModel` реализует модель для работы с директориями. Этот класс пригодится вам, если вы надумаете писать собственный файловый менеджер или свою версию диалоговых окон открытия и сохранения файла. Следует отметить, что один и тот же объект, реализующий модель, может взаимодействовать (в том числе, одновременно) с объектами нескольких разных классов, отвечающими за представление данных. Например, уже упомянутый класс `QDirModel` может использовать для представления информации о директориях классы `QTableView`, `QTreeView`, и `QListView`. Класс `QStringListModel`, как можно догадаться по его названию, реализует модель, основой которой является список строк.

Классы `QAbstractTableModel` и `QAbstractListModel` могут служить основой для ваших собственных классов-моделей, предполагающих представление данных в виде таблиц и списков соответственно.

Такие классы как `QTreeView` и `QListView` предназначены для работы с моделями, но использовать их в качестве самостоятельных виджетов затруднительно. Для решения этой проблемы на базе классов `QTableView`, `QTreeView` и `QListView` созданы классы `QTableWidget`, `QTreeWidget` и `QListWidget`. Объекты этих классов представляют собой обычные визуальные компоненты, при работе с которыми пользователь может добавлять и удалять данные, не заботясь о моделях и делегатах. На самом деле, эти классы просто реализуют свои собственные модели данных, изолированные от программиста. Как видим, система Interview Framework играет в Qt бóльшую роль, чем может показаться на первый взгляд!

Хотя парадигма "модель–контроллер–вид" (а, следовательно, и Interview Framework) может применяться при написании программ самых разных типов, разработчики Interview Framework, судя по всему, ориентировались в основном на создание клиентских приложений для работы с базами данных. Именно на примере клиентского приложения БД проще всего понять, как работает Interview Framework. В клиентском приложении БД, использующем Interview Framework, модель играет роль посредника между БД и интерфейсом пользователя, представляющим данные БД. Именно модель определяет логику представления данных. Когда пользователь хочет получить информацию о текущем состоянии БД, пользовательский интерфейс (компонент "вид") обращается к модели напрямую. Для работы с отдельными элементами данных служат делегаты, которые передают команды пользователя модели.

Принцип модульности и простое приложение баз данных

Для демонстрации принципа модульности мы рассмотрим простое приложение, работающее с локальной базой данных. Наша программа — музыкальный каталог, который использует для хранения данных движок SQLite (рис. 3.2). Исходные тексты программы вы найдете в папке Ch3/musicdatabase.

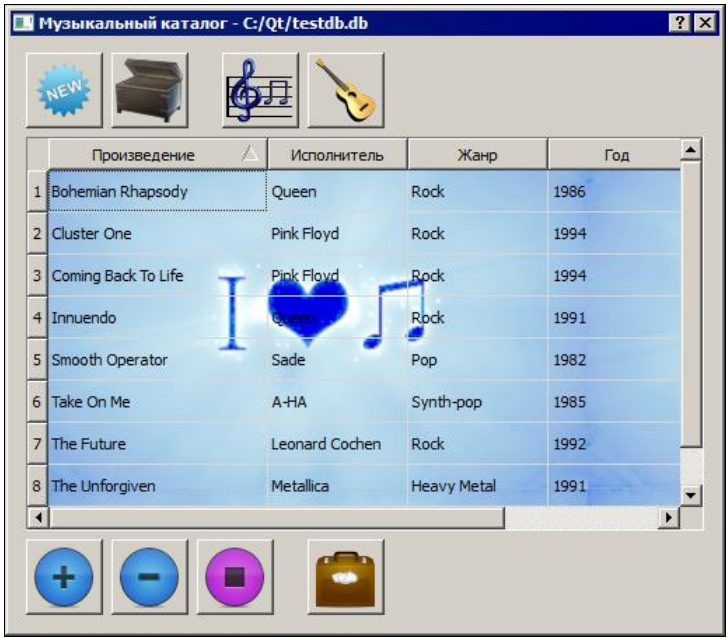


Рис. 3.2. Музыкальный каталог

SQLite (**www.sqlite.org**) — это простая (наверное, самая простая) система управления базами данных, использующая язык SQL (в несколько упрощенной форме). Исходные тексты SQLite являются общественным достоянием (public domain). Система SQLite не годится для хранения больших объемов данных и ее нельзя использовать в качестве сервера, но она очень хорошо подходит для написания демонстрационных программ, работающих с базами данных и для отладки серьезных приложений БД. Одно из преимуществ SQLite заключается в том, что эта система не нуждается в какой-либо специальной установке или настройке. Каждая база данных SQLite может храниться в отдельном файле и этот файл содержит все настройки, необходимые для работы с конкретной БД. База данных для нашего примера хранится в файле testdb.db, который вы найдете в папке исходных текстов. SQLite входит в состав дистрибутивов Qt и все, что требуется сделать для подключения этой системы к нашей программе, — включить в файл проекта соответствующие модули.

Поскольку Qt использует единый механизм доступа к различным СУБД, приложения, разработанные сначала для SQlite, не так уж трудно адаптировать к работе

с более мощной системой баз данных. В этой главе мы не будем вникать в детали работы Qt с базами данных, а обсудим общие вопросы, возникающие при проектировании приложений, работающих с БД.

Наша база данных музыкального каталога имеет довольно простую структуру (рис. 3.3).

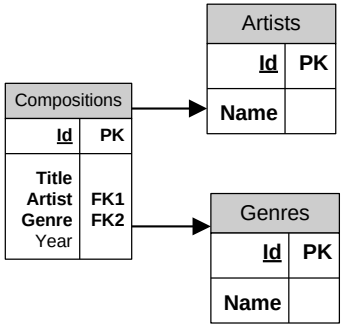


Рис. 3.3. Схема базы данных музыкального каталога

Примечание

Проектирование приложений баз данных всегда лучше начинать с проектирования самой базы данных. Требования, предъявляемые к базе данных, исходят из информационной модели данных, а уже модель базы данных в значительной мере формирует требования, предъявляемые к приложению. Разумеется, эти требования, так же как и модель базы данных, могут меняться со временем. Это значит, что мы должны написать такую программу, которая могла бы легко адаптироваться к меняющимся требованиям.

Центральную роль в нашей базе данных играет таблица `Compositions`, которая содержит информацию о музыкальных произведениях. Помимо первичного ключа, у таблицы `Compositions` есть поля `Title` (название произведения), `Artist` (исполнитель), `Genre` (жанр) и `Year` (год). Поля `Artist` и `Genre` являются внешними ключами, которые ссылаются на таблицы `Artists` и `Genres` (списки исполнителей и жанров соответственно).

Примечание

Если вы не знакомы с такими понятиями как "внешний ключ", "первичный ключ" и не владеете языком SQL, я рекомендую вам обратиться к специальной литературе, например к книге [3].

От модели базы данных перейдем к приложению, которое будет работать с этой БД. Наше приложение содержит объекты, которые отражают модель данных и позволяют манипулировать данными в соответствии с требованиями пользователя. Кроме того, приложение, естественно, включает в себя интерфейс пользователя. В соответствии с принципом модульности сходные компоненты приложения должны быть объединены, а различные — разделены. Посмотрим, как это можно реализовать на практике, и какая от этого может быть польза.

Центральную роль в построении модели данных в приложении играет класс `Table` (рис. 3.4). Этот класс можно рассматривать как представителя таблицы базы дан-

ных в нашем приложении. Класс `Table` позволяет программно манипулировать данными, добавлять к таблице ссылки на другие таблицы, синхронизировать данные, хранящиеся в объекте класса с реальными таблицами БД, которые эти объекты представляют, и многое другое, но в нем отсутствуют методы для взаимодействия с визуальными элементами ввода/вывода содержимого таблицы.

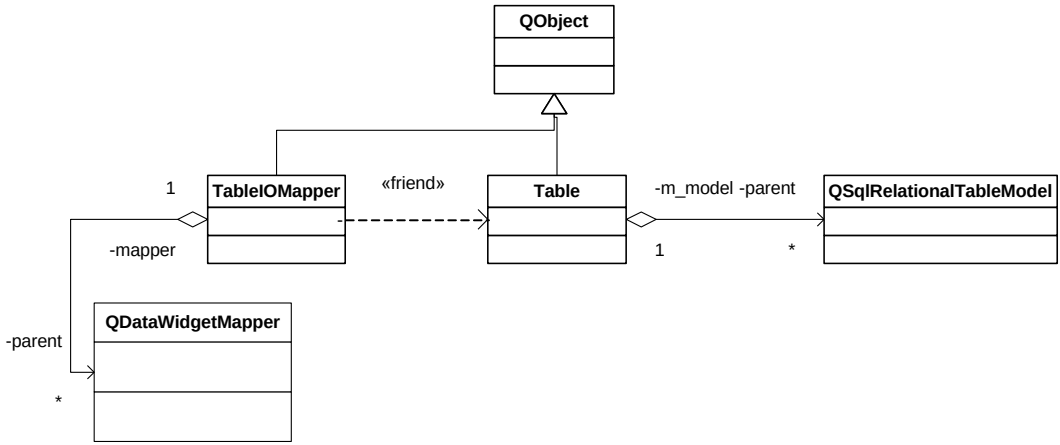


Рис. 3.4. Диаграмма классов, реализующих модель данных приложения

Примечание

Для описания отношений и взаимодействий между классами я пользуюсь нотацией UML 2.0, приведенной в книге [1]. Представленные диаграммы UML не очень подробны, зато они позволяют сразу увидеть главные идеи, заложенные мной в проект приложения.

Взаимодействие с объектами ввода/вывода берет на себя класс `TableIOMapper`, который является "другом" класса `Table`. Обратите внимание на то, что класс `Table` агрегирует объект класса `QSqlRelationalTableModel`, а класс `TableIOMapper` агрегирует объект класса `QDataWidgetMapper`. Таким образом, элементы системы Interview Framework "завернуты" в классы `Table` и `TableIOMapper`. В результате классы `Table` и `TableIOMapper` реализуют шаблон проектирования (паттерн) "адаптер". О том, почему мне понадобились обертки для классов Interview Framework, и почему используется агрегация, а не наследование, мы подробно поговорим в главах, посвященных Interview Framework. Сейчас замечу только, что агрегация позволяет сравнительно легко модифицировать как саму обертку, так и ее "содержимое", т. е. при необходимости заменить одни классы Interview Framework на другие.

Объекты класса `Table` представляют собой внутренний движок нашей программы. Как уже говорилось, взаимодействие этого движка с элементами пользовательского интерфейса возложено на объекты класса `TableIOMapper` (рис. 3.5). Объекты пользовательского интерфейса приложений баз данных Qt можно разделить на две категории: объекты, предназначенные для отображения и ввода самих данных (например, объекты класса `QTableView`) и "обычные" объекты, такие как кнопка или строка ввода. Классы объектов первой категории сами являются частью Qt Interview

Framework, и их довольно трудно изолировать от других частей этой системы. Вероятно, делать этого вообще не следует. Что же касается объектов второй категории, то они могут (и должны) быть изолированы от движка программы.

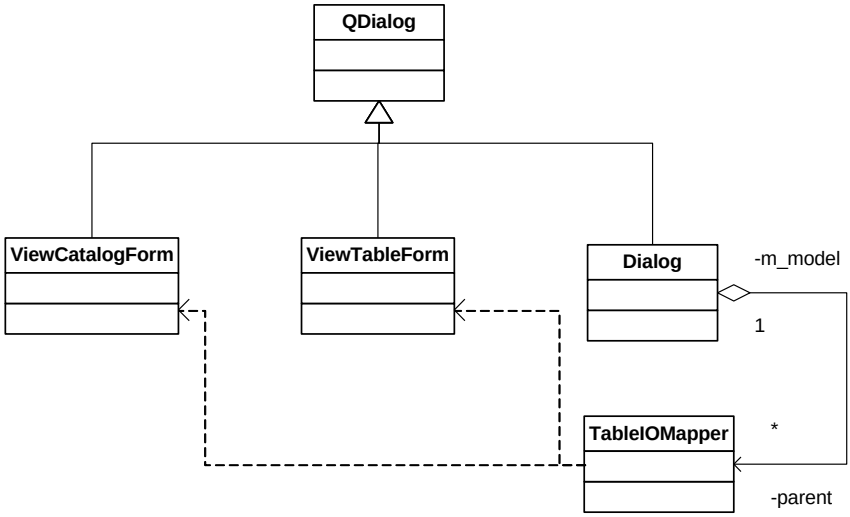


Рис. 3.5. Диаграмма классов, реализующих пользовательский интерфейс приложения

Представленные диаграммы UML не отражают многие детали реализации, но эти детали нам пока и не нужны, мы ведь исследуем структуру приложения в общем и целом. Создание объектов модели данных и подключение приложения к базе данных выполняется в классе `Dialog`, который не следует путать с классом `Ui::Dialog`. Инициализацию объектов и другие операции можно было бы представить схемой взаимодействия UML, но, я думаю, исходный текст будет более нагляден (листинг 3.3).

Листинг 3.3. Создание программной модели данных

```
void Dialog::deleteModel()
{
    delete artistsMapper;
    delete genresMapper;
    delete compositionsMapper;
    delete artists;
    delete genres;
    delete compositions;
}

void Dialog::createModel()
{
    deleteModel();
    artists = new Table(0, "Artists");
```

```
    artists->openTable();
    artistsMapper = new TableIOMapper(artists);
    genres = new Table(0, "Genres");
    genres->openTable();
    genresMapper = new TableIOMapper(genres);
    compositions = new Table(0, "Compositions");
    compositions->addRelation("Artist", "Artists", "id", "Name");
    compositions->addRelation("Genre", "Genres", "id", "Name");
    compositionsMapper = new TableIOMapper(compositions);
}
void Dialog::setupView()
{
    genresForm->setMapper(genresMapper);
    artistsForm->setMapper(artistsMapper);
    compositionsMapper->setView(ui->tableView);
    compositionsMapper->refresh();
    compositionsMapper->setColumnName(0,
       QString::fromUtf8("Произведение"));
    compositionsMapper->setColumnName(1,
       QString::fromUtf8("Исполнитель"));
    compositionsMapper->setColumnName(2, QString::fromUtf8("Жанр"));
    compositionsMapper->setColumnName(3, QString::fromUtf8("Год"));
    ui->tableView->hideColumn(4);
}
```

Метод `createModel()` класса `Dialog` создает модель данных, используя объекты классов `Table` и `TableIOMapper`. Объект `compositions` отвечает за представление в программе таблицы `Compositions`. Обратите внимание на метод `addRelation()`. Этот метод позволяет воспроизвести в модели данных, которую мы создаем в программе, реляционную структуру базы данных. Строка

```
compositions->addRelation("Artist", "Artists", "id", "Name");
```

сообщает объекту `compositions`, что поле `Artist` таблицы `Compositions` является внешним ключом и ссылается на поле `id` таблицы `Artists`. Но это не все. Когда мы хотим наглядно отобразить данные, содержащиеся в таблице, которая ссылается на другие таблицы, мы хотим, чтобы вместо численных ссылок в соответствующих столбцах таблицы были представлены фактические данные. На уровне SQL мы можем решить эту проблему с помощью представления. Например, для таблицы `Artists` это представление может быть создано так:

```
CREATE VIEW Compositions_View AS SELECT Compositions.Title,
    Artists.Name AS Artist,
    Genres.Name AS Genre, Compositions.Year FROM Compositions, Artists,
    Genres WHERE Compositions.Artist = Artists.id AND
                                       Compositions.Genre = Genres.id
```

Для той же цели можно применять конструкции типа `INNER JOIN`. Однако при работе с Qt Interview Framework нам не обязательно создавать такие представления,

чтобы показать пользователю содержимое таблицы `Compositions` в осмысленном виде. Обратите внимание на последний аргумент метода `addRelation()`. Он сообщает, каким столбцом внешней таблицы нужно заменить столбец таблицы `Artists`, который ссылается на эту таблицу. В результате объект `compositions` будет содержать данные в виде представления `Compositions_View`, хотя самого представления в базе данных не существует. При этом объект `compositions` по-прежнему сможет обновлять данные таблицы `Compositions` базы данных, чего представление `Compositions_View` делать не могло бы без определения дополнительных элементов (триггеров). Все эти удобства предоставляет нам объект класса `QSqlRelationalTableModel`, который содержится внутри объекта `Table`.

> ***Примечание***
>
> А почему мы вообще должны указывать объектам `Table`, какие отношения существуют между таблицами, которые эти объекты представляют в программе? Нельзя ли сделать так, чтобы объекты `Table` сами извлекали эту информацию из базы данных? Для многих СУБД это действительно можно сделать. Однако не существует универсального для всех СУБД способа определить, в каких отношениях находится одна таблица с другими таблицами. Метод `addRelation()` предназначен для воспроизведения структуры БД в приложении, но в принципе он не зависит от структуры базы данных. С помощью этого метода в программной модели данных можно устанавливать отношения, которые не существуют в модели БД. Иногда это бывает полезно.

Класс `QSqlRelationalTableModel` предъявляет к таблицам базы данных, участвующим в реляционных отношениях, два важных требования:

- у всех таблиц должны быть явно определены первичные ключи;
- столбцы, являющиеся внешними ключами, не должны содержать значений `NULL`.

Второе требование относится не только к внешним ключам, определенным в самой БД, но и к псевдоключам, которые мы создаем, когда используем метод `addRelation()` для генерации отношений, которые отсутствуют в БД. Нарушение второго требования приведет к тому, что строка просто не попадет в результирующую таблицу. Объясняется это тем, что внутреннее представление, которое создает `QSqlRelationalTableModel`, строится с помощью той же конструкции SQL, которую я использовал ранее при демонстрации создания представления `Compositions_View`. Читатели, знакомые с SQL, поймут, что при таком подходе поля `Compositions.Artist` и `Compositions.Genre` не должны содержать значений `NULL` (впрочем, этих значений стоит избегать и по многим другим причинам).

Хотя в программе существуют объекты `artists` и `genres`, которые представляют соответствующие таблицы, для моделирования отношений между таблицами класса `Compositions` (который содержит свои представления данных из всех необходимых таблиц), `Artists` и `Genres` они не нужны. Отношения полностью существуют внутри объекта `compositions`, а объекты `artists` и `genres` требуются для того, чтобы пользователи могли редактировать соответствующие таблицы. В результате возникает некоторый разнобой. Объект `compositions` хранит данные из таблиц `Artists` и `Genres`, которые могут не совпадать с данными, хранимыми в объектах `artists` и `genres`. И все это может не совпадать с теми данными, которые хранятся в "настоя-

щих" таблицах `Artists` и `Genres` в базе данных. Эта проблема не имеет решения в рамках Interview Framework, и в будущем мы найдем для нее свое решение. Пока что я только хочу отметить этот факт: данные, которые пользователь ввел в каталоги исполнителей и жанров, могут быть не сразу доступны для редактирования каталога произведений.

Из всего вышесказанного следует, что модель данных, которую мы создаем в программе с помощью объектов класса `Table`, хотя и отражает модель данных базы данных, но во многом независима от нее, и даже может дополнять модель данных БД собственными элементами. Эта модель также независима от визуальных элементов программы, за исключением тех элементов, которые непосредственно предназначены для отображения и ввода данных таблиц.

Метод `setupView()` связывает объекты модели с визуальными элементами пользовательского интерфейса. Диалоговые окна `artistsForm` и `genresForm` (предназначенные для редактирования списков исполнителей и жанров соответственно) получают указатели на объекты класса `TableIOMapper` и дальше настраивают взаимодействие с интерфейсом пользователя так же, как это делается для объекта `compositionsMapper`, поэтому мы рассмотрим только настройку `compositionsMapper`. Метод `setView()` позволяет передать объекту класса `TableIOMapper` указатель на объект, отвечающий за графическое представление данных в рамках системы Interview Framework. Как было сказано выше, эти объекты принадлежат классам, происходящим от `QAbstractItemView`, поэтому аргумент метода `setView()` имеет тип `QAbstractItemView *`.

В рассматриваемом примере мы используем объекты класса `QTableView`. Класс `QTableView` представляет данные в виде таблицы (что, как вы понимаете, очень хорошо подходит для наших целей) и дает нам широкие возможности для определения внешнего вида таблицы. Кроме того, этот класс располагает всем необходимым для редактирования значений в уже существующих ячейках таблицы. Как вы помните, содержимое объекта `compositions` фактически является представлением, сконструированным из нескольких таблиц. Если пользователь захочет отредактировать в объекте `QTableView` содержимое ячейки, которая содержит ссылку на внешнюю таблицу (т. е. является внешним ключом), будет открыт выпадающий список. Причем список будет содержать не значения идентификатора, на которые ссылается внешний ключ, а те значения, которыми мы их заменяем. Например, если пользователь нашей программы щелкнет мышью в столбце **Исполнитель**, то будет открыт список с именами исполнителей, если пользователь щелкнет мышью в столбце **Жанр**, то будет открыт список с названиями жанров.

Обратите внимание, что эта возможность досталась нам "бесплатно" — мы ничего не делали, чтобы настроить объект `QTableView` таким образом. Объекты `QTableView` сами извлекают информацию о типе столбцов той таблицы, с которой они работают и при попытке редактирования содержимого столбца (если редактирование разрешено) сами создают наиболее подходящий виджет для редактирования.

Если столбец содержит строковые значения, для редактирования будет создан виджет "строка ввода", если целые числа, то для редактирования значений нам будет предложен наборный счетчик `QSpinBox`. Если ячейка содержит числа с плавающей

точкой, будет создан другой вариант наборного счетчика — QDoubleSpinBox. Разумеется, у QTableView нет встроенных редакторов для всех возможных типов данных. Например, если поле таблицы содержит двоичные данные, QTableView по умолчанию не сможет адекватно их обработать. Возможно также, что виджеты-редакторы, создаваемые по умолчанию, вас не устроят. В этом случае вы можете заменить их на другие. О том, как это сделать, будет рассказано в следующих главах. Две важные операции редактирования данных, которые QTableView выполнять не умеет, — это добавление и удаление строк. Эти операции должны выполняться на уровне модели данных, а QTableView только отобразит внесенные изменения.

Метод deleteModel() удаляет существующую модель данных. Это необходимо, когда мы хотим заменить ее на новую.

Интеллектуальные указатели

Вы, возможно, обратили внимание, что в методе deleteModel() я совершенно не забочусь о содержимом указателей, для которых вызывается оператор delete. Я могу себе это позволить потому, что использую в качестве указателей на объекты моделей объекты классов на основе шаблона QPointer. QPointer — простейший из интеллектуальных указателей (smart pointers), используемых в Qt. Для получения из него класса ему необходимо сообщить тип данных, на который (простите за тавтологию) указывает указатель. В результате у нас появляется класс, который эмулирует поведение указателя на некоторый тип данных. Например, конструкция QPointer<Table> создает интеллектуальный указатель на объект класса Table.

Такой класс совместим по присваиванию с обычными указателями (Table *) и поддерживает ряд операций над указателями, например операцию разыменования указателя (оператор *). Но кроме этого, интеллектуальные указатели обладают различными функциями, которых нет у обычных указателей: счетчиком ссылок, или, например, возможностью выполнять атомарные операции. Библиотека Qt library активно использует интеллектуальные указатели в своем коде, но на уровне внешних интерфейсов предпочитает обычные. Все-таки интеллектуальные указатели сложнее в обслуживании, нежели обычные, а некоторые аспекты их использования могут сбить неподготовленного программиста с толку.

Самое интересное в интеллектуальных указателях — их дуализм. Когда мы обращаемся к переменной типа TablePtr, используя оператор разыменовывания (явно, или неявно, в виде оператора ->), мы получаем доступ к объекту Table, как и должно быть с указателем. Но если мы обращаемся к переменной типа TablePtr, используя точку, то получаем доступ к методам объекта класса, реализующего указатель (в нашем примере — QPointer<Table>). Когда же мы обращаемся к значению самой переменной типа TablePtr, например, в операциях сравнения или присваивания, мы, опять-таки, имеем дело с указателем на объект Table. Описанный дуализм достигается достаточно просто — путем перегрузки операторов, которые используются для работы с указателями.

Зачем нам нужен указатель QPointer? Согласно документации главное достоинство этого интеллектуального указателя в том, что мы всегда можем узнать, действи-

тельно ли он указывает на объект соответствующего типа (аргументом шаблона может быть только объект класса `QObject` или его потомка). Если где-то в программе объект был удален (или указатель так и не был инициализирован), и содержащийся в указателе адрес не соответствует никакому объекту, метод `isNull()` класса, основанного на `QPointer`, сообщит нам об этом (если указатель `QPointer` не содержит корректный адрес объекта, он возвращает значение 0). Документация рекомендует использовать указатели `QPointer` в том случае, когда ваш класс получает указатель на некий объект от другого класса. Если в какой-то момент другой класс удалит объект, на который ссылался указатель, ваш `QPointer` сможет сообщить вам об этом. По моему мнению, такое использование указателя `QPointer` провоцирует неряшливый стиль программирования. Если ваши классы обмениваются указателями, жизненный цикл которых не подчиняется определенным правилам, вам лучше пересмотреть структуру своей программы.

Однако указатель `QPointer` может быть полезен в другой ситуации. В программе musicdatabase объекты модели данных создаются тогда, когда пользователь создает или открывает базу данных. При выполнении операций с указателями на объекты модели данных необходимо знать, созданы ли уже эти объекты. Мы могли бы присвоить в конструкторе `Dialog` значения `NULL` всем этим указателям и затем проверять, сохранили ли они эти значения. Указатели `QPointer` делают это за нас и тем самым экономят нам несколько строк кода. Еще одно удобство, которое предоставляет нам указатель `QPointer`, — безопасная операция удаления объекта. Поскольку указатель становится равным нулю при удалении объекта, на который он указывает, а вызов оператора `delete` для нулевого указателя не имеет последствий, мы можем вызывать `delete`, не беспокоясь о значении указателя. Если значением указателя является адрес объекта, объект будет удален. Если нет — ничего не случится.

Еще один интересный интеллектуальный указатель — `QScopedPointer`, появившийся в Qt 4.6. Особенность этого указателя заключается в том, что он уничтожает (по умолчанию — с помощью оператора `delete`) объект, на который указывает, при выходе указателя из области видимости, в том числе если это произошло в результате исключения. Ту же задачу можно решить с помощью класса-помощника, локальный объект которого создается внутри функции и в деструкторе которого можно выполнить все действия, необходимые при выходе локального объекта из области видимости, но шаблон `QScopedPointer` упрощает решение этой задачи. Поскольку по умолчанию `QScopedPointer` вызывает для удаления объекта оператор `delete`, по умолчанию же он должен указывать на объект, созданный с помощью оператора `new`. Однако это поведение можно изменить. Указатель `QScopedPointer` позволяет использовать при работе с Qt некоторые приемы и шаблоны программирования, которые обычно применяются в системах разработки с автоматической сборкой мусора. Иначе говоря, при использовании указателя `QScopedPointer` вам не придется заботиться о постановке операторов `delete` или функций, освобождающих память, везде, где указатель может стать недоступным.

Вторым параметром шаблона может быть класс-"удалитель", который и выполняет фактическое удаление объекта, на который ссылается указатель. Qt 4.7 предостав-

ляет нам несколько таких классов. Помимо `QScopedPointerDeleter`, который используется по умолчанию, мы можем задействовать `QScopedPointerArrayDeleter`, предназначенный для удаления массивов, созданных с помощью оператора `new` (для той же цели существует указатель `QScopedArrayPointer`, который специально предназначен для работы с массивами, созданными с помощью `new`). Еще один вспомогательный класс, `QScopedPointerPodDeleter`, вызывает для удаления объекта, на который ссылается указатель, функцию `free()`. Иными словами, с его помощью можно удалить область памяти, выделенную с помощью `malloc()`. Если возможностей классов, выполняющих удаление, которые предоставлены библиотекой Qt, вам не хватает, можете написать свой собственный класс.

Есть несколько ситуаций, в которых вам может понадобиться собственный вспомогательный объект-"удалитель". Во-первых, такая необходимость возникает, когда вы пользуетесь собственным менеджером памяти, и соответственно, собственной функцией для удаления объектов. Вторая ситуация — использование разделяемой библиотеки, которая создает и удаляет некие непрозрачные для вас объекты, а затем требует уничтожать их с помощью специальной функции.

Примером такой библиотеки может служить библиотека mpg123, предназначенная для работы с различными файлами формата MPEG, в том числе MP3. В начале работы с mpg123 вы должны получить указатель на структуру `mpg123_handle`. Сама структура непрозрачна для вас, иначе говоря, у вас в программе не должно быть ее определения.

> ***Примечание***
>
> Поскольку исходные тексты mpg123 доступны всем желающим, вы можете включить определение структуры `mpg123_handle` из исходных текстов mpg123 в текст своей программы. Но делать этого не следует. Разработчики библиотеки предполагают, что строение структуры `mpg123_handle` является "внутренним делом" библиотеки, и могут в любой момент изменить его. Если вы не хотите, чтобы ваша программа внезапно перестала работать при обновлении версий разделяемых библиотек, не пытайтесь напрямую использовать структуры, помеченные как внутренние.

Экземпляр `mpg123_handle` создается так:

```
mpg123_handle * mpg123_new (const char *decoder, int *error)
```

Указатель, возвращенный этой функцией, затем передается в качестве первого параметра другим функциям mpg123, так что вам совершенно не обязательно знать, на что именно он указывает. Поскольку внутреннее строение типа данных `mpg123_handle` вам неизвестно, вы, в принципе, можете рассматривать указатель, который возвращает функция `mpg123_new()`, как `void *`. После того как работа с экземпляром типа `mpg123_handle` закончена, его следует удалить так:

```
mpg123_delete (mpg123_handle *mh)
```

Поскольку вы не знаете, как создается экземпляр `mpg123_handle`, то и пытаться удалить его с помощью какой-либо другой функции просто опасно. Так что если вы хотите, чтобы жизненным циклом переменной типа `mpg123_handle` управлял интеллектуальный указатель `QScopedPointer`, вам придется создать собственный класс для удаления экземпляра (листинг 3.4).

Листинг 3.4. Использование указателя `QScopedPointer`

```
struct Mpg123HandleDeleter
{
    static inline void cleanup(mpg123_handle * handle)
    {
         mpg123_delete(handle)
    }
};
...
int error
QScopedPointer<mpg123_handle, Mpg123HandleDeleter>
mpg123Handle(mpg123_new(NULL, &error));
```

Переменная `mpg123Handle` будет содержать интеллектуальный указатель на структуру `mpg123_handle`. В первом параметре функции `mpg123_new()` можно указать имя кодека, а второй параметр вернет информацию об ошибке, если таковая произойдет, но все это относится к особенностям интерфейса mpg123, который мы здесь не рассматриваем. Для нас важно, что переменную `mpg123Handle` можно использовать везде, где можно было бы поставить переменную типа `mpg123_handle *`, и что при выходе переменной `mpg123Handle` из области видимости будет вызвана функция `cleanup()` структуры `mpg123HandleDeleter`.

Функция `cleanup()` должна быть объявлена как публичная (все функции структур, в отличие от методов классов, по умолчанию считаются публичными), статическая и встраиваемая функция структуры или класса и не должна возвращать значение. Единственным аргументом этой функции должен быть простой указатель на тот объект, на который указывает переменная типа `QScopedPointer`.

Интеллектуальные указатели – "за и против"

Программисты, впервые познакомившиеся с концепцией интеллектуальных указателей, часто совершают одну и ту же ошибку. Они пытаются создать интеллектуальный указатель с безопасной операцией разыменовывания (иначе говоря, указатель, разыменовывание которого не приводит к ошибкам, даже если указатель не был инициализирован). Создать такой указатель можно, если при объявлении переменной соответствующего типа присвоить ему адрес некоего "объекта по умолчанию". Поскольку фактически такой указатель не является нулевым, операция разыменовывания не приведет к ошибке. Аргумент против такого рода интеллектуальных указателей сводится к тому, что, во-первых, "объект по умолчанию", как правило, не имеет смысла, а во-вторых (и это самое главное), безопасное разыменовывание скрывает от программиста его собственные ошибки. Если программа пытается получить значение указателя, который программист забыл инициализировать, значит, программист допустил ошибку и программа должна сообщить ему об этом, сгенерировав исключение. Наличие самой ошибки не зависит от того, указатель какого типа мы используем, но безопасное разыменовывание затруднит обнаружение этой ошибки.

Между прочим, подобный аргумент можно применить и против указателя QPointer. Этот указатель не скрывает от нас ошибку, возникающую при попытке разыменовывания неинициализированного указателя. Такая попытка будет равносильна разыменовыванию нулевого указателя, что приведет к генерации исключения. Однако указатель QPointer может скрыть от нас другую ошибку — ошибку, связанную с многократным удалением одного и того же объекта. Поскольку операция delete с этим указателем безопасна, мы никогда не узнаем, что удаляем уже удаленный объект. Можно возразить, что повторное удаление объекта — не такая страшная ошибка, как попытка разыменовывания неинициализированного указателя, но сам факт повторного удаления может свидетельствовать об изъянах в структуре программы, о которых программисту следует знать. Любопытно, однако, что интеллектуальный указатель не позволит инициализировать себя "мусором". Следующий фрагмент кода (листинг 3.5) приведет к генерации исключения, чего не случилось бы, если бы переменная obj была обычным указателем.

Листинг 3.5. Попытка инициализации интеллектуального указателя "мусором"

```
typedef QPointer<QObject> ObjPtr;
...
ObjPtr obj
QObject * f = (QObject *) 0xbadf00d;
obj = f;
```

В общем, по моему мнению, QPointer — не самый полезный из интеллектуальных указателей. Впрочем, нельзя сказать, что концепция интеллектуальных указателей вообще не нужна. Далее мы рассмотрим интеллектуальные указатели QSharedPointer и QAtomicPointer, весьма полезные в определенных ситуациях.

Паттерны и анти-паттерны

Я уже много говорил о том, что графический интерфейс пользователя следует отделять от остального кода программы. Размещение кода, отвечающего за логику событий программы, например в обработчике события, возникающего при нажатии на кнопку, считается анти-паттерном, т. е. примером неправильного программирования. Как же программировать правильно? Вернемся к нашему примеру. Класс Dialog, несмотря на свое название, сам по себе не связан с графическим интерфейсом пользователя. Класс, который непосредственно отвечает за графический интерфейс, включен в класс Dialog как свойство ui. На языке UML это отношение можно выразить как отношение принадлежности (рис. 3.6). При таком подходе было бы логично, если бы класс Dialog вообще не содержал методов, связанных непосредственно с элементами класса Ui::Dialog. Однако добиться этого полностью очень трудно, к тому же Qt Creator провоцирует нас на использование стиля программирования, столь популярного среди адептов Borland. Если в редакторе графического интерфейса Qt Creator щелкнуть правой кнопкой мыши по какому-либо виджету, то в открывшемся контекстном меню мы увидим пункт **Перейти к сло-**

ту.... Если мы выберем этот пункт, то сначала должны будем указать сигнал виджета, а затем попадем в окно редактора кода класса `Dialog`, в котором уже будет присутствовать заготовка слота специально для заданного сигнала этого виджета. По простоте такой стиль программирования сравним с программированием на Delphi, но фактически получается, что в классе `Dialog` мы создаем слоты, предназначенные специально для обработки сигналов класса `Ui::Dialog`. Это противоречит принципу разделения движка программы, частью которого является класс `Dialog`, и интерфейса пользователя, который представлен классом `Ui::Dialog`. Если эти классы связаны такими специфическими связями сигнал–слот, нам будет очень трудно модифицировать графический интерфейс независимо от логики работы программы (или наоборот). А ведь схема с агрегацией класса `Ui::Dialog` классом `Dialog` именно ради этого и создавалась. Ту же мысль в терминах объектно-ориентированного анализа можно выразить так: создание в классе `Dialog` специфических слотов, связанных с конкретными виджетами `Ui::Dialog`, устанавливает между классами отношение композиции, тогда как мы хотим установить более общее отношение — агрегации.

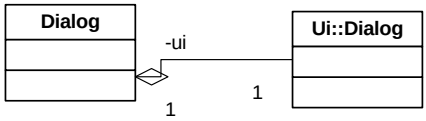


Рис. 3.6. Диаграмма отношений между классами `Dialog` и `Ui::Dialog`

Для решения этой задачи лучше всего вообще отказаться от автоматически сгенерированных слотов в классе `Dialog` и заменить их на свои собственные, отражающие логику работы программы. Если вы посмотрите на объявление класса `Dialog` (листинг 3.6), то увидите восемь слотов, каждый из которых предназначен не для какого-то определенного элемента пользовательского интерфейса, а реализует некую функцию программы.

Листинг 3.6. Класс `Dialog`, изолированный от пользовательского интерфейса

```
class Dialog : public QDialog
{
    Q_OBJECT

public:
    explicit Dialog(QWidget *parent = 0);
    ~Dialog();

private:
    Ui::Dialog *ui;
    bool connectDB(const QString &dBName);
    bool createDB(const QString &dBName);
    void createModel();
    void deleteModel();
```

```
    void setupView();
    TablePtr artists;
    TablePtr genres;
    TablePtr compositions;
    TableMapperPtr artistsMapper;
    TableMapperPtr genresMapper;
    TableMapperPtr compositionsMapper;
    ViewCatalogForm * artistsForm;
    ViewCatalogForm * genresForm;
    QString dbName;
public slots:
    void newDB();
    void openDB();
    void showArtistsForm();
    void showGenresForm();
    void addRecord();
    void deleteRecord();
    void cancelChanges();
    void save();
};
```

Слот `newDB()` позволяет пользователю создать новый файл базы данных на диске. Слот `openDB()` предназначен для открытия уже существующего файла базы данных. Слоты `showArtistsForm()` и `showGenresForm()` выводят окна редактирования списков исполнителей и жанров соответственно. Слоты `addRecord()` и `deleteRecord()` управляют добавлением и удалением записей из таблицы `Artists`. Слот `cancelChanges()` позволяет отменить несохраненные изменения, и, наконец, слот `save()` сохраняет изменения в базе данных. Правило, которое мы здесь используем, можно сформулировать так: если класс не отвечает непосредственно за формирование пользовательского интерфейса, его слоты должны соответствовать действиям программы, а не событиям виджетов.

> ***Примечание***
>
> Может показаться, что класс `Calculator` программы-калькулятора, с его методами `digitButtonPressed()` и `operationButtonPressed()`, нарушает это правило. На самом деле движок в программе-калькуляторе хорошо отделен от интерфейса, а кнопки, естественные для класса `Calculator`, являются частью программной модели калькулятора. Даже если вы реализуете калькулятор без кнопок, например с речевым вводом, вы все равно сможете использовать эти методы.

Будет логично, если мы свяжем каждый из этих слотов с соответствующим объектом `QAction`, или, как в нашем, упрощенном, интерфейсе — с соответствующим виджетом, но делать это лучше не в классе `Dialog`, а в классе `Ui::Dialog`. Как мы помним, код класса `Ui::Dialog` генерируется автоматически визуальным редактором, так что и связывание сигналов и слотов тоже следует выполнять в визуальном редакторе.

Прежде всего нужно сообщить визуальному редактору о слотах, которые мы добавили в класс `Dialog`. Для этого щелкните правой кнопкой мыши где-нибудь в об-

ласти окна `Dialog` и выберите команду **Изменить сигналы/слоты...**. Откроется окно редактирования сигналов и слотов (рис. 3.7), в котором мы увидим выделенные серым цветом сигналы и слоты, унаследованные классом `Dialog` от `QDialog`. В разделе **Слоты** добавляем новые слоты, которые мы ввели в классе `Dialog`. Слоты добавляются так же, как при вызове метода `connect()`, со списком параметров (или пустыми скобками, если параметров нет), например: `save()`, `move(int)`. Новые слоты выделены черным шрифтом (что означает, что их можно редактировать, в отличие от унаследованных слотов).

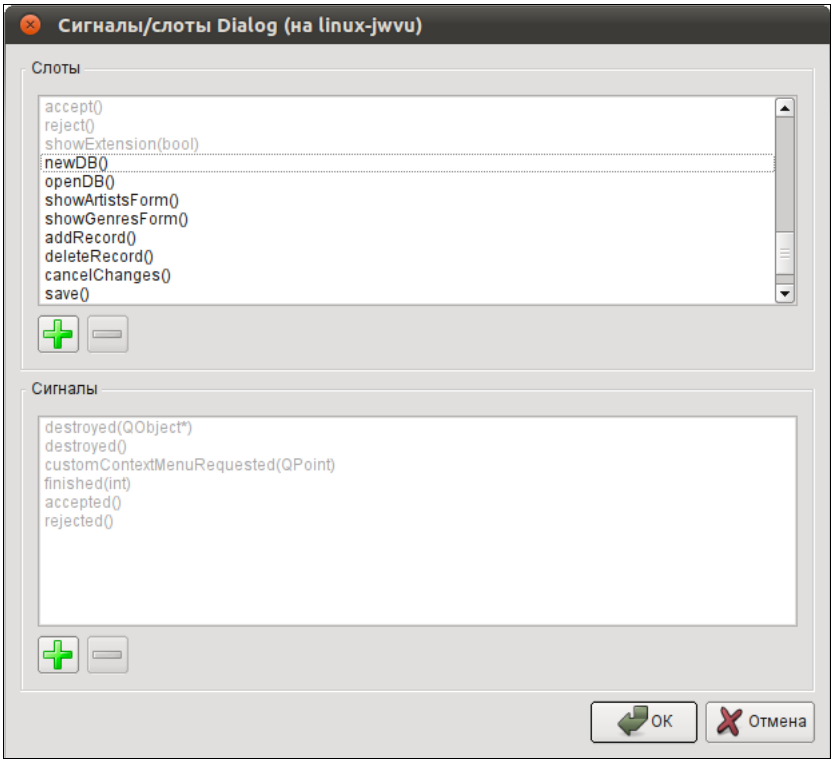


Рис. 3.7. Добавление новых слотов в визуальном редакторе

Теперь переходим в редактор сигналов и слотов (рис. 3.8) и связываем новые слоты с сигналами виджетов, добавленных в форму `Dialog`. В результате связывание сигналов и слотов произойдет автоматически в классе `Ui::Dialog`, и нам не нужно больше беспокоиться о нем. Более того, если мы захотим изменить графический интерфейс программы, нам, скорее всего, вообще не придется менять что-либо в классе `Dialog` (или эти изменения будут минимальными).

Итак, окинем общим взглядом схему нашего приложения баз данных (рис. 3.9). Приложение состоит из трех элементов: базы данных, программной модели БД и пользовательского интерфейса. Все элементы содержат данные, причем данные не всегда синхронизированы между элементами. Этот факт не представляет проблемы для персонального приложения баз данных, с которым работает только один поль-

зователь. Рано или поздно, так или иначе, данные будут синхронизированы. Серьезные проблемы могут возникнуть, когда с базой данных работает несколько пользователей, которые, возможно, имеют разные прикладные средства для взаимодействия с БД. Эту ситуацию мы обсудим в последующих главах.

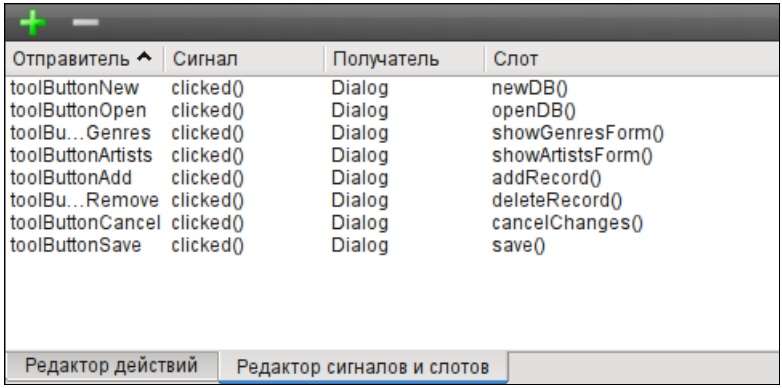


Рис. 3.8. Связывание новых слотов с сигналами виджетов

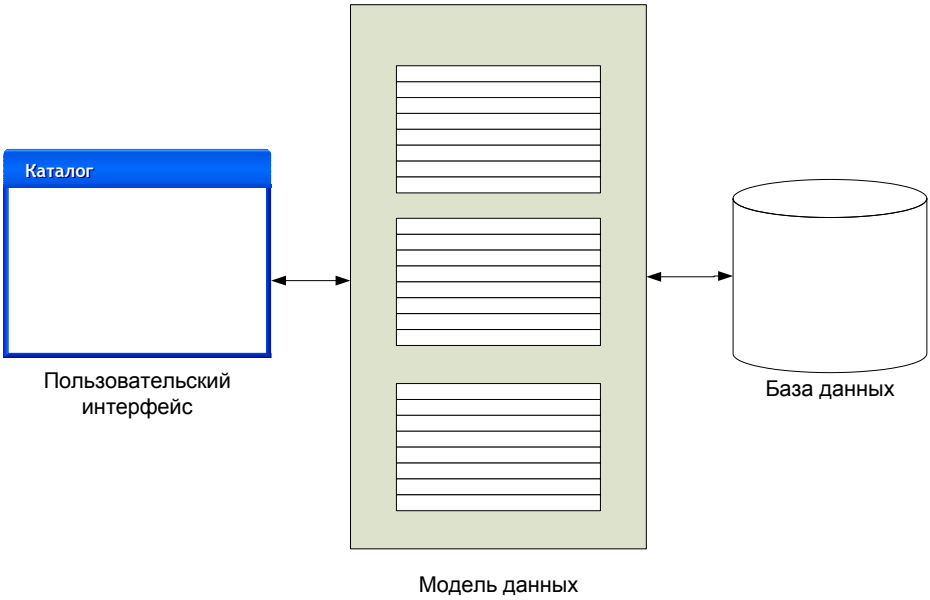


Рис. 3.9. Общая модель нашего приложения

В завершение этого раздела рассмотрим еще одну проблему архитектуры приложений баз данных. Грамотный проектировщик базы данных не полагается на прикладные средства в том, что касается поддержания логической целостности данных. Поддержание логической целостности (т. е. соответствия фактической структуры данных логической модели данных) осуществляется различными средствами самой БД. В простейшем случае это могут быть ограничения типа `UNIQUE`, `NOT NULL`,

`FOREIGN KEY` и т. д. В более сложных случаях используются триггеры и хранимые процедуры. В результате в грамотно построенной БД целостность модели не может быть нарушена случайными или злонамеренными действиями прикладной программы, а также ошибками, допущенными пользователем (пользователь может нарушить информационную целостность базы данных, но это совсем другая проблема). Означает ли это, что в прикладной программе мы можем не беспокоиться о таких действиях пользователя, которые могут нарушить логическую целостность данных, а проверка операций пользователя на предмет ненарушения логической целостности является избыточной? Нет, не означает. Грамотно спроектированная база данных не позволит пользователю внести изменения, нарушающие логическую целостность, но если эта защита сработает на уровне базы данных, часть данных, введенных пользователем в программе (в том числе данных, которые ничего не нарушают) может быть потеряна. Дабы избежать такого развития событий, желательно дублировать в приложении проверки целостности данных, которые выполняются в самой базе данных с целью минимизировать потерю данных пользователем.

Модульное тестирование

Если вы еще не прониклись всей глубиной и мудростью принципа модульности построения приложений, то вот вам дополнительный аргумент: при модульном построении приложения мы можем задействовать мощный и удобный инструмент проверки программы на наличие ошибок, называемый модульным тестированием (unit testing).

Главное преимущество модульного тестирования заключается в автоматизации самого процесса тестирования. Написать автоматический тестер для всей программы в целом трудно, во-первых, потому, что программа, как правило, рассчитана на взаимодействие с человеком, а во-вторых, потому, что программа в целом предоставляет слишком много возможных сценариев взаимодействия, которые просто нельзя учесть. В то же время, если программа разбита на модули, каждый из которых выполняет сравнительно небольшое количество функций, мы можем написать автоматизированные тестеры для каждого модуля.

Автоматизированное тестирование важно потому, что в ходе работы над проектом тестирование одних и тех же модулей приходится выполнять многократно. В процессе развития сложный проект неизбежно выходит на такую стадию, когда внесение изменений в одну часть кода может привести к непредвиденным изменениям в работе других его частей. Само разделение программы на модули в некоторой степени решает эту проблему, однако оно не может решить проблему непредвиденных изменений внутри самого модуля. Именно для этой цели и служит модульное тестирование. Внося изменения в функциональность модуля, мы не можем быть уверенными в том, что ранее протестированный код будет работать так же, как и прежде. Но если тестирование модуля автоматизировано, мы можем провести тестирование ранее протестированного кода (регрессивное тестирование), не выполняя при этом большого объема рутинной работы.

Автоматизированное тестирование модулей упрощает такую операцию как рефакторинг кода. Благодаря автоматизированному тестированию мы легко можем убедиться, что после рефакторинга код работает как и прежде.

Наконец, набор автоматизированных тестов для модуля можно рассматривать как некую спецификацию его работы. Автоматизированное тестирование играет важную роль в таких областях как экстремальное программирование и разработка, управляемая тестами, что, однако, выходит за рамки этой книги.

До сих пор я акцентировал ваше внимание на положительных сторонах модульного тестирования. Однако его не следует переоценивать. Модульное тестирование может обнаружить только те ошибки, проверки на которые заложены в тестер. Это значит, что модульное тестирование не может обнаружить ошибок, существование которых программист не предполагает в принципе и которые он не мог бы обнаружить каким-то другим способом. Иначе говоря, модульное тестирование упрощает рутинную часть тестирования, но оно не сделает тестирование более глубоким или тщательным, чем сделал бы его разработчик теста, если бы тестировал вручную.

Модульное тестирование хорошо работает в ситуации, когда тестируемый модуль способен функционировать отдельно от остальных модулей, а такое бывает далеко не всегда. Классический пример — модуль взаимодействия с базой данных, который не может работать без базы данных. Однако принципы модульного тестирования требуют изолировать модуль (иначе мы просто не сможем убедиться, что ошибка, буде таковая обнаружится, возникает именно в данном модуле). Обычно для решения этой проблемы программисты пишут эмуляторы тех интерфейсов, с которыми модуль должен взаимодействовать. Но это, во-первых, трудоемкая операция, а, во-вторых, для корректного тестирования мы должны быть уверены, что эмулятор сам не содержит ошибок.

Другой подход — начинать тестирование программы с модуля, который не зависит от других модулей, затем тестировать модули, которые зависят от этого модуля и т. д. Вообще говоря, модульное тестирование эффективно только тогда, когда оно применяется на всех этапах разработки всеми участниками проекта.

Что представляет собой автоматизированный тестер для модульного тестирования? Обычно в набор пакетов для генерации автоматизированных тестеров входят средства создания специальных приложений. Эти приложения выполняют тестирование переданного им модуля и выводят отчет о его результатах. Вы пишете приложение-тестер, подключаете к нему тестируемый модуль, описываете контрольные примеры для тестирования, а затем собираете приложение-тестер. В процессе выполнения приложение сообщает вам, прошел ли модуль контрольные примеры, и если не прошел, то какие именно. Когда текст модуля меняется, вы просто повторно собираете приложение-тестер. При этом, если модулей в программе много, вовсе не обязательно писать отдельную программу для тестирования каждого модуля. Одна программа может тестировать несколько модулей.

Примечание

Уже из этого описания должно быть понятно, что модульное тестирование будет выполнено настолько хорошо, насколько хорошо составлены контрольные примеры. Но вам, по крайней мере, не придется вводить один и тот же контрольный пример несколько раз.

Qt и модульное тестирование

Библиотека Qt предоставляет нам собственное средство модульного тестирования — подсистему Qt Unit Test Library. Мы рассмотрим работу Unit Test Library на практическом примере, для чего воспользуемся программой-калькулятором из *главы 2*. Класс `Calculator` этой программы представляет собой идеальный объект для модульного тестирования. Пример тестируемого калькулятора вы найдете в папке Ch3/calculator-testing. Помимо текстов программы-калькулятора в папке ut-calculator вы найдете исходные тексты программы-тестера, использующей Qt Unit Test Library (исходные тексты программ к книге доступны на сайте **http://symmetrica.net/qt47book/**).

Создание программы-тестера мы начнем с класса `CalculatorTest`, в котором опишем контрольные примеры для класса `Calculator` (листинг 3.7).

Листинг 3.7. Объявление класса `CalculatorTest`

```
class CalculatorTest : public QObject
{
    Q_OBJECT
public:
    explicit CalculatorTest(QObject *parent = 0);

private slots:
    void digitButtonPressed();
    void operationButtonPressed();
    void cancelButtonPressed();
};
```

Объявление класса `CalculatorTest` выглядит очень просто. Помимо конструктора (который ничего не делает) класс содержит три приватных слота. Именно в этих слотах и происходит самое интересное (листинг 3.8).

Листинг 3.8. Реализация класса `CalculatorTest`

```
#include "calculatortest.h"
#include "../calculator.h"
#include <QtTest/QTest>
#include <QtTest/QSignalSpy>

CalculatorTest::CalculatorTest(QObject *parent) :
    QObject(parent)
{
}

void CalculatorTest::digitButtonPressed()
{
    Calculator calculator;
```

```
    QSignalSpy spy(&calculator, SIGNAL(valueChanged(int)));
    calculator.digitButtonPressed(digit1);
    calculator.digitButtonPressed(digit0);
    QVERIFY(spy.count() == 2);
    QList<QVariant> args = spy.takeLast();
    QVERIFY(args.at(0) == 10);
}

void CalculatorTest::operationButtonPressed()
{
    Calculator calculator;
    QSignalSpy spy(&calculator, SIGNAL(valueChanged(int)));
    calculator.digitButtonPressed(digit1);
    calculator.operationButtonPressed(opPlus);
    calculator.digitButtonPressed(digit1);
    calculator.digitButtonPressed(digit0);
    calculator.operationButtonPressed(opEqual);
    QVERIFY(spy.count() == 4);
    QList<QVariant> args = spy.takeLast();
    QVERIFY(args.at(0) == 11);
    spy.clear();
    calculator.digitButtonPressed(digit2);
    calculator.operationButtonPressed(opMinus);
    calculator.digitButtonPressed(digit1);
    calculator.digitButtonPressed(digit6);
    calculator.operationButtonPressed(opEqual);
    QVERIFY(spy.count() == 4);
    args = spy.takeLast();
    QVERIFY(args.at(0) == -14);

}

void CalculatorTest::cancelButtonPressed()
{
    Calculator calculator;
    QSignalSpy spy(&calculator, SIGNAL(valueChanged(int)));
    calculator.digitButtonPressed(digit7);
    calculator.digitButtonPressed(digit1);
    calculator.operationButtonPressed(opCancel);
    calculator.digitButtonPressed(digit3);
    QVERIFY(spy.count() == 4);
    QList<QVariant> args = spy.at(2);
    QVERIFY(args.at(0) == 0);
    args = spy.takeLast();
    QVERIFY(args.at(0) == 3);
}
```

Изучение этого листинга разумно начать с заголовочных файлов. Помимо файлов calculatortest.h и calculator.h (последний содержит объявление тестируемого класса) мы включаем еще два заголовочных файла: `<QtTest/QTest>` и `<QtTest/QSignalSpy>`. Первый из этих файлов содержит объявления различных констант и макросов, которые могут пригодиться при автоматизации тестирования, второй — объявление полезного класса `QSignalSpy`. Кстати, для подключения бибилиотеки не забудьте добавить в файл с расширением pro программы-тестера такую строку:

```
CONFIG += qtestlib
```

В самом простом случае тестирование выглядит так: мы вызываем методы тестируемого объекта, передавая им различные аргументы. Затем мы сравниваем значения, возвращаемые методами объекта, с контрольными значениями с помощью специальных макросов `QCOMPARE()`, `QTEST()` или `QVERIFY()`. Если возвращенные значения совпадают с ожидаемыми, значит, данный тест пройден. Если нет, программа-тестер выдаст информацию об этом. Объем информации об ошибке зависит от выбранного макроса. Самый информативный макрос — `QVERIFY()`. В нашем случае все несколько сложнее. Класс `Calculator` передает результаты своей работы с помощью сигналов, и для того чтобы его протестировать, мы должны перехватывать эти сигналы. Для этого нам и нужен класс `QSignalSpy`.

Мы создаем объект тестируемого класса в каждом методе тестера. Можно было бы воспользоваться единым объектом, созданным в методе `initTestCase()` (этот метод объявляется в разделе `private slots:`, как и остальные методы тестера, но всегда вызывается первым). Однако, по моему мнению, это снизило бы независимость отдельных тестов друг от друга. Если по окончании выполнения основных контрольных примеров необходимо выполнить (и протестировать) некие специальные завершающие действия, в класс-тестер необходимо ввести метод `cleanupTestCase()`, который будет выполнен последним. Мы также создаем объект `QSignalSpy`. В конструкторе этого объекта мы передаем указатель на тестируемый объект и описание сигнала, за которым надо следить. Далее мы вызываем методы объекта `calculator`, имитируя, фактически, действия пользователя. Процесс имитации выглядит гораздо более трудоемким, чем щелканье нескольких кнопок мыши. В качестве утешения мы можем напомнить себе, что эту имитацию нам нужно сделать один раз и потом ее можно много раз использовать.

> ***Примечание***
>
> Если мы не определили методы `initTestCase()`и `cleanupTestCase()`, программа-тестер все равно будет работать.

После того как ввод данных контрольного примера окончен, все, что нам остается сделать, — проанализировать информацию, собранную объектом `spy`. По сути, этот объект представляет собой список перехваченных сигналов со списками значений их аргументов. Метод `count()` возвращает общее количество перехваченных сигналов. Поскольку мы знаем, сколько сигналов `valueChanged()` должен послать объект класса `Calculator` в том или другом случае, мы можем сравнить фактическое число переданных сигналов с ожидаемым. В некоторых случаях это поможет выявить ошибки.

Тест digitButtonPressed() проверяет работу кнопок числового ввода. В ответ на вызовы

```
calculator.digitButtonPressed(digit1);
calculator.digitButtonPressed(digit0);
```

объект calculator должен послать два сигнала valueChanged() со значениями 1 и 10 соответственно. Мы проверяем количество сигналов с помощью метода count(). Метод takeLast() возвращает список значений аргументов последнего из отслеженных сигналов. Доступ к значениям аргументов других сигналов можно получить с помощью других методов класса QList, от которого частично ведет свое происхождение класс QSignalSpy. Список значений аргументов сигнала возвращается в виде объекта QList<QVariant>. Нам остается только сравнить аргумент последнего сигнала с ожидаемым значением, что мы и делаем с помощью макроса QVERIFY(). Аргументом макроса является значение булева типа и, в принципе, туда можно записать любое выражение, которое возвращает true или false. Если оно равно false, тест считается непройденным и программа выдаст сообщение об этом. В нашем распоряжении есть также макрос QVERIFY2(), который позволяет не только проверить выполнение некоего условия, но и, в случае его невыполнения, выдать дополнительное текстовое сообщение. Например, строка

```
QVERIFY2(args.at(0) == 10, args.at(0).toByteArray().data());
```

в случае невыполнения условия args.at(0) == 10 распечатает фактическое значение args.at(0). Таким образом, мы можем совместить тестирование и отладку.

Точно так же в методе CalculatorTest::operationButtonPressed() мы проверяем выполнение двух арифметических операций, которые выполняет калькулятор, а в методе CalculatorTest::cancelButtonPressed() — работу операции opCancel.

Для того чтобы превратить тестер в самостоятельную программу, мы должны определить функцию main(). В случае с программой, использующей Qt Unit Testing Library, эта функция определяется с помощью макросов (листинг 3.9).

Листинг 3.9. Создание главной функции приложения-тестера

```
#include <QtTest/QTest>
#include "calculatortest.h"
QTEST_MAIN(CalculatorTest);
```

Макрос QTEST_MAIN(), которому в качестве аргумента передается имя класса CalculatorTest, сам создаст необходимую функцию. Если вы хотите знать, как это работает, то вот приблизительное объяснение: функция qExec(), объявленная в том же файле <QtTest/QTest>, получает в качестве аргумента указатель на объект класса-тестера (не путайте его с тестируемым классом), с помощью системы метаобъектов определяет список слотов этого объекта и вызывает последовательно все эти слоты. При этом стоит отметить, что создание объекта происходит за пределами функции qExec(), так что любые ошибки, возникающие в процессе создания, не попадут в стандартный отчет.

Если класс `Calculator` не содержит ошибок (по крайней мере, таких которые мог бы обнаружить автоматизированный тестер), то вывод программы-тестера будет выглядеть примерно так, как показано в листинге 3.10.

Листинг 3.10. Типичный вывод тестирующей программы

```
********* Start testing of CalculatorTest *********
Config: Using QTest library 4.7.0, Qt 4.7.0
PASS : CalculatorTest::initTestCase()
PASS : CalculatorTest::digitButtonPressed()
PASS : CalculatorTest::operationButtonPressed()
PASS : CalculatorTest::cancelButtonPressed()
PASS : CalculatorTest::cleanupTestCase()
Totals: 5 passed, 0 failed, 0 skipped
********* Finished testing of CalculatorTest *********
```

Теперь, когда у меня есть автоматизированный тестер класса `CalculatorTest`, я могу проделать над ним операцию, которая, возможно, нуждается в автоматическом тестировании больше, чем любая другая. Я имею в виду рефакторинг. Метод `operationButtonPressed()` оригинального класса `Calculator` выглядит несколько неуклюже (листинг 3.11).

Листинг 3.11. Исходный метод `operationButtonPressed()`

```
void Calculator::operationButtonPressed(int button)
{
    if ((button == opMinus)&&(mode == InputMode)&&(fgValue == 0)) {
        sign = -1;
    } else {
        sign = 1;
    }
    if ((button != opCancel)&&(button != opEqual)) {
        mode = CopyMode;
        operation = (int) button;
        return;
    }
    if (button == opEqual) {
        switch (operation) {
            case opPlus:
              fgValue += bgValue;
              break;
            case opMinus:
              fgValue = bgValue — fgValue;
            default:;
          }
          mode = CopyMode;
          bgValue = 0;
    }
```

```
    if (button == opCancel) {
        bgValue = fgValue = 0;
        operation = opNone;
        mode = InputMode;
    }
    emit valueChanged(fgValue);
}
```

Я выделю из этого метода два новых метода: `equalPressed()` и `otherOpPressed()`. Первый метод обрабатывает ситуацию, когда нажата кнопка с символом =, второй метод обрабатывает нажатие остальных кнопок операций. Чтобы не менять внешний интерфейс класса `Calculator`, я сохраню метод `operationButtonPressed()` (листинг 3.12).

Листинг 3.12. Методы класса `Calculator` после рефакторинга

```
void Calculator::operationButtonPressed(int button)
{
    if (button == opEqual)
        equalPressed();
    else
        otherOpPressed(button);
}

void Calculator::otherOpPressed(int button)
{
    if ((button == opMinus)
        &&(((mode == InputMode)&&(fgValue == 0))
        || (mode == CopyMode))) {
        sign = -1;
    } else {
        sign = 1;
        if ((button != opCancel)) {
            mode = CopyMode;
            operation = (int) button;
        } else {
            bgValue = fgValue = 0;
            operation = opNone;
            mode = InputMode;
            emit valueChanged(fgValue);
        }
    }
}

void Calculator::equalPressed()
{
    sign = 1;
```

```
    switch (operation) {
    case opPlus:
        fgValue += bgValue;
        break;
    case opMinus:
        fgValue = bgValue — fgValue;
    default:;
    }
    mode = CopyMode;
    emit valueChanged(fgValue);
}
```

Теперь я могу заново собрать программу-тестер и убедиться, что рефакторинг выполнен правильно (точнее говоря, тестер не находит ошибок).

> ***Примечание***
>
> В результате рефакторинга у нас появился новый метод — `equalPressed()`. Если теперь мы захотим добавить в калькулятор поддержку новых операций, например умножения и целочисленного деления, нам потребуется изменить только содержимое этого метода.

Допустим теперь, что я ошибся, забыл добавить генерацию сигнала `valueChanged()` в метод `otherOpPressed()` (листинг 3.13).

Листинг 3.13. Метод с ошибкой

```
void Calculator::otherOpPressed(int button)
{
    if ((button == opMinus)
        &&(((mode == InputMode)
        &&(fgValue == 0)) || (mode == CopyMode))) {
        sign = -1;
    } else {
        sign = 1;
        if ((button != opCancel)) {
            mode = CopyMode;
            operation = (int) button;
        } else {
            bgValue = fgValue = 0;
            operation = opNone;
            mode = InputMode;
        }
    }
}
```

В этом случае тестер выдаст сообщение об ошибке (листинг 3.14).

Листинг 3.14. Сообщение об ошибке

```
********* Start testing of CalculatorTest *********
Config: Using QTest library 4.7.0, Qt 4.7.0
PASS : CalculatorTest::initTestCase()
PASS : CalculatorTest::digitButtonPressed()
PASS : CalculatorTest::operationButtonPressed()
FAIL! : CalculatorTest::cancelButtonPressed()
'spy.count() == 4' returned FALSE. ()
..\ut-calculator\calculatortest.cpp(54) : failure location
PASS : CalculatorTest::cleanupTestCase()
Totals: 4 passed, 1 failed, 0 skipped
********* Finished testing of CalculatorTest *********
```

Поскольку результатом моей преднамеренной ошибки стало то, что операция `opCancel` не генерирует сигнал `valueChanged()`, счетчик количества сигналов в методе `CalculatorTest::cancelButtonPressed()` не сработал.

Следует отметить, что возможности библиотеки Qt Unit Testing Library не исчерпываются перечисленными выше. В библиотеке есть, например, средства для создания тестов графического пользовательского интерфейса, в том числе средства имитации событий мыши и клавиатуры. Все это подробно описано в официальной документации по Qt. Документация по Qt Unit Testing Library содержит несколько примеров автоматизированного тестирования, правда, на момент написания этой книги эти примеры довольно сильно отстали от жизни, т. е. от новых версий Qt. Так что будьте готовы к тому, что вам придется адаптировать их. Впрочем, это нетрудно.

Еще один интересный раздел функциональности, появившийся в Unit Testing Library в версии Qt 4.7, — возможность определения производительности методов. Вообще-то определение производительности и выявление узких мест традиционно относится к несколько иной сфере тестирования — профилировке. Но благодаря новым возможностям Unit Testing Library вы можете проверить, например, насколько рефакторинг замедлил работу модуля. Для этого нужно использовать макрос `QBENCHMARK` (листинг 3.15).

Листинг 3.15. Применение макроса `QBENCHMARK`

```
class MytBenchmarkTest: public QObject
 {
     Q_OBJECT
 private slots:
     void lengthyFunctionTest()
     {
         QBENCHMARK {
             lengthyFunction();
         }
     }
 };
```

Этот тест позволит узнать, сколько времени занимает выполнение функции `lengthyFunction()`.

Тест производительности может выдавать различные данные, например физическое время, которое потребовалось на выполнение функции. Для получения более точного результата тест физической продолжительности выполняется многократно, и вы должны это учитывать. Если функция `lengthyFunction()` не допускает многократных повторных вызовов, этот тест для нее не годится. Другая форма выдачи результата — количество тактов процессора, которое потребовалось на выполнение функции. Этот тест требует меньше повторов, но он, естественно, зависит от модели процессора, настроек компиляции и фазы Луны. О других возможностях вывода данных при определении производительности вы узнаете из документации.

Приложения-тестеры, созданные с использованием приведенных выше макросов, принимают ряд аргументов командной строки. Например, аргументы `-v1`, `-v2`, `-vs` контролируют объем информации, которую выводит программа, аргумент `-xml` заставляет программу сохранять данные в формате XML, а аргумент `-tickcounter` указывает тесту производительности, если таковой присутствует, что результат нужно выдавать в количестве тактов процессора. Об остальных ключах командной строки вы можете узнать из документации к Qt.

ГЛАВА 4

Инструментарий профессионального разработчика

Программы, которые мы рассмотрим в этой главе, не являются частью Qt library. Возможно, вы уже пользуетесь этими программами или их аналогами в работе над другими проектами. Наша цель — показать, насколько важны для работы со сложными проектами Qt некоторые вспомогательные средства. Тому, кто начал их использовать, уже практически невозможно представить разработку проектов без их помощи.

Отладчик GDB

Отладчик GDB (GNU Debugger) является частью инструментария GNU, и если вы пишете программы для Linux, вам неизбежно придется иметь с ним дело. Кроме того, специальная версия GDB для Windows входит в состав MinGW, и ее использует Qt Creator в качестве стандартного средства отладки. Отладчик GDB можно использовать как в составе интегрированных сред (Qt Creator, Eclipse), так и самостоятельно.

При первом знакомстве GDB может напугать программистов, привыкших к встроенным отладчикам графических интегрированных сред разработки. На самом деле все не так страшно. Выучив несколько простых команд, вы сможете сделать с помощью отладчика GNU все, что вы могли бы сделать в плане отладки в средах Microsoft или Borland. Выучив еще несколько команд, вы сможете делать такое, что пользователям графических IDE и не снилось.

> ***Примечание***
>
> На платформе Windows Qt Creator может также использовать отладчик CDB, который входит в пакеты Windows SDK и Debugging Tools for Windows (и то, и другое доступно для бесплатной загрузки с сайта Microsoft), но мы этот вариант рассматривать не будем.

Одна из причин, по которой Qt SDK по умолчанию использует именно этот отладчик (и одна из причин того, почему мы будем говорить именно о нем) заключается в его кросс-платформенности (рис. 4.1). Если вы компилилируете программы для процессоров семейства ARM, то можете выполнять отлаживать эти программы, не покидая среду Windows или Linux вашего настольного компьютера, основанного на

процессоре Intel. Сама программа, конечно, должна при этом выполняться на устройстве с процессором ARM или в его эмуляторе.

Для того чтобы извлечь максимум из отладчика GNU, программу, предназначенную для отладки, следует скомпилировать с дополнительной отладочной информацией. Если вы запускаете сборку отдельно от интегрированной среды разработки, то добавлением отладочной информации управляет ключ `-g` команд `gcc` и `g++`. В интегрированных средах разработки формат генерации кода зависит от выбранной настройки сборки (`Debug` или `Release`). Нас, естественно, интересует `Debug`. Qt Creator по умолчанию создает именно такую конфигурацию. Далее отладчиком в Qt Creator управляет меню **Отладка**.

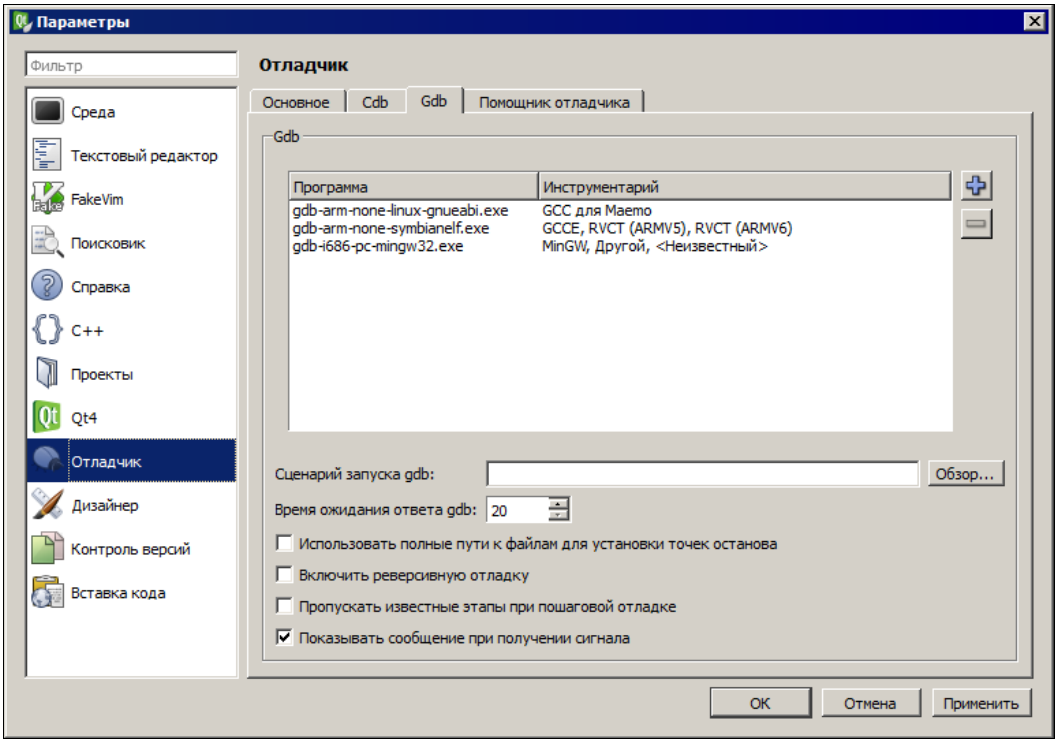


Рис. 4.1. Демонстрация кросс-платформенности отладчика GDB

Примечание

Файл программы или библиотеки, скомпилированный с добавлением отладочной информации, "весит" больше, чем обычный файл, однако вам вовсе не обязательно перекомпилировать модуль по окончании отладки. На платформе Linux отладочную информацию можно в любой момент удалить из двоичного модуля с помощью утилиты `strip`.

Для настройки взаимодействия Eclipse и GDB необходимо создать конфигурацию отладчика (рис. 4.2). Поскольку Eclipse — универсальная среда, предназначенная для работы с проектами разных типов, конфигурацию отладчика нужно создавать

отдельно для каждого проекта. Впрочем, это не так уж и сложно, поскольку однажды созданную конфигурацию можно копировать из проекта в проект с минимальными изменениями.

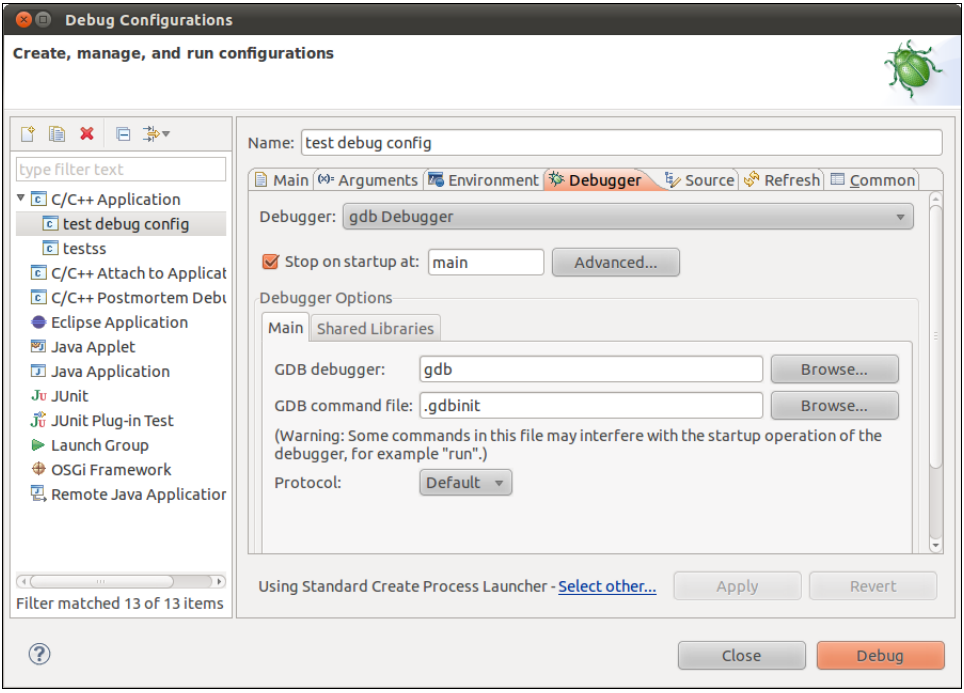


Рис. 4.2. Конфигурация отладчика в Eclipse

Если запустить GDB в отдельном окне консоли (рис. 4.3), то появится приглашение командной строки (`gdb`). Далее все команды GDB вводятся так же, как в консольной оболочке операционной системы.

> ***Примечание***
>
> Поскольку GDB создавался для Linux, а между архитектурами Linux и Windows существуют серьезные различия, GDB под Windows не обладает всей функциональностью GDB под Linux. Нельзя, например, прервать выполнение отлаживаемой программы с помощью комбинации клавиш <Ctrl>+<C>. Если вы используете GDB внутри таких интегрированных сред как Qt Creator или Eclipse, различия между платформами практически незаметны.

Для того чтобы получить справку по работе с GDB, вам не обязательно выходить из интерактивного режима программы gdb. Команда `help` позволит вам получить справочную информацию. Вы можете получить информацию о любой команде gdb с помощью команды `help` *`имя_команды`*. Например, команда

```
(gdb) help exec-file
```

распечатает справочную информацию о команде `exec-file`. Получать справку о конкретной команде имеет смысл, если вы знаете, какая команда вам нужна. Если

вы новичок в работе с отладчиком, вам, прежде всего, понадобится информация о том, какие вообще команды существуют в системе. В интерактивном режиме отладчика GNU программисту доступны сотни команд (бóльшая часть которых никогда ему не понадобится). Для удобства навигации в справочной системе GDB команды сгруппированы по разным разделам:

- `aliases` — псевдонимы команд;
- `breakpoints` — команды, управляющие точками останова;
- `data` — команды, позволяющие манипулировать данными программы в процессе отладки;
- `files` — команды для работы с файлами;
- `internals` — внутренние команды GDB;
- `obscure` — команды для опытных пользователей;
- `running` — команды, управляющие запуском отлаживаемой программы;
- `stack` — команды, управляющие просмотром стека;
- `status` — команды, сообщающие различные сведения об отлаживаемой программе, отладчике и системе;
- `support` — команды, управляющие отладчиком;
- `tracepoints` — команды, управляющие трассировкой;
- `user-defined` — команды, определенные пользователем.

Например, для того, чтобы получить сведения о командах, управляющих точками останова, напишем:

```
(gdb) help breakpoints
```

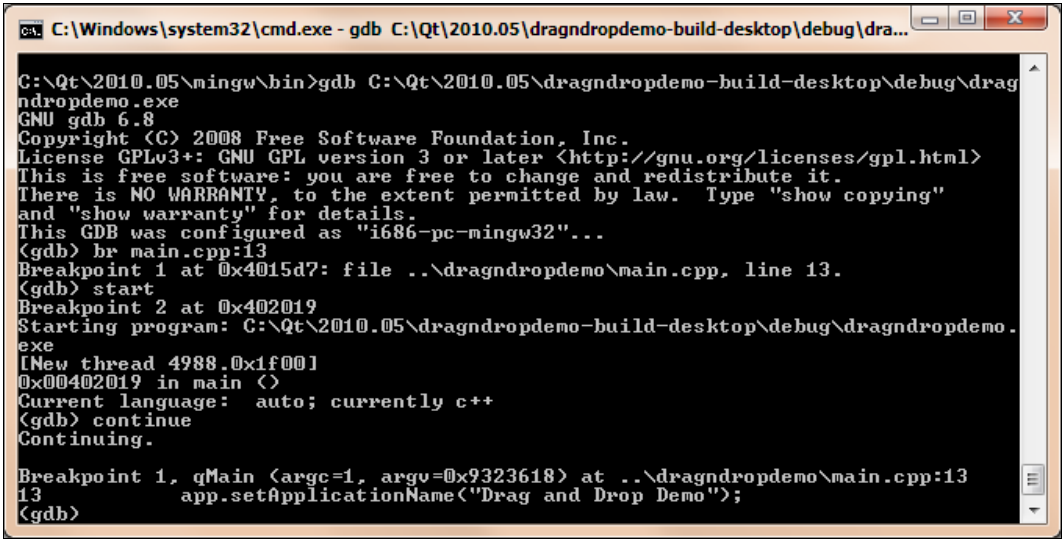


Рис. 4.3. Отладчик GDB в окне консоли Windows

Если вы хотите отлаживать программу в самостоятельном сеансе GDB, то ее нужно сначала загрузить в отладчик. Это можно сделать во время запуска GDB из командной строки операционной системы:

```
gdb dragndropdemo
```

А можно и после запуска GDB, из командной строки отладчика:

```
(gdb) exec-file dragndropdemo
```

В любом случае отладчик лишь подготовит среду для отладки программы, но не запустит программу на выполнение. Вообще следует помнить, что отладчик всегда находится в одном из трех режимов: отлаживаемая программа работает, отлаживаемая программа приостановлена и программа не выполняется. Большую часть команд отладчика можно вводить в последних двух режимах. Для запуска программы служат команды `run` и `start`.

Точки останова

Точки останова, без сомнения, важнейший инструмент отладчика, т. к. именно с приостановки выполнения программы начинаются все наши попытки понять, что же именно в программе происходит не так. И вообще, для того чтобы выполнить какую-то команду отладчика, программу надо сначала остановить.

Самое важное, что следует знать про точки останова, — это то, что не каждой строке текста программы соответствуют операции микропроцессора. Если мы попытаемся создать точку останова в строке, которой отладчик не может сопоставить исполнимый код, GDB создаст точку останова на первой строке после данной, которой можно сопоставить исполнимый код. В результате иногда точки останова оказываются в довольно неожиданном месте (листинг 4.1).

Листинг 4.1. Фрагмент текста отлаживаемой программы

```
151    if (!strchr(",.():;!?\" ""%", nxt->vers->Alt[0].Code[0])) {
152        rast->vers->Alt[0].Code[0] = ' ';
153        rast->vers->Alt[0].Code[1] = 0;
154        rast = CSTR_GetPrev(rast);
155        continue;
156    } else{
157        i++
        ...
    }
```

Если мы попытаемся создать точку останова в строке 155, отладчик создаст точку останова в строке 157. Дело в том, что строке 155 (как и строке 156) нельзя сопоставить машинный код. Первая строка после строки 155, для которой это можно сделать, — строка 157, но она находится в другом логическом блоке. Очевидно, что это совсем не то, чего мы хотели. При запуске программы на отладку в Qt Creator иногда можно даже увидеть, как заданная нами точка останова "перепрыгивает"

с одной строки на другую. В Eclipse точки останова остаются на своих местах, но фактически программа все равно будет остановлена там, где ее можно остановить.

При непосредственной работе с отладчиком точка останова создается командой `break` (специально для ленивых: эта команда может быть сокращена до одной буквы – `b`) Например, команда

```
(gdb) b foo
```

создаст точку останова в начале функции `foo()`. Существует и другой способ указать точки останова. Например, команда

```
(gdb) break mainform.cpp:15
```

создаст точку останова в строке 15 файла mainform.cpp (с учетом сказанного ранее фактическая точка останова может быть создана на строке с другим номером, бóльшим 15). Возобновить выполнение программы можно с помощью команд `step` или просто `s` (будет выполнена одна строка), `continue` или просто `c` (программа будет выполняться до следующей точки останова или пока не завершится) и других.

> ***Примечание***
>
> Точки останова можно задавать только тогда, когда программа остановлена, но не тогда, когда она выполняется. Это одна из особенностей работы с GDB отдельно от среды разработки.

Обзор данных

Когда программа приостановлена, мы можем заглянуть в содержимое переменных, видимых в данном контексте (ради этого программы чаще всего и приостанавливают). Интегрированные среды разработки позволяют нам без труда заглянуть практически в любую переменную, находящуюся в области видимости (рис. 4.4). Можно также отредактировать значения переменных.

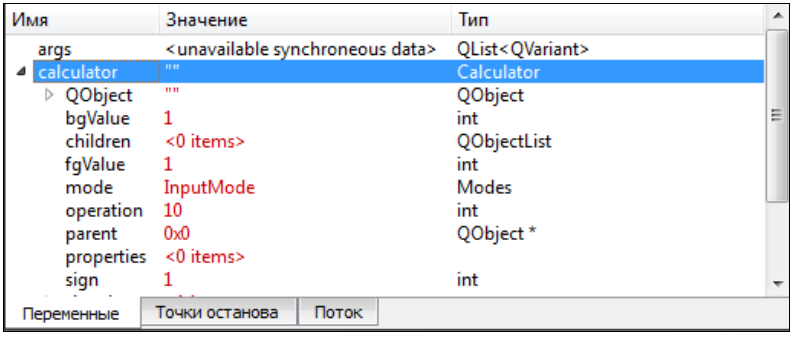


Рис. 4.4. Просмотр значений переменных в Qt Creator

При использовании GDB в автономном режиме все выглядит немного сложнее. Для того чтобы просто посмотреть содержимое переменной, достаточно вызвать команду `print`, передав ей в качестве аргумента имя переменной.

Очень часто в процессе отладки бывает необходимо отследить, как меняется значение некоторой переменной по ходу выполнения программы. Чтобы не вызывать каждый раз команду `print`, мы можем создать контрольную точку доступа к данным. Контрольная точка доступа будет оповещать нас о том, что содержимое некоторой переменной изменилось. Для реализации этой функции GDB стремится использовать аппаратные средства — так называемые ловушки процессора, которые позволяют зафиксировать обращение к определенному адресу памяти. В результате контрольные точки доступа к данным позволяют отслеживать не только явные изменения переменных, которые происходят в процессе выполнения операции присваивания, но и непреднамеренные изменения, которые могут произойти, например, в результате случайного перезаписывания области данных операцией копирования. Например, если для установки контрольной точки для наблюдения за переменной `i` мы напишем:

```
(gdb) watch i
```

Отладчик ответит:

```
Hardware watchpoint 1: i
```

> ***Примечание***
>
> К сожалению, количество аппаратных ловушек обращений к адресам памяти ограничено (для разных процессоров и систем — разными значениями), так что использовать контрольные точки следует рационально. Для удаления контрольных точек доступа служит команда `delete`.

Теперь при пошаговом проходе программы мы получим предупреждение в тот момент, когда содержимое памяти, соответствующее переменной `i`, изменится:

```
Hardware watchpoint 1: i
Old value = (int) 0x957a19c
New value = (int) 0x95776d0
```

Данное предупреждение может появиться там, где мы его совсем не ожидаем, и это будет свидетельствовать о том, что какая-то операция записи в память вырвалась за отведенные ей границы. Разумеется, такие чудеса аппаратного слежения доступны только для обработки простых типов данных.

Отслеживать изменение содержимого переменной с помощью комбинации команд `watch` и `step/next` не очень-то удобно. У отладчика GNU есть другой, лучший способ. Команда `awatch` будет останавливать выполнение программы всякий раз, когда значение заданной переменной считывается или записывается. Так же, как и команда `watch`, команда `awatch` стремится использовать аппаратные ловушки для выявления обращений к конкретному адресу и может отлавливать неявные обращения к нему.

В заключение отметим один аргумент в пользу отладки в Qt Creator. Эта среда разработки использует специальные помощники отладчика, которые позволяют просматривать содержимое переменных таких типов, как `QString`. В стандартном отладчике GDB, так же как и в отладчике Microsoft Visual Studio, вы не увидите в со-

держимом этих переменных ничего, кроме указателей на непрозрачные структуры данных.

Стек вызовов

Если в программе во время отладки происходит нечто, требующее вашего внимания, вам может понадобиться не только информация о функции, при вызове которой случилось страшное, но и сведения о полном состоянии стека вызовов.

В интегрированных средах стек вызовов оформлен красиво и удобно (рис. 4.5) и позволяет вам перемещаться по цепочке вызовов одним щелчком мыши.

Уровень	Функция	Файл	Строка	Адрес
0	MainWindow	mainwindow.cpp	16	0x00401897
1	qMain	main.cpp	14	0x0040162c
2	WinMain@16	qtmain_win.cpp	131	0x004023f2
3	main		0	0x00402118

Рис. 4.5. Стек вызовов в Qt Creator

Что же касается самостоятельного отладчика, то команда `backtrace` позволяет узнать, какими путями программа попала в данную точку. Для распечатки стека вызовов требуется просто написать:

```
(gdb) backtrace
```

В результате будет распечатан довольно длинный список имен функций и их адресов. Эти данные можно использовать для создания дополнительных точек останова, но перемещаться по исходным текстам проекта так, как в интегрированной среде, вы, естественно, не сможете.

Статический анализ кода C++

Если отладчики и им подобные инструменты позволяют находить ошибки во время выполнения программы, то инструменты статического анализа пытаются найти подозрительные места еще в исходном коде программы. Эффективность статических анализаторов зависит от пытливости ума программистов, которые их пишут. Ни один статический анализатор не способен обнаружить все ошибки в коде. Возможны также ложные тревоги, когда анализатор находит ошибочный код там, где на самом деле все правильно. Но если анализатор нашел настоящую ошибку, то он, как правило, способен точно локализовать ее и даже дать советы по ее устранению. Процедура статического анализа кода выполняется сравнительно быстро и может быть автоматизирована, а потому ее совсем нетрудно включить в стандартный цикл разработки программного обеспечения.

Лучше всего статические анализаторы проявляют себя при обнаружении ошибок, ведущих к утечкам каких-либо ресурсов. К таковым относятся, например, утечки памяти, возникающие, когда указатель, инициализированный с помощью `new`, выходит из области видимости, без вызова `delete`. Статический анализ может обнаружить также выход индекса за границы массива (в том случае, если эти границы определены статически) или утечку других ресурсов (например, выход переменной, хранящей дескриптор файла, из области видимости без вызова функции `close()`) и тому подобное.

Мы рассмотрим открытую систему статического анализа кода cppcheck (**http://cppcheck.sourceforge.net**). У среды Eclipse есть специальный модуль для интеграции cppcheck (такие модули есть и для некоторых других интегрированных сред, которые мы в этой книге не рассматриваем). Впрочем, ничто не мешает вам использовать cppcheck в отдельности от какой-либо интегрированной среды. Программа cppcheck доступна как для Linux, так и для Windows. При этом стоит отметить, что, поскольку cppcheck проверяет корректность кода с точки зрения C++, результат проверки не зависит ни от платформы, на которой вызвана программа cppcheck, ни от платформы, для которой предназначен код C++.

Программа cppcheck постоянно обретает новые возможности, так что к моменту, когда вы будете читать эту книгу, cppcheck, возможно, будет уметь что-то из того, что она не умеет сейчас. К полезным возможностям cppcheck я отношу способность находить в коде неиспользуемые функции (которые могут остаться после слишком лихорадочного программирования или неумелого рефакторинга), неиспользуемые методы классов, объявленные в разделе `private`, поиск ошибок, возникающих при использовании шаблонов STL, а также способность "разворачивать" различные макросы, в том числе управляющие ходом компиляции приложений.

Кроме всего перечисленного cppcheck позволяет вам создавать собственные правила для проверки синтаксиса кода C++. Вот как, например, выглядит файл правил (листинг 4.2) для проверки наличия в коде конструкции:

```
if(foo) {delete foo}
```

где `foo` — указатель на что-либо, выделенное с помощью `new`. Эта конструкция не является ошибочной, но она избыточна, т. к. передача оператору `delete` указателя, имеющего значение `NULL`, не вызывает ошибки.

Листинг 4.2. Файл правил для программы cppcheck

```
<?xml version="1.0"?>
<rule version="1">
  <pattern>if \( p \) { free \( p \) ; }</pattern>
  <message>
    <id>redundantCondition</id>
    <severity>style</severity>
    <summary>Redundant condition.
         It is valid to free a NULL pointer.</summary>
  </message>
</rule>
```

Здесь мы задаем регулярное выражение для поиска конструкции, уровень "опасности кода" (указание `style` означает, что мы имеем дело не с ошибкой, а с нарушением правил "хорошего стиля" программирования) и краткое описание проблемы.

Рассмотрим фрагмент вывода программы cppcheck для "настоящего" проекта (листинг 4.3). Для эксперимента использовались исходные тексты открытого проекта filezilla.

Листинг 4.3. Вывод программы cppcheck

```
[./src/interface/LocalTreeView.cpp:701]:
 (style) struct or union member 't_dir::dir' is never used
 [./src/interface/defaultfileexistsdlg.h:13] ->
 [./src/interface/queue.h:30]: (style) Variable 'download' hides
 enumerator with same name
 [./src/interface/Mainfrm.cpp:620]: (error) Null pointer dereference
 [./src/interface/QueueView.cpp:2009]:
 (style) Variable 'download' is assigned a value that is never used
[./src/interface/QueueView.cpp:2051]: (style) Variable 'download' is
 assigned a value that is never used
```

Для получения такого подробного отчета (представленные в листинге данные — лишь небольшой его фрагмент, полный отчет занял бы несколько страниц) команда была вызвана так:

```
cppcheck --verbose --enable=style
```

Читатель может возразить, что программа cppcheck не так уж и полезна, поскольку подобные предупреждения выдает любой хороший компилятор C++ в режиме "выдавать все предупреждения". На это возражение есть три контраргумента. Во-первых, программа cppcheck все же способна находить отдельные ошибки (и "помарки", вроде совпадения имени переменной и типа), которые пропускает хотя бы один компилятор из тех, что мы рассматриваем в этой книге. Во-вторых, получить полный список предупреждений компилятора можно только в процессе полной сборки проекта. Полная сборка требует гораздо больше времени, чем статический анализ кода, и программисты не прибегают к ней без необходимости. В-третьих, cppcheck может выполнять анализ кода по всем направлениям компиляции, определяемым управляющими макросами, тогда как в процессе сборки компиляция выполняется только по одному направлению.

Да и вообще, польза cppcheck нагляднее всего иллюстрируется приведенным выше перечнем ошибок и "помарок", обнаруженных в исходном коде солидного и зрелого приложения, готового к релизу.

Инструменты рефакторинга

Мартин Фаулер (Martin Fowler) определяет рефакторинг как "внесение в код программы таких изменений, которые упрощают его понимание и модификацию, но при этом не изменяют внешнего поведения программы" [5]. Важнейшей целью

рефакторинга является превращение небольшого количества методов, содержащих большие и запутанные фрагменты кода, в набор компактных методов, каждый из которых содержит сравнительно небольшой объем кода (причем код одного метода не повторяется в других методах), легко понятного программисту. Рефакторинг упрощает поиск ошибок, модификацию программы, повторное использование кода и многое другое.

Рефакторинг можно выполнять вручную, однако в большом проекте этот процесс может стать невероятно трудоемким. При этом творческая часть рефакторинга составляет лишь незначительный процент от общего объема работы, которую необходимо проделать, остальное — рутина, которую можно и нужно автоматизировать.

К сожалению, Qt Creator позволяет выполнять автоматически лишь самые элементарные операции рефакторинга. Будем надеяться, что положение улучшится в следующих версиях этой интегрированной среды разработки.

Примечание

Если вы попытаетесь поискать независимые средства рефакторинга для C++, то увидите, что в основном они привязаны к тому или иному средству разработки. Это не удивительно, поскольку для выполнения многих операций рефакторинга необходимо сначала произвести синтаксический разбор программы, а эту задачу проще "перепоручить" тому инструменту, который уже обладает способностью ее выполнить. Тем более странно, что в Qt Creator до сих пор нет полноценного рефакторинга, ведь синтаксический разбор текста программы на C++ эта среда делать умеет.

Мы рассмотрим средства рефакторинга, которые доступны в виде надстроек Microsoft Visual Studio и среды Eclipse, настроенной на работу с C++.

Рефакторинг в Microsoft Visual Studio

Сама по себе среда Microsoft Visual Studio небогата средствами рефакторинга, но для нее существует несколько надстроек, которые позволяют автоматизировать многие операции рефакторинга и не только. Одна из самых удобных и функциональных надстроек — Visual Assist X компании Whole Tomato Software (**http://www.wholetomato.com**). Помимо собственно инструментов рефакторинга Visual Assist X предоставляет нам множество дополнительных удобных функций, например, улучшенные средства автоматического завершения имен идентификаторов (IntelliSense), поиска имен в проектах и многое другое.

Для изучения рефакторинга в Visual Assist X мы воспользуемся тем же классом `Calculator`, что и в предыдущей главе. Попробуем снова выделить метод `equalPressed()` из метода `operationButtonPressed()` (разумеется, для этого нужно использовать исходный, необработанный вариант класса `Calculator`).

Одна из важнейших операций рефакторинга — выделение метода (method extraction). Эта операция важна не только сама по себе. Она также является составной частью многих других, более сложных, операций рефакторинга. Одной из очевидных ситуаций, когда выделение нового метода может оказаться полезным, явля-

ется удаление из методов класса повторяющегося кода (этот код просто выделяется в свой собственный метод). Другая ситуация, в которой полезно прибегнуть к выделению метода, — упрощение метода со слишком сложной внутренней структурой. Наш пример с классом `Calculator` условно подпадает под этот случай. Я пишу "условно" потому, что методы класса `Calculator` трудно назвать уж очень сложными.

> ***Примечание***
>
> Как отмечает Мартин Фаулер в книге [4], преимущества рефакторинга трудно продемонстрировать на "книжных" примерах кода. Сложные примеры, в которых польза рефакторинга становится очевидной, в книге просто не поместятся, а рефакторинг на простых примерах не очень впечатляет читателя. Фаулер пишет, что от читателя требуется фантазия, чтобы представить себе, как то, что демонстрируется на простых примерах, будет работать в сложных случаях. Я присоединяюсь к его мнению.

Серьезная проблема, с которой мы сталкиваемся в процессе выделения метода, — проблема передачи параметров. Когда фрагмент будущего нового метода находится в теле своей исходной функции, ему потенциально доступны все переменные, объявленные в соответствующей области видимости (хотя, скорее всего, он использует не все из них). С точки зрения выделения фрагмента кода в отдельный метод переменные, используемые в этом фрагменте, можно разделить на три группы:

- переменные, значения которых присваиваются за пределами фрагмента и затем используются фрагментом;
- переменные, которым фрагмент присваивает значения и которые потом используются другими фрагментами;
- переменные, которые инициализируются и используются внутри данного фрагмента.

При выделении фрагмента в отдельный метод значения переменных первой группы следует передавать методу как параметры (или сделать их доступными новому методу другим способом). Значения, присваиваемые переменным второй группы, следует рассматривать как результат вызова нового метода, а переменные третьей группы — как локальные переменные метода. При реализации этих, внешне простых правил, можно столкнуться с некоторыми трудностями, связанными с выделением локальных переменных, а также с тем случаем, когда результаты работы метода возвращаются в нескольких переменных. Впрочем, решение этой задачи может быть формализовано, что, помимо прочего, и делает Visual Assist X. Если мы выделим в методе `operationButtonPressed()` фрагмент кода, который выполняется, если параметр `button` равен `opEqual` (листинг 4.4), и щелкнем по выделенному фрагменту правой кнопкой мыши, откроется контекстное меню. Если в вашей среде разработки установлен Visual Assist X, то контекстное меню будет включать в себя дочернее меню **Refactor (VA X)** (Выполнить рефакторинг с помощью VA X). В этом, весьма обширном, меню вам будет доступна команда **Extract Method...** (Выделить метод). Если выбрать эту команду, откроется диалоговое окно (рис. 4.6), в котором вы сможете указать имя нового метода.

Листинг 4.4. Код для выделения метода

```
if (button == opEqual) {
        switch (operation) {
            case opPlus:
              fgValue += bgValue;
              break;
            case opMinus:
              fgValue = bgValue — fgValue;
        break;
            default:;
          }
          mode = CopyMode;
          bgValue = 0;
 }
```

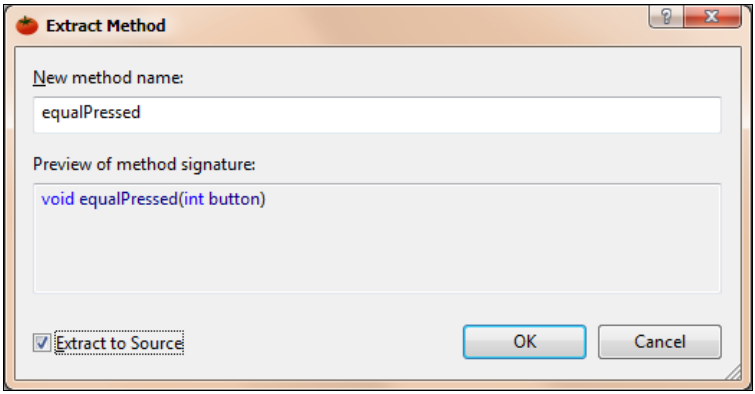


Рис. 4.6. Диалоговое окно извлечения метода

В нижней части окна вы сможете посмотреть заголовок метода, который будет создан в результате рефакторинга. Поскольку в выделенном фрагменте кода используется значение переменной `button`, присвоенное этой переменной за пределами фрагмента, Visual Assist X определил эту переменную как параметр нового метода (что верно с точки зрения рефакторинга). Но, как бы ни был хорош Visual Assist X в анализе кода C++, думать за нас он не станет. Получившийся в результате метод `equalPressed()` выглядит не очень элегантно (листинг 4.5), т. к. сразу же начинается с оператора `if()`. Если условие оператора не выполнено, метод не будет делать ничего. Здравый смысл подсказывает нам, что проверку `button == opEqual` лучше оставить там, где она была, а в метод `equalPressed()` включить только код, который будет выполняться в том случае, если выполнено условие проверки.

Дело в том, что с точки зрения ресурсов процессора вызов метода стоит гораздо дороже, чем проверка значения переменной `button`, а в нашем (неудачном) варианте рефакторинга вызов метода `equalPressed()` выполняется независимо от того, будет этот метод делать что-нибудь или нет.

Здравый смысл подсказывает также, что виноваты в сложившейся ситуации только мы сами, поскольку мы изначально выделили для рефакторинга фрагмент кода, включающий в себя проверку значения `button`.

Листинг 4.5. Результат рефакторинга

```
void Calculator::equalPressed( int button )
{
    if (button == opEqual) {
        switch (operation) {
case opPlus:
    fgValue += bgValue;
    break;
case opMinus:
    fgValue = bgValue — fgValue;
default:;
        }
        mode = CopyMode;
        bgValue = 0;
    }
}
```

Собственно говоря, "неправильное" выделение кода понадобилось нам для того, чтобы продемонстрировать способность Visual Assist X анализировать код и находить входные параметры.

Рассмотрим кратко некоторые другие полезные возможности надстройки. Команда **Create Implementation...** (Создать реализацию) позволяет автоматически создать заготовку метода, объявленного в заголовочном файле, в файле C++. Впрочем, в Qt Creator существует не менее удобная опция: в файле C++ можно набрать имя класса, потом два двоеточия и будет раскрыт список методов класса, в том числе и методов, объявленных в заголовочном файле. Команда **Encapsulate Field...** (Инкапсулировать поле) выполняет инкапсуляцию выделенного поля, т. е. автоматически создает метод-геттер и метод-сеттер для этого поля.

Еще одна удобная опция Visual Assist X — команда **Change Signature...** (Изменить заголовок метода). Эта команда позволяет быстро и просто изменить заголовок метода, который уже объявлен в заголовочном файле и в файле C++ (вы меняете заголовок метода в одном из файлов, и он автоматически меняется в другом).

Рефакторинг в среде Eclipse

Если у вас есть среда Eclipse, настроенная на работу с C++ и Qt, то инструменты рефакторинга в ней уже установлены (меню **Refactoring** (Рефакторинг) в главной строке меню). Средства рефакторинга Eclipse не так многообразны, как средства Visual Assist X, но выполнять базовую операцию выделения метода Eclipse умеет. Посмотрим, как он это делает. После того как мы выделяем требуемый фрагмент

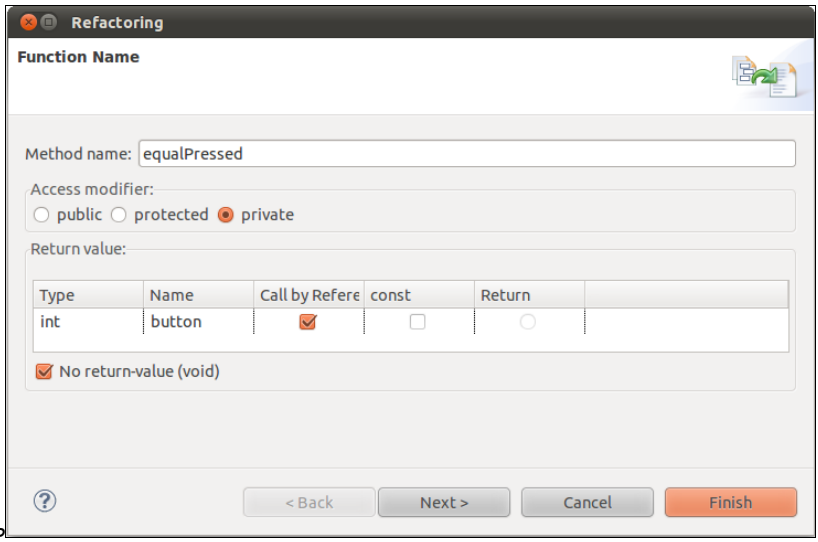


Рис. 4.7. Диалоговое окно извлечения метода в Eclipse

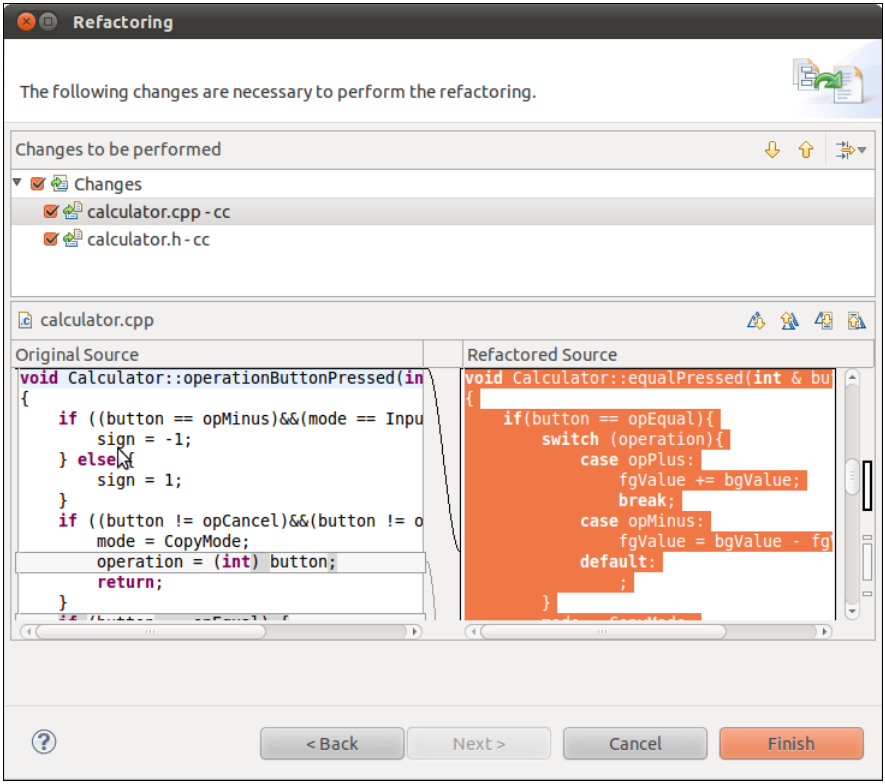


Рис. 4.8. Предварительный просмотр результатов рефакторинга

кода и вызываем команду **Extract method** (Выделить метод), перед нами открывается диалоговое окно, которое позволяет задать имя нового метода. Это окно не похоже на окно, которое мы видели в настройке Visual Assist X (рис. 4.7). Инструмент рефакторинга Eclipse тоже проанализировал код и "догадался" о том, что новому методу потребуется входной параметр. Однако далее нам предлагается гораздо более тонкая настройка. Мы можем выбрать способ передачи параметра в метод (по значению или через указатель). Мы также можем указать, в каком разделе класса — `public`, `private` или `protected` должен располагаться новый метод. Обычно выделенный метод располагается в разделе `private`, т. к. он вряд ли будет востребован как часть публичного или наследуемого интерфейса класса.

Если мы хотим сделать новый метод более доступным, то можем указать это непосредственно в окне рефакторинга. Щелчок по кнопке **Next** (Далее) откроет новое окно, в котором будет показан текущий код и код после применения рефакторинга (рис. 4.8). В верхней части окна представлен список файлов, затронутых изменениями. Мы можем запретить изменения в некоторых файлах, хотя обычно этого делать не требуется. Ниже мы видим две текстовых области, в которых показан, собственно, оригинальный и измененный код. Если мы видим, что рефакторинг делает не то, что нам нужно, мы можем отказаться от всей операции, щелкнув кнопку **Cancel** (Отмена). В противном случае щелкаем **Finish** (Завершить).

Рефакторинг "вручную"

Существуют ситуации, когда автоматизированный рефакторинг помочь не в силах. Рассмотрим пример (листинг 4.6).

Листинг 4.6. Нетривиальный случай рефакторинга

```
int DataTable::columnIndex( const QString &column ) const
{
    for (int i = 0; i < columnCount(); i++) {
        if (headerData(i, Qt::Horizontal).toString() == column)
            return i;
    }
    return -1;
}
```

Что именно делает метод `columnIndex()`, нам сейчас знать необязательно. Этот метод интересует нас, как пример для рефакторинга. Конечно, метод `columnIndex()` слишком прост, чтобы применить к нему рефакторинг в реальной жизни, но этот пример демонстрирует одну вполне реальную проблему. Проблема связана с оператором `return`.

Выражение

```
return i;
```

заставляет программу выйти из метода `columnIndex()`. Теперь допустим, что мы решили выделить цикл `for()` в отдельный метод (довольно частая операция рефак-

торинга) и воспользовались для этого автоматическим выделением. В результате у нас получится метод `newMethod()` (листинг 4.7).

Листинг 4.7. Результат автоматизированного рефакторинга

```
int DataTable::newMetod(const QString &column)
{
for (int i = 0; i < columnCount(); i++) {
        if (headerData(i, Qt::Horizontal).toString() == column)
            return i;
    }
}
```

С методом `newMethod()` возникает сразу несколько проблем. Во-первых, мы видим, что значение, возвращаемое методом, может быть не определено (в том случае, если переменная `i` достигнет значения `columnCount()`). Некоторые компиляторы C++ рассматривают эту ситуацию как непреодолимую ошибку, другие ограничиваются предупреждением, но, в любом случае, это очень плохой стиль программирования. Однако на этом проблемы не заканчиваются. Посмотрим, как будет выглядеть после такого рефакторинга метод `columnIndex()` (листинг 4.8).

Листинг 4.8. Вызов извлеченного метода

```
int DataTable::columnIndex( const QString &column ) const
{
    newMethod(column);
    return -1;
}
```

Теперь нам понятно, что даже если мы исправим ошибку в методе `newMethod()`, метод `columnIndex()` все равно будет работать неправильно, поскольку оператор `return`, который раньше возвращал программу из метода `columnIndex()`, больше не делает этого (он теперь спрятан внутри метода `newMethod()`). В новой редакции метод `columnIndex()` всегда будет возвращать –1. Поскольку в C++ нет операции "выйти из той функции, которая вызвала данную функцию", нам придется редактировать метод `columnIndex()` (листинг 4.9).

Листинг 4.9. Корректный рефакторинг

```
int DataTable::columnIndex( const QString &column ) const
{
    int result = newMethod(column);
    if (result >= 0)
        return result;
```

```
    else {
        // Выполнить какие-нибудь полезные действия
        return -1;
}
}
```

Рефакторинг и производительность

Целью рефакторинга является сделать код более понятным и упростить его модификацию. Создание высокооптимизированного быстродействующего кода не входит в список приоритетных задач рефакторинга. Означает ли это, что рефакторинг снижает производительность кода? Практически нет. Современные компиляторы C++ очень хороши в деле оптимизации кода (причем некоторые оптимизации специально рассчитаны на код, прошедший рефакторинг). Так что когда вы компилируете программу не для отладки, а для релиза, с включенной оптимизацией, многие неэффективные элементы, введенные в программу в процессе рефакторинга, просто исчезают из результирующего кода.

Например, М. Фаулер рекомендует разбивать сложные математические выражения на небольшие блоки, причем результат вычисления каждого блока присваивается отдельной переменной с осмысленным названием. Таким образом, сложное выражение становится более понятным. Рассмотрим упрощение сложных выражений на примере настолько простом, что его, возможно, и не следует упрощать (листинг 4.10).

Листинг 4.10. Функция с временными переменными

```
long double getTotalPayment(long double currentSum, long double newSum)
{
    long double newSumWithVAT = newSum*1.18;
    long double total = currentSum + newSumWithVAT;
    return total;
}
```

При включенной оптимизации (GCC с ключом `-O2`) в результирующем коде не остается никаких следов переменных `newSumWithVAT` и `total` (листинг 4.11).

Листинг 4.11. Оптимизированный результат компиляции

```
pushl   %ebp
movl    %esp, %ebp
fldt    8(%ebp)
fldt    20(%ebp)
popl    %ebp
fmull   .LC0
faddp   %st, %st(1)
ret
```

Если вы не сильны в чтении ассемблерных листингов, то просто сравните приведенный код с тем, как преобразуется в код выражение `return currentSum + newSum*1.18;` (листинг 4.12), и подсчитайте количество операций микропроцессора.

Листинг 4.12. Результат компиляции оптимизированного выражения

```
pushl   %ebp
movl    %esp, %ebp
fldt    20(%ebp)
fmull   .LC0
fldt    8(%ebp)
popl    %ebp
faddp   %st, %st(1)
ret
```

Три строчки кода на C++ при оптимизации производят столько же машинного кода, сколько и одна строчка. Так что если три строчки делают код более читабельным, нет смысла экономить.

Системы контроля версий

Чем дальше заходит написание программы, тем осторожнее мы становимся при внесении в нее изменений. Нам требуется инструмент, который позволял бы вернуть исходные тексты к тому состоянию, в котором они были час назад, вчера, неделю назад, но сохранял при этом все изменения, сделанные с тех пор. Если над проектом работает несколько человек, нам хочется отслеживать изменения, внесенные каждым разработчиком отдельно. Иначе говоря, кода мы работаем над сложным проектом, нам необходим инструмент, который мог бы восстановить историю всех изменений исходного кода во времени и "в пространстве" (т. е. с учетом изменений, которые одновременно вносят разные разработчики). Именно таким инструментом и являются системы контроля версий. Некоторые считают, что использовать системы контроля версий имеет смысл только в проектах, над которыми работает большое количество человек. Мое же мнение заключается в том, что системы контроля версий необходимы даже в том случае, когда над проектом работает только один человек.

Каждый, кто пользовался системой контроля версий на практике, может подтвердить, что она придает программисту огромную уверенность при внесении изменений в текст программы. Кроме того, системы контроля версий являются практически незаменимой составляющей рассмотренного выше процесса рефакторинга, такой же важной, как и система автоматизированного тестирования. Допустим, мы выполнили рефакторинг и провели автоматические тесты. Тесты показали, что после рефакторинга в коде появились ошибки, которых раньше не было. Самое разумное, что можно сделать в такой ситуации, — вернуть код в состояние, в котором он находился до рефакторинга, и попробовать новый подход (а может быть, и вовсе отказаться от рефакторинга).

Системы контроля версий можно сравнить с расширенной системой отмены и возврата операций (Undo/Redo), но между ними есть одно существенное различие. Дело не только в том, что системы контроля версий могут хранить изменения кода в течение многих лет, а системы отмены и возврата операций обычно "помнят" только те изменения, которые были сделаны во время работы программы. Системы контроля версий хранят все состояния исходного кода программы, которые мы зафиксировали в этих системах. Если мы вернемся к некоему состоянию, в котором код пребывал в прошлом, и продолжим разработку из этой точки, изменения, которые мы сделали перед тем, как вернуться к прежнему состоянию, не пропадут (если, конечно, мы их зафиксировали). И мы всегда сможем использовать эти изменения, если захотим. Другими словами, системы отмены и возврата операций всегда хранят линейную последовательность операций и "забывают" все ответвления. Системы контроля версий хранят все изменения, независимо от сложности структуры.

В настоящее время существует много систем контроля версий. Самые известные, пожалуй, это Subversion (сокращенно SVN), git (эта система используется разработчиками ядра Linux) и Bazaar. Все перечисленные системы контроля версий являются программами с открытыми исходными текстами. Кроме них, есть еще много других систем, как коммерческих, так и открытых. Компания Microsoft выпускает коммерческий пакет Visual Studio Application Lifecycle Management (ранее известный как Microsoft Visual Studio Team System), который, помимо системы контроля версий, содержит систему автоматического тестирования и многое другое.

Если вы не являетесь фанатичным поклонником продукции Microsoft, то я не вижу никаких причин, по которым вы не могли бы пользоваться открытыми системами контроля версий. Такие системы как Subversion, git и Bazaar зарекомендовали себя с самой лучшей стороны в условиях максимальной нагрузки (они используются в проектах, над которыми одновременно работают сотни человек, разбросанных по всему миру). По части удобства использования эти системы не уступают никаким своим аналогам. Все эти инструменты являются кросс-платформенными (даже git!), так что вы можете использовать их и под Windows, и под Linux, и в смешанных средах. Осталось ответить только на один вопрос.

Какую систему контроля версий выбрать?

Прежде всего, следует знать, что системы контроля версий делятся на централизованные и децентрализованные. В централизованных системах все изменения исходных текстов, зафиксированные в системе, сохраняются в одном центральном хранилище. В децентрализованных системах центрального хранилища не существует. Вместо этого есть множество локальных хранилищ, данные из которых объединяются по особым правилам. Примерами централизованных систем являются Subversion и CVS. К децентрализованным системам относятся, например, git и Bazaar. Если ваш проект не очень большой (не такой большой, как ядро Linux) и ваши партнеры редко работают над одним и тем же файлом исходного текста одновременно, то вам, скорее всего, будет удобнее воспользоваться централизованной системой контроля версий.

В этой книге мы рассмотрим одну систему контроля версий — Subversion. Почему именно Subversion? Во-первых, на данный момент Subversion является, пожалуй, самой популярной системой контроля версий (хотя в будущем, возможно, уступит это место git). Такие среды разработки, как Qt Creator и Eclipse, легко интегрируются с Subversion. Microsoft традиционно не поддерживает некоммерческие системы контроля версий, однако для Microsoft Visual Studio существуют сторонние надстройки, которые позволяют интегрировать Visual Studio и Subversion. Примером является открытая надстройка AnkhSvn, которую мы рассмотрим далее. Также под Windows и Linux существуют очень удобные графические клиенты Subversion, так что использовать эту систему контроля версий можно и без интеграции со средой разработки.

Примечание

Между прочим, системы контроля версий можно использовать не только для сохранения изменений в исходных текстах программ. Я, например, использую Subversion для сохранения текстов этой книги, которые я пишу в редакторе Microsoft Word.

Subversion

Как и любая централизованная система контроля версий, Subversion (**http://subversion.tigris.org**) состоит из клиентской и серверной части. Мы рассмотрим ситуацию, когда сервер Subversion установлен на компьютере под управлением Linux, а клиенты работают как под Linux, так и под Windows. Сервер Subversion можно установить и на компьютере под управлением Windows. При этом Subversion для Windows предоставляет графический интерфейс управления сервером. Если вы научитесь управлять сервером Subversion для Linux с помощью консоли, то без труда освоите и интерфейс сервера для Windows. Предполагается, что помимо сервера Subversion на компьютере установлен сервер OpenSSH.

Работа с Subversion начинается с создания репозитория, который будет хранить наши проекты. Репозиторий создается командой:

```
svnadmin create /usr/local/repo
```

Утилита svnadmin служит для управления сервером. Команда `create` указывает, что мы хотим создать новый репозиторий. Далее следует путь к директории, в которой будет храниться репозиторий. Эту директорию нужно создать заранее, и она должна быть пустой. Директория может быть расположена, где угодно, в том числе в домашней директории одного из пользователей.

Наличие сервера OpenSSH позволяет использовать ссылки на репозиторий, начинающиеся с префикса `svn+ssh://`. В этом случае каждый пользователь, имеющий учетную запись на сервере Linux, сможет получить доступ к серверу Subversion, используя свой логин и пароль.

Примечание

Другие способы доступа к серверу Subversion включают в себя доступ непосредственно к серверу, с использованием встроенных механизмов аутентификации пользователей (при таком доступе ссылки на репозиторий начинаются с префикса `svn://`) и с

помощью web-сервера (в том случае ссылки начинаются с префикса `https://` и web-сервер должен быть настроен на поддержку соответствующего протокола). Преимущество обоих этих методов перед описанным заключается в том, что при добавлении нового пользователя Subversion для него не нужно создавать учетную запись на сервере. Это может быть удобно, если количество пользователей Subversion очень велико.

Для того чтобы пользователи могли что-то изменить в репозитории Subversion, они должны иметь доступ к файлам репозитория. Если вы — единственный пользователь Subversion, достаточно разместить репозиторий в вашей домашней директории. Если доступ к репозиторию планируется организовать для нескольких пользователей, необходимо создать специальную группу, сделать ее владельцем файлов репозитория и предоставить ей разрешения на чтение и запись в репозиторий, а затем всех пользователей, которым необходим доступ к серверу Subversion, следует сделать членами этой группы.

После того как репозиторий создан и доступ пользователей к нему налажен, в репозиторий можно импортировать проекты. Импортировать проект можно как непосредственно на сервере, так и с клиентского компьютера. На сервере импорт проекта выполняется командой:

```
svn import myproject file:///usr/local/repo/myproject -m
 'initial project'
```

В этой команде `myproject` — имя директории проекта, за которым следует ссылка на директорию репозитория. Ключ `-m` позволяет создать аннотацию. Команда `svn` считает, что вы должны аннотировать каждую операцию добавления данных в репозиторий, поэтому, если вы забудете добавить ключ `-m` и текст, то откроется окно редактора для ввода текста.

Структура директорий проекта Subversion

Вообще говоря, вы можете структурировать директории проекта, как вам угодно, но если вы хотите, чтобы другие пользователи, имеющие опыт работы с Subversion, понимали структуру вашего проекта, необходимо следовать некоторым несложным правилам. Директория вашего проекта должна содержать три поддиректории: trunk, branches и tags.

Директория trunk (ствол) является главной. В ней ведется основная работа над проектом и она содержит все изменения исходных текстов вплоть до самых последних.

Директория branches (ветви) содержит зафиксированные состояния исходных текстов, которые вы хотите выделить из директории trunk. Чаще всего выделение новой ветви происходит потому, что проект в директории trunk "дозрел" до состояния, при котором можно выпускать очередной релиз. Для каждого релиза создается поддиректория директории branches, которая может иметь имя вида RB-x.x.x, где RB означает release branch (ветвь релиза), а x.x.x — номер версии релиза. Иногда поддиректории релизов состоят просто из номеров версий.

Для переноса текущего состояния директории trunk в директорию branches можно воспользоваться командой `svn copy`:

```
svn copy
file:///usr/local/repo/myproject/trunc
file:///usr/local/repo/myproject/branches/1.1.0
-m 'release 1.1.0'
```

Команда `svn copy` не станет копировать все содержимое директории trunk в новую директорию, а скопирует только то, что необходимо для создания "ветви".

Директория tags также используется для сохранения состояний репозитория на определенный момент. Разница между директориями tags и branches заключается в том, что в директории branches еще может вестись работа над текстами, тогда как код, помещенный в поддиректории директории tags, считается окончательно зафиксированным и больше не меняется.

Предполагается, что работа над проектом в директории trunk ведется непрерывно, а состояния, предназначенные для релиза, периодически переносятся в директорию branches, где над ними выполняются все специальные действия, необходимые для подготовки релиза. Таким образом, разработчики, работающие в директории trunk, могут не беспокоиться о том, что их действия нарушат подготовку очередного релиза. Когда релиз готов, соответствующий код копируется из директории branches в директорию tags и помечается как соответствующий релиз. Теперь внешние пользователи без труда найдут нужный им релиз в директории tags. Поток данных между директориями в процессе работы над проектом можно представить в виде, своего рода, конечного автомата (рис. 4.9). Помимо поддиректорий релизов, в директории branches могут размещаться обновления к уже сделанным релизам.

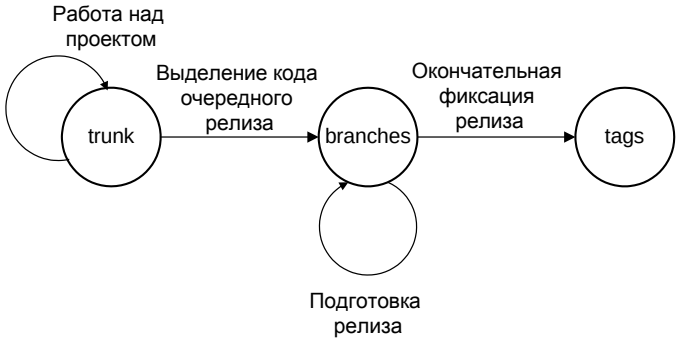


Рис. 4.9. Структура директорий Subversion и цикл разработки

Каждому изменению, зафиксированному в subversion, присваивается номер ревизии. Этот номер можно использовать как идентификатор состояния файлов репозитория на момент, когда изменения были применены. Если вы хотите выделить в директорию branches не текущее состояние директории trunk, а одну из прежних ревизий, можете воспользоваться такой командой:

```
svn copy SRC@REV DST
```

где `SRC` и `DST` — ссылки на директорию-источник и директорию-приемник соответственно, а `REV` — номер ревизии.

Необходимо пояснить, что ветвления в директории branches не имеют прямого отношения к тем ветвлениям ревизий, о которых говорилось ранее. Директория trunk хранит информацию обо всех ветвлениях ревизий, которые вы сделали. Существуют графические инструменты, которые позволяют наглядно показать дерево таких ветвлений. Работать с несколькими ветками проекта в директории trunk в принципе можно, но очень неудобно. Лучше выделить одну ветку в поддиректорию директории branches, например в branches/experimental.

Создание резервной копии репозитория

Уже само то, что исходные тексты программы, помимо компьютеров разработчиков, хранятся еще и на сервере Subversion, снижает вероятность потери данных. Однако если вы хотите надежно хранить всю информацию о проекте, зафиксированную в репозитории, желательно периодически создавать резервные копии репозитория. Самый простой способ выглядит так:

```
svnadmin dump /usr/loca/repo | gzip > svnbackup.svn.gz
```

Обратите внимание, что в этой команде используется не ссылка на директорию /usr/loca/repo, начинающаяся с префикса file://, а имя директории в файловой системе. Попутно мы сжимаем данные репозитория с помощью утилиты gzip. Результат сохраняется в файле svnbackup.svn.gz. Дополнительные параметры команды `svnadmin dump` позволяют создавать резервные копии не всего репозитория, а отдельных ревизий или интервалов ревизий.

Для восстановления репозитория из резервной копии служит команда `svnadmin load`, подробную информацию о которой вы найдете в справке по команде `svn`.

Для получения справок по командам утилит svnadmin и svn можно воспользоваться командой `help`. Например,

```
svnadmin help
```

распечатывает список всех команд утилиты svnadmin, а

```
svnadmin help load
```

распечатывает справку по конкретной команде `load`.

Справочная система утилиты svn работает аналогично.

Клиенты Subversion

Для большинства проектов не имеет особого значения, какая именно из современных систем контроля версий в них используется. Что действительно важно, так это наличие удобных средств взаимодействия с сервером. Их мы теперь и рассмотрим.

Программа svn

Все инструменты, описанные ниже, можно рассматривать как оболочки консольной программы svn (и, как правило, они действительно либо содержат фрагменты кода этой программы, либо используют ее как внешний процесс). Программа svn, с ко-

торой мы уже отчасти познакомились, включает в себя все необходимые команды для работы клиента с сервером Subversion. Все остальные средства лишь упрощают доступ к этим командам из конкретной среды. Поэтому, если вы понимаете, как работает svn, вы без труда поймете принципы работы любого другого клиента Subversion. Мы не станем описывать все команды программы svn, опишем только наиболее важные из тех, что не были рассмотрены выше.

Команда `svn checkout` извлекает исходные тексты из репозитория и копирует их в указанную директорию в вашей системе. По умолчанию команда извлекает самую свежую ревизию, но вы можете указать ей любую другую. Желательно, чтобы директория, в которую извлекаются данные с помощью `svn checkout`, изначально была пустой. После извлечения данных с помощью `svn checkout` эта директория попадает под управление системы контроля версий и в нем можно выполнять остальные команды svn. То, что создается в результате выполнения `svn checkout` в вашей системе, называется рабочей копией.

Команда `svn add` добавляет файл, директорию или группу файлов под контроль системы контроля версий. Добавляемые элементы должны располагаться в директории, которая уже находится под контролем Subversion.

Команда `svn commit` фиксирует сделанные вами изменения рабочей копии в репозитории Subversion. В результате появляется новая ревизия, которая в точности соответствует вашей рабочей копии.

Необходимо понимать разницу между добавлением файла под контроль Subversion и фиксацией файла в репозитории. Добавление файла под контроль Subversion — локальная операция. На сервере репозитория при этом ничего не происходит.

Команда `svn update` синхронизирует вашу рабочую копию с определенной ревизией репозитория (сохраняя при этом изменения, сделанные вами с момента последней фиксации вашей рабочей копии в репозитории).

Если другой разработчик внес изменения в те же области тех же файлов, что и вы, и зафиксировал эти изменения в репозитории раньше вас, при выполнении вами команды `svn update` возникнет конфликт, который нельзя разрешить автоматически. Такой же конфликт возникнет, если вы пытаетесь зафиксировать изменения, используя рабочую копию, которая конфликтует с изменениями, уже зафиксированными другими разработчиками. Программа svn позволяет разрешить такие конфликты на уровне сервера, но принять решение о том, какие именно из конфликтующих изменений следует зафиксировать на сервере, должны люди.

Команда `svn diff` позволяет сравнить текущее состояние файлов на сервере и файлов вашей рабочей копии, сравнить вашу рабочую копию и ревизию, из которой она была сделана, сравнить вашу рабочую копию с какой-либо другой ревизией и сравнить две ревизии между собой. Формат вывода команды `svn diff` аналогичен формату вывода утилиты GNU diff, на которой она и основана.

Примечание

Если вы используете исключительно ОС Linux, то для работы с Subversion вам, может быть, больше ничего и не нужно, кроме программы svn. Для того чтобы управлять

версиями файлов проекта, вовсе не обязательно внедрять поддержку контроля версий в используемую среду разработки, а работать с консольными программами в Linux гораздо удобнее, чем в Windows.

Программа svn перенесена и на платформу Windows (далее мы рассмотрим вариант одного из переносов), но использовать программу svn под Windows напрямую неудобно (точно так же, как и любую другую консольную программу).

Qt Creator и Subversion

Для работы с Subversion Qt Creator нуждается в программе svn[.exe]. Если вы работаете под Linux, то все необходимое у вас, скорее всего, уже есть (а если нет, то установите программу svn из репозитория вашего дистрибутива). Под Windows ожидать наличия svn.exe по умолчанию не приходится. Но такая программа для Windows есть, и даже не одна. Мы воспользуемся пакетом Slik Subversion (**http://www.sliksvn.com**). Теперь, если вы используете предложенную мной систему svn+ssh, вам придется немного поработать. Дело в том, что Slik Subversion не содержит программы для работы с SSH. Но такая программа есть в пакете Tortoise SVN, о котором будет сказано ниже, и который я рекомендую установить в любом случае. Допустим, что вы установили этот пакет. Тогда откройте файл C:\Users\UserName\AppData\Roaming\Subversion\config, найдите в нем строку, начинающуюся с `# ssh = $SVN_SSH`, замените ее на такую:

```
ssh = $SVN_SSH C:\\Program Files\\TortoiseSVN\\bin\\TortoisePlink.exe
-ssh
```

и сохраните файл. Теперь ваш клиент Subversion может работать с ссылками типа svn+ssh://.

Далее нам необходимо настроить Qt Creator на работу с SVN. Для этого выбираем команду меню **Инструменты | Параметры** и в диалоговом окне настроек выбираем раздел **Контроль версий**. Переходим на вкладку **Subversion** (рис. 4.10). На этой вкладке можно указать путь к клиенту Subversion (что мы и сделали), а также имя пользователя и пароль. При использовании схемы svn+ssh имя пользователя и пароль, веденные на этой вкладке, игнорируются. Кроме того, следует учесть, что введенный на этой вкладке пароль хранится в файле QtCreator.ini в незашифрованном виде, так что лучше оставить соответствующее поле пустым. Теперь мы можем создать проект, контролируемый системой Subversion.

Пусть этот проект называется testsvn. Создайте заготовку проекта (например, консольного приложения Qt) обычным образом. Пусть директория проекта именуется C:\Qt\testsvn. Теперь перейдите в директорию, которая содержит программу svn.exe, и выдайте такую команду:

```
svn import C:\Qt\testsvn svn+ssh://svnserver/usr/local/repo/testsvn
-m 'initial rivision'
```

Здесь `svnserver/repo/` — путь к вашему серверу Subversion. Если все прошло успешно, программа svn сообщит о добавлении нового проекта в репозиторий и присвоит порядковый номер соответствующей ревизии.

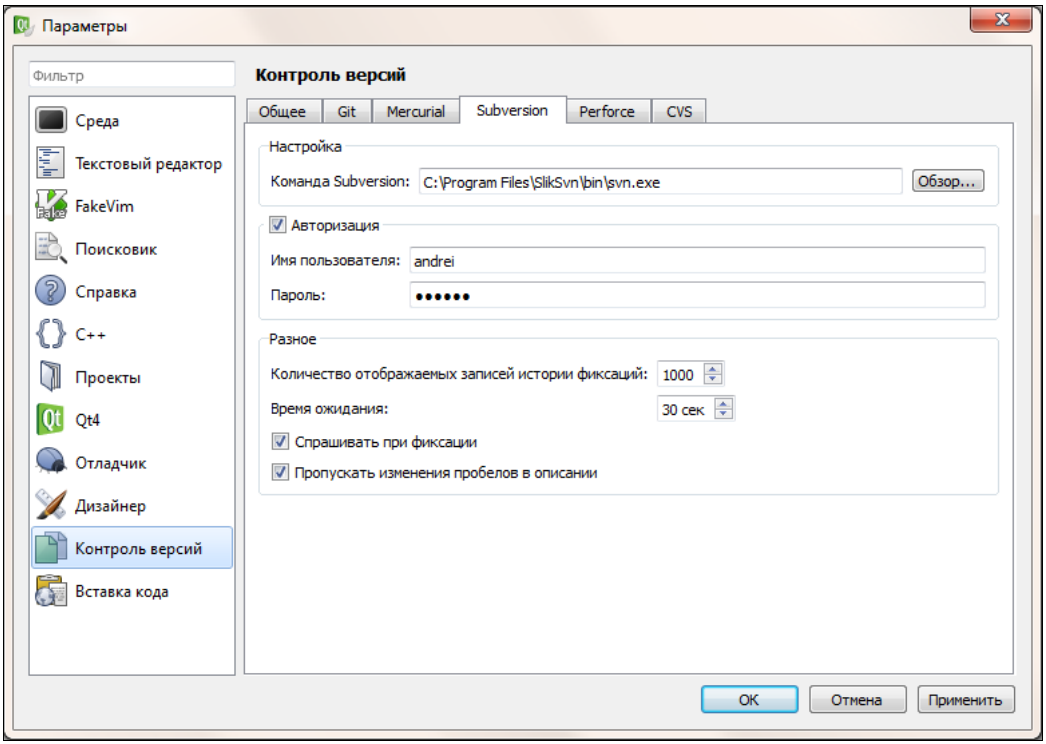


Рис. 4.10. Настройка Subversion в Qt Creator

Теперь нужно проделать нечто, на первый взгляд странное. Удалите все содержимое директории C:\Qt\testsvn, включая скрытую поддиректорию .svn. Далее в Qt Creator выберите команду **Файл | Новый файл или проект**. В открывшемся диалоговом окне перейдите на вкладку **Проект** из системы контроля версий и выберите пункт **Извлечь из Subversion** (рис. 4.11). Щелкните кнопку **Выбрать...**. В новом диалоговом окне (рис. 4.12) нужно указать путь к проекту на сервере репозитория и директорию локального размещения проекта (мы указываем ту же директорию, в которой проект был создан, вот почему нам потребовалось очищать ее). Щелкаем кнопку **Далее**. Система запрашивает пароль пользователя `username`, после чего происходит извлечение файлов проекта из репозитория Subversion в указанную директорию. Некоторая странность всех этих операций вызвана тем, что Qt Creator умеет только извлекать проекты из репозиториев Subversion и фиксировать изменения в них, добавлять же свои проекты в репозитории среда не умеет. Разумеется, если вам требуется просто загрузить из репозитория уже существующий там проект, все шаги, связанные с импортом проекта в репозиторий, следует пропустить.

Теперь, когда все мучения позади, мы можем работать с Subversion непосредственно из Qt Creator. В меню **Инструменты** Qt Creator появляется дочернее меню **Subversion** (рис. 4.13), которое содержит все необходимые команды для работы с системой контроля версий. Более того, при добавлении нового файла в проект в списке **Добавить под контроль версий** появляется пункт **Subversion** (по умолчанию там только Mercurial и git). Создается ощущение, что Qt Creator содержит все

необходимое для работы с subversion, но максимально усложняет для нас выбор этой системы контроля версий!

Для того чтобы изменения попали на сервер, их необходимо зафиксировать, что делает команда **Фиксировать...** меню **Subversion**. При выборе этой команды откроется окно фиксации, включающее в себя окно редактора аннотаций (это окно

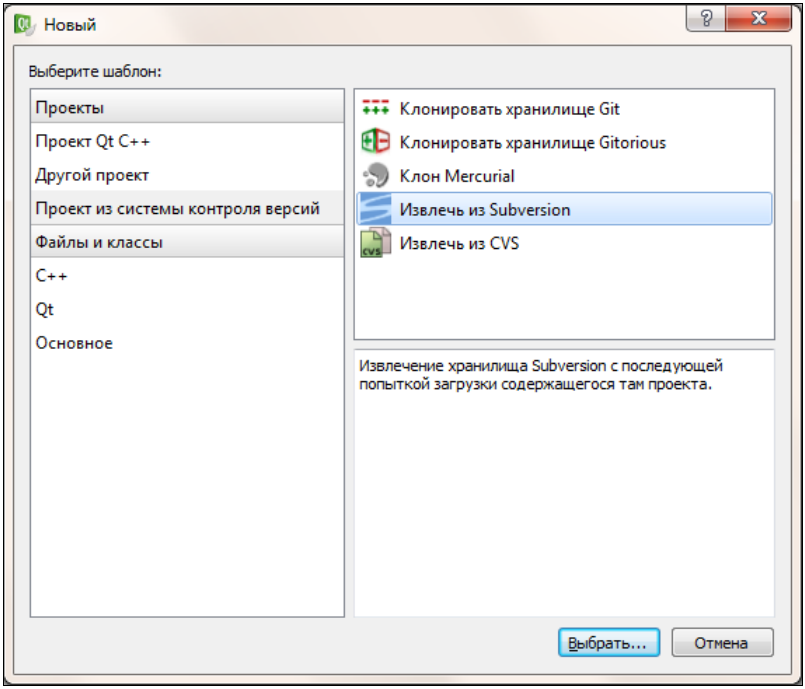


Рис. 4.11. Извлечение проекта Subversion в Qt Creator, шаг 1

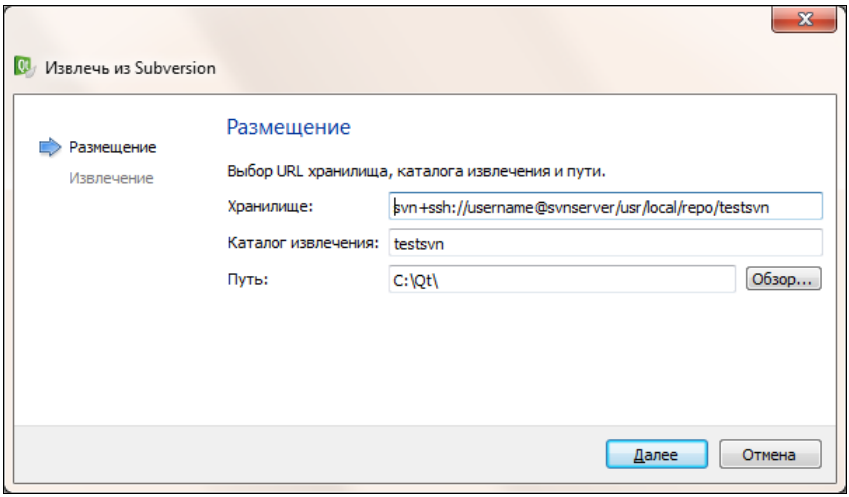


Рис. 4.12. Извлечение проекта Subversion в Qt Creator, шаг 2

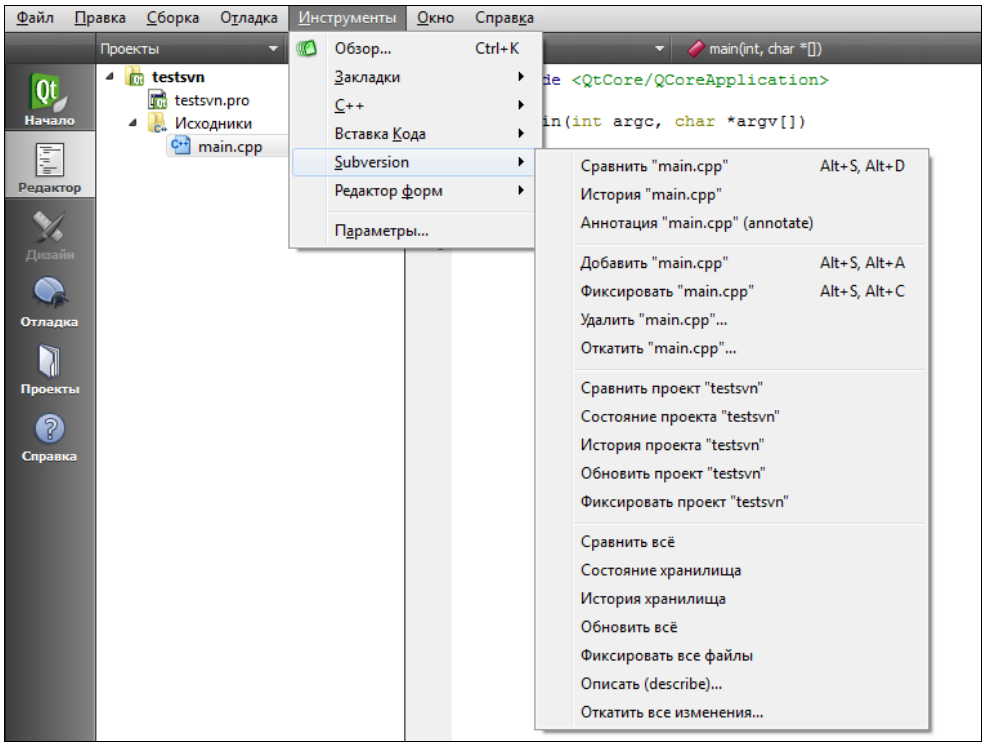


Рис. 4.13. Qt Creator с активированным меню Subversion

можно оставить пустым). При щелчке по кнопке **Фиксировать** система снова спросит пароль для входа на сервер Subversion, после чего будет выполнена сама фиксация. Если все прошло успешно, в окне сообщений системы контроля версий будет напечатано примерно следующее:

```
18:19 Выполняется в C:\Qt\testsvn: C:\Program Files\SlikSvn\bin\svn.exe
--username ******** --password ******** commit --non-interactive
--file C:/Users/Andrei/AppData/Local/Temp/qt_temp.wV4172 test.cpp
Adding         test.cpp
Transmitting file data .
Committed revision 486.
```

Последняя строка означает, что очередная ревизия зафиксирована.

Eclipse и Subversion

Для интегрированной среды Eclipse существует несколько проектов интеграции с Subversion. Мы рассмотрим проект Subversive. Как это часто бывает в проектах, связанных с Eclipse, Subversive существует в нескольких версиях. Лучше всего устанавливать надстройку из самой среды Eclipse, так, по крайней мере, выше вероятность, что вы не запутаетесь с версиями. После установки надстройки ее необходимо настроить (команда меню **Window** (Окно) | **Preferences** (Настройки)). В диалоговом окне настроек открываем группу **Team** (Группа) и подгруппу **SVN** (рис. 4.14). После того как взаимодействие с Subversion будет настроено, можно

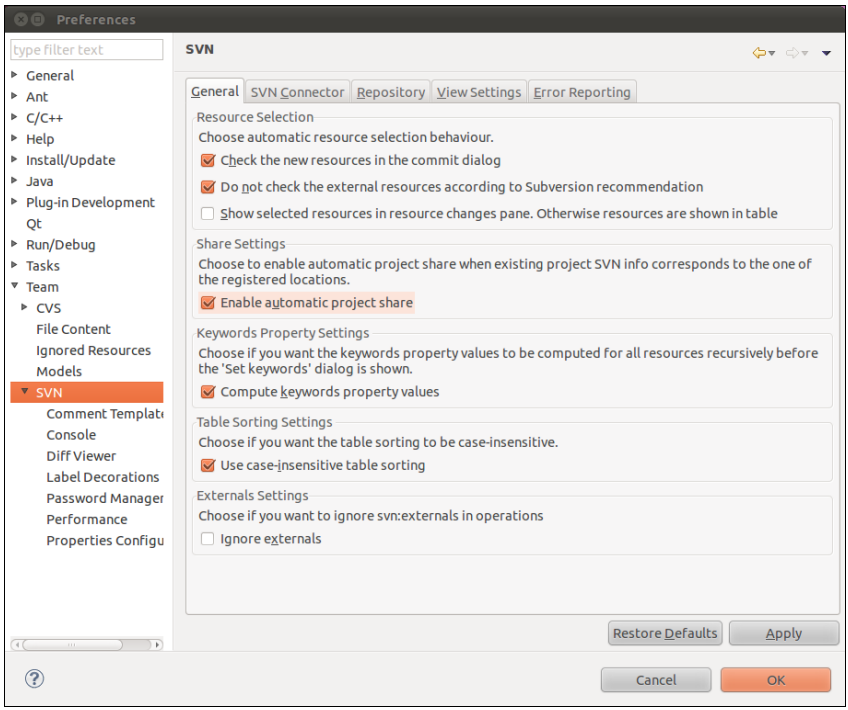


Рис. 4.14. Настройка Subversive

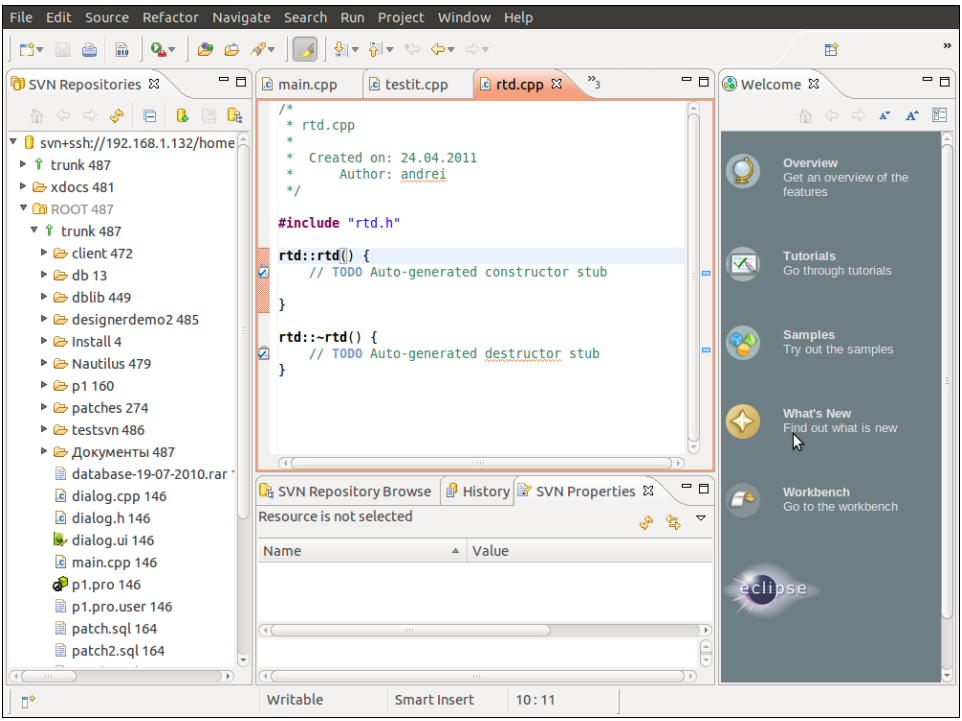


Рис. 4.15. Обзор репозиториев

открыть перспективу **Team** (Группа) и увидеть репозитории, к которым у нас настроен доступ (рис. 4.15).

> ***Примечание***
>
> В отличие от Qt Creator, Eclipse шифрует сохраненные пароли для доступа к серверу SVN, правда, стопроцентной гарантии безопасности не дает, о чем и предупреждает.

Для выгрузки из репозитория существующего там проекта лучше всего воспользоваться командой **Check Out As...** (Извлечь под именем) (рис. 4.16). Нам будет предложено выбрать рабочее пространство и набор рабочих настроек (рис. 4.17).

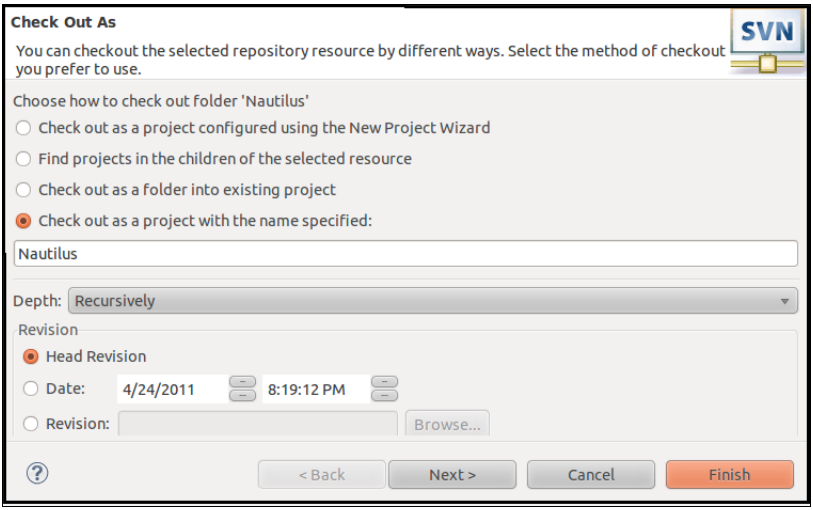


Рис. 4.16. Окно загрузки проекта на локальный компьютер

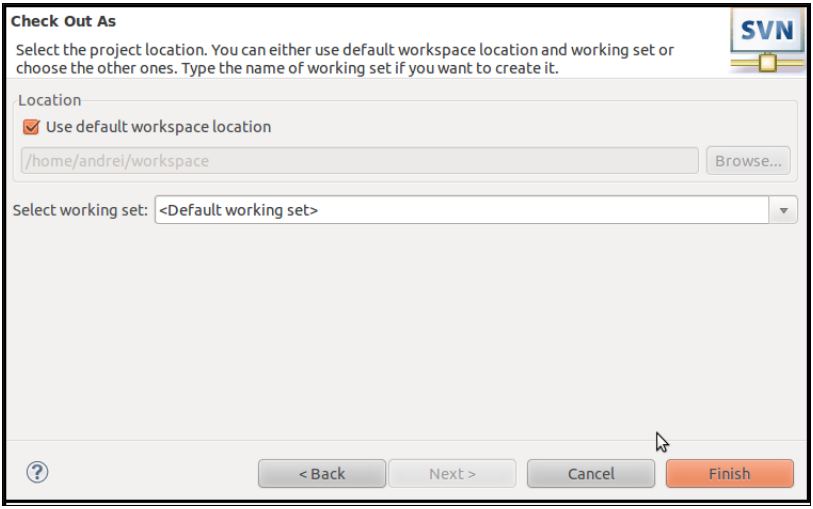


Рис. 4.17. Окно настроек рабочего пространства

Новый проект будет добавлен в соответствующую "перспективу". В нашем случае — Qt C++ (рис. 4.18). Символ массива дисков над значком папки указывает на то, что проект управляется системой контроля версий.

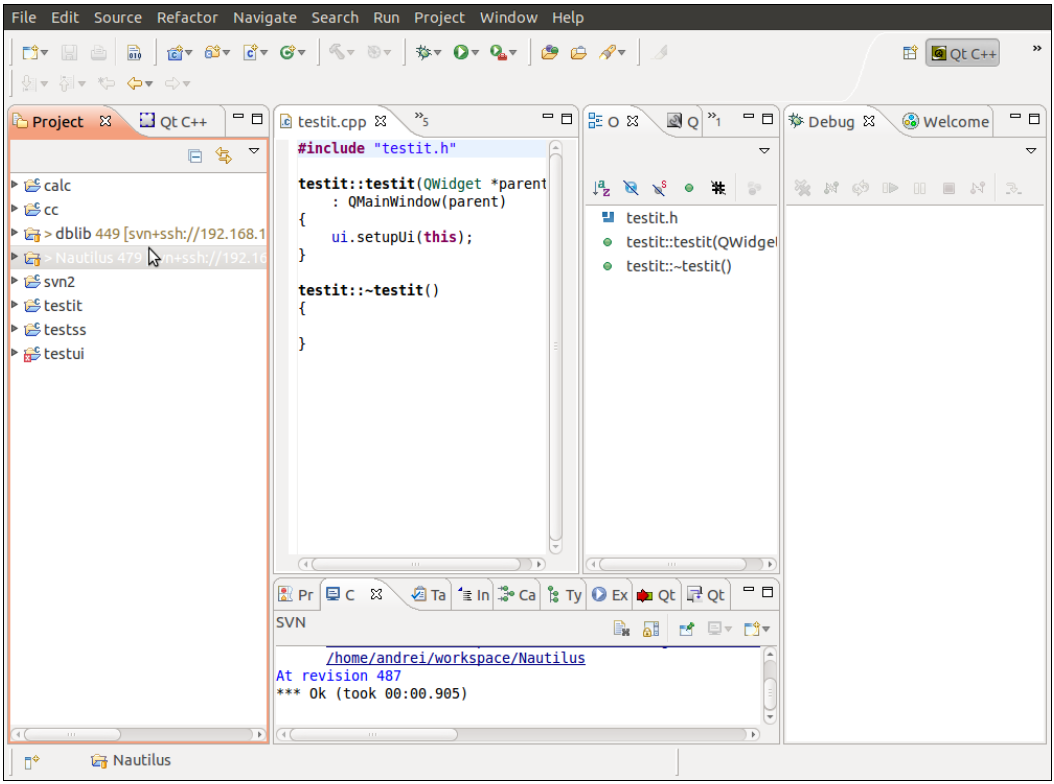


Рис. 4.18. Пространство Qt C++

Microsoft Visual Studio и Subversion

Как уже упоминалось, для Microsoft Visual Studio существует надстройка AnkhSvn, основанная на коде инструмента TortoiseSVN, о котором мы скажем ниже. Вы можете скачать дистрибутив этой надстройки бесплатно с сайта **http://ankhsvn.open.collab.net**. Из всех интегрированных средств SVN, рассмотренных до сих пор, AnkhSvn, пожалуй, самая удобная. AnkhSvn умеет работать со схемой svn+ssh "прямо из коробки", никаких мучений, как в случае с Qt Creator. Желаете добавить существующий проект в репозиторий Subversion? Проще простого! Щелкните правой кнопкой мыши по имени "решения" Microsoft Visual Studio и в открывшемся контекстном меню выберите пункт **Add Solution to Subversion...** (Добавить проект под контроль Subversion). В открывшемся диалоговом окне (рис. 4.19) вам предлагается ввести путь к репозиторию, в который следует добавить проект. Если ссылка на репозиторий корректна, вам будет предложено авторизоваться на сервере, после чего вы сможете пролистывать содержимое репозитория для выбора папки, в кото-

рую вы собираетесь импортировать проект. Тут же, с помощью кнопки **Create Folder...** (Создать папку) можно создать новую директорию. Из всех рассмотренных надстроек AnkhSvn — единственная, которая предлагает (но не обязывает!) создать в директории проекта на сервере Subversion поддиректорию trunk. В нижней части окна можно увидеть, как будет выглядеть полная ссылка на репозиторий вашего проекта.

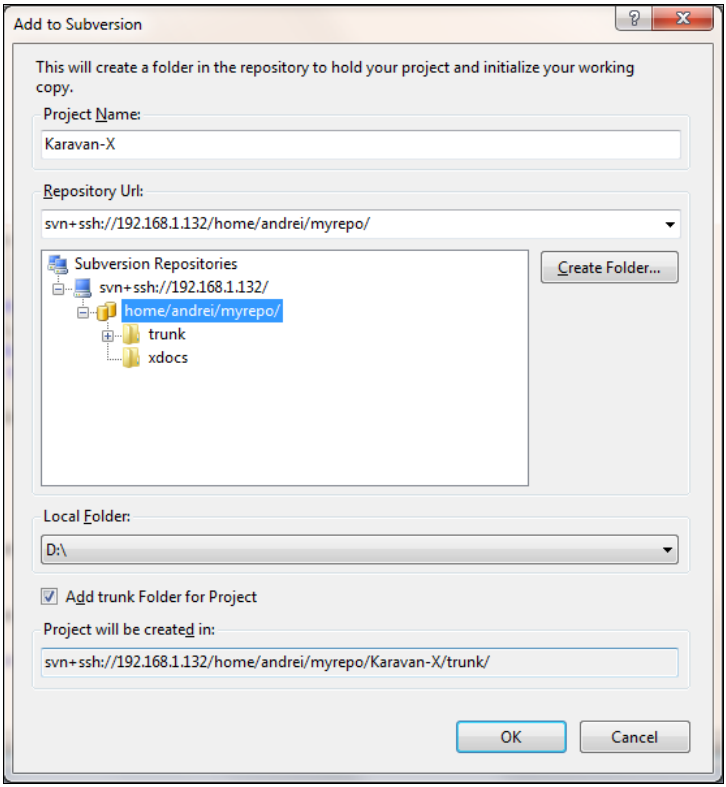


Рис. 4.19. Добавление нового проекта на сервер Subversion

Для извлечения из репозитория уже добавленного в него проекта следует воспользоваться командой **File** (Файл) | **Open from subversion...** (Открыть в репозитории Subversion) (рис. 4.20). Окно выбора проекта очень похоже на окно выбора файла на локальном диске, за исключением того, что навигация происходит по репозиторию Subversion.

Следующее окно (рис. 4.21) позволяет выбрать директорию репозитория (не выбирайте директорию высокого уровня, иначе AnkhSvn может извлечь слишком много!) и локальную директорию, в которую будет извлечен проект.

Когда файлы находятся под контролем Subversion, управлять ими можно из контекстного меню **Solution Explorer** (Обозреватель проекта) (рис. 4.22). Команды меню меняются в зависимости от состояния файла (файл добавлен под управление Subversion, изменен и т. д.).

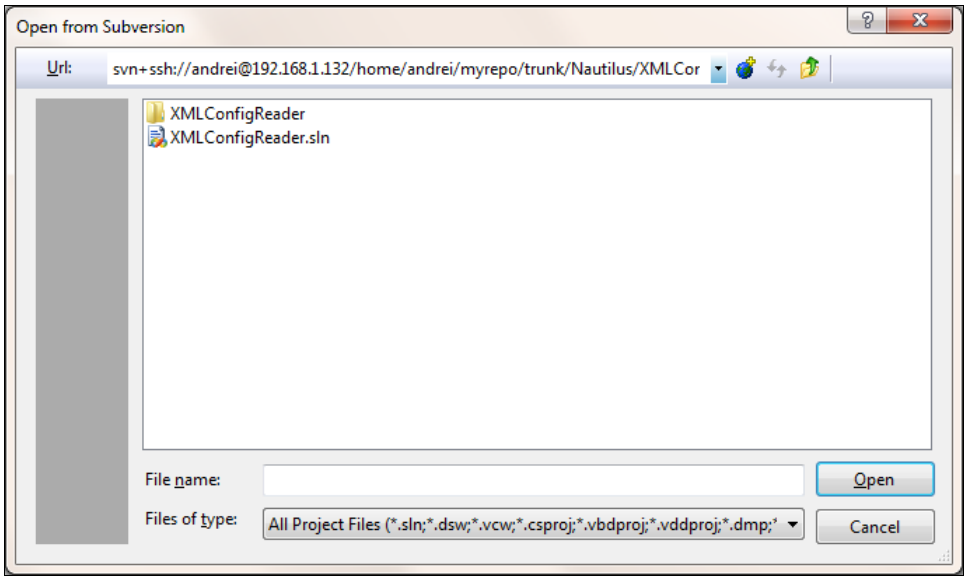


Рис. 4.20. Открытие проекта на сервере

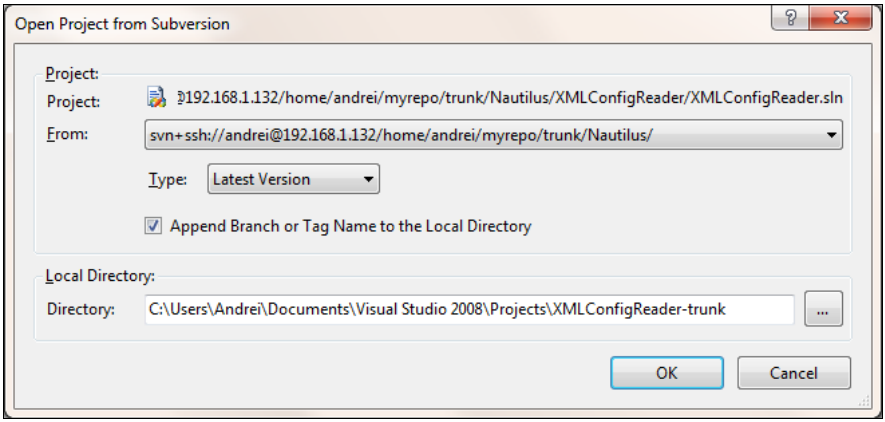


Рис. 4.21. Настройка параметров загрузки проекта

Весьма полезный инструмент открывается по команде меню **Show Changes...** (Показать изменения). Это графический аналог утилиты diff (рис. 4.23), который позволяет наглядно увидеть изменения, сделанные в файле, с момента его последней фиксации в репозитории.

Еще одна приятная особенность AnkhSvn заключается в том, что по умолчанию эта настройка добавляет под контроль Subversion только те файлы, которые нужно. В процессе сборки приложения Qt немало файлов исходных текстов генерируется автоматически (например, файлы с префиксами moc_ и ui_). Помещать под контроль Subversion эти файлы в общем случае не имеет смысла, т. к. они все равно регенерируются в процессе сборки. В Visual Studio эти файлы попадают в раздел

проекта **Generated Files** (Создать файлы), и AnkhSvn следит за тем, чтобы они не были помещены под контроль Subversion (в окне **Solution Explorer** эти файлы помечаются значком наподобие дорожного знака "Stop").

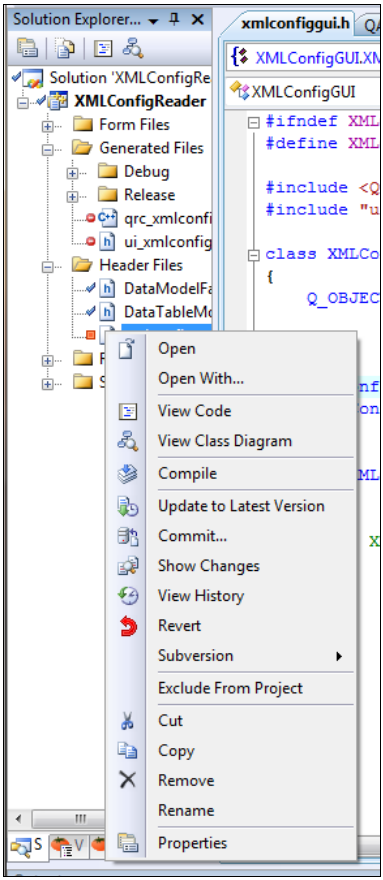


Рис. 4.22. Контекстное меню **Subversion** в окне **Solution Explorer**

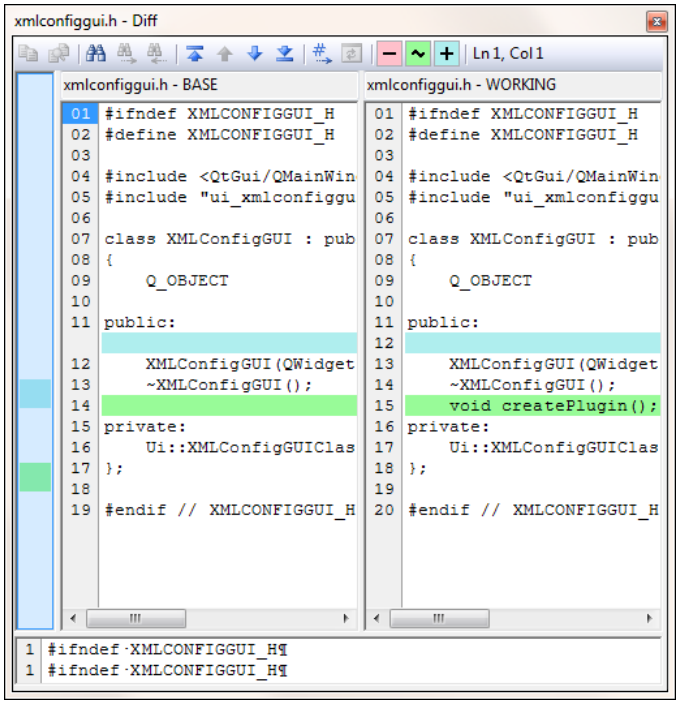


Рис. 4.23. Графический инструмент просмотра различий между файлами в AnkhSvn

TortoiseSVN

Если в Linux большинству пользователей достаточно консольной программы svn, то в Windows, как уже отмечалось, пользоваться этой программой неудобно. Если мы хотим применять Subversion независимо от различных средств разработки, нам необходим инструмент с удобным для пользователей Windows интерфейсом. Именно таким инструментом является бесплатный TortoiseSVN (**http://tortoisesvn.net**). TortoiseSVN интегрируется с оболочкой Windows, так что выполнять операции с файлами, находящимися под контролем Subversion, так же просто, как и выполнять обычные операции с файлами Windows. TortoiseSVN не содержит собственной программы svn (весь код скрыт в расширении оболочки), так что не

пытайтесь интегрировать его с Qt Creator. Зато TortoiseSVN включает в себя клиент SSH Plink.exe, который можно использовать с другими средствами Subversion, когда требуется схема svn+ssh://, а соответствующее средство не поддерживает SSH.

Все управление TortoiseSVN можно выполнять через контекстные меню оболочки Windows (рис. 4.24).

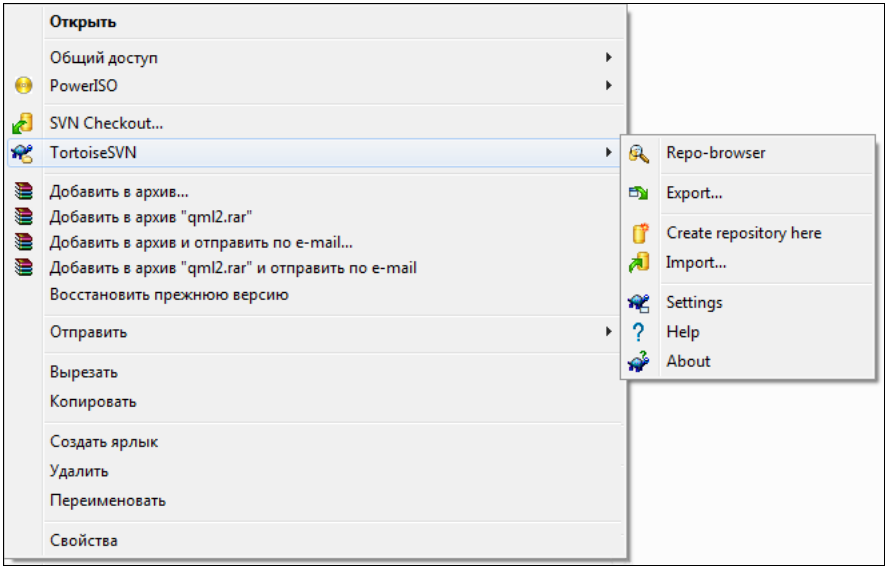


Рис. 4.24. Контекстное меню оболочки Windows и TortoiseSVN

Посмотреть структуру выбранного репозитория Subversion можно из окна **Repository Browser** (Обозреватель репозитория), которое вызывается из контекстного меню (рис. 4.25). Вы можете просматривать структуру репозитория на состояние в любой момент его существования, достаточно указать соответствующий номер ревизии.

Файлы, находящиеся под управлением Subversion, помечаются в папках оболочки специальными значками (рис. 4.26). Галочка в зеленом круге означает, что соответствующий файл не менялся со времени последней фиксации рабочей копии. Восклицательный знак в красном круге означает, что со времени последней фиксации рабочей копии данный файл был изменен. Вопросительный знак в синем круге означает, что данный файл не находится под контролем Subversion, но, возможно, его следует внести туда. Если вам нужно внести под контроль Subversion или зафиксировать в репозитории сразу несколько файлов или директорий, вы просто выделяете их мышью, а затем выбираете соответствующую команду из контекстного меню. Для удобства выполнения групповых операций с файлами и папками вы можете пометить некоторые файлы как игнорируемые. В этом случае TortoiseSVN не будет выполнять никаких операций с данными файлами, даже если они попали в выделение. Задавать шаблоны игнорируемых файлов можно как по расширению, так и по имени файла. Можно также добавлять в список игнорирования отдельные файлы. Вполне уместно пометить как игнорируемые различные временные файлы,

создаваемые в процессе сборки, такие как *.obj, *.lib и т. д. Если в репозитории хранятся только исходные тексты, имеет смысл пометить в качестве игнорируемых файлы с расширениями exe и dll.

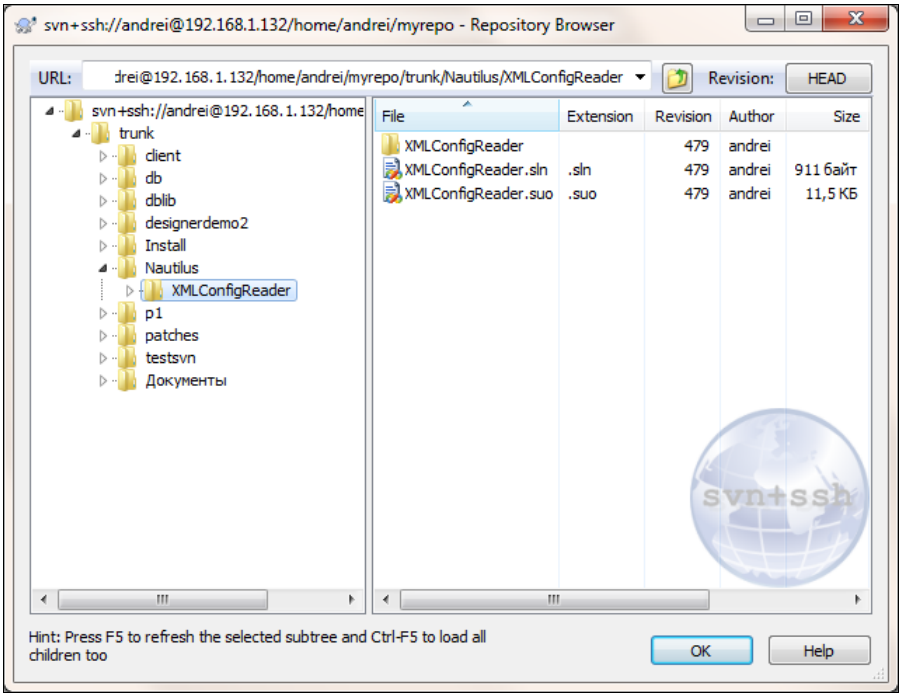


Рис. 4.25. Обзор репозиториев в TortoiseSVN

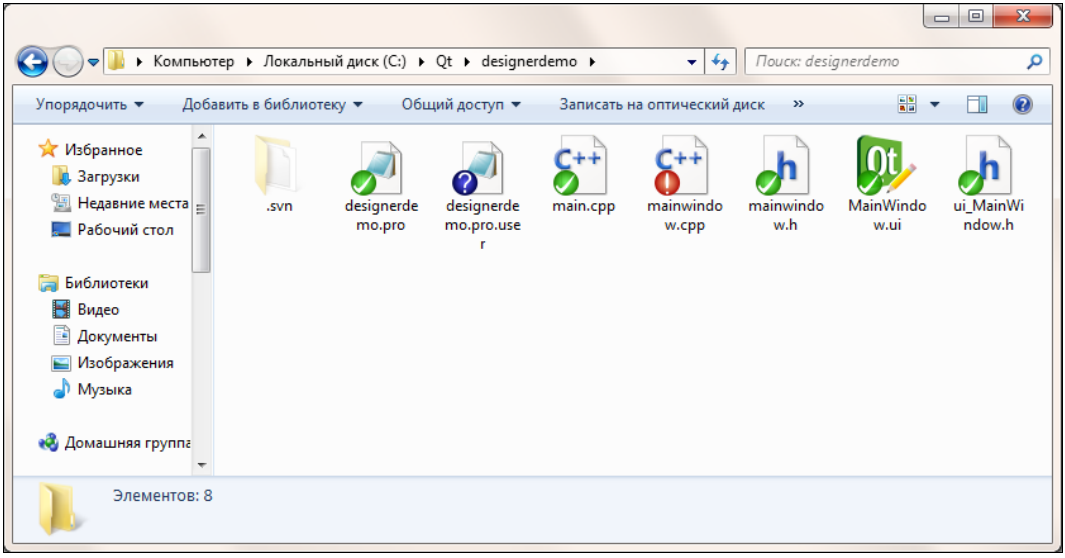


Рис. 4.26. Значки на пиктограммах файлов, находящихся под контролем Subversion

Примечание

Каким образом TortoiseSVN узнает, какие файлы находятся под контролем Subversion и в каком состоянии они находятся? Как и любой другой клиент Subversion, программа получает эту информацию из скрытой директории .svn, расположенной в той же директории, что и контролируемые файлы. Это означает, что TortoiseSVN пометит значками любые файлы, находящиеся под контролем Subversion, независимо от того, какой программой они были внесены под контроль. Если, помимо TortoiseSVN, вы используете и другие клиенты Subversion, то сами можете в этом убедиться. Сказанное означает также, что с помощью TortoiseSVN можно управлять рабочими копиями проектов Subversion, созданными с помощью других программ.

У TortoiseSVN также есть графический инструмент сравнения файлов (рис. 4.27), который позволяет сравнивать рабочую копию с различными ревизиями и различные ревизии между собой. С помощью этого инструмента можно также объединять содержимое разных ревизий одного файла и разрешать конфликты.

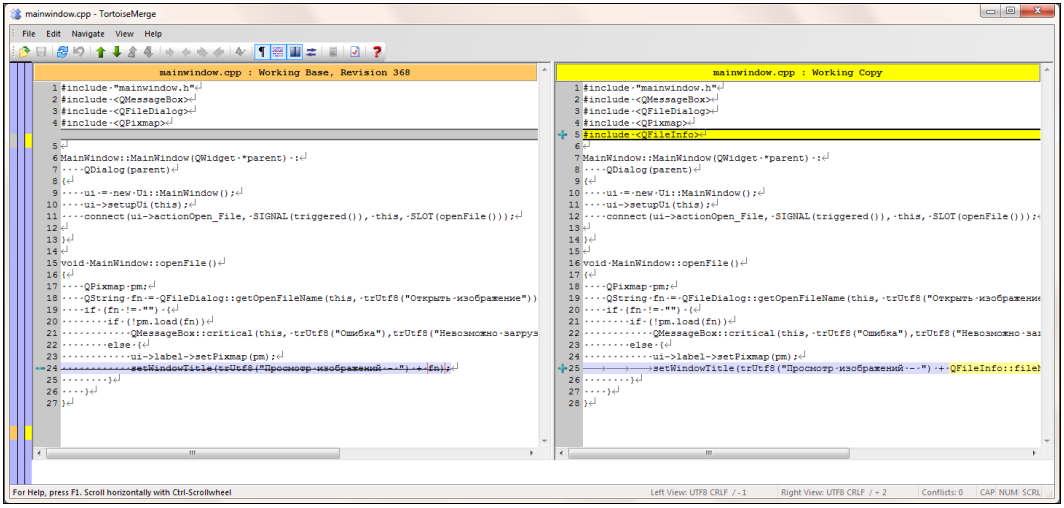


Рис. 4.27. Графический инструмент сравнения файлов TortoiseSVN

Примечание

Инструмент визуального сравнения TortoiseSVN умеет сравнивать не только простые текстовые файлы. С его помощью можно сравнивать, например, содержимое файлов Microsoft Word, причем в читабельном формате, а не в "сыром", как это сделает обычная утилита diff. Различающиеся фрагменты версий документа будут выделены цветом.

Даже когда вы уже приняли решение о фиксации определенных файлов в репозитории, TortoiseSVN еще раз дает вам возможность проверить и, при необходимости, изменить список файлов (рис. 4.28).

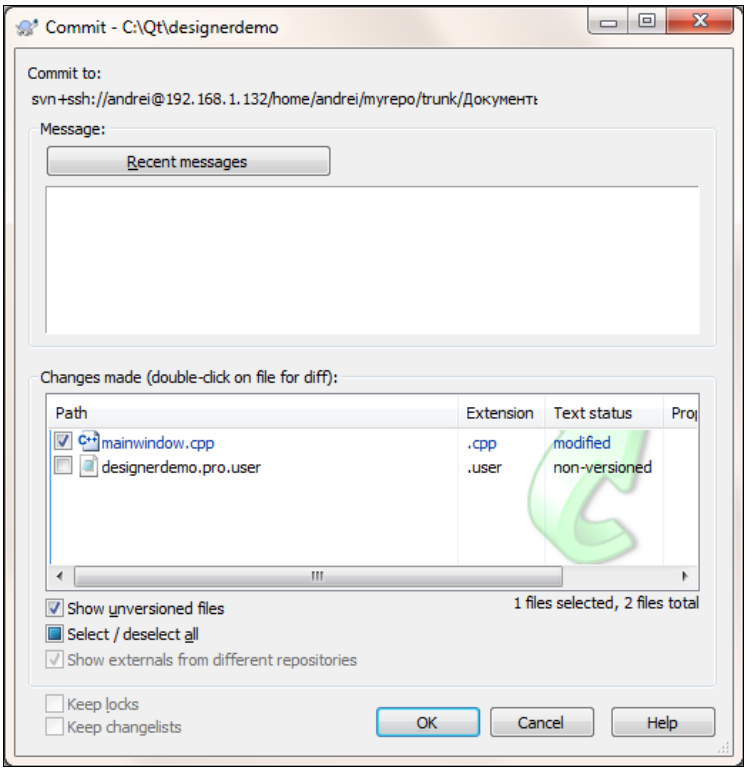


Рис. 4.28. Окно подтверждения операции TortoiseSVN

ЧАСТЬ II

Развиваем проекты

Глава 5. Qt и многопоточность

Глава 6. Высокоуровневый интерфейс потоков

Глава 7. Возвращаемся к Interview Framework

Глава 8. Библиотека Qt и ваша видеокарта

Глава 9. Растровая графика и текст

Глава 10. Система Graphics View Framework

ГЛАВА 5

Qt и многопоточность

Я пишу эту книгу на компьютере с четырехъядерным процессором. Рядом стоит компьютер, процессор которого обладает четырьмя ядрами, и каждое из них может выполнять четыре потока с помощью технологии HyperThreading. В последнее время производительность процессоров наращивается, в основном, за счет увеличения количества потоков, которые процессор может выполнять одновременно, и, следовательно, производительность приложений может повышаться благодаря параллельной обработке данных. Поскольку линейную последовательность операций человеку проще осмыслить, чем нелинейную структуру из ветвящихся цепочек операций, программирование с использованием потоков считается сложным делом, и многие программисты стараются этим не заниматься. Мы покажем, что все не так уж сложно, если помнить и не нарушать несколько формальных правил обращения с потоками.

Не следует, однако, думать, что с несколькими потоками любую задачу можно решить быстрее, чем с одним потоком. Иногда бывает наоборот. Главная проблема, которая возникает при разделении задачи на несколько потоков, — проблема синхронизации. Нередко бывает так, что один поток проводит столько времени в ожидании, когда другие потоки создадут необходимые условия для его работы, что весь выигрыш от использования потоков сходит на нет.

> ***Примечание***
>
> В каких случаях потоки могут быть полезны практически всегда? Лучше всего для оформления в виде отдельного потока подходят те задачи, которые выполняют много работы и не требуют при этом частого обновления графического интерфейса программы (например, шифрование данных файла, поиск информации в большом массиве и т. п.). Если решение такой задачи будет размещено в потоке графического интерфейса, интерфейс может просто "зависнуть" — виджеты будут реагировать на действия пользователя с очень большой задержкой (или не реагировать вовсе). Еще одна ситуация, когда потоки являются беспроигрышным решением, — та, в которой задачи, решаемые потоками, не связаны между собой и потокам практически не приходится обмениваться данными друг с другом. Примером такой ситуации является web-сервер, в котором для выполнения каждого запроса клиента используется отдельный поток.

Если поток посылает большое количество сигналов элементам графического интерфейса, имеет смысл перенести задачу в поток, которому принадлежат элементы

графического интерфейса. Сказанное относится, разумеется, только к Qt. При работе с игровым графическим движком, например, распределение задач может быть совсем другим.

Модель потоков графического приложения Qt предполагает, что существует один главный поток и несколько вспомогательных. Объекты графического интерфейса могут принадлежать только главному потоку. Но что делать, если у нас есть один или несколько потоков, которые должны выводить результаты своей работы, используя графический интерфейс пользователя? Сделать это совсем несложно, если вспомнить, что сигналы можно передавать между потоками.

Рассмотрим пример threadsdemo, в котором вспомогательный поток управляет состоянием строки индикатора в диалоговом окне. Если мы хотим, чтобы поток во всем вел себя как часть приложения Qt, мы должны создать класс-потомок класса `QThread` (листинг 5.1).

Листинг 5.1. Простейший класс потока

```
class Thread : public QThread
 {
    Q_OBJECT
 public:
     void run();
 signals:
     void progressed();
 };
```

Минимум, что нужно сделать в классе-потомке `QThread`, — это перекрыть метод `run()`. Именно этот метод является процедурой потока, которая будет выполняться одновременно с главной процедурой приложения. Метод `run()` нашего потока тоже достаточно прост (листинг 5.2).

Листинг 5.2. Простейший метод `run()`

```
void Thread::run()
{
   QTimer timer;
   timer.setInterval(100);
   timer.setSingleShot(false);
   connect(&timer, SIGNAL(timeout()), this, SIGNAL(progressed()));
   timer.start();
   exec();
}
```

В этом методе мы подготавливаем и запускаем таймер, который будет периодически эмитировать сигнал `progressed()` (обратите внимание на связь "сигнал–сигнал", которую мы устанавливаем с помощью метода `connect()`). Далее мы запускаем

таймер и вызываем метод `exec()` для запуска цикла обработки событий потока. Появление у класса `QThread` метода `exec()` (введенное в Qt 4) позволяет вспомогательному потоку обрабатывать сигналы и события в соответствии с моделью Qt, так же, как это делается в главном потоке приложения. Иногда такое поведение удобно, иногда — нежелательно. Далее мы рассмотрим пример потока, который обходится без цикла обработки событий. В этом же примере процедура потока будет выполняться до тех пор, пока объект потока метод `exec()` не получит приказ завершить цикл обработки.

Некоторые методы класса `QThread` (например, `start()`) предназначены для вызова из внешних потоков. Другие методы, такие как `exec()`, предназначены исключительно для вызова из самого потока. Как отличить одну группу методов от другой? Разработчики Qt нашли элегантное и очевидное решение: все методы, которые предназначены исключительно для потока, в котором выполняется метод `run()`, объявлены в разделе `protected` класса `QThread`. Соответственно, они видимы в методе, перекрывающем метод `run()` класса `QThread`, и невидимы за его пределами.

Рассмотрим определения двух методов из примера threadsdemo (листинг 5.3).

Листинг 5.3. Фрагмент примера threadsdemo

```
void Dialog::on_pushButton_clicked()
{
    if (!thread) {
        progressDialog = new ProgressDialog(this);
        progressDialog->show();
        thread = new Thread();
        connect(thread, SIGNAL(progressed()),
        progressDialog, SLOT(progressed()));
        thread->start();
    }
}
void Dialog::close_thread()
{
    if (thread) {
        thread->quit();
        thread->wait();
        delete thread;
        progressDialog->close();
        thread = 0;
    }
}
```

В методе `on_pushButton_clicked()` мы создаем новый объект класса `ProgressDialog`, (окно с индикатором прогресса), новый объект класса `Thread`, связываем сигнал `progressed()` объекта `thread` со слотом `progressed()` объекта `progressDialog` (этот слот изменяет значение индикатора прогресса). После этого мы запускаем новый

поток с помощью метода `start()`. Хотя поток `thread` и не владеет элементами графического интерфейса, он может управлять элементами, принадлежащими главному потоку (рис. 5.1).

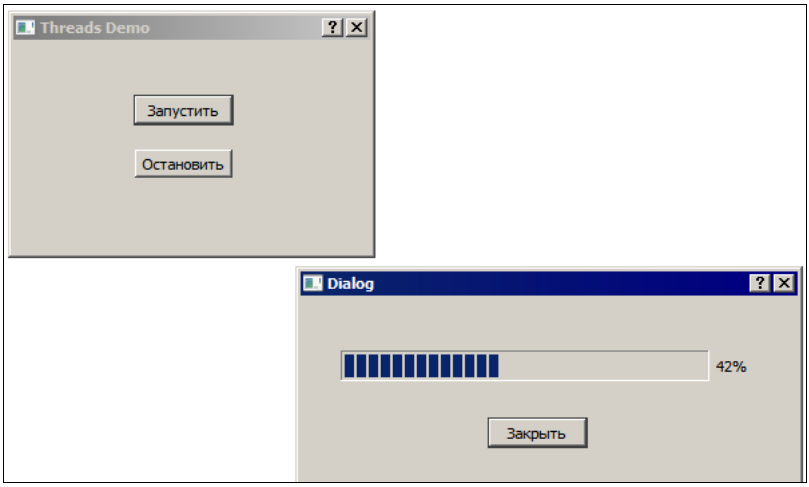


Рис. 5.1. Управление графическими элементами из вспомогательного потока

Существует распространенное заблуждение, что к тому моменту, когда метод `start()` возвращает управление, метод `run()` потока уже выполняется. Это предположение может быть верным на некоторых платформах и ошибочным на других. Если вам необходимо сделать что-то в основном потоке, точно зная, что вспомогательный поток уже выполняется, дождитесь сигнала `started()`.

Метод `close_thread()` проверяет, существует ли объект потока (переменная `thread` не равна 0) и если объект существует, вызывает его метод `quit()`. Это корректный способ завершения потока, который, в отличие от метода `terminate()`, не прерывает поток немедленно, а посылает в цикл обработки событий потока событие, приказывающее потоку завершиться. Если у потока нет собственного цикла обработки событий (мы не вызвали метод `exec()` в процедуре потока), стандартный метод `quit()` не сможет его завершить.

Поскольку метод `quit()` не может завершить поток немедленно, мы ожидаем завершения потока с помощью метода `wait()`. Этот метод блокирует выполнение вызвавшей его функции до тех пор, пока поток действительно не завершится. В качестве аргумента методу `wait()` можно передать максимальное время ожидания в миллисекундах. По истечении этого времени `wait()` вернет управление в любом случае. Поскольку мы уверены, что наш поток не зависнет навсегда, мы вызываем метод `wait()` без параметров, что означает неограниченное ожидание. После этого мы удаляем объект потока.

Примечание

Если вы попытаетесь удалить объект потока до того, как его функция `run()` завершила все важные действия (в том числе удаление объектов, созданных в стеке функции), это, скорее всего, приведет к ошибкам, самой частой из которых является ошибка доступа к памяти.

Когда метод `exec()` главного потока приложения получает команду завершиться (что обычно означает завершение приложения), дополнительные потоки продолжают выполняться, как ни в чем не бывало. Если вы хотите, чтобы ваша программа завершалась корректно, а не продолжала работать как зомби, без интерфейса, удостоверьтесь в деструкторе класса главного окна (или в другом методе, который гарантированно вызывается в процессе завершения приложения), что все дополнительные потоки закончили свою работу. Возможно, их придется завершить прямо в деструкторе главного окна, и это лучше делать с помощью комбинации `quit()`/`wait()`, а не метода `terminate()`.

Для объекта `progressDialog` мы вызываем метод `close()`, что приводит к закрытию окна и удалению объекта, поскольку в конструкторе окна мы присвоили ему атрибут `WA_DeleteOnClose`. По умолчанию метод `close()` не приводит к уничтожению объекта окна, что позволяет нам, например, прочитать данные, введенные пользователем в диалоговом окне. В нашем случае пользователь в окне ничего не вводит, так что объект можно удалять непосредственно в процессе закрытия.

Весьма распространенная ошибка связана с непониманием, в контексте какого потока существует объект `QThread`. Если в методе `run()` мы попытаемся создать объект `QTimer` динамически, указав в конструкторе в качестве родителя `this` (листинг 5.4), то получим сообщение об ошибке, примерно такое: "QObject: Cannot create children for a parent that is in a different thread". Этим сообщением система предупреждает нас о том, что нельзя создать объект, у которого родитель принадлежит другому потоку.

Листинг 5.4. Создание динамического объекта в процедуре потока (неправильное)

```
void Thread::run()
{
   QTimer * timer = new QTimer(this);
   ...
}
```

Все дело в том, что переменная `this` содержит указатель на объект `Thread`, а этот объект создается в главном потоке приложения и принадлежит ему. Переменная `this` не принадлежит тому потоку, в контексте которого вызывается метод `run()`. У объектов Qt есть метод `moveToThread()`, который позволяет передать объект из одного потока в другой. Но передавать объект потока в сам поток не рекомендуется, т. к. это может привести к разным странным ошибкам. Между прочим, метод `moveToThread()` не сработает, если у передаваемого объекта есть родитель, который, естественно, принадлежит исходному потоку. Прежде чем передать объект другому потоку, его надо "отцепить" от родителя, например, с помощью метода `setParent()`. Если же вам совершенно необходимо создать в потоке динамические структуры (как правило, хватает структур, объявленных в стеке), то можно вообще не указывать родителя (листинг 5.5).

Листинг 5.5. Создание динамического объекта в процедуре потока (правильное)

```
void Thread::run()
{
   QTimer * timer = new QTimer(0);
   ...
}
```

Основы теории многопоточности

Взаимной блокировкой называют такое состояние, когда несколько процессов или потоков заблокированы и выход каждого процесса (потока) из блокировки зависит от выхода из блокировки всех остальных.

Эдвард Коффман (Edward G. Coffman, Jr) сформулировал в 1971 году четыре условия, выполнение которых может привести к взаимоблокировкам в системе.

1. Наличие ресурсов, которые могут использоваться только одним процессом (потоком) в каждый момент времени. Такие ресурсы называются критическими.
2. Процесс (поток), имеющий контроль над критическим ресурсом, может попытаться захватить контроль над другими критическими ресурсами.
3. Пока процесс (поток) владеет критическим ресурсом, никакой другой процесс (поток) не может "отобрать" этот ресурс силой.
4. Циклическое ожидание: процессы захватывают ресурсы в таком порядке, что для продолжения работы процесса А необходимо, чтобы процесс В освободил критический ресурс, но процесс В может продолжить работу только тогда, когда процесс А освободит свой критический ресурс.

Последний пункт самый важный и нуждается в более подробном разъяснении. Допустим, один процесс захватил ресурс А и пытается захватить ресурс В. Если ресурс В захвачен другим процессом, выполнение первого процесса приостановится до тех пор, пока второй процесс не освободит В. В это время второй процесс пытается захватить А, не освобождая перед этим В. В результате его выполнение тоже приостанавливается. Подобный тупик может возникнуть только в том случае, если процессы (или потоки) пытаются получить ресурсы в разном порядке (один использует сначала А, потом В; второй — сначала В, потом А). Два простых правила, которые помогают избежать взаимоблокировок: во-первых, не следует захватывать два критических ресурса одновременно, а во-вторых, нужно стараться освободить критический ресурс как можно скорее (ведь его может ожидать другой процесс или поток). Если же процессам (потокам) требуется одновременно держать под контролем более одного критического ресурса, то нужно удостовериться в том, что все процессы (потоки) пытаются получить этот контроль в одном и том же порядке (рис. 5.2).

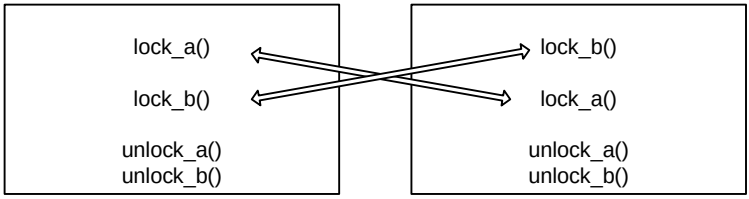


Рис. 5.2. Возникновение взаимоблокировки при разном порядке обращения к критическим ресурсам

Критические области

Критической областью называется часть программы, в которой она получает доступ к критическому ресурсу. Для того чтобы разные потоки программы не конфликтовали из-за критических ресурсов, следует соблюдать четыре правила:

1. Два потока не могут одновременно находиться в критической области.
2. Какие-либо предположения относительно быстродействия процессоров и их количества недопустимы.
3. Поток, находящийся вне критической области, не должен блокировать выполнение других потоков.
4. Ни один поток не должен ожидать бесконечно своей очереди для входа в критическую область.

Я хотел бы обратить особое внимание на правило 2. Это правило относится к работе с потоками вообще. Нельзя программировать многопоточные приложения, основываясь на предположении, что какой-то фрагмент кода выполняется быстрее, чем другой, даже если среднестатистически это так и есть. В многозадачной операционной системе вы не можете знать, сколько времени фактически займет выполнение определенного участка кода (если только вы не контролируете процессор специальными средствами). Четырем вышеперечисленным условиям должны отвечать все примитивы синхронизации, управляющие доступом в критические области.

Реентерабельность

Мы говорим, что функция реентерабельна, если она может быть вызвана повторно из того же потока в то время, когда она уже вызвана в каком-то другом участке программы (и это не приведет к ошибкам). Реентерабельность имеет значение при обработке сигналов и исключений, а также для функций, вызываемых рекурсивно, и функций обратного вызова. Тем, кто никогда не задумывался о свойстве реентерабельности функций, может показаться странным, что большинство функций, которые мы пишем, обладают этим свойством. Рассмотрим, например, рекурсивную функцию вычисления факториала (листинг 5.6).

Листинг 5.6. Типичная реентерабельная функция

```
unsigned long factorial(unsigned long x)
{
    if (x < 2) return 1;
    return x*factorial(x-1);
}
```

При каждом вызове функции `factorial()` в стеке создается область для хранения данных, связанных именно с этим вызовом. И до тех пор, пока стек не переполнится, один вызов `factorial()` не будет мешать другим вызовам. Иначе говоря, функция `factorial()` реентерабельна. Из этого примера можно вывести правило: все функции, которые работают только с локальными данными, расположенными в стеке, являются реентерабельными. Нереентерабельными могут быть (а могут и не быть) функции, которые обращаются к глобальным данным, расположенным за пределами стека. Большинство интересных и важных функций системы вынуждено обращаться к данным за пределами стека, но это не означает, что такие функции не могут быть реентерабельными.

Потоковая безопасность

Помимо реентерабельности важной характеристикой функции является потоковая безопасность (thread safety). Мы говорим, что функция потокобезопасна, если она может вызываться одновременно из разных потоков приложения и это не приведет к ошибкам. Может показаться, что любая реентерабельная функция потокобезопасна, но это не так. Рассмотрим пример (листинг 5.7).

Листинг 5.7. Реентерабельная, но не потокобезопасная функция

```
volatile bool lock = false;
bool foo()
{
1:      if (lock) return false;
2:      lock = true;
        ... // критический код
n:      lock = false;
        return true;
}
```

Вызов функции `foo()`, прерывающий другой вызов `foo()`, безопасен, если оба вызова выполняются из одного потока. Действительно: допустим, что значение переменной `lock` равно `false` на момент выполнения строки 1, и что вызов `foo()` был прерван другим вызовом `foo()` (например, в результате поступления сигнала), между строками 1 и 2, причем второй вызов выполнен в том же потоке, что и первый. Управление к первому вызову не может вернуться до тех пор, пока второй вызов `foo()` не вернет управление, а в этот момент переменная `lock` снова будет равна

`false`. Сама строка 2 не может быть прервана вызовом, сделанным из того же потока, т. к. она выполняется одной инструкцией микропроцессора (сказанное верно для процессоров семейства Intel, но если для какой-то другой модели процессора это не так, то функция `foo()` потеряет реентерабельность). То же относится и к строке n. Если второй вызов `foo()` произойдет между строками 2 и n, это приведет к тому, что второй вызов функции вернет управление в строке 1, т. е. функция сообщит нам, что доступ к критическому коду ей запрещен. Однако если функция `foo()` будет вызываться из нескольких потоков, то два вызова вполне могут попасть в область между строками 2 и n одновременно.

Точно так же, не всякая потоко-безопасная функция является реентерабельной. Например, если в начале функции вызывается мьютекс (подробнее о них будет рассказано далее), повторный вызов функции из того же потока может привести к взаимоблокировке, когда одна функция ждет завершения другой, но не может вернуть ей управление. Для того чтобы избежать блокировок этого типа, мьютексы часто проектируются так, чтобы вызовы мьютекса из одного потока не препятствовали друг другу.

Примечание

Если функция работает только с локальными данными (объявленными в стеке), то она является реентерабельной и потокобезопасной одновременно.

Потоки без цикла обработки событий

Возможность запуска собственного цикла обработки событий в потоках Qt, появившаяся в версии Qt 4, очень удобна, если использовать обычную для Qt модель приложения, которое управляется событиями и сигналами. Но иногда мы создаем потоки именно для того, чтобы избежать модели "сигнал–слот". Иногда нам просто нужно выполнить какую-то длительную операцию с возможностью ее досрочного завершения или временной приостановки. Использовать слоты для этой цели неудобно. Допустим, мы поместим нашу длительную операцию в один слот. Во время работы этого слота цикл обработки сообщений просто ничего не может сделать. Можно, конечно, периодически активировать цикл из нашего слота с помощью метода `processMessages()` или чего-то подобного, но если в процессе работы нашего "длительного" слота будет получен сигнал, требующий корректного завершения потока, этот сигнал не сможет сделать ничего, пока слот не завершит работу. Иначе говоря, в такой ситуации стандартный сигнал корректного завершения будет просто бесполезен. В этом случае гораздо эффективнее будет другая модель потока, которую мы сейчас и реализуем. Класс `ExtThread` (листинг 5.8) является частью программы exthreads, полный текст которой вы найдете в папке Ch5/extthreads.

Листинг 5.8. Объявление класса потока `ExtThread`

```
#ifndef EXTTHREAD_H
#define EXTTHREAD_H
#include <QThread>
```

```
#include <QAtomicInt>
enum ExtThreadStates {
    Ready,
    Working,
    Sleeping,
    Finishing,
    Finished
};
class ExtThread : public QThread
{
    Q_OBJECT
public:
    explicit ExtThread(QObject *parent = 0);
    bool pause();
    bool resume();
    bool wait(unsigned long time = ULONG_MAX);
    ExtThreadStates getCurrentState();
    void start();
signals:
    void paused();
    void resumed();
public slots:
    void quit();
protected:
    bool CancellationPoint();
    bool pauseFor(unsigned long milliseconds = ULONG_MAX);
    virtual void beforeQuit();
    void done();
private:
    QAtomicInt currentState;
};
#endif // EXTTHREAD_H
```

Класс `ExtThread` перекрывает некоторые методы класса `QThread` и вводит некоторые новые методы. Мы разделим эти методы на две группы: методы, предназначенные для управления потоком извне (обычно их вызывает тот поток, который создает экземпляр `ExtThread`), и методы, которые используются внутри метода `run()`, т. е. в контексте самого потока. К первой группе относятся конструктор `ExtThread()`, методы `start()`, `pause()`, `resume()`, `wait()`, `getCurrentState()` и слот `quit()`. Ко второй группе относятся методы `CancellationPoint()`, `pauseFor()`, `beforeQuit()` и `done()`. Вы также можете пользоваться методами, унаследованными от `QThread`, за исключением тех методов, которые работают с очередью обработки событий.

Примечание

Как уже отмечалось, слоты не являются виртуальными методами. Это значит, что хотя указатели на переменные класса `ExtThread` совместимы по присваиванию с указате-

лями на QThread, попытка вызвать метод quit() после такого присваивания приведет к тому, что будет вызван метод quit() класса QThread, а он в нашем потоке не сработает.

Методы start(), quit() и wait() выполняют в нашем классе те же действия, что и в классе QThread, хотя внутренне делают это немного иначе. Метод getCurrentState() возвращает текущее состояние потока:

- Ready — поток готов к работе, но еще не запущен;
- Working — поток выполняется;
- Sleeping — поток приостановлен;
- Finishing — поток получил команду завершиться, но еще не завершился;
- Finished — поток завершился.

Последний пункт списка требует некоторых пояснений. Состояние Finished устанавливается тогда, когда поток уже не выполняет никаких явных действий, но до того, как произойдет выход из процедуры потока. Это значит, что в момент установки состояния Finished некоторые объекты, автоматически созданные процедурой потока, могут все еще существовать.

Методы pause() и resume() соответственно приостанавливают и возобновляют выполнение потока. Выполнение потока приостанавливается не сразу, а только тогда, когда процедура потока достигнет вызова CancellationPoint(). В момент фактического приостановления поток эмитирует сигнал paused(). Для возобновления выполнения потока необходимо вызвать метод resume(). Если на момент вызова resume() поток был приостановлен, он эмитирует сигнал resumed(). Вызов pause() не оказывает никакого воздействия на приостановленный поток, так же как и вызов resume() — на работающий.

Среди методов, предназначенных для использования внутри потока, самый интересный — CancellationPoint(). Этот метод обрабатывает внешние команды, передаваемые потоку, такие как команда на завершение, приостановку или возобновление работы процедуры потока. Если на момент вызова CancellationPoint() поток не получал никаких команд, выполнение процедуры потока продолжается в обычном порядке. Сила метода CancellationPoint() заключается в том, что вы сами решаете, в каких участках процедуры потока вызывать этот метод. Вы можете разместить метод CancellationPoint() в тех точках, где поток может быть безопасно приостановлен или завершен. Если внешний поток использует корректный способ завершения потока ExtThread с помощью метода quit(), фактическое завершение произойдет в одной из точек, где будет вызван метод CancellationPoint(). Естественно, если вы забудете добавить вызовы CancellationPoint() в свой метод потока, поток завершится только тогда, когда его процедура завершится сама собой или когда будет вызван метод terminate(), что является очень неудачным способом завершения функции потока, особенно при программировании на C++.

Выход из процедуры потока, написанной на C++, очень сильно отличается от выхода из процедуры потока, написанной на C. Язык C не создает автоматически никаких сложных структур данных, которые бы требовалось также автоматически

уничтожить при выходе соответствующих переменных из области видимости. Если же вы создали такие структуры данных явно, то вы сможете явно их уничтожить перед выходом из процедуры потока (помните принцип "кто создавал, тот и удаляет"). В отличие от C, язык C++ может создавать структуры данных автоматически и уничтожать их автоматически при выходе из процедуры (например, при вызове оператора `return`). Ключевые компоненты операционных систем пишутся на языке C, и практически во всех операционных системах есть функция досрочного выхода из потока. Но эта функция, как правило, не учитывает специфику C++ и по этой причине нам не подходит. Корректный выход из функции потока, написанной на C++, должен завершаться вызовом оператора `return` (явно или не явно, т. е. вместе со скобкой, обозначающей завершение функции).

Этот факт нашел свое отражение в приемах работы с `CancellationPoint()`. Функция `CancellationPoint()` возвращает значение типа `bool`. Если функция обнаружила, что поток необходимо завершить, она возвращает `true`, в противном случае — `false`. Так что корректный вызов `CancellationPoint()` должен выглядеть так:

```
if(CancellationPoint()) return;
```

Метод `done()` выполняет действия, связанные с завершением работы потока. Этот метод нужно вызывать тогда, когда вы завершаете работу потока без участия `CancellationPoint()` (например, в конце процедуры потока). Функция `CancellationPoint()`, при необходимости, вызовет этот метод сама. Помимо прочего, метод `done()` вызывает виртуальный метод `beforeQuit()`, который вы можете переопределить в своем классе потока, если вам требуется выполнять какие-то особые действия перед завершением процедуры потока. Необходимость в таком методе возникает потому, что у процедуры потока есть только одна точка входа, но может быть много точек выхода. Метод `beforeQuit()` избавляет вас от необходимости дублировать код.

Метод `pauseFor()` приостанавливает выполнение потока `ExtThread` на заданное количество миллисекунд. Перейдем теперь к реализации класса `ExtThread` (листинг 5.9).

Листинг 5.9. Реализация класса `ExtThread`

```
#include "extthread.h"
#include <QAtomicInt>
ExtThread::ExtThread(QObject *parent) :
    QThread(parent)
{
    currentState = Ready;
}
bool ExtThread::CancellationPoint()
{
    if (currentState == Finishing) {
        done();
        return true;
    }
```

```
    bool toSleep = false;
    while (currentState == Sleeping) {
        if (!toSleep) {
            emit paused();
            toSleep = true;
        }
        yieldCurrentThread();
    }
    if (toSleep)
        emit resumed();
    if (currentState == Finishing) {
        done();
        return true;
    }
    return false;
}
bool ExtThread::pause()
{
    bool res = currentState.testAndSetOrdered(Working, Sleeping);
    if (res) emit paused();
    return res;
}
bool ExtThread::resume()
{
    bool res = currentState.testAndSetOrdered(Sleeping, Working);
    if (res) emit resumed();
    return res;
}
 ExtThreadStates ExtThread::getCurrentState()
{
    return (ExtThreadStates) (int) currentState;
}
 bool ExtThread::wait(unsigned long time)
 {
     while (time > 0) {
         if (currentState == Finished){
             QThread::wait(time);
             return true;
         }
         QThread::msleep(1);
         time--;
         if (currentState == Finished) {
             QThread::wait(time);
             return  true;
         }
     }
     return false;
 }
```

```
bool ExtThread::pauseFor(unsigned long milliseconds)
{
    if (currentState.testAndSetOrdered(Working, Sleeping)) {
       while (milliseconds > 0) {
           QThread::sleep(1);
           milliseconds--;
       }
    }
    return currentState.testAndSetOrdered(Sleeping, Working);
}
void ExtThread::quit()
{
    currentState.testAndSetAcquire(Ready, Finished);
    while ((!currentState.testAndSetOrdered(Working, Finishing))&&
           (!currentState.testAndSetOrdered(Sleeping, Finishing))&&
           (currentState != Finishing)&& (currentState!= Finished));
}
void ExtThread::beforeQuit()
{
}
void ExtThread::done()
{
    beforeQuit();
    currentState = Finished;
    emit finished();
}
void ExtThread::start()
{
    if (currentState.testAndSetOrdered(Ready,
        Working)||currentState.testAndSetOrdered(Finished, Working))
       QThread::start();
}
```

Для того чтобы разобраться в работе основных методов этого класса, необходимо понять один важный принцип, который имеет большое значение для многопоточного программирования вообще. Речь идет об атомарности операции "проверить и установить значение". Рассмотрим это на практическом примере. В методе `resume()` происходит примерно следующее:

```
if (currentState == Sleeping) currentState = Working
```

Эта операция состоит из двух элементарных операций: проверка текущего значения переменной `currentState` и (в зависимости от результатов проверки) установка нового значения. Даже если элементарные операции являются атомарными (а это не обязательно так), то вся строка, написанная выше, не является атомарной. То есть в интервале между `if (currentState == Sleeping)` и `currentState = Working` другой поток мог бы "вклиниться", например, с такой операцией:

```
if (currentState == Sleeping) currentState = Finishing;
```

В результате этой операции переменная `currentState` должна была бы получить значение `Finishing`, но по факту она все равно получила бы значение `Working`, чего другой поток мог бы и не знать. Этой неприятной ситуации можно избежать, если операции "проверить и установить значение" с переменной `currentState` сделать атомарными, т. е. такими, что когда один поток выполняет эту операцию, ни один другой поток не может выполнять ту же (или другую) операцию с теми же операндами до тех пор, пока данная операция не завершится. Современные микропроцессоры обладают специальными командами, позволяющими выполнять атомарные операции с простыми типами данных (например, с целыми числами или указателями). Кроме того существуют алгоритмы, с помощью которых можно реализовать такие операции в неатомарных системах.

Для атомарных операций с целыми числами библиотека Qt library предоставляет нам класс `QAtomicInt`. В некоторых аспектах переменные типа `QAtomicInt` ведут себя как обычные целые числа. Но у класса `QAtomicInt` есть ряд методов, реализующих атомарные операции с этими числами. Метод `testAndSetOrdered()` представляет собой один из вариантов такой операции. Например, строка

```
currentState.testAndSetOrdered(Sleeping, Working);
```

эквивалентна строке

```
if (currentState == Sleeping) currentState = Working;
```

за исключением того, что другой поток не может "вклиниться" в выполнение этой операции. Метод `testAndSetOrdered()` возвращает значение типа `bool`, которое соответствует результату проверки. То есть, если метод возвращает `true`, присваивание состоялось, если `false`, — значит, нет.

Насколько важен тип `QAtomicInt` и подобные ему типы в многопоточном программировании, можно продемонстрировать на следующем примере. Имея в своем распоряжении `QAtomicInt`, мы можем без труда реализовать мьютекс — один из важнейших примитивов синхронизации потоков (листинг 5.10).

Листинг 5.10. Простой мьютекс на основе `QAtomicInt`

```
class Mutex : public QObject
{
public:
    explicit Mutex(QObject *parent = 0) : QObject(parent)
{
    m = 0;
}
void lock()
{
    while (!m.testAndSetOrdered(0, 1))
        QThread::yieldCurrentThread();

}
```

```
bool tryLock()
{
    return m.testAndSetOrdered(0, 1);
}
void unlock()
{
    m.testAndSetOrdered(1, 0);
}
private:
    QAtomicInt m;
};
```

Вызов метода `lock()` класса `Mutex` приводит к закрытию доступа в критическую область. После этого любой другой поток, который попытается вызвать `lock()`, будет заблокирован до тех пор, пока обладатель доступа в критическую область не вызовет метод `unlock()`. Метод `tryLock()` представляет собой попытку получить доступ в критическую область. Если какой-то другой поток получил этот доступ (т. е. вызвал `lock()` успешно), метод `tryLock()` вернет значение `false`, и это будет означать, что у потока, вызвавшего `tryLock()`, нет доступа в критическую область. Если же метод `tryLock()` вернет `true`, это значит, что данный поток получил доступ в критическую область и должен вызвать `unlock()`, когда закончит с ней работать.

> ***Примечание***
>
> Вы скажете, что другой поток может получить доступ в занятую критическую область, вызвав метод `unlock()` мьютекса. Конечно, это так. Впрочем, другой поток может войти в занятую критическую область, просто не вызвав метод `lock()`, поскольку такой вызов — единственное, что защищает критическую область от попадания в нее двух потоков. Вообще говоря, разделение доступа к критическим областям внутри одного приложения основано на добровольном согласии всех потоков играть по правилам.

Действительная слабость предложенного мьютекса (который является просто демонстрационным примером) заключается в другом. Когда метод `lock()` блокирует потоку доступ в критическую область, мы хотим, чтобы заблокированный поток требовал как можно меньше процессорного времени. В идеале, чтобы вообще не требовал. Наш мьютекс в этом случае входит в цикл, в котором вызывается метод

```
QThread::yieldCurrentThread();
```

который просто возвращает системе порцию машинного времени, изначально выделенную потоку. Это неплохой вариант холостого цикла, но не самый лучший. Самый лучший могут обеспечить низкоуровневые средства операционной системы. Другая проблема связана с исключительными ситуациями. В результате возникновения исключительной ситуации поток может выйти из критической области, не вызвав метод `unlock()`. Из-за этого другие потоки никогда не смогут войти в критическую область. Вот почему все же лучше пользоваться стандартными примитивами синхронизации, которые предоставляет нам Qt library.

Кроме того, описанный выше вариант мьютекса делает функцию, использующую его, нереентерабельной.

Атомарные операции и порядок доступа к памяти

Современные процессоры могут, в целях оптимизации, изменить порядок операций доступа к памяти по сравнению с тем, как он представлен в программе. Очень часто нам требуется, чтобы определенная операция доступа к памяти была выполнена до или после атомарной операции, иначе сама атомарная операция потеряет смысл. Иными словами, при выполнении атомарных операций желательно, чтобы изменение порядка доступа к памяти не затрагивало саму операцию, т. е. все, что должно выполняться до атомарной операции, выполнялось бы до нее, независимо от взглядов компилятора и процессора на оптимизацию, а все операции доступа к памяти, которые в программе следуют за атомарной операцией, ни в коем случае не выполнялись до выполнения этой операции. У классов `QAtomicInt` и `QAtomicPointer` есть несколько вариантов метода `TestAndSet*` и других атомарных методов, которые поразному управляют изменением порядка доступа к памяти. Более подробные описания каждого метода вы найдете в документации.

Пример с использованием класса *ExtThread*

Мы используем класс `ExtThread`, чтобы создать в демонстрационной программе поток, сканирующий директории, начиная с заданной, в поисках файла с именем, соответствующим маске. Наша процедура сканирует все директории и выдает имена всех подходящих файлов. Для того чтобы написать собственную процедуру потока, как и в случае с `QThread`, мы создаем класс-потомок `ExtThread`, в котором перекрываем метод `run()`. Этот класс называется `FileFinder` (листинг 5.11).

Листинг 5.11. Класс `FileFinder`

```
class FileFinder : public ExtThread
{
    Q_OBJECT
public:
    explicit FileFinder(QObject *parent = 0);
    ~FileFinder();
    void findFiles(const QString &startFrom, const QString &filter);
signals:
    void updateList(QString str);
public slots:
protected:
    void run();
private:
    QString startFrom;
    QString filter;
};
```

Помимо перекрытого метода `run()` класс `FileFinder` имеет метод `findFiles()`, который позволяет задать начальную директорию для поиска и маску имени файла.

После вызова метода `findFiles()` объект класса `FileFinder` начинает посылать сигналы `updateList()`, которые передают параметр типа `QString`. В параметре содержится список полных имен файлов, найденных с момента отправки предыдущего сигнала `updateList()`. Имена файлов разделены символами `\n`. Думаю, внешний интерфейс класса `FileFinder` не вызывает у вас вопросов. Перейдем к реализации (листинг 5.12).

Листинг 5.12. Реализация класса `FileFinder`

```
FileFinder::FileFinder(QObject *parent) :
   ExtThread(parent)
{
}

FileFinder::~FileFinder()
{
}

void FileFinder::run()
{
   QStringList results;
   QDir dir(startFrom, filter, QDir::SortFlags( QDir::Name |
      QDir::IgnoreCase), QDir::AllDirs | QDir::Files);
   QFileInfoList fil;
   for (int i = 0; i < dir.entryInfoList().count(); i++) {
      if (dir.entryInfoList().at(i).isFile())
         results.append(dir.entryInfoList().at(i).absoluteFilePath());
      if (dir.entryInfoList().at(i).isDir())
         if ((dir.entryInfoList().at(i).fileName() != "." )
             && (dir.entryInfoList().at(i).fileName() != ".."))
         fil.append(dir.entryInfoList().at(i));
   }
   QString str;
   for (int k = 0; k < results.count(); k++)
      str = str + results.at(k) + "\n";
   emit updateList(str);
   results.clear();
   if (CancellationPoint()) return;
   while (fil.count() != 0) {
      for (int i = fil.count() — 1; i >= 0 ; i--) {
         QDir dir(fil.at(i).absoluteFilePath(), filter,
            QDir::SortFlags( QDir::Name | QDir::IgnoreCase),
            QDir::AllDirs|QDir::Files);
         fil.removeAt(i);
         for (int j = 0; j < dir.entryInfoList().count(); j++) {
            if (dir.entryInfoList().at(j).isDir()) {
```

```
                if ((dir.entryInfoList().at(j).fileName() != ".")
                && (dir.entryInfoList().at(j).fileName() != ..."))
                   fil.append(dir.entryInfoList().at(j));
             }
             else
             results.append(dir.entryInfoList().at(j).absoluteFilePath());
          }
          if (results.count() > 0) {
             QString str;
             for (int k = 0; k < results.count(); k++)
                str = str + results.at(k) + "\n";
             emit updateList(str);
             results.clear();
          }
          if (CancellationPoint()) return;
       }
       if (CancellationPoint()) return;
    }
    done();
}

void FileFinder::findFiles(const QString &startFrom,
const QString &filter)
{
    this->startFrom = startFrom;
    this->filter = filter;
    start();
}
```

Метод `findFiles()` заполняет поля класса `FileFinder` и вызывает метод `start()` своего базового класса, т. е. потока `ExtThread`. В методе `run()` реализована нерекурсивная процедура обхода дерева директорий, начиная с заданной. Вдаваться в детали мы не будем. Нетрудно заметить, что в теле процедуры мы периодически эмитируем сигнал `updateList()` и вызываем метод `CancellationPoint()`.

> ***Примечание***
>
> Одна из причин, по которой я отказался от рекурсивной функции обхода дерева директорий, заключается в необходимости корректного выхода из потока при вызове функции `CancellationPoint()`. Во всех операционных системах, поддерживаемых Qt, есть функция `ExitThread()` (или ей подобная), которая может завершить данный поток, будучи вызвана из функции любой глубины вложенности. Но при программировании в C++ эта функция неприменима. Вот почему на всем протяжении метода `run()` мы должны иметь возможность вызвать `return` непосредственно из метода `run()`. Впрочем, это ограничение не является существенным, ведь теория учит нас, что любая рекурсивная функция может быть преобразована в нерекурсивную.

Подключить наш класс `FileFinder` к графическому интерфейсу приложения очень просто (листинг 5.13).

Листинг 5.13. Создание графического интерфейса для `FileFinder`

```
Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
    ui->setupUi(this);
    fileFinder = new FileFinder();
    paused = false;
    searching = false;
    connect(fileFinder, SIGNAL(finished()), this,
        SLOT(serachFinished()));
    connect(fileFinder, SIGNAL(updateList(QString)), this,
        SLOT(updateList(QString)));
    counter = 0;
}
```

Здесь мы связываем два сигнала объекта `fileFinder` (унаследованный `finished()` и собственный `updateList()`) со слотами класса `Dialog`. Теперь у нас есть собственный класс для поиска файлов (рис. 5.3).

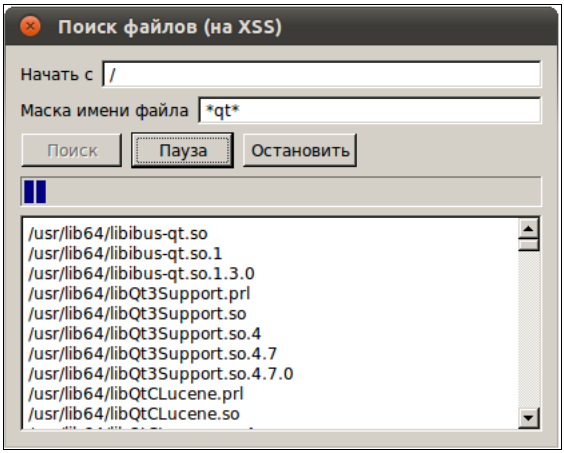


Рис. 5.3. Поиск файлов с помощью многопоточного приложения

Примечание

При тестировании программы под ОС Linux помните, что благодаря ссылкам в файловых системах Linux могут появиться циклы. То есть структура файловой системы Linux представляет собой не дерево, а граф в более общем смысле. Поиск циклов на графе — задача увлекательная, но мы ею не занимаемся. Бесконечного зацикливания нашей функции на циклах графа все равно не произойдет, т. к. поиск файла по конкретному пути завершится, когда суммарная длина пути превысит максимальную длину имени файла.

В связи с рассмотренным примером возникает один весьма важный вопрос: что делать, если поток A посылает сигналы потоку B быстрее, чем поток B успевает их

обрабатывать? Мы знаем, что это приведет к резкому замедлению или даже полному "зависанию" программы. В нашем примере поток `fileFinder` выполняет больше работы, чем основной поток и потому основной поток успевает обрабатывать сигналы, которые посылает `fileFinder`. Так, по крайней мере, обстоит дело в тех системах, в которых пример тестировался. Но мы помним, что, вообще говоря, мы не должны делать никаких предположений относительно сравнительной скорости выполнения потоков, и нам необходимы механизмы, которые бы гарантировали, что сигналы от потока-источника не будут поступать слишком часто. Один из таких механизмов — метод `blockSignals()`, определенный в классе `QObject`.

Если наш главный поток вызовет метод `fileFinder->blockSignals(true)`, сигналы от потока `fileFinder` перестанут поступать до тех пор, пока главный поток не выполнит вызов `fileFinder->blockSignals(false)`. При этом поток `fileFinder` ничего не заметит, в том смысле, что его операции не будут приостановлены. Глядя на исходный текст `fileFinder`, нетрудно убедиться, что даже при краткосрочном блокировании сигналов приложение может пропустить часть найденных файлов. Объект может проверить, заблокированы ли его сигналы, с помощью метода `signalsBlocked()`. Если этот механизм используется, мы должны строить работу объекта, исходя из предположения, что не все его сигналы достигнут адресата.

> ***Примечание***
>
> Другое распространенное применение метода `blockSignals()` заставить объект прекратить рассылку сигналов при завершении его жизненного цикла.

Еще один способ решения той же проблемы — передать константу `Qt::BlockingQueuedConnection` в последнем аргументе метода `connect()` при связывании сигнала одного потока со слотом другого. В результате поток, пославший сигнал, будет заблокирован до тех пор, пока слот потока-приемника не завершит обработку этого сигнала. Это довольно специфический метод связывания сигналов и слотов. Он подходит для тех случаев, когда один поток посылает сигналы другому потоку нечасто, и при этом поток-источник должен точно знать, когда обработка этих сигналов закончится (этот способ, между прочим, можно использовать для разделения доступа потоков к критическим ресурсам). Однако этот способ сильно тормозит работу потока-источника сигналов в том случае, если сигналы посылаются часто (в предельном случае два потока могут выполнять операцию медленнее, чем это сделал бы один поток!). Кроме того, такой подход чреват взаимоблокировками.

Локальная память потоков

Давайте вспомним, что кому принадлежит. Объект потока и все его данные принадлежат тому потоку, который создал этот объект. Все переменные, созданные в стеке метода `run()`, принадлежат потоку, который инкапсулирован в классе потока. Что касается данных этого класса, то они, в принципе, доступны всем потокам, т. к. память потоков — это общая память приложения. Поскольку другие классы не имеют доступа к структурам данных, объявленным в разделе `private:` класса пото-

ка, и эти структуры создаются в конструкторе класса для каждого объекта потока, фактически эти структуры данных являются собственностью конкретного потока.

Но иногда потоку требуется иметь собственные данные, отделенные на более глубоком уровне (на уровне компилятора или операционной системы, т. к. архитектуры современных процессоров не защищают области памяти потоков друг от друга). Qt позволяет вам использовать локальную память потока, правда, пользы от этого немного.

Класс `QThreadStorage`, который управляет локальной памятью потоков, позволяет вам хранить только один указатель. Причем содержимое этого указателя должно быть либо нулем, либо адресом памяти, выделенным с помощью `new`. Класс `QThreadStorage` берет на себя заботу о жизненном цикле этого указателя и уничтожает соответствующий объект оператором `delete` при завершении процедуры потока с помощью оператора `return`. Вообще говоря, локальная память потоков важна при программировании на языке C и его потомках. В объектно-ориентированном мире C++ возможность связывать данные с потоком именно таким способом не имеет особого смысла.

Передача данных межу потоками

До сих пор мы пользовались самым простым методом передачи данных между потоками — данными, передаваемыми через аргументы связи сигнал–слот. Поскольку потоки могут использовать для обмена данными любую глобально видимую область памяти (т. е. область памяти, выделенную в "куче", а не в стеке процедуры потока), основная проблема заключатся в том, как обеспечить разграничение доступа к памяти между потоками. Очень часто модель взаимодействия потоков можно описать в терминах "писатель" (поток, записывающий данные в общую область памяти) и "читатель" (поток, читающий данные из общей области). В этом случае задачу разграничения доступа к памяти можно свести к требованию, чтобы читатель и писатель не производили свои операции над одной областью памяти одновременно. Если писателей несколько, они также не должны работать с одной и той же областью памяти одновременно. Сделать это можно по-разному. Например, можно представить себе модель, в которой читатель не имеет доступа ко всей общей области памяти, пока к ней имеет доступ писатель. Затем, когда писатель освобождает память, исключительный доступ к ней получает доступ читатель, и писатель ждет, когда читатель освободит память. Другие модели основаны на том факте, что ни читатель, ни писатель не выполняют операций на всем пространстве разделяемой памяти одновременно. Это значит, что писатель может работать с одним фрагментом разделяемой памяти, тогда как читатель будет работать с другим фрагментом. Очевидно, что эти модели эффективнее.

Мы рассмотрим стандартные средства, предоставляемые Qt library, и два более эффективных варианта.

Класс *QSharedMemory*

Класс `QSharedMemory` полезен при создании области памяти, разделяемой между процессами (и мы рассмотрим его подробнее в этом контексте). Однако разделение доступа к памяти с помощью `QSharedMemory` возможно и в случае нескольких потоков. Для потоков класс `QSharedMemory` может предоставить методы `lock()` и `unlock()`, первый из которых предоставляет полный доступ ко всей области памяти вызвавшему потоку и перекрывает его для остальных. Таким образом, `QSharedMemory` работает по первой из перечисленных выше моделей. Обсуждать класс `QSharedMemory` более подробно мы сейчас не будем. Все, что нужно знать об этом классе, вы прочтете в документации.

Мьютексы и семафоры

С принципом работы мьютекса мы уже знакомы. Само слово мьютекс (mutex) происходит от английских слов mutual exclusion — взаимное исключение. Назначение мьютекса заключается в том, чтобы сигнализировать, какой из потоков обладает доступом к критическому ресурсу в данный момент. Я пишу "сигнализировать", потому что ограничения, наложенные мьютексами, не являются обязательными для исполнения. Все зависит от "доброй воли" потоков. Класс Qt `QMutex` обладает теми же методами, что и наш "самодельный" мьютекс, рассмотренный выше. Мы останавливаемся на классе `QMutex` подробнее, поскольку он и его "помощник" класс `QMutexLocker` упрощают решение нескольких деликатных проблем многопоточного программирования. Вспомним ситуацию, описанную выше: функция, вызвавшая метод `lock()` мьютекса, неожиданно завершается, например в результате возникшего исключения. Можно, конечно, "завернуть" всю критическую область в такую конструкцию:

```
try {
} catch(...) {
mutex.unlock()
}
```

Но тогда и все исключения придется обрабатывать в этой же конструкции, а это может быть нежелательно по многим причинам. Функции-обертки программируются иначе. Другая ситуация — функция со многими точками выхода. Как правило, мы хотим, чтобы перед выходом из функции был вызван метод `unlock()`. Если функция обладает многими точками выхода, мы можем просто забыть вызвать `unlock()` перед одной из них. В результате возникнет ошибка, которая будет проявлять себя лишь в некоторых, возможно, довольно редких случаях.

Разработчики Qt предлагают следующее решение этой проблемы: в функции в качестве локальной переменной создается объект класса `QMutexLocker`. Конструктору объекта передается указатель на объект класса `QMutex` (который может быть создан где угодно). Объект `QMutexLocker` пытается вызвать метод `lock()` переданного ему объекта `QMutex`, и как только эта операция завершается успехом, выполнение процедуры продолжается. Вы можете открыть мьютекс явным образом, вызвав метод

unlock() объекта QMutexLocker. Если же вы этого не сделаете, метод unlock() будет вызван в момент уничтожения переменной QMutexLocker (а поскольку переменная локальная, это уничтожение произойдет при любом выходе из функции, соответствующем правилам C++, в том числе при выходе в результате исключения). Вот так просто. Используя локальную переменную типа QMutexLocker, можно написать функцию, позволяющую нескольким потокам добавлять записи в список stringList (листинг 5.14).

Листинг 5.14. Использование класса QMutexLocker

```
void TextDevice::write(const QString &text)
{
  QMutexLocker locker(&mutex);
  stringList->append(text);
}
```

Обратите внимание, что объект QMutexLocker создается в стеке функции. Этот код эквивалентен представленному в листинге 5.15, с тем отличием, что в листинге 5.14 мьютекс будет открыт по выходе из функции write(), даже если этот выход произойдет в результате исключения.

Листинг 5.15. Аналог функции добавления записей без использования QMutexLocker

```
void TextDevice::write(const QString &text )
{
  mutex.lock();
  stringList->append(text);
  mutex.unlock();
}
```

Примечание

Обратите внимание на то, что в отличие от нашего демонстрационного мьютекса (листинг 5.10), закрытый мьютекс QMutex нельзя открыть из другого потока. Впрочем, это все равно не мешает при желании обойти ограничения, налагаемые мьютексами.

Мы можем использовать этот же трюк в нашем классе ExtThread. Нам, как вы помните, необходимо, чтобы метод done() вызывался перед завершением работы метода run(). Мы можем "автоматизировать" вызов done(), если внесем в класс ExtThread небольшие изменения (листинг 5.16).

Листинг 5.16. Новая версия класса ExtThread

```
class ExitHelper;

class ExtThread : public QThread
{
    Q_OBJECT
```

```
public:
    explicit ExtThread(QObject *parent = 0);
    bool pause();
    bool resume();
    bool wait(unsigned long time = ULONG_MAX);
    ExtThreadStates getCurrentState();
    void start();
signals:
    void paused();
    void resumed();
public slots:
    void quit();
protected:
    friend class ExitHelper;
    bool CancellationPoint();
    bool pauseFor(unsigned long milliseconds = ULONG_MAX);
    virtual void beforeQuit();
    void done();
private:
    QAtomicInt currentState;
};

class ExitHelper
{
public:
    explicit ExitHelper(ExtThread * thread)
    {
        this->thread = thread;
    }
    ~ExitHelper()
    {
        thread->done();
    }
private:
    ExtThread * thread;
};
```

В метод `CancellationPoint()` тоже придется внести изменения (листинг 5.17).

Листинг 5.17. Новая версия `CancellationPoint()`

```
bool ExtThread::CancellationPoint()
{
    if (currentState == Finishing)
        return true;
    bool toSleep = false;
```

```
    while (currentState == Sleeping) {
        if (!toSleep) {
            emit paused();
            toSleep = true;
        }
        yieldCurrentThread();
    }
    if (toSleep)
        emit resumed();
    if (currentState == Finishing)
        return true;
    return false;
}
```

Поскольку теперь метод `done()` будет вызываться неявно, нам не следует вызывать его явно, внутри `CancellationPoint()`, иначе метод будет вызван дважды. В этой версии и метод `run()`, перекрываемый в потомках `ExtThread`, будет выглядеть по-другому (листинг 5.18).

Листинг 5.18. Новые правила для метода `run()`

```
void FileFinder::run()
{
    ExitHelper onExit(this);
    ...
}
```

Теперь нам не нужно беспокоиться о вызове `done()` перед каждым оператором `return`. Подобные приемы гарантируют выполнение определенных действий, связанных с завершением потока, даже в случае возникновения исключения. Но само исключение должно быть где-то обработано, иначе программа, скорее всего, аварийно завершится, и наши действия окажутся бесполезными. Для предотвращения такой ситуации нужно весь код функции `run()` поместить в какую-нибудь дополнительную функцию, например `runInternal()`, и вызывать ее из функции `run()`, заключив в конструкцию `try - catch`.

Семафоры представляют собой более сложные структуры по сравнению с мьютексами. Если мьютекс используется для разграничения доступа к одному критическому ресурсу, семафоры могут защищать несколько критических ресурсов одновременно. С помощью метода `acquire()` класса `QSemaphore` мы можем указать, сколько именно разделяемых ресурсов (например, слотов разделяемой памяти) мы хотим получить. Количество ресурсов передается методу в качестве параметра. Если требуемого количества свободных ресурсов нет, поток будет заблокирован до тех пор, пока они не появятся. Метод `release()` освобождает ресурсы, захваченные методом `acquire()`. Как и в случае `acquire()`, вы можете указать количество освобождаемых ресурсов, которое может не совпадать с количеством захваченных. Се-

мафоры полезны в ситуации, когда n потоков пытаются получить доступ к m ресурсам, причем каждому потоку нужно r ресурсов и r * n > m.

В документации Qt сказано, что лучшее применение семафоров — разграничение доступа потоков к кольцевому буферу. В числе примеров, которыми сопровождается Qt, есть и пример использования семафоров вместе с кольцевым буфером. К этому нечего добавить, за исключением того, что далее будет рассмотрен кольцевой буфер, который вообще не нуждается в примитивах синхронизации, таких как семафоры и мьютексы. Правда, этот тип буфера можно использовать только в одной, зато весьма распространенной, ситуации.

Еще один интересный примитив синхронизации — класс `QWaitCondition`. С помощью объектов этого класса поток может заставить один или несколько других потоков приостановить свою работу до наступления некоторого события. Ожидающие потоки приостанавливают свое выполнение, вызывая метод `wait()` объекта `QWaitCondition`. Когда управляющий поток должен разрешить возобновление выполнения ожидающих потоков, он вызывает метод `wakeOne()`, если необходимо возобновить работу одного, случайно выбранного потока, или `WakeAll()`, если требуется возобновить выполнение всех ожидающих потоков.

Методу `wait()` в качестве параметра можно передать либо указатель на объект `QMutex`, либо указатель на объект `QReadWriteLock`. В момент вызова `wait()` `QMutex` и `QReadWriteLock` должны быть открыты. В момент выхода из метода `wait()` соответствующий объект (`QMutex` или `QReadWriteLock`) закрывается и гарантирует потоку исключительный доступ к тому, к чему поток обращается после выхода из `wait()`. Это сделано для того, чтобы не возник конфликт доступа между "разбуженными потоками". Фактически мы имеем атомарную операцию "ждать наступления произвольного события и закрыть доступ к критическому ресурсу, когда оно наступит".

Примечание

А если ожидающим потокам нечего делить между собой? Тогда каждый из них может использовать свой собственный мьютекс.

Кольцевой буфер без блокировок

В последнее время широкую популярность приобретают безблокировочные (lockless) алгоритмы доступа к разделяемой памяти. Безблокировочные алгоритмы отличаются от традиционных алгоритмов следующим:

- при выполнении такого алгоритма ни один поток не может приостановить выполнение других потоков на сколько-нибудь значительное время;
- при выполнении безблокировочного алгоритма, как правило, сразу несколько потоков могут получить доступ к (разным частям) разделяемой памяти;
- при выполнении безблокировочного алгоритма невозможна взаимная блокировка потоков;
- безблокировочные алгоритмы не используют традиционные примитивы синхронизации (такие как мьютексы и семафоры).

Все вышеперечисленное делает безблокировочные алгоритмы более эффективными по сравнению с традиционными. Вызов того же мьютекса или семафора очень часто бывает весьма дорогостоящей операцией (в смысле машинного времени). Алгоритм, который обходится без этой операции, может существенно сэкономить время процессора. Отсутствие необходимости ждать во время блокировки — весьма заметный фактор ускорения. По безблокировочным алгоритмам существует довольно много публикаций (в основном, в научных журналах, т. к. применение этих алгоритмов еще не стало преобладающей тенденцией программирования). Мы рассмотрим два таких алгоритма, которые уже хорошо зарекомендовали себя в деле.

Рассмотрим ситуацию "один писатель — один читатель" (я надеюсь, что эта книга не попадет в подобную ситуацию!). Отличным средством обмена данными между одним писателем и одним читателем является кольцевой буфер (рис. 5.4) Действительно, в кольцевом буфере запись и чтение всегда будут происходить в разных областях разделяемой памяти. При этом в описываемой ситуации кольцевому буферу не требуются примитивы синхронизации. Мы реализуем кольцевой буфер в классе `CircularBuffer` (листинг 5.19).

Рис. 5.4. Кольцевой буфер

Листинг 5.19. Класс `CircularBuffer`

```
class CircularBuffer : public QObject
{
    Q_OBJECT
public:
    explicit CircularBuffer(quint32 bufSize, QObject *parent = 0);
    ~CircularBuffer();
    quint32 read(void * buf, quint32 bufSize, quint32 blockSize = 1);
    quint32 write(void * buf, quint32 bufSize, quint32 blockSize = 1);
    bool wouldWriteBlock(quint32 blockSize);
    bool wouldReadBlock(quint32 blockSize);
```

```
    quint32 freeSpace();
    quint32 bytesAvailable();
    void reset();
    void stop();
    bool isEOF();
signals:
    void bufferFull();
    void bufferEmpty();
    void dataAdded(quint32 size);
    void hasFreeSpace(quint32 size);
    void reachedEOF();
private:
    void exposeSingleBufferWrite(void ** buffer, quint32 *  bytes);
    void exposeSingleBufferRead(void ** buffer, quint32 *  bytes);
    void addBytesWritten(quint32 bytes);
    void addBytesRead(quint32 bytes);
    quint32 bufSize;
    char * buffer;
    quint64 writePos;
    quint64 readPos;
    bool eof;
    bool stopped;
protected:
    void connectNotify(const char * signal);
};
```

Рассмотрим сначала публичный интерфейс класса. В конструкторе буфера можно указать его размер в байтах. Что делают методы `read()` и `write()`, в общем, должно быть понятно. Необходимо, однако, пояснить некоторые нюансы. Ни один из методов класса `CircularBuffer` не является блокирующим. Методы `read()` и `write()` возвращают количество байтов (соответственно, прочитанных или записанных). Если метод `read()` не смог прочитать данные по какой-либо причине (обычно потому, что в буфере нет данных), он возвращает значение 0. Точно так же ведет себя метод `write()`, если он не смог записать данные в буфер, например, потому что буфер заполнен. Функция `freeSpace()` возвращает объем свободного места (в байтах), если вы хотите знать, будет ли вызов `write()` успешен, вызовите сначала эту функцию.

> ***Примечание***
>
> Поскольку буфер предназначен для модели с одним писателем, фактический объем свободного места в буфере не может уменьшиться в промежуток между вызовами `freeSpace()` и `write()`. Он может только увеличиться.

Аналогичным образом функция `bytesAvailable()` возвращает количество байтов, доступных для чтения. И точно так же, как в случае с `freeSpace()`, это значение не может уменьшиться до очередного вызова `read()`. Второй параметр методов `read()` и `write()` позволяет задать размер минимального блока записи или чтения. По

умолчанию этот параметр имеет значение 1 (1 байт), но очень часто мы передаем данные блоками фиксированного размера (например, экземплярами структур, или даже простыми типами, размер которых превышает 1 байт). В этом случае полезно указывать размер блока. Если количество байтов, доступных для чтения или записи, меньше размера блока, то соответствующий метод (`read()` или `write()`) ничего не сделает и вернет значение 0. При использовании блоков одинакового размера при чтении и записи такого, конечно, не должно происходить. Если размер используемых блоков превышает 1 байт, то при создании буфера крайне рекомендуется выбирать размер буфера кратным размеру блока. Метод `write()` может записать меньше данных, чем вы потребовали (если количество данных не превышает значение, возвращенное методом `freeSpace()`, то этого не случится). Рекомендуется, однако, проверять значение, которое возвращает метод. То же относится и к методу `read()`.

Методы `wouldWriteBlock()` и `wouldWriteBlock()` позволяют определить, можно ли записать в буфер или прочитать из буфера определенное количество байтов. Фактически это другой вариант методов `freeSpace()` и `bytesAvailable()`.

Метод `isEOF()` очень важен. Допустим, что метод `read()` вернул значение 0. Это означает, что в буфере нет данных. Но почему? Возможно, писатель просто не успел записать данные в буфер и надо немного подождать. Но может быть и так, что писатель закончил записывать данные в буфер и ждать больше нечего. Когда писатель заканчивает запись данных в буфер, он вызывает метод `stop()`, которым объявляет, что больше в буфер ничего писать не будет. Если после вызова `stop()` в буфере все еще имеются какие-то данные, читатель будет читать их как обычно, но когда данные закончатся, будет эмитирован сигнал `reachedEOF()`, а метод `isEOF()` вернет значение `true`. Таким образом, читатель может узнать, что не следует ждать появления новых данных в буфере. Метод `reset()` возвращает буфер в то состояние, в котором он находился сразу после создания. Во избежание конфликтов этот метод следует вызывать тогда, когда ни писатель, ни читатель работать с буфером не будут (например, при получении сигнала `reachedEOF()`). Метод `reset()` сбрасывает состояние, установленное методом `stop()`, сбрасывает флаг `eof` и очищает буфер. В результате буфер снова готов к передаче данных. Сигналы `dataAdded()` и `hasFreeSpace()` эмитируются после успешной операции записи или чтения соответственно. В параметрах сигналов передается не общий объем свободного места и не общее количество байт в буфере, а то количество байт, на которые изменились эти значения. Перейдем теперь к реализации класса (листинг 5.20).

Листинг 5.20. Реализация класса `CircularBuffer`

```
CircularBuffer::CircularBuffer(quint32 bufSize, QObject *parent) :
    QObject(parent)
{
    this->bufSize = bufSize;
    buffer =  new char[bufSize];
    reset();
}
```

```
CircularBuffer::~CircularBuffer()
{
    delete[] buffer;
}

void CircularBuffer::exposeSingleBufferWrite(void **buffer,
    quint32 *bytes)
{
    quint64 writePtrPos = writePos % bufSize;
    quint64 readPtrPos = readPos % bufSize;
    *buffer = &(this->buffer[writePtrPos]);
    if (writePtrPos > readPtrPos) {
        *bytes = bufSize - writePtrPos;
    } else {
        if (writePtrPos == readPtrPos) {
            *bytes = bufSize - writePos + readPos;
            if (*bytes > bufSize - writePtrPos)
                *bytes = bufSize - writePtrPos;
        } else {
            *bytes = readPtrPos - writePtrPos;
        }
    }
}

void CircularBuffer::exposeSingleBufferRead(void **buffer,
    quint32 *bytes)
{
    quint64 writePtrPos = writePos % bufSize;
    quint64 readPtrPos = readPos % bufSize;
    *buffer = &(this->buffer[readPtrPos]);
    if (writePtrPos > readPtrPos) {
      (*bytes) = writePtrPos - readPtrPos;
    } else {
        if (writePtrPos == readPtrPos) {
            *bytes = writePos - readPos;
            if (*bytes > bufSize - readPtrPos)
                *bytes = bufSize - readPtrPos;
        } else {
            *bytes = bufSize - readPtrPos;
        }
    }
}

void CircularBuffer::addBytesWritten(quint32 bytes)
{
  writePos += bytes;
  emit dataAdded(bytes);
```

```
  if (writePos == readPos + bufSize)
      emit bufferFull();
}

void CircularBuffer::addBytesRead(quint32 bytes)
{
  readPos += bytes;
  emit hasFreeSpace(bytes);
  if (writePos == readPos) {
      emit bufferEmpty();
  }
}

quint32 CircularBuffer::read(void *buf,
    quint32 bufSize, quint32 blockSize)
{
    if (eof) return 0;
    void * b;
    quint32 bytes;
    exposeSingleBufferRead(&b, &bytes);
    if ((bytes == 0) && stopped) {
        eof = true;
        emit reachedEOF();
    }
    if (bytes < blockSize) return 0;
    bytes = bytes < bufSize ? bytes : bufSize;
    bytes = bytes — (bytes % blockSize);
    memcpy(buf, b, bytes);
    addBytesRead(bytes);
    return bytes;
}

quint32 CircularBuffer::write(void *buf, quint32 bufSize,
    quint32 blockSize)
{
    if (stopped) return 0;
    void * b;
    quint32 bytes;
    exposeSingleBufferWrite(&b, &bytes);
    if (bytes < blockSize) return 0;
    bytes = bytes < bufSize ? bytes : bufSize;
    bytes = bytes — (bytes % blockSize);
    memcpy(b, buf, bytes);
    addBytesWritten(bytes);
    return bytes;
}
```

```
void CircularBuffer::reset()
{
    writePos = 0;
    readPos = 0;
    eof = false;
    stopped = false;
    emit bufferEmpty();
}
void CircularBuffer::stop()
{
    stopped = true;
}

bool CircularBuffer::isEOF()
{
    if (!stopped) return false;
    if (!eof) {
        void * b;
        quint32 bytes;
        exposeSingleBufferRead(&b, &bytes);
        if (bytes == 0)
            eof = true;
    }
    return eof == true;
}

bool CircularBuffer::wouldReadBlock(quint32 blockSize)
{
    if (eof) return true;
    void * b;
    quint32 bytes;
    exposeSingleBufferRead(&b, &bytes);
    return bytes < blockSize;
}

bool CircularBuffer::wouldWriteBlock(quint32 blockSize)
{
    if (stopped) return true;
    void * b;
    unsigned int bytes;
    exposeSingleBufferWrite(&b, &bytes);
    return bytes < blockSize;
}

quint32 CircularBuffer::freeSpace()
{
    void * b;
    quint32 bytes;
```

```
    exposeSingleBufferWrite(&b, &bytes);
    return bytes; }

quint32 CircularBuffer::bytesAvailable()
{
    void * b;
    quint32 bytes;
    exposeSingleBufferRead(&b, &bytes);
    return bytes; }

void CircularBuffer::connectNotify(const char *signal)
{
    void * b;
    unsigned int bytes;
    exposeSingleBufferWrite(&b, &bytes);
    if (bytes == bufSize) emit bufferEmpty();
    else
    if (bytes == 0) emit bufferFull();
    if (bytes != 0) emit dataAdded(bytes);
    if (bytes < bufSize) emit hasFreeSpace(bufSize — bytes);
}
```

Хотя наш буфер работает как кольцевой, в его основе, естественно, лежит линейная область памяти. Как вы, конечно, знаете, такой буфер содержит два указателя: один для записи, другой для чтения. Эти указатели движутся друг за другом как стрелки часов, но указатель для чтения никогда не должен обгонять указатель для записи. Проблемы возникают, когда один из указателей переходит через границу области (из конца в начало). При этом относительные положения указателей могут измениться. Например, указатель для чтения может иметь большее значение, чем указатель для записи, хотя, на самом деле, он по-прежнему отстает. Просто указатель для записи достиг верхней границы буфера и вернулся в начало. Приватные переменные `writePos` и `readPos` содержат абсолютные положения указателей, которые определяются общим количеством прочитанных и записанных байтов (если вам требуется эта информация, вы можете добавить в класс методы, возвращающие значения указанных переменных). Положения указателей относительно области памяти буфера соответствуют остаткам от целочисленного деления значения указателя на размер буфера. Сама память буфера представляется массивом переменных типа `char`. Даже на 64-битных машинах тип `char` по-прежнему имеет размер 1 байт.

При работе с нашим буфером нередко возникает ситуация, когда, например, свободная область разбивается на две части (от указателя для записи и до конца буфера и от начала буфера до указателя для чтения). Более сложная (и более эффективная) реализация буфера должна работать с обеими частями, наша упрощенная реализация работает только с одной. Это не очень страшно (снижается только быстродействие — вместо одного вызова `read()` или `write()` иногда приходится делать два), но это очень важно в другом аспекте. Если вы записываете и читаете

данные фиксированными блоками, то размер буфера обязательно должен быть кратным размеру блока, иначе буфер не будет работать правильно. Методы `freeSpace()` и `bytesAvailable()` возвращают количество байтов, которое может быть записано или прочитано за один вызов соответствующего метода, т. е., например, от указателя для записи и до границы буфера. Фактические значения свободного или занятого пространства могут быть больше, чем значения, возвращенные этими методами.

Если вы хотите создать улучшенный вариант буфера, то вот вам подсказка: замените методы

```
void exposeSingleBufferWrite(void ** buffer, quint32 *  bytes);
void exposeSingleBufferRead(void ** buffer, quint32 *  bytes);
```

на

```
void exposeDoubleBufferWrite(void ** buffer1, quint32 *  bytes1,
void ** buffer2, quint32 *  bytes2);
void exposeDoubleBufferRead(void ** buffer1, quint32 *  bytes1,
void ** buffer2, quint32 *  bytes2);
```

Они при необходимости смогут возвращать линейные фрагменты в начале и в конце буфера одновременно (если доступен только один линейный фрагмент, в переменную, на которую указывает `bytes2`, следует записать значение 0). Затем переработайте методы, которые вызывают `exposeSingleBufferWrite()` и `exposeSingleBufferRead()` с учетом сделанных изменений.

Кольцевой буфер хорош для передачи данных между потоками не только тем, что он не использует примитивы синхронизации, но и тем, что он позволяет регулировать интенсивность обмена динамически. Например, частое появление сигнала `bufferEmpty()` может подсказать потоку-читателю, что надо снизить интенсивность опроса буфера, а частое появление сигнала `bufferFull()` — что интенсивность опроса необходимо, наоборот, увеличить.

Обратите внимание на метод `connectNotify()`. Этот виртуальный метод, который мы перекрыли в своем классе, вызывается тогда, когда один из сигналов нашего класса связывается с другим сигналом или слотом. Имя сигнала передается в параметре метода в кодировке Latin-1. На мой взгляд, метод `connectNotify()` полезен, в основном, тем, что с его помощью можно оповестить объекты, только что связанные с нашим объектом, о каких-то важных событиях или параметрах нашего объекта. Например, если к моменту подключения слотов объекта-читателя в буфере уже есть данные, имеет смысл послать сигнал, оповещающий читателя об этом. В нашем примере мы не проверяем, какой именно сигнал был связан, и просто эмитируем все сигналы, отражающие состояние буфера.

Для демонстрации работы кольцевого буфера мы напишем программу мониторинга псевдосигналов. Источником сигналов будет поток на основе `ExtThread` (листинг 5.21), который будет записывать случайные числа в четыре кольцевых буфера. Класс главного окна нашей программы будет считывать эти значения из буферов и отображать их с помощью компонентов `QLCDNumber`.

Листинг 5.21. Поток-генератор случайных сигналов

```
#ifndef SIGNALGENERATOR_H
#define SIGNALGENERATOR_H
#include "extthread.h"
class CircularBuffer;
class SignalGenerator : public ExtThread
{
    Q_OBJECT
public:
    explicit SignalGenerator(QObject *parent = 0);
    void setBuffer(CircularBuffer * buffer, CircularBuffer * buffer2,
                   CircularBuffer * buffer3, CircularBuffer * buffer4);
signals:
public slots:
protected:
    void run();
    void beforeQuit();
private:
    CircularBuffer * cb;
    CircularBuffer * cb2;
    CircularBuffer * cb3;
    CircularBuffer * cb4;
};
#endif // SIGNALGENERATOR_H

#include "signalgenerator.h"
#include "circularbuffer.h"
#include <cstdlib>
using namespace std;
SignalGenerator::SignalGenerator(QObject *parent) :
    ExtThread(parent)
{
}

void SignalGenerator::setBuffer(CircularBuffer *buffer,
    CircularBuffer *buffer2, CircularBuffer *buffer3,
    CircularBuffer *buffer4)
{
    cb = buffer;
    cb2 = buffer2;
    cb3 = buffer3;
    cb4 = buffer4;
}

void SignalGenerator::run()
{
    for (int i = 0; i < 200; i++) {
        float f = (float)rand()/RAND_MAX;
```

```
        cb->write(&f, sizeof(float));
        QThread::msleep(200);
        f = (float)rand()/RAND_MAX;
        cb2->write(&f, sizeof(float));
        QThread::msleep(200);
        f = (float)rand()/RAND_MAX;
        cb2->write(&f, sizeof(float));
        QThread::msleep(200);
        f = (float)rand()/RAND_MAX;
        cb3->write(&f, sizeof(float));
        QThread::msleep(200);
        f = (float)rand()/RAND_MAX;
        cb4->write(&f, sizeof(float));
        if (CancellationPoint()) return;
    }
    done();
}

void SignalGenerator::beforeQuit()
{
    cb->stop();
    cb2->stop();
    cb3->stop();
    cb4->stop();
}
```

Панель отображения информации о сигналах (рис. 5.5) считывает данные из буферов периодически, используя таймеры.

Примечание

До сих пор, когда нам требовался периодически эмитируемый сигнал, мы использовали объект класса `QTimer`. Возможно, вы не знаете, что свой таймер есть у каждого объекта-потомка `QObject`. Для управления этим таймером служат методы `startTimer()` и `killTimer()`, а для того чтобы обрабатывать событие таймера, нужно перекрыть в своем классе метод `timerEvent()`.

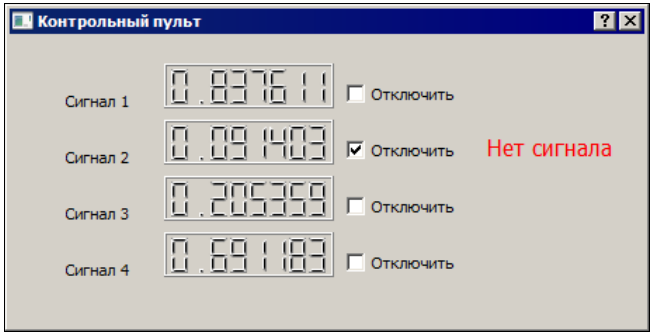


Рис. 5.5. Программа, имитирующая панель управления

В начале нам надо создать объекты буферов и поток-генератор сигналов, а также связать сигналы и слоты Qt между собой (листинг 5.22).

Листинг 5.22. Создание и настройка буферов

```
Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
    ui->setupUi(this);
    circularBuffer1 = new CircularBuffer(sizeof(float)*64);
    circularBuffer2 = new CircularBuffer(sizeof(float)*64);
    circularBuffer3 = new CircularBuffer(sizeof(float)*64);
    circularBuffer4 = new CircularBuffer(sizeof(float)*64);
    signalGenerator = new SignalGenerator();
    signalGenerator->setBuffer(circularBuffer1, circularBuffer2,
        circularBuffer3, circularBuffer4);
    signalGenerator->start();
    connect(circularBuffer1, SIGNAL(reachedEOF()), this,
        SLOT(no_signal1()));
    connect(circularBuffer2, SIGNAL(reachedEOF()), this,
        SLOT(no_signal2()));
    connect(circularBuffer3, SIGNAL(reachedEOF()), this,
        SLOT(no_signal3()));
    connect(circularBuffer4, SIGNAL(reachedEOF()), this,
        SLOT(no_signal4()));
    startTimer(100);
    startTimer(250);
}
```

Мы используем кольцевые буферы для передачи значений типа `float`. Обратите внимание, что мы устанавливаем размер кольцевого буфера кратным размеру типа `float`. Чтение значений сигналов из буферов позволяет нам рассмотреть еще один интересный аспект работы кольцевого буфера (листинг 5.23).

Листинг 5.23. Чтение данных из буферов

```
void Dialog::timerEvent(QTimerEvent *event)
{
    float f;
    if (circularBuffer1->bytesAvailable()) {
        circularBuffer1->read(&f, sizeof(float), sizeof(float));
        ui->lcdNumber->display(f);
    } else circularBuffer1->read(&f, 0);
    if (circularBuffer2->bytesAvailable()) {
        circularBuffer2->read(&f, sizeof(float), sizeof(float));
        ui->lcdNumber_2->display(f);
```

```
    }  else circularBuffer2->read(&f, 0);
    if (circularBuffer3->bytesAvailable()) {
        circularBuffer3->read(&f, sizeof(float), sizeof(float));
        ui->lcdNumber_3->display(f);
    } else circularBuffer3->read(&f, 0);
    if (circularBuffer4->bytesAvailable()) {
        circularBuffer4->read(&f, sizeof(float), sizeof(float));
        ui->lcdNumber_4->display(f);
    } else circularBuffer4->read(&f, 0);
}
```

Обратите внимание на попытку прочитать из буфера 0 байтов в том случае, если метод `bytesAvailable()` возвращает 0. Эта попытка в любом случае обречена на неудачу, но она приведет к генерации сигнала `reachedEOF()`, если писатель закончил передавать данные и буфер пуст. Такой экзотический метод генерации сигнала нужен нам потому, что перед каждым чтением из буфера мы проверяем, есть ли в нем данные. Если данные есть, значит, состояние `eof` еще не наступило, а если данных нет, мы не будем читать их из буфера и никогда не получим сигнал `reachedEOF()`. Сигнал `reachedEOF()` полезен в том случае, если мы читаем данные из буфера без предварительных проверок. В нашем же случае лучше подходит вызов метода `isEOF()`, а сигнал мы "провоцируем" просто в целях демонстрации.

Очередь без блокировок

Рассмотрим теперь другую модель, которая часто встречается на практике: "много писателей, один читатель". Примером реализации такой модели может служить многопоточное приложение, в котором главный поток (читатель) ведет журнал событий вспомогательных потоков (имеются в виду функциональные события: подключение клиента к сетевому серверу, ошибка скрипта и т. п.). Вспомогательные потоки в этой схеме выступают в роли писателей. Если события имеют одинаковый приоритет, то в роли структуры обмена можно использовать очередь FIFO ("первым вошел, первым вышел"). Эта структура называется буфером обратного магазинного типа. Соответственно, очередь LIFO ("последним вошел, первым вышел") именуется буфером магазинного типа. При этом имеется в виду аналогия с магазином огнестрельного оружия, а не с магазином, в который мы ходим за продуктами, и где все, обычно, происходит наоборот.

> ***Примечание***
>
> Рассмотренный ниже алгоритм нетрудно модифицировать для обработки событий с приоритетом (когда события с более высоким приоритетом читатель получает раньше, даже если они появились позже). Предлагаю вам самостоятельно реализовать такую очередь в качестве упражнения.

Итак, нам нужна очередь FIFO, в которую могли бы одновременно помещать данные несколько потоков и при этом один поток мог бы считывать эти данные в то же самое время. Мы реализуем такую очередь в два этапа. Первый этап — создание связного списка, в который несколько потоков могут добавлять данные, не блоки-

руя работу друг друга (и не используя примитивов синхронизации). Для реализации такого списка потребуется уже знакомая нам атомарная операция "проверить значение и присвоить новое". Только теперь эта операция будет выполняться не над целыми числами, а над указателями.

Такую возможность дает нам класс `QAtomicPointer`. Поскольку указатели являются целыми числами, мы могли бы воспользоваться и классом `QAtomicInt`, но `QAtomicPointer` позволяет нам сделать гораздо больше, чем `QAtomicInt`. Рассмотрим, например, как выглядит структура, соответствующая элементу связного списка, которая использует указатель `QAtomicPointer` (листинг 5.24).

Листинг 5.24. Узел очереди без блокировок с атомарным указателем

```
#include <QAtomicPointer>
struct LLRecord;

typedef QAtomicPointer<LLRecord> LLPtr;

struct LLRecord
{
    bool inUse;
    RecordType recordType;
    char message[256];
    LLPtr next;
    LLRecord * shadowPtr;
};
```

Секрет типа `QAtomicPointer` заключается в том, что это шаблон. Для получения из него класса необходимо сообщить ему тип данных, на который (простите за тавтологию) указывает указатель. В результате у нас появляется класс, который эмулирует поведение указателя на некоторый тип данных. Такие классы, — их обычно называют интеллектуальными указателями (smart pointers), — совместимы по присваиванию с обычными указателями и поддерживают ряд операций над указателями, например, операцию разыменования указателя (оператор "`*`"). Но кроме этого интеллектуальные указатели обладают множеством возможностей и функций, которых нет у обычных указателей: счетчиком ссылок, безопасным разыменовыванием нулевого указателя (когда обращение к указателю со значением `NULL` не приводит к ошибке защиты) и, например, возможностью выполнять атомарные операции. Библиотека Qt активно использует интеллектуальные указатели на внутреннем уровне, но на уровне внешних интерфейсов предпочитает обычные. Все-таки интеллектуальные указатели сложнее в обслуживании, нежели обычные, а некоторые аспекты их использования могут сбить неподготовленного программиста с толку.

Вернемся к нашей задаче. Допустим, у нас есть связный список с двумя служебными узлами: `Head` (голова) и `Tail` (хвост) и некоторым количеством информационных узлов между ними. В соответствии с принципом FIFO мы добавляем новые узлы в конец списка (перед узлом `Tail`). Допустим, два потока одновременно хотят доба-

вить узлы в наш список. Вот как это выглядит с применением атомарных операций (рис. 5.6). Сначала полям `next` обоих узлов присваивается (не атомарно) указатель на узел `Tail`. Затем выполняется атомарная операция, в процессе которой новые узлы пытаются присвоить полю `next` прежнего последнего узла в списке указатель

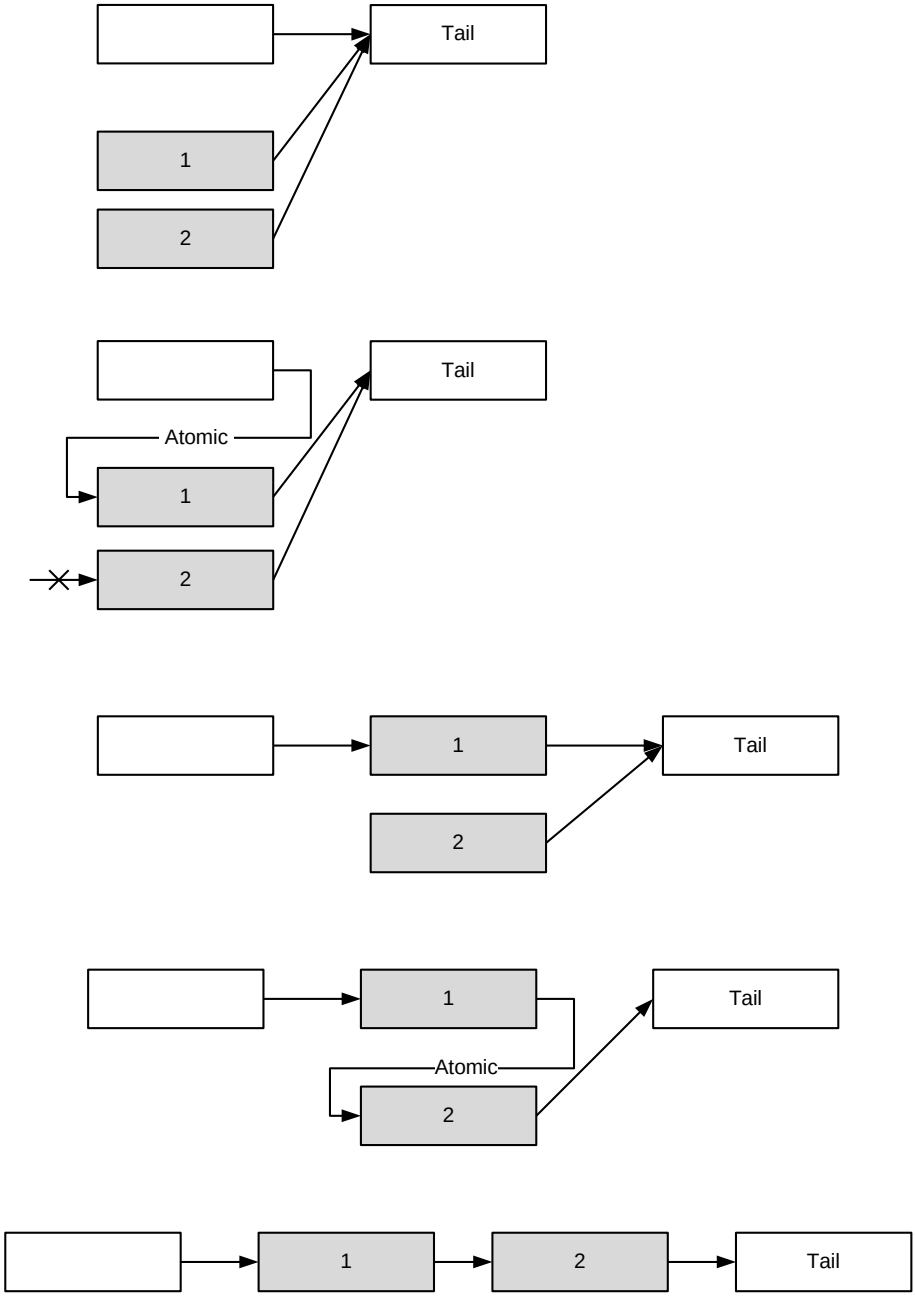


Рис. 5.6. Добавление нового узла в связный список

на себя. Поскольку операция атомарная, она закончится успешно только для одного узла, который и станет новым последним узлом списка (т. е. узлом, предшествующим узлу `Tail`). Для второго узла (и всех остальных претендентов, если они появятся) операция закончится неудачей, о чем ему сообщит соответствующий метод. Все, что ему (и другим узлам) останется сделать, — это повторить операции с самого начала (или только вторую, если первая уже выполнена). В результате каждой итерации список будет увеличиваться на один узел, при этом целостность списка будет сохраняться всегда.

На что указывает поле `next` типа `QAtmicPointer` бывшего последнего элемента списка? Поскольку указатель атомарный, он всегда указывает либо на узел `Tail`, либо на новый узел списка, чье поле `next`, в свою очередь, указывает либо на узел `Tail`, либо на новый узел списка. Это происходит потому, что мы сначала присваиваем полю `next` нового узла указатель на узел `Tail`, и только потом вставляем узел в сам список. Иначе говоря, список остается целостным даже непосредственно в момент добавления нового узла, и если в этот момент какая-то другая операция проходит список от головы к хвосту, используя поля `next`, она ничего не "заметит". Порядок выполнения обращений к памяти в этом случае очень важен. Подумайте, что может произойти, если узел сначала встроится в список и только потом получит указатель на `Tail`.

От пояснений и иллюстраций перейдем к коду. Класс `LLQueue` (листинг 5.25) реализует очередь FIFO с возможностью конкурентного добавления узлов без блокировок. Исходные тексты примеров к книге доступны на сайте **http://symmetrica.net/qt47book/**.

Листинг 5.25. Узел очереди без блокировок с атомарным указателем

```
class LLQueue : public QObject
{
    Q_OBJECT
public:
    explicit LLQueue(QObject *parent = 0);
    bool empty();
signals:
public slots:
protected:
  bool insertRecord(LLRecord *record, LLRecord * before);
  LLRecord * removeFirstRecord();
  LLRecord blank;
  LLRecord head;
  LLRecord tail;
};

LLQueue::LLQueue(QObject *parent) :
    QObject(parent)
{
    head.recordType = RecordTypeHead;
```

```
    tail.recordType = RecordTypeTail;
    blank.recordType = RecordTypeBlank;
    head.next = &tail;
    head.shadowPtr = &tail;
    tail.next = 0;
}

bool LLQueue::insertRecord(LLRecord *record, LLRecord * before)
{
    LLPtr current = &head;
    while (current->next != before) {
        LLRecord * tmpPtr = current->next;
        if (tmpPtr ==  current->shadowPtr)
            current = tmpPtr;
        else return false;
        if (current == NULL) return false;
    }
    record->shadowPtr = record->next = before;
    if (current->next.testAndSetOrdered(before, record)) {
        current->shadowPtr = record;
        return true;
    }
    return false;
}

LLRecord * LLQueue::removeFirstRecord()
{
    if (head.next == &tail) return 0;
    LLPtr current = head.next;
    while (current->recordType == RecordTypeBlank)
        current = current->next;
    if (current == &tail) return 0;
    while (!insertRecord(&blank, current->next));
    head.next.fetchAndStoreOrdered(blank.next);
    head.shadowPtr = blank.next;
    current->next.fetchAndStoreOrdered(0);
    return current;
}

bool LLQueue::empty()
{
    LLPtr current = head.next;
    while (current != &tail) {
        if (current == NULL) current = head.next;
        if (current->recordType == 0)
            return false;
        if (current->recordType == RecordTypeTail)
            return true;
```

```
        current = current->next;
    }
    return true;
}
```

Обратите внимание на семантику интеллектуальных указателей. Например, объявление

```
LLPtr current;
```

практически во всем эквивалентно объявлению

```
LLRecord * current;
```

за исключением того, что указатель `LLPtr` гораздо мощнее.

Когда мы обращаемся к переменной типа `LLPtr`, используя оператор разыменовывания (явно или неявно, в виде оператора "`->`"), мы получаем поля структуры `LLRecord`, как и должно быть с указателем. Но если мы обращаемся к переменной типа `LLPtr`, используя точку, то получаем доступ к методам объекта класса `QAtomicPointer`, например к методу `testAndSetOrdered()`. Когда мы обращаемся к значению самой переменной, например в операциях сравнения или присваивания, мы опять-таки имеем дело с указателем на `LLRecord`.

Нас сейчас интересует функция `insertRecord()`, которая добавляет в список новые узлы. Хотя мы договорились добавлять новые узлы только в конец списка, по техническим соображениям, которые станут понятны далее, нам нужен метод, способный добавлять новый узел перед любым узлом, следующим за узлом `Head`, и вы, конечно, понимаете, что рассмотренный выше алгоритм вполне подходит для этой цели.

Удаление первого узла после узла `Head`, которое выполняет метод `removeFirstRecord()`, выглядит несколько сложнее (рис. 5.7). Обратите внимание на то, что поскольку мы добавляем новые узлы в конец списка, узел, предшествующий предпоследнему узлу списка (если считать последним узел `Tail`), и все предшествующие ему узлы, могут быть удалены без конфликта с потоками, добавляющими узлы. Если список содержит два или больше узлов между узлами `Head` и `Tail`, мы можем просто удалить узел, следующий за `Head` (на самом деле, это "просто" потребует атомарной операции, как мы увидим далее). Но, допустим, между `Head` и `Tail` содержится только один узел. Тогда мы добавляем специальный пустой узел `Blanc` после этого узла:

```
while (!insertRecord(&blank, current->next));
```

Здесь метод `insertRecord()` вызывается до тех пор, пока не вернет значение `true`, т. е. пока узел `Blanc` не будет добавлен. Узел `current` — это узел, на который указывает `head.next`, а узел, на который указывает `current->next`, — это узел перед которым мы хотим добавить узел `Blanc`. Обратите внимание, что значение `current->next` может меняться (если в этот момент в список добавляются узлы), но когда узел `Blanc` будет добавлен, он будет третьим узлом в списке. Теперь все, что происходит справа от `Blanc`, нас не интересует.

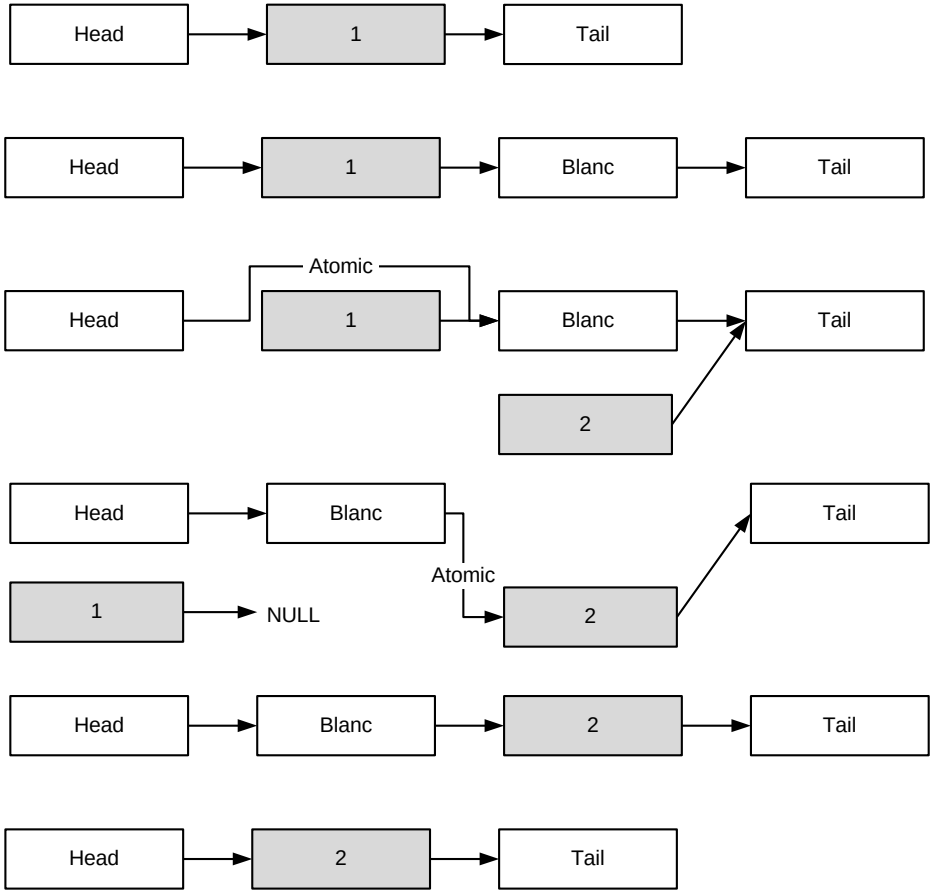


Рис. 5.7. Удаление узлов из списка

Для удаления второго узла списка мы используем конструкцию:

```
head.next.fetchAndStoreOrdered(&blank);
```

которая просто заставляет `head.next` указывать на узел `Blanc`. Метод `fetchAndStoreOrdered()` — это атомарный способ присвоить указателю новое значение. Дальше можно удалить узел `Blanc`, если он не является единственным узлом между `Head` и `Tail`. Казалось бы, на этом операцию удаления второго узла списка можно было бы считать законченной, но это не так. Что произойдет, если в процессе удаления узла другой поток вызовет процедуру, сканирующую список путем перехода по полю `next`? Для того чтобы ответить на этот вопрос, необходимо разобраться, на что указывают поля `next` узлов `Head` и `current` в процессе удаления узла.

Поле `next` узла `Head` указывает либо на узел `current`, либо на узел `Blanc` (который в этот момент является частью списка), либо на узел, следующий за `current`. Поле `next` узла `current` либо указывает на узел `Blanc` (который в этот момент является частью списка), либо содержит значение `NULL`. Если в процессе сканирования спи-

ска функция (а таких функций у нас две: `insertRecord()` и `empty()`) обнаружит значение `NULL`, она либо вернет значение `false`, (это случай `insertRecord()`, для которой возврат `false` означает, что она не смогла добавить новый узел в ходе данной операции и надо повторить попытку), либо начнет сканирование заново, с узла `Head` (функция `empty()`). Таким образом, сканирование списка не приведет к ошибкам на любом этапе удаления узла.

В классе `LLQueue` самые интересные методы объявлены в разделе `protected`. Это сделано потому, что класс `LLQueue` является базовым для классов, которые действительно работают с очередью. В нашем случае это класс `LLFIFO` (листинг 5.26).

Листинг 5.26. Класс `LLFIFO`

```
#ifndef LLFIFO_H
#define LLFIFO_H
#include "llqueue.h"
class LLFIFO : public LLQueue
{
    Q_OBJECT
public:
    explicit LLFIFO(QObject *parent = 0);
    ~LLFIFO();
    void fetch(LLRecord * record);
    LLRecord * unfetch();
    void release(LLRecord * record);
};
#endif // LLFIFO_H

#include "llfifo.h"

LLFIFO::LLFIFO(QObject *parent) :
    LLQueue(parent)
{}

LLFIFO::~LLFIFO()
{
    LLRecord * r;
    while ((r = unfetch()) != 0)
        release(r);
}

void LLFIFO::fetch(LLRecord * record)
{
    record->inUse = true;
    record->recordType = RecordTypeOrdinary;
    while(!insertRecord(record, &tail));
}
```

```
LLRecord * LLFIFO::unfetch()
{
    return removeFirstRecord();
}

void LLFIFO::release(LLRecord * record)
{
    record->inUse = false;
}
```

Как ясно из названия, этот класс реализует список, который следует правилам FIFO. Метод `fetch()` служит для добавления записей в список. Этот метод подготавливает узел `LLRecord` к добавлению и затем вызывает метод `insertRecord()` базового класса до тех пор, пока тот не вернет значение `true` (а это произойдет обязательно за конечное число итераций). Метод `unfetch()` извлекает первый информационный элемент из списка. Он просто вызывает метод `removeFirstRecord()` базового класса.

Обратите внимание, что наши классы `LLQueue` и `LLFIFO` ничего не знают о том, откуда берутся экземпляры структуры `LLRecord` и что происходит с этими экземплярами, когда они становятся ненужными. Это сделано специально, т. к. управление динамической памятью при работе со списками лучше вынести за пределы списка. В простой реализации списка память выделяется динамически для каждого нового узла и освобождается после обработки этого узла. Подобные выделения/освобождения небольших блоков памяти могут серьезно замедлить работу системы. Мы можем существенно оптимизировать работу с памятью, если заметим, что программам, которые используют связные списки, часто требуются блоки одного и того же размера. Самая простая стратегия — создать массив блоков, размеры которого будут примерно соответствовать потребностям программы. Если в процессе работы программы обнаружится, что элементов массива не хватает, программа может увеличить массив одним из доступных способов. При использовании этой стратегии выделение нового блока для вставки в список будет сводиться к установке отметки о том, что данный блок используется, а освобождение блока — к снятию этой отметки.

Поскольку мы имеем дело с конкурентным доступом к пулу блоков, перед нами может возникнуть проблема синхронизации меток. А может и не возникнуть. Если все потоки используют один общий пул, то нужно предпринять специальные меры для того, чтобы несколько потоков одновременно не пометили один и тот же блок как используемый. Но мы пойдем другим путем. У каждого потока будет свой пул блоков. В этом случае проблема синхронизации меток отпадает, поскольку метку об использовании блока может поставить только поток-владелец пула, а снять эту метку может только читатель. В нашем примере роль метки играет поле `inUse` структуры `LLRecord`.

В рассматриваемом примере метка об использовании блока устанавливается в методе `fetch()`, который вызывается потоками-писателями. Метка снимается читате-

лем с помощью метода `release()`, после того, как читатель обработает содержимое блока. Если вы захотите использовать общий пул для всех потоков, вам потребуется переписать метод `fetch()` так, чтобы из всех конкурирующих потоков-писателей только один мог пометить блок, как используемый, и чтобы этот поток знал, что метку поставил именно он. Этого можно добиться, например, сменив тип поля `inUse` с `bool` на `QAtomicInt`. При этом поток-читатель по-прежнему может сбрасывать метку неатомарным способом, поскольку конкурентов у него нет.

> ***Примечание***
>
> Что лучше, общий пул блоков для всех потоков или индивидуальный пул для каждого потока? С одной стороны, у общего пула есть преимущества. Если потоки добавляют блоки в список с разной интенсивностью, общий пул гарантирует более "справедливое" распределение блоков, при котором поток с большей интенсивностью получит больше блоков. Кроме того, если один из потоков завершится аварийно, с общим пулом памяти ничего не случится и целостность связного списка не будет нарушена. С другой стороны, общий пул придется увеличивать всякий раз, когда мы добавляем новый поток, а это может быть трудоемкой операцией. Да и проблема синхронизации меток не упрощает дело. Кроме того, поиск свободных блоков в общем пуле будет занимать больше времени, чем поиск свободных блоков в индивидуальном пуле каждого потока. Конечно, общий пул можно реализовать не в виде линейной структуры, а в виде дерева, но это создаст дополнительные проблемы синхронизации. Так что ответ на вопрос о преимуществах двух типов пулов не однозначен и его исследование выходит за рамки этой книги.

В нашем примере мы используем несколько потоков класса `MessageThread` (листинг 5.27) для записи сообщений в одну очередь `LLFIFO`.

Листинг 5.27. Класс потока-писателя

```
#ifndef MESSAGETHREAD_H
#define MESSAGETHREAD_H
#include "extthread.h"
class LLFIFO;
struct LLRecord;
const int MaxRecordsPool = 256;
class MessageThread : public ExtThread
{
    Q_OBJECT
public:
    explicit MessageThread(QObject *parent = 0);
    ~MessageThread();
    void setFIFO(LLFIFO * fifo);
protected:
    void run();
private:
    LLFIFO * fifo;
    LLRecord * records;
};
#endif // MESSAGETHREAD_H
```

```
#include "messagethread.h"
#include "llfifo.h"
#include <string.h>
#include <QString>
#include <QDateTime>

MessageThread::MessageThread(QObject *parent) :
    ExtThread(parent)
{
    records = new LLRecord[MaxRecordsPool];
    for (int j = 0; j < MaxRecordsPool; j++) {
        records[j].inUse = false;
        records[j].recordType = RecordTypeOrdinary;
        records[j].message[0] = '\0';
    }
}
MessageThread::~MessageThread()
{
    delete [] records;
}

void MessageThread::setFIFO(LLFIFO *fifo)
{
    this->fifo = fifo;
}

void MessageThread::run()
{
    for (int i = 0; i < MaxRecordsPool*2; i++) {
        for (int j = 0; j < MaxRecordsPool; j++) {
            if (!records[j].inUse) {
                QString s = "%1 — thread %2 — message %3";
                s = s.arg(QDateTime::currentDateTime().toString())
                    .arg((quint64) QThread::currentThreadId()).arg(i);
                memcpy(records[j].message, s.toLocal8Bit().data(),
                    s.toLocal8Bit().length());
                records[j].message[s.toLocal8Bit().length()] = 0;
                fifo->fetch(&records[j]);
                break;
            }
        }
        QThread::msleep(10);
    }
    done();
}
```

Обратите внимание на то, что пул блоков создается динамически, в куче, а не как локальная переменная метода `run()`. Это делается для того, чтобы в случае аварий-

ного завершения потока блоки, часть которых может все еще принадлежать общему списку, не были уничтожены. Конечно, в этом случае у нас возникнет утечка памяти, но это все же лучше, чем крушение всего приложения из-за ошибки обращения к памяти. Обратите внимание также на то, что мы ничего не делаем для увеличения количества блоков, если их не хватает. Вы можете сами придумать, как реализовать такое увеличение.

> ***Примечание***
>
> От чего зависит необходимое количество блоков? Прежде всего, оно зависит от средней фактической длины списка. Представим, что все блоки, которыми располагает поток-писатель, добавлены в список. Они все будут помечены как используемые до тех пор, пока поток-читатель не прочитает хотя бы один из них и не вызовет метод `release()` класса `LLFIFO`. Если потоку-писателю понадобится отправить сообщение до наступления этого времени, ему придется увеличивать размер пула. В этой ситуации было бы лучше использовать общий пул, т. к. один поток мог бы использовать блоки, освобожденные после обработки сообщений других потоков.

В нашем приложении-примере создается несколько потоков (листинг 5.28), которые тут же запускаются на выполнение.

Листинг 5.28. Создание очереди и нескольких потоков-писателей

```
fifo = new LLFIFO();
for (int i = 0; i < 40; i++) {
     MessageThread * thread = new MessageThread();
     thread->setFIFO(fifo);
     thread->start();
}
startTimer(5);
```

В процедуре `timerEvent()` главного потока, который является по совместительству потоком-читателем, сообщения считываются из очереди `fifo` и добавляются в компонент `textBrowser` (листинг 5.29).

Листинг 5.29. Чтение сообщений из очереди

```
void Dialog::timerEvent(QTimerEvent *event)
{
    if (!fifo->empty()) {
        LLRecord * record = fifo->unfetch();
        ui->textBrowser->append(QString::fromLocal8Bit(record->message));
        fifo->release(record);
    }
}
```

Можете убедиться, что наша безблокировочная очередь добавляет сообщения с невероятной быстротой (рис. 5.8).

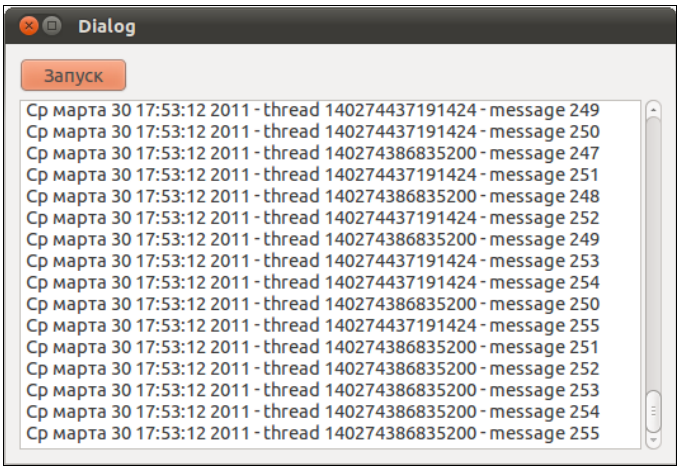


Рис. 5.8. Программа, демонстрирующая обработку сообщений потоков

ГЛАВА 6

Высокоуровневый интерфейс потоков

Возможно, вы удивитесь, если узнаете, что, все, чем мы занимались до сих пор, представляет собой работу с потоками на низком уровне. Но в терминологии Qt это так и есть. Начиная с версии Qt 4.4, у библиотеки появился другой интерфейс, призванный упростить работу с потоками. Этот интерфейс сосредоточен в пространстве имен `QtConcurrent`. Он также включает в себя классы `QRunnable` и `QThreadPool`. Новый интерфейс преследует две задачи: упростить работу с потоками, когда все, что нам нужно, — выполнить в дополнительном потоке одну функцию (это одна из тех задач, которые мы решили в классе `ExThread`). Вторая задача, которую стремится решить новый интерфейс, — упростить управление множеством потоков программы. Как я уже говорил, большое количество потоков не всегда ускоряет работу программы. Необходимость синхронизации данных между потоками — это лишь одна из проблем. Другая проблема — довольно высокая стоимость (в терминах машинного времени) создания нового потока. Именно эта проблема выходит на передний план в таких приложениях, как серверы, которые обрабатывают множество запросов разных клиентов и не сохраняют информацию о клиентах между запросами (stateless servers).

Одно из популярных ныне решений этой проблемы — разделение физических и логических потоков. Идея заключается в том, что, когда приложению требуется много потоков, среда, в которой выполняется приложение, создает ограниченное количество "физических потоков", т. е. потоков, которые воспринимаются как таковые на уровне операционной системы. Далее "логические потоки", которые создает программа, распределяются между физическими потоками наиболее оптимальным (с точки зрения среды выполнения) образом. То есть один физический поток может выполнять несколько логических потоков. Самый существенный выигрыш при этом достигается благодаря тому, что создание и уничтожение логического потока (который настоящим потоком не является) обходится системе гораздо дешевле, чем создание физического потока.

Классы *QRunnable* и *QThreadPool*

Класс `QRunnable` предназначен для выполнения непродолжительных (в терминах машинного времени) процедур. Так же как и в случае `QThread`, для того чтобы создать процедуру, выполнимую с помощью `QRunnable`, необходимо создать класс-

потомок этого класса и перекрыть в нем метод `run()`. Разница между `QThread` и `QRunnable` заключается в том, что класс `QRunnable` не создает поток, а просто инкапсулирует фрагмент кода, который необходимо выполнить. Класс `QRunnable` не является потомком `QObject`. Этот класс не может отправлять или получать сигналы и не может обрабатывать события (но может посылать события другим классам). Можно сказать, что класс `QRunnable` представляет собой максимально легкую обертку вокруг фрагмента кода, который нужно выполнить в отдельном потоке. Выполнение этих фрагментов кода осуществляется с помощью объектов класса `QThreadPool`. Если пользоваться введенной выше терминологией логических и физических потоков, можно сказать, что `QRunnable` создает логические потоки, а `QThreadPool` распределяет их между физическими потоками. В теории класс `QThreadPool` должен полностью скрыть от нас низкоуровневую реализацию потоков и обеспечить оптимальное распределение нагрузки между потоками. На практике это не всегда удается. Тем не менее класс `QThreadPool` заслуживает подробного обсуждения. Для того чтобы поставить объект класса `QRunnable` в очередь на выполнение, необходимо вызвать метод `start()` класса `QThreadPool`. Если в распоряжении `QThreadPool` есть свободные потоки, то метод `QRunnable::run()` будет выполнен немедленно. Если свободных потоков нет, объект-потомок `QRunnable` будет добавлен в очередь, и соответствующий метод будет выполнен позже. Метод `start()` принимает два параметра: указатель на объект класса `QRunnable` и численный приоритет, согласно которому будет определен порядок этого объекта в очереди, если его придется туда добавлять.

Важно понимать, что объект `QThreadPool` полностью берет на себя управление переданным ему объектом класса-потомка `QRunnable`. По умолчанию `QThreadPool` автоматически уничтожает этот объект, когда его метод `run()` завершит работу, так что мы можем не заботиться об этом. Но это означает также, что мы не должны обращаться к объекту, переданному под управление `QThreadPool`, напрямую.

Сколько физических потоков создает объект `QThreadPool`? Вообще говоря, количеством потоков можно управлять с помощью свойства `maxThreadCount` объекта `QThreadPool`. По умолчанию это свойство имеет значение, равное значению, возвращаемому методом `QThread::idealThreadCount()`. Если вы думаете, что "идеальное" количество потоков вычисляется методом `idealThreadCount()` по какому-то сложному алгоритму, то ошибаетесь. Обычно возвращаемое методом значение равно количеству ядер процессора системы (физических или виртуальных). Самый простой и верный способ изменить количество потоков, доступных объекту `QThreadPool` — использовать методы `reserveThread()` и `releaseThread()`. Метод `reserveThread()` сообщает объекту `QThreadPool`, что мы хотим использовать один поток за пределами этого объекта, и поэтому объект должен уменьшить количество используемых потоков. Метод `releaseThread()` "возвращает" поток объекту `QThreadPool`, т. е. попросту сообщает объекту, что он может снова увеличить количество используемых потоков.

Объект класса `QThreadPool` не держит потоки в рабочем состоянии постоянно. Если у объекта `QThreadPool` нет ожидающих обработки объектов `QRunnable`, то через некоторое время (определяемое свойством `expiryTimeout`) потоки просто завершатся.

Потоки будут созданы вновь, как только у объекта класса `QThreadPool` появится работа.

Из всего сказанного о классе `QThreadPool` следует, что в программе должен быть только один объект этого класса. Создавать объекты `QThreadPool` с помощью оператора `new` можно, но не нужно. Статический метод `globalInstance()` класса `QThreadPool` возвращает указатель на единственный, требуемый программе объект `QThreadPool`.

Что выбрать: *QRunnable* или *ExtThread*?

Как вы помните, класс `ExtThread` был предложен мной для того, чтобы уйти от модели событий в процедуре потока. Класс `QRunnable` позволяет решить ту же задачу, в связи с чем возникает вопрос, какой из классов лучше использовать в той или иной ситуации. `QRunnable` — очень "легкий" класс. Сам по себе он не создает никакого потока, но его можно встроить в существующий поток с помощью менеджера потоков, распределяя нагрузку между потоками. Когда объект `QRunnable` завершит свою работу, он может быть автоматически удален менеджером потоков. Эти достоинства `QRunnable` делают его оптимальным решением для ситуации, когда программе приходится выполнять множество сравнительно коротких процедур. Но у "легкого" `QRunnable` есть и недостатки. Поскольку этот класс не является потомком `QObject`, из него нельзя эмитировать сигналы. Если мы хотим, чтобы объект `QRunnable` обменивался сообщениями с другими объектами, нам придется делать это либо с помощью событий, либо с помощью специальных "многопоточных" структур данных (как это сделано в нашем примере). Мы вообще можем потерять все преимущества `QRunnable`, если попытаемся "утяжелить" его такими же средствами, как потомки класса `QThread`. В общем, можно сказать, что `QRunnable` лучше подходит для выполнения сравнительно простых и коротких операций, тогда как для длительных и сложных операций лучше воспользоваться потомками класса `QThread`.

> ***Примечание***
>
> А зачем вообще использовать `QRunnable` для простых и коротких операций? Не лучше ли разместить их в главном потоке приложения? Это зависит от количества простых и коротких операций, которые приходится выполнять. Если таких операций очень много, как, например, у сервера, обслуживающего множество клиентов, их лучше разместить в дополнительных потоках.

Еще одна проблема, связанная с использованием `QRunnable` и `QThreadPool`, — отсутствие стандартных средств досрочного завершения потока. Может показаться, что это не очень важно, поскольку объектам `QRunnable` обычно "поручаются" короткие процедуры, тем не менее проблема может возникнуть, если объектов `QRunnable` создано очень много, т. е. гораздо больше, чем выполняющих их физических потоков. В этом случае объект `QThreadPool` ставит поступающие объекты `QRunnable` в очередь и потоки не завершатся до тех пор, пока эта очередь не опустеет. У нас же нет ни возможности получить доступ к этим потокам, ни возможности принудительно опустошить очередь `QThreadPool`.

У этой проблемы есть несколько решений. Можно вместо метода `start()` пользоваться методом `tryStart()`, и, если метод возвращает `false`, т. е. если объект `QRunnable` не может быть поставлен на выполнение немедленно, помещать этот объект в собственную очередь и затем периодически вызывать `tryStart()` для объектов из очереди. В этом случае очередь объектов `QRunnable` будет у нас под контролем, и мы сможем очистить ее при необходимости досрочного завершения.

Другое решение (то, которым воспользовался я в своей демо-программе) заключается в том, чтобы объявить глобальную переменную, значение которой будут периодически проверять функции `QRunnable`.

Программа поиска файлов по содержимому

Для демонстрации возможностей классов `QRunnable` и `QThreadPool` мы рассмотрим программу, которая выполняет поиск файлов на компьютере по маске имени файла и по строке, содержащейся в этом файле. Исходные тексты программы вы найдете в папке Ch6/FileFinder на сайте **http://symmetrica.net/qt47book/**.

Помимо использования интерфейса программирования потоков высокого уровня, здесь мы затронем также некоторые другие полезные темы, а именно алгоритм Бойера—Мура для быстрого поиска в тексте и метод быстрой работы с файлами с помощью отображения их в память.

Структура программы FileFinder довольно сложна (рис. 6.1).

Поиск файлов с именем, соответствующим заданной маске, выполняет объект `scanner` класса `FSScan`. По сути, этот класс представляет собой модифицированный класс `FileFinder` из программы extthreads, рассмотренной в *главе 5*. Иначе говоря, поиск файлов с подходящим именем выполняется объектом класса-потомка класса `ExtThread`. Когда такой файл найден, поток эмитирует сигнал `scanFile()`, предназначенный для объекта `taskManager` класса `TaskManager`. Этот класс управляет объектом класса `QThreadPool`, а также объектами класса `SearchTask`, которые являются потомками класса `QRunnable`.

Разумеется, необходимо определить, каким образом объекты `SearchTask` будут сообщать другим объектам о результатах своей работы. Как уже говорилось, эмитировать сигналы эти объекты не могут. Вместо этого мы воспользуемся безблокировочной очередью `LLFIFO`, в которую объекты `SearchTask` будут помещать свои сообщения, а заинтересованные объекты — считывать их. Как мы помним, при работе с безблокировочной очередью мы можем использовать два разных типа пулов записей — локальный и глобальный. Поскольку объекты `SearchTask` уничтожаются сразу после завершения своей задачи, локальный пул записей нам не подходит. Мы используем глобальный пул, который создаем в объекте класса `TaskManager`. Этот глобальный пул (переменная `records`) представляет собой массив записей, в котором объекты `SearchTask` ищут незанятую запись и помещают эту запись в очередь.

Слот класса `TaskManager::newTask()` (листинг 6.1) вызывается в ответ на сигнал `scanFile()`, посылаемый потокам `scanner`. В качестве параметра этот сигнал передает имя найденного файла.

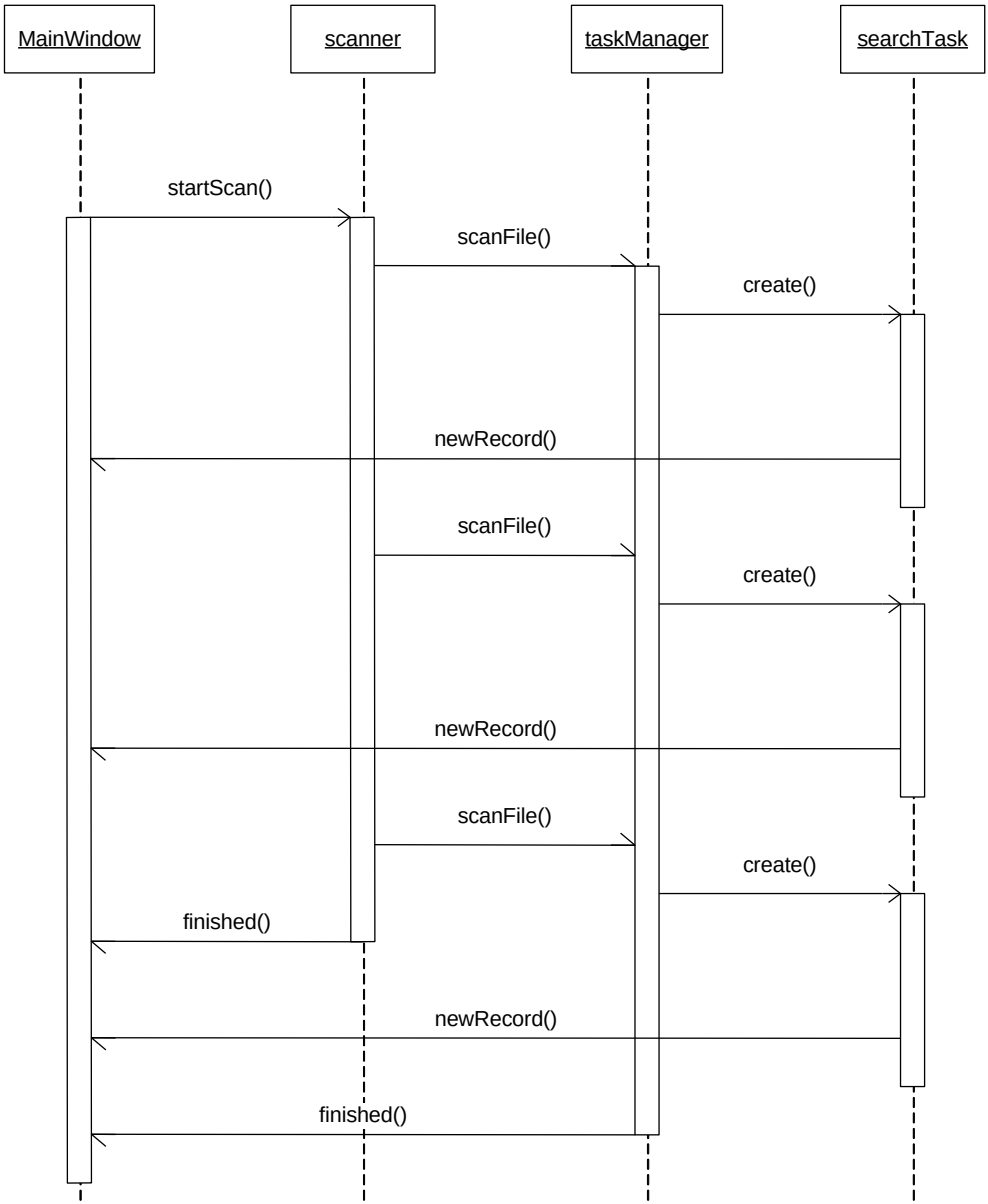


Рис. 6.1. Последовательность действий в программе FileFinder

Листинг 6.1. Слот `newTask()`

```
void TaskManager::newTask(QString fileName)
{
    globalStop = false;
    SearchTask * st = new SearchTask(fileName, m_pattern,
        SearchTask::Local8Bit|SearchTask::UTF8, records,
```

```
        MaxRecordsPool, fifo);
    threadPool->start(st);
}
```

Переменная `globalStop` используется для того, чтобы быстро завершить потоки объекта `QThreadPool`, если пользователь прерывает поиск или программа завершает свою работу. В этом случае переменной присваивается значение `true`. Каждый раз, когда мы создаем новую задачу для поиска, мы, на всякий случай, присваиваем этой переменной значение `false`.

В конструкторе объекта `SearchTask` мы передаем объекту полное имя файла, в котором следует искать строку, строку для поиска, набор флагов, определяющих, в каких кодировках следует выполнять поиск (программа FileFinder поддерживает поиск в локальной восьмибитной кодировке и в кодировке UTF-8), массив `records`, который содержит глобальный пул записей для очереди `fifo`, количество элементов в этом массиве и указатель на безблокировочную очередь, в которую объект `SearchTask` должен помещать свои сообщения.

Переменная `threadPool` содержит указатель на глобальный объект класса `QThreadPool`. Эта переменная инициализируется строкой

```
threadPool = QThreadPool::globalInstance();
```

и нужна, в основном, для удобства, чтобы нам не приходилось вызывать статический метод `QThreadPool::globalInstance()` всякий раз, когда потребуется обратиться к объекту `QThreadPool`.

Еще один интересный метод класса `TaskManager` — метод `waitForFinish()`. Этот метод приостанавливает вызвавшую его процедуру до тех пор, пока потоки, управляемые объектом `QThreadPool`, не завершат свою работу. Проблема в том, что, в отличие от объектов `QThread`, объект `QThreadPool` не посылает никаких сигналов, информирующих другие потоки о том, что его потоки завершились. У класса `QThreadPool` есть только блокирующий метод `waitForDone()`, который позволяет дождаться завершения потоков. Нам же нужен асинхронный способ получения той же информации. Будем надеяться, что в следующих версиях разработчики Qt решат эту проблему, а пока прибегнем к хитрости (листинг 6.2).

Листинг 6.2. Метод `waitForFinish()`

```
void TaskManager::waitForFinish()
{
    while (threadPool->activeThreadCount() > 0)
        QApplication::processEvents();
}
```

Этот метод просто ждет, пока количество работающих потоков (возвращаемое методом `activeThreadCount()`) не станет равно нулю. Внутри цикла вызывается метод обработки событий приложения, так что, хотя функция, вызвавшая `waitForFinish()`, будет приостановлена, главный поток приложения продолжит

обработку событий, даже если приостановленная функция была вызвана в этом потоке.

Тепер подробно рассмотрим класс `SearchTask`, поскольку он является, пожалуй, самым интересным классом программы FileFinder (листинг 6.3).

Листинг 6.3. Класс `SearchTask`

```
class SearchTask : public QRunnable
{

public:
    enum Encoding {
        Local8Bit = 0x0001,
        UTF8 = 0x0002
    };
    Q_DECLARE_FLAGS(Encodings, Encoding)
    explicit SearchTask(const QString &aFileName,
        const QString &aPattern, SearchTask::Encodings encodings,
        LLRecord * records, int recCount, LLFIFO * aFIFO);
    void run();
private:
    QString fileName;
    QString pattern;
    LLRecord * recordPool;
    LLFIFO * fifo;
    int recordsCount;
    void newRecord(const QString &fileName);
    bool doUtf8;
    bool doLocal8Bit;
};

Q_DECLARE_OPERATORS_FOR_FLAGS(SearchTask::Encodings)
```

Одна из особенностей класса, на которые я сразу хотел бы обратить ваше внимание, — объявление флагов, указывающих тип кодировки. На первый взгляд тут нет никаких неясностей. Тип `Encoding` — обычный перечислимый тип C++, который, во избежание конфликта имен, определен внутри класса, как это принято в Qt. Однако в конструкторе класса `SearchTask` мы хотим передавать не только отдельные элементы типа `Encoding`, но и их комбинации, объединенные с помощью оператора `|`. Можно, конечно, пойти простым путем и объявить:

```
typedef int Encodings;
```

Но в Qt принято поступать иначе. Макрос `Q_DECLARE_FLAGS(Encodings, Encoding)` создает тип `Encodings` (набор значений типа `Encoding`) на основе типа `Encoding`. Фактически этот макрос делает следующее объявление:

```
typedef QFlags<Encoding> Encodings;
```

где `QFlags` — шаблон класса, предназначенный для работы с флагами типа-параметра. Теперь у нас есть тип `Encodings`, объявленный, как принято в Qt. Но этого недостаточно. Для того чтобы новый тип мог работать, необходимо перегрузить операторы `QFlags`, такие как `|=`. К счастью, нам не придется делать это самим. Всю необходимую работу выполнит макрос `Q_DECLARE_OPERATORS_FOR_FLAGS()`.

Примечание

В чем преимущество такого способа определения типа `Encodings` перед более простым определением на основе `int`? Оно состоит в том, что мы получаем действительно уникальный тип для наших флагов, который не преобразуется автоматически в тип `int` и не запутает программиста.

Быстрый доступ к содержимому файла

Стандартные функции доступа к дисковым файлам не являются самыми быстрыми средствами доступа. Это относится ко всем операционным системам, т. к. следование давно утвержденному стандарту создает одинаковые проблемы для всех ОС. По мере развития современных операционных систем широкое применение нашла концепция виртуальной памяти, а вместе с ней и концепция отображения файлов в память процесса. Отображение файлов в память процесса позволяет достичь значительного выигрыша в производительности по сравнению со стандартными функциями `read()` и `write()` и их аналогами.

Чтобы понять преимущества отображения файлов в память, нужно схематично представить себе, что происходит, когда данные из файла читаются стандартной функцией. Детали этого процесса различаются в разных операционных системах, но в основных чертах они схожи. Когда функция `read()` направляет системе запрос на чтение файла, данные считываются с диска, обычно в большем количестве, чем запросила функция, и сохраняются в кэше операционной системы. Функция `read()` не имеет прямого доступа к кэшу. Обычно функции передается указатель на программный буфер — некоторую область памяти, выделенную в программе для данных файла. Операционная система должна скопировать данные из кэша в эту область. При отображении файла в память система не копирует данные в область, которую указываем ей мы, а отображает адреса своих внутренних буферов в адресное пространство нашего процесса. В результате мы "выигрываем" одну операцию копирования данных, а если программа обрабатывает множество файлов, как это делает программа поиска, то этот выигрыш может оказаться весьма заметным. Кроме того, отсутствие кэширования при отображении файлов в память может ускорить работу с большими файлами.

Выигрыш в быстродействии может быть достигнут также за счет того, что при обращении к файлу, отображенному в память процесса, системе не приходится поддерживать некоторые внутренние структуры, определяющие состояние доступа, которые необходимы при использовании стандартных интерфейсов.

Хотя, в принципе, даже очень большой файл можно отобразить в память процесса целиком, обычно это не делается, поскольку отображение в память очень большого

объема данных создает слишком большую нагрузку на менеджер виртуальной памяти. Вместо этого файл отображается в память процесса по частям, по мере необходимости. Допустим теперь, что программа нашла искомую строку в начале большого файла. В этом случае задача поиска в данном файле завершится (нам ведь важен факт наличия хотя бы одного вхождения строки в файл). Поскольку файл не кэшировался, с диска были прочитаны только те фрагменты, с которыми мы работали, а значит, количество обращений к диску было меньше, чем при использовании стандартных функций. Может возникнуть вопрос: а не обернется ли выигрыш, достигнутый при работе с большими файлами благодаря отсутствию кэширования, проигрышем при работе с небольшими файлами. На это мы ответим, что, поскольку кэширование на уровне системы отсутствует, мы можем позаботиться о нем сами и будем просто отображать небольшие файлы в память целиком. Что бы там ни происходило в системе на низком уровне, операция чтения файла, отображенного в память целиком, не может занять больше времени, чем операция чтения того же файла при включенном кэше. Осталось только ответить на вопрос, где проходит граница между "небольшими" и "большими" файлами. Ответ на этот вопрос нуждается в отдельном исследовании и в любом случае зависит от параметров конкретной системы.

Примечание

Если несколько процессов отображают в свое адресное пространство одну и ту же область одного и того же файла, то данные, записанные в этот файл одной программой, будут видимы другой программе еще до сохранения их на диске. Эта особенность файлов, отображенных в память, иногда используется для организации быстрого обмена данными между программами.

Из сказанного выше не стоит делать вывод, что стандартные интерфейсы работы с файлами устарели и больше не нужны. Во-первых, не всем программам требуется так интенсивно работать с файлами, как программе поиска. Во-вторых, идея, лежащая в основе стандартных интерфейсов (данные помещаются туда, куда нужно нам, а не туда, где это удобно операционной системе) часто бывает очень полезной. В-третьих, сила стандартных интерфейсов заключается в их способности к абстракции. Стандартные интерфейсы доступа к файлам позволяют представить в виде файлов что угодно — файл на диске, сетевой канал обмена данными, устройство, передающее данные в компьютер с помощью интерфейса USB, и многое другое.

Интерфейс отображения файлов в память появился в Qt, начиная с версии 4.4. Этот интерфейс невероятно прост. У класса `QFile` появились методы `map()` и `unmap()`. После того как файл открыт с помощью класса объекта класса `QFile`, его содержимое (или часть этого содержимого) можно отобразить в память процесса с помощью метода `map()`. Первый аргумент этого метода — смещение отображаемой области относительно начала файла. Второй аргумент — размер отображаемой области. Если отображение выполнено, метод возвращает указатель на начало области адресного пространства процесса, в которую выполнено отображение. В противном случае возвращается 0. Для того чтобы удалить отображение, мы вызываем метод `unmap()`, которому в качестве параметра передается адрес области отображения.

Примечание

В UNIX-системах широко распространены специальные типы файлов, которым не соответствует никакая область диска (такие файлы можно найти, например, в директориях /dev/ и /proc/). Некоторые из этих файлов можно отобразить в память процесса (как, например, файл /dev/zero), но для большинства из них сделать это не удастся.

В программе FlieFinder чтение файлов выполняет процедура `run()` класса `SearchTask` (листинг 6.4).

Листинг 6.4. Процедура `run()` класса `SearchTask`

```
void SearchTask::run()
{
    if (globalStop)
        return;
    QFile file(fileName);
    file.open(QIODevice::ReadOnly);
    qint64 size = file.size();
    int minLen = pattern.toLocal8Bit().length();
    int utf8Len = pattern.toUtf8().length();
    if (size < minLen)
        return;
    BMSearcher searcher((unsigned char*) pattern.toLocal8Bit().data(),
        minLen);
    BMSearcher utf8Searcher((unsigned char*) pattern.toUtf8().data(),
        utf8Len);
    if ((minLen == utf8Len) && (doLocal8Bit)) {
        if (!strncmp(pattern.toLocal8Bit().data(),
            pattern.toUtf8().data(), minLen))
            doUtf8 = false;
    }
    qint64 offset = 0;
    uchar * text;
    while (offset < size — minLen + 1) {
        if (globalStop)
            return;
        qint64 tlen = MSize < size — offset ? MSize : size — offset;
        text = file.map(offset, tlen);
        if (!text)
            return;
        if (searcher.containsPattern(text, tlen)) {
            newRecord(fileName);
            return;
        }
        if (doUtf8) {
            if (utf8Searcher.containsPattern(text, tlen)) {
                newRecord(fileName);
```

```
                return;
            }
        }
        file.unmap(text);
        if (utf8Len <= tlen)
            offset = offset + tlen — utf8Len + 1;
        else
            offset = offset + tlen — minLen + 1;
    }
}
```

Максимальный размер отображаемого блока определяется полем `MSize`, значение которого в программе соответствует 2 мегабайтам. Если размер файла меньше `MSize`, файл отображается в память процесса целиком, иначе — частями. Обратите внимание, что при чтении файла частями нам приходится считывать эти части с незначительным перекрытием (иначе мы можем пропустить вхождение строки, если оно находится на стыке двух частей). В этом интерфейс отображения файлов в память тоже гораздо удобнее, чем стандартный интерфейс.

Обратите внимание, что поиск в кодировке UTF-8 выполняется только тогда, когда это имеет смысл. С точки зрения используемого алгоритма при поиске в разных кодировках отличается только кодировка искомого образца. Но в случае, если искомая строка содержит только базовые символы латиницы, текст этой строки в кодировке UTF-8 будет совпадать с текстом в локальной восьмибитной кодировке. В этом случае отдельный поиск строки в кодировке UTF-8 можно не выполнять, и мы его не выполняем.

Если строка в файле найдена, мы помещаем в очередь `fifo` сообщение с именем файла. Это делает метод `newRecord` (листинг 6.5).

Листинг 6.5. Метод `newRecord`

```
void SearchTask::newRecord(const QString &fileName)
{
    int i = 0;
    while (!recordPool[i].BelongsTo.testAndSetOrdered(0, 1))
    i = (i+1)%recordsCount;
    int l = fileName.toLocal8Bit().size();
    l = l < 255 ? l : 254;
    strncpy(recordPool[i].message, fileName.toLocal8Bit().data(), l);
    recordPool[i].message[l] = '\0';
    fifo->fetch(&recordPool[i]);
}
```

Для передачи имен файлов используется локальная восьмибитная кодировка. Эта кодировка самая компактная, так что полное имя файла в ней, скорее всего, уло-

жится в выделенные 256 байт. Имена большинства файлов в вашей системе без проблем отобразятся в локальной восьмибитной кодировке, хотя возможны и исключения.

Обратите внимание, что теперь для пометки записи как используемой, мы применяем поле типа `QAtomicInt` (а не `bool`). Вызвано это тем, что пул сообщений теперь разделяется между несколькими потоками, а значит несколько потоков могут пытаться пометить одну и ту же запись.

Данные из очереди считываются в главном классе приложения с помощью таймера (листинг 6.6).

Листинг 6.6. Метод `timerEvent()`

```
void MainWindow::timerEvent(QTimerEvent *event)
{
    while (!taskManager->getFIFO()->empty()) {
        LLRecord * record = taskManager->getFIFO()->unfetch();
        QString str = QString::fromLocal8Bit(record->message);
        taskManager->getFIFO()->release(record);
        ui->textBrowser->append(str);
    }
    event->accept();
}
```

Быстрый поиск строк

Поиск вхождений строки в файле выполняется с помощью алгоритма Бойера—Мура. Все, что касается этого алгоритма, инкапсулировано в отдельный класс `BMSearcher` (листинг 6.7)

Листинг 6.7. Класс `BMSearcher`

```
class BMSearcher
{
public:
    BMSearcher(unsigned char * pattern, int length);
    bool containsPattern(unsigned char * text, int len);

private:
    void makeTable();
    int skipTable[256];
    int patternLength;
    unsigned char mpattern[256];
};
```

В конструкторе классу передается строка для поиска (переменная `pattern`).

Примечание

Строка `pattern`, как и текст, в котором выполняется поиск, приводится к типу "массив элементов `unsigned char`". Поскольку наш класс создан для поиска в тексте в различных кодировках, мы не можем полагаться на символ `\0`, как на маркер конца строки, поэтому вместе с массивом `unsigned char` мы передаем значение, указывающее длину массива.

Метод `containsPattern()` возвращает `true`, если массив `text` содержит искомую строку, и `false` в противном случае. Алгоритм Бойера—Мура позволяет выполнять поиск вхождений строки в тексте существенно быстрее, чем алгоритм "грубой силы", т. к. при использовании этого алгоритма нет необходимости сравнивать каждый символ текста с начальным символом строки. Те, кого интересуют подробности, могут почитать мою статью об алгоритме Бойера—Мура "Алгоритмы поиска в тексте" (**http://symmetrica.net/algorithms/bm1.htm**). Из этой статьи вы узнаете, между прочим, что использованный в примере FileFinder вариант алгоритма не является самым быстродействующим и не позволяет реализовать некоторые дополнительные полезные возможности, такие как поиск без учета регистра. Вы, если хотите, можете доработать программу FileFinder, реализовав более совершенный вариант алгоритма Бойера—Мура, тогда у вас появится программа, способная конкурировать с "настоящими" поисковыми программами, которые входят в состав операционных систем. Не вдаваясь в принцип работы алгоритма Бойера—Мура, отметим одну неочевидную особенность этого алгоритма. Алгоритм работает тем быстрее, чем длиннее искомая строка.

Теперь мы можем посмотреть на работающее приложение (рис. 6.2).

Увидеть, что программа работает, конечно, хорошо, но когда речь идет о многопоточном приложении, вполне естественно желание заглянуть глубже, и увидеть, как многопоточность реализуется на практике.

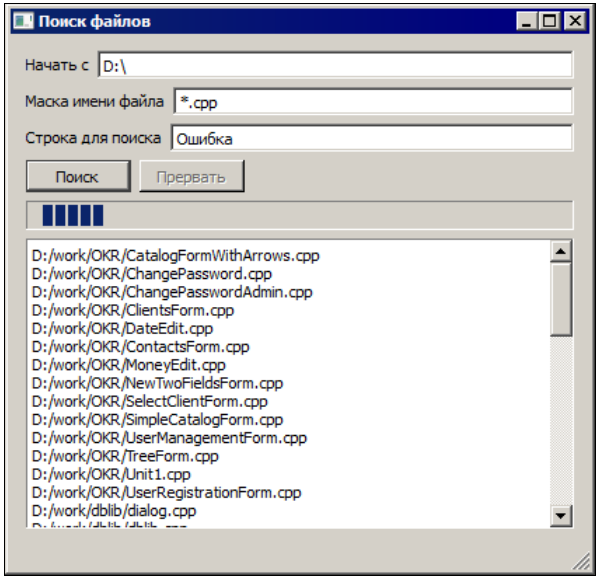
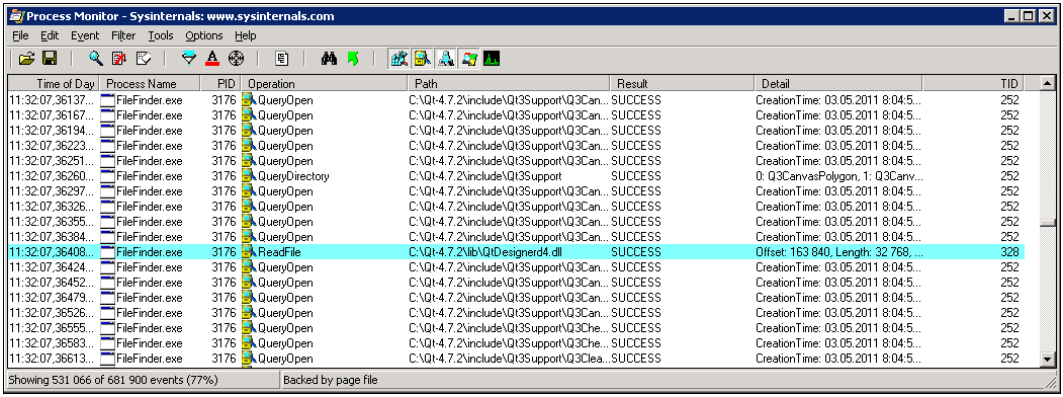


Рис. 6.2. Программа FileFinder в работе

На платформе Windows для этого можно воспользоваться утилитой Process Monitor, написанной Марком Руссиновичем и доступной на сайте Microsoft (**http://technet.microsoft.com/ru-ru/sysinternals/bb896645**). Программа Process Monitor (рис. 6.3) позволяет получить множество информации о выполняющихся в системе процессах, в том числе о вызовах системных функций процессами с указа-



Time of Day	Process Name	PID	Operation	Path	Result	Detail	TID
11:32:07,36137...	FileFinder.exe	3176	QueryOpen	C:\Qt-4.7.2\include\Qt3Support\Q3Can...	SUCCESS	CreationTime: 03.05.2011 8:04:5...	252
11:32:07,36167...	FileFinder.exe	3176	QueryOpen	C:\Qt-4.7.2\include\Qt3Support\Q3Can...	SUCCESS	CreationTime: 03.05.2011 8:04:5...	252
11:32:07,36194...	FileFinder.exe	3176	QueryOpen	C:\Qt-4.7.2\include\Qt3Support\Q3Can...	SUCCESS	CreationTime: 03.05.2011 8:04:5...	252
11:32:07,36223...	FileFinder.exe	3176	QueryOpen	C:\Qt-4.7.2\include\Qt3Support\Q3Can...	SUCCESS	CreationTime: 03.05.2011 8:04:5...	252
11:32:07,36251...	FileFinder.exe	3176	QueryOpen	C:\Qt-4.7.2\include\Qt3Support\Q3Can...	SUCCESS	CreationTime: 03.05.2011 8:04:5...	252
11:32:07,36260...	FileFinder.exe	3176	QueryDirectory	C:\Qt-4.7.2\include\Qt3Support	SUCCESS	0: Q3CanvasPolygon, 1: Q3Canv...	252
11:32:07,36297...	FileFinder.exe	3176	QueryOpen	C:\Qt-4.7.2\include\Qt3Support\Q3Can...	SUCCESS	CreationTime: 03.05.2011 8:04:5...	252
11:32:07,36326...	FileFinder.exe	3176	QueryOpen	C:\Qt-4.7.2\include\Qt3Support\Q3Can...	SUCCESS	CreationTime: 03.05.2011 8:04:5...	252
11:32:07,36355...	FileFinder.exe	3176	QueryOpen	C:\Qt-4.7.2\include\Qt3Support\Q3Can...	SUCCESS	CreationTime: 03.05.2011 8:04:5...	252
11:32:07,36384...	FileFinder.exe	3176	QueryOpen	C:\Qt-4.7.2\include\Qt3Support\Q3Can...	SUCCESS	CreationTime: 03.05.2011 8:04:5...	252
11:32:07,36408...	FileFinder.exe	3176	ReadFile	C:\Qt-4.7.2\lib\QtDesignerd4.dll	SUCCESS	Offset: 163 840, Length: 32 768, ...	328
11:32:07,36424...	FileFinder.exe	3176	QueryOpen	C:\Qt-4.7.2\include\Qt3Support\Q3Can...	SUCCESS	CreationTime: 03.05.2011 8:04:5...	252
11:32:07,36452...	FileFinder.exe	3176	QueryOpen	C:\Qt-4.7.2\include\Qt3Support\Q3Can...	SUCCESS	CreationTime: 03.05.2011 8:04:5...	252
11:32:07,36479...	FileFinder.exe	3176	QueryOpen	C:\Qt-4.7.2\include\Qt3Support\Q3Can...	SUCCESS	CreationTime: 03.05.2011 8:04:5...	252
11:32:07,36526...	FileFinder.exe	3176	QueryOpen	C:\Qt-4.7.2\include\Qt3Support\Q3Can...	SUCCESS	CreationTime: 03.05.2011 8:04:5...	252
11:32:07,36555...	FileFinder.exe	3176	QueryOpen	C:\Qt-4.7.2\include\Qt3Support\Q3Che...	SUCCESS	CreationTime: 03.05.2011 8:04:5...	252
11:32:07,36583...	FileFinder.exe	3176	QueryOpen	C:\Qt-4.7.2\include\Qt3Support\Q3Che...	SUCCESS	CreationTime: 03.05.2011 8:04:5...	252
11:32:07,36613...	FileFinder.exe	3176	QueryOpen	C:\Qt-4.7.2\include\Qt3Support\Q3Clea...	SUCCESS	CreationTime: 03.05.2011 8:04:5...	252

Рис. 6.3. Программа Process Monitor отслеживает работу FileFinder

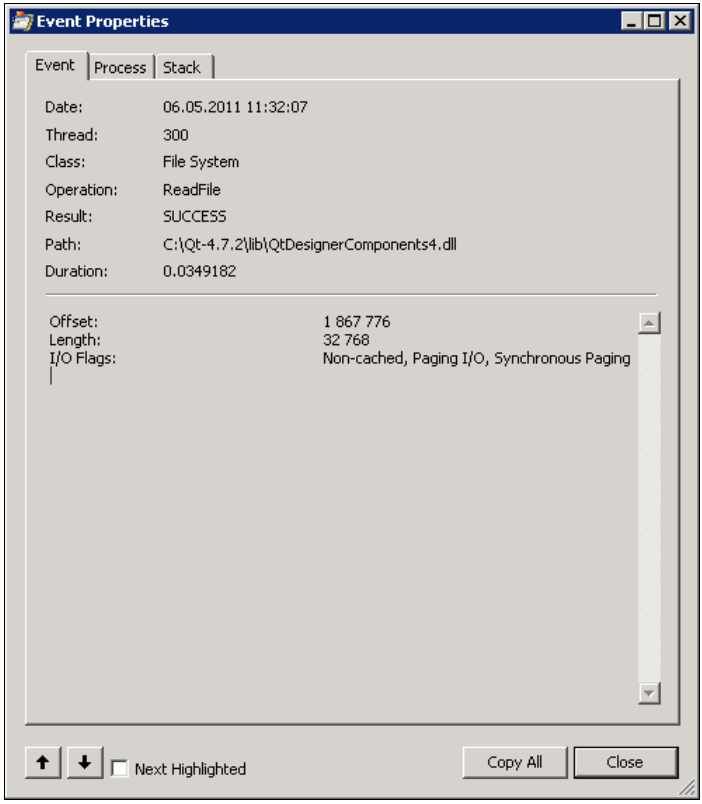


Рис. 6.4. Информация об операции `ReadFile`

нием аргументов, идентификаторов потоков, вызвавших функции, и т. п. Программа удобна еще и тем, что позволяет отфильтровать огромный объем поступающей информации по различным параметрам (например, по имени процесса). Выбрав интересующее нас событие, мы можем получить дополнительную информацию о нем (рис. 6.4).

Может показаться, что наша программа не способна продемонстрировать преимущества многопоточности в полную силу, т. к. ограничивающим фактором при работе программы является диск. Два потока не могут читать данные с одного диска одновременно (в данном случае имеется в виду настоящая, а не "кажущаяся" одновременность, к которой мы привыкли со времен распространения многозадачных ОС). Тем не менее в системе с двумя ядрами нам удалось довольно "справедливо" распределить нагрузку между ними обоими (рис. 6.5).

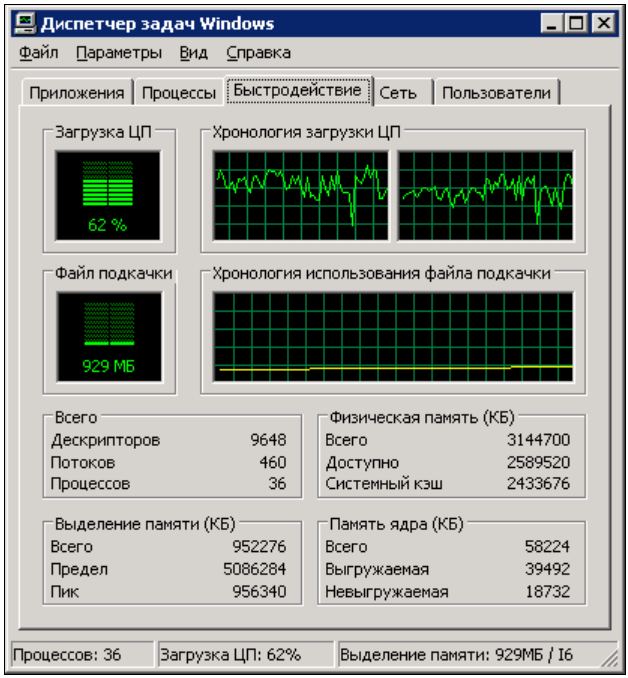


Рис. 6.5. Загрузка процессоров программой FileFinder

Что еще, помимо повышения производительности, дает нам использование потоков? Мы видим, что пользовательский интерфейс не "замерзает" во время работы программы, а результаты поиска в файлах начинают появляться до того, как будут отсканированы все файлы.

Функция *QtConcurrent::run()*

Функция `run()`, определенная в пространстве имен `QtConcurrent`, представляет собой интерфейс еще более высокого уровня, чем класс `QThreadPool`. Эта функция позволяет выполнить переданную ей функцию в отдельном потоке, совершенно не

заботясь о потоках (всю заботу берет на себя глобальный объект `QThreadPool`, обращаться к которому напрямую не приходится). Для того чтобы использовать функцию `QtConcurrent::run()` в своей программе, вам нужно включить в текст программы заголовочный файл `<QtConcurrentRun>`.

Мы уже привыкли к тому, что функции, предназначенные для выполнения в отдельных потоках, имеют определенные ограничения. В частности такие функции не возвращают значений в основной поток. Причина этого очевидна. После вызова функции в отдельном потоке основной поток продолжает выполняться, и, если мы не примем специальных мер, основной поток не узнает о том, что функция в другом потоке завершилась и не сможет сохранить возвращенное ею значение. Интерфейс, связанный с функцией `QtConcurrent::run()`, решает проблему передачи возвращаемых значений (а также генерируемых функцией исключений) между потоками.

Рассмотрим простейший пример `QtConcurrent::run()`. Пусть у нас есть функция для вычисления чисел Фибоначчи (листинг 6.8).

Листинг 6.8. Функция `fibonacci()`

```
quint64 fibonacci(qint32 n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;
    quint64 a = 0;
    quint64 b = 1;
    for (qint32 i = 2; i <= n; i++) {
        b = a + b;
        a = b - a;
    }
    return b;
}
```

Для выполнения этой функции в отдельном потоке нам нужно определить некоторые структуры данных и вызвать функцию `QtConcurrent::run()`(листинг 6.9).

Листинг 6.9. Вызов функции `fibonacci()` в отдельном потоке

```
QFuture<quint64> future;
QFutureWatcher<quint64> watcher;
connect(&watcher, SIGNAL(finished()), this, SLOT(finished()));
watcher.setFuture(future);
qint32 n = 100;
future = QtConcurrent::run(fibonacci, n);
```

Шаблон класса с фантастическим названием `QFuture` предназначен для асинхронного получения результата функции, выполняемой в потоке, а также для получения

других событий, возникающих во время выполнения потока. Название `Future` отражает, вероятно, тот факт, что интересные события происходят с объектом `QFuture` не в момент присваивания, а после него.

> ***Примечание***
>
> Функция `QtConcurrent::run()` — не единственный элемент пространства `QtConcurrent`, который использует объекты `QFuture`. При использовании других функций, подобных `QtConcurrent::run()`, объекты `QFuture` могут даже управлять выполнением соответствующего потока. Описание этих элементов можно найти в документации по Qt.

Объект `future` получит значение функции `fibonacci()`, когда оно будет возвращено, но ничего не сообщит нам об этом. Мы можем периодически опрашивать объект `future`, используя метод `isFinished()`. Можно также воспользоваться методом `result()`, но этот метод заблокирует вызывающий поток до тех пор, пока функция `fibonacci()` не вернет значение. Если мы хотим, чтобы об изменении состояния потока, в котором выполняется функция `fibonacci()`, нас извещали сигналы, следует воспользоваться шаблоном класса `QFutureWatcher`. Класс на основе этого шаблона объявляется так же, как и класс на основе `QFuture` — в качестве параметра указывается тип значения, возвращаемого функцией. Затем с помощью метода `setFuture()` мы передаем объекту `QFutureWatcher` ссылку на конкретный объект `QFuture`. Обратите внимание, что все это необходимо выполнить до вызова функции `run()`, иначе события, которые мы хотим отслеживать, могут произойти в промежутке между вызовом `run()` и настройкой `QFutureWatcher`. Если у функции, вызываемой с помощью `run()`, есть параметры, эти параметры указываются при вызове `run()` после имени самой вызываемой функции. По окончании выполнения функции объект `QFutureWatcher` эмитирует сигнал `finished()`. В обработчике этого сигнала мы можем воспользоваться методом `result()` объекта `QFuture`. Поскольку работа функции уже завершилась, метод `result()` не заблокирует вызывающий поток.

Помимо аргументов, механизм `QtConcurrent` может передавать в другие потоки возникшие в рабочем потоке исключения. Правда, возможно это только в том случае, если поток-приемник вызвал блокирующую функцию (например, `QFuture::result()`) до появления исключения. Другими словами, для того чтобы "отловить" исключение в потоке `QtConcurrent`, поток-приемник должен приостановить свою работу до тех пор, пока поток `QtConcurrent` не завершится. Это, очевидно, сводит на нет преимущества использования потоков. В документации по Qt почему-то нет упоминания об этом факте.

Так что если в вашей функции потока может возникнуть ошибка, лучше продумать какой-то способ сообщить об этом главному потоку без использования исключений.

ГЛАВА 7

Возвращаемся к Interview Framework

Система Interview Framework может применяться для решения самого широкого круга задач. Например, с ее помощью можно несколькими щелчками мыши создать окно просмотра файловой системы. Однако чаще всего Interview Framework применяется при программировании приложений баз данных. Именно на приложениях этого типа будет сосредоточено наше внимание.

Класс *QSqlDatabase*

Многим программам для работы с базами данных, сколько бы таблиц они ни использовали, хватает одного-единственного соединения с базой данных. Можно, конечно, создавать соединения для каждого обращения к базе данных, но это, во-первых, неудобно, а во-вторых, в зависимости от настроек, одно соединение может заблокировать другие. Кроме того, такое поведение противоречит принципу модульности. Лучше всего использовать одно соединение с базой данных (если нет причин поступать иначе), создать и настроить его в одном модуле, а затем сделать доступным через внешний интерфейс всем остальным модулям программы. Библиотека Qt упрощает решение указанной задачи.

В Qt соединением с SQL СУБД управляет класс `QSqlDatabase`. Обычно объект этого класса создается с помощью статического метода `QSqlDatabase::addDatabase()`. Этот метод существует в нескольких вариантах, каждому из которых при вызове можно передать несколько аргументов. Один из обязательных аргументов метода — имя драйвера СУБД. Поскольку мы в своих примерах используем SQLite, строка инициализации соединения выглядит так:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
```

где `QSQLITE` — имя драйвера Qt SQLite. Если бы мы воспользовались PostgreSQL, программе нужно было бы указать драйвер с именем `QPSQL`. Если вы захотите использовать MySQL, вам понадобится драйвер `QMYSQL`. Если вы работаете под Windows, с серверами Microsoft SQL Server или Oracle, удобнее всего использовать драйвер ODBC. Система ODBC не только позволит вам забыть о таких проблемах как разные версии СУБД, но и вообще позволит легко перенастроиться с одной системы управления базами данных на другую.

По умолчанию драйверы скомпилированы в виде отдельных модулей, но если вы хотите, то можете встроить код драйвера в свое приложение. Для этого вам понадобятся исходные тексты Qt. Исходные тексты всех драйверов вы найдете в поддиректориях директории QTDIR/src/sql/drivers/.

> ***Примечание***
>
> Готовые сборки Qt обычно уже содержат модули для основных СУБД, поддерживаемых на данной платформе. Если вы собираете Qt сами, из исходных текстов, то в процессе настройки сборки вы должны указать, какие модули драйверов следует собирать. Под Windows зачастую достаточно драйверов SQLite и ODBC.

Метод `addDatabase()` возвращает объект класса `QSqlDatabase` (обратите внимание, именно объект класса, а не указатель и не ссылку). Этот объект уже "знает", с каким драйвером ему предстоит работать, но для настройки соединения с конкретным сервером баз данных этого, разумеется, недостаточно. Настройка соединения с СУБД Postgress может выглядеть, например, так, как показано в листинге 7.1.

Листинг 7.1. Настройка объекта `QSqlDatabase` для работы с Postgress

```
QSqlDatabase db = QSqlDatabase::addDatabase("QPSQL");
db.setHostName("localhost");
db.setDatabaseName("testdb");
db.setUserName("user");
db.setPassword("password");
```

Объекту `db` сообщаются адрес сервера (в этом примере — `localhost`), имя базы данных, имя пользователя и пароль.

> ***Примечание***
>
> Разумеется, следуя правилу разделения движка и политики, мы не будем закладывать конкретные значения всех этих параметров в код настоящей программы. Такие параметры, как адрес сервера и имя базы данных, должны храниться в файле настроек программы, а имя пользователя и пароль должны вводиться пользователем в процессе авторизации.

Для соединения с Microsoft SQL Server с помощью драйверов ODBC применяется другая последовательность (листинг 7.2).

Листинг 7.2. Настройка объекта `QSqlDatabase` для работы с ODBC

```
QSqlDatabase db = QSqlDatabase::addDatabase("QODBC");
db.setDatabaseName(QString("Driver={SQL SERVER};
Server=%1;Database={%2};
UID=%3;PWD=%4").arg(server, database, user, password));
```

Переменные `server`, `database`, `user`, `password` содержат, соответственно, адрес сервера, имя базы данных, имя пользователя и его пароль. Обратите внимание на фигурные скобки вокруг имени драйвера и имени сервера. Их необходимость вызвана

тем, что имя драйвера содержит пробелы, а SQL Server также поддерживает имена баз данных, состоящие из нескольких слов, разделенных пробелами. Фигурные скобки указывают на то, что все, что включено между ними, является именем базы данных, в том числе и пробелы. Использование фигурных скобок, как и все, что используется в строке инициализации драйвера ODBC, является частью синтаксиса команд ODBC и не имеет прямого отношения к Qt, так что вдаваться в подробности мы не будем.

В наших примерах мы используем движок SQLite как наиболее простой и кросс-платформенный. Последовательность для инициализации SQLite тоже одна из самых простых, — вам достаточно указать имя сервера и имя базы данных (листинг 7.3).

Листинг 7.3. Настройка объекта `QSqlDatabase` для работы с SQLite

```
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
db.setHostName(hostName);
db.setDatabaseName(dBName);
```

Настроенное соединение открывается с помощью метода `open()`. Этот метод возвращает значение типа `bool`, которое указывает, удалось ли установить соединение с сервером базы данных. Если `open()` возвращает `false`, наша программа выводит сначала краткое, а затем более подробное сообщение об ошибке (с помощью метода `db.lastError().text()`) и завершает работу.

> ***Примечание***
>
> Если ваше приложение использует разделяемые библиотеки Qt (а не статичные), необходимо позаботиться о том, чтобы файлы драйверов баз данных находились там, где программа сможет их найти. Одним из таких мест может быть директория sqldrivers, расположенная в директории программы. В дистрибутиве Qt драйверы хранятся обычно в поддиректории sqldrivers директории plugins.

После того как соединение с базой данных успешно установлено, мы можем приступить к созданию модели. Хотя все объекты, реализующие модели элементов базы данных, имеют в своих конструкторах параметр типа `QSqlDatabase`, значение этого параметра указывать не обязательно. Вы можете спросить, откуда в таком случае объекты модели "узнают", из какой базы данных, иначе говоря, из какого объекта `QSqlDatabase` они должны получить данные. Ответ на этот вопрос прост. При вызове статического метода `addDatabase()`, которым создается объект `QSqlDatabase`, можно указать уникальное имя соединения. Если этого не сделать, новое соединение с базой данных станет соединением по умолчанию. Получить объект, реализующий соединение по умолчанию, можно с помощью статического метода `QSqlDatabase::database()`, вызванного без параметров. Именно так любой модуль нашей программы может получить доступ к соединению с базой данных (при том условии, конечно, что программа использует только одно соединение). Заголовок метода `database()` выглядит так:

```
QSqlDatabase database (const QString & connectionName =
QLatin1String( defaultConnection ), bool open = true)
```

Это означает, что если бы мы работали с несколькими именованными соединениями, мы могли бы передавать методу `database()` имя требуемого соединения и получать соответствующий объект.

Схема работы Interview Framework

В общих чертах схему стандартного приложения Interview Framework (в том числе приложения баз данных) можно описать как "модель–делегат–представление" (рис. 7.1).

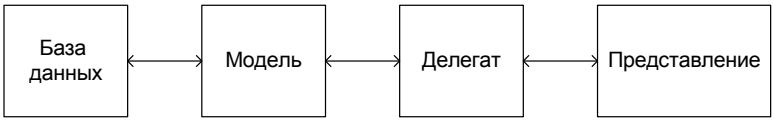


Рис. 7.1. Схема взаимодействия компонентов приложения Interview Framework

Мы обычно работаем напрямую с моделями и представлениями, работа делегатов скрыта от нас, хотя они играют очень важную роль.

Обсудим подробнее, как работает система "модель–делегат–представление" (рис. 7.2). Мы рассматриваем простейший случай, когда объекту класса `QTableView` требуется заполнить ячейку данными из связанного с этим объектом объекта класса-потомка `QAbstractItemModel` (все классы моделей, такие как `QSqlRelationalTableModel` или `QDirModel`, являются потомками этого класса). Для этого объект `QTableView` вызывает метод `paint()` своего объекта-делегата. У каждого объекта `QTableView` есть объект-делегат, даже если вы не создавали его явным

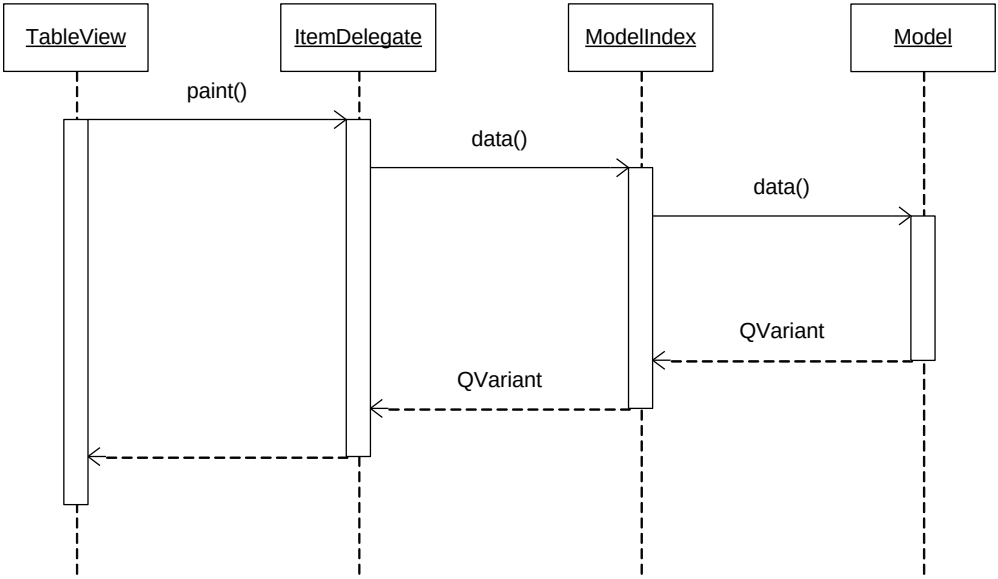


Рис. 7.2. Взаимодействие модели, делегата и представления данных

образом. По умолчанию это объект класса `QItemDelegate` или, начиная с Qt 4.4, `QStyledItemDelegate`. Этот объект вполне справляется с отображением данных модели (если данные могут быть отображены в виде текстовой строки) и умеет редактировать стандартные типы данных, однако с нестандартными ситуациями, особенно когда дело касается редактирования, он справиться не способен. В этом случае нам требуется специальный класс делегата, как, например, `QSqlRelationalDelegate`. Обычно специальные классы делегатов связаны со специальными моделями (исключение из этого правила представляют делегаты, предназначенные для отдельных столбцов и строк, которые мы рассмотрим ниже). Так, например, `QSqlRelationalDelegate` связан с моделью `QSqlRelationalTableModel`. С другими моделями его использовать не следует (да, скорее всего, и не получится).

Когда объекту, графически представляющему таблицу, требуется заполнить ячейку, он вызывает метод `paint()` объекта-делегата. Помимо прочего, методу `paint()` передается объект класса `QModelIndex`. Это еще один очень важный элемент цепочки "модель–делегат–представление". Объект класса `QModelIndex` содержит информацию, с помощью которой модель может найти требуемую ячейку данных. Поскольку в рамках Interview Framework все данные представляются в виде двумерных таблиц (даже если это линейный список или дерево), объект `QModelIndex` содержит поля `row` (строка) и `column` (столбец). Древовидные структуры отличаются от табличных тем, что их индексы содержат ссылки на индексы родительских, дочерних и "братских" элементов. Но во внутреннем представлении данные этих структур все равно хранятся в виде таблиц. В зависимости от конкретной структуры данных один из этих элементов может не использоваться. Объект-делегат вызывает метод `data()` объекта-индекса, который, в свою очередь вызывает метод `data()` объекта-модели. При этом делегат может трансформировать переданный ему индекс, если структура данных в графическом отображении не соответствует их структуре в модели данных.

Помимо индекса, объекту делегату и объекту-модели может быть передана еще и роль (`role`) — одно из специальных значений, которое указывает, какие именно данные требуется вернуть. Дело в том, что в последних реализациях Interview Framework, помимо собственно данных, модель данных может предоставить средствам графического отображения различные вспомогательные данные (поясняющий текст, элементы декоративного оформления и т. п.). Также параметр `role` указывает, требуются ли данные лишь для отображения или для редактирования.

Все методы `data()` класса-модели возвращают значение типа `QVariant`. Если стандартный объект-делегат запрашивает содержательные данные модели для отображения, он просто конвертирует это значение в `QString` и отображает его в соответствии с "подсказками", которые он получил от объекта-модели (если они были).

Объекты, графически представляющие данные модели, будь то таблица, дерево или список, позволяют редактировать существующие данные модели, но у них отсутствуют средства, с помощью которых можно было бы добавлять или удалять элементы. Такие методы есть у классов, реализующих модели данных. Нельзя не заметить, что в результате возникает некоторая раздробленность. Возможность удалять или

добавлять элементы модели должна быть реализована там же, где реализована и возможность редактировать эти элементы. Почему же возможности управления элементами моделей отсутствуют у графических виджетов, представляющих данные модели? Дело в том, что управление элементами моделей в значительной степени зависит от контекста приложения, так что разработчики Qt оставили нам в этом вопросе свободу выбора. Самое простое (и разумное), что мы можем сделать, — это возложить управление элементами модели на объекты `QAction`, добавить эти объекты в список `actions()` виджета, отображающего данные, и затем добавить в виджет, например, контекстное меню, в котором будут отображены соответствующие действия.

Пусть, например, у нас есть виджет `tableView`, отображающий данные некоторой табличной модели, и есть объект `actionAddRow` класса `QAction`, добавляющий строку в модель. Тогда создание контекстного меню для виджета `tableView` может выглядеть так:

```
tableView->addAction(actionAddRow);
tableView->setContextMenuPolicy(Qt::ActionsContextMenu);
```

В результате у виджета `tableView` появится контекстное меню, с помощью которого можно будет добавлять строки (этот прием используется в программе widgetmapperdemo, которую мы рассмотрим далее). Этот метод можно использовать для любого виджета, а не только виджета, предназначенного для отображения данных модели.

Говоря о моделях, предназначенных для взаимодействия с базами данных, нельзя не упомянуть параметр `EditStrategy`, который имеется у класса `QSqlTableModel` и его потомков, и определяет, в какой момент изменения, внесенные в модель данных, будут передаваться в базу данных. Существует три режима: `OnFieldChange`, `OnRowChange`, `OnManualSubmit`. При выборе первого режима модель данных пытается зафиксировать любые изменения в базе данных сразу после того, как она была изменена. Я не пользуюсь этим режимом и вам не советую. Во-первых, он затрудняет отмену ошибочного ввода данных, поскольку данные сразу же сохраняются на сервере базы данных. Во-вторых, он может просто не работать, когда пользователь добавляет в таблицу новые строки. Если для некоторых полей таблицы установлены ограничения `NOT NULL`, строку невозможно будет добавить в таблицу до тех пор, пока все эти поля не будут заполнены. Модель же будет пытаться добавить строку после каждого изменения, что вынудит сервер отправлять сообщения об ошибках. Еще один аргумент против использования режима `OnFieldChange` при работе с базами данных заключается в том, что приложение, как правило, должно проверить корректность введенных данных, прежде чем передавать их на сервер базы данных. Некоторые проверки могут быть встроены в сами виджеты с помощью масок ввода и объектов-валидаторов, но иногда перед отправкой данных требуются более сложные проверки.

В режиме `OnRowChange` модель будет пытаться внести изменения в базу данных после завершения пользователем редактирования данной строки. Момент, когда пользователь завершает редактирование строки, определяется сменой текущего индекса

(например, вызовом метода `setCurrentIndex()` класса `QDataWidgetMapper`). Этот режим передачи изменений на сервер гораздо лучше предыдущего и хорошо подходит для многих ситуаций.

В режиме `OnManualSubmit` мы сами определяем, когда изменения, внесенные в модель, будут переданы на сервер. Этот режим лучше всего подходит для сложных стратегий редактирования. При этом следует учитывать, что если в работе программы (или в работе сервера, или в соединении между программой и сервером) произойдет сбой, в режиме `OnManualSubmit` могут быть потеряны большие объемы введенных пользователем данных. В режиме `OnRowChange` мы рискуем потерять максимум одну строку, а в режиме `OnFieldChange` и того меньше — одну ячейку.

Утилита sqlite3

Прежде чем перейти к конкретному примеру, рассмотрим один весьма полезный сторонний инструмент. Поскольку в наших примерах приложений баз данных мы пользуемся встроенным движком SQLite, нам полезно иметь средство, которое бы позволяло просматривать и редактировать базы данных SQLite. Разумеется, такую программу можно написать и с помощью Qt, но, во-первых, зачем делать что-то, если это уже сделано, а во-вторых, для поиска возможных ошибок в программах нам требуется "эталонная" утилита управления базами данных SQLite. Речь идет о консольной утилите sqlite3. Если вы работаете под Linux, эта утилита, возможно, уже установлена в вашей системе (если нет, то ее легко установить из репозитория). Пользователи Windows могут загрузить дистрибутив с сайта **http://www.sqlite.org**. Утилита sqlite3 (рис. 7.3) позволяет редактировать подключенную базу данных, используя команды языка SQL. Кроме того, утилита реализует ряд дополнительных команд. Например, для загрузки базы данных из бинарного файла можно воспользоваться командой `.restore`.

Примечание

Имена всех дополнительных команд sqlite3 начинаются с точки. Краткую справку по командам можно получить, введя команду `.help` в командной строке утилиты.

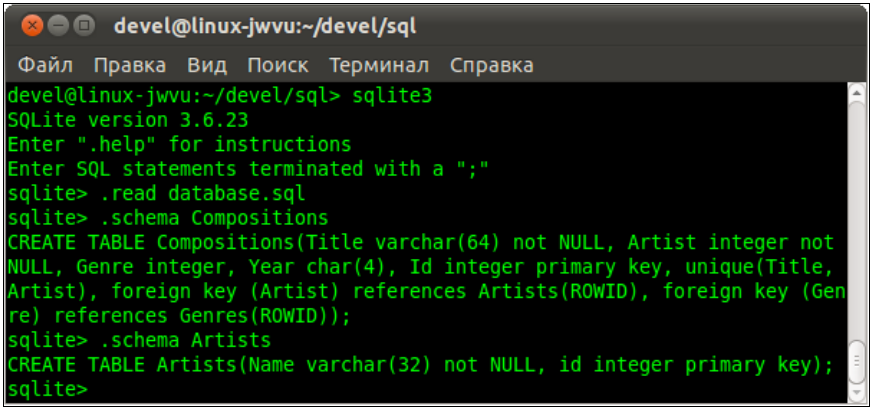


Рис. 7.3. Утилита sql3

Команда `.read` позволяет загрузить базу данных из текстового файла, содержащего описание базы данных на языке SQL, а команда `.dump` — получить полное описание базы данных на этом языке.

Отношения, допускающие значение *NULL*

Перейдем теперь к практическому примеру. В общих чертах вы уже знаете, что представляют собой модели данных. Мы не будем подробно описывать работу со стандартными моделями данных SQL — `QSqlQueryModel`, `QSqlTableModel` и `QSqlRelationalTableModel`. Все это есть в документации, которую я не собираюсь дублировать.

Вместо этого мы рассмотрим решение одной нетривиальной, но распространенной задачи. Когда я впервые упомянул модель `QSqlRelationalTableModel`, я отметил, что таблицы будут отображаться корректно только в том случае, если столбцы, ссылающиеся на внешние таблицы, не содержат значений `NULL`. Вообще говоря, это одно из правил хорошего стиля разработки реляционных баз данных. Грамотные разработчики запрещают добавление значений `NULL` в столбцы, являющиеся внешними ключами, ставя ограничение `NOT NULL`. А наиболее квалифицированные разработчики вообще не допускают значений `NULL` в своих базах данных. Иногда приходится слышать возражение, что значения `NULL` нужны потому, что при постепенном заполнении таблиц значения некоторых полей могут быть еще неизвестны. На это можно возразить, что поля с неизвестными значениями необязательно оставлять пустыми. Для таких случаев во внешних таблицах можно предусмотреть специальные значения, например "не заполнено", "не определено" и т. п. В общем, если бы мы жили в идеальном мире, мы могли бы просто не рассматривать ситуацию, когда столбцы, ссылающиеся на внешние таблицы, содержат значения `NULL`. Но мы живем в реальном мире, где такие таблицы все же встречаются и не всегда у нас есть возможность это исправить. Специально для таких случаев я написал набор классов, аналогичных классам, предназначенным для работы с `QSqlRelationalTableModel`. Особенность этих классов заключается в том, что они корректно отображают даже те строки таблиц, которые содержат значения `NULL` во внешних ссылках.

При этом я вовсе не пытаюсь оправдать небрежный стиль проектирования баз данных. Ценность рассмотренных ниже классов заключается не столько в том, что они позволяют обойти ошибки проектировщиков баз данных, сколько в том, что они позволяют нам увидеть работу системы "модель–делегат–представление" изнутри.

Вопрос о том, как заставить объекты класса `QSqlRelationalTableModel` отображать строки со значениями `NULL` во внешних ссылках, возникает на программистских форумах довольно часто. Самое распространенное решение, которое мне приходилось видеть, — модификация класса `QSqlRelationalTableModel` таким образом, чтобы SQL-запрос, формирующий таблицу, содержал объединение, построенное с помощью операторов `LEFT JOIN`. Самый большой недостаток такого подхода заключается в том, что приходится модифицировать один из стандартных классов Qt.

Модификация стандартных классов — это прием, к которому следует прибегать в последнюю очередь, когда все остальные средства не помогают. Проблема с модификацией стандартных классов заключается в том, что вы пытаетесь модифицировать нечто, находящееся не под вашим контролем (если, конечно, вы не можете сделать свою модификацию частью стандартной библиотеки Qt). Вам придется модифицировать каждый экземпляр Qt library, который вы используете для сборки своего приложения. А если в следующих версиях Qt внутренняя структура модифицированного вами класса изменится, ваши модификации вообще перестанут работать. Правильный путь расширения возможностей и адаптации Qt (как и любой другой библиотеки классов) — создание классов, производных от стандартных, причем эти производные классы должны опираться только на те интерфейсы стандартных классов, которые объявлены как публичные (в данном случае подразумеваются интерфейсы, содержащиеся в разделах `public` и `protected`). Поскольку разработчики Qt предполагают, что эти интерфейсы будут использоваться программистами, они стараются менять их как можно реже. А это значит, что ваши расширения будут максимально совместимы с различными версиями Qt.

Классы `WeakRelationalTable` и `WeakRelationalDelegate` подобны классам `QSqlRelationalTableModel` и `QSqlRelationalDelegate`, за исключением того, что они не теряют строки таблицы, в которых столбцы, ссылающиеся на внешние таблицы, содержат значения `NULL`. В директории примера вы найдете файлы music.bd и database.sql, которые содержат такую базу данных (первый — в бинарном формате SQLite, второй — в виде набора команд SQL). Текстовый файл SQL удобен для быстрого редактирования базы данных, а преобразовать его в бинарный файл базы данных, с которым может работать демонстрационная программа, можно с помощью описанной ранее утилиты sqlite3.

Класс *WeakRelationalTable*

Суть предлагаемого решения очень проста: вместо того, чтобы создать одно объединение, как это делается в классе `QSqlRelationalTableModel`, мы создаем программные модели нескольких таблиц — главной таблицы и таблиц, на которые она ссылается. Для программного представления таблиц используются объекты класса `QSqlTableModel`. Далее все отношения между таблицами реализуются чисто программными средствами. Сразу отмечу один недостаток такого подхода: если таблицы, с которыми работают наши классы, содержат очень много строк, обработка всех отношений в программе может занять много ресурсов процессора (хотя я постарался сделать такую обработку максимально быстрой с помощью алгоритма поиска методом половинного деления, что будет продемонстрировано далее). Однако на мой взгляд, этот недостаток не является существенным. Если таблицы баз данных содержат очень много строк, то приложения, работающие с базами данных, обычно не показывают все содержимое таблицы целиком (иначе пользователь просто растеряется при виде этого потока данных). В таких случаях обычно показывается только часть строк таблицы, отфильтрованная по какому-либо признаку. Класс `QSqlTableModel` умеет фильтровать данные, и наш класс `WeakRelationalTable` тоже способен делать это.

Впрочем, программная обработка отношений может быть не только недостатком, но и достоинством, а именно в том случае, если источником данных для таблиц является не SQL СУБД. На самом деле класс `WeakRelationalTable` был конвертирован из класса, используемого в реальном приложении, не предназначенном для работы с SQL. Как мы увидим, структура класса `WeakRelationalTable` такова, что объекты класса `QSqlTableModel` можно довольно легко заменить на объекты-таблицы, получающие данные не из СУБД. При этом сам класс `WeakRelationalTable` будет работать точно так же.

Исходный текст примера вы найдете в папке Ch7/weakdb на сайте **http://symmetrica.net/qt47book/**.

Сначала рассмотрим класс `WeakRelationalTable` (листинг 7.4).

Листинг 7.4. Объявление класса `WeakRelationalTable`

```
class WeakRelationalTable : public QAbstractTableModel
{
public:
      WeakRelationalTable(const QString &tableName);
      WeakRelationalTable(QObject * parent = 0,
         QSqlDatabase db = QSqlDatabase());
      ~WeakRelationalTable();
      void setTableName(const QString &tableName);
      bool select();
      void addRelation(const QString &column, const QString &table,
         const QString &idColumn, const QString &displayColumn);
      void setRelationFilter(const QString &column,
         const QString &filter);
      void setReadOnly(bool b);
      int rowCount (const QModelIndex & parent = QModelIndex()) const;
      int columnCount (const QModelIndex & parent = QModelIndex())
         const;
      QVariant data (const QModelIndex & index, int role =
         Qt::DisplayRole) const;
      QVariant data (int row, int column) const;
      QVariant data (int row, const QString &column) const;
      bool setData (const QModelIndex & index, const QVariant & value,
         int role = Qt::EditRole);
      bool setData (int row, int column, const QVariant & value);
      bool setData (int row, const QString &column, const QVariant
         &value);
      Qt::ItemFlags flags (const QModelIndex & index) const;
      QVariant headerData (int section, Qt::Orientation orientation,
         int role = Qt::DisplayRole ) const;
      bool setHeaderData (int section, Qt::Orientation orientation,
         const QVariant & value, int role = Qt::EditRole);
```

```
    bool insertRows (int row, int count, const QModelIndex & parent =
       QModelIndex());
    bool removeRows (int row, int count, const QModelIndex & parent =
       QModelIndex());
    bool insertColumns(int column, int count, const QModelIndex
       &parent = QModelIndex());
    bool removeColumns (int column, int count, const QModelIndex
       &parent = QModelIndex());
    QModelIndex index(int row, int column, const QModelIndex & parent
       = QModelIndex()) const;
    int columnIndex(const QString &column) const;
    QString columnName(int column) const;
    bool setColumnName(int column, const QString &name);
    bool addColumn(const QString &name);
    QSqlTableModel * relationModel (int column);
    int modelColumn(int column);
    WeakRelation * relation(int column);
    void setFilter(const QString & filter);
    QString filter() const;
    void updateRelations();
public slots:
    bool submitAll();
    bool submit();
    void revert();
    void revertAll();
protected:
    void sort ( int column, Qt::SortOrder order = Qt::AscendingOrder );
private:
    void init(QObject * parent = 0, QSqlDatabase db = QSqlDatabase());
    QSqlTableModel * mainModel;
    WeakRelationMap relMap;
    bool readOnly;
};
```

Класс `WeakRelationalTable` мог бы быть потомком класса `QSqlTableModel`. Вместо этого он наследует классу `QAbstractTableModel`, а класс `QSqlTableModel` в него агрегируется. Причина того, что вместо наследования выбрана агрегация, была объяснена ранее. Если бы `WeakRelationalTable` был потомком класса `QSqlTableModel`, его, в случае необходимости, было бы очень трудно "отвязать" от классов, связанных с SQL. В рамках же используемой модели этот класс можно переориентировать на другие источники данных с помощью сравнительно простого рефакторинга. Главный недостаток агрегации заключается в том, что нам приходится явно определять те методы, которые, в случае наследования, достались бы нам по наследству от класса-предка.

Кроме стандартных методов класса `QAbstractTableModel` класс `WeakRelationalTable` вводит ряд методов, которые лично я считаю весьма удобными. Например, метод

`setData()` в классе `WeakRelationalTable` существует не в одном, а в нескольких перегруженных вариантах. Перегруженные варианты предназначены для записи данных в таблицу непосредственно, без использования делегатов. Первый из дополнительных методов `setData()` позволяет указать номер строки и столбца, на пересечении которых расположена ячейка, куда следует записать данные (нумерация столбцов и строк таблицы начинается с нуля). Второй вариант позволяет указать номер строки и имя столбца, ведь в ряде случаев столбец гораздо удобнее запомнить по имени, чем по номеру. Добавлять данные можно и с помощью исходного метода `setData()`, который содержит в заголовке ссылку на объект `QModelIndex`. Нужно только учитывать, что объект `QModelIndex`, который передается методу `setData()`, должен содержать индекс самой модели данных. Этот вызов может выглядеть, например, так:

```
dataModel->setData(dataModel->index(row, column),
QVariant("Мое значение"));
```

Один из важнейших методов класса `WeakRelationalTable` — метод `addRelation()`. Этот метод добавляет ссылку на внешнюю таблицу. Первый аргумент метода — имя столбца главной таблицы, который будет содержать ссылки на записи внешней таблицы. Далее следуют: имя внешней таблицы, имя столбца-идентификатора во внешней таблице (значения этого столбца будут сопоставляться со значениями столбца ссылок главной таблицы) и имя столбца внешней таблицы, значениями которого нужно заменить значения-ссылки в столбце главной таблицы. Например, в результате выполнения кода

```
WeakRelationalTable * table = new WeakRelationalTable("Compositions");
table->addRelation("Genre", "Genres", "Id", "Name");
```

значения столбца `Genre` таблицы `Compositions` будут заменены значениями столбца `Name` из тех строк таблицы `Genres`, у которых значение поля `Id` совпадает со значением поля `Genre` таблицы `Compositions`.

Обратите внимание, что при добавлении отношений к таблице `table` все столбцы обозначаются именами. Если вы переименуете столбцы таблицы `table` до вызова `addRelation()`, вы должны указать новые имена столбцов. Если переименование столбцов происходит после вызова `addRelation()`, все отношения перенастраиваются автоматически. Рассмотрим теперь некоторые фрагменты реализации класса `WeakRelationalTable` (листинг 7.5).

Листинг 7.5. Фрагменты реализации класса `WeakRelationalTable`

```
void WeakRelationalTable::addRelation(const QString &column,
    const QString &table, const QString &idColumn,
    const QString &displayColumn)
{
    WeakRelation * relation = new WeakRelation(mainModel, table,
        idColumn, displayColumn);
    relMap.insert(column, relation);
}
```

```
QVariant WeakRelationalTable::data(const QModelIndex & index,
    int role /*= Qt::DisplayRole*/) const
{
    QString column = this->columnName(index.column());
    if (relMap.contains(column)) {
        return relMap.value(column)->getDisplayValue(
            mainModel->data(mainModel->index(index.row(),
            index.column()), role));
    }
    else
        return mainModel->data(mainModel->index(index.row(),
            index.column()), role);
}

bool WeakRelationalTable::setData(const QModelIndex & index,
    const QVariant & value, int role /*= Qt::EditRole*/)
{
    beginResetModel();
    bool res;
    QString column = this->columnName(index.column());
    if (relMap.contains(column)) {
        if (role != Qt::EditRole) {
                endResetModel();
                return false;
        }
    }
    res = mainModel->setData(mainModel->index(index.row(),
        index.column()), value, role);
    endResetModel();
    return res;
}
```

Как вы уже, наверное, поняли, главная таблица представлена объектом `mainModel`. Информация о внешних таблицах и их связях с главной таблицей хранится в объектах класса `WeakRelation` (этот класс будет рассмотрен далее). Все объекты `WeakRelation` хранятся в коллекции `relMap`, которая определена как переменная типа `QMap<QString, WeakRelation*>`. Метод `addRelation()` просто добавляет в эту коллекцию новый элемент. Метод `data()` использует данные о внешних таблицах на практике. Получая индекс, который содержит номер строки и столбца, метод `data()` проверяет, не связан ли данный столбец с внешней таблицей. Если данный столбец с внешней таблицей не связан, вызывается метод `data()` объекта `mainModel` с соответствующим ему индексом. Значение, возвращенное методом `mainModel->data()`, возвращается как результирующее значение метода `WeakRelationalTable::data()`. Если же столбец, для которого извлекаются данные, связан с внешней таблицей, то значение, которое должен вернуть метод `WeakRelationalTable::data()`, извлекается из внешней таблицы с помощью метода `getDisplayValue()` объекта `WeakRelation`. Таким образом, выполняется описанная выше подстановка значений в столбцы мо-

дели главной таблицы, связанные с внешней таблицей. В самой главной таблице при этом, естественно, ничего не меняется.

Функция `WeakRelationalTable::setData()`, которая вызывается, в основном, для редактирования данных, тоже, прежде всего, проверяет, не связан ли столбец из переданного ей индекса с внешней таблицей. Если не связан, функция просто вызывает метод `mainModel->setData()`, который изменяет содержимое ячейки главной таблицы и возвращает значение, которое возвращает ей этот метод. Если же переданный индекс содержит столбец, связанный с внешней таблицей, и параметр `role` не содержит значение `Qt::EditRole`, т. е. метод был вызван не для редактирования данных, функция возвращает `false`, что означает для вызвавшего ее делегата, что данные не были изменены. Это сделано для того, чтобы стандартный делегат, связанный, например, с объектом `QTableView`, не пытался редактировать значения в столбцах, ссылающихся на внешние таблицы (стандартный делегат ничего не знает про внешние таблицы и не сможет отредактировать данные правильно). Для редактирования данных `WeakRelationalTable` потребуется специальный делегат.

Есть еще одна проблема, которая не нашла стандартного решения в рамках модели `QSqlRelationalTableModel` при использовании класса `QsqlRelationalDelegate`. Что делать, если в процессе редактирования главной таблицы во внешние таблицы, на которые она ссылается, были добавлены новые значения, и эти значения требуются пользователю, редактирующему главную таблицу? Новые значения может добавить либо сам пользователь, перейдя в режим редактирования внешних таблиц, либо другие пользователи, если данные хранятся в многопользовательской СУБД. Проблема заключается в том, что структуры данных, из которых раскрывающийся список берет свои значения, не будут обновлены. В классе `WeakRelationalTable` эту проблему решает метод `updateRelations()`, который просто заставляет все объекты, хранящие данные внешних таблиц, перезагрузить свои данные.

Эта проблема существует также и для класса `QSqlRelationalTableModel`. Если вы работаете с классом `QSqlRelationalTableModel`, вы можете решить проблему обновления данных из внешних таблиц с помощью специально написанного кода (листинг 7.6).

Листинг 7.6. Функция `updateRelations()` для класса `WeakRelationalTable`

```
void updateRelations(QSqlRelationalTableModel * model)
{
      for(int i = 0; i < model->columnCount(); i++) {
            QSqlTableModel * m = model->relationModel(i);
            If (m)
                  m->select();
      }
}
```

В любом случае, поскольку мы не получим от хранилища данных оповещения о том, что содержимое внешних таблиц было изменено, нам придется самим решать, когда вызывать этот код.

На практике вам также пригодится возможность показывать в раскрывающемся списке внешних значений не все доступные значения, а только некоторую их часть, отфильтрованную по определенному признаку. Эту задачу решает метод `setRelationFilter()` (листинг 7.7).

Листинг 7.7. Создание фильтра для внешних отношений модели таблицы

```
void WeakRelationalTable::setRelationFilter(const QString &column,
    const QString &filter)
{
    relMap.value(column)->getModel()->setFilter(filter);
}
```

Класс *WeakRelation*

Этот вспомогательный класс описывает внешние таблицы и отношения, установленные между ними и главной таблицей. Как и при использовании класса `QSqlRelationalTableModel`, эти отношения могут соответствовать отношениям внешних ключей, существующим в базе данных, но могут быть заданы и произвольно. По сравнению с классом `WeakRelationalTable` у него не так уж много методов (листинг 7.8).

Листинг 7.8. Класс `WeakRelation`

```
class WeakRelation
{
public:
    WeakRelation(QSqlTableModel * parent, const QString &aTable,
        const QString &anIdColumn, const QString &aDisplayColumn);
    ~WeakRelation();
    QVariant getDisplayValue(const QVariant &idValue);
    QVariant getIdValue(const QVariant &displayValue);
    QSqlTableModel * getModel();
    QString getTable();
    QString getIdColumn();
    QString getDisplayColumn();
private:
    int findRowById(const QVariant &value);
    QString table;
    QString idColumn;
    QString displayColumn;
    QSqlTableModel * model;
};
```

Данные внешней таблицы хранятся в объекте `model` класса `QSqlTableModel`. Самые интересные методы этого класса — `getDisplayValue()` и `getIdValue()`. Первый ме-

тод получает в качестве аргумента значение, идентифицирующее запись во внешней таблице, и возвращает соответствующее ей значение из столбца `displayColumn`. Как мы уже знаем, этот метод используется методом `data()` класса `WeakRelationalTable` для отображения значений столбца, содержащего внешнюю ссылку. Это значит, что данный метод вызывается довольно часто, поэтому желательно сделать его быстродействующим. Для быстрого поиска строки таблицы, соответствующей `idValue`, используется метод половинного деления.

> ***Примечание***
>
> Метод половинного деления можно применять только в том случае, если последовательность, в которой выполняется поиск, отсортирована. Поэтому при создании объекта `model` его содержимое сортируется в восходящем порядке по значениям столбца `idColumn`.

Метод `getIdValue()` выполняет обратную операцию по отношению к методу `getDisplayValue()` — по заданному значению ячейки столбца `displayColumn` находит первую строку, содержащую это значение, и возвращает значение ячейки столбца `idColumn`, соответствующее этой строке. Метод `getIdValue()` необходим при редактировании таблицы, когда требуется преобразовать репрезентативное значение, выбранное пользователем, в идентификатор для ссылки на строку таблицы. Поскольку метод `getIdValue()` вызывается гораздо реже, чем `getDisplayValue()`, его не обязательно делать быстродействующим и для поиска подходящей строки таблицы в нем используется обычный перебор строк.

Класс *WeakRelationalDelegate*

Как вам уже должно быть понятно, класс `WeakRelationalDelegate` играет очень важную роль во взаимодействии между компонентом графического отображения и моделью данных. Мы уже рассмотрели, как именно делегат передает запросы объекта-представления объекту-модели, который хранит данные, с целью отображения данных. В процессе редактирования данных делегаты играют еще более сложную роль (рис. 7.4). Мы уже знаем о "волшебной" способности класса `QTableView` и других подобных классов редактировать данные непосредственно в таблице. Волшебство здесь заключается в том, что для редактирования каждой ячейки таблицы создается виджет-редактор, соответствующий типу данных, хранящемуся в ячейке (для строковых данных — компонент ввода текста, для целых чисел — наборный счетчик и т. д.). Более того, при работе с объектами классов `QSqlRelationalTableModel` и `QSqlRelationalDelegate`, при редактировании ячейки, содержащей ссылку на другую таблицу, появляется раскрывающийся список, из которого можно выбрать значение внешней таблицы.

Такое интеллектуальное поведение объектов `QTableView` становится возможным благодаря классам-делегатам. В процессе редактирования данных делегат получает запрос на создание виджета-редактора данных. Именно делегат определяет, какой виджет нужно создать для редактирования конкретного типа данных. Задача класса `QTableView` и его аналогов сводится к тому, чтобы встроить созданный делегатом виджет в нужном месте и управлять дальнейшим жизненным циклом этого виджета.

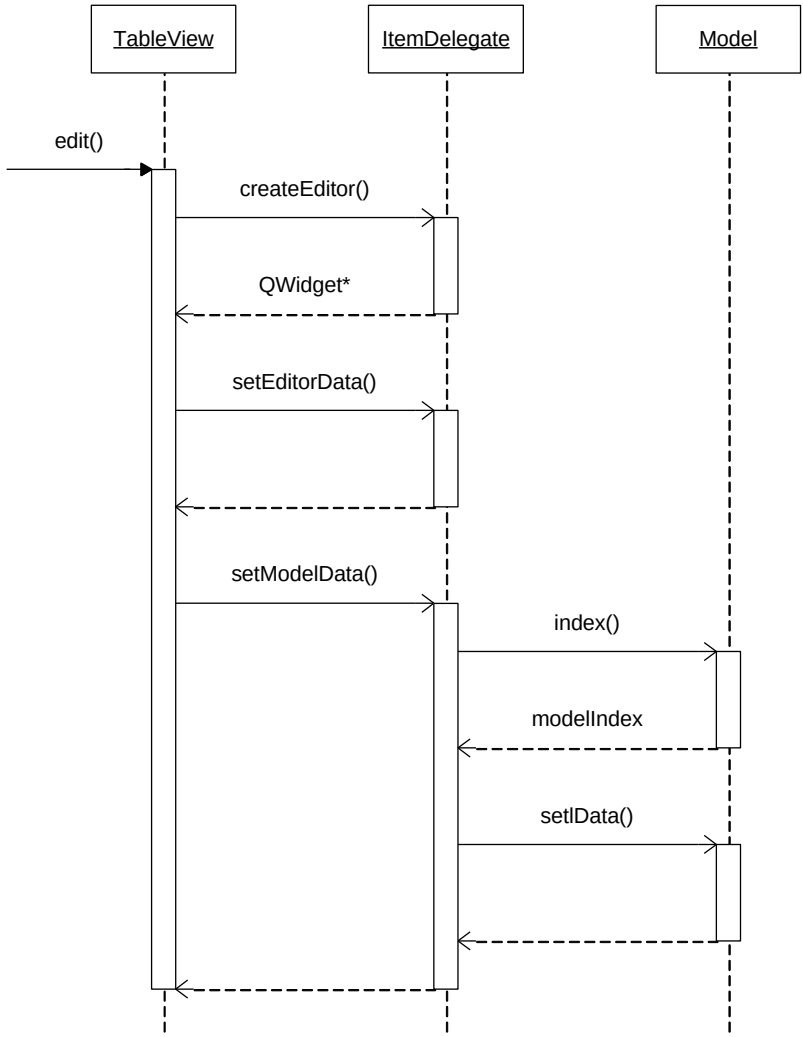


Рис. 7.4. Редактирование данных модели с использованием делегатов

Классы-делегаты Interview Framework ведут свой род от класса `QAbstractItemDelegate`. Как указывает имя этого класса, `QAbstractItemDelegate` является абстрактным классом, т. е. формирует базовый интерфейс для всех классов-делегатов, но сам делает не так уж много полезного. Это значит, что, как правило, вы не захотите объявлять собственные классы-делегаты непосредственно наследниками `QAbstractItemDelegate` (если, конечно, вы не хотите воспроизвести всю необходимую функциональность делегата самостоятельно). Гораздо лучше на роль базового класса для специализированных делегатов подходит класс `QStyledItemDelegate`, который является потомком класса `QAbstractItemDelegate` и реализует функциональность стандартного делегата.

В соответствии с документацией, класс-делегат, наследующий классу `QStyledItemDelegate`, должен перекрыть четыре метода последнего: `createEditor()`,

`setEditorData()`, `updateEditorGeometry()` и `setModelData()`. Первый из этих методов выполняет ту самую волшебную операцию, которая была описана ранее, — создает редактирующий виджет, тип которого зависит от типа редактируемых данных. Метод `setEditorData()` участвует во взаимодействии между делегатом и представлением данных, а именно, с помощью этого метода редактирующий виджет заполняется текущими данными, которые необходимо изменить. Метод `updateEditorGeometry()` позволяет изменить геометрию редактирующего виджета в зависимости от изменений геометрии объекта, представляющего данные. Метод `setModelData()` передает отредактированные данные объекту-модели.

Вообще говоря, перечисленный в документации список методов, которые необходимо перекрыть, не является ни полным, ни обязательным. Например, не обязательно перекрывать метод `updateEditorGeometry()`, т. к. базовый метод класса `QStyledItemDelegate` хорошо справляется с соответствующей задачей для большинства виджетов. Обратите внимание также на то, что в перечисленном в документации списке методов, которые следует перекрыть, присутствуют только методы, участвующие в процессе редактирования данных, в то время как делегаты используются и для отображения. Возможно, это объясняется тем, что методы базовых классов-делегатов успешно справляются с отображением данных большинства типов, так что вам, как правило, просто не нужно перекрывать эти методы. Если же вам все-таки требуется делегат с нестандартными функциями отображения данных, вам может понадобиться перекрыть метод `paint()` базового класса (например, `QStyledItemDelegate`). Реализация метода `paint()` может быть довольно сложной, если вы захотите реализовать полное стандартное поведение этого метода (что, в принципе, необязательно). Тем не менее даже минимальный метод `paint()` (листинг 7.9) дает вам большую свободу в управлении параметрами отображения.

Листинг 7.9. Минимальный метод `paint()`

```
void WeakRelationalDelegate::paint(QPainter *painter,
                                   const QStyleOptionViewItem &option,
                                   const QModelIndex &index) const {
        int margin = 0;
        QStyleOptionViewItemV4 opt = QStyleOptionViewItemV4(option);
        initStyleOption(&opt, index);
        const QWidget *widget = opt.widget;
        QStyle *style = widget ? widget->style() : QApplication::style();
        painter->save();
        style->drawControl(QStyle::CE_ItemViewItem, &opt, painter);
        painter->restore();
    }
```

Метод `paint()`, представленный в листинге, выполняет, по сути, те же действия, что и стандартный метод `paint()`, унаследованный классом `WeakRelationalDelegate` от его предка. Рассмотрим подробнее эти действия.

Класс QStyleOptionViewItemV4 предназначен для описания различных параметров отображения значения в ячейке таблицы. Помимо прочего, этот класс должен содержать строку текста, которая будет выведена. Метод initStyleOption() заполняет объект класса QStyleOptionViewItemV4 значениями, переданными методу paint() в параметре index (шрифт, стиль начертания, параметры кисти и т. д.). Помимо прочего, при вызове этого метода свойству text объекта opt присваивается значение текста, который требуется вывести в ячейке графической таблицы. Если вы хотите заменить или модифицировать этот текст, вы можете присвоить этому свойству другое значение, но делать это нужно после вызова initStyleOption().

Сам вывод данных выполняется методом drawControl() объекта style класса QStyle. В данном случае этот метод выполняет стандартный вывод текстового содержимого ячейки. Однако у класса QStyle есть и другие методы (drawItemText(), drawItemPixmap()), применение которых позволяет получить больший контроль над стилем отображения информации.

Для создания нестандартного вывода данных, приводимых к текстовому типу, можно перекрывать не метод paint(), а метод displayText(), который используется делегатом для получения строки данных для отображения. Задача этого метода — вернуть строку текста, которую он может преобразовать из переданного ему значения типа QVariant. Очевидно, что этот метод проще в реализации, чем метод panint().

Класс WeakRelationalDelegate добавляет к функциям QStyledItemDelegate тот самый раскрывающийся список, который позволяет редактировать значения, являющиеся ссылками на внешние таблицы. Соответственно, нам необходимо перекрыть методы createEditor(), setEditorData() и setModelData() (листинг 7.10).

Листинг 7.10. Класс WeakRelationalDelegate

```
class WeakRelationalDelegate : public QStyledItemDelegate
{
public:
    WeakRelationalDelegate(QObject * parent = 0);
    QWidget * createEditor ( QWidget * parent,
        const QStyleOptionViewItem & option,
        const QModelIndex & index ) const;
    void setEditorData ( QWidget * editor,
        const QModelIndex & index ) const;
    void setModelData ( QWidget * editor, QAbstractItemModel * model,
        const QModelIndex & index ) const;
};
```

Вы, конечно, хотите, первым делом узнать, как реализован метод createEditor(). Все не так сложно, как можно было бы ожидать, особенно если учесть, что часть работы может быть передана базовому классу (листинг 7.11).

Листинг 7.11. Метод `createEditor()`

```
QWidget * WeakRelationalDelegate::createEditor(QWidget *parent,
    const QStyleOptionViewItem &option,
    const QModelIndex &index) const
{
    WeakRelationalTable * model = (WeakRelationalTable *) index.model();
    if (!model->relationModel(index.column()))
        return QStyledItemDelegate::createEditor(parent, option, index);
    else {
      QComboBox * comboBox = new QComboBox(parent);
      comboBox->setModel(model->relationModel(index.column()));
      comboBox->setModelColumn(model->modelColumn(index.column()));
      return comboBox;
    }
}
class WeakRelationalDelegate : public QstyledItemDelegate
```

Метод `createEditor()` возвращает значение типа `QWidget *`. На самом деле это может быть указатель на любой объект класса-потомка `QWidget`. Наша первая задача, как и прежде, заключается в том, чтобы определить, указывает ли индекс, для которого мы создаем редактирующий виджет, на обычную ячейку или на ячейку, которая ссылается на внешнюю таблицу. Помимо методов, возвращающих номер строки и столбца, у класса `QModelIndex` есть метод `relationalModel()`, возвращающий указатель на объект-модель, которая создала этот индекс. Благодаря этому мы можем получить указатель на тот объект `WeakRelationalTable`, для которого создается редактирующий виджет. Затем мы вызываем метод `relationModel()` этого объекта, передавая ему в качестве аргумента номер столбца. Если в заданном столбце содержится ссылка на внешнюю таблицу, метод `relationalModel()` возвращает указатель на объект `QSqlTableModel`, соответствующий внешней таблице. В противном случае возвращается `NULL`.

Если метод `relationalModel()` вернул `NULL`, значит, мы имеем дело с обычной ячейкой и вызываем для нее метод `createEditor()` базового класса. В противном случае мы создаем виджет класса `QComboBox`. Как уже было сказано, нам не нужно заботиться о встраивании созданного виджета. Все, что мы должны сделать, — это правильно настроить его для работы с ячейками реляционной таблицы. Для настройки созданного объекта `QComboBox` мы вызываем два метода: `setModel()` и `setModelColumn()`. Первый метод устанавливает модель, из которой раскрывающийся список должен брать данные. Второй метод указывает столбец модели, который содержит данные для показа пользователю.

Примечание

Для того чтобы раскрывающийся список работал корректно, во внешних таблицах должны быть явным образом объявлены первичные ключи. Это важно для объекта `QComboBox`, который использует их для получения значений, возвращаемых методом `currentIndex()`.

Рассмотрим теперь перекрытые методы `setEditorData()` и `setModelData()` (листинг 7.12).

Листинг 7.12. Методы, перекрытые в классе `WeakRelationalDelegate`

```
void WeakRelationalDelegate::setEditorData(QWidget *editor,
    const QModelIndex &index) const
{
    WeakRelationalTable * wmodel = (WeakRelationalTable *) index.model();
    if (!wmodel->relationModel(index.column()))
        return QStyledItemDelegate::setEditorData(editor, index);
    if (editor->inherits("QComboBox")) {
        QComboBox * combo = (QComboBox *)editor;
        combo->setCurrentIndex(combo->findText(index.data().toString()));
    }
}

void WeakRelationalDelegate::setModelData(QWidget *editor,
    QAbstractItemModel *model, const QModelIndex &index) const
{
    WeakRelationalTable * wmodel = (WeakRelationalTable *) index.model();
    if (!wmodel->relationModel(index.column()))
        return QStyledItemDelegate::setModelData(editor, model, index);
    WeakRelation * r = wmodel->relation(index.column());
    if ((editor->inherits("QComboBox"))&& r)
        model->setData(index, r->getIdValue(
            QVariant( ((QComboBox*)editor)->currentText())));
}
```

Эти методы работают по той же схеме, что и метод `createEditor()`: сначала, с помощью метода `relationModel()` выясняется, является ячейка таблицы обычной или ссылочной. Если ячейка является обычной, вся работа передается одноименному методу базового класса.

Если методы вызваны для ячейки, содержащей ссылку, их дальнейшая работа связана с переданным им указателем на виджет-редактор. Все функции, работающие с редактором, передают указатель на него как указатель на базовый класс `QWidget`. Разумеется, фактический объект будет иметь другой тип, и прежде всего, мы должны выяснить, какой. В Qt существует множество способов определить тип объекта во время выполнения (на этом основана значительная часть внутренней механики Qt). Один из таких способов — вызов метода `inherits()`, объявленного в классе `QObject`. В качестве параметра этому методу передается строка с именем класса. Метод возвращает `true`, если объект, для которого он вызван, принадлежит к этому классу (или к классу-потомку этого класса). В противном случае метод возвращает `false`.

Примечание

Если вы хотите, чтобы механизмы определения типов во время выполнения, используемые в Qt, работали и для ваших классов, вы должны соблюсти два условия: во-первых, ваш класс должен быть потомком класса `QObject` или одного из его потомков. Во-вторых, в объявлении класса должен присутствовать макрос `Q_OBJECT`, который заставит систему сборки Qt создать метаобъект для вашего класса.

Вернемся к методу `setEditorData()`. Этот метод должен заполнить данными виджет-редактор, но в его заголовке отсутствует параметр, в котором бы передавались сами данные. Здесь нам опять поможет могущественный класс `QModelIndex`, у которого есть метод `data()`. В данном случае этот метод вернет нам объект класса `QVariant`, который содержит данные, необходимые для заполнения редактора.

В методе `setModelData()` мы должны сделать нечто противоположное тому, что мы делали в методе `data()` класса `WeakRelationalTable`, а именно — заменить значение ячейки, которое видит пользователь, значением индекса для ссылки на внешнюю таблицу и передать это значение модели. Для получения индекса используется описанный выше метод `getIdValue()` класса `WeakRelation`.

Теперь мы можем посмотреть, как работают наши классы в программе, позволяющей просматривать и редактировать таблицу (рис. 7.5).

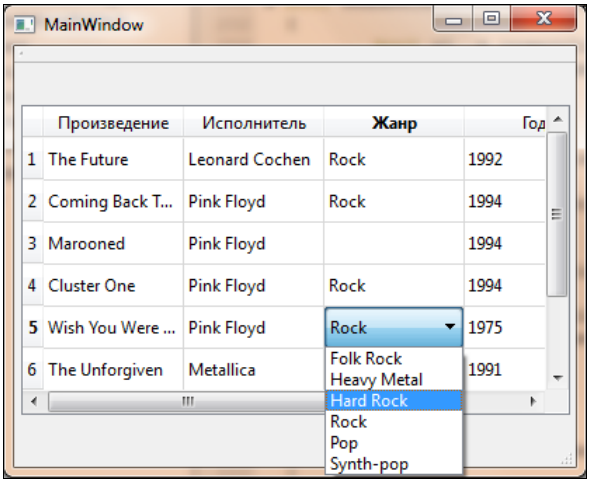


Рис. 7.5. Редактирование таблицы с неполными столбцами отношений

Творческое использование делегатов

Документация Qt подробно описывает, как создать нестандартные редактирующие виджеты для определенных типов данных. Практика показывает, что зачастую важнее другая задача — настройка параметров стандартных виджетов в зависимости от смысла (а не от типа) редактируемых данных. Пусть, например, один из столбцов редактируемой таблицы содержит телефонные номера в формате +7 (XXX) XXX-XXXX, и мы хотим, чтобы при редактировании этого столбца вид-

жет-редактор использовал маску ввода. Тут возникает одна проблема. С точки зрения Interview Framework эти данные имеют тип `QString`, такой же, как, например, данные об имени и фамилии обладателя телефонного номера. Если мы просто прикажем объекту-делегату создавать виджет с маской ввода для всех полей типа `QString`, то результат будет совсем не похож на то, чего мы хотели. В нашем случае нужны разные типы виджетов в зависимости от того, в каком столбце расположена редактируемая ячейка. Но и это не все. Нет смысла создавать класс-потомок виджета `QLineEdit` для того, чтобы реализовать маскированный ввод. Класс `QLineEdit` уже обладает всем необходимым для этого. Нужно только настроить его соответствующим образом. Это лишь один из примеров, когда нам может потребоваться настраивать параметры виджетов-редакторов в зависимости от того, для каких данных они вызываются.

Указанную проблему можно решить путем создания специализированных делегатов для каждой модели данных. Однако разработчики Qt позаботились о том, чтобы, написав код делегата один раз, мы могли использовать его в самых разных моделях. Система Interview Framework позволяет связывать делегаты не только с целыми моделями, но и с отдельными столбцами моделей. Например, для ввода номеров телефонов можно написать специальный делегат `PhoneNumberDelegate` (листинг 7.13).

Листинг 7.13. Класс `PhoneNumberDelegate`

```
class PhoneNumberDelegate : public QStyledItemDelegate
{
    Q_OBJECT
public:
    QWidget * createEditor ( QWidget * parent,
        const QStyleOptionViewItem & option,
        const QModelIndex & index ) const
    {
        QWidget * w = QStyledItemDelegate::createEditor(parent, option,
            index);
        QLineEdit * lineEdit = qobject_cast<QLineEdit *>(w);
        if (lineEdit)
            lineEdit->setInputMask("+7 (999) 999-9999");
        return w;
    }
};
```

Класс `PhoneNumberDelegate` наследует классу `QStyledItemDelegate` и перекрывает только один метод — `createEditor()`. В перекрытом методе сначала вызывается метод базового класса, который создает сам виджет. Затем указатель на `QWidget` приводится к типу "указатель на `QLineEdit`". Функтор `qobject_cast<QLineEdit *>` выполняет интеллектуальное приведение типов. Если объект, указатель на который передан ему в качестве аргумента, может быть приведен к типу `QLineEdit *`, функ-

тор возвращает указатель на `QLineEdit`. В противном случае возвращается 0. Нам остается только установить маску телефонных номеров с помощью метода `setInputMask()`. Теперь мы можем установить сразу два делегата — один для всей таблицы и второй специально для столбца телефонных номеров:

```
ui->tableView->setItemDelegate(new QStyledItemDelegate());
ui->tableView->setItemDelegateForColumn(phoneNumberColumn,
    new PhoneNumberDelegate());
```

В результате мы получаем возможность вводить данные в соответствии с правилами, заданными маской (рис. 7.6). Это позволяет пользователю избежать ошибок при вводе (по крайней мере, некоторых) и гарантирует единообразное представление данных (в нашем случае — телефонных номеров) в таблице.

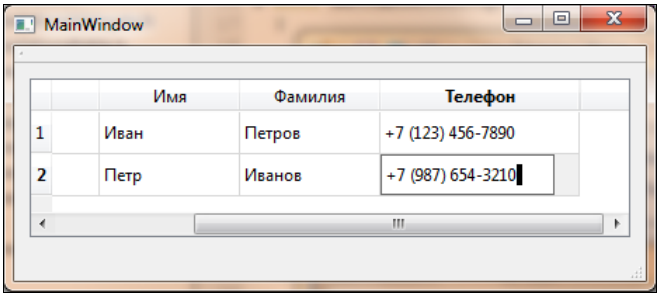


Рис. 7.6. Виджет-редактор с маской телефонных номеров

С помощью собственных делегатов можно не только изменять параметры редактора данных, но и параметры их отображения. Например, делегат `MoneyDelegate` (листинг 7.14), предназначенный для редактирования и отображения денежных сумм, устанавливает формат чисел, соответствующий принятому представлению сумм, и добавляет подпись, указывающую валюту.

Листинг 7.14. Класс `MoneyDelegate`

```
class MoneyDelegate : public QStyledItemDelegate
{
    Q_OBJECT
public:
    QWidget * createEditor ( QWidget * parent,
        const QStyleOptionViewItem & option,
        const QModelIndex & index ) const
    {
        QWidget * w = QStyledItemDelegate::createEditor(parent, option,
            index);
        QDoubleSpinBox * sb = qobject_cast<QDoubleSpinBox *>(w);
        if (sb) {
            sb->setDecimals(2);
            sb->setSuffix(QString::fromUtf8(" руб."));
        }
```

```
        return w;
    }
    QString displayText ( const QVariant & value,
        const QLocale & locale ) const
    {
        QString val = QStyledItemDelegate::displayText(value, locale);
        double d = val.toDouble();
        QString s = QString::number(d, 'f', 2);
        s = s + QString::fromUtf8("\tруб.");
        return s;
    }
};
```

Применение этого делегата к столбцу, содержащему значения денежных сумм, изменяет и редактирование, и отображение столбца (рис. 7.7). Вместо того чтобы перекрыть метод `paint()`, как было описано ранее, мы перекрываем метод `displayText()`, который возвращает текст для отображения в виде строки `QString`. Это более простой способ форматирования строки, нежели перекрытие метода `paint()`, а ничего другого нам от делегата и не требуется. Манипуляции со значением `double`, которые мы выполняем в методе `displayText()`, необходимы для того, чтобы превратить стандартное представление чисел с плавающей точкой, наподобие `7.5e+6` в формат, принятый для представления денежных значений — `7500000.00`.

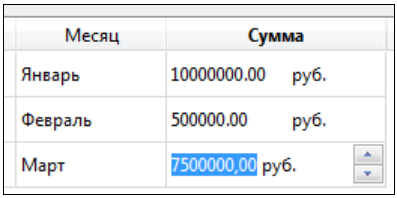


Рис. 7.7. Редактирование денежных сумм

Обратите внимание на то, что кроме параметра `value` метод `displayText()` получает параметр `locale` класса `QLocale`. Поскольку источник данных для объекта-модели может передавать текст в различных кодировках, этот текст должен быть преобразован в строку с учетом локали источника (что он и делает по умолчанию с помощью метода `locale.toSting()`). В некоторых случаях может потребоваться более сложное перекодирование строки или создание текстового эквивалента данных, которые, сами по себе, не являются текстовыми. Например, если ячейка модели содержит двоичные данные, в представлении данных их можно заменить текстовыми описаниями. Именно для этих целей разработчики Qt и создали метод `displayText()`, что, конечно, не мешает нам использовать его и в других целях.

Главное преимущество делегатов, создаваемых для отдельных столбцов, заключается в их универсальности. Например, если вы используете модели `QSqlTableModel`, `QSqlRelationalTableModel` или `WeakRelationalTable`, вы можете применять, соответственно, делегаты `QStyledItemDelegate`, `QSqlRelationalDelegate` и

WeakRelationalDelegate к моделям в целом, и делегаты типа PhoneNumberDelegate и MoneyDelegate к отдельным столбцам этих моделей.

Еще один полезный делегат (листинг 7.15) расширяет возможности раскрывающихся списков реляционных делегатов (таких как QSqlRelationalDelegate). Как вы, конечно, заметили, раскрывающийся список, который реляционные делегаты создают по умолчанию, позволяет только выбирать значения. Это удобно, когда значений в списке немного. Если же нам приходится выбирать из большого количества элементов, удобнее сделать так, чтобы первые буквы искомого элемента можно было набрать в строке раскрывающегося списка. Все, что нужно для этого сделать, — передать значение true методу setEditable() объекта QComboBox. Но для этого нам тоже понадобится делегат.

Мы, конечно, могли бы создать классы-потомки для делегатов QSqlRelationalDelegate и WeakRelationalDelegate, перекрыть в них методы createEditor(), но такой подход противоречил бы принципу повторного использования кода (а если мы захотим использовать эту функциональность с каким-то другим делегатом?). Мы пойдем по уже знакомому пути и создадим делегат для отдельного столбца. Но этот делегат будет отличаться от всех рассмотренных ранее. В предыдущих примерах, когда нам требовалось изменить стандартный виджет-редактор, мы создавали свой виджет сами, а затем вносили в него изменения. Виджет QComboBox отличается от других редактирующих виджетов тем, что перед его использованием его нужно настроить в соответствии со структурой модели данных, а о том, как это сделать, "знает" только делегат, предназначенный для работы именно с этой моделью. Тем не менее мы можем сделать наш делегат EditableComboBoxDelegate универсальным. Для этого достаточно вспомнить, что делегаты для отдельных столбцов назначаются совместно с делегатами, предназначенными для модели в целом. Если столбцу назначен специальный делегат, делегат для модели в целом для этого столбца уже не вызывается. Однако ничто не мешает нам вызвать этот "базовый" делегат из своего делегата. Первый параметр конструктора EditableComboBoxDelegate является указателем на общий делегат модели данных (подразумевается, что этот делегат умеет создавать виджеты-редакторы для всех столбцов модели, в том числе и виджеты типа QComboBox). Когда метод createEditor() делегата EditableComboBoxDelegate вызывается для некоторого столбца таблицы, он, в свою очередь, вызывает метод общего делегата для этого же столбца и получает виджет класса QComboBox, настроенный на работу с соответствующей моделью данных. Все, что остается сделать методу createEditor() класса EditableComboBoxDelegate, — вызывать метод setEditable() виджета QComboBox с аргументом true.

Листинг 7.15. Класс EditableComboBoxDelegate

```
class EditableComboBoxDelegate : public QStyledItemDelegate
{
public:
    EditableComboBoxDelegate(QAbstractItemDelegate * itemDelegate,
        QObject * parent = 0):QStyledItemDelegate(parent)
```

```
    {
        delegate = itemDelegate;
    }
    QWidget *createEditor(QWidget *aParent,
                          const QStyleOptionViewItem &option,
                          const QModelIndex &index) const
    {
        QWidget * w = delegate->createEditor(aParent, option, index);
        QComboBox * combo  = qobject_cast<QComboBox*>(w);
        if (combo) {
            combo->setEditable(true);
        }
        return w;
    }
private :
    QAbstractItemDelegate * delegate;
};
```

Назначение делегата `EditableComboBoxDelegate` объекту `QTableView` может выглядеть так:

```
ui->tableView->setItemDelegate(new QSqlRelationalDelegate());
ui->tableView->setItemDelegateForColumn(relationalRow,
    new EditableComboBoxDelegate(ui->tableView->itemDelegate()));
```

Еще один тип часто используемых делегатов, который мы не будем подробно рассматривать, — делегаты, предназначенные только для чтения. Их назначение — защитить данные от модификации пользователем, если данные для этого не предназначены. Если приложение работает с базой данных, мы можем (и должны) продумать такую защиту на уровне базы данных (что, однако, не исключает ее дублирования на уровне приложения). Но не все приложения, использующие Interview Framework, работают с базами данных и иногда данные нельзя (или, скажем так, очень сложно) защитить иначе, нежели на уровне делегатов. Я думаю, что вы теперь знаете о делегатах достаточно, чтобы самостоятельно написать делегат, предназначенный только для чтения. Добавлю к этому, что помимо метода `setItemDelegateForColumn()`, у компонентов, выполняющих отображение данных в виде таблиц, есть метод `setItemDelegateForRow()`, позволяющий назначать делегаты отдельным строкам таблицы. Это означает, что в сочетании с делегатами, предназначенными только для чтения, вы можете защищать от модификации не только таблицы в целом, но и отдельные столбцы или строки.

Создание стиля заголовков таблиц

До сих пор мы, в основном, обсуждали такие значения параметра `role`, как `Qt::DisplayRole` и `Qt::EditRole`. Использование других ролей поможет нам украсить таблицы дополнительными декоративными элементами. Ниже приводятся две функции, способные украсить наши таблицы (листинг 7.16).

Листинг 7.16. Функции для дополнительного оформления заголовков таблицы

```
bool setTableToolTip(QAbstractTableModel * model, int column,
    const QString &toolTip)
{
    return model->setHeaderData(column, Qt::Horizontal, toolTip,
        Qt::ToolTipRole);
}

bool setTableIcon(QAbstractTableModel * model, int column,
    const QString &iconName)
{
    QIcon icon;
    icon.addPixmap(QPixmap(iconName), QIcon::Normal, QIcon::On);
    return model->setHeaderData(column, Qt::Horizontal, icon,
        Qt::DecorationRole);
}
```

Функция `setTableToolTip()` добавляет всплывающую подсказку к заголовку столбца таблицы. Первый аргумент функции — указатель на объект-модель таблицы, второй аргумент — номер столбца, к которому добавляется подсказка, третий аргумент — текст подсказки. Функция вызывает метод `QAbstractItemModel::setHeaderData()`, который мы уже использовали для переименования столбцов, только на этот раз в аргументе `role` передается значение `Qt::ToolTipRole`, которое указывает, что передаваемые методом данные являются всплывающей подсказкой. Функция `setTableIcon()` работает сходным образом, только в аргументе `role` метода `setHeaderData()` передается значение `Qt::DecorationRole`, а в качестве данных — объект класса `QIcon`. В результате применения обеих функций мы получаем таблицу с пиктограммой и всплывающей подсказкой в заголовке (рис. 7.8).

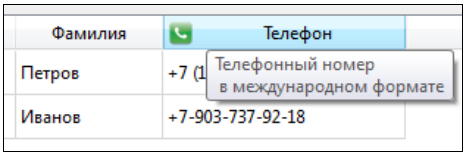


Рис. 7.8. Заголовок таблицы с пиктограммой и всплывающей подсказкой

Класс *QDataWidgetMapper*

Далеко не всегда мы хотим, чтобы данные модели были представлены нам для просмотра и редактирования в виде таблицы, списка или дерева. Иногда бывает полезно просматривать записи таблицы одну за другой. В таких случаях применяется не объект отображения модели данных, типа `QTableView`, а специальная форма. Самый простой пример, когда форма лучше, чем таблица или список, — приложение, регистрирующее пользователя. В таком приложении пользователь должен ввести

придуманные им логин и пароль, а также некоторую информацию о себе, например адрес электронной почты. Все эти данные сохраняются в таблице некоей базы данных, причем пользователю совершенно ни к чему видеть другие записи этой таблицы. В Qt существует возможность связывать стандартные виджеты с определенными элементами модели данных, так что виджеты смогут отображать и редактировать данные модели. Роль посредника между виджетами и моделью данных играет класс `QDataWidgetMapper`. Этот класс во многом похож на классы-потомки `QAbstractItemView`, хотя он и не является одним из них. Так же как объекты классов-потомков `QAbstractItemView`, объекты `QDataWidgetMapper` могут связываться с моделями данных и делегатами. Но, кроме того, они могут связываться с обычными виджетами для того, чтобы мы могли использовать эти виджеты при взаимодействии с моделью. Каждый виджет связывается с одной из секций модели (если модель имеет вид таблицы, то под секцией понимается столбец). В каждый отдельный момент времени набор виджетов, связанный с моделью через объект класса `QDataWidgetMapper`, может редактировать только один информационный элемент модели (т. е. только одну строку). Какой именно элемент доступен для редактирования, определяет объект `QDataWidgetMapper` с помощью метода `setCurrentItem()`.

> ***Примечание***
>
> Те, кто программировал с помощью средств разработки Borland, наверняка знакомы с концепцией компонентов, управляемых данными (data aware components). В библиотеке VCL таких компонентов было немного. В противоположность этому, в Qt практически любой виджет может взаимодействовать с моделью данных, используя объект `QDataWidgetMapper` в качестве посредника.

Связывание виджета с объектом `QDataWidgetMapper` выполняется методом `addMapping()` класса `QDataWidgetMapper`. Первым параметром метода должен быть указатель на виджет, который следует связать с моделью, вторым параметром — номер секции модели, к которой привязывается виджет (в нашем случае это столбец таблицы). Метод `addMapping()` существует в двух вариантах: с третьим параметром и без него. Третий параметр применяется для указания имени свойства виджета, которое следует использовать для обмена данными между виджетом и моделью. Если вы не указываете свойство явным образом, используется свойство, предназначенное для обмена данными по умолчанию. В классе виджета у такого свойства должен быть объявлен атрибут `USER` со значением, равным `true`, например:

```
Q_PROPERTY(int value READ value WRITE setValue NOTIFY valueChanged
           USER true)
```

В этой строке атрибуту `USER` свойства `value` некоего виджета присваивается значение `true`, и именно это свойство будет использоваться для отображения и редактирования элемента модели данных.

Программа widgetmapperdemo демонстрирует использование объекта `QDataWidgetMapper` при создании формы регистрации пользователя. Данные о "пользователе" сохраняются в объекте `model` класса `QStandardItemModel` (на время работы программы). Для наглядности модель `model` связана также с объектом `QTableView`,

с помощью которого вы можете видеть данные, занесенные в объект `model`. Это, кстати, еще одна полезная возможность объектов-моделей — они могут быть связаны с несколькими делегатами одновременно. Благодаря системе сигналов, которые модели посылают делегатам, все делегаты "узнают" об изменениях, внесенных в модель другим делегатом, и эти изменения сразу же отображаются в соответствующих виджетах.

Исходные тексты программы вы найдете в папке ch7/widgetmapperdemo на сайте **http://symmetrica.net/qt47book/**.

Изучение программы widgetmapperdemo мы начнем с класса `NewUserDialog`, который является потомком QDialog и реализует логику работы формы регистрации нового пользователя. Через переменную `ui` этот класс связан с формой `Ui::NewUserDialog` (рис. 7.9). Отметим, что для расположения виджетов в форме использовался объект `formLayout` класса `QFormLayout`. Этот класс специально создан для построения форм, содержащих несколько полей. Главное удобство его применения заключается в том, что подписи и поля выравниваются автоматически, а при изменении размеров окна меняются только размеры полей, размеры подписей остаются фиксированными. Сам класс `NewUserDialog` добавляет к своему предку не так уж много элементов (листинг 7.17).

Листинг 7.17. Класс `NewUserDialog`

```
class NewUserDialog : public QDialog
{
    Q_OBJECT
public:
    explicit NewUserDialog(QDataWidgetMapper * aMapper,
        QWidget *parent = 0);
    ~NewUserDialog();
public Q_SLOTS:
    int exec();
    void accept();
private:
    Ui::NewUserDialog *ui;
    QDataWidgetMapper * mapper;
};
```

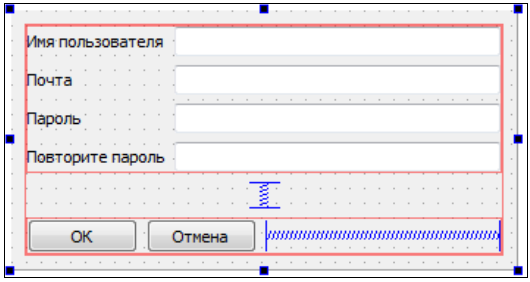


Рис. 7.9. Диалоговое окно в редакторе формы

Конструктор класса `NewUserDialog` отличается от конструктора `QDialog` (листинг 7.18).

Листинг 7.18. Конструктор класса `NewUserDialog`

```
NewUserDialog::NewUserDialog(QDataWidgetMapper * aMapper,
    QWidget *parent) :
    QDialog(parent),
    ui(new Ui::NewUserDialog)
{
    ui->setupUi(this);
    mapper = aMapper;
    mapper->addMapping(ui->userNameLineEdit, 0);
    mapper->addMapping(ui->passwordLineEdit, 1);
    mapper->addMapping(ui->mailLineEdit, 2);
    ui->passwordLineEdit->setEchoMode(QLineEdit::Password);
    ui->password2LineEdit->setEchoMode(QLineEdit::Password);
    connect(ui->pushButton, SIGNAL(clicked()), this, SLOT(accept()));
    connect(ui->pushButton_2, SIGNAL(clicked()), this, SLOT(reject()));
}
```

Главный класс программы передает в конструктор класса `NewUserDialog` указатель на объект класса `QDataWidgetMapper`. В конструкторе мы связываем виджеты диалогового окна с объектом `mapper`. Обратите внимание на то, что в форме диалогового окна присутствуют два виджета ввода пароля (пароль и подтверждение). Соответствующим объектам присвоены имена `passwordLineEdit` и `password2LineEdit`. С объектом `mapper` связывается, естественно, только один из них.

Виджеты, связанные с объектом `QDataWidgetMapper`, с помощью метода `addMapping()`, утрачивают эту связь либо в результате вызовов методов `removeMapping()` или `clearMapping()` (последний метод уничтожает все связи между виджетами и объектом `QDataWidgetMapper`), либо при уничтожении виджета. Таким образом, использование объекта `QDataWidgetMapper` — это еще и удобный способ передать данные из формы в тот объект, который создал эту форму.

Рассмотрим теперь процесс создания и настройки объекта. Эти операции выполняются в конструкторе главного объекта приложения (листинг 7.19).

Листинг 7.19. Конструктор главного объекта приложения

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    model = new QStandardItemModel(0, 3, this);
    model->setHeaderData(0, Qt::Horizontal,
        QString::fromUtf8("Имя пользователя"));
```

```
    model->setHeaderData(1, Qt::Horizontal, QString::fromUtf8("Пароль"));
    model->setHeaderData(2, Qt::Horizontal, QString::fromUtf8("Почта"));
    ui->tableView->setModel(model);
    ui->tableView->addAction(ui->actionAddUser);
    ui->tableView->setContextMenuPolicy(Qt::ActionsContextMenu);
    mapper = new QDataWidgetMapper(this);
    mapper->setModel(model);
    newUserDialog = new NewUserDialog(mapper, this);
}
```

После создания объекта `QDataWidgetMapper` мы назначаем ему модель с помощью метода `setModel()`, так же как и для объекта `tableView`. У класса `QDataWidgetMapper` есть свойство `submitPolicy`, которое может принимать одно из двух значений — `QDataWidgetMapper::AutoSubmit` или `QDataWidgetMapper::ManualSubmit`. В первом режиме (именно он установлен по умолчанию) `QDataWidgetMapper` посылает новые данные модели автоматически, как только закончится их редактирование (т. е. как только соответствующий виджет пошлет сигнал `editingFinished()`, `currentIndexChanged()` и т. д.). Обычно это происходит, когда пользователь нажимает клавишу <Enter> или когда соответствующий виджет теряет фокус ввода. В режиме `QDataWidgetMapper::ManualSubmit` передача данных модели происходит только при вызове метода-слота `submit()`.

Если данные, введенные пользователем в диалоговом окне, предназначены для передачи в настоящую базу данных, то лучше, вероятно, пользоваться режимом `ManualSubmit`. В этом случае класс диалогового окна сможет выполнить хотя бы первичную проверку корректности введенных пользователем данных прежде, чем данные попадут в базу данных. Сочетание режимов `ManualSubmit` объекта `QDataWidgetMapper` и объекта, представляющего таблицу базы данных (например, `QSqlTableModel`), дает вам два уровня, на которых вы можете проверить корректность данных, введенных пользователем, перед записью их в базу данных.

У класса `QDataWidgetMapper` есть также метод `setItemDelegate()`, который позволяет назначить объекту делегат, отличный от стандартного. К сожалению, нельзя назначать делегаты для отдельных элементов модели, как мы делали это для объектов `tableView`.

В нашей программе мы выполняем два типа проверок. Первая проверка чисто формальная — в классе диалогового окна мы проверяем, совпадают ли пароли, введенные пользователем, и не остались ли какие-нибудь поля незаполненными. Для этого мы перекрываем метод-слот `accept()` класса `QDialog` (листинг 7.20).

Листинг 7.20. Перекрытый метод `accept()`

```
void NewUserDialog::accept()
{
    if ((!ui->userNameLineEdit->text().isEmpty()) &&
        (!ui->mailLineEdit->text().isEmpty())) {
```

```
        if ((ui->password2LineEdit->text() ==
            ui->passwordLineEdit->text()) &&
            (!ui->passwordLineEdit->text().isEmpty()))
            return QDialog::accept();
        else
            QMessageBox::critical(this,
            QString::fromUtf8("Ошибка при вводе пароля"),
            QString::fromUtf8("Введенные пароли не совпадают!"));
    } else
        QMessageBox::critical(this,
        QString::fromUtf8("Ошибка при вводе"),
        QString::fromUtf8("Все поля должны быть заполнены!"));
}
```

Перекрытие метода `accept()` удобно тем, что оно позволяет не закрывать диалоговое окно в случае, если введенные данные некорректны. Конечно, имеется в виду попытка закрыть окно с помощью кнопки **OK**. Если пользователь отменит ввод, проверка корректности данных не нужна. Диалоговое окно может быть закрыто кнопкой **OK** только в том случае, если мы вызовем метод `accept()` базового класса и передадим возвращаемое им значение (`QDialog::Accepted`) оператору `return`. Если этого не делать, окно останется открытым.

Далее нам требуется выполнить более сложную проверку: имена пользователей и адреса электронной почты не должны повторяться. Если бы мы работали с базой данных, для модели данных можно было бы выбрать режим `EditStrategy OnManualSubmit` и отправить специальный запрос к базе данных прежде, чем записывать в базу данные о новом пользователе. В нашем примере (листинг 7.21) мы используем метод `findItems()` для того чтобы определить, содержит ли определенный столбец таблицы заданный текст. Если введенные имя пользователя или адрес почты в таблице уже присутствуют, мы снова вызываем диалоговое окно `newUserDialog`, и так до тех пор, пока пользователь не сделает одно из двух: либо не введет корректные данные, либо не откажется от ввода. Такое разделение проверок на два этапа представляется логичным. Класс `NewUserDialog` отделен от модели данных и не вмешивается в ее работу, поэтому он выполняет только "механические" проверки (можно было бы добавить еще проверку наличия пробелов в имени пользователя и почтовом адресе). Проверка второго уровня, связанная с содержанием данных, выполняется классом `MainWindow`.

Листинг 7.21. Проверка введенных данных

```
void MainWindow::addUser()
{
    model->insertRow(0);
    mapper->setCurrentIndex(0);
    if (!newUserDialog->exec())
        model->removeRow(0);
```

```
    while (model->findItems(model->data(model->index(0, 0)).toString(),
        Qt::MatchFixedString, 0).count() > 1) {
        QMessageBox::critical(this,
        QString::fromUtf8("Ошибка ввода"),
        QString::fromUtf8("Пользователь с таким именем уже существует"));
        if (!newUserDialog->exec())
            model->removeRow(0);
    }
    while (model->findItems(model->data(model->index(0, 2)).toString(),
        Qt::MatchFixedString, 2).count() > 1) {
        QMessageBox::critical(this, QString::fromUtf8("Ошибка ввода"),
        QString::fromUtf8("Пользователь с такой почтой уже существует"));
        if (!newUserDialog->exec())
            model->removeRow(0);
    }
}
```

Метод `addUser()` представляет собой слот. Этот слот мы делаем видимым в окне редактора форм с помощью команды **Изменить сигналы/слоты...** (рис. 7.10). Далее в редакторе действий мы добавляем действие `actionAddUser` и, в редакторе сигналов и слотов, связываем слот `addUser` сигналом `triggered()` этого действия.

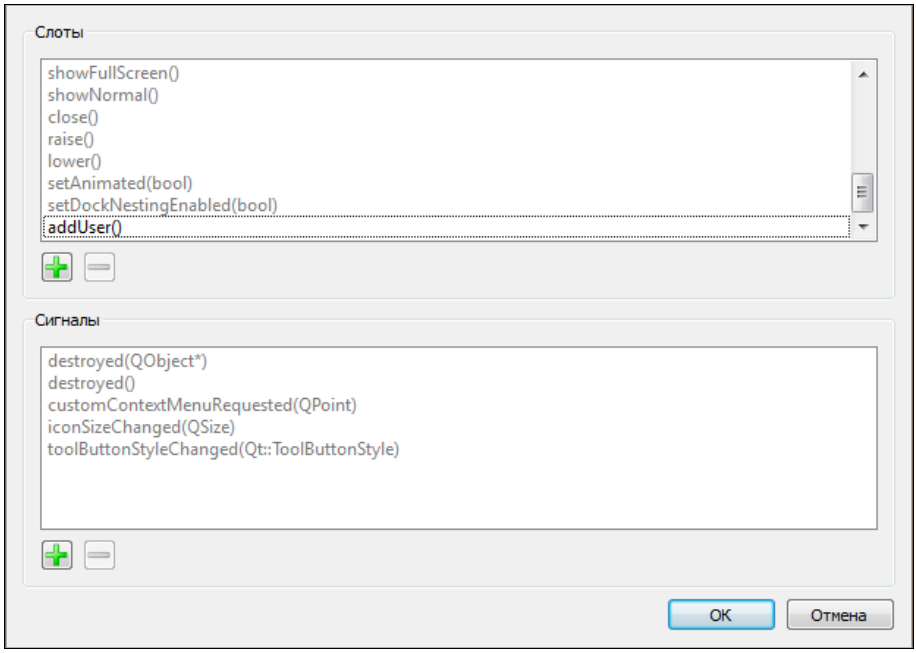


Рис. 7.10. Добавление слота

Классы *QDataWidgetMapper* и *QComboBox*

Мы уже видели, что при использовании модели `QSqlRelationalTableModel` и делегата `QSqlRelationalDelegate` (или их аналогов), в процессе редактирования столбца, ссылающегося на другую таблицу, в объекте `QTableView` раскрывается диалоговое окно с выбором подходящих значений. Можно было бы ожидать, что для объекта `QDataWidgetMapper` все будет так же просто: назначаем в качестве делегата объект `QSqlRelationalDelegate` (это вполне допустимо), с помощью метода `addMapping()` связываем объект `QComboBox` с соответствующим столбцом таблицы, и все работает. На самом деле, связанный таким образом объект `QComboBox` не будет работать так, как мы ожидаем. Скорее всего, он вообще не будет работать.

Тем не менее мы можем связать объект `QComboBox` с реляционной моделью с помощью объекта `QDataWidgetMapper` таким образом, чтобы он работал совсем как редактирующий виджет в `QTableView`. Для этого объекту `QComboBox` следует явно указать модель, из которой он будет брать данные для отображения. В дистрибутиве Qt вы найдете пример combowidgetmapper, где показано, как настроить объект `QComboBox` для работы с объектами `QDataWidgetMapper` и `QStandardItemModel`.

Для объекта `QSqlRelationalTableModel` эта настройка может выглядеть иначе (листинг 7.22).

Листинг 7.22. Настройка объекта `QComboBox` для работы с реляционной моделью

```
QSqlRelationalTableModel * model = new QSqlRelationalTableModel();
model->setTable("Compositions");
model->setRelation(2, QSqlRelation("Genres", "id", "name"));
model->select();
mapper = new QDataWidgetMapper(this);
mapper->setModel(model);
QComboBox * genresComboBox = new QComboBox(this);
genresComboBox->setModel(model()->relationModel(2));
mapper->addMapping(ui->comboBox, 2, "currentIndex");
```

Сначала мы создаем объект `QSqlRelationalTableModel`, указываем таблицу, которую он должен представлять, и отношение "внешний ключ" одного из столбцов этой таблицы к другой таблице. Напоминаю, что вызов метода `setRelation()` в данном случае является обязательным, т. к. сам объект `QSqlRelationalTableModel` не может получить информацию о реляционных связях таблицы. Не забудьте также вызвать метод `select()`, который, собственно, заполняет модель данными.

Теперь переходим к настройке объекта `mapper`. С помощью метода `setModel()` мы указываем ему модель, с которой он должен работать. Но, для того чтобы связать объект `mapper` с объектом класса `QComboBox`, этому объекту необходимо указать еще

одну модель — модель внешней таблицы. Если мы имеем дело с объектом класса `QSqlRelationalTableModel`, то мы можем получить указатель на внешнюю модель (объект класса `QSqlTableModel`) с помощью метода `relationModel()`, принимающего в качестве параметра номер столбца таблицы `model`, для которого установлено отношение. Получив эту модель, мы можем передать ее методу `setModel()` объекта `QComboBox`. Все это мы и делаем в строке:

```
genresComboBox->setModel(model()->relationModel(2));
```

Теперь объект `genresComboBox` будет работать так же, как работал бы раскрывающийся список в окне объекта `QTableView`.

ГЛАВА 8

Библиотека Qt и ваша видеокарта

В этой главе речь пойдет о том, как, используя дополнительные возможности видеокарты, решить различные графические и неграфические задачи. Примеры из этой главы будут работать у вас только в том случае, если ваша видеокарта и драйвер поддерживают аппаратное ускорение OpenGL. На это предупреждение должны обратить особое внимание пользователи Linux и компьютеров со встроенными графическими чипсетами.

Графическая система Arthur

Хотя изменения графической системы Qt 4 по сравнению с Qt 3 не очень заметны на первый взгляд, новая система предоставляет весьма широкие возможности. Старые классы `QPainter` и `QPaintDevice` остались, но к ним добавился абстрактный класс `QPaintEngine`. Теперь все функции, реализующие специфику графического вывода на разных устройствах, собраны в классах-потомках `QPaintEngine`, соответствующих этим устройствам. Классы `QPainter` и `QPaintDevice` используют методы `QPaintEngine` для доступа к графическим устройствам, а не обращаются к этим устройствам напрямую, как раньше. Программисту же, наоборот, не придется иметь дела с потомками `QPaintEngine`, если только он не захочет расширить функциональность Qt 4, реализовав графический вывод на каком-нибудь неподдерживаемом устройстве. На практике это означает, что "сладкая парочка" `QPainter` и `QPaintDevice` теперь может рисовать практически на всех графических устройствах, доступных на данной платформе, причем работа с разными устройствами, будь то принтер или окно OpenGL, в значительной степени унифицирована. Еще одно преимущество новой системы заключается в том, что многие графические операции, которые раньше были реализованы чисто программными средствами, теперь могут использовать аппаратное ускорение и другие функции, поддерживаемые "железом" (раньше это было невозможно потому, что между `QPainter` и устройством лежала "прослойка" эмулятора растрового массива). Например, класс `QGLWidget`, работающий с OpenGL, теперь поддерживает операции класса `QPainter`, используя, где возможно, аппаратное ускорение.

Новая архитектура позволила добавить новые возможности, среди которых стоит отметить градиентные кисти (заполняющие фигуры заданным градиентом) и поддержку альфа-канала, которая позволяет создавать полупрозрачные изображения.

Обсудим, как все это работает на практике. В качестве примера рассмотрим визуальный компонент `QGLWidget`. Благодаря новой архитектуре мы можем создавать изображения средствами `QPainter` в рабочем окне `QGLWidget` точно так же, как и на поверхности любого другого виджета. Не могу не отметить некоторую диалектичность процесса: когда-то я демонстрировал, как выводить графику OpenGL на поверхности объекта-потомка `QWidget`. Теперь мы воспользуемся `QGLWidget` для вывода "обычных" изображений, которые не являются частью трехмерных сцен. Такое использование OpenGL "не по назначению" отражает популярную в последнее время тенденцию — задействовать мощь 3D-ускорителей в традиционно двухмерных задачах, например при выводе окон на экран или работе с растровыми изображениями.

Самый простой способ добавить в программу сцену, которая рисуется средствами OpenGL, — создать класс-потомок `QGLWidget`. В простейшем случае у дочернего класса достаточно перекрыть метод `paintEvent()`, который, как мы знаем, перерисовывает виджет всякий раз, когда виджет в этом нуждается.

В Qt 4, вплоть до версии 4.7, центральным виджетом программ, использующих OpenGL, обычно является виджет `QGLWidget`. В Qt 4.8 ситуация может измениться, о чем мы еще поговорим в конце этой главы. Обычно мы создаем класс-потомок этого класса, в котором перекрываем некоторые методы класса `QGLWidget`. Рассмотрим метод `paintEvent()` класса `GLWidget` (который является потомком `QGLWidget`) в программе arthur-demo (вы найдете исходные тексты в папке ch8/arthur-demo на сайте **http://symmetrica.net/qt47book/**).

Листинг 8.1. Метод `paintEvent()` класса `GLWidget`

```
void GLWidget::paintEvent(QPaintEvent *event)
{
        QPainter painter;
        QPen pen;
        painter.begin(this);
        painter.setRenderHint(QPainter::HighQualityAntialiasing);
        painter.eraseRect(QRect(0, 0, width(), height()));
        painter.drawPixmap(0, 0, width(), height(), *pixmap);
        painter.translate(width()/2, height()/2);
        painter.rotate(angle);
        painter.translate(-width()/2, -height()/2);
        pen.setColor(QColor(0, 127, 0));
        pen.setWidth(4);
        painter.setPen(pen);
        painter.drawLine(0, 0, width(), height());
        pen.setColor(QColor(255, 0, 0));
        painter.setPen(pen);
        painter.drawLine(0, height(), width(), 0);
        painter.setBrush(QColor(255, 0, 0, 127));
        painter.drawRect(0, 0, width()/2, height());
```

```
        painter.setBrush(QColor(0, 0, 255, 127));
        painter.drawRect(0, 0, width(), height()/2);
        painter.setBrush(QColor(0, 255, 0, 127));
        painter.drawRect(width()/2, 0, width()/2, height());
        painter.end();
        event->accept();
}
```

Прежде всего, обратите внимание на то, что хотя виджет `GLWidget` наследует `QGLWidget`, мы пользуемся исключительно методами класса `QPainter`. Рисование начинается с вызова метода `begin()` объекта `painter`. Аргументом этого метода должен быть указатель на объект `QPaintDevice`, к типу которого теперь приводится и объект класса `QGLWidget` (и, естественно, его потомки). Останавливаться на каждой инструкции вывода графики мы не будем. Обращаю внимание читателя только на некоторые "продвинутые" особенности `painter`.

При рисовании сцены мы включаем поддержку сглаживания контуров с помощью вызова метода `setRenderHint()`. Этому методу передается константа `QPainter::HighQualityAntialiasing`, которая определяет самое качественное сглаживание. С помощью метода `setRenderHint()` можно настроить и другие тонкие параметры вывода графики, например сглаживание текста или сглаживание при преобразовании растровых изображений.

Помимо настройки сглаживания мы используем смешивание цветов (alpha blending) — обратите внимание на четвертый аргумент конструктора класса `QColor()`. Наконец, чтобы сцена была анимированной, мы используем методы трансформации `translate()` и `rotate()`. Сглаживание, смешивание и геометрические преобразования могут выполняться с использованием аппаратной поддержки (например, поддержки 3D-ускорителя), если она включена в вашей системе.

В приведенном примере специально не использовались функции OpenGL, чтобы показать, что в графической системе Arthur можно задействовать возможности OpenGL, не используя сами команды OpenGL (например, воспользоваться методом `eraseRect()` вместо `glClearColor()`). В результате один и тот же код может быть задействован для вывода графики в окне с помощью 3D-ускорителя, для записи графики в растровые форматы хранения изображений или для вывода изображений на принтер PostScript.

> ***Примечание***
>
> Впрочем, не все так просто. Если вы захотите повторно использовать тот же самый код `GLWidget::paintEvent()`, например, для вывода на печать (а не просто скопировать его в другой класс), то ни метод `QPainter::setRedirected()`, ни более продвинутый метод `QWidget::render()` вам не помогут, т. к. ни тот ни другой не умеют работать с графикой OpenGL. В лучшем случае вы получите пресловутый черный квадрат.

Если вы хотите вывести на принтер именно картинку, отображаемую средствами OpenGL, придется пойти в обход. Сначала, с помощью метода `grabFrameBuffer()`, объявленного в классе `QGLWidget`, получаем объект `QImage`, который содержит сни-

мок графической сцены OpenGL. Теперь этот объект можно выводить на печать, например в псевдопринтер, "печатающий" в файлы PDF (листинг 8.2).

Листинг 8.2. Сохранение вывода виджета-потомка `QGLWidget` в формате PDF

```
QImage img = glWidget->grabFrameBuffer(true);
QPrinter printer;
printer.setOutputFormat(QPrinter::PdfFormat);
printer.setOutputFileName("D:\\pdfs\\page1.pdf");
QPainter painter;
painter.begin(&printer);
painter.drawImage(0, 0, img);
painter.end();
```

Вернемся теперь немного назад и посмотрим на конструктор класса `GLWidget` (листинг 8.3).

Листинг 8.3. Конструктор класса `GLWidget`

```
GLWidget::GLWidget(QWidget *parent) : QGLWidget(parent)
{
    angle = 0;
    pixmap = new QPixmap();
    pixmap->load("mashrooms.png");
    if (pixmap->isNull())
        qWarning() <<
            QString::fromUtf8("Файл 'mashrooms.png' не загружен.");
}
```

Здесь мы загружаем графический файл в объект `QPixmap`. Если загрузить файл не удалось (метод `isNull()` возвращает `true`), мы выводим предупреждающее сообщение на отладочную консоль. Функция `qWarning()` — одна из семейства функций, предназначенных для того, чтобы сообщить вам, что в программе пошло что-то не так. Для того чтобы использовать синтаксис вызова `qWarning()` с оператором `<<`, необходимо включить заголовочный файл `<QDebug>` в исходный текст. Qt Creator на всех платформах перехватывает все сообщения, которые графическая программа выводит на консоль. Запуская программу из-под Eclipse (при использовании инструментария GNU), вы также можете увидеть эти сообщения. Наконец, все сообщения, выводимые на консоль программой и ее компонентами, можно прочитать, если программа запущена из консоли Linux. А вот при отладке программы в среде Microsoft Visual Studio эти сообщения проще всего увидеть, если сначала запустить специальную консоль Visual Studio Command Prompt.

Для анимации изображения мы используем встроенный таймер класса `QMainWindow` (листинг 8.4).

Листинг 8.4. Анимация сцены OpenGL с помощью таймера

```
void OGLWindow::timerEvent(QTimerEvent *event)
{
    glWidget->setAngle(glWidget->getAngle()+0.1);
    glWidget->update();
}
```

Методы `setAngle()` и `getAngle()` позволяют получить доступ к переменной `angle`, которую мы используем в методе `GLWidget::paintEvent()`. В результате получаем анимированное изображение с аппаратно ускоренными анимацией, сглаживанием и альфа-каналом (рис. 8.1).

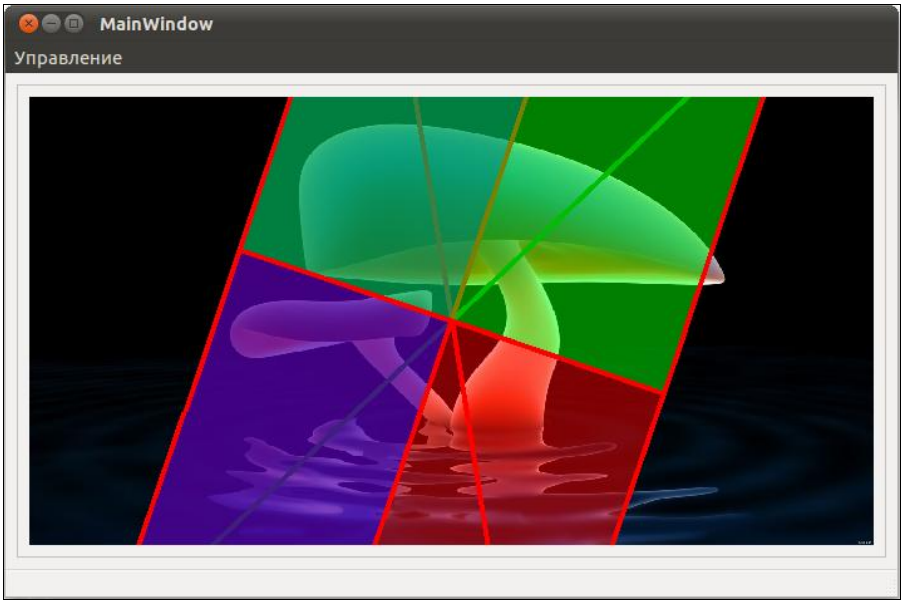


Рис. 8.1. Программа, демонстрирующая возможности Arthur

Еще один пример, где аппаратное ускорение OpenGL может оказаться полезным, — работа с очень большими изображениями. Имеются в виду изображения, разрешение которых многократно превышает разрешение монитора. Необходимость работы с такими изображениями возникает в геоинформационных приложениях, приложениях обработки сканированных изображений и цифровой фотографии. Программа scalefast, которую вы найдете в папке Ch8/scalefast, демонстрирует возможности аппаратно ускоренного масштабирования и прокручивания больших изображений. Для изменения масштаба изображения пользуйтесь клавишами "плюс" или "минус", а для прокрутки — клавишами со стрелками (рис. 8.2).

Для вывода изображения мы используем метод `drawPixmap()` класса `QPainter`, в той его версии, которая позволяет скопировать прямоугольную область объекта `QPixmap` в буфер устройства ввода/вывода.

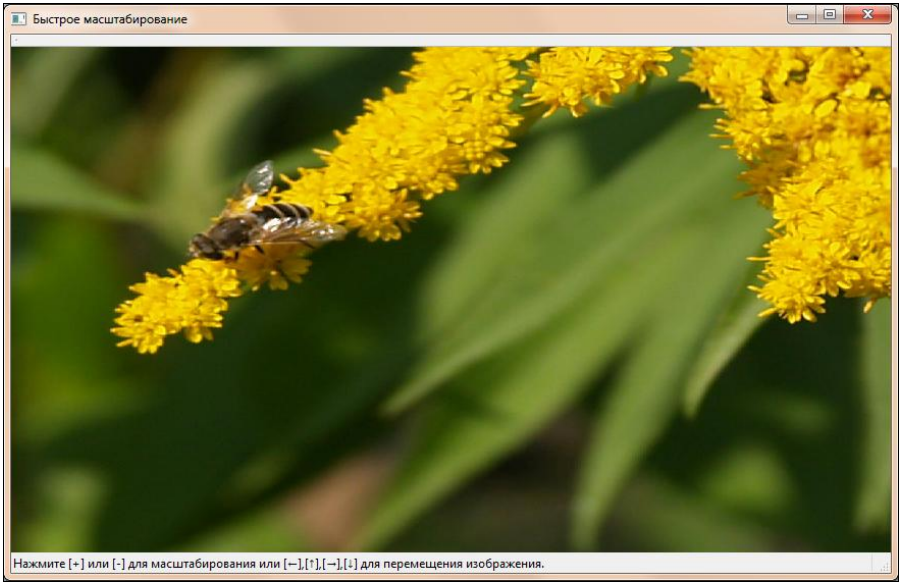


Рис. 8.2. Программа scalefast

Аппаратная поддержка масштабирования и прокрутки изображения действительно может ускорить эти операции, особенно в операционной системе Linux, однако у нее есть ограничение, связанное с максимально допустимым размером изображения, которое можно загрузить в буфер OpenGL (имеется в виду размер исходного изображения, возможности масштабирования практически не ограничены). В моей системе максимальный размер изображения составляет 4098×4098 пикселов, на других графических картах он, вероятно, может быть другим. И хотя указанный размер превышает разрешающую способность большинства мониторов и видеокарт, нам, конечно, хочется работать с изображениями еще большего размера. В следующей главе мы рассмотрим один из способов обхода указанного ограничения.

Класс *QGLContext*

Контекстом OpenGL называется набор переменных, поддерживающих внутреннее состояние OpenGL. Все команды OpenGL выполняются с учетом определенного контекста, но сам контекст при этом не указывается. Если в приложении используется несколько контекстов OpenGL, один из них должен быть выбран в качестве текущего контекста. Объекты класса `QGLContext` помогают настроить множество параметров OpenGL, о которых вы можете прочитать в документации. Параметры контекста обычно задаются с помощью объекта класса `QGLFormat`. Важно знать, что у каждого объекта `QGLWidget` есть свой контекст OpenGL. Чтобы сделать этот контекст текущим, необходимо вызвать метод `makeCurrent()`.

Прежде чем перейти к обсуждению конкретного примера, коснемся одного специфического вопроса, связанного с программированием OpenGL в Qt. Работая с пото-

ками, мы никогда не обновляли содержимое графических объектов из вторичного потока. Причины этого очевидны, и главная из них — необходимость четко разделять доступ к критическому ресурсу, которым является графический объект. Так что до сих пор мы либо располагали весь код, обновляющий графический объект, в главном потоке, либо передавали главному потоку сообщения, в ответ на которые он должен был обновлять графические данные. Вторая модель, очевидно, не подходит при работе с графикой OpenGL, по крайней мере, в большинстве случаев. Сцена OpenGL должна обновляться слишком часто, чтобы это обновление можно было поручить системе сообщений.

Начинающие программисты Qt, пишущие для Windows, почему-то часто используют для команд OpenGL отдельные потоки, которые выводят данные непосредственно в виджет OpenGL, принадлежащий главному потоку. Я бы не советовал делать это без крайней необходимости. Если объект графической информации для вывода не очень велик, лучше расположить весь код обновления сцены в главном потоке. Помимо обычных причин, по которым мы стремимся не обновлять виджеты напрямую из вторичных потоков, в случае с OpenGL есть еще одна.

Как уже отмечалось, сами по себе команды OpenGL не привязываются ни к какому средству вывода (окну, виджету и т. д.). Команды OpenGL абстрагированы от конкретных компонентов вывода, управлением которыми занимается операционная система. Разные операционные системы делают это по-разному. В некоторых системах, например в Linux, результаты работы OpenGL привязываются к потоку, в котором выполняются команды OpenGL (строго говоря, в этом случае к потоку привязывается контекст OpenGL). В Windows, наоборот, можно выполнять команды OpenGL в одном потоке, а графические результаты отображать в другом. Если вы хотите, чтобы ваша программа работала предсказуемо во всех поддерживаемых Qt операционных системах, следует соблюдать строгие правила относительно обновления виджетов. Ну и наконец, если в процессе рисования сцены OpenGL не выполняются интенсивные расчеты (а если такие расчеты необходимы, они часто могут быть возложены на графический процессор), то использование потоков будет мало способствовать ускорению работы даже на многоядерных процессорах. Поскольку рисование сцены OpenGL сводится, в основном, к отправке драйверу видеокарты относительно коротких сообщений, эта операция не очень сильно нагружает поток.

Приложение doublecontexts (рис. 8.3) демонстрирует вывод двух независимых сцен OpenGL в два виджета Qt. Вы найдете эту программу в папке Ch8/doublecontexts. Поскольку в этом приложении мы имеем дело не с двухмерной графикой, ускоренной с помощью 3D-ускорителя, а с "настоящей" трехмерной графикой OpenGL, в виджете, в классе `GLWidget`, необходимо выполнить весь стандартный цикл обработки графики OpenGL: инициализацию сцены OpenGL, рисование, изменение размеров сцены. Для этих трех составляющих в классе `QGLWidget` объявлены специальные виртуальные функции, которые должны быть перекрыты в его потомках. Рассмотрим подробно один из двух виджетов, выполняющих вывод графики OpenGL (листинг 8.5).

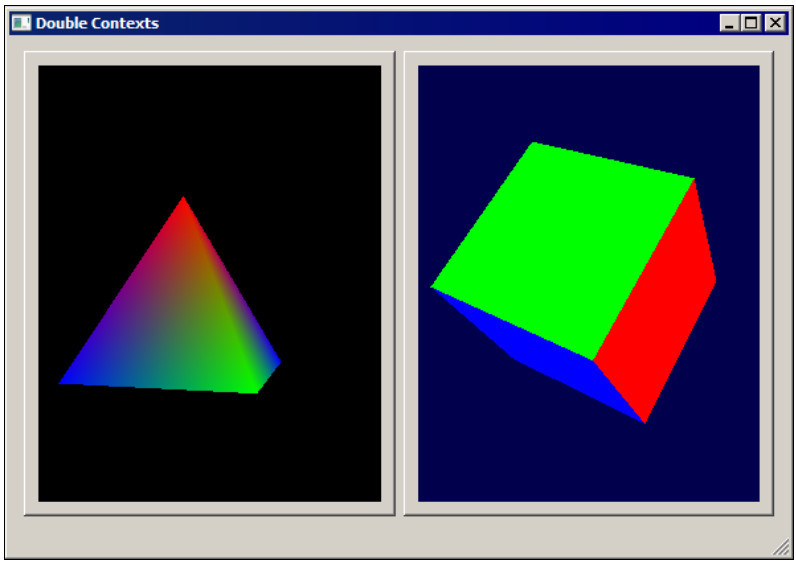


Рис. 8.3. Две сцены OpenGL в одной программе

Листинг 8.5. Класс `GLWidget1`

```
class GLWidget1 : public QGLWidget
{
    Q_OBJECT
public:
    explicit GLWidget1(QWidget *parent = 0);
    ~GLWidget1();
signals:
public slots:
protected:
    void initializeGL();
    void resizeGL(int width, int height);
    void paintGL();
private:
    double x;
};
```

Три метода, которые мы должны перекрыть в классе-потомке `QGLWidget`, — это `initializeGL()`, `resizeGL()` и `paintGL()`. Первый из этих методов вызывается тогда, когда виджету необходимо инициализировать сцену OpenGL. За время жизни виджета этот метод может быть вызван больше одного раза, поэтому в нем не следует размещать код, который должен быть выполнен только один раз (такой код лучше разместить в конструкторе класса). Метод `resizeGL()` вызывается тогда, когда виджет меняет свои размеры. В этом методе выполняется традиционный код OpenGL, регулирующий размеры области видимости сцены в соответствии с размерами пор-

та вывода. Наконец, метод `paintGL()` выполняет рисование сцены. Рассмотрим реализацию этих методов (листинг 8.6).

Листинг 8.6. Перекрытые методы в классе `GLWidget1`

```
void GLWidget1::initializeGL()
{
    glShadeModel(GL_SMOOTH);
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
    glClearDepth(1.0f);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
}

void GLWidget1::resizeGL(int width, int height)
{
    glViewport(0,0,width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0f,(GLfloat)width/(GLfloat)height,0.1f,100.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void GLWidget1::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(-0.3f,0.0f,-6.0f);
    glRotatef(x, 0.0f, 1.0f, 0.0f);
    glBegin(GL_TRIANGLES);
    ...
    glEnd();
    x += 1.0;
}
```

Те, кто знаком с языком OpenGL, поймут, что именно делается в этих методах. Обращаю внимание на то, что хотя по умолчанию виджеты OpenGL используют двойную буферизацию, нет необходимости переключать буферы явным образом, если конечно, вы не отключите автоматическое переключение буферов с помощью метода `setAutoBufferSwap()`. В этом случае переключать буферы придется с помощью вызова метода `swapBuffers()`. Обратите внимание, что мы переключаем буферы методами виджета `QGLWidget`, а не командами OpenGL.

Анимация сцены, как и в предыдущем примере, выполняется с помощью встроенного таймера главного окна (листинг 8.7).

Листинг 8.7. Перекрытые методы в классе `GLWidget1`

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    glw1 = new GLWidget1(ui->frame);
    glw2 = new GLWidget2(ui->frame_2);
    ...
    startTimer(50);
}

void MainWindow::timerEvent(QTimerEvent *)
{
    glw1->makeCurrent();
    glw1->updateGL();
    glw2->makeCurrent();
    glw2->updateGL();
}
```

Метод `updateGL()` заставляет виджет перерисовывать сцену.

Шейдеры OpenGL в Qt

Жизнь не стоит на месте, и, начиная с версии 4.6, библиотека Qt обзавелась поддержкой шейдеров OpenGL и OpenGL-ES. Qt предоставляет нам два класса, предназначенных для работы с шейдерами: `QGLShaderProgram` и `QGLShader`. Класс `QGLShaderProgram` управляет шейдерами, компилирует и связывает шейдерные программы, написанные на языке GLSL (OpenGL Shading Language). Этот класс также позволяет обмениваться данными между нашими программами, написанными на Qt, и шейдерными программами. Класс `QGLShaderProgram` поддерживает работу с вершинными, геометрическими и фрагментными шейдерами. В качестве языка программирования используется GLSL или GLSL-ES. Фактическая компиляция шейдера того или иного поддерживаемого типа выполняется объектами класса `QGLShader`.

В качестве демонстрации поддержки шейдеров в Qt мы рассмотрим программу shader-demo, которая отображает квадрат и использует фрагментный шейдер для его раскраски (эту программу вы сможете найти в каталоге Ch8/shader-demo).

Как и в предыдущем примере, самым главным и самым интересным виджетом программы-демонстратора шейдеров будет виджет `GLShaderWidget`, являющийся потомком класса `QGLWidget`.

Имеет смысл подробнее рассмотреть класс `GLShaderWidget` демонстратора шейдеров (листинг 8.8).

Листинг 8.8. Перекрытые методы в классе GLWidget1

```
class GLShaderWidget : public QGLWidget
{
    Q_OBJECT
public:
    explicit GLShaderWidget(QWidget *parent = 0);
    ~GLShaderWidget();
signals:
public slots:
protected:
    void initializeGL();
    void resizeGL(int width, int height);
    void paintGL();
    void loadShader(const QString &vshader, const QString &fshader);
private:
    int frameCounter;
    QGLShaderProgram * shaderProgram;
    QGLShader * vertexShader, * fragmentShader;
    double x;
};
```

Помимо стандартных методов, которые необходимо перекрыть к классе-потомке `QGLWidget`, мы добавили еще один метод — `loadShader()`. Этот метод нужен нам просто для того, чтобы сделать код более читабельным. Задача метода `loadShader()` заключается в том, чтобы загрузить шейдерные программы, соответственно, для вершинного и фрагментного шейдеров из файлов шейдерных программ, написанных на языке GLSL. Имена этих файлов передаются методу в качестве аргументов. Мы также добавили в определение класса переменные-указатели на объекты `QGLShaderProgram` и `QGLShader`. Начнем исследование класса с метода `loadShader()` (листинг 8.9).

Листинг 8.9. Перекрытые методы в классе GLWidget1

```
void GLShaderWidget::loadShader(const QString &vshader,
    const QString &fshader)
{
    if(shaderProgram)
        {
        shaderProgram->release();
        shaderProgram->removeAllShaders();
        }
    else shaderProgram = new QGLShaderProgram;
    if(vertexShader)
        {
        delete vertexShader;
```

```
        vertexShader = NULL;
        }
    if(fragmentShader)
        {
        delete fragmentShader;
        fragmentShader = NULL;
        }
    QFileInfo vsh(vshader);
    if(vsh.exists())
        {
        vertexShader = new QGLShader(QGLShader::Vertex);
        if(vertexShader->compileSourceFile(vshader))
            shaderProgram->addShader(vertexShader);
        else qWarning() << QString::fromUtf8("Ошибка вершинного шейдера")
            << vertexShader->log();
        }
    QFileInfo fsh(fshader);
    if(fsh.exists())
        {
        fragmentShader = new QGLShader(QGLShader::Fragment);
        if(fragmentShader->compileSourceFile(fshader))
            shaderProgram->addShader(fragmentShader);
        else qWarning()
            << QString::fromUtf8("Ошибка фрагментного шейдера")
            << fragmentShader->log();
        }
    if(!shaderProgram->link())
        {
        qWarning()
            << QString::fromUtf8("Ошибка компоновки шейдерной программы")
            << shaderProgram->log();
        }
}
```

В начале своей работы метод `loadShader()` проверяет, созданы ли объекты для работы с шейдерами (в конструкторе класса `GLShaderWidget` необходимо присвоить значения `NULL` соответствующим указателям). Это связано с тем, что метод `loadShader()` вызывается из метода `initializeGL()`, а этот метод, как мы уже знаем, может быть вызван не один раз в ходе жизненного цикла виджета `GLShaderWidget`. Если объекты, необходимые для работы с шейдерами, уже созданы, незачем выполнять эту операцию повторно. Мы создаем вершинный и фрагментный шейдеры, указывая в конструкторах соответствующих объектов значения `QGLShader::Vertex` и `QGLShader::Fragment`. Преобразование программы на языке GLSL в скомпилированную шейдерную программу выполняет метод `compileSourceFile()` класса `QGLShader`. У этого метода есть "двойник" — метод `compileSourceCode()`, который компилирует шейдерную программу из исходного текста, переданного методу в качестве строки. При этом строка исходного текста может быть объектом `QString`, `QByteArray` или

строкой в стиле C (массивом `char` с нулевым окончанием). Стоит отметить, что компилятор GLSL, используемый Qt, не принимает исходные тексты программ ни в каких кодировках, кроме восьмибитной. При использовании объекта `QString` текст будет автоматически преобразован в нужную кодировку, а вот при использовании объектов `QByteArray` вам самим придется позаботиться об этом. То же самое касается загрузки текста программ из файлов. Поскольку у загрузчика нет способа выяснения кодировки текста (для его последующего преобразования), файлы программ должны быть написаны в восьмибитной кодировке.

Связывание шейдерных программ и превращение их в исполнимый объект OpenGL выполняет объект класса `QGLShaderProgram`. Сначала мы передаем этому объекту объекты, представляющие шейдеры (с помощью метода `addShader()`). Затем мы выполняем компоновку шейдерных программ с помощью метода `link()` объекта этого класса. Перекрыв метод `link()`, мы можем инициализировать униформные переменные, но мы этого делать не будем, хотя в используемой нами программе фрагментного шейдера такая переменная есть (листинг 8.10).

Листинг 8.10. Программа для фрагментного шейдера

```
uniform vec3 myColor;
void main()
{
    gl_FragColor = vec4(myColor, 1.);
}
```

Как именно мы передаем значение этой переменной, будет рассказано далее.

> ***Примечание***
>
> Мы не будем останавливаться подробно на языке GLSL, так же как мы не останавливались на описании OpenGL. Эти темы выходят за рамки данной книги. Тем более, что GLSL — не единственное средство, которой Qt предполагает использовать для программирования графического процессора.

В случае с шейдерными программами мы в первый (но не последний) раз сталкиваемся с ситуацией, когда некоторая часть нашей программы компилируется во время выполнения самой программы. Это означает, что во время выполнения программы, использующей шейдеры, нам придется иметь дело как с ошибками времени выполнения, так и с ошибками времени компиляции, когда шейдерные программы просто не удается скомпилировать, в основном, из-за неверного синтаксиса. У классов `QGLShader` и `QGLShaderProgram` есть метод `log()`, который возвращает информацию о таких ошибках в текстовой форме.

Если шейдерные программы собраны успешно, наше приложение выводит на консоль сообщение:

```
QGLShader::link: "Vertex shader(s) linked, fragment shader(s) linked."
("Вершинные шейдеры скомпонованы, фрагментные шейдеры скомпонованы.")
```

Это сообщение создается самими классами Qt.

Рассмотрим теперь остальные методы класса `GLShaderWidget` (листинг 8.11).

Листинг 8.11. Реализация методов класса `GLShaderWidget`

```
GLShaderWidget::GLShaderWidget(QWidget *parent) : QGLWidget(parent)
{
    makeCurrent();
    shaderProgram = NULL;
    vertexShader = NULL;
    fragmentShader = NULL;
}

GLShaderWidget::~GLShaderWidget()
{
    delete shaderProgram;
    delete vertexShader;
    delete fragmentShader;
}
void GLShaderWidget::initializeGL()
{
    loadShader(":/Basic.vsh", ":/Basic.fsh");
    shaderProgram->bind();
    x = 0;
    frameCounter = 0;
}

void GLShaderWidget::resizeGL(int width, int height)
{
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.,((GLfloat)width)/((GLfloat)height),0.1f,1000.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void GLShaderWidget::paintGL()
{
    glLoadIdentity();
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glClearColor(0.25f, 0.25f, 0.25f, 0.0f);
    frameCounter ++;
    glTranslatef(0.0f, 0.0f, -5.0f);
    glRotatef(frameCounter,0.0f,0.0f,0.5f);
    glBegin(GL_QUADS);
        glVertex3f(-1.0, -1.0,0.0);
        glVertex3f(1.0, -1.0,0.0);
```

```
        glVertex3f(1.0, 1.0,0.0);
        glVertex3f(-1.0, 1.0,0.0);
    glEnd();
    x += 0.1;
    shaderProgram->setUniformValue("myColor", sin(x), cos(x), 1.);
}
```

Начнем с конструктора. Поскольку виджет `GLShaderWidget` — единственный компонент, использующий графику OpenGL в нашей демонстрационной программе, мы делаем контекст OpenGL этого виджета текущим контекстом непосредственно в его конструкторе. Далее мы присваиваем значение `NULL` всем указателям на объекты, работающие с шейдерами. Делается это по причинам, которые обсуждались ранее.

В методе `initializeGL()` мы создаем шейдеры с помощью метода `loadShader()`, а затем вызываем метод `bind()` созданного в `loaShader()` объекта `shaderProgram`. Этот метод связывает шейдерную программу и текущий контекст OpenGL. Следует заметить, что у класса `QGLShaderProgram` есть метод `programId()`, который возвращает идентификатор шейдерной программы, присвоенный ей OpenGL. Этот идентификатор можно использовать для связывания шейдерной программы с контекстом с помощью команды OpenGL `glUseProgram()`.

Класс `QGLShaderProgram` позволяет не только собрать шейдерную программу, но и обмениваться с ней значениями. В тексте программы для фрагментного шейдера была определена переменная `myColor`, имеющая тип `vec3`, т. е. вектор из трех чисел с плавающей точкой. Для передачи значения униформной переменной используется метод `setUniformValue()`. Первый аргумент метода — имя переменной в шейдерной программе. Затем следуют значения, которыми мы инициализируем шейдерную переменную. У метода `setUniformValue()` 50 перегруженных вариантов (вероятно, этот метод является чемпионом Qt по количеству перегрузок), каждый из которых соответствует определенному типу униформной переменной. Кроме метода

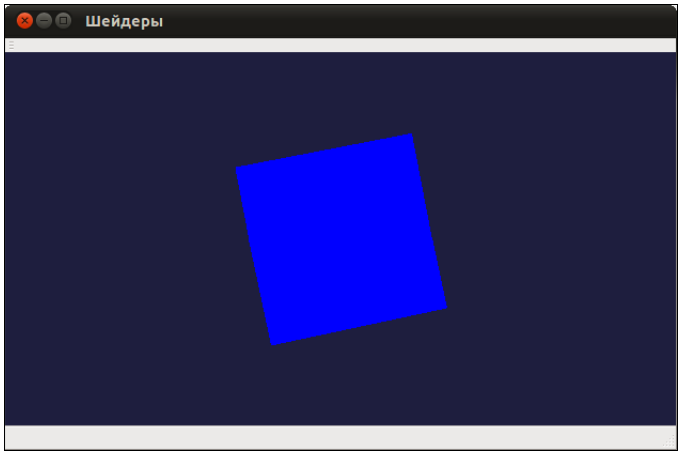


Рис. 8.4. Программа shader-demo

`setUniformValue()`, у класса `QGLShaderProgram` есть еще метод `setAttributeValue()`, позволяющий задавать значения атрибутов.

Если ваша система поддерживает шейдеры, то, собрав программу shader-demo, вы увидите вращающийся квадрат постоянно меняющегося цвета (рис. 8.4).

Взгляд в будущее

Обилие технологий, которые использует Qt, настолько велико, что ни одна книга, вероятно, не способна описать их все (и эта книга — не исключение). Отсюда следует несколько выводов. Во-первых, вряд ли в настоящее время даже среди сертифицированных разработчиков Qt найдется человек, который знает все технологии Qt. Во-вторых, многие серьезные программы могут быть написаны целиком на Qt, без привлечения сторонних библиотек. Если, конечно, вы сумеете отыскать в океане технологий Qt то, что вам нужно.

Уже сейчас мы можем познакомиться с подсистемой Qt/3D, которая войдет в версию 4.8 библиотеки Qt.

Подсистема Qt/3D

Если дать простейшее определение Qt/3D, это надстройка над OpenGL/ES. Однако ее цель, — ни много ни мало, — перенести на платформы OpenGL/ES те методы, к которым мы привыкли при работе с двухмерной графикой. В некотором смысле задачи Qt/3D противоположны задачам Arthur. В Arthur мы использовали трехмерную "подложку" для ускорения и расширения возможностей двухмерных графических операций. В Qt/3D технологии, отработанные в области двухмерной графики, переносятся в трехмерный мир. Помимо этого, Qt/3D позволяет поддерживать ряд устройств, с которыми раньше нельзя было работать средствами Qt, например стереоочки и трехмерные мыши.

Рассмотрим базовые классы Qt/3D и сравним их с классами других графических систем Qt. Класс `QGLPainter` является своего рода аналогом класса `QPainter` для OpenGL/ES. Так же, как и у класса `QPainter`, у `QGLPainter` есть методы `begin()` и `end()`. Вместо семейства методов `draw*` (`drawLine()`, `drawRect()`) у `QGLPainter` есть один метод `draw()`, доступный в нескольких перегруженных вариантах. Первым параметром у всех этих перегруженных методов является значение типа `QGL::DrawingMode`, которое определяет, что же именно будет рисовать метод `draw()`. Конкретные значения перечислимого типа `QGL::DrawingMode` соответствуют стандартным графическим примитивам OpenGL: `QGL::Lines`, `QGL::Triangles`, `QGL::TriangleStrip` и т. д. Координаты для рисования примитивов берутся из вершинных массивов (vertex arrays). Метод `isCullable()`, существующий в двух перегруженных вариантах, позволяет проверить, находится ли заданная точка (или, во втором варианте, трехмерный "ящик", т. е. прямоугольный параллелепипед) внутри видимого пространства. Второй вариант метода возвращает значение `false` только в том случае, если прямоугольный параллелепипед целиком находится за пределами видимого пространства. Кроме того, `QGLPainter` позволяет манипулировать мат-

рицами преобразования, материалами, освещением, камерой и другими объектами OpenGL.

Предком класса `QGLPainter` является класс `QOpenGLFunctions`. Этот класс инкапсулирует функции OpenGL/ES 2.0 таким образом, что при использовании этого класса вам не придется беспокоиться о переносе приложения с одной платформы на другую. Просто вместо функций OpenGL нужно вызывать одноименные методы класса `QOpenGLFunctions`. Таким образом, классы Qt/3D полностью скрывают разницу между обычным OpenGL и OpenGL для встроенных систем. Один и тот же код будет одинаково работать и на настольном компьютере, и на мобильном телефоне.

Класс `QGLView` является потомком знакомого нам класса `QGLWidget`. В отличие от `QGLWidget`, который, сам по себе, может только показывать трехмерную сцену, `QGLView` обрабатывает события клавиатуры и мыши таким образом, чтобы зритель мог перемещаться по трехмерной сцене. Кроме того, класс `QGLView` позволяет создавать изображения для тех стереоустройств, которые формируют отдельное изображение для каждого глаза.

Очень полезны классы `QPlane3D`, `QRay3D`, `QTriangle3D`. Эти классы упрощают выполнение таких важных в программировании трехмерной графики операций, как нахождение точки пересечения прямой и плоскости, принадлежности точки прямой и т. д.

Перед программистами OpenGL часто возникает проблема, связанная с трансляцией координат окна виджета в координаты трехмерного мира модели OpenGL. Например, когда пользователь щелкает мышью в окне, в котором выполняется вывод графики OpenGL, по одному из трехмерных объектов, нужно, чтобы этот объект как-то отреагировал на щелчок пользователя. Для этого требуется геометрическое преобразование, которое по координатам точки, заданным в системе координат окна виджета, создает луч, соответствующий этой точке.

Напомню, что с точки зрения OpenGL, изображение, которое формируется в окне программы, представляет собой проекцию трехмерной сцены на двухмерную плоскость окна. Луч, совпадающий с линией проекции некоторой точки трехмерного мира, проецируется на плоскость виджета как точка. При преобразовании координат виджета в координаты трехмерного мира мы решаем обратную задачу: находим луч, совпадающий с линией проекции, соответствующей данной точке. Решение этой задачи требует знаний аналитической геометрии и громоздких вычислений. К счастью, такие классы, как `QPlane3D`, `QRay3D`, `QVector3D`, и специальные методы классов `QGLView` и `QGLCamera`, упрощают решение этой задачи. Программа tank, входящая в дистрибутив Qt/3D (рис. 8.5), демонстрирует преобразование точки окна программы в луч в трехмерной сцене OpenGL.

Если после всего сказанного вы захотите получить библиотеки Qt/3D в свое распоряжение, готовьтесь преодолевать препятствия. Исходные тексты Qt/3D открыты на тех же условиях, что и тексты Qt, но вас ждут технические трудности. Поскольку на момент написания этих строк Qt/3D еще не является официальной частью Qt, библиотеки придется собирать из исходных текстов. Получить исходные тексты можно из репозитория git. Подробная инструкция содержится на сайте **http://**

doc.qt.nokia.com/qt3d-snapshot/qt3d-building.html. Отмечу только, что для сборки Qt/3D вам понадобятся еще и исходные тексты самой Qt, не ниже версии 4.7. То есть сначала надо собрать библиотеку Qt из исходных текстов (готовый SDK тут не подойдет), а затем уже собирать Qt/3D. Если вы все еще не передумали, я предлагаю рассмотреть простой пример использования Qt/3D.

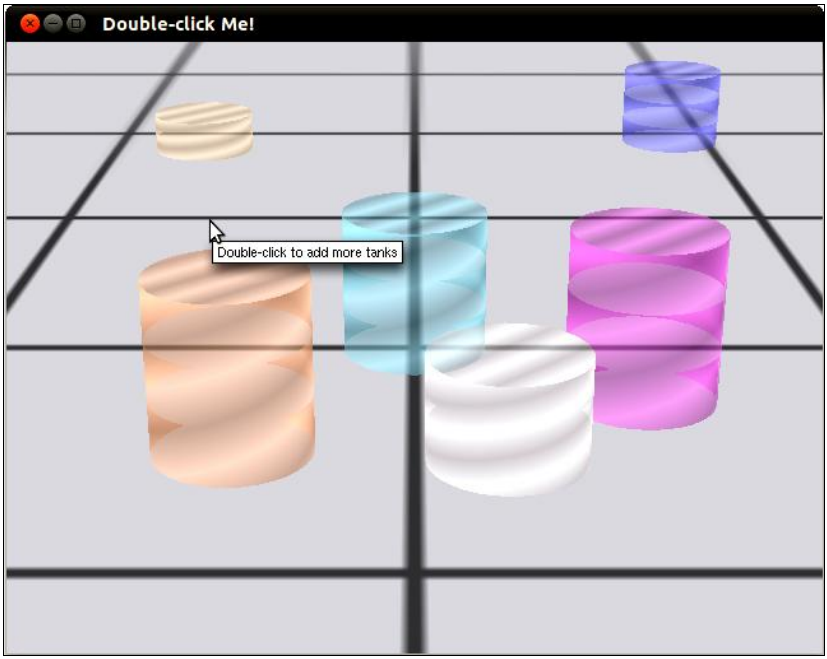


Рис. 8.5. Программа tank взаимодействует с мышью

Сколько строк кода OpenGL необходимо, чтобы вывести на экран текстурированную полусферу? Вероятно, несколько десятков. В Qt/3D эта же задача решается значительно меньшим количеством строк кода. Программа dome, исходные тексты которой вы найдете на сайте **http://symmetrica.net/qt47book/**, наглядно это демонстрирует. Основой программы является объект класса `DomeView`, который наследует классу `QGLView` (листинг 8.12).

Листинг 8.12. Объявление класса `DomeView`

```
class DomeView : public QGLView
{
    Q_OBJECT
public:
    DomeView(QWidget *parent = 0);
    ~DomeView();
protected:
    void paintGL(QGLPainter *painter);
    void timerEvent (QTimerEvent *);
```

```
private:
    QGLSceneNode * dome;
    QGLTexture2D tex;
    int angle;
};
```

Первое, на что нужно обратить внимание при анализе класса `DomeView`, — мы перекрываем только метод `paintGL()`, но не `initializeGL()` и не `resizeGL()`. В классе `QGLView` эти методы уже выполняют все, что требуется нам по умолчанию. Что, разумеется, не мешает нам перекрыть их, если настройки по умолчанию нас не устраивают.

Функции класса `QGLTexture2D` должны быть понятны из его названия. Этот класс отвечает за работу с двухмерными текстурами. Гораздо интереснее второй класс — `QGLSceneNode`. Класс `QGLSceneNode` и его производные позволяют нам строить трехмерные сцены как иерархии объектов. При этом родительский элемент иерархии "отвечает" за поведение дочерних элементов. Например, при применении геометрических преобразований, описаний свойств материалов или графических эффектов к родителю те же преобразования и эффекты по умолчанию применяются к дочерним элементам. Создав элемент класса `QGLSceneNode` один раз, мы можем манипулировать им и всеми связанными с ним элементами, как единым графическим примитивом. Например, для графического отображения иерархической сцены достаточно вызвать один метод `draw()` объекта класса `QGLSceneNode`. От обсуждения общей структуры класса `DomeView` перейдем к его реализации (листинг 8.13).

Листинг 8.13. Реализация класса `DomeView`

```
DomeView::DomeView(QWidget *parent)
    : QGLView(parent)
{
    QGLBuilder builder;
    builder << QGL::Faceted;
    builder << QGLDome();
    dome = builder.finalizedSceneNode();
    startTimer(50);
    tex.setImage(QImage(QLatin1String(":/stripes.png")));
}

DomeView::~DomeView()
{
    delete dome;
}

void DomeView::paintGL(QGLPainter *painter)
{
    painter->setStandardEffect(QGL::LitDecalTexture2D);
    painter->setFaceColor(QGL::AllFaces, QColor(170, 202, 0));
```

```
    tex.bind();
    painter->modelViewMatrix().rotate(angle, 1, 1, 1.0f);
    painter->modelViewMatrix().rotate(-45, 1,  1, 1.0f);
    dome->draw(painter);
}

void DomeView::timerEvent (QTimerEvent *)
{
  angle +=1;
  updateGL();
}
```

В процессе изучения реализации класса `DomeView` нам придется познакомиться с еще одним классом Qt/3D — `QGLBuilder`. Для того чтобы создать объект класса `QGLSceneNode`, содержащий требуемые нам данные в оптимальной для отображения форме, необходимо воспользоваться услугами "помощника", каковым и является объект класса `QGLBuilder`. Идея, которая лежит в основе `QGLBuilder`, заключается в следующем: объект класса `QGLSceneNode` может содержать множество треугольников и других примитивов OpenGL. Кроме того, объект `QGLSceneNode` может содержать дочерние объекты с не меньшим, а то и бóльшим количеством графических примитивов. Очевидно, что рисование этих сложных геометрических структур может быть оптимизировано. Именно это и делает класс `QGLBuilder`. В рассматриваемом примере мы используем перегруженный оператор <<. Для класса `QGLBuilder` этот оператор существует в двух вариантах. В первом варианте операндом оператора должно быть значение типа `Smoothing`, которое указывает, должны ли сглаживаться грани фигуры, и, если должны, то как именно. Константа `QGL::Faceted` означает, что мы не хотим сглаживания граней (даже полусфера, заданная по умолчанию, содержит так много граней, что сглаживание не требуется).

Во втором варианте этот оператор принимает операнд типа `QGeometryData`, который представляет собой набор вершин и их атрибутов. Как ни странно, класс `QGLDome`, который собственно и отвечает за рисование полусферы, не является потомком `QGeometryData`. В текущей версии Qt/3D у этого класса вообще нет предка (поскольку документация по Qt/3D, доступная на сайте проекта, не упоминает иерархические отношения классов, мы можем сделать вывод, что в будущем они могут меняться). Класс `QGLDome` обладает собственным перегруженным оператором <<, который позволяет добавлять объект этого класса в коллекцию объекта `QGLBuilder`. Вероятно, такая сложная структура тоже реализована в интересах оптимизации. Объект `dome` класса `QGLSceneNode` создается с помощью метода `finalizedSceneNode()` класса `QGLBuilder`. Все, что нам теперь нужно сделать для рисования полусферы, — добавить вызов `dome.draw()` в метод `paintGL()` класса `DomeView`.

Назначение текстуры объекту в методе `paintGL()` соответствует распространенной практике программирования в OpenGL, но не очень соответствует идеологии класса `QGLSceneNode`, который должен, по идее, содержать всю информацию, касающуюся своего трехмерного объекта. Можно ли задать информацию о текстуре объекта `QGLSceneNode` в самом объекте? Да, можно (листинг 8.14).

Листинг 8.14. Привязка текстуры к объекту `QGLSceneNode`

```
QGLBuilder builder;
builder << QGL::Faceted;
builder << QGLDome();
dome = builder.finalizedSceneNode();
tex.setImage(QImage(QLatin1String(":/stripes.png")));
QGLMaterial * material = new QGLMaterial();
material->setTexture(&tex);
dome->setMaterial(material);
```

Метод `setMaterial()` позволяет задать материал объекта (описываемый объектом класса `QGLMaterial`), а среди прочих параметров материала можно указать и текстуру (метод `material->setTexture()`). Обратите внимание на то, что объекты `material` и `tex` должны существовать, пока существует объект `dome`, который получил на них ссылки. Теперь из метода `paintGL()` можно убрать вызов `tex.bind()`, поскольку сам объект `dome` "знает", какая текстура ему назначена. В результате мы получим текстурированную полусферу, которую можно перемещать при помощи клавиш-стрелок или приближать и отдалять с помощью колесика мыши (рис. 8.6).

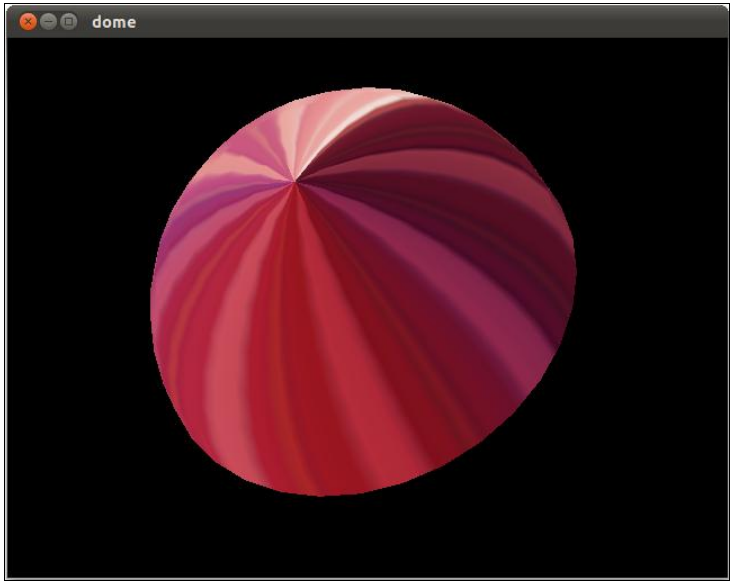


Рис. 8.6. Полусфера, покрытая текстурой

Примечание

Необходимо сказать несколько слов о сборке проектов Qt/3D. Даже после выполнения make install эта подсистема не становится частью стандартной установки Qt. Вы должны позаботиться о доступности всех заголовочных файлов вашему проекту, а также о том, чтобы библиотека libQt3D.so (или, соответственно, Qt3D.dll) была доступна для связывания и динамической загрузки в вашей программе.

Если вы пишете программу, которая использует различные "продвинутые" функции OpenGL, вам пригодится средство надежного определения возможностей видеокарты той машины, на которой ваша программа запущена. В Qt/3D эту задачу решает класс `QGLInfo`. У класса `QGLInfo` есть один-единственный метод `report()`, возвращающий в одной строке все данные, которые удалось собрать системе. В настоящий момент этого класса нет в основной ветке исходных текстов Qt/3D, и документация о нем отсутствует. Исходные тексты класса расположены в директории qt-labs/qt3d/blobs/master/util/qglinfo/. Там же находится программа, которая демонстрирует его использование (рис. 8.7). Формат, в котором класс `QGLInfo` выдает данные об OpenGL, может вам не подойти, но, изучив исходные тексты класса, вы сумеете получать необходимые сведения в любом формате.

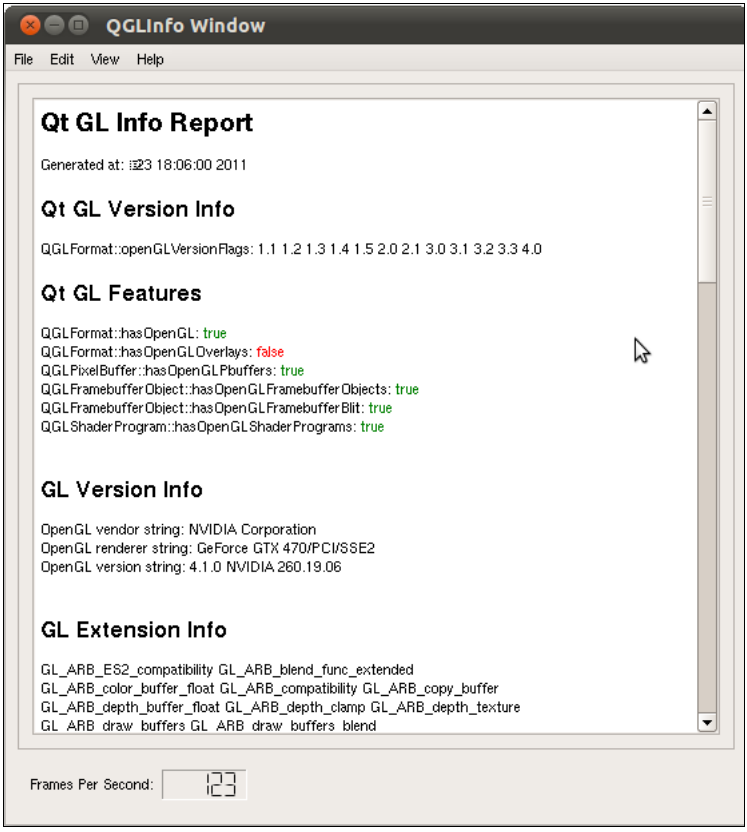


Рис. 8.7. Окно программы qglinfo

Поддержка OpenCL в Qt 4.8

Видеокарты современных компьютеров вполне сопоставимы по мощности с их основными блоками (а в решении некоторых задач даже превосходят центральный процессор). Тем не менее эта мощь редко используется на все 100%, разве что в

новейших компьютерных играх. Идея использовать расширенные возможности видеокарт не только для игр и трехмерного моделирования возникла давно (и не первый год применяется на практике). Например, язык GLSL, с которым мы познакомились в этой главе, может использоваться не только для программирования шейдеров, но и, например, для вычислений в области векторной алгебры. Основная проблема, связанная с применением графических процессоров в неграфических целях, до сих пор заключается в отсутствии единого стандарта такого их использования. Очень часто решение приходится создавать под конкретную систему. Ситуация осложняется еще и тем, что традиционные языки программирования плохо поддерживают те инструкции современных процессоров, которые ориентированы на одновременную обработку больших массивов данных. Для реализации таких инструкций приходится либо опускаться на уровень ассемблера, либо использовать специальные наборы функций, написанных на том же уровне. Однако ситуация начинает постепенно меняться. Язык OpenCL (Open Computing Language, открытый язык вычислений) должен стать для сложных вычислений тем же, чем стал язык OpenGL для трехмерной компьютерной графики, — кросс-платформенным средством, позволяющим максимально задействовать имеющиеся в системе ресурсы для параллельной обработки данных.

Адрес официальной страницы OpenCL — **www.khronos.org/opencl**. На этой странице вы можете узнать о целях и возможностях языка, а также о текущем состоянии его разработки и распространения. Поскольку OpenCL ориентирован на работу со специализированными процессорами (или процессорными модулями), предназначенными для параллельной обработки данных, представленных в виде векторов, лучше всего он подходит именно для решения задач, в которых используются массивы данных. Примерами таких задач являются быстрое преобразование Фурье, вычисление сверток, применение к данным цифровых фильтров и т. п. Задачи такого рода возникают не только при обработке графики или в научных расчетах, но и, например, при обработке аудиоданных, так что язык OpenCL может пригодиться многим разработчикам.

Примечание

Хотя программы, использующие OpenCL, могут загрузить специализированные процессоры вашей системы так, что у нее не хватит мощности для выполнения обычных задач, такая ситуация вовсе не является типичной. Как правило, программы, использующие OpenCL для решения математических задач, могут выполняться совместно с другими программами, например графическими.

Важная особенность языка OpenCL — высокий уровень абстракции. OpenCL не делает принципиальной разницы между графическим процессором видеокарты и расширенными наборами инструкций центрального процессора. Это означает, что программы OpenCL могут выполняться, хотя и с разным уровнем производительности, как на системах с мощными видеокартами, так и на системах, где сами видеопроцессоры не поддерживают сложные наборы инструкций.

Не следует, однако, быть слишком большими оптимистами в том, что касается современного состояния OpenCL. Далеко не все видеокарты, потенциально способ-

ные поддерживать OpenCL, могут делать это на практике, т. к. для них отсутствуют специальные драйверы (ситуация, аналогичная той, в которой аппаратные средства видеокарты поддерживают ускорение операций OpenGL, а реализация драйвера — нет).

Но это еще не все. Если OpenGL создавалась как клиент-серверная система, так что для работы с аппаратно-ускоренными средствами OpenGL вам необходимы только видеокарта и драйвер, поддерживающий эти операции, то для работы с OpenCL необходим еще и специальный компилятор, который преобразует программу OpenCL в последовательность команд для определенного процессора. У каждого производителя графических чипсетов эти компиляторы, естественно, свои.

> ***Примечание***
>
> На практике применяются две стратегии работы с OpenCL. В одном случае программа OpenCL распространяется в виде исходных текстов и компилируется во время выполнения основной программы компилятором, установленным в конкретной системе, с учетом особенностей этой системы. Если вы не хотите распространять свои модули OpenCL в исходных текстах, вы можете предварительно скомпилировать их для нескольких целевых платформ. Основная программа во время выполнения определит, на какой платформе она выполняется, и загрузит подходящий модуль.

В чем же заключается поддержка OpenCL в Qt? Как и в случае с Qt/3D, основная цель здесь — создание совместимого с Qt интерфейса, который максимально скрывал бы от программиста особенности работы OpenCL на разных платформах.

Хотя поддержка OpenCL в Qt появится только в версии 4.8, протестировать работу QtOpenCL можно уже с версией 4.7 и даже с более ранними версиями, причем исходных текстов самих библиотек Qt для этого не нужно. Так же как и Qt/3D, QtOpenCL необходимо собрать из исходных текстов, которые доступны в репозитории git. Подробное описание процесса сборки и установки, а также другую информацию, можно найти на сайте **http://doc.qt.nokia.com/opencl-snapshot/**. Отмечу сразу, что, поскольку реализация QtOpenCL носит пока еще экспериментальный характер, сборка этой системы может оказаться делом нетривиальным. Мне, например, пришлось вносить изменения в исходные тексты только для того, чтобы система QtOpenCL скомпилировалась. Впрочем, разработка не стоит на месте и к моменту выхода книги из печати положение дел со сборкой QtOpenCL может улучшиться.

Как уже отмечалось, для того чтобы задействовать видеокарту в решении математических задач, вам нужен драйвер видеокарты с поддержкой OpenCL 1.1 и соответствующий SDK с компилятором и библиотеками среды времени выполнения. Убедиться в том, что все это у вас есть и правильно настроено, поможет программа clinfo. Исходные тексты этой программы вы найдете в директории util пакета QtOpenCL.

Эта программа выводит довольно много информации обо всех обнаруженных платформах OpenCL. Посмотрим на сокращенный вариант вывода clinfo в моей системе (листинг 8.15). Если вы пользуетесь Qt Creator, то для того чтобы увидеть этот вывод, вам необходимо запустить программу в режиме отладки.

Листинг 8.15. Вывод программы clinfo

```
OpenCL Platforms:
    Platform ID       : 5851472
    Profile           : FULL_PROFILE
    Version           : OpenCL 1.1 WINDOWS
    Name              : Intel(R) OpenCL
    Vendor            : Intel(R) Corporation
    Extension Suffix  : Intel
    Extensions        :
...
    Platform ID       : 239706308
    Profile           : FULL_PROFILE
    Version           : OpenCL 1.1 AMD-APP-SDK-v2.4 (595.10)
    Name              : AMD Accelerated Parallel Processing
    Vendor            : Advanced Micro Devices, Inc.
    Extension Suffix  : AMD
...
OpenCL Devices:
...
    Device ID         : 240713712
    Platform ID       : 239706308
    Vendor ID         : 4098
    Type              : GPU
    Profile           : FULL_PROFILE
    Version           : OpenCL 1.1 AMD-APP-SDK-v2.4 (595.10)
    Driver Version    : CAL 1.4.1385 (VM)
    Language Version  : OpenCL C 1.1
    Name              : Cypress
    Vendor            : Advanced Micro Devices, Inc.
    Available         : true
    Compute Units     : 14
    Clock Frequency   : 800 MHz
    Address Bits      : 32
    Byte Order        : Little Endian
    ...
    Device ID         : 240718192
    Platform ID       : 239706308
    Vendor ID         : 4098
    Type              : CPU
    Profile           : FULL_PROFILE
    Version           : OpenCL 1.1 AMD-APP-SDK-v2.4 (595.10)
    Driver Version    : 2.0
    Language Version  : OpenCL C 1.1
    Name              : Intel(R) Core(TM)2 Quad CPU    Q9400  @ 2.66GHz
    Vendor            : GenuineIntel
    Available         : true
```

```
    Compute Units      : 4
    Clock Frequency    : 2666 MHz
    Address Bits       : 32
    Byte Order         : Little Endian
   ...
```

Как видим, программа нашла две платформы OpenCL (Intel и AMD) и два устройства (центральный процессор Intel и графический процессор AMD, расположенный на видеокарте). Все это означает, что я могу использовать программы OpenCL в программах Qt, задействуя в качестве процессора как ЦП, так и видеопроцессор.

В качестве примера применения графического процессора для решения неграфических задач рассмотрим программу vectoradd из пакета `QtOpenCL`. Эта программа использует OpenCL для сложения двух векторов, каждый из которых состоит из 2048 чисел типа `int`. Я внес небольшие модификации в исходный текст этой программы (листинг 8.16).

Листинг 8.16. Модифицированный вариант программы vectoradd

```
int main(int, char **)
{
    QCLContext context;
    if (!context.create(QCLDevice::GPU)) {
        fprintf(stderr, "Could not create OpenCL context for the GPU\n");
        return 1;
    }
    QCLVector<int> input1 = context.createVector<int>(2048);
    QCLVector<int> input2 = context.createVector<int>(2048);
    for (int index = 0; index < 2048; ++index) {
        input1[index] = index/2;
        input2[index] = (index + 1)/2;
    }
    QCLVector<int> output = context.createVector<int>(2048);
    QCLProgram program =
        context.buildProgramFromSourceFile(":/vectoradd.cl");
    QCLKernel kernel = program.createKernel("vectorAdd");
    kernel.setGlobalWorkSize(2048);
    kernel(input1, input2, output);
    for (int index = 0; index < 2048; ++index) {
        if (output[index] != index) {
            fprintf(stderr, "Answer at index %d is %d, should be %d\n",
                    index, output[index], index);
            return 1;
        } else
        printf("Answer is correct: %d\n", index);
    }
    return 0;
}
```

Описание состояния системы OpenCL хранится в объекте `context` класса `QCLContext`. Как и в случае `QGLContext`, класс `QCLContext` является не изобретением разработчиков Qt, а отображением одноименной концепции OpenCL. Для инициализации контекста OpenCL используется метод `create()` класса `QCLContext`. В качестве параметра метода `create()` можно указать устройство OpenCL, на котором следует выполнять программу. Если устройство не указывать, система выберет то устройство, которое она считает наиболее подходящим. Я хочу, чтобы программа выполнялась именно на графическом процессоре, поэтому в качестве аргумента команды указываю `QCLDevice::GPU`.

Затем мы готовим векторы для ввода и вывода данных, собираем программу OpenCL, создаем объект `kernel` (вызов `program.createKernel()`) и локальную рабочую область. После выполнения программы OpenCL мы сравниваем результат сложения двух векторов с ожидаемым.

Как и программа clinfo, программа vectoradd является консольной (не могу не обратить внимания на иронию ситуации: программа, в которой задействованы самые передовые возможности видеопроцессора, выводит данные в консольном режиме). Так что, если вы работаете в Qt Creator, то, как и в случае с программой clinfo, для того чтобы увидеть вывод данной программы, необходимо запустить Qt Creator в режиме отладки.

ГЛАВА 9

Растровая графика и текст

В этой главе мы создадим виджет для показа изображений. Вы, конечно, знаете, что в Qt подобный виджет уже есть, и даже не один. Но наш виджет сможет то, чего не могут виджеты Qt, — показывать изображения, которые целиком не поместятся в памяти программы. Затем мы рассмотрим работу с обогащенным (содержащим элементы форматирования) текстом в Qt.

Вывод на экран больших изображений

Под большими изображениями в этом контексте понимаются такие изображения, которые не могут целиком поместиться в области памяти, выделенной программе, по крайней мере, в 32-битной системе. В наше время, когда географические карты, отсканированные изображения и фотоснимки, сделанные электронными фотоаппаратами с разрешающей способностью около сотни мегапикселов, становятся обычным делом, многим программам нужно уметь обрабатывать такие изображения.

Какой объем памяти занимает изображение? Одно дело — изображение, сохраненное в формате, применяющем алгоритмы сжатия. Такое изображение может занимать совсем немного памяти, особенно если применяется сжатие с потерями. Но допустим, для вывода изображения на экран вы используете растровый формат, в котором каждому пикселу соответствуют три байта — три составляющих цвета. В этом случае изображение, имеющее форму квадрата со стороной в 26 755 пикселов (такие изображения можно найти в Интернете или получить самому, с помощью сканера), займет 2 гигабайта памяти, т. е. все пространство, которое в обычной 32-битной системе выделяется программе. Хотя, будучи сжато в формате JPEG, это изображение может занимать около десятка мегабайт, а в формате PNG, который сжимает данные без потерь, — несколько десятков. Можно, конечно, отмахнуться от проблемы, сказав, что 32-битные системы устарели, по крайней мере, для работы с такими изображениями. Можно добавить также, что форматы, которые используются для вывода изображений, например, в классе `QPixmap`, обычно применяют простейшие алгоритмы сжатия (какой именно формат используется для внутреннего представления данных в объекте `QPixmap`, зависит от платформы, так что делать предварительных предположений тут не стоит). В любом случае, даже в 64-битной системе работа с такими изображениями "в лоб" создаст слишком большую нагрузку на систему и затруднит работу других программ или работу

других компонентов программы, которая выводит большие изображения. Очевидно, для того, чтобы работать с растровой графикой, с произвольным объемом пикселов, придется идти в обход и применять различные трюки. Прежде чем перейти к этим трюкам, рассмотрим три класса, отвечающих за работу с растровой графикой (описание этих классов и различий между ними есть в документации, однако различия эти настолько важны, что есть смысл обсудить их здесь).

Классы *QImage*, *QPicture* и *QPixmap*

Класс `QImage` предназначен для того, чтобы манипулировать графическими данными. Объект этого класса позволяет вам получить доступ к каждому пикселу изображения как к элементу массива. В результате, загрузив растровое изображение в объект `QImage` в нужном формате, вы можете делать с ним все, что угодно. Но за такую свободу действий надо платить. Например, если для представления пикселов вы выбрали формат `QImage::Format_RGB32`, каждому пикселу будет выделено 4 байта (хотя, для представления собственно информации о пикселе достаточно трех, для удобства доступа каждый пиксел представлен 32-битным числом). Это значит, что объем памяти, занимаемый изображением, хранимым в объекте `QImage` при данном формате, следует рассчитывать по формуле h * w * 4, где h и w — высота и, соответственно, ширина изображения в пикселах. Разработчики Qt, конечно, понимают, чем это грозит, так что класс `QImage` может загружать не все изображение из файла, а только его часть. Как именно это делается, описано далее.

Класс `QPicture` позволяет сохранить и затем воспроизвести последовательность операций рисования, выполняемых с помощью объекта класса `QPainter`. Данные сохраняются в собственном формате Qt и могут быть записаны на диск. Объекты `QPicture` похожи на метафайлы Windows. Сейчас они нас не интересуют.

Класс `QPixmap` предназначен специально для вывода растровой графики. Как и другие перечисленные здесь классы, он происходит от класса `QPaintDevice`, и это значит, что с его помощью тоже можно рисовать, но возможности манипуляции графикой в классе `QPixmap` гораздо скромнее, чем в классе `QImage`. Основной заботой разработчиков при создании класса `QPixmap` была скорость вывода хранящегося в нем изображения на экран, для чего `QPixmap` использует внутренний формат хранения данных, наиболее оптимальный для вывода графики на данной конкретной платформе. Но тот формат, который удобен для вывода графики, например, на платформе Linux, может быть не очень удобен для модификации изображений, поэтому функции модификации носят в `QPixmap` вторичный характер. В частности получить значение пиксела, заданного координатами, в классе `QPixmap` гораздо сложнее, чем в классе `QImage` (фактически самый простой способ — преобразовать `QPixmap` в `QImage`). Вместе с тем класс `QPixmap` поддерживает геометрические преобразования изображения в целом и позволяет копировать части изображения в новые объекты `QPixmap`, что может быть очень полезно в некоторых ситуациях.

Виджет для вывода больших изображений

Мы рассмотрим все этапы создания виджета, способного выводить на экран изображения большого размера и не занимающего при этом много места в оператив-

ной памяти. Основная идея этого виджета, в общем-то, проста: разбить изображение на несколько частей и в каждый данный момент времени хранить в памяти ту часть, которая является видимой. Основные трудности, как это часто бывает, заключаются в практических деталях реализации. Как разбить изображение на части, как сделать так, чтобы при прокрутке изображения эти части выглядели единым целым, как ускорить саму прокрутку, учитывая, что все изображение держать в памяти нельзя? Ответы на эти вопросы вы найдете далее.

Начнем с проблемы прокрутки. Допустим, у вас есть изображение с длиной стороны 2000 пикселов. Такое изображение поместится в области памяти, выделенной вашей программе, но наверняка полностью не уместится на экране монитора. Для того чтобы вывести это изображение на экран, мы можем воспользоваться объектом класса `Qlabel`, а для того чтобы добавить полосу прокрутки — объектом класса `QScrollArea`. Схематично это можно представить следующим кодом (листинг 9.1).

Листинг 9.1. Схематичное решение проблемы прокрутки

```
QPixmap * pixmap = new pixmap("picture.jpg");
QLabel * label = new QLabel();
label->setPixmap(*pixmap);
QScrollArea * scrollArea = new QScrollArea();
scrollArea->setWidget(label);
```

Обращаю ваше внимание на важный момент: после вызова `scrollArea->setWidget(label)` объект `label` становится собственностью объекта `scrollArea`. В частности объект `scrollArea` берет на себя управление жизненным циклом объекта `label`.

Теперь давайте представим, что у нас есть некий виджет, например `QFrame`, на котором расположено несколько дочерних виджетов, осуществляющих вывод растровой графики. Пусть, для примера, их будет четыре. Тогда загружаемое изображение можно разбить на четыре части и каждый из четырех виджетов будет загружать и показывать свою часть только в том случае, когда она (или ее фрагмент) будет видима на экране. Класс, который отвечает за загрузку и вывод фрагмента изображения, а также за выгрузку этого фрагмента, когда он невидим, мы назовем `PictureDrawer`. Класс, который отвечает за вывод изображения по частям, — `ImageFrame`. Тогда представленный в листинге 9.1 код можно модифицировать так (листинг 9.2).

Листинг 9.2. Прокрутка с виджетом `ImageFrame`

```
ImageFrame * imf = new ImageFrame ("picture.jpg");
// Конструктор класса ImageFrame создает нужное количество объектов
// PictureDrawer и назначает им фрагменты изображения.
QScrollArea * scrollArea = new QScrollArea();
scrollArea->setWidget(imf);
```

Этот код выглядит проще, чем код из листинга 9.1, но реализация используемых в нем методов на самом деле гораздо сложнее. Мы начнем обсуждение реализации "снизу вверх" — с класса `PictureDrawer`.

Первая задача — загрузить из файла изображения, хранящегося на диске, прямоугольную область, соответствующую части изображения. В этом нам может помочь класс `QImageReader`. Этот класс позволяет загрузить прямоугольный фрагмент изображения, заданный координатами в системе пикселов изображения. Результатом считывания является объект `QImage`, который содержит считанный фрагмент в распакованном виде (т. е. как доступную последовательность пикселов). Класс `QImageReader` позволяет читать фрагмент изображения из форматов, использующих сжатие, не распаковывая все изображение целиком (иначе эта операция не имела бы особого смысла).

> ***Примечание***
>
> Возможность считывать фрагменты изображений реализована для всех форматов графических файлов, которые официально поддерживает Qt. Если вы хотите добавить поддержку какого-то нового формата, вам самостоятельно придется написать кодек, способный это сделать.

Кроме того, класс `QImageReader` позволяет считывать отдельные кадры из некоторых анимационных форматов и многое другое, но эти возможности `QImageReader` нас сейчас не интересуют. Рассмотрим фрагмент метода `load()` класса `PictureDrawer` (листинг 9.3).

Листинг 9.3. Метод `load()` (вариант 1)

```
void PictureDrawer::load(const QString &fileName, const int x1, int y1,
    int x2, int y2)
{
    if (!pixmap) {
        QImageReader ir(fileName);
        ir.setClipRect(QRect(x1, y1, x2 — x1, y2 — y1));
        QImage img = ir.read();
        pixmap = new QPixmap();
        pixmap->convertFromImage(img);
        label->setPixmap(*pixmap);
    }
}
```

Этот метод загружает фрагмент изображения, заданный координатами `x1`, `y1`, `x2`, `y2` из файла, имя которого передается в строке `fileName`. Здесь мы сначала создаем объект `QImageReader`, указывая в качестве аргумента конструктора имя файла, в котором хранится изображение. Затем с помощью метода `setClipRect()` класса `QImageReader` мы указываем прямоугольный фрагмент изображения. Данные о фрагменте изображения передаются методу в виде объекта класса `QRect`.

Примечание

У класса `QRect` есть несколько конструкторов. В том варианте конструктора, который принимает четыре целочисленных параметра, два последних параметра указывают ширину и высоту прямоугольника (а не координаты нижней правой точки, как можно было бы ожидать). На практике об этом часто забывают, и эта забывчивость становится источником досадных и труднообнаруживаемых ошибок.

После того как прямоугольник задан, мы можем прочитать соответствующий фрагмент изображения в объект класса `QImage`. Если фрагмент по каким-то причинам прочитать не удалось, метод `isNull()` объекта `QImage` возвращает значение `true`. Наша следующая задача — преобразовать объект `QImage` в объект `QPixmap`. Это можно сделать с помощью метода `convertFromImage()` класса `QPixmap`. Обратите внимание, что все перечисленные операции выполняются методом `load()` только в том случае, если на момент вызова метода объект `pixmap` класса `QPixmap` не существует (указатель равен 0). Это сделано для того, чтобы не загружать объект `pixmap`, если он уже загружен.

У класса `PictureDrawer` есть также метод `discard()`, который позволяет выгрузить изображение, когда его не нужно показывать. Реализация этого метода очень проста (листинг 9.4).

Листинг 9.4. Метод `discard()`

```
void PictureDrawer::discard()
{
   label->setPixmap(0);
   delete pixmap;
   pixmap = 0;
}
```

Теперь перейдем к классу `ImageFrame`, который, напомню, является потомком класса `QFrame`. В конструкторе этого класса мы должны создать четыре объекта `PictureDrawer` (листинг 9.5).

Листинг 9.5. Конструктор класса `ImageFrame` (вариант 1)

```
ImageFrame::ImageFrame(const QString &fileName) : QFrame()
{
    QImageReader ir(fileName);
    int w = ir.size().width();
    int h = ir.size().height();
    setMinimumWidth(w);
    setMinimumHeight(h);
    setGeometry(0, 0, w, h);
    QGridLayout * layout = new QGridLayout(this);
    layout->setMargin(0);
    layout->setVerticalSpacing(0);
    layout->setHorizontalSpacing(0);
```

```
    setLayout(layout);
    pd1 = new PictureDrawer(this);
    pd1->setGeometry(0, 0, w/2-1, h/2-1);
    pd1->load(fileName, 0, 0, w/2, h/2);
    layout->addWidget(pd1, 0, 0);
    pd2 = new PictureDrawer(this);
    pd2->setGeometry(0, 0, w — w/2, h/2);
    pd2->load(fileName, w/2, 0, w — w/2, h/2);
    layout->addWidget(pd2, 0, 1);
    pd3 = new PictureDrawer(this);
    pd3->setGeometry(0, 0, w/2, h — h/2);
    pd3->load(fileName, 0, h/2, w/2, h — h/2);
    layout->addWidget(pd3, 1, 0);
    pd4 = new PictureDrawer(this);
    pd4->setGeometry(0, 0, w — w/2, h — h/2);
    pd4->load(fileName, w-w/2, h-h/2, w/2, h/2);
    layout->addWidget(pd4, 1, 1);
}
```

В конструкторе `ImageFrame` мы тоже используем объект класса `QImageReader`. Он нужен здесь для того, чтобы определить длину и ширину загружаемого файла в пикселах. Для всех поддерживаемых Qt графических форматов класс `QImageReader` может получить эти данные, не распаковывая сам файл. Затем мы создаем четыре дочерних виджета `PictureDrawer`. Для компоновки виджетов используется менеджер компоновки `QGridLayout`, который позволяет расположить виджеты в виде таблицы. С помощью вызовов

```
layout->setVerticalSpacing(0);
layout->setHorizontalSpacing(0);
```

мы настраиваем менеджер компоновки так, чтобы расстояние между дочерними виджетами было равно нулю. В результате изображение, собранное из четырех частей, будет выглядеть как единое целое.

Деструктор класса `ImageFrame` мы рассматривать не будем, он очевиден, но есть еще один метод, который нужно перекрыть в классе `ImageFrame`, и код которого представляет интерес. Это метод `paintEvent()` (листинг 9.6).

Листинг 9.6. Метод `paintEvent()`

```
void ImageFrame::paintEvent(QPaintEvent * e)
{
    if (e->rect().intersects(QRect(0, 0, width()/2, height()/2))) {
        pd1->update();
    } else pd1->discard();

    if (e->rect().intersects(QRect( width()/2, 0, width() — width()/2,
            height()/2))) {
        pd2->update();
```

```
    } else pd2->discard();
    if (e->rect().intersects(QRect(0, height()/2-10, width()/2,
            height() — height()/2))) {
        pd3->update();
    } else pd3->discard();
    if (e->rect().intersects(QRect( width()-width()/2,
            height()-height()/2, width()/2, height()/2))) {
        pd4->update();
    } else pd4->discard();
    QFrame::paintEvent(e);
}
```

Когда какой-нибудь виджет получает для обработки событие `QPaintEvent`, мы обычно перерисовываем весь виджет потому, что так проще. Но это вовсе не обязательно. У объекта класса `QPaintEvent` есть метод `rect()`, который возвращает объект класса `QRectangle`, т. е. прямоугольник. Этот прямоугольник примерно соответствует той области виджета, которая видима на экране и которую действительно необходимо перерисовать. Я пишу "примерно", потому что область виджета, видимая на экране, может быть вообще непрямоугольной формы, например, если часть виджета закрыта другим виджетом. Вообще, метод `rect()`, что называется, консервативен. Это значит, что область, возвращенная методом `rect()`, всегда включает в себя видимую область, которую необходимо перерисовать, но может быть и больше этой области. Особенно удобно то, что координаты верхней левой точки этого прямоугольника передаются в системе координат виджета (а не экрана).

Мы используем метод `rect()` для определения того, какая часть изображения является видимой в данный момент. У объекта класса `QRectangle`, который возвращает метод `rect()`, есть метод `intersects()`, позволяющий определить, пересекается ли данный прямоугольник с другим прямоугольником, который тоже должен быть задан с помощью объекта `QRectangle`, в системе координат виджета. Мы проверяем, пересекается ли видимый прямоугольник с каждой из четвертей изображения. Если пересекается, мы выводим соответствующее изображение с помощью метода `update()` класса `PictureDrawer` (этот метод, листинг которого мы тут не приводим, вызывает метод `load()`). В противном случае мы вызываем метод `discard()` того же класса `PictureDrawer`.

Итак, наш набор классов, который можно рассматривать как исходную версию виджета, который мы хотим создать, демонстрирует основные принципы вывода изображений по частям. Однако для практического применения этот набор классов пока что неприемлем. Существуют две проблемы, одна из которых затрудняет его применение, а вторая делает и вовсе бессмысленным.

Начнем со второй (тем более, что ее проще решить). Для того чтобы отображать на экране фрагменты изображений, размеры которых в целом слишком велики (для оперативной памяти программы, для конкретной системы и т. д.), недостаточно разбить изображение на четыре части. Дело в том, что при определенных условиях нам все равно придется загружать все изображение целиком (рис. 9.1).

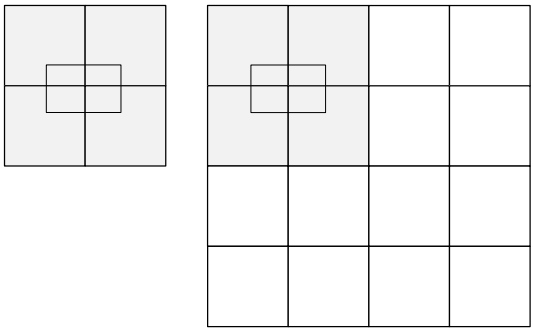


Рис. 9.1. Разбиение изображения на 4 и 16 частей

Квадраты на рисунке соответствуют частям изображения. Тонкий прямоугольник — область экрана, которая перемещается по изображению в процессе прокрутки. Левая часть рисунка демонстрирует, что может произойти, если изображение разбито на четыре части. Если в процессе прокрутки область экрана захватит все четыре части изображения, для показа картинки это изображение придется загружать в память целиком. То есть мы ничего не выиграли. В правой части рисунка показано изображение, разбитое на 16 частей. В этом случае максимум, что может захватить область экрана, — одну четвертую часть изображения. Предполагается, что каждая из частей, на которые мы разбиваем изображение, больше области экрана. Если область экрана по размерам превышает 1/4 изображения, то это изображение, скорее всего, может быть загружено в память целиком и нам не о чем беспокоиться.

Таким образом, первое, что нам следует сделать, — разбить изображение на большее количество частей. Лучше всего сделать это количество произвольным, следуя правилу "ноль–один–бесконечность". Рассмотрим новый вариант конструктора класса `ImageFrame` (листинг 9.7).

Листинг 9.7. Конструктор класса `ImageFrame` (вариант 2)

```
typedef class QList<PictureDrawer *> DrawersList;
const int Division = 4;

ImageFrame::ImageFrame(const QString &fileName) : QFrame()
{
    ir.setDevice(fileName);
    int w = ir.size().width();
    int h = ir.size().height();
    setGeometry(0, 0, w, h);
    setMinimumWidth(w);
    setMinimumHeight(h);
    QGridLayout * layout = new QGridLayout(this);
    layout->setMargin(0);
    layout->setVerticalSpacing(0);
    layout->setHorizontalSpacing(0);
```

```
    setLayout(layout);
    DrawersList * dl = new DrawersList();
    for (int i = 0; i < Division; i++)
        for (int j = 0; j < Division; j++) {
            PictureDrawer * pd = new PictureDrawer(this);
            pd->setGeometry(0, 0, w/Division, h/Division);
            pd->load(fileName, i*w/Division, j*h/Division,
                (i+1)*w/Division, (j+1)*h/Division);
            layout->addWidget(pd, j, i);
            pd->discard();
            dl->append(pd);
        }
    drawersList = dl;
}
```

В этом варианте мы сохраняем указатели на объекты `PicturewDrawer` в списке `DrawersList`, который создан на основе шаблона `QList`. Теперь, как вы понимаете, количество частей, на которые мы разбиваем изображение, может быть произвольным. Для простоты мы делим изображение на одинаковое количество строк и столбцов, задаваемое константой `Division` (иными словами, в приведенном примере изображение будет разделено на 16 частей). Но ничто не мешает нам сделать значение `Division` одним из параметров класса `ImageFrame`, передаваемым, например, в конструкторе. Теперь все действия, связанные с созданием объекта класса `PictureDrawer`, выполняются во вложенном цикле, а каждый объект `PictureDrawer` добавляется в список, указатель на который мы сохраняем в переменной `dl`. Обратите внимание на то, что, создав очередной объект `PictureDrawer`, мы тут же вызываем для него метод `discard()`, чтобы освободить занимаемую им память.

В класс `PictureDrawer` мы добавим метод `drawingRect()`, который будет возвращать объект `QRectangle`, соответствующий прямоугольной области изображения, которую хранит объект `PictureDrawer`. Метод `paintEvent()` класса `ImageFrame` теперь тоже изменится (листинг 9.8).

Листинг 9.8. Новая версия метода `paintEvent()` класса `ImageFrame`

```
void ImageFrame::paintEvent(QPaintEvent * e)
{
    DrawersList * dl = (DrawersList * ) drawersList;
    foreach (PictureDrawer * pd, *dl) {
        if ((e->rect().intersects(pd->drawingRect())))  {
            pd->update();
        } else {
            pd->discard();
        }
    }
    QFrame::paintEvent(e);
}
```

В методе `paintEvent()` мы теперь перебираем элементы списка объектов `PictureDrawer`, и, в зависимости от того, пересекается ли прямоугольник объекта с видимым прямоугольником, вызываем либо метод `update()`, либо метод `discard()`.

Итак, мы решили одну из проблем, связанных с выводом больших изображений: теперь нам действительно не придется загружать всю картинку целиком. Но осталась еще одна проблема: когда в процессе прокрутки прямоугольник видимой части переходит с одной области изображения на другую, нам приходится загружать новую область из файла, хранящегося на диске, причем не просто загружать изображения, а загружать его фрагмент с помощью объекта `QImageReader`. Хотя современные диски работают очень быстро, эта операция все равно способна затормозить прокрутку.

Задачи повышения быстродействия кода относятся, по-моему, к одному из самых интересных типов задач. Первоначальная идея, как правило, проста и очевидна. Различные сложности и нетривиальные решения появляются тогда, когда возникает необходимость повысить производительность прототипа кода.

Что можно сделать для ускорения работы нашего виджета просмотра больших изображений? Во-первых, попробуем ускорить работу с диском. Обычно большие изображения загружаются из файлов, которые используют сжатие. Иногда сжатый файл занимает в сотни раз меньше места, чем распакованное изображение. Я не шучу. Для тестирования этого примера я использовал квадратное изображение со стороной квадрата, равной 28 000 пикселов. Исходный файл был сохранен в формате JPEG (с потерями, разумеется) и занимал около 28 мегабайт, тогда как распакованное изображение занимало примерно 3,2 гигабайта (файл содержал спутниковый снимок африканского континента и прибрежных вод, т. е. довольно однородное изображение, чем и объясняется высокая степень сжатия). При таком высоком коэффициенте сжатия мы можем просто загрузить файл в оперативную память и таким образом ускорить все операции, выполняемые с ним.

Ранее мы рассматривали отображение файлов в память, однако сейчас мы воспользуемся другим способом. При отображении файла в память мы получали указатель на блок памяти. Но простой блок памяти нельзя передать объекту `QImageReader`, который играет в загрузке изображений очень важную роль. Нам нужен класс, который снабдил бы блок памяти интерфейсом класса `QIODevice`. Этот интерфейс наследуют все классы, с которыми можно работать как с классом `QFile` (который тоже его наследует).

Такой класс у нас есть. Это класс `QBuffer`. Объект класса `QBuffer` можно рассматривать как файл, хранящийся в памяти. Для хранения данных `QBuffer` использует объект `QByteArray`. Массив `QByteArray` может принадлежать объекту `QBuffer`, а может быть и внешним, и тогда мы можем получить к нему доступ. Итак, наше первое улучшение работы класса `ImageFrame` заключается в том, что исходный файл мы загружаем в объект `QBuffer`. Поскольку этот объект является объектом класса-потомка `QIODevice`, мы можем передать указатель на него объекту `QImageReader` (с помощью метода `setDevice()` класса `QImageReader`). В результате объект `QImageReader` будет работать с объектом `QBuffer` точно так же, как он работал бы

с файлом, хранящимся на диске. Рассмотрим фрагмент конструктора класса `ImageFrame`, который использует объект `QBuffer` для работы с файлом, загруженным в память (листинг 9.9).

Листинг 9.9. Конструктор класса `ImageFrame` (вариант 3)

```
QFile f(fileName);
f.open(QIODevice::ReadOnly);
QByteArray * ba = new QByteArray(f.readAll());
io = new QBuffer(ba);
f.close();
io->open(QIODevice::ReadOnly);
ir.setDevice(io);
int w = ir.size().width();
int h = ir.size().height();
...
delete ba;
```

В этом фрагменте переменная `io` имеет тип `QBuffer *`, а переменная `ir` — тип `QimageReader`. Поскольку конструктору класса `ImageFrame` передается имя файла, мы сначала считываем содержимое этого файла в переменную `ba` типа `QByteArray`, а затем создаем для этой переменной "обертку" `io` класса `QBuffer`. Теперь объект `ir` может работать с содержимым `ba` как с файлом.

Примечание

В этом примере класс `QBuffer` реализует паттерн "декоратор" для класса `QByteArray`, позволяющий последнему функционировать как устройству ввода/вывода.

Дальнейшая оптимизация напрашивается сама собой. Если файл изображения уже загружен в память и уже создан объект `QBuffer`, который позволяет получить доступ к данным файла, загруженным в память, то зачем загружать этот файл заново в каждом объекте `PictureDrawer`? Не лучше ли вместо имени файла передавать в методе `load()` объекта `PictureDrawer` указатель на объект `QBuffer`? Действительно, так будет лучше (листинг 9.10).

Листинг 9.10. Метод `load()` (вариант 2)

```
DrawersList * dl = new DrawersList();
    for (int i = 0; i < Division; i++)
        for (int j = 0; j < Division; j++) {
            PictureDrawer * pd = new PictureDrawer(this);
            pd->setGeometry(0, 0, w/Division, h/Division);
            pd->load(io, i*w/Division, j*h/Division, (i+1)*w/Division,
                (j+1)*h/Division);
            layout->addWidget(pd, j, i);
            dl->append(pd);
        }
```

Вы можете спросить, почему мы передаем указатель на объект QBuffer, а не на объект QImageReader? Дело в том, что вследствие каких-то внутренних ограничений для загрузки каждой части файла приходится создавать собственный объект класса QImageReader. Посмотрим, как изменился метод load() класса PictureDrawer (листинг 9.11).

Листинг 9.11. Метод load() (вариант 3)

```
void PictureDrawer::load(QIODevice *ioDevice, const int x1, int y1,
    int x2, int y2)
{
    io = ioDevice;
    io->reset();
    QImageReader ir(io);
    ir.setClipRect(QRect(x1, y1, x2 — x1, y2 — y1));
       QImage img = ir.read();
...
}
```

Далее с объектом img можно работать, как и в предыдущей версии метода load(), т. е., конвертировать его в объект QPixmap.

Если мы проверим быстродействие новых версий классов PictureDrawer и ImageFrame, то обнаружим, что прирост скорости при прокрутке между двумя фрагментами изображения незначителен. Очевидно, что выполненных действий по оптимизации недостаточно. Узким местом является метод load(), в котором каждый раз нужный фрагмент изображения извлекается из файла с помощью объекта QImageReader. И тут мы опять можем воспользоваться тем фактом, что в сжатом виде изображение занимает гораздо меньше памяти, чем в распакованном. Что же мы можем сделать? Вместо того, чтобы каждый раз считывать фрагмент изображения с помощью объекта QImageReader и затем преобразовывать полученный объект QImage в QPixmap, мы можем считать каждый фрагмент изображения один раз и затем сохранить его в формате со сжатием без потерь (формат PNG хорошо подходит для этого). Сжатый фрагмент изображения можно хранить в оперативной памяти (в массиве QByteArray) или, если он слишком велик, на диске. В любом случае, когда нам понадобится загрузить этот фрагмент для показа, мы сможем прочитать его непосредственно в объект QPixmap (либо из файла, либо из памяти). В результате скорость загрузки нового фрагмента изображения при прокрутке ощутимо возрастет. Посмотрим теперь на окончательную версию класса PictureDrawer (листинг 9.12).

Листинг 9.12. Окончательная версия класса PictureDrawer

```
class PictureDrawer : public QWidget
{
    Q_OBJECT
```

```
public:
    explicit PictureDrawer(QWidget *parent = 0);
    ~PictureDrawer();
    void load(QIODevice * ioDevice, const int x1, int y1,
        int x2, int y2);
    void update();
    void discard();
    void setPersistence(int p);
    void optimizeForSpeed(bool b);
    QRect drawingRect();
private:
    QPixmap * pixmap;
    int _x1, _y1, _x2, _y2;
    QIODevice * io;
    QLabel * label;
    QByteArray * chank;
    int resource;
    int persistence;
    bool noDiscard;
};
```

Переменная `chank` предназначена для хранения сжатого фрагмента изображения, за вывод которого отвечает класс `PictureDrawer` (мы выбрали вариант, при котором сжатый фрагмент хранится в памяти). Смысл остальных переменных станет понятен после обсуждения реализации методов (листинг 9.13).

Листинг 9.13. Реализация методов класса `PictureDrawer`

```
PictureDrawer::PictureDrawer(QWidget *parent) :
    QWidget(parent)
{
    pixmap = 0;
    label = new QLabel(this);
    resource = 0;
    persistence = 0;
    io = 0;
    noDiscard = false;
    chank = new QByteArray();
}

PictureDrawer::~PictureDrawer()
{
    delete pixmap;
    delete chank;
}
```

```
void PictureDrawer::setPersistence(int p)
{
    persistence = p;
}

void PictureDrawer::optimizeForSpeed(bool b)
{
    noDiscard = b;
}

void PictureDrawer::load(QIODevice *ioDevice, const int x1, int y1,
    int x2, int y2)
{
    _x1 = x1;
    _y1 = y1;
    _x2 = x2;
    _y2 = y2;
    io = ioDevice;
    io->reset();
    QImageReader ir(io);
    ir.setClipRect(QRect(_x1, _y1, _x2 - _x1, _y2 - _y1));
    QImage img = ir.read();
    if (img.isNull())
        qWarning() << "Image is NULL" << QRect(_x1, _y1, _x2 - _x1,
           _y2 - _y1);
    QIODevice * chankIntf = new QBuffer(chank, this);
    chankIntf->open(QIODevice::WriteOnly);
    if (!img.save(chankIntf, "PNG"))
        qWarning() << "Failed saving image"
            << QRect(_x1, _y1, _x2 - _x1, _y2 - _y1);
    delete chankIntf;
}

QRect PictureDrawer::drawingRect()
{
    return QRect(_x1, _y1, _x2 - _x1, _y2 - _y1);
}

void PictureDrawer::discard()
{
    if (noDiscard) return;
    if(resource > 0) resource--;
    if(resource == 0) {
        label->setPixmap(0);
        delete pixmap;
        pixmap = 0;
    }
}
```

```
void PictureDrawer::update() {
    resource = persistence;
    if (io&&(!pixmap)) {
        pixmap = new QPixmap();
        if (!pixmap->loadFromData(*chank, "PNG"))
            qWarning() << "Failed drawing image"
                << QRect(_x1, _y1, _x2 — _x1, _y2 — _y1);
        label->setPixmap(*pixmap);
        label->setMinimumWidth(pixmap->width());
        label->setMinimumHeight(pixmap->height());
    }
}
```

Сначала рассмотрим метод `load()`. Этот метод сохраняет переданные ему параметры `x1`, `y1`, `x2`, `y2` в полях `_x1`, `_y1`, `_x2` и `_y2` соответственно. Затем метод выполняет поиск заданного ему фрагмента в исходном файле (поскольку теперь исходный файл загружен в память, объект класса `QImageReader` работает с объектом `QBuffer`, указатель на который передается методу в параметре `ioDevice`). Далее метод `load()` ведет себя не так, как в предыдущих версиях. Вместо того чтобы преобразовать объект `img` в объект класса `QPixmap`, мы сохраняем содержимое объекта `img` в объект `chank` в формате PNG. Для того чтобы объект `img` класса `QImage` мог работать с объектом `chank` класса `QByteArray` как с файлом, используется вспомогательный объект `chankIntf` класса `QBuffer`. Обратите внимание на то, что в конструкторе класса `QBuffer`, создающем объект `chankIntf`, мы указываем объект `QByteArray`, который и будет хранить записанные данные. Теперь объект `chank` содержит фрагмент изображения в формате PNG, а объект `chankIntf` можно удалить, т. к. он больше не нужен. В принципе, указатель на объект класса `QIODevice`, который мы передаем в параметре `ioDevice` и присваиваем полю `io`, тоже не нужен нигде за пределами метода `load()`. Мы используем поле `io` в методе `update()` просто как флаг, указывающий на то, что метод `load()` уже был вызван (и значит, объект `chank` уже содержит фрагмент изображения).

Рассмотрим теперь метод `update()`, который вызывается объектом `ImageFrame`, когда требуется перерисовать часть изображения. Этот метод загружает изображение из объекта `chank` в объект `pixmap`, в процессе чего оно распаковывается. Поскольку сжатый фрагмент изображения хранится в памяти, мы используем метод `loadFromData()` класса `QPixmap`.

Метод `discard()` тоже претерпел некоторые изменения по сравнению с предыдущими версиями. Эти изменения также связаны с нашим стремлением ускорить прокрутку. Для этой цели были введены три режима работы класса `PictureDrawer`. Первый режим, "скоростной", обеспечивает максимальную скорость прокрутки. Он устанавливается, если присвоить полю `noDiscard` значение `true`. В этом случае метод `discard()` не уничтожает объект `pixmap`, а метод `update()`, соответственно, не загружает его заново. Прокрутка в этом режиме выполняется с максимальной ско-

ростью, но применение этого режима возможно, разумеется, только в том случае, если все изображение умещается в памяти.

Далее следует компромиссный режим, при котором невидимый фрагмент изображения уничтожается, но не сразу, а только после генерации некоторого количества событий `QPaintEvent`, при которых данный фрагмент невидим. Обратите внимание на переменную-счетчик `resource`. Если полю `noDiscard` присвоено значение `false` и значение счетчика `resource` больше нуля, метод `discard()` не удаляет объект `pixmap`, а только уменьшает значение счетчика. Метод `update()`, вызванный для данного фрагмента изображения, увеличивает значение счетчика до определенной величины, заданной значением `persistence`. Благодаря этому фрагмент изображения не удаляется сразу, как только перестает быть видимым. В результате удается значительно повысить производительность при прокрутке, когда пользователь, загрузив новую порцию изображения, хочет вернуться назад. Разумеется, этот вариант требует дополнительной памяти по сравнению с самым экономичным вариантом, поскольку при его использовании некоторые невидимые фрагменты изображения остаются в памяти.

Наконец третий режим, максимально экономящий память, работает так, как и было описано изначально (невидимые фрагменты, точнее, их распакованные варианты, удаляются). Для того чтобы установить этот режим, необходимо переменной `noDiscard` присвоить значение `false`, а переменной `persistence` — значение \ 0.

Класс `ImageFrame` теперь тоже выглядит немного иначе (листинг 9.14).

Листинг 9.14. Класс `ImageFrame`

```
class ImageFrame : public QFrame
{
public:
    typedef enum {
        OptimizeForMemory = 0,
        OptimizeForSpeed,
        TradeoffOptimization
    } Optimizations;
    ImageFrame(const QString &fileName,
        Optimizations opt = ImageFrame::OptimizeForMemory);
    ~ImageFrame();
protected:
    void paintEvent(QPaintEvent * e);
private:
    void optimizePictureDrawer(PictureDrawer * pd,
        ImageFrame::Optimizations op);
    void * drawersList;
};
```

Мы ввели новый тип данных — `Optimizations`. Значения этого типа соответствуют трем режимам, описанным ранее. Режим `OptimizeForMemory` экономит память

за счет скорости прокрутки. Режим `OptimizeForSpeed` — антипод режима `OptimizeForMemory`, а режим `TradeoffOptimization` — тот самый режим, при котором невидимые фрагменты изображения не уничтожаются сразу. Метод `optimizePictureDrawer()` настраивает параметры оптимизации объектов `PictureDrawer` в соответствии с выбранным режимом. Из методов класса `ImageFrame` мы рассмотрим только два: конструктор и метод `optimizePictureDrawer()` (листинг 9.15).

Листинг 9.15. Методы класса `ImageFrame`

```
ImageFrame::ImageFrame(const QString &fileName, Optimizations opt)
    : QFrame()
{
    QFile f(fileName);
    f.open(QIODevice::ReadOnly);
    QImageReader ir;
    QByteArray ba = f.readAll();
    QBuffer io(&ba);
    f.close();
    io.open(QIODevice::ReadOnly);
    ir.setDevice(&io);
    int w = ir.size().width();
    int h = ir.size().height();
    setGeometry(0, 0, w, h);
    setMinimumWidth(w);
    setMinimumHeight(h);
    QGridLayout * layout = new QGridLayout(this);
    layout->setMargin(0);
    layout->setVerticalSpacing(0);
    layout->setHorizontalSpacing(0);
    setLayout(layout);
    DrawersList * dl = new DrawersList();
    for (int i = 0; i < Division; i++)
        for (int j = 0; j < Division; j++) {
            PictureDrawer * pd = new PictureDrawer(this);
            pd->setGeometry(0, 0, w/Division, h/Division);
            pd->load(&io, i*w/Division, j*h/Division,
                (i+1)*w/Division, (j+1)*h/Division);
            layout->addWidget(pd, j, i);
            optimizePictureDrawer(pd, opt);
            dl->append(pd);
        }
    drawersList = dl;
}

void ImageFrame::optimizePictureDrawer(PictureDrawer *pd,
    ImageFrame::Optimizations op)
```

```
{
    switch(op) {
    case ImageFrame::OptimizeForMemory:
        pd->optimizeForSpeed(false);
        pd->setPersistence(0);
        break;
    case ImageFrame::OptimizeForSpeed:
        pd->optimizeForSpeed(true);
        pd->setPersistence(0);
        break;
    case ImageFrame::TradeoffOptimization:
        pd->optimizeForSpeed(false);
        pd->setPersistence(32);
        break;
    default:
        break;
    }
}
```

Начнем с конструктора. Обратите внимание, что поскольку теперь метод `update()` класса `PictureDrawer` считывает фрагмент изображения не из исходного файла, а из своей сжатой копии, то и хранить данные исходного файла на протяжении всей жизни класса `ImageFrame` не нужно. Соответствующий объект (`ba`) объявлен как локальная переменная конструктора. То же самое касается и объектов классов `QImageReader` и `QBuffer`. Адрес объекта `io` класса `QBuffer` передается методам `load()` объектов `PictureDrawer` и, как уже было отмечено, используется за пределами этих методов только в качестве флага, так что сам объект может быть уничтожен `io` при выходе из конструктора. Метод `optimizePictureDrawer()` устанавливает значения полей объекта `PictureDrawer` в соответствии с выбранным типом оптимизации. Как именно различные значения полей влияют на поведение фрагмента изображения при прокрутке, было описано ранее.

Программу, которая демонстрирует работу описанных классов, вы найдете в папке Ch9/bigimages на сайте **http://symmetrica.net/qt47book/**. Там же расположено тестовое изображение bigimg.jpg.

Наша программа (рис. 9.2) занимает в памяти несколько сотен (200–400) мегабайт при работе с изображением, которое в полностью "распакованном" виде занимает более 2 гигабайт. При этом прокрутка выполняется довольно быстро. Если немного пожертвовать качеством и заменить формат сжатия фрагментов изображения с PNG на JPEG, прокрутка будет работать еще быстрее. Другой вариант, сокращающий место, занимаемое программой в памяти, но замедляющий прокрутку, — хранение фрагментов изображения в файлах на диске. Хотелось бы обратить ваше внимание лишь на тот факт, что скорость загрузки изображения замедляется, причем нелинейно, с увеличением размеров изображения. Но тут, видимо, ничего сделать нельзя. Размер все-таки имеет значение.

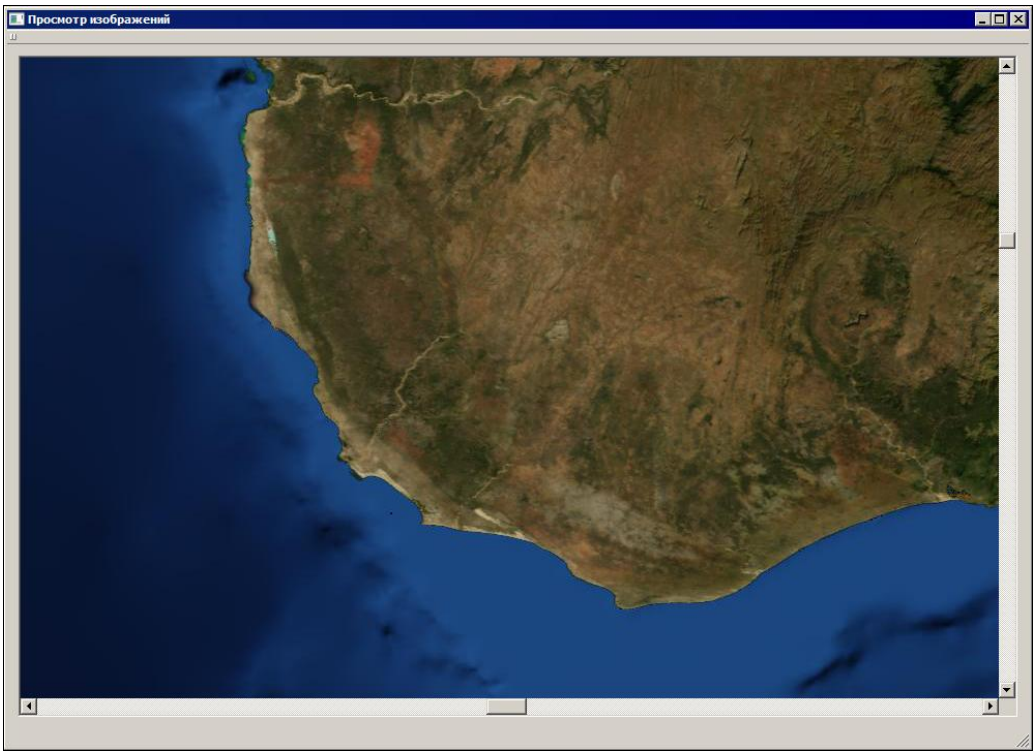


Рис. 9.2. Южная оконечность острова Мадагаскар, общий размер изображения — 1 гигапиксел

Текст с элементами форматирования

Основой для работы с текстовыми документами в системе вывода текста Qt 4, получившей название Scribe, служит класс `QTextDocument`. Объект этого класса хранит всю информацию о структуре форматированного документа, а также предоставляет функции для его редактирования. Мы можем использовать виджеты `QTextEdit` для ручного редактирования текста, содержащегося в `QTextDocument`, и класс `QTextBrowser` для просмотра и навигации по гиперссылкам. Таким образом, здесь еще раз продемонстрирован принцип модульности Qt. Модель документа, определяемая объектами класса `QTextModel`, отделена от средств представления документа, определяемых объектами классов `QTextDocument` и `QTextBrowser`.

Классы *QTextDocument* и *QTextEdit*

Класс `QTextDocument` предназначен для хранения и обработки текста, содержащего элементы оформления (обогащенного текста). В англоязычной документации такой текст называется rich text, но следует сразу отметить, что формат текста в `QTextDocument` не имеет отношения ни к принятому в Microsoft Rich Text Format, ни к Open Document Format, используемому в OpenOffice.org. Это собственный формат Qt.

Структура обогащенного текста в `QTextDocument` имеет вид дерева. Корневым узлом дерева является корневой фрейм, содержащий текстовые блоки, таблицы, а также дочерние фреймы, которые, в свою очередь, могут содержать текстовые блоки таблицы и дочерние фреймы.

Каждому структурному элементу обогащенного текста соответствует свой класс. Фрейму соответствует класс `QTextFrame` (получить доступ к корневому фрейму можно с помощью метода `rootFrame()`). Метод `childFrames()` класса `QTextFrame` возвращает список дочерних фреймов. Блок текста представлен классом `QTextBlock` (обычно в отдельный блок выделяется текст, принадлежащий одному фрейму и имеющий однородное форматирование). Интересной особенностью класса `QTextDocument` является то, что хотя во внутренней структуре документа текстовый блок является дочерним элементом фрейма, при работе с объектом `QTextDocument` обогащенный текст можно рассматривать как цепочку текстовых блоков, представленных объектами `QTextBlock`. Доступ к первому текстовому блоку документа можно получить с помощью метода `begin()` класса `QTextDocument`, а доступ к последнему — с помощью метода `end()`. У класса `QTextBlock` есть методы `next()` и `previous()`, с помощью которых можно получить доступ, соответственно, к следующему и предыдущему текстовым блокам документа.

Таблицы представлены объектами класса `QTextTable`. Каждая ячейка таблицы представлена объектом класса `QTextTableCell`. С точки зрения структуры документа каждой ячейке соответствует один фрейм, содержащий один или несколько текстовых блоков.

Графические изображения представлены в структуре документа `QTextDocument` специальным символом-"якорем", дополнительные атрибуты формата которого содержат сведения о файле изображения. Вообще говоря, источником изображений для документа `QTextDocument` служат внешние графические файлы, как в классическом HTML. Документ `QTextDocument` можно сохранить в формате ODF, который способен инкапсулировать изображения, но загрузить документ из этого формата нельзя. Тем не менее когда документ открыт, все содержащиеся в нем изображения копируются в специальный кэш, из которого их можно извлечь. Далее вы увидите, как это делается.

Для специальных типов форматирования предусмотрены специализированные классы. Например, класс `QTextList` представляет маркированный список (группа абзацев, каждый из которых начинается с цифры или специального значка "жирной точки"). С точки зрения интерфейса программиста содержимое объекта `QTextList` является последовательностью текстовых блоков, представленных объектами `QTextBlock`.

Любопытно отметить, что, хотя в документации к Qt разработчики советуют использовать повсеместно объекты `QTextDocument` (а не `QString`) для хранения текста, у класса `QTextEdit` нет конструктора, которому можно было бы передать ссылку на объект `QTextDocument`, а конструктор со ссылкой на объект `QString` имеется. Для того чтобы назначить объекту класса `QTextEdit` объект класса `QTextDocument`, необходимо вызвать метод `setDocument()`.

Важную роль в редактировании содержимого `QTextDocument` играет класс `QTextCursor`. Объекты `QTextCursor` используются как для указания текущей позиции в документе, так и для обозначения выбранных фрагментов текста. Кроме того, объекты `QTextCursor` предоставляют в распоряжение программиста ряд методов, предназначенных для редактирования текста и изменения форматирования, начиная с выбранной позиции.

Допустим, что в окне редактора `QTextEdit` мы хотим выполнить проверку орфографии так, чтобы слова, отсутствующие в словаре, подчеркивались красной волнистой линией, как это принято в большинстве редакторов. Для проверки орфографии мы воспользуемся системой Aspell. Эта система распространяется на условиях лицензии GNU и первоначально была ориентирована на UNIX-системы, а позднее портирована и под Windows. Если вы пишете программы для Linux, вам надо убедиться в наличии пакетов aspell и aspell-devel (в вашем дистрибутиве аналогичные пакеты могут называться по-другому) и, если их нет, установить их из репозитория вашего дистрибутива. Установите также орфографический словарь Aspell для русского языка, и можете считать, что все, что нужно для работы данного примера, у вас есть. Если вы пишете программы для Windows, все становится немного сложнее.

Установка aspell под Windows

Прежде всего, загрузите файл Aspell-0-50-3-3-Setup.exe с сайта **ftp.gnu.org/gnu/aspell/w32/** и установите его. Этот дистрибутив содержит библиотеку aspell-15.dll, которая нам нужна, и несколько двоичных файлов, которые нам не требуются. Загрузите и установите орфографический словарь русского языка Aspell-ru-0.50-2-3.exe с того же самого сайта. Для разработки собственных программ, использующих Aspell, нам понадобится файл aspell-dev-0-50-3-3.zip с того же сайта. Этот файл содержит заголовочные файлы и файлы библиотек (lib) для компоновки с приложениями, написанными в среде Microsoft Visual Studio. Если вы разрабатываете приложения именно в этой среде, то все, что вам нужно, у вас уже есть. Если вы пишете программы в среде Qt SDK, которая использует MinGW, то вам нужно создать файл с расширением a для библиотеки aspell-15.dll. Для этого требуется внести небольшие изменения в файл aspell-15.def, который входит в состав архива aspell-dev-0-50-3-3.zip. Будучи извлечен из архива, этот файл выглядит так (листинг 9.16).

Листинг 9.16. Исходный файл aspell-15.def

```
EXPORTS
_ZN24aspeller_default_suggest11SuggestImpl5scoreEPKcS2_ @1
_ZN24aspeller_default_suggest11SuggestImpl7suggestEPKc @2
    _ZN24aspeller_default_suggest11SuggestImpl8set\\
        _modeEN7acommon10ParmStringE @3
_ZN24aspeller_default_suggest11SuggestImplD0Ev @4
...
```

Здесь представлено только начало этого довольно большого файла. Измените его так, как показано в листинге 9.17.

Листинг 9.17. Модифицированный файл aspell-15.def

```
LIBRARY aspell-15.dll
EXPORTS
_ZN24aspeller_default_suggest11SuggestImpl5scoreEPKcS2_ @1
_ZN24aspeller_default_suggest11SuggestImpl7suggestEPKc @2
    _ZN24aspeller_default_suggest11SuggestImpl8set\\
        _modeEN7acommon10ParmStringE @3
_ZN24aspeller_default_suggest11SuggestImplD0Ev @4
...
```

Теперь откройте консоль Windows, перейдите в директорию MinGW/bin/, расположенную в корневой директории Qt SDK, и выдайте команду:

```
dlltool.exe  -d aspell-15.def -l libaspell.a
```

При этом, разумеется, нужно указать полный путь к файлу aspell-15.def в вашей системе, и полный путь к той директории (разумнее всего выбрать MinGW/lib/), в которой вы хотите сохранить файл libaspell.a. Теперь вы тоже готовы к сборке.

Завершая обзор конфигурации Aspell под Windows, следует сказать, что, в отличие от Linux, где для установки словарей существуют стандартные директории, в Windows словари Aspell устанавливаются там, где указывает пользователь во время установки. Проблема в том, что по умолчанию библиотека aspell-15.dll использует смесь из абсолютных и относительных ссылок на файлы словарей, так что для того, чтобы ваша программа могла найти их, проще всего установить программу в той же директории, где установлена система Aspell. Например, C:\Program Files\Aspell\bin\. Альтернативный вариант, требующий пересборки библиотеки aspell-15.dll под MinGW, мы рассматривать не будем.

Подготовка программы

Теперь, независимо от платформы, в файл проекта Qt необходимо добавить строку:

```
LIBS += -laspell
```

Таким образом мы сообщаем проекту, что к приложению надо подключить нестандартную (с точки зрения Qt) библиотеку.

Подключение сторонней библиотеки влечет за собой проблему, с которой мы раньше не сталкивались. Сколь бы широки ни были возможности Qt, рано или поздно вашей программе понадобится делать то, что Qt делать не умеет. Принцип повторного использования кода подсказывает, что в таких случаях следует, прежде всего, поискать решение, созданное другими программистами.

Примечание

Многие начинающие программисты гордятся тем, что сами написали весь код своей программы "с нуля". Такие утверждения не только не соответствуют действитель-

ности, (ведь значительная часть кода все равно не контролируется программистом), но и методологически неверны. Нет смысла в изобретении велосипедов. Квалификация программиста определяется его способностью придумывать новое, а не воспроизводить то, что кто-то уже сделал, и сделал хорошо.

Значительная часть сторонних библиотек экспортирует свои интерфейсы в формате, совместимом с языком C, поскольку этот формат наиболее прост и поддерживается наибольшим количеством сред программирования. Такие библиотеки можно использовать не только в программах, написанных на C и C++, но также на Delphi и многих интерпретируемых языках программирования. Так что если вы создаете собственную библиотеку, рассчитанную на широкое использование, то имеет смысл подумать о том, чтобы экспортировать интерфейс библиотеки в формате C (сама библиотека может быть написана при этом на C++, в том числе с использованием Qt).

Библиотека Aspell экспортирует интерфейс в формате C. По сути, этот интерфейс представляет собой набор функций и структур данных. Интерфейс Aspell выглядит громоздко сам по себе и совершенно не соответствует принципам программирования, принятым в Qt. Как лучше всего поступить в подобной ситуации? Лучше всего в подобной ситуации создать класс C++, в котором инкапсулировать интерфейс внешней библиотеки, причем "выставить наружу" только те части интерфейса, которые действительно нужны нашему приложению. Такая архитектура соответствует паттерну проектирования "Фасад".

Класс `SpellChecker`, который инкапсулирует интерфейс Aspell в нашей программе, предоставляет программисту всего четыре публичных метода (листинг 9.18), не считая конструктора и деструктора (сравните это с несколькими десятками функций интерфейса Aspell).

Листинг 9.18. Класс `SpellChecker`

```
class SpellChecker
{
public:
    SpellChecker(QTextEdit *textEdit);
    ~SpellChecker();
    void unSpellCheck();
    bool spellCheck();
    void checkWord();
    QStringList suggestions();
private:
    void _checkWord(QTextCursor *cursor);
    QTextEdit *m_textEdit;
    QRegExp *m_regExp;
    QString m_lang1;
    QString m_lang2;
    StringMap *m_map;
    AspellConfig *spell_config1;
```

```
    AspellConfig *spell_config2;
    AspellSpeller *spell_checker1;
    AspellSpeller *spell_checker2;
};
```

Глядя на этот класс, нельзя не заметить, что он очень тесно связан с виджетом `QTextEdit`, указатель на который передается ему в качестве параметра конструктора. Таким образом, этот класс реализует еще и паттерн "декоратор" для класса `QTextEdit`.

Метод `spellCheck()` проверяет орфографию в тексте документа, который содержит объект `QTextEdit`, с которым работает данный объект `SpellChecker`. При этом слова, которых нет в орфографическом словаре, подчеркиваются волнистой красной чертой. Метод `checkWord()` проверяет слово, находящееся в данный момент под курсором `QtextEdit`, и, в случае необходимости, тоже подчеркивает его красной волнистой линией. Метод `unSpellCheck()` удаляет признаки проверки орфографии (подчеркивания нераспознанных слов) из всего документа. Наконец, метод `suggestions()` выдает список вариантов написания слова, находящегося под курсором `QTextEdit`, предложенный орфографическим словарем Aspell. Обратите внимание, что у всех методов класса `SpellChecker`, кроме конструктора, нет параметров. Таким образом, мы действительно существенно упростили интерфейс блока проверки орфографии.

> ***Примечание***
>
> Классы-декораторы не только упрощают интерфейсы. С их помощью можно решать и другие задачи. Предположим, что интерфейс Aspell различался бы на платформах Windows и Linux (такое иногда встречается). В этом случае все различия можно было бы "спрятать" внутри класса `SpellChecker`. Остальной код программы был бы независим от платформы. Кроме того, если в будущем вы захотите сменить Aspell на какую-то другую систему проверки орфографии, все, что вам понадобится сделать, — заменить "начинку" класса `SpellChecker`.

Мы рассмотрим реализацию только некоторых методов класса `SpellChecker`, а именно тех, которые имеют прямое отношение к классам `QTextEdit` и `QTextDocument` (листинг 9.19).

Листинг 9.19. Методы класса `SpellChecker`

```
bool SpellChecker::spellCheck()
{
   if ((spell_checker1 == 0) || (spell_checker2 == 0)) {
      QMessageBox::critical(0, QString::fromUtf8("Ошибка"),
         QString::fromUtf8("Словарь не найден"));
      return false;
   }
   QTextCursor cursor(m_textEdit->document());
   while (!cursor.isNull() && !cursor.atEnd()) {
      _checkWord(&cursor);
```

```
      QTextCursor oldc = cursor;
      if (!cursor.movePosition(QTextCursor::NextWord,
                               QTextCursor::MoveAnchor))
         break;
      int oldpos = oldc.position();
      int newpos = cursor.position();
      if (abs(newpos — oldpos) < 3)
         cursor.setPosition(newpos + 1);
   }
   if (!cursor.isNull())
      _checkWord(&cursor);
   return true;
}

void SpellChecker::_checkWord(QTextCursor *cursor)
{
   cursor->select(QTextCursor::WordUnderCursor);
   QString selText = cursor->selectedText();
   static const QRegExp nonDigits("\\D");
   if (!selText.contains(nonDigits))
      return;
   selText = selText.remove(QString::fromUtf8("""));
   selText = selText.remove(QString::fromUtf8("""));
   QByteArray ba = selText.toUtf8();
   if ((aspell_speller_check(spell_checker1, ba.data(), ba.size()) == 0)
   && (aspell_speller_check(spell_checker2, ba.data(), ba.size()) == 0))
   {
      QTextCharFormat fmt = cursor->charFormat();
      fmt.setUnderlineColor(QColor(Qt::red));
      fmt.setUnderlineStyle(QTextCharFormat::SpellCheckUnderline);
      cursor->setCharFormat(fmt);
   } else {
      QTextCharFormat fmt = cursor->charFormat();
      fmt.setUnderlineStyle(QTextCharFormat::NoUnderline);
      cursor->setCharFormat(fmt);
   }
   cursor->clearSelection();
}

void SpellChecker::checkWord()
{
   if ((spell_checker1 == 0) || (spell_checker2 == 0))
      return;
   QTextCursor cursor = m_textEdit->textCursor();
   _checkWord(&cursor);
}
```

Метод `_checkWord()` — "рабочая лошадка" класса `SpellChecker`. Этот метод выполняет проверку орфографии для того слова, на которое указывает курсор `QTextCursor`. В соответствии с полиморфной структурой модели документа, хранящейся в объекте класса `QTextDocument`, курсор позволяет выделить слово, строку, текстовый блок или документ в целом, к которым он принадлежит. Нам нужно выделить слово, на которое указывает курсор. Для этого мы вызываем метод `select()` объекта `cursor` и передаем ему константу `QTextCursor::WordUnderCursor`. Выделение не обязательно отражается визуально. Интересен вопрос о том, что курсор `QTextCursor` считает отдельным словом. В качестве разделителей между словами воспринимаются пробелы, символы переноса строки, символы знаков препинания и европейские кавычки. Если курсор указывает на последовательность символов, которая не воспринимается как слово, метод `selectedText()` вернет пустую строку. Стоит отметить, что по умолчанию `QTextCursor` не считает специальными знаками парные кавычки "«" и "»", распространенные в отечественной типографии. Это значит, что если проверяемое слово заключено в кавычки, то кавычки попадут в выделенный текст вместе с проверяемым словом. Чтобы это не произошло, мы удаляем парные кавычки с помощью метода `remove()` класса `QString`. Если парных кавычек в выделенном тексте нет, метод `remove()` не вносит в текст никаких изменений. Дальше с помощью функции системы Aspell мы проверяем, находится ли слово, которое содержит строка `selText`, в орфографическом словаре. Обратите внимание на кодировку текста. Aspell поддерживает работу со специализированными русскоязычными кодировками, например KOI8-R. Кодировку, с которой будет работать Aspell, можно настроить (в нашем примере это делается в конструкторе класса `SpellChecker`), и мы, следуя принципу кросс-платформенности, выбираем универсальную UTF-8.

Вернемся к проверке орфографии. Если слово в орфографическом словаре не встречается, то его надо подчеркнуть. Метод `charFormat()` позволяет получить текущий формат символов слова (параметры шрифта, данные о цвете, и т. п.). Этот формат сохраняется в объекте класса `QTextCharFormat`. Затем с помощью метода `setUnderlineColor()` (определяющего цвет подчеркивания) и метода `setUnderlineStyle()` (определяющего вид подчеркивающей линии) мы добавляем к формату подчеркивание в виде волнистой красной линии. После этого остается вызвать метод `setCharFormat()` класса `QTextCursor`, задать для выделенного слова новый формат и снять выделение. Обратите внимание, что если проверяемое слово присутствует в словаре, мы выполняем обратную операцию, т. е. убираем подчеркивание (если оно есть). Для чего это нужно? Допустим, что некоторое слово было напечатано пользователем с ошибкой, и наша функция подчеркнула его волнистой красной чертой. Пользователь заметил это и исправил слово. Если мы не снимем прежний формат, исправленное слово так и останется подчеркнутым.

Метод `checkWord()`, который используется для проверки слова, находящегося под курсором, вызывает метод `_checkWord()` напрямую, передавая ему текущий курсор. Этот метод требуется для проверки орфографии слова непосредственно в процессе редактирования документа, поэтому его следует вызывать всякий раз, когда содержимое документа меняется. При этом возникает одна проблема, которую мы обсудим позже.

А можно ли с помощью метода `_checkWord()` организовать проверку орфографии во всем документе? Это делает метод `spellCheck()`. В начале мы проверяем, удалось ли интерфейсу Aspell инициализировать орфографический словарь. Если сделать это не удалось (например, потому, что файл словаря не найден), метод выводит сообщение об ошибке и возвращает значение `false` (в противном случае метод возвращает значение `true`, что свидетельствует только о том, что проверка орфографии была выполнена, а не о том, что проверяемый текст не содержал ошибок).

Затем в цикле мы перебираем все слова, содержащиеся в тексте, который хранится в `QTextEdit`. Для этого мы используем метод `movePosition()` класса `QTextCursor`, указывая единицу перемещения с помощью константы `QTextCursor::NextWord` (т. е. перемещаемся к следующему слову). Точно так же можно перебрать все строки и все параграфы текста. Между прочим, метод `movePosition()` может перемещать курсор более чем на одну позицию (в этом случае количество позиций, на которое нужно сдвинуться, следует задавать явным образом). Найдя очередное слово, мы вызываем для него метод `_checkWord()`, как вызывали в методе `checkWord()`.

Вызов метода `setCharFormat()` класса `QTextCursor` может вызывать ошибку Segmentation Fault (ошибка доступа к памяти) где-то в глубине кода Qt, если этот метод вызван из обработчика сигнала `textChanged()` объекта класса `QTextEdit`. Об ошибке разработчикам Qt было сообщено уже давно (например, здесь: **http://bugreports.qt.nokia.com/browse/QTBUG-5923**), но она до сих пор (до версии Qt 4.7.2) не исправлена. Для нас этот неприятный факт означает, что мы не можем вызывать метод `checkWord()` класса `SpellChecker` из обработчика указанного сигнала (и некоторых других). Но все не так уж и страшно. В нашем распоряжении есть сигнал `cursorPositionChanged()` класса `QTextDocument`. Связывание обработчика, вызывающего метод `checkWord()`, с этим сигналом не приводит к ошибкам. Указатель на объект класса `QTextDocument` возвращает метод `document()` виджета `QTextEdit`, но связывать сигналы этого объекта со слотами главного объекта программы визуальным способом (с помощью редактора сигналов и слотов) нельзя, так что придется сделать это "вручную", добавив в конструктор класса главного окна строки:

```
connect(ui->textEdit->document(),
SIGNAL(cursorPositionChanged(QTextCursor)), this,
SLOT(richtextChanged()));
```

Исходные тексты демонстрационной программы вы найдете в папке Ch9/spellcheck. Центральным виджетом этой программы является виджет `QTextEdit`. В программе мы можем проверить орфографию всего текста (слот `spellcheckAllText()` класса `MainWindow`) и орфографию редактируемого слова (слот `richtextChanged()`, который обрабатывает сигнал `cursorPositionChanged()` объекта `QTextDocument`). Возможно, вызывать метод `richtextChanged()` всякий раз, когда позиция курсора в тексте меняется, не очень разумно, но это один из наиболее простых способов обойти описанную выше ошибку. Теперь мы можем проверять орфографию текста практически так же, как в Microsoft Word (рис. 9.3).

Но в дополнение ко всему этому мы хотим, чтобы программа, проверяющая орфографию, предлагала нам список вариантов замены отсутствующего в словаре слова,

тем более что интерфейс Aspell и класс `SpellChecker` предоставляют нам такую возможность. Обычно варианты замены слова добавляются в контекстное меню окна редактора. Создать контекстное меню с вариантами замены нетрудно (рис. 9.4).

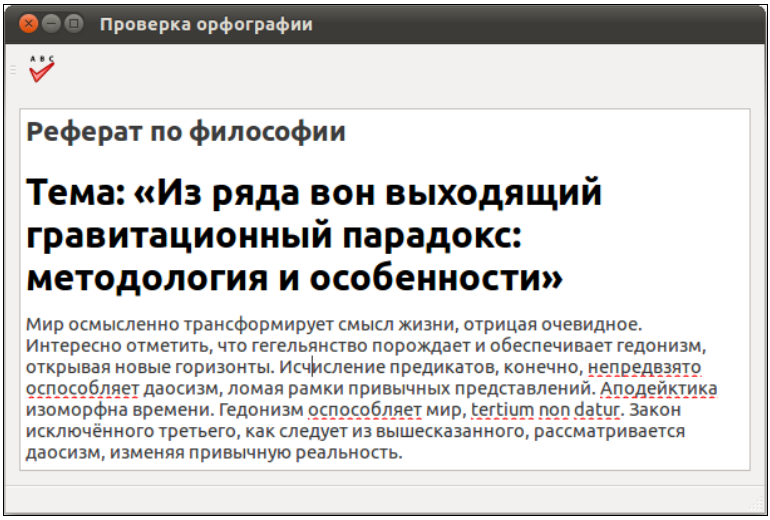


Рис. 9.3. Проверка орфографии в окне `QTextEdit`

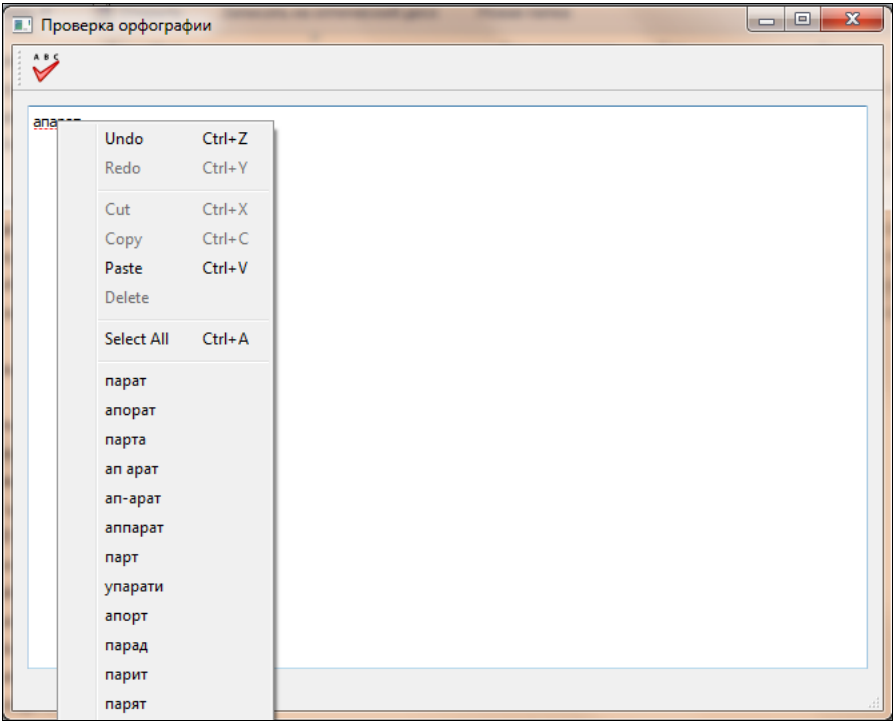


Рис. 9.4. Варианты замены слова в контекстном меню

Но тут возникает проблема объединения нашего собственного меню и контекстного меню виджета `QTextEdit`, которое содержит много полезных команд. Разработчики Qt осознали эту проблему, и, начиная с версии 4.4, предоставили решение для нее. Для того чтобы добавить варианты замены слова в стандартное контекстное меню виджета `QTextEdit`, нужно выполнить несколько действий. Первое: в конструкторе класса главного окна изменяем "политику" создания контекстного меню виджета (листинг 9.20).

Листинг 9.20. Замена стандартного меню виджета на пользовательское

```
ui->textEdit->setContextMenuPolicy(Qt::CustomContextMenu);
    connect(ui->textEdit, SIGNAL(customContextMenuRequested(
        const QPoint&)), this, SLOT(showContextMenu(const QPoint&)));
```

Вызов метода `setContextMenuPolicy()` с аргументом `Qt::CustomContextMenu` приводит к тому, что при щелчке в окне виджета правой кнопкой мыши стандартное контекстное меню этого виджета не выводится, а вместо этого эмитируется сигнал `customContextMenuRequested()`, аргументом которого является точка в системе координат виджета (объект `QPoint`), в которой находился курсор мыши в момент щелчка. Этот аргумент передается для того, чтобы контекстное меню, которое мы создаем сами, можно было вывести там, где пользователь ожидает его увидеть. Этот механизм работает для всех классов-потомков `QWidget`. Слот `showContextMenu()` в нашем примере выглядит так, как показано в листинге 9.21.

Листинг 9.21. Интеграция стандартного и пользовательского контекстного меню

```
void MainWindow::showContextMenu(const QPoint &p)
{
    QMenu * menu = ui->textEdit->createStandardContextMenu();
    QStringList sl = spellChecker->suggestions();
    if (sl.count() > 0)
        menu->addSeparator();
    foreach(QString str, sl) {
        QAction * action = menu->addAction(str);
        connect(action, SIGNAL(triggered()), this, SLOT(replaceWord()));
    }
    menu->exec(ui->textEdit->mapToGlobal(p));
    delete menu;
}
```

Метод `createStandardContextMenu()` — как раз тот метод, которого нам не хватало ранее для интеграции команд стандартного контекстного меню с нашими собственными. Метод возвращает указатель на объект `QMenu`, который уже содержит все команды стандартного контекстного меню виджета `QTextEdit`. Мы получаем список вариантов замены с помощью метода `suggestions()` объекта `SpellChecker` и добавляем команду меню для каждого слова из этого списка.

Добавить новую команду в меню проще всего с помощью метода `addAction()` класса `QMenu`. Этот метод существует в нескольких перегруженных вариантах. Мы используем самый простой из них, принимающий имя команды в том виде, в каком оно должно быть отображено в меню. Данный вариант метода `addAction()` создает объект класса `QAction` для новой команды и возвращает нам указатель на этот объект. Для того чтобы команда работала, мы должны связать сигнал `triggered()` этого объекта с каким-либо слотом в нашей программе.

После этого нам остается вывести контекстное меню с помощью метода `exec()` класса `QMenu`. Параметром метода `exec()` является объект `QPoint`, который задает координаты верхней левой точки выводимого меню, но не в системе координат виджета, а в системе координат экрана, так что координаты, полученные в качестве аргумента сигнала `customContextMenuRequested()`, следует преобразовать с помощью метода `mapToGlobal()` того виджета, в системе координат которого была задана исходная точка. После того, как метод `exec()` вернет управление, объект `QMenu`, созданный методом `createStandardContextMenu()`, можно удалить.

Метод `createStandardContextMenu()`, позволяющий нам модифицировать стандартное контекстное меню виджета, весьма полезная вещь, но он есть не у всех виджетов (хотя бы потому, что не у всех виджетов есть стандартное контекстное меню). Кроме виджета `QTextEdit`, этим методом обладают виджеты `QPlainTextEdit`, `QLineEdit` и `QWebPage`.

Посмотрим теперь на слот `replaceWord()`, который вызывается вновь созданными командами меню и заменяет слово, находящееся под курсором, словом из словаря (листинг 9.22).

Листинг 9.22. Обработка события контекстного меню

```
void MainWindow::replaceWord()
{
     QAction * action = (QAction *) sender();
     QTextCursor cursor = ui->textEdit->textCursor();
     cursor.select(QTextCursor::WordUnderCursor);
     cursor.removeSelectedText();
     cursor.insertText(action->text());
}
```

Поскольку у всех созданных нами объектов `QAction` сигнал `triggered()` связан с одним и тем же слотом, нужно определить, какой именно из объектов стал источником сигнала. Мы делаем это с помощью метода `sender()`. Объект `QTextCursor` позволяет не только изменять форматирование выделенного текста, но и выполнять с ним некоторые другие операции. Например, удалять выделенный текст (метод `removeSelectedText()`) и вставлять текст там, где находится курсор (метод `insertText()`). Для замены неправильного слова на правильное мы пользуемся методом `text()` объекта `action`.

Теперь можно сравнить, как работает старая версия словаря для Windows (см. рис. 9.4) и более новая версия для Linux (рис. 9.5). Старая Windows-версия предлагает заменить написанное с ошибкой слово "апарат" на довольно странные варианты "парат" и "апорат", а правильная замена "аппарат" стоит лишь на шестом месте. Версия для Linux предлагает наиболее очевидную замену "аппарат" первой, да и странные слова в списке вариантов замены отсутствуют.

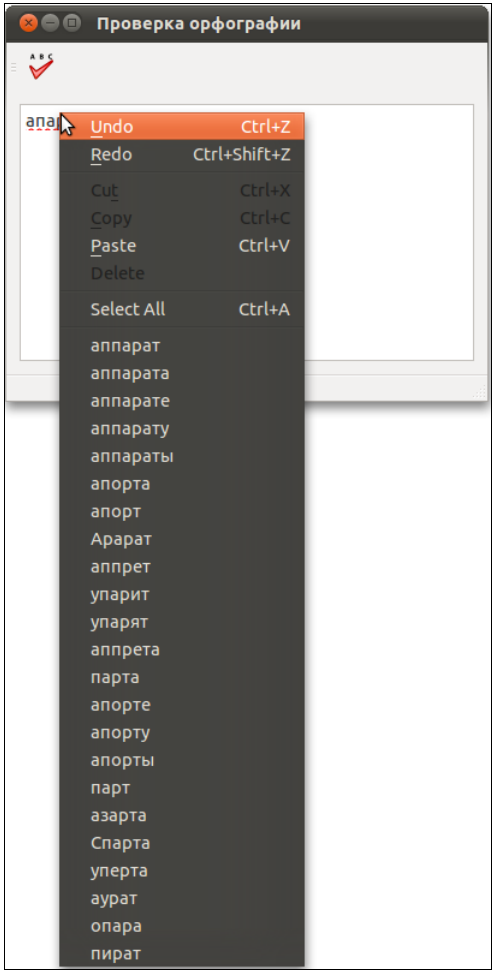


Рис. 9.5. Варианты замены слова в версии словаря для Linux

Примечание

Если вы любите и умеете собирать программы под MinGW, то можете перенести новейшую версию системы Aspell для Linux на платформу Windows и получить такое же высокое качество проверки орфографии, что и в новейших Linux-версиях.

Сохранение документа *QTextDocument* в различных форматах

Класс `QTextDocumentWriter`, который появился в Qt 4.5, играет для класса `QTextDocument` ту же роль, что `QImageWriter` для класса `QImage`, т. е. позволяет сохранять содержимое `QTextDocument` в различных форматах с расширенными возможностями. Стоит отметить, что класса `QTextDocumentReader`, подобного `QImageReader`, в Qt нет. Список форматов, которые `QTextDocumentWriter` поддерживает "из коробки", невелик: простой текст без форматирования, HTML, ODF. Впрочем, возможно, что этого достаточно, особенно если учесть, что ODF (OpenDocument Format, открытый формат документа) имеет все шансы стать стандартом формата офисных документов и уже поддерживается OpenOffice и Microsoft Office. Кроме того, для сохранения обогащенного текста в ряде форматов можно прибегнуть к различным

хитростям, например использовать "псевдопринтер" Qt для сохранения текста в формате PDF (листинг 9.23).

Листинг 9.23. "Псевдопринтер" Qt

```
QPrinter printer;
printer.setOutputFormat(QPrinter::PdfFormat);
printer.setOutputFileName("output.pdf");
ui->textEdit->document()->print(&printer);
```

Добавление изображений в документ *QTextDocument*

Если изображение хранится отдельно и на него можно получить ссылку в формате HTML (т. е. такую же, как при работе с тегом `img`), то добавить отображение этого изображения в обогащенный документ совсем несложно (листинг 9.24).

Листинг 9.24. Добавление изображения, хранящегося во внешнем файле

```
void addImage(QTextCursor cursor, const QString &url)
{
    QTextImageFormat imageFormat;
    imageFormat.setName(url);
    cursor.insertImage(imageFormat);
}
```

Функция `addImage()` добавляет в позиции, указанной курсором, изображение, ссылка на которое передается ему в параметре `url`. Ссылка может указывать на файл на жестком диске вашего компьютера или сетевой ресурс. Этот метод предполагает, что изображение хранится отдельно от документа (со всеми недостатками такого подхода).

> ***Примечание***
>
> Между прочим, попытка вставить таким способом в документ изображение непосредственно из Интернета может натолкнуться на препятствие. Многие сайты запрещают прямые ссылки на хранящиеся на них изображения с других сайтов (hot linking). Вставка изображения из сети с помощью функции `addImage()` будет выглядеть именно как такая попытка.

Для того чтобы вставить в документ изображение из буфера обмена, потребуется другая функция (листинг 9.25).

Листинг 9.25. Добавление изображения из буфера обмена

```
bool addImageFromClipboard(QTextCursor cursor,
    const QString &internalName)
{
    QClipboard * clipboard = QApplication::clipboard();
```

```
    if (!clipboard->pixmap().isNull()) {
        QImage img = clipboard->pixmap().toImage();
        cursor.insertImage(img, internalName);
        return true;
    }
    return false;
}
```

Функция `addImageFromClipboard()` добавляет в позицию курсора изображение, скопированное из буфера обмена (если таковое в нем имелось). Параметр `internalName` необходим для того, чтобы задать внутреннее имя изображения, которое должно быть уникально для каждого изображения. Однако если теперь вы захотите сохранить полученный документ в формате HTML, изображения не сохранятся. Будет сгенерирован документ HTML, в котором будет присутствовать тег `img`, но в качестве источника изображения для этого тега будет указано имя, заданное в параметре `internalName`, а сам файл изображения сохранен не будет. Однако если вы сохраните документ в формате ODF, изображение будет добавлено в документ (листинг 9.26).

Листинг 9.26. Сохранение документа в формате ODF

```
addImageFromClipboard(ui->textEdit->textCursor(), "img01");
QTextDocumentWriter writer("ouput.odt", "ODF");
writer.write(ui->textEdit->document());
```

Вероятно, вы уже поняли, в чем дело: Qt умеет только сохранять обогащенные документы в формате ODF, но не загружать их. Если вам требуется загрузка из формата ODF, то можно воспользоваться кодом из программы Okular. Эта открытая программа распространяется на условиях GPL, так что использовать код из нее напрямую в коммерческом приложении нельзя. Вообще говоря, класс `QTextDocument` все еще ориентирован на работу с HTML.

Если вам понадобится заняться сохранением данных объекта `QTextDocument`, вам, наверное, пригодится класс, извлекающий из документа графические ресурсы (листинг 9.27).

Листинг 9.27. Извлечение графических ресурсов из объекта `QTextDocument`

```
typedef QMap<QString, QPixmap> PixmapCollection;

class PixmapExtractor : public QObject {
public:
    PixmapExtractor(QTextDocument * doc, QObject * parent = 0)
        : QObject(parent)
    {
        textDoc = doc;
    }
```

```
    void extract(PixmapCollection * collection)
    {
        unsigned short sim = 0xFFFC;
        QString str = QString::fromUtf16(&sim, 1);
        QTextCursor cursor = QTextCursor(textDoc);
        cursor.setPosition(0);
        while (!cursor.atEnd()) {
            cursor = textDoc->find(str, cursor);
            if (cursor.isNull()) break;
            QTextCharFormat fmt = cursor.charFormat();
            if (fmt.isImageFormat()) {
                QTextImageFormat ifmt = fmt.toImageFormat();
                QVariant var =
                    textDoc->resource(QTextDocument::ImageResource,
                    QUrl(ifmt.name()));
                QPixmap pixmap = var.value<QPixmap>();
                collection->insert(ifmt.name(), pixmap);
            }
        }
    }
private:
    QTextDocument * textDoc;
};
```

Метод `extract()` класса `PixmapExtractor` заполняет объект класса `PixmapCollection` данными об извлеченных из документа изображениях. Класс `PixmapCollection` представляет собой коллекцию `QMap`, в которой ключом является строка с именем графического ресурса, а значением — объект `QPixmap`. Как уже отмечалось, каждому изображению в документе, хранящемся в `QTextDocument`, соответствует специальный символ кодировки Unicode — `U+FFFC`. В цикле метода `extract()` мы ищем в документе такие символы с помощью метода `find()` объекта `QTextDocument`. Этот метод возвращает объект класса `QTextCursor`. Если этот курсор не является нулевым (метод `isNull()` возвращает значение `false`), значит, он указывает на очередной найденный символ. Затем мы получаем формат символа с помощью уже знакомого нам метода `charFormat()`. Если это символ-якорь изображения, то объект `QTextCharFormat` может быть преобразован в объект `QTextImageFormat` (возможность такого преобразования можно проверить с помощью метода `isImageFormat()` класса `QTextCharFormat`). Из объекта `QTextImageFormat` мы можем извлечь имя графического ресурса (URL или внутреннее имя). Это имя нужно нам, в том числе, для того, чтобы извлечь из документа сам ресурс. Это делается с помощью метод `resource()` класса `QTextDocument`. Метод `resource()` возвращает значение типа `QVariant`, которое мы преобразуем в `QPixmap`. Все, что нам остается сделать, — добавить имя ресурса и объект `QPixmap` в коллекцию.

Если вы хотите сохранить извлеченные ресурсы на диске, сделать это можно, например, так, как показано в листинге 9.28.

Листинг 9.28. Сохранение извлеченных изображений

```
PixmapExtractor pe(ui->textEdit->document());
PixmapCollection pc;
pe.extract(&pc);
foreach(QString key, pc.keys()) {
        pc.value(key).save(key);
}
```

В результате графические ресурсы будут сохранены под теми же именами, под которыми на них ссылается обогащенный документ в HTML-представлении, что упростит конвертацию всего документа в HTML. Только учтите, что ссылки на изображения в документе могут быть и относительными, например /images/image1.jpg. Файл с таким именем без предварительной обработки сохранить не удастся, а в результате придется менять ссылку на изображение и в самом документе.

Новшества в *QTextDocument* и сопутствующих классах

Одной из весьма удобных особенностей класса `QTextDocument` является наличие стеков отката и возврата (undo stack и redo stack). В Qt 4.7 появилась возможность очищать эти стеки с помощью метода `clearUndoRedoStacks()`. У класса `QTextCursor` появилось несколько полезных методов. Методы `keepPositionOnInsert()` и `setKeepPositionOnInsert()` позволяют определить, должен ли менять курсор свое местоположение при вставке нового текста. Метод `setVerticalMovementX()` позволяет сделать так, что при вертикальном перемещении курсора по строкам, использующим пропорциональный шрифт, горизонтальная позиция курсора не будет меняться. У класса `QTextBlock` появился метод `textDirection()`, который позволяет определить направление заполнения текстом (слева направо или справа налево).

Класс *QTextLayout*

Работа со сложными документами — не единственный козырь Scribe. Помимо прочего эта система позволяет выполнять фигурный вывод текста. Если вы хотите, чтобы вместо ровных полей слева и справа выводимый текст был выровнен по контуру какой-нибудь сложной фигуры, воспользуйтесь классом `QTextLayout`. Класс `QTextLayout` управляет компоновкой неформатированного текста, т. е. текста, который выводится одним шрифтом. Объекты класса `QTextLayout` позволяют разбить текст на строки с учетом параметров выбранного шрифта и ширины каждой строки, а также задать расположение каждой строки относительно левого края "виртуального листа". После этого вывести фигурно расположенный текст на экран (на принтер или на другое устройство) можно одной командой.

В качестве демонстрации вывода нестандартно расположенного текста мы рассмотрим программу scribe-demo, полный исходный текст которой вы найдете в

папке Ch9/scribe-demo. Как и предыдущий пример, эта программа добавляет в главное окно свой собственный виджет. На этот раз наш виджет реализуется классом `Label`. Класс `Label` происходит не от класса `QLabel`, а непосредственно от класса `QWidget`, ведь мы собираемся выводить текст нашими собственными средствами и функциональность класса `QLabel` нам ни к чему. Объявление класса `Label` выглядит просто (листинг 9.29).

Листинг 9.29. Класс `Label`

```
class Label : public QWidget
{
public:
    Label(QWidget *parent);
private:
    QTextLayout * textLayout;
    void makeLayout();
protected:
    void paintEvent(QPaintEvent *event);
};
```

В классе `Label` мы перекрываем метод `paintEvent()` класса-предка. Кроме того, мы вводим вспомогательный метод `makeLayout()`. Рассмотрим определения всех трех методов класса `Label` — конструктора, метода `makeLayout()` и метода `paintEvent()` (листинг 9.30).

Листинг 9.30. Методы класса `Label`

```
Label::Label(QWidget *parent) : QWidget(parent)
{
   QFont font("Times", 22, -1, true);
   QString text =
      QObject::trUtf8("Этот фрагмент текста выведен на экран с помощью\\
         системы Scribe...");
   textLayout = new QTextLayout(text, font);
}

void Label::makeLayout()
{
   int indent = 20;
   qreal vertPos = 10;
   QTextLine line;
   textLayout->beginLayout();
   line = textLayout->createLine();
   while (line.isValid())
   {
      line.setLineWidth(width() — 2 * indent);
```

```
        line.setPosition(QPointF(indent, vertPos));
        vertPos += line.height();
        indent += 20;
        line = textLayout->createLine();
    }
    textLayout->endLayout();
}

void Label::paintEvent(QPaintEvent *event)
{
    QPainter painter;
    QPen pen;
    makeLayout();
    painter.begin(this);
    painter.eraseRect(QRect(0, 0, width(), height()));
    pen.setColor(QColor(0, 0, 127));
    painter.setPen(pen);
    textLayout->draw(&painter, QPoint(0,0));
    painter.end();
}
```

В конструкторе `Label` мы создаем объект класса `QTextLayout`. Конструктору объекта `textLayout` передаются два аргумента — ссылка на строку текста и ссылка на объект `QFont`, который определяет используемый шрифт. Все самое интересное сосредоточено в методе `makeLayout()`. Мы начинаем работу с компоновщиком текста с вызова метода `beginLayout()`. Для каждой строки выводимого текста мы создаем объект класса `QTextLine` с помощью метода `createLine()` объекта `textLayout`. Этот метод будет возвращать объекты `QTextLine` со значением `isValid()`, равным `true`, до тех пор, пока весь текст не будет распределен на строки (общее количество строк, разумеется, зависит от размера шрифта и ширины каждой строки). Ширина строки устанавливается с помощью метода `setLineWidth()`, а ее позиция — методом `setPosition()`. Для того чтобы строки не накладывались друг на друга, мы смещаем отступ очередной строки от верхнего края на значение, равное высоте строки. В этом нам помогает метод `height()` объекта класса `QTextLine`. После того как создание и расположение строк закончены, мы вызываем метод `endLayout()`.

Метод `makeLayout()` создает единый графический примитив, своего рода шаблон, содержащий текст. Для вывода этого текста в виджет достаточно вызвать метод `draw()` объекта `textLayout`. Первым аргументом метода `draw()` должна быть ссылка на объект класса `QPainter`, вторым аргументом — ссылка на объект `QPoint`, определяющий расположение левого верхнего угла той области, в которой выводится текст (фактическое расположение строк, заданное в методе `makeLayout()`, при выводе текста будет отсчитываться относительно этой точки). В результате получается фигурная надпись, которая, возможно, не везде следует заданному контуру (все-таки текст состоит из слов разной длины), но довольно точно повторяет его (рис. 9.6).

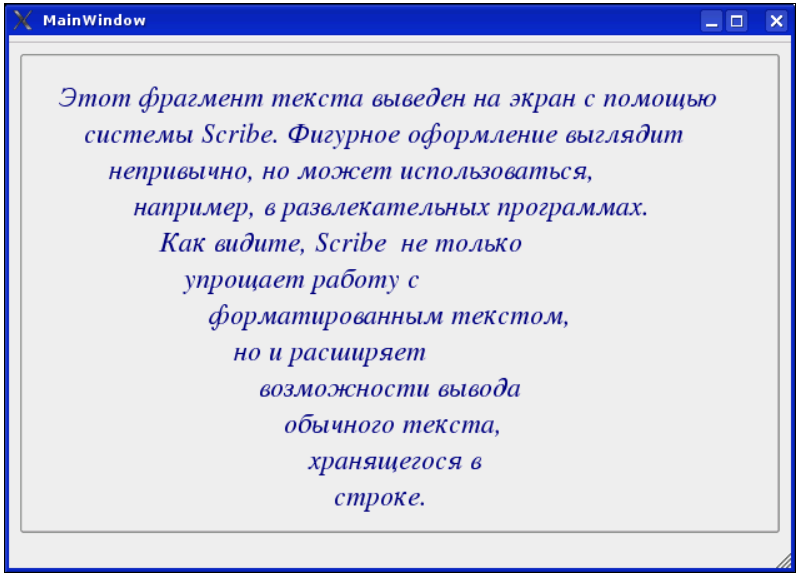


Рис. 9.6. Фигурное расположение текста

Обратите внимание на важную особенность класса `QTextLayout`. Каждый раз, когда мы вызываем метод `beginLayout()`, информация о предыдущей компоновке текста (и всех созданных объектах `QTextLine`) теряется. Для новичка эта особенность может стать источником труднообъяснимых ошибок, но нам она позволяет создать объект класса `QTextLayout` один раз (в конструкторе `Label`), а затем использовать его многократно для генерации разных компоновок текста в методе `makeLayout()`. Ведь даже при одних и тех же параметрах шрифта и ограничивающей линии компоновка текста зависит от ширины виджета. В связи с этим любопытно отметить, что в релизе Qt 4.4 у класса `QTextLayout` появился метод `clearLayout()`, который очищает список строк компоновщика текста.

Класс *QStaticText*

Этот класс, предназначенный для вывода "статичного" текста, т. е. текста, который меняется редко, появился в Qt 4.7. Основное преимущество этого класса — скорость вывода текста, так что его лучше использовать там, где требуется выводить большие блоки текста с определенными опциями. В отличие от класса `QTextLayout`, класс `QStaticText` позволяет отображать обогащенный текст. При описании класса `QTextLayout` было заказано, что объекты этого класса можно использовать как отдельные графические примитивы, не заботясь об их внутреннем содержимом. Класс `QStaticText` развивает эту идею даже дальше, хотя и в несколько ином направлении. Объект класса `QStaticText` можно выводить на экран (или любое другое устройство графического вывода) с помощью специального метода класса `QPainter` (метод `drawStaticText()`) точно так же, как, например, прямоугольник или изображение `QPixmap`. Иначе говоря, у нас появляется возможность работать с блоком обогащенного текста как с графическим примитивом.

Одна из ситуаций, где применение класса `QStaticText` может принести существенную пользу, — рисование сложной сцены с использованием объекта `QPainter`, например при комбинировании графики OpenGL и "обычной" графики, создаваемой с помощью `QPainter`. Мы воспользуемся классом `QStaticText` для вывода статического обогащенного текста поверх сцены OpenGL (программа textover3d). Поскольку программа textover3d имеет дело с выводом графики OpenGL, ее центральный виджет будет потомком `QGLWidget` (листинг 9.31).

Листинг 9.31. Главный виджет программы textover3d

```
class GLSTWidget : public QGLWidget
{
    Q_OBJECT
public:
    explicit GLSTWidget(QWidget *parent = 0);
    ~GLSTWidget();
protected:
    void initializeGL();
    void resizeGL(int width, int height);
    void paintGL();
private:
    int texid;
    QStaticText * staticText;
};
```

Демонстрационная программа выводит статическую надпись поверх созданного средствами OpenGL прямоугольника, покрытого текстурой. Хотя объявление класса `GLSTWidget` очень похоже на объявления других классов-потомков `QGLWidget`, которыми мы пользовались, его методы реализованы по-другому. Рассмотрим сначала конструктор класса `GLSTWidget` (листинг 9.32).

Листинг 9.32. Конструктор класса `GLSTWidget`

```
GLSTWidget::GLSTWidget(QWidget *parent) : QGLWidget(parent)
{
    this->setAutoBufferSwap(false);
    QFile f(":/text.txt");
    f.open(QIODevice::ReadOnly);
    QByteArray ba = f.readAll();
    f.close();
    ba.append('\0');
    QString s = QString::fromUtf8(ba.data());
    staticText = new QStaticText(s);
    staticText->setTextOption(QTextOption(Qt::AlignHCenter));
    setAutoFillBackground(false);
    makeCurrent();
}
```

В конструкторе мы создаем объект класса `QStaticText` и загружаем в него текст с помощью параметра конструктора, который имеет тип `QString`. Текст, который мы загружаем из файла-ресурса text.txt, содержит разметку HTML, которая будет соответствующим образом отображена объектом `staticText`. Как и виджет `QLabel`, класс `QStaticText` может "угадать" по наличию определенных символов, что переданный ему текст является фрагментом HTML, и включить режим вывода обогащенного текста автоматически. Возможно, что объект `QStaticText` ошибется и попытается обработать фрагмент обычного текста как фрагмент HTML. Возможно также, что мы захотим, чтобы объект `QStaticText` выводил текст на языке HTML в режиме отображения тегов. Для подобных случаев существует метод `setTextFormat()`. Передав этому методу в качестве аргумента константу `Qt::PlainText`, мы заставим объект класса `QStaticText` обрабатывать переданный ему текст как простой текст, не содержащий форматирования. Метод `setTextOption()` позволяет нам задать некоторые дополнительные параметры статического текста, например, его выравнивание внутри блока. Задать шрифт и его размеры с помощью методов класса `QStaticText` мы не можем (у этого класса таких методов просто нет), но ничто не мешает нам определить эти параметры в самом обогащенном тексте с помощью тегов HTML.

> ***Примечание***
>
> В отличие от классов `QTextEdit` и `QTextBrowser`, класс `QStaticText` не умеет обрабатывать изображения. Если переданный объекту `QStaticText` фрагмент HTML сдержит тег `img`, этот тег будет учтен при компоновке текста, но сама картинка не будет показана. Ее нужно выводить отдельно.

Способ вывода графики, которым пользуется программа textover3d (сначала создаем сцену OpenGL, а затем дорисовываем элементы с помощью объекта `QPainter`), называется в документации Qt "over-painting" (рисование поверх). Документация рекомендует полностью восстановить состояние контекста OpenGL перед тем, как обращаться к методам класса `QPainter`. Иногда это действительно может быть необходимо, но в общем случае — нет. Обязательным при этом способе вывода графики является вызов метода `setAutoFillBackground(false)` виджета-потомка `QGLWidget`. Благодаря этому вызову методы объекта `QPainter` не будут перерисовывать фон виджета там, где в этом нет явной необходимости, и, соответственно, сцена, созданная методами OpenGL, не будет затерта.

Рассмотрим методы `initializeGL()` и `paintGL()` класса `GLSTWidget` (листинг 9.33).

Листинг 9.33. Методы `initializeGL()` и `paintGL()` класса `GLSTWidget`

```
void GLSTWidget::initializeGL()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClearDepth(1.0f);
    glHint(GL_POLYGON_SMOOTH_HINT, GL_NICEST);
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
```

```
    glEnable(GL_TEXTURE_2D);
    QPixmap pm;
    pm.load(":/bw.png");
    texid = bindTexture(pm);
    paintGL();
}

void GLSTWidget::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(-0.0f,0.0f,-0.5f);
    glBindTexture(GL_TEXTURE_2D, texid);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f);
    glVertex3f( 1.0f,-1.0f,-1.8f);
    glTexCoord2f(1.0f, 0.0f);
    glVertex3f(-1.0f,-1.0f,-1.8f);
    glTexCoord2f(1.0f, 1.0f);
    glVertex3f(-1.0f, 1.0f,-1.0f);
    glTexCoord2f(0.0f, 1.0f);
    glVertex3f( 1.0f, 1.0f,-1.0f);
    glEnd();
    QPainter painter(this);
    painter.drawStaticText((width() — staticText->size().width())/2,
       10, *staticText);
    painter.end();
    swapBuffers();
}
```

Новое для нас в методе `initializeGL()` — загрузка текстуры из объекта `QPixmap` (сама текстура хранится в файле формата PNG). Метод `bindTexture()` с параметром `QPixmap` полностью подготавливает текстуру к использованию, экономя нам массу времени и избавляя от написания большого объема кода. В методе `paintGL()` мы сначала создаем покрытый текстурой квадрат, а затем выводим надпись с помощью объектов `painter` (класс `QPainter`) и `staticText`. Метод `drawStaticText()` позволяет задать координаты верхнего левого угла блока текста относительно поверхности, с которой работает объект `painter`, и это единственное, что мы можем изменить при выводе объекта `staticText`. Поскольку объект `staticText` представляет собой графический примитив, мы не в состоянии выяснить ни количество строк текста в нем, ни параметры отдельных строк. Все, что мы можем узнать, — это размеры ограничивающего прямоугольника, который возвращает нам метод `size()` класса `QStaticText`. По этим же причинам размеры текстового блока, созданного с помощью `QStaticText`, не могут измениться вместе с размерами виджета (в отличие от блоков `QTextLayout`).

В результате мы получаем блок обогащенного текста поверх трехмерной сцены, который перерисовывается с молниеносной быстротой (рис. 9.7). Изменяя размеры окна, вы можете увидеть, как разные элементы изображения (надпись и "стена" на заднем плане) движутся отдельно друг от друга.

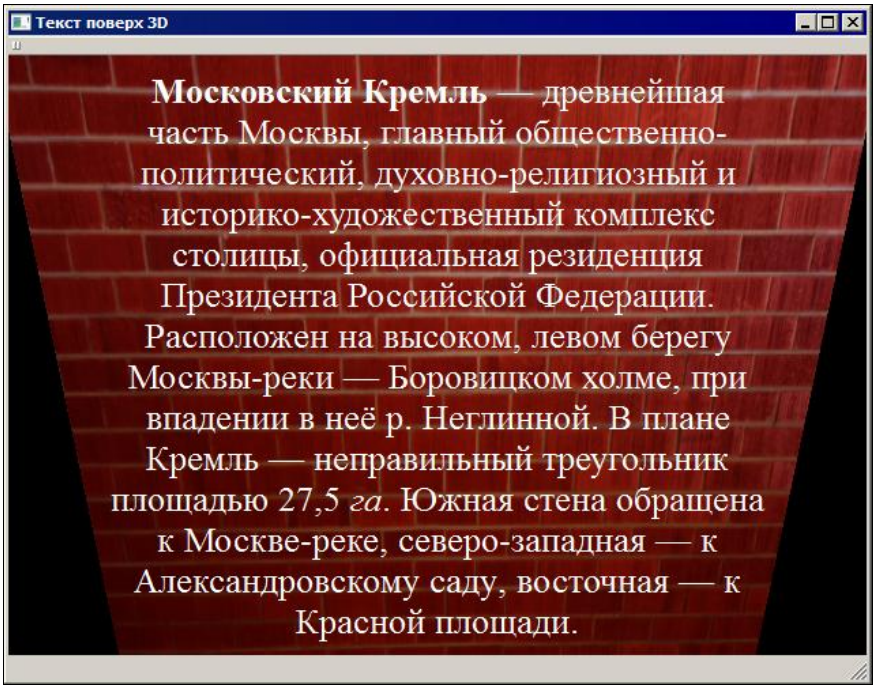


Рис. 9.7. Вывод статического текста поверх графики OpenGL

ГЛАВА 10

Система Graphics View Framework

Система Graphics View Framework, появившаяся в Qt, начиная с версии 4.2, пришла на смену графической системе, основанной на классе `QCanvas`. Graphics View Framework — это не только система вывода графики с широкими возможностями, но и реализация уже знакомой нам парадигмы "модель–контроллер–вид" для программ, работающих с двухмерной графикой. Основное отличие Graphics View Framework от других систем вывода графики Qt заключается в том, что элементы изображения, созданного с помощью Graphics View Framework, рассматриваются как отдельные объекты, а не как наборы пикселов. С помощью Graphics View Framework вы можете создавать динамичные интерактивные графические сцены. Графические примитивы Graphics View Framework можно рассматривать как специализированные виджеты (между прочим, графическими примитивами Graphics View Framework могут быть самые настоящие виджеты). Из всего сказанного вытекает еще одна возможность применения Graphics View Framework — создание сложных и нестандартных виджетов, требующих особой работы с графикой.

> ***Примечание***
>
> Незнакомые с возможностями Qt 4.x программисты, которым требуется создать сложную интерактивную графическую схему, часто "рисуют" ее с помощью стандартных виджетов, используя кнопки и панели в качестве графических примитивов, которые, во-первых, интерактивны, во-вторых, могут сами "позаботиться" о своей перерисовке. Мне лично приходилось видеть такие сцены, созданные в Delphi и в Qt 3.x. Подобный подход не очень удобен и весьма неэффективен. Все-таки кнопки и панели предназначены не для рисования. Система Graphics View Framework была создана, отчасти, для того, чтобы решить эту проблему.

Предположим, вы решили написать двухмерную компьютерную игру. Применение подхода "модель–контроллер–вид" может существенно упростить процесс создания такой игры. Описание игрового мира представляет собой модель данных программы. Визуализацию игрового мира выполняет объект отображения (вид). Контроллер транслирует действия пользователя в события модели. Система Graphics View Framework предоставляет вам заготовки для создания модели, контроллера и объекта отображения, изначально наделенные широкой функциональностью. Кроме того, Graphics View Framework берет на себя решение таких задач, как обнаружение столкновений (collision detection) и геометрические преобразования изображений.

Разумеется, Graphics View Framework может найти применение не только в играх, но и в любых программах, которым приходится отображать интерактивные графические модели, состоящие из большого количества элементов. Основу Graphics View Framework составляют три класса Qt Library (рис. 10.1).

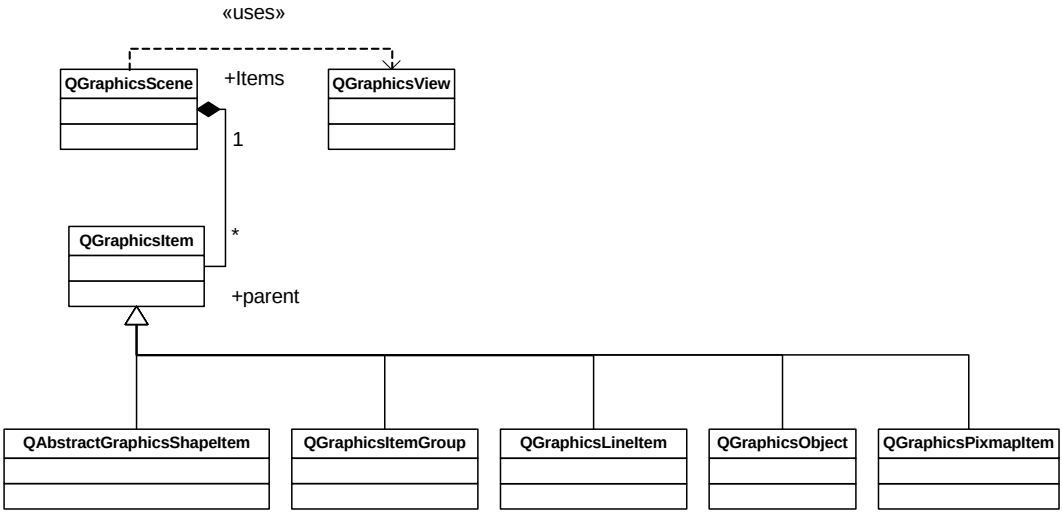


Рис. 10.1. Основные классы Graphics View Framework

Модель данных реализована с помощью объекта класса `QGraphicsScene`. Элементами модели данных являются графические примитивы (геометрические фигуры, группы фигур, растровые изображения, виджеты). Все графические примитивы реализованы с помощью классов-потомков класса `QGraphicsItem`. Таким образом, объект класса `QGraphicsScene` можно рассматривать как контейнер для набора объектов классов-потомков `QGraphicsItem`. Для отображения модели, созданной в `QGraphicsScene`, служит объект класса `QGraphicsView`. Работая в системе Graphics View Framework, вы не рисуете изображение непосредственно в окне `QGraphicsView` (хотя, в принципе, можете это делать). Вместо этого вы управляете объектами, хранящимися в модели `QGraphicsScene`. Все изменения объектов модели автоматически отображаются в окне `QGraphicsView`. При этом вам не нужно заботиться, например, о перерисовке изображения при изменении размеров окна. Поскольку объект класса `QGraphicsView` связан с моделью, он "знает", что нужно отображать в окне, и обновляет содержимое окна автоматически. Точно так же как невизуальный объект класса `QTableDataModel` может отображаться несколькими визуальными компонентами `QTableWidget` и `QTableView` одновременно, один невизуальный объект `QGraphicsScene` может отображаться несколькими визуальными компонентами `QGraphicsView`. Таким образом, в модели Graphics View Framework мы стакиваемся с той же идеей отделения данных от их представления, что и в Interview Framework.

Вторая важная задача, которую решает связка объектов `QGraphicsView` и `QGraphicsScene` — преобразование действий пользователя (таких как щелчок мышью, перемещение указателя мыши над объектом или нажатие клавиши) в события

модели. События модели могут быть переданы дальше отдельным примитивам, формирующим модель. В документации Qt эта система передачи событий между разными уровнями Graphics View Framework называется event propagation.

Упомянутые функции обнаружения столкновений и геометрических преобразований реализованы в классах `QGraphicsScene` и `QGraphicsItem`. Все эти операции выполняются независимо от уровня отображения (на уровень отображения передается только конечный результат операций).

Сами объекты `QGraphicsItem` могут образовывать иерархии. При этом, например, геометрические преобразования, применяемые к родительскому объекту, автоматически применяются и к дочерним объектам. В результате вы можете создать сложный графический объект из нескольких простых объектов и затем работать с ним как с единым целым.

Класс `QGraphicsItem` не является потомком класса `QObject`. Это означает, что он не наследует такие механизмы `QObject`, как механизм сигналов и слотов и стандартный механизм событий Qt (для обработки событий у класса `QGraphicsItem` есть собственный метод `sceneEvent()`). Если вам нужно, чтобы элемент графической сцены вел себя как потомок `QObject`, вам следует сделать свой класс потомком класса `QGraphicsObject`. Этот класс, введенный в Qt 4.6, является потомком как `QGraphicsItem`, так и `QObject`. Он уступает в быстродействии классам, наследующим только от `QGraphicsItem`, так что использовать его в качестве базового класса для своих классов следует только в том случае, если это действительно нужно.

Кроме классов-потомков `QGraphicsItem`, для отображения элементов сцены могут использоваться классы-потомки `QGraphicsWidget`. Этот класс, введенный в Qt 4.4, представляет собой базовый класс для виджетов, специально предназначенных для `QGraphicsScene` (как уже отмечалось, в Graphics View Framework можно использовать и обычные виджеты, потомки `QWidget`, но замена их потомками `QGraphicsWidget` повышает скорость вывода сцены на экран). Вопреки своему названию, класс `QGraphicsWidget` не является потомком `QWidget` (но является потомком `QObject`). Это означает, что хотя класс `QGraphicsWidget` эмулирует многие аспекты поведения класса `QWidget`, его нельзя использовать во многих ситуациях, где можно использовать `QWidget` (например, в паре с классом `QDataWidgetMapper`).

Знакомство с системой

Рассмотрим работу простейшей программы Graphics View Framework, выводящей на экран статическое изображение. Эта программа должна выполнить минимальную последовательность операций, необходимых для работы с Graphics View Framework: создать объекты `QGraphicsScene` и `QGraphicsView` и связать их между собой, затем заполнить объект `QGraphicsScene` графическими примитивами и сделать объект `QGraphicsView` видимым. Написание программы мы начнем с редактирования визуальной части.

Класс `QGraphicsView` является потомком `QFrame` и его удобно сделать центральным визуальным элементом главного окна. Затем следует создать объект класса

QGraphicsScene (это можно сделать, например, в конструкторе главного окна). С помощью метода setScene() объекта QGraphicsView мы связываем объект QGraphicsScene с объектом QGraphicsView:

```
QGraphicsScene * scene = new QGraphicsScene;
graphicsView->setScene(scene);
```

Добавлять графические примитивы в объект QGraphicsScene можно разными способами, в том числе с помощью методов группы add* класса QGraphicsScene. Например, для того чтобы добавить в сцену эллипс, вызываем:

```
scene->addEllipse(QRectF(-100.0, -100.0, 100.0, 100.0));
```

где scene — объект QGraphicsScene. Обратите внимание на то, что координаты эллипса (точнее, координаты прямоугольника, в который он вписан) задаются числами с плавающей точкой, а не целыми числами, как обычно принято в растровой графике. Далее мы увидим, что встроенная в Graphics View Framework система геометрических преобразований, а также наличие нескольких систем координат, делают использование чисел с плавающей точкой совершенно необходимым. К тому же, в системе Graphics View Framework графические данные, инкапсулированные сценой, отделены от любого физического способа отображения, что предоставляет дополнительный аргумент в пользу более гибкой системы координат, представленных числами с плавающей точкой.

Координаты, которые мы указали при добавлении эллипса, являются координатами модели, а не графического окна. При отображении модели объектом QGraphicsView они будут автоматически переведены в координаты окна QGraphicsView. Как соотносятся точки начала координат модели и начала координат окна? Ответ на этот вопрос может показаться неожиданным: соотношение систем координат зависит от размеров изображения и размеров окна. По умолчанию графическая система располагает изображение, созданное в объекте класса QGraphicsScene, таким образом, чтобы его геометрический центр совпадал с центром окна QGraphicsView. Если размеры изображения превышают размеры окна, в окне появляются полосы прокрутки. Все это означает, что не существует простой формулы для перевода координат окна в координаты модели и обратно. Если вас не увлекают занятия аналитической геометрией, для пересчета координат лучше воспользоваться специальными функциями, предоставляемыми системой.

Коль скоро речь зашла о координатах, следует отметить, что, кроме систем координат окна и модели, нам придется иметь дело с еще одной системой координат — системой координат примитива. У каждого графического примитива есть своя система координат, которая используется при выполнении геометрических преобразований примитива (перенос, вращение, масштабирование), и мы подробно обсудим ее далее в этой главе.

В процессе передачи событий от одного уровня к другому система Graphics View Framework выполняет преобразования координат. Например, если ваша модель обрабатывает щелчки мыши, координаты указателя мыши в окне QGraphicsView в момент щелчка будут автоматически переведены в координаты модели. Если событие мыши связано с одним из графических примитивов, то координаты указателя будут

отображены и в систему координат примитива. Таким образом, в Graphics View Framework зачастую приходится иметь дело с тремя наборами координат одной и той же точки (правда, не все эти координаты будут нам нужны).

Теперь мы должны сделать объект `QGraphicsView` видимым с помощью метода `show()`. Затем можно скомпилировать программу. Система Graphics View Framework является частью ядра Qt, поэтому подключать дополнительные модули не требуется. В результате работы нашей программы мы получаем окно, в котором на белом фоне изображена черная окружность. Рисунок этот, конечно, не особенно впечатляет, но наше знакомство с Graphics View Framework состоялось.

Пишем свою игру

Для более подробного знакомства с возможностями Graphics View Framework мы напишем обещанную игру, подобие всем известного "Сокобана" (рис. 10.2). Напомню правила этой древней и мудрой игры: по лабиринту ходит грузчик, задача которого заключается в том, чтобы перенести хаотично разбросанные ящики в заранее определенное место. Грузчик может только толкать ящик перед собой (тащить за собой не может), причем в каждый момент времени он может толкать только один ящик. Полный исходный текст программы вы найдете в папке Ch10/sokoban на сайте **http://symmetrica.net/qt47book/**.

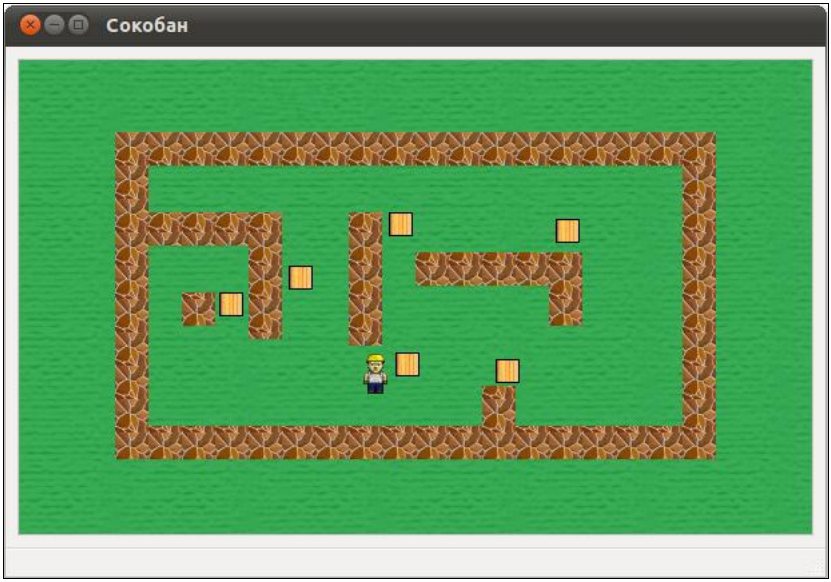


Рис. 10.2. Игра "Сокобан"

Формат файла данных

Прежде чем перейти к работе с графикой, рассмотрим один вопрос, который имеет значение для любого разработчика игр, а именно, вопрос о хранении данных игры.

То, что данные, описывающие игровой мир, должны храниться в отдельном файле, я думаю, очевидно. Вопрос в том, каким должен быть формат этого файла и как реализовать загрузку. Нам предстоит выбрать между текстовым и двоичным форматом. Под текстовым форматом я подразумеваю такой формат, который может быть прочитан (и понят) человеком с помощью обычного текстового редактора. В былые времена разработчики игр (и не только игр) отдавали предпочтение двоичным форматам.

> ***Примечание***
>
> Термин "двоичный" или "бинарный" применительно к форматам файлов в противоположность "текстовым" форматам, конечно, не очень удачен. В конце концов, любые файлы хранятся в двоичном виде, но, поскольку такая терминология устоялась, мы будем ею пользоваться.

Одно из преимуществ двоичных форматов — компактность. Во времена дисков малой емкости было важно, что число 128 занимает один байт, а не три, как в текстовом представлении. Двоичные форматы и сейчас процветают там, где требуется компактность. Вряд ли мы когда-нибудь увидим текстовый формат для передачи видео или звука, хотя теоретически это, конечно, возможно. Второе преимущество двоичных форматов заключается в том, что данные в них обычно хранятся так, как удобно программе, а не так, как удобно человеку. Это значит, что специализированным программам проще сохранять и считывать данные в двоичных форматах.

Преимущества текстовых форматов не так очевидны, но не менее важны. Одно из главных преимуществ заключается в том, что человек всегда может разобраться в формате файла, и, например, восстановить его в случае возникновения ошибки. Это также полезно при отладке. Файлы текстовых форматов можно редактировать в любом редакторе, а не только в специализированных программах. Наконец, третье преимущество заключается в том, что текстовые форматы лучше поддаются формализации и обычно являются более гибкими. Представьте себе группу программистов, работающих над одним проектом. Допустим, они используют двоичный формат обмена данными. Как правило, такие форматы имеют очень жесткую структуру, и работоспособность отдельных компонентов проекта напрямую зависит от соблюдения правил этой структуры, которые зачастую недостаточно полно описаны и могут меняться в ходе разработки. При внесении изменений в формат одним из участников проекта в различных модулях могут возникнуть ошибки, источник которых будет трудно обнаружить. Неслучайно большинство протоколов Интернета обмениваются сообщениями в текстовом формате (исключение составляют протоколы передачи мультимедийных файлов, и причины этого вполне понятны). В общем, если средний размер файлов данных невелик и не превышает десяти мегабайт, и скорость его передачи и чтения не являются критичными для системы, я рекомендую использовать текстовые форматы.

Одна из трудностей, связанных с текстовыми форматами, заключается в том, что они должны быть устойчивы к мелким ошибкам и неточностям. Иначе значительная часть преимуществ такого формата просто утрачивается. Файлы текстовых форматов если и не предназначены, то, по крайней мере, допускают редактирова-

ние их человеком, "вручную", в текстовом редакторе. А людям труднее следовать жестким правилам синтаксиса, чем машинам нарушать их. Текстовый формат, который жестко фиксирует количество пробелов между элементами, количество знаков числа, количество строк и т. п., на самом деле представляет собой гибрид из собственно текстового и двоичного форматов, причем берет от обоих вариантов худшее, а не лучшее. Практическим примером грамотно спроектированного текстового формата данных является формат HTML, который, при наличии жесткого стандарта, позволяет множество отклонений от этого стандарта, причем большинство браузеров постараются отобразить страницу HTML корректно, даже если многие правила синтаксиса HTML были нарушены.

Одна из самых сложных составляющих работы с файлом в текстовом формате — это его синтаксический разбор, который в данном случае сводится к преобразованию текста, который понимает человек, в структуру, которую "понимает" программа. Однако библиотека Qt предоставляет нам простые и мощные средства для синтаксического разбора текстов, так что использовать текстовый формат для хранения данных нашей игры будет несложно.

Прежде всего, надо четко определить, что именно мы собираемся хранить в файлах данных к игре. Описание уровня игры "Сокобан" состоит из трех элементов: координат стен, координат ящиков и начальных координат игрока. Нам, соответственно, требуется хранить эти три блока координат. Поскольку все ящики имеют фиксированный размер, как и фигурка игрока, для того, чтобы указать их расположение, требуется только две координаты (верхнего левого угла). Стены представляют собой прямоугольники различной формы, поэтому для определения положения одной стены требуется две пары координат (это могут быть координаты двух точек прямоугольника, либо координата верхнего левого угла плюс пара значений, определяющих высоту и ширину прямоугольника). Поскольку система Graphics View Framework работает с координатами, представленными числами с плавающей точкой, наш формат тоже должен допускать такое представление координат.

За основу формата файлов данных взят формат XML. Можно много говорить о гибкости и расширяемости языка XML, но дело еще и в том, что в состав Qt входит очень удобный синтаксический анализатор (парсер) XML. Так что, выбирая XML в качестве основы, мы уже упрощаем себе задачу. Но это не все. Формат описания уровня в XML тоже может иметь разную структуру. Можно, например, написать отдельную пару тегов для каждой координаты (листинг 10.1).

Листинг 10.1. Неэффективный вариант структуры файла

```
<wall>
<wall:x>0</wall:x>
<wall:y>10</wall:y>
<wall:with>100</wall:with>
<wall:height>20</wall:height>
</wall>
```

И так далее для каждой стены и каждого ящика. Подобная структура очень удобна для разбора с помощью синтаксического анализатора XML, но слишком громоздка для человека. Найти или исправить что-то в таком файле будет довольно трудно.

Попробуем упростить структуру файла данных для читателя (и усложнить для разбора в программе). Структура описания уровня игры предствлена в листинге 10.2.

Листинг 10.2. Улучшенный вариант структуры файла

```
<walls>{(x, y), (w, h)}, {(x, y), (w, h)},... </walls>
<boxes>(x, y), (x, y) ,... </boxes>
<player>(x, y) </player>
```

Здесь `x` и `y` — координаты верхнего левого угла объекта, `w` и `h` — ширина и высота прямоугольника соответственно. Все значения являются числами с плавающей точкой. Для синтаксического разбора подобной структуры анализатора XML будет уже недостаточно, поскольку придется выполнять синтаксический анализ текста, заключенного между тегами. Но, как мы увидим, сделать это тоже будет нетрудно.

Если вы внимательно посмотрите на формат описания уровня, то увидите, что он содержит повторяющиеся элементы (имеются в виду элементы структуры, а не элементы конкретного файла-описания, которые, естественно, будут повторяться). Речь идет о паре чисел, разделенных запятой и заключенных в круглые скобки. Этот элемент встречается и в описании стен, и в описании расположения ящиков, и в указании позиции игрока, поэтому разработку синтаксического анализатора файла данных мы начнем с разработки анализатора этого базового элемента.

В синтаксическом анализе текста, расположенного между тегами XML, мы воспользуемся классом `QRegExp`. Этот класс, как и следует из его названия, предназначен для работы с регулярными выражениями. С помощью объектов `QRegExp` можно выполнить много полезных действий, например, проверить соответствие введенной пользователем строки определенному формату. Кроме того, найдя в строке текста фрагмент, удовлетворяющий регулярному выражению, объект `QRegExp` позволяет выделить из строки фрагмент, соответствующий определенной части регулярного выражения. Именно этой возможностью класса `QRegExp` мы и воспользуемся.

> ***Примечание***
>
> Описывать язык регулярных выражений, который использует `QRegExp`, мы не будем. Он достаточно подробно описан в документации, а для обсуждения всех его возможностей понадобилась бы отдельная книга. Здесь, как и в других случаях, предполагается знакомство читателя с документацией Qt.

Рассмотрим сначала класс `CoordinateParser` (листинг 10.3). Этот класс получает строку текста вида `" (x, y), (x, y), ... "` и превращает ее в список объектов `QPointF`. Следуя изложенному выше принципу устойчивости текстовых форматов к мелким неточностям, строка текста, которую способен разобрать объект `CoordinateParser`, допускает произвольное количество пробелов между элементами и может даже иметь вид `"(x, y) (x, y) ..."`.

Листинг 10.3. Класс `CoordinateParser`

```
typedef QList<QPointF> PointList;

class CoordinateParser : public QObject
{
    Q_OBJECT
public:
    explicit CoordinateParser(const QString &text, QObject *parent = 0);
    PointList const * points();
protected:
    virtual void parse(const QString &text);
    QString m_text;
    QString number;
    QString coordinate;
    PointList pointList;
};
```

Непосредственный синтаксический разбор выполняет метод `parse()`, который объявлен как виртуальный для того, чтобы его можно было перекрыть в классах-потомках. Метод `points()` возвращает результат разбора строки, т. е. список точек в том порядке, в котором они следовали в строке. Перейдем теперь к реализации методов класса `CoordinateParser` (листинг 10.4).

Листинг 10.4. Методы класса `CoordinateParser`

```
CoordinateParser::CoordinateParser(const QString &text, QObject *parent):
    QObject(parent), m_text(text)
{
    number = "\\d+(\\.\\d+)?";
    coordinate = QString("\\(\\s*%1\\s*,\\s*%1\\s*\\)").arg(number);
}

PointList const *  CoordinateParser::points() {
     if (!pointList.count())
         parse(m_text);
     return &pointList;
}

void CoordinateParser::parse(const QString &text) {
     QRegExp pairsExp(coordinate);
     QRegExp numbersExp(number);
     int pos  = 0;
     while ((pos = pairsExp.indexIn(text, pos)) != -1) {
         QPointF p;
         int pos1 = numbersExp.indexIn(pairsExp.cap(0), 0);
         p.setX(numbersExp.cap(0).toFloat());
```

```
            pos1 += numbersExp.matchedLength();
            numbersExp.indexIn(pairsExp.cap(0), pos1);
            p.setY(numbersExp.cap(0).toFloat());
            pointList.append(p);
            pos += pairsExp.matchedLength();
        }
    }
```

При работе с регулярными выражениями возникает одна проблема: они могут стать слишком сложными для восприятия. Например, не самое сложное регулярное выражение для строк текста, с которыми работает `CoordinateParser`, выглядит так:

```
"\\(\\s*\\d+(\\.\\d+)?\\s*,\\s*\\d+(\\.\\d+)?\\s*\\)"
```

В таких строках легко запутаться и совсем непросто что-то изменить в них. Один из способов упрощения работы с регулярными выражениями заключается в разбиении их на части. Посмотрим, как это делается в конструкторе класса `CoordinateParser`. Строке `number` присваивается регулярное выражение, описывающее число с плавающей точкой. Допустимыми для этого выражения являются, например, строки `"123"`, `"1.018"`, `"2.00"` (но не `"5."` и не `".123"`, хотя синтаксис C++ допускает и такое).

> ***Примечание***
>
> Дело в том, что язык C++, как и многие другие, делит форматы чисел на целые и числа с плавающей точкой, в которых эта самая точка должна присутствовать обязательно. Если точки нет, число в C++ по умолчанию считается целым, а все операции над ним — целочисленными. Думаю, все программисты хотя бы раз сталкивались с неприятными последствиями этого факта при выполнении операции деления. Хотя с точки зрения описания синтаксиса чисел такое разделение упрощает работу.

Строка `coordinate` содержит уже полное регулярное выражение, способное обрабатывать списки пар чисел, заключенных в круглые скобки. Нетрудно видеть, как мы конструируем эту строку, используя строку `number` и метод `arg()` класса `QString`.

Перейдем теперь к самому процессу синтаксического разбора. Наша задача заключается не столько в том, чтобы определить, соответствует ли строка текста заданной структуре, сколько в том, чтобы выделить определенные элементы из этой строки. Мы решаем эту задачу в два этапа. С помощью объекта `pairsExp` разбиваем переданную ему строку на фрагменты вида `"(x, y)"`. Для этого мы используем метод `indexIn()`, который позволяет найти в тексте очередное вхождение строки, заданной регулярным выражением. В результате сами фрагменты, соответствующие регулярному выражению, могут быть разделены практически любыми символами. То есть, например, между отдельными элементами `"(x, y)"` в группе `boxes` могут стоять не только пробелы и запятые, но и символы табуляции, точки с запятой, переноса строки и даже строки комментариев. Наш синтаксический анализатор просто пропустит весь этот "мусор". Сама очередная найденная строка извлекается вызовом метода `cap(0)`. Наша следующая задача — извлечь из строки вида `"(x, y)"` числа `x` и `y`. Мы делаем это с помощью объекта `numbersExp`. Работа с объектом `numbersExp` принципиально ничем не отличается от работы с объектом `pairsExp`,

только искомыми элементами теперь являются числа, регулярное выражение для которых задано строкой `number`.

Обратите внимание на то, что метод `points()` построен таким образом, что метод `parse()` вызывается только один раз. Если бы это было не так, например, если бы класс `CoordinateParser` предназначался для работы с несколькими разными строками текста, создание объектов класса `QRegExp` было бы лучше перенести в конструктор. Причина в том, что операция создания такого объекта — сравнительно трудоемкий процесс (с точки зрения затрат машинного времени) и создавать один и тот же объект `QRegExp` несколько раз нерационально.

Структура строки, описывающей расположение стен на уровне, является, в некотором роде, потомком структуры, описывающей расположение ящиков. Поэтому и класс, который выполняет синтаксический разбор этой строки, удобно сделать потомком класса `CoordinateParser` (листинг 10.5).

Листинг 10.5. Класс `RectangleParser`

```
class RectangleParser : public CoordinateParser
{
    Q_OBJECT
public:
    explicit RectangleParser(const QString &text, QObject *parent = 0);
protected:
    void parse(const QString &text);
    void parse2(const QString &text);
    QString pair;
};
```

В конструкторе этого класса передается строка вида `"{(x,y),(w,h)},{(x,y),(w,h)}..."`, которую объект класса преобразует в список точек `QPointF`. В отличие от класса `CoordinateParser`, количество точек в этом списке всегда равно четному числу. Первая пара точек соответствует первой стене, вторая — второй и т. д. Почему мы уверены, что количество точек в списке всегда будет четным числом? Регулярное выражение, предназначенное для распознавания конструкции `"{(x,y),(w,h)}"`, гарантирует нам это. Любая другая конструкция будет признана неподходящей и проигнорирована. Как и в случае с классом `CoordinateParser`, блоки вида `"{(x,y),(w,h)}"` могут быть разделены любыми последовательностями символов, а внутренняя запятая между `(x,y)` и `(w,h)` не является обязательной. Рассмотрим реализацию методов класса `RectangleParser` (листинг 10.6).

Листинг 10.6. Методы класса `RectangleParser`

```
RectangleParser::RectangleParser(const QString &text, QObject *parent) :
    CoordinateParser(text, parent)
{
    pair = QString("\\{\\s*%1\\s*,?\\s*%1\\s*\\}").arg(coordinate);
}
```

```
void RectangleParser::parse(const QString &text)
{
    QRegExp quadsExp(pair);
    int pos  = 0;
    while ((pos = quadsExp.indexIn(text, pos)) != -1) {
        parse2(quadsExp.cap(0));
        pos += quadsExp.matchedLength();
    }
}

void RectangleParser::parse2(const QString &text)
{
    QRegExp numbersExp(coordinate);
    int pos  = 0;
    while((pos = numbersExp.indexIn(text, pos)) != -1) {
        CoordinateParser::parse(numbersExp.cap(0));
        pos += numbersExp.matchedLength();
    }
}
```

В конструкторе класса `RectangleParser` мы инициализируем строку `pair`, которая должна содержать регулярное выражение для распознавания конструкции `"{(x, y), (w, h)}"`. Мы просто добавляем в шаблон регулярного выражения для строки `pair` регулярное выражение для строки `coordinate`, унаследованное от базового класса. Выделение элементов из строки в классе `RectangleParser` выполняется в три этапа. Сначала метод `parse()` разбивает строку на блоки `"{(x, y), (w, h)}"`, затем метод `parse2()` разбивает уже эти блоки на блоки `"(a, b)"`, а уже метод `parse()` класса-предка (`CoordinateParser`) выделяет из этих блоков отдельные пары чисел, которые и добавляются в список.

Чтобы продемонстрировать удобство предложенного объектно-ориентированного подхода к построению синтаксических распознавателей, предположим, что мы захотели изменить формат чисел. Заменим имеющуюся строку `number` на `"\\d{1,4}(\\.\\d{1,4})?"`. Такой формат ограничивает количество знаков числа справа и слева от точки не более чем четырьмя знаками. Поскольку строка `number` используется при построении всех анализаторов, данное ограничение будет действовать во всех структурах описания сцены.

Теперь, когда у нас есть классы, выполняющие синтаксический разбор строк с последовательностями координат, поднимемся на уровень выше и рассмотрим класс, который выполняет синтаксический разбор всего файла, описывающего уровень (листинг 10.7).

Листинг 10.7. Класс `SceneReader`

```
class SceneReader : public QObject
{
    Q_OBJECT
```

```
public:
    explicit SceneReader(QObject *parent = 0);
    void read(const QString &fileName);
signals:
    void addWallBlock(QPointF p1, QPointF p2);
    void addBox(QPointF p);
    void addPlayer(QPointF p);
};
```

Интерфейс класса `SceneReader`, который отвечает за распознавание и разбор всего описания уровня, довольно прост. Основная операция выполняется методом `read()`, которому в качестве параметра передается имя файла описания уровня. В процессе выполнения метода `read()` эмитируются сигналы `addBox()`, `addPlayer()` и `addWall()`, которые указывают, что нужно добавить в сцену ящик, фигурку игрока и стену соответственно. Необходимые для выполнения этих операций значения координат передаются в параметрах сигналов. Обращаю ваше внимание на то, что класс `SceneReader` "ничего не знает" о том, каким образом создается сцена игры. В этом смысле модуль разбора файла уровня полностью отделен от остальных модулей программы. Он только сообщает другим модулям, что они должны сделать для создания сцены. Самое интересное в классе `SceneReader` — это, как вы уже поняли, реализация метода `read()` (листинг 10.8).

Листинг 10.8. Метод `read()`

```
void SceneReader::read(const QString &fileName)
{
    QFile f(fileName);
    f.open(QIODevice::ReadOnly | QIODevice::Text);
    QXmlStreamReader reader(&f);
    while (!reader.atEnd()) {
        if (reader.isStartElement()) {
            QString element = reader.name().toString();
            if (element == "walls") {
                QString coordinates = reader.readElementText(
                    QXmlStreamReader::IncludeChildElements);
                RectangleParser parser(coordinates);
                for (int i = 0; i < parser.points()->count(); i+=2) {
                    emit (addWallBlock(parser.points()->at(i),
                        parser.points()->at(i+1)));
                }
            } else
                if (element == "boxes") {
                    QString coordinates = reader.readElementText(
                        QXmlStreamReader::IncludeChildElements);
                    CoordinateParser parser(coordinates);
```

```
                    foreach(QPointF p, *parser.points()) {
                        emit (addBox(p));
                    }
                } else
                    if (element == "player") {
                        QString coordinates = reader.readElementText(
                            QXmlStreamReader::IncludeChildElements);
                        CoordinateParser parser(coordinates);
                        foreach(QPointF p, *parser.points()) {
                            emit (addPlayer(p));
                        }
                    }
        }
        reader.readNext();
    }
}
```

Помимо классов `CoordinateParser` и `RectangleParser`, этот метод использует классы `QFile` и `QXmlStreamReader`. Класс `QXmlStreamReader` реализует один из двух вариантов разбора XML-файла, которые предоставляет библиотека Qt 4.7. В этом варианте мы сами управляем процессом разбора, передвигаясь по файлу XML с помощью метода `readNext()` класса `QXmlStreamReader`. Этот вариант разбора не сохраняет информацию о контексте, в котором мы находимся. В каждый данный момент мы разбираем конкретный элемент документа XML сам по себе. Когда разбор элемента окончен, класс `QXmlStreamReader` "забывает" о нем. В противоположность этому, при использовании для разбора документа XML класса `QDomDocument` и связанных с ним классов, в памяти программы создается модель XML-документа, соответствующая его структуре. Использование `QDomDocument` оправдано в случае, когда документ XML имеет сложную структуру и одни и те же теги XML могут появляться в разных контекстах, причем нам необходимо знать эти контексты. В данном случае мы можем прибегнуть к более простому средству — классу `QXmlStreamReader`.

Если мы находим в начале элемента XML (открывающий тег), метод `isStartElement()` класса `QXmlStreamReader` возвращает значение `true`. В этом случае метод `readElementText()` вернет нам весь текст, находящийся между открывающим и закрывающим тегами. Метод `read()` передает этот текст соответствующему объекту для синтаксического разбора, получает список координат в виде объектов `QPointF` и эмитирует соответствующие события.

Переходим к графике

Для реализации игры нам понадобится создать потомок класса `QGraphicsScene` (листинг 10.9).

Листинг 10.9. Определение класса `MvScene`

```
class MvScene : public QGraphicsScene
{
    Q_OBJECT
```

```
public:
    MvScene(QObject *parent = 0);
protected:
    virtual void mousePressEvent(QGraphicsSceneMouseEvent * mouseEvent);
    virtual void keyPressEvent(QKeyEvent * keyEvent);
private slots:
    void addWall(QPointF p1, QPointF p2);
    void addBox(QPointF p);
    void addPlayer(QPointF p);
private:
    QGraphicsPixmapItem * worker;
    QGraphicsItem * itemCollidesWith(QGraphicsItem * item);
    void placeBox(float x, float y);
};
void ImageFrame::paintEvent(QPaintEvent * e)
{
    DrawersList * dl = (DrawersList * ) drawersList;
    foreach (PictureDrawer * pd, *dl) {
        if ((e->rect().intersects(pd->drawingRect()))  {
            pd->update();
        } else {
            pd->discard();
        }
    }
    QFrame::paintEvent(e);
}
```

В отличие от обычной картинки сцена из игрового мира должна реагировать на действия пользователя. В классе `MvScene` мы перекрываем функции-обработчики событий `mousePressEvent()` и `keyPressEvent()`. Кроме того, в нашем классе реализовано несколько вспомогательных функций-слотов, которые должны обрабатывать сигналы, посылаемые объектом класса `SceneReader` в процессе чтения файла описания уровня и создания сцены (все это происходит в конструкторе класса `MvScene`). Метод `placeBox()` нужен для добавления одного ящика в лабиринт, а метод `itemCollidesWith()` используется для обнаружения столкновений объектов игрового мира.

Конструктор класса `MvScene` (листинг 10.10) позволяет пользователю выбрать файл описания уровня и производит загрузку сцены.

Листинг 10.10. Конструктор класса `MvScene`

```
MvScene::MvScene(QObject *parent) : QGraphicsScene(parent)
{
        SceneReader sceneReader;
        connect(&sceneReader, SIGNAL(addWallBlock(QPointF, QPointF)),
            this, SLOT(addWall(QPointF, QPointF)));
```

```
        connect(&sceneReader, SIGNAL(addBox(QPointF)), this,
            SLOT(addBox(QPointF)));
        connect(&sceneReader, SIGNAL(addPlayer(QPointF)), this,
            SLOT(addPlayer(QPointF)));
        QFileDialog dialog(0, QString::fromUtf8("Открыть файл данных"),
            QString(), QString::fromUtf8("файлы XML (*.xml)"));
        if (dialog.exec())
            sceneReader.read(dialog.selectedFiles().at(0));
        setBackgroundBrush(QBrush(QColor(255,255,255),
             QPixmap(":/bg.jpg")));
}
```

Объект `sceneReader` существует только на протяжении жизни метода-конструктора, но больше он нам и не нужен, если, конечно, мы не захотим добавить возможность загрузки новых уровней, не выходя из игры. Соответственно и слоты (листинг 10.11) класса `MvScene` востребованы только в самом начале работы программы.

Листинг 10.11. Слоты класса `MvScene`

```
void MvScene::placeBox(float x, float y)
{
        QGraphicsItem * box = addPixmap(QPixmap(":/box.gif"));
    box->translate(x, y);
    if (itemCollidesWith(box))
        removeItem(box);
    else
        box->setData(0, "Box");
}

void MvScene::addBox(QPointF p)
{
    placeBox(p.x(), p.y());
}

void MvScene::addWall(QPointF p1, QPointF p2)
{
    QBrush brush(QColor(255, 255, 255), QPixmap(":/wall.jpg"));
    QPen pen(Qt::NoPen);
    QGraphicsItem * item =
    addRect(QRectF(p1.x(), p1.y(), p2.x(), p2.y()), pen, brush);
    item->setData(0, "Wall");
}

void MvScene::addPlayer(QPointF p)
{
    worker = addPixmap(QPixmap(":/Worker.gif"));
```

```
    worker->translate(p.x(), p.y());
    worker->setData(0, "Worker");
}
```

Прямоугольники добавляются в сцену с помощью метода `addRect()`. Начиная с версии Qt 4.4, этот метод доступен в нескольких перегруженных вариантах. Первым аргументом метода `addRect()` является объект `QRectF`, который содержит координаты верхнего левого угла прямоугольника, его ширину и высоту. Второй и третий аргументы — соответственно перо (объект класса `QPen`) и кисть (объект `QBrush`), с помощью которых рисуется прямоугольник. Когда мы рисуем стену, мы хотим, чтобы соответствующий прямоугольник был заполнен определенной текстурой. Все, что нам нужно сделать, — передать методу `addRect()` кисть, заполняющую фигуру узором из файла wall.jpg. Метод `addRect()` возвращает указатель на объект класса `QGraphicsRectItem`, являющегося потомком `QGraphicsItem`.

Ящики и фигурка игрока отличаются от стен тем, что их требуется не заполнить текстурой, а просто нарисовать, т. е. вывести в нужном месте содержимое файла, в котором хранится соответствующее изображение. Для этого мы пользуемся методом `addPixmap()`. Этот метод создает объект класса `QGraphicsPixmapItem` (этот класс тоже является потомком `QGraphicsItem`), специально предназначенного для вывода изображений из объектов `QPixmap`. Ранее было сказано, что объекты `QGraphicsItem` ведут себя как виджеты. Это значит, что при перемещении объекта `QGraphicsPixmapItem` по сцене нам не нужно беспокоиться ни о его перерисовке, ни об уничтожении изображения на прежнем месте. Заботу обо всем этом система берет на себя. С точки зрения Graphics View Framework объект класса `QGraphicsPixmapItem` является прямоугольником, высота и ширина которого задаются соответствующими параметрами изображения (с учетом преобразования масштаба, если таковые заданы), а положение определяется по верхнему левому углу с помощью таких методов класса `QGraphicsScene` как `translate()` или `rotate()`.

Рассмотрим подробнее метод `setData()` класса `QGraphicsItem`. Помимо графических свойств, таких как координаты и параметры кисти и пера, примитивы Graphics View Framework могут быть наделены дополнительными свойствами, определяющими их поведение в модели данных. Мы можем наделить объекты дополнительными свойствами, создавая новые классы на базе классов графических примитивов, но Graphics View Framework предлагает нам и более простой путь. Каждый объект класса-потомка `QGrapihcsItem` является контейнером, в который можно добавлять произвольные данные. Именно это и делает метод `setData()`. Первым аргументом метода является численный идентификатор элемента данных (ключ), вторым аргументом — сами данные, представленные в виде значения типа `QVariant`. В нашей программе мы добавляем в каждый графический примитив один дополнительный элемент данных с ключом 0 и строковым значением. В строке записывается название предмета, которому соответствует данный примитив — стена (`Wall`) или ящик (`Box`). Эта информация понадобится нам для ответа на вопрос, как грузчик (объект `worker`) должен реагировать на столкновение с соответствующим примитивом.

Перейдем теперь к методу `keyPressEvent()`, который является движущей силой всей нашей игры (листинг 10.12).

Листинг 10.12. Реализация метода `keyPressEvent()`

```
void MvScene::keyPressEvent(QKeyEvent * keyEvent)
{
    QPointF np;
    np.setX(0);
    np.setY(0);
    switch (keyEvent->key()) {
        case Qt::Key_Left:
            np.setX(-10);
            break;
        case Qt::Key_Right:
            np.setX(10);
            break;
        case Qt::Key_Up:
            np.setY(-10);
            break;
        case Qt::Key_Down:
            np.setY(10);
            break;
    }
    worker->translate(np.x(), np.y());
    QGraphicsItem * obstacle = itemCollidesWith(worker);
    if (obstacle) {
        if (obstacle->data(0) == "Wall") {
            worker->translate(-np.x(), -np.y());
            printf("Здравствуй, стена!\n");
        }
        else
        if (obstacle->data(0) == "Box") {
            obstacle->translate(np.x(), np.y());
            if (itemCollidesWith(obstacle) || itemCollidesWith(worker)) {
                obstacle->translate(-np.x(), -np.y());
                worker->translate(-np.x(), -np.y());
                printf("Слишком тяжело!\n");
            }
        }
    }
}
```

В этом методе мы решаем несколько задач: перемещаем грузчика по игровому полю в направлении, заданном нажатой клавишей (для управления грузчиком используются клавиши со стрелками), выявляем столкновения грузчика с предметами игрового мира и обрабатываем эти столкновения согласно правилам игры. Перемещение грузчика по сцене выполняется с помощью метода `translate()` класса `QGraphicsItem`. Этот метод, наряду с методами `rotate()` и `scale()`, входит в базовый

интерфейс геометрических преобразований Graphics View Framework. Для того чтобы понять, как работают эти методы, нужно вернуться к описанию различных систем координат, которые используются в графической системе Qt 4. Методы, выполняющие геометрические преобразования примитива, работают в системе координат примитива. Особенность этой системы координат заключается в том, что координаты примитива в ней никогда не меняются. Иначе говоря, при переносе, вращении и масштабировании примитива его система координат также подвергается переносу, вращению и масштабированию относительно других систем координат. Например, после поворота примитива на 60 градусов оси его системы координат также будут повернуты на 60 градусов и, в результате, перенос примитива вдоль одной из осей будет выполняться под углом к границе экрана. Начиная с версии Qt 4.3, у класса `QGraphicsItem` появились методы, позволяющие напрямую манипулировать матрицей преобразований (мы рассмотрим эти методы в разделе, посвященном встраиваемым виджетам). При таких сложных отношениях между системами координат функции, предназначенные для перевода значений из одной системы координат в другую, играют особую роль. Метод `mapToScene()` класса `QGraphicsItem` выполняет перевод значений из системы координат примитива в систему координат сцены, а метод `mapToItem()` — перевод из системы координат сцены в систему координат примитива.

Вернемся к нашей программе. Мы перемещаем грузчика в новую позицию и с помощью вспомогательной функции `itemCollidesWith()` проверяем, не столкнулся ли он с другим примитивом. Если грузчик натолкнулся на стену, мы просто возвращаем его в исходную позицию. Если препятствием оказался ящик, мы перемещаем ящик в новую позицию и проверяем, не натолкнулся ли ящик на препятствие. После перемещения ящика мы также проверяем, не сталкивается ли грузчик еще с каким-либо препятствием. В классическом "Сокобане" грузчик, ящик и сегмент стены имеют одинаковые размеры, поэтому в каждый момент времени грузчик может столкнуться только с одним препятствием. Наш вариант в этом смысле сложнее, поскольку все объекты имеют разные размеры, и грузчик может натолкнуться на ящик и стену одновременно. Если хотя бы одно из перечисленных выше условий не выполнено, и грузчик, и ящик, который он сдвинул, возвращаются в исходные позиции. Поскольку перерисовка сцены выполняется только после выхода из метода, пользователь не увидит всех этих пробных перемещений.

Обнаружение столкновений в нашей игре выполняет вспомогательная функция `itemCollidesWith()` (листинг 10.13).

Листинг 10.13. Реализация метода, обнаруживающего столкновения

```
QGraphicsItem * MvScene::itemCollidesWith(QGraphicsItem * item)
{
    QList<QGraphicsItem *> collisions = collidingItems(item);
    foreach (QGraphicsItem * it, collisions) {
            if (it == item)
                 continue;
```

```
        return it;
    }
    return NULL;
}
```

Функция возвращает первый примитив, с которым столкнулся проверяемый примитив, или NULL, если проверяемый примитив ни с чем не столкнулся. В основе функции лежит метод collidingItems() класса QGraphicsScene. Этот метод возвращает список примитивов, находящихся в состоянии столкновения с примитивом, переданным методу в качестве параметра (под столкновением понимается частичное или полное перекрытие примитивов в системе координат сцены). Список, возвращаемый методом collidingItems(), никогда не бывает пустым. В нем всегда содержится, как минимум, один примитив — тот, который мы проверяем на столкновения. С точки зрения графической системы примитив всегда сталкивается с самим собой. Любители философской диалектики могут увидеть в этом глубокий смысл, нам же при обнаружении столкновения просто приходится пропускать один из элементов списка.

Наша программа обрабатывает также щелчки мыши. Вообще-то в игре "Сокобан" мыши делать нечего, но в нашем варианте щелчок левой кнопкой мыши позволяет добавить ящик в лабиринт, а щелчок правой кнопкой — удалить уже существующий ящик. С помощью метода itemAt() класса QGraphicsScene можно проверить, попал ли указатель в какой-нибудь графический примитив (в этом случае метод itemAt() возвращает указатель на соответствующий объект). В качестве аргумента методу itemAt() передаются координаты указателя мыши в системе сцены. Координаты указателя мыши в системе координат сцены мы можем получить с помощью метода scenePos() объекта mouseEvent (указатель на этот объект передается методу-обработчику события мыши mousePressEvent()). Помимо метода itemAt() у нас есть еще один способ заставить сцену реагировать на события мыши. Мы можем назначать собственные обработчики событий мыши графическим примитивам (объектам QGraphicsItem). Благодаря системе Event propagation обработчик будет вызываться только в том случае, если указатель мыши попал в соответствующий примитив, однако этот механизм лучше оставить для работы с примитивами-виджетами.

У двухмерных примитивов Graphics View Framework есть и третья координата — z. Эта координата определяет, какой из примитивов будет виден, если несколько примитивов частично или полностью перекрываются. Кроме того, от значения третьей координаты зависит порядок, в котором располагаются примитивы в списке, возвращаемом методом collidingItems() (первым в этом наборе располагается примитив с наименьшим значением z). Если данный примитив полностью скрыт другим примитивом с более высоким значением z, метод isObscured() возвращает значение true. Изменить значение координаты z графического примитива можно с помощью метода setZValue() класса QGraphicsItem.

Встраивание виджетов

Начиная с версии Qt 4.4, система Graphics View Framework обогатилась еще одной весьма интересной возможностью. Речь идет о встраивании виджетов в графическую сцену. В Qt 4.4 у класса `QGraphicsScene` появился метод `addWidget()`, который позволяет добавлять в сцену виджеты как обычные графические примитивы. Виджеты, встроенные в графическую сцену, не теряют своей функциональности. Благодаря механизму передачи событий Graphics View Framework встроенные виджеты реагируют на действия пользователя точно так же, как и их обычные собратья. Впрочем, некоторые отличия в поведении встроенных виджетов все-таки присутствуют. Например, диалоговое окно, встроенное в графическую сцену, будет вести себя не совсем так, как независимое диалоговое окно. Одновременно с этим встроенные виджеты обладают свойствами графических примитивов Graphics View Framework. Со встроенными виджетами можно выполнять те же геометрические преобразования, что и с остальными примитивами, для них также работает функция обнаружения столкновений и другие функции графической системы. Встраивание виджетов является логическим развитием одной из основных идей Graphics View Framework — использования возможностей двухмерной графики для построения сложных пользовательских интерфейсов. Дальнейшим развитием этой идеи являются разработка класса `QGraphicsWidget` и системы Qt Declarative и языка QML, а также появление в Qt 4.6 класса `QGraphicsWebView`.

В то же время с помощью встраивания виджетов можно создать интерфейсы, которые будут выглядеть, мягко говоря, необычно, как, например, в программе crasyiface (рис. 10.3).

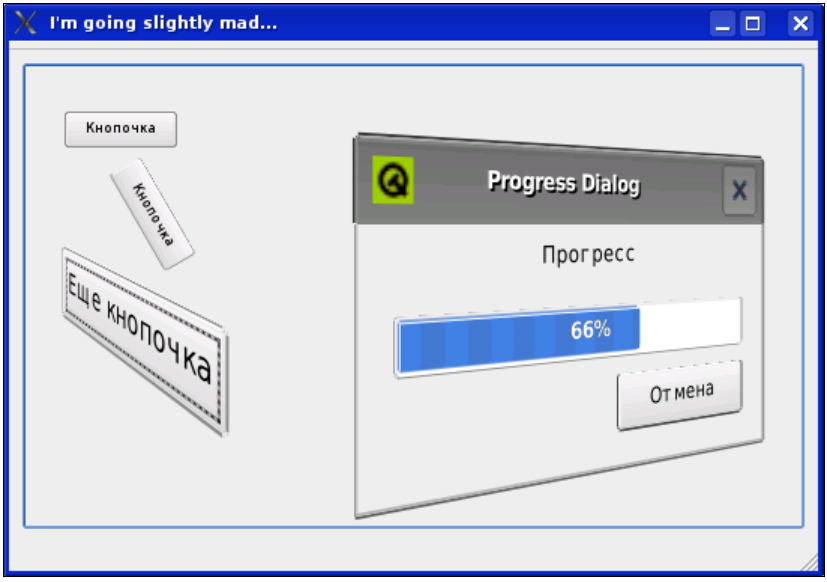


Рис. 10.3. Необычный интерфейс пользователя, созданный средствами Qt

Рассмотрим фрагмент конструктора объекта-сцены программы (листинг 10.14).

Листинг 10.14. Фрагмент конструктора сцены с встроенными виджетами

```
QPushButton * button = new QPushButton(trUtf8("Кнопочка"), 0);
    QGraphicsProxyWidget * item = addWidget(button);
    button->show();
    button = new QPushButton(trUtf8("Кнопочка"), 0);
    item = addWidget(button);
    button->show();
    QTransform transform = item->transform();
    transform.translate(50., 30.);
    transform.rotate(60.0);
    item->setTransform(transform);
    button = new QPushButton(trUtf8("Еще кнопочка"), 0);
    item = addWidget(button);
    button->show();
    transform = item->transform();
    transform.rotate(80.0, Qt::YAxis);
    transform.translate(-10., 90.);
    transform.scale(5., 2.);
    item->setTransform(transform);
    QProgressDialog * dialog = new QProgressDialog(trUtf8("Прогресс"),
                                        trUtf8("Отмена"), 0, 100);
    dialog->setWindowTitle(trUtf8("Progress Dialog"));
    item = addWidget(dialog);
    dialog->show();
    dialog->setValue(66);
    transform = item->transform();
    transform.translate(200., 75.);
    transform.rotate(-45.0, Qt::YAxis);
    transform.scale(2.5, 2.);
    item->setTransform(transform);
```

Для того чтобы добавить виджет в графическую сцену, мы сначала создаем объект соответствующего виджету класса, а потом вызываем метод `addWidget()`. Метод `addWidget()` класса `QGraphicsScene` возвращает указатель на объект класса `QGraphicsProxyWidget`. Этот класс является отдаленным потомком класса `QGraphicsItem` и представляет встроенный виджет в графической сцене. Для выполнения геометрических преобразований виджета мы воспользуемся матрицей преобразований, которая появилась еще в Qt 4.3. Матрица может быть создана многими способами (да, Нео, это так). Мы получаем ссылку на объект, инкапсулирующий матрицу (объект класса `QTransform`) с помощью метода `transform()` объекта класса `QGraphicsProxyWidget`. У класса `QTransform` есть методы `translate()`, `rotate()` и `scale()`, которые работают не совсем так, как одноименные методы класса `QGraphicsItem`. При вызове метода `rotate()` мы, помимо угла поворота, можем ука-

зать ось, вокруг которой должно выполняться вращение. Вращать примитивы можно не только вокруг оси `z` (что соответствует вращению в плоскости `x-y`), но и вокруг осей `x` и `y`. В результате графической сцене можно придать трехмерный вид. Разумеется, это не настоящая "трехмерность", т. к. координата `z` не является по-настоящему независимой. После того как мы внесли изменения в матрицу преобразований, мы снова назначаем эту матрицу примитиву с помощью метода `setTransform()`.

Обратите внимание на то, что виджеты, встроенные в окно программы crasyiface, сохраняют свою функциональность. Кнопки реагируют на щелчки мыши, а встроенное диалоговое окно можно даже закрыть, щелкнув соответствующую кнопку в его заголовке.

Использование встроенных виджетов в качестве элементов управления

Разумеется, встраивание виджетов было введено в Qt не ради баловства. Встроенные виджеты могут выполнять в Graphics View Framework две функции: они могут быть частью графической сцены и при этом использоваться либо как полнофункциональные виджеты, либо как детальные изображения виджетов (например, в схеме, которая включает в себя изображения элементов пользовательского интерфейса), либо как самостоятельные элементы управления, которые не имеют прямого отношения к остальному содержимому графической сцены. Мы рассмотрим встраиваемые виджеты именно во втором качестве. Дело в том, что объединение широких графических возможностей Graphics View Framework и функциональности традиционных виджетов позволяет нам разрабатывать необычные и удобные виджеты, затратив на это гораздо меньше времени по сравнению с ситуацией, в которой мы создавали бы эти виджеты с нуля.

В качестве примера рассмотрим виджет, ориентированный на презентационные интерфейсы. Интерфейсы программ можно условно разделить на две группы: те, которые предназначены для предоставления пользователю возможности вводить данные либо управлять чем-то, и те, главная задача которых показывать пользователю содержимое некоего информационного ресурса. Интерфейсы второго типа мы и называем презентационными. К типичным примерам программ, которые нуждаются в презентационных интерфейсах, можно отнести видеоплееры и Web-браузеры. Проектировщикам интерфейсов презентационных приложений приходится решать непростую задачу. С одной стороны, желательно отдать максимальную область окна программы для показа информационного содержимого. С другой стороны, презентационная программа все же нуждается в некоторых элементах управления и, в соответствии с принципом самоочевидности интерфейса, эти элементы управления не должны быть спрятаны от пользователя. Иначе говоря, интерфейс, в котором для появления элементов управления на экране приходится набирать сложную комбинацию клавиш или проделывать сложные движения мышью, нельзя назвать удачным.

Наиболее популярное сегодня решение сводится к тому, чтобы, во-первых, оставлять на виду только часть элементов управления (остальная часть появится, если пользователь наведет указатель мыши на то, что видно), а, во-вторых, к тому, чтобы сделать элементы управления полупрозрачными, так, чтобы, даже будучи видимы, они не скрывали полностью заслоненную ими информационную часть. Чтобы наглядно убедиться, в каком направлении движется эволюция интерфейсов презентационных программ, сравните интерфейс Internet Explorer 6, выпущенный в 2001 году (рис. 10.4), и интерфейс Internet Explorer 9, выпущенный в 2011 году (рис. 10.5).

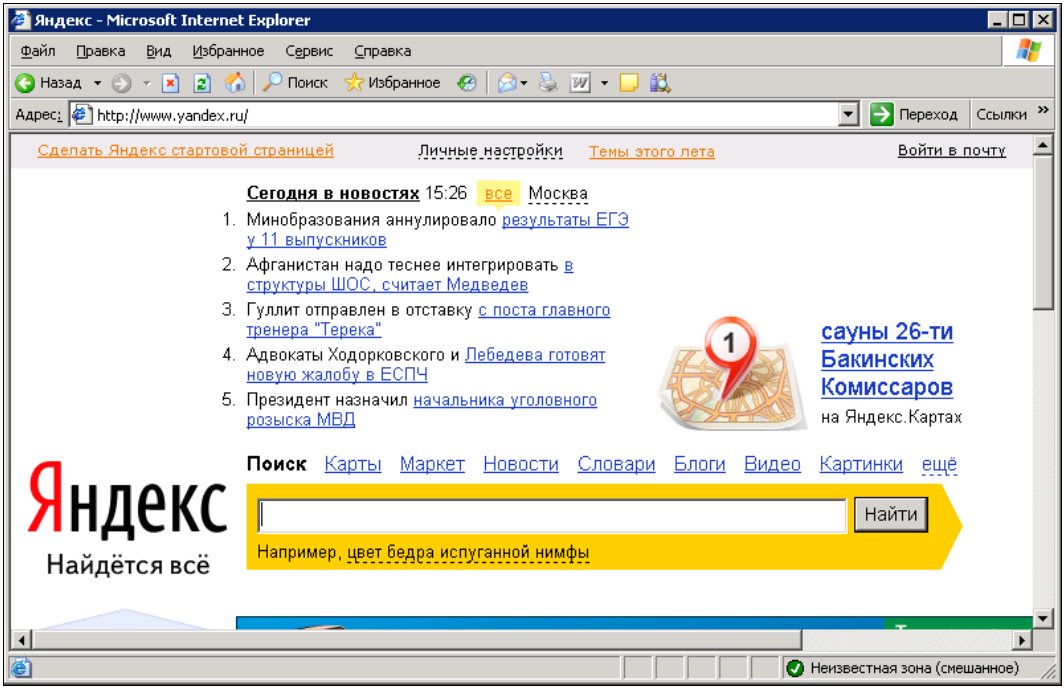


Рис. 10.4. Окно браузера Internet Explorer 6

Разумеется, рост производительности видеокарт тоже сыграл в эволюции интерфейсов свою роль. Стандартной "офисной" видеокарте 2001 года было бы трудновато поддерживать множество полупрозрачных виджетов, меняющих свой цвет в зависимости от движений мыши.

Следуя генеральной линии, мы создадим виджет для презентационной программы — полупрозрачную панель с кнопками, прозрачность которой зависит от того, находится ли указатель мыши в области панели или вне ее.

Создание нового виджета начнем с визуального программирования (рис. 10.6).

В качестве основы можно использовать обычное окно `QWidget` или `QDialog`, в зависимости от того, что нам больше подходит. Мы хотим, чтобы панель меняла прозрачность в зависимости от того, находится ли над ней указатель мыши. Первое, что приходит в голову, — начать отслеживать события мыши, которые посылаются

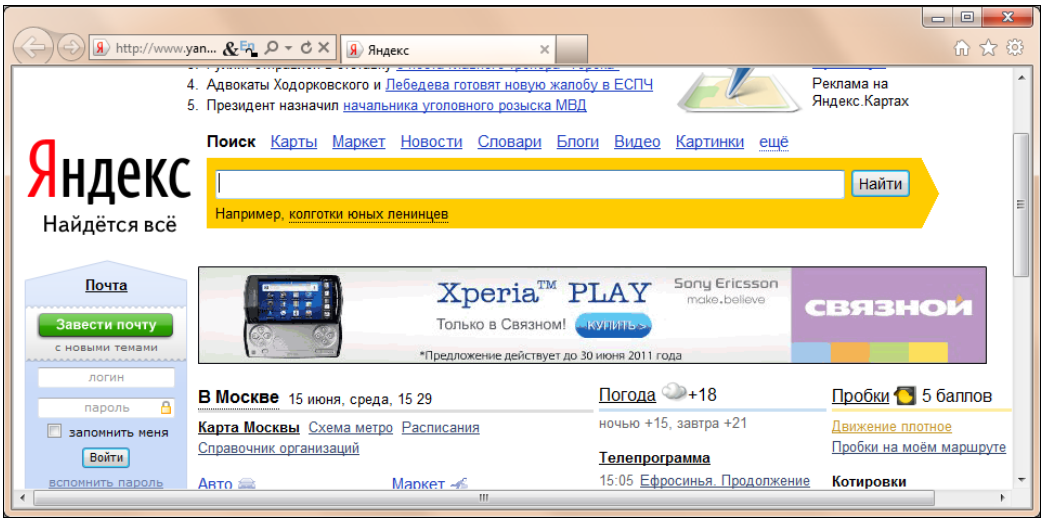


Рис. 10.5. Окно браузера Internet Explorer 9

Рис. 10.6. Визуальная основа виджета

окну, когда указатель мыши находится над ним (событие `QMouseMove`). Главная сложность, связанная с этим подходом, заключается в том, что очень трудно определить момент, когда указатель покидает область окна. Можно применять разные трюки и хитрости, но при достаточно быстром "скачке" указателя за пределы окна все они могут не сработать (и виджет по-прежнему будет выглядеть так, словно указатель находится над ним). Поэтому мы пойдем другим путем (которым, кстати, нередко идут разработчики профессиональных виджетов) — мы будем периодически (с помощью таймера) проверять, находится ли указатель мыши над окном. Недостаток этого подхода состоит в том, что окно отреагирует на перемещение мыши не сразу. Но есть у него и свои достоинства. Использование таймера позволит нам организовать эффект анимации, когда непрозрачность окна включается и отключается постепенно. Итак, посмотрим, как выглядит класс виджета, который, по сути, реализует обычно диалоговое окно с кнопками (листинг 10.15). Полный исходный текст вы найдете в папке Ch10/extwidgets.

Листинг 10.15. Класс виджета

```
class Dialog : public QDialog
{
    Q_OBJECT
public:
    explicit Dialog(QWidget *parent = 0);
    ~Dialog();
```

```
signals:
    void zoomIn();
    void zoomOut();
protected:
    void timerEvent(QTimerEvent *);
private:
    Ui::Dialog *ui;
};

void Dialog::timerEvent(QTimerEvent *)
{
    if (underMouse()) {
        if (windowOpacity() < 0.7)
                setWindowOpacity(windowOpacity() + 0.1);
        QList<QToolButton *> l = findChildren<QToolButton *>();
        foreach(QToolButton * b, l) {
            b->setIconSize(QSize(32,32));
        }
    } else {
        if (windowOpacity() > 0.4)
                setWindowOpacity(windowOpacity() — 0.1);
        QList<QToolButton *> l = findChildren<QToolButton *>();
        foreach(QToolButton * b, l) {
           b->setIconSize(QSize(24,24));
        }
    }
}
```

Из всех методов класса `Dialog` мы рассмотрим только `timerEvent()`, в котором и происходит самое интересное. Мы проверяем, находится ли указатель мыши в области окна, с помощью метода `underMouse()`, введенного в классе `QWidget`. Если указатель мыши в данный момент находится в области окна, мы уменьшаем прозрачность окна с помощью методов `windowOpacity()` и `setWindowOpacity()`. Кроме того, мы изменяем размеры пиктограмм на кнопках (в результате чего изменятся размеры и самих кнопок). Таким образом, когда указатель мыши находится над окном, оно постепенно становится менее прозрачным, а размер кнопок увеличивается. Когда указатель покидает область окна, происходит обратный процесс.

В принципе, наш виджет готов к тому, чтобы мы встроили его в графическую сцену, но поскольку мы хотим, чтобы этот виджет вел себя как обычный элемент управления, то придется предпринять некоторые дополнительные действия. По умолчанию виджеты, встроенные в графическую сцену, ведут себя так же, как и обычные графические примитивы (что и было показано ранее). Однако если мы задумали использовать встроенный виджет как элемент управления, мы, скорее всего, захотим, чтобы виджет вел себя не как обычный графический примитив. Это желательно еще и потому, что позволит пользователю программы понять, что виджет является общим элементом управления, а не частью сцены. Наш виджет не

должен реагировать на такие преобразования сцены, как масштабирование и вращение, не должен исчезать из видимой части сцены при ее прокрутке в окне. Можно, конечно, не делать всего этого, а в документации к виджету написать, что при масштабировании или вращении сцены графический примитив, содержащий наш виджет, не должен подвергаться этой процедуре, но лучше все же не перекладывать эту задачу на плечи программиста, использующего виджет, а автоматизировать ее. Для реализации всего этого нам понадобятся два вспомогательных класса.

Класс `QXtGraphicsProxyWidget` (листинг 10.16) является потомком класса `QGraphicsProxyWidget` и блокирует некоторые аспекты стандартного поведения графического примитива.

Листинг 10.16. Класс `QXtGraphicsProxyWidget`

```
class QXtGraphicsView;
class QXtGraphicsProxyWidget : public QGraphicsProxyWidget
{
    Q_OBJECT
public:
    QXtGraphicsProxyWidget(QGraphicsItem * parent = 0,
        Qt::WindowFlags wFlags = 0);
    void setView(QXtGraphicsView * view);
public slots:
    void viewScrolled();
protected:
    QVariant itemChange(GraphicsItemChange change,
        const QVariant & value);
private:
    QXtGraphicsView * mview;
};
```

Одна из задач этого класса, который мы будем использовать для встраивания виджета вместо `QGraphicsProxyWidget`, заключается в том, чтобы заблокировать геометрические преобразования, которые могут быть применены к виджету. Рассмотрим конструктор класса `QXtGraphicsProxyWidget` и его метод `itemChange()` (листинг 10.17).

Листинг 10.17. Методы класса `QXtGraphicsProxyWidget`

```
QXtGraphicsProxyWidget::QXtGraphicsProxyWidget(QGraphicsItem * parent,
    Qt::WindowFlags wFlags) : QGraphicsProxyWidget(parent, wFlags)
{
    setFlag(QGraphicsItem::ItemSendsGeometryChanges, true);
}

QVariant QXtGraphicsProxyWidget::itemChange(GraphicsItemChange change,
    const QVariant & value)
```

```
{
    QVariant v;
    switch (change) {
    case QGraphicsItem::ItemScaleChange:
        v = QVariant(1.0);
        return v;
    case QGraphicsItem::ItemRotationChange:
        v = QVariant(0.0);
        return v;
    case QGraphicsItem::ItemTransformChange:
            v = QVariant(matrix());
            return v;
    default:
            return value;
    }
}
```

По умолчанию графический примитив даже "не знает" о том, что к нему применяются геометрические преобразования. Для того чтобы изменить это, в конструкторе класса `QXtGraphicsProxyWidget` мы устанавливаем значение флага `QGraphicsItem::ItemSendsGeometryChanges` равным `true`. В результате перед любым изменением внешнего вида примитива система будет вызывать его виртуальный метод `itemChange()` и фактические изменения будут зависеть от результата, возвращенного этим методом. По умолчанию этот механизм отключен с целью повышения быстродействия.

У метода `itemChange()` два параметра. Первый параметр, имеющий перечислимый тип `GraphicsItemChange`, указывает, какие именно преобразования примитива собирается выполнить система. Второй параметр, принадлежащий типу `QVariant`, передает параметры преобразования. Метод `itemChange()` возвращает значение типа `QVariant`, которое содержит (возможно, измененные) параметры преобразования. Для того чтобы все преобразования выполнялись без изменений, будет достаточно, если метод `itemChange()` возвратит переданное ему значение `value` (но тогда зачем вообще активировать этот метод?). В нашем варианте метода `itemChange()` мы модифицируем аргументы некоторых (фактически самых важных) преобразований.

Если метод `itemChange()` вызван со значением параметра `change`, равным `QGraphicsItem::ItemScaleChange`, значит, система собирается масштабировать графический примитив. В этом случае параметр `value` содержит коэффициент масштабирования, а значение, возвращаемое методом `itemChange()`, должно содержать "исправленный" коэффициент. Наш метод в этом случае возвращает единицу, что, фактически, отменяет масштабирование примитива-виджета. Если в параметре `change` передано значение `QGraphicsItem::ItemRotationChange`, значит, целевое преобразование — вращение, а в параметре `value` передается угол поворота. Возвращая значение 0, наш метод отменяет вращение. Наконец, если параметр `change` содержит значение `QGraphicsItem::ItemTransformChange`, значит, преобразованию подвергается непосредственно матрица геометрических преобразований объекта.

Как было сказано ранее, этим способом можно выполнить самые сложные преобразования. В параметре `value` в этом случае передается новая матрица преобразований, а для того чтобы отменить их, методу `itemChange()` следует вернуть прежнюю матрицу, которую можно получить с помощью метода `matrix()`. Вы можете подумать, что, поскольку все преобразования так или иначе затрагивают матрицу, достаточно заблокировать только вызов, модифицирущий матрицу, но это не так. Во всех остальных случаях метод `itemChange()` никак не модифицирует параметры преобразований.

Итак, если для встраивания виджета в сцену воспользоваться объектом класса `QXtGraphicsProxyWidget`, виджет не будет подвергаться масштабированию или вращению вместе с остальными элементами сцены. Но перед нами стоит еще одна задача: сделать так, чтобы виджет не перемещался в процессе прокрутки сцены в окне показа. Решить эту задачу несколько труднее, чем предыдущую, поскольку в данном случае не только объект класса `QGraphicsItem` (или его потомка) "не знает" о том, что происходит прокрутка, как правило, этого "не знает" и объект класса `QGraphicsScene`. Да иначе и быть не может, ведь прокрутку выполняет объект класса `QGraphicsView` или его потомка, а одна графическая сцена может быть связана с несколькими такими объектами, и в каждом из них прокрутка будет выполняться независимо. Для решения возникшей перед нами задачи мы примем одно допущение: графическая сцена связана только с одним объектом отображения (в большинстве программ это так и есть). Необходимо, чтобы объект, выполняющий отображение сцены, сообщал графическому примитиву, содержащему виджет, о том, когда происходит прокрутка. Класс `QGraphicsView` не посылает при прокрутке никаких сигналов, так что нам придется его модифицировать, создав класс-потомок (листинг 10.18).

Листинг 10.18. Класс `QXtGraphicsView`

```
class QXtGraphicsView : public QGraphicsView
{
    Q_OBJECT
public:
    QXtGraphicsView(QWidget * parent = 0):QGraphicsView(parent)
    {}
signals:
    void scrolled();
protected:
    void scrollContentsBy (int dx, int dy)
    {
        QGraphicsView::scrollContentsBy(dx, dy);
        emit scrolled();
    }
};
```

Здесь все очень просто. Мы вводим в класс сигнал `scrolled()` и перекрываем метод `scrollContentsBy()` (унаследованный классом `QGraphicsView` от класса

QAbstractScrollArea), таким образом, чтобы он посылал сигнал scrolled(). Сам метод scrollContentsBy() виртуальный и вызывается всякий раз, когда окно выполняет прокрутку.

Теперь, когда объект класса QXtGraphicsProxyWidget (или любой другой объект) может получать информацию о прокрутке, надо сделать так, чтобы он действительно получал и обрабатывал ее. Рассмотрим два оставшихся метода класса QXtGraphicsProxyWidget (листинг 10.19).

Листинг 10.19. Методы класса QXtGraphicsProxyWidget (продолжение)

```
void QXtGraphicsProxyWidget::setView(QXtGraphicsView * view)
{
    setPos(0,0);
    mview = view;
    connect(mview, SIGNAL(scrolled()), this, SLOT(viewScrolled()));
}

void QXtGraphicsProxyWidget::viewScrolled()
{
    QPointF p = mview->mapToScene(0,0);
    setPos(p.x(), p.y());
}
```

Метод setView() связывает графический примитив QXtGraphicsProxyWidget с объектом, выполняющим отображение информации. Это, конечно, нарушает стройную модель Graphics View Framework, где представление данных отделено от остальных данных, но другого выхода у нас нет. Кроме того, мы ведь создаем элемент управления, и вполне естественно, что он будет связан не только с моделью данных, но и с объектом, который осуществляет их отображение. После вызова метода setView() при каждой прокрутке окна будет вызываться метод-слот viewScrolled() объекта QXtGraphicsProxyWidget. С помощью метода mapToScene() этот метод получает точку сцены, которая в данный момент совпадает с левым верхним углом окна отображения. Затем с помощью метода setPos() объект QXtGraphicsProxyWidget просто переносится в эту точку. В этом случае, как и в некоторых других, важно не запутаться, какой метод работает в какой системе координат. Как мы уже знаем, метод setPos() позволяет задать положение графического примитива в системе координат графической сцены. Если мы хотим, чтобы примитив был привязан к определенной точке окна, мы должны найти точку сцены, соответствующую этой точке окна.

А что будет, если программист, использующий наш виджет, не вызовет метод setView(), например, в том случае, если для отображения сцены используется объект класса QGraphicsView, а не QXtGraphicsView?

Примечание

На всякий случай напоминаю, что преобразовать виджет QGraphicsView в QXtGraphicsView очень просто. Это можно сделать в окне визуального редактора форм Qt.

Ничего страшного в этом случае не произойдет. Просто наш виджет будет прокручиваться вместе с остальным содержимым сцены. В некоторых ситуациях это даже лучше.

Почему для нас так важно сделать панель кнопок быстрого доступа частью графической сцены Graphics View Framework? Потому что это соответствует еще одной тенденции развития Qt — использованию окна графической сцены в качестве главного окна приложения, в котором содержатся все необходимые приложению визуальные элементы. В результате мы получаем полупрозрачный виджет в современном стиле (рис. 10.7), который при наведении на него указателя мыши станет менее прозрачным (рис. 10.8). Если бы мы использовали в качестве главного окна сам объект просмотра графических сцен, приложение получилось бы еще более "воздушным".

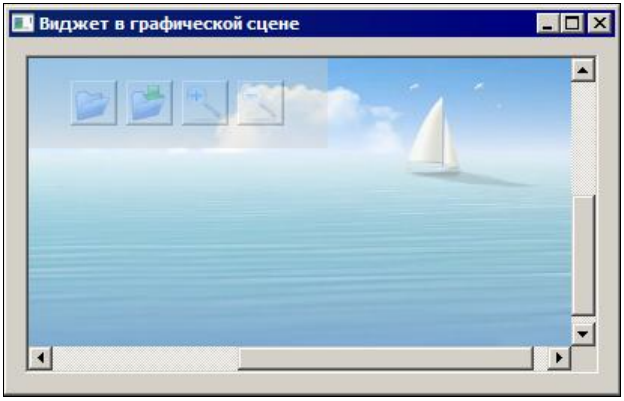


Рис. 10.7. Полупрозрачный виджет в состоянии покоя

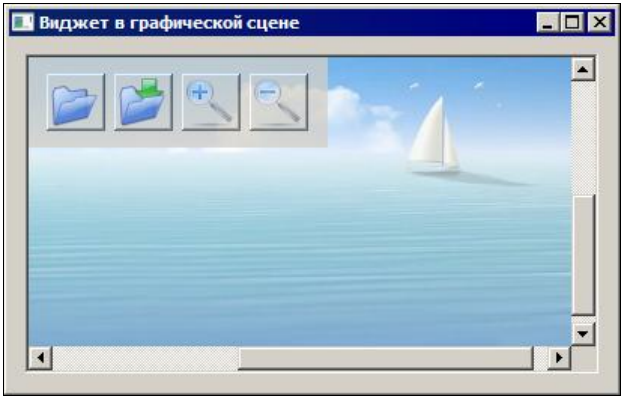


Рис. 10.8. Полупрозрачный виджет под указателем мыши

В общем и целом последовательность действий при встраивании нашего виджета выглядит так: сначала мы связываем объект `QGraphicsScene` с объектом `QXtGraphicsView` или `QGraphicsView` (с помощью метода `setScene()`). Здесь важно, чтобы еще до внедрения виджета объект, содержащий данные графической сцены,

и объект, выполняющий отображение данных, были связаны между собой. После этого мы создаем объект класса `QXtGraphicsProxyWidget` и с помощью метода `setWidget()` связываем его с виджетом, который он должен представлять. Важно, чтобы в качестве родительского объекта этого виджета не был указан какой-либо визуальный объект программы, иначе этот "родитель" попытается управлять виджетом, а нам это не нужно. Лучше всего в конструкторе виджета в параметре `parent` передать 0. Затем мы связываем объект `QXtGraphicsProxyWidget` с объектом-сценой с помощью метода `addItem()` класса `QGraphicsScene`, после чего связываем этот же объект класса `QXtGraphicsProxyWidget` с объектом класса `QXtGraphicsProxyWidget` с помощью метода `setView()` этого класса.

Виджет для выбора фрагментов изображений

Иногда всю графическую сцену полезно превратить в виджет. В качестве примера мы рассмотрим виджет, который позволяет пользователю выбирать прямоугольные фрагменты изображения с помощью мыши для последующих манипуляций. Идея этого виджета настолько очевидна, что остается только удивляться тому, что в самой библиотеке Qt до сих пор не появилось ничего подобного. Вам может показаться, что использовать мощь Graphics View Framework для создания такого виджета — это слишком расточительно. Но на самом деле ничто менее мощное, чем Graphics View Framework, не сможет удовлетворительно решить эту задачу (в рамках Qt, разумеется). Мы все привыкли к тому, что когда в графической программе мы выделяем фрагмент изображения, полупрозрачный (или иной) прямоугольник, отмечающий область выделения, послушно следует за указателем мыши. Именно это поведение довольно трудно реализовать на основе стандартных виджетов Qt, таких как `QLabel`, особенно если требуется высокая производительность. Корень проблемы заключается в том, что при использовании стандартных средств прямоугольник, отмечающий выделенную область, должен непосредственно взаимодействовать с базовым изображением, изменяя цвет его пикселов и затем восстанавливая их обратно.

> ***Примечание***
>
> Для решения таких задач часто применяется инвертирующая кисть, которая изменяет цвет заполняемых пикселов на инвертированный (дополнительный). Для восстановления прежнего цвета пикселов достаточно применить к ним инвертирующую кисть еще раз. Хранить информацию о цвете специально не требуется. В Qt подобного эффекта можно добиться с помощью свойства `CompositionMode` класса `QPainter`.

При использовании Graphics View Framework решение этой задачи оказывается неожиданно простым. Прямоугольник, обозначающий выделенную область, можно реализовать с помощью отдельного графического примитива, который вообще не будет взаимодействовать с выделяемым изображением. А уж о том, чтобы соответствующие операции выполнялись достаточно быстро, разработчики Graphics View Framework позаботились. Можно пойти дальше и использовать дополнительные

возможности Graphics View Framework для расширения функциональности виджета. Можно предоставить пользователю возможность выделять несколько блоков, и использовать встроенный в Graphics View Framework механизм определения столкновений для того, чтобы исключить пересечение этих блоков. Наш виджет позволяет выбрать в области изображения несколько прямоугольных непересекающихся блоков. У пользователя есть также возможность изменить границы уже выбранного блока. Кроме того, пользователь может изменять масштаб сцены. Особенности той программы, из которой был взят этот виджет, не позволяет выбирать масштаб более чем 1:1, хотя можно уменьшать масштаб. Если вам потребуется изменить поведение виджета в отношении масштабирования, вы без труда сможете сделать это.

Класс `GraphicsInput` (листинг 10.20), в котором реализована описанная функциональность и некоторые другие возможности, является потомком класса `QGraphicsScene`.

Листинг 10.20. Класс `GraphicsInput`

```
class GraphicsInput : public QGraphicsScene
{
    Q_OBJECT
public:
    explicit GraphicsInput(const QRectF &sceneRect,
        QGraphicsView *view = 0);
    ~GraphicsInput();
    bool loadImage(const QPixmap &image, bool clearBlocks = true);
    bool loadNewImage(const QPixmap &image);
    void setView(QGraphicsView *view);
    QPixmap getActiveBlock();
    QPixmap getCurrentBlock();
    void setViewScale(qreal scale, qreal angle);
    qreal viewScale();
    void rotateImage(qreal angle, qreal x, qreal y);
    int blocksCount();
    void deleteBlock(int index);
    void deleteCurrentBlock();
    QPixmap getBlockByIndex(int index);
    QRectF getBlockRectByIndex(int index);
    void clearBlocks();
    qreal getRealScale();
    qreal getRealAngle();
    QPixmap getImage();
    void setMagnifierCursor(QCursor *cursor);
protected:
    virtual void mousePressEvent(QGraphicsSceneMouseEvent *event);
    virtual void mouseMoveEvent(QGraphicsSceneMouseEvent *mouseEvent);
    virtual void mouseReleaseEvent(QGraphicsSceneMouseEvent *mouseEvent);
    virtual void wheelEvent(QGraphicsSceneWheelEvent *wheelEvent);
```

```
    virtual void keyReleaseEvent(QKeyEvent *keyEvent);
    virtual void keyPressEvent(QKeyEvent *keyEvent);
signals:
    void leftMouseClicked(int x, int y, bool blockSelected);
    void rightMouseClicked(int x, int y, bool inTheBlock);
    void keyPressed(int key);
    void blockSelected();
    void blocksCleared();
public slots:
private:
    void leftMouseRelease(qreal x, qreal y);
    void rightMouseRelease(qreal x, qreal y);
    int nearActiveBorder(qreal x, qreal y);
    void clearTransform();
    QPixmap extractPixmap(QGraphicsRectItem *item);
    void deleteBlockRect(QGraphicsRectItem *item);
    ...
    QGraphicsRectItem *m_CurrentBlockRect;
    QGraphicsRectItem *m_LastSelected;
    SelectStates selecting;
    QRectF blockRect;
    QRectF selBlockRect;
    bool hasImage;
    qreal  m_scale;
    qreal real_scale;
    qreal m_rotate;
    qreal real_rotate;
    Qt::MouseButton buttonPressed;
    QCursor *magnifierCursor;
    int near_res;
};
```

Класс позаимствован из реального приложения и потому довольно сложен. Поскольку нас сейчас интересует только взаимодействие объектов этого класса в рамках Graphics View Framework, листинг был несколько сокращен. Полный текст класса и программу, демонстрирующую его работу, вы найдете в папке Ch10/ginput.

При проектировании нового виджета полезно составить диаграмму состояний виджета и переходов между ними, наподобие диаграммы состояний конечного автомата (которым виджет, по сути, и является). Сделаем это и мы (рис. 10.9).

На диаграмме отражены не все состояния виджета, но и она поможет разработать базовые методы соответствующего класса.

Поскольку наш виджет работает как конечный автомат, нужно где-то сохранять его текущее состояние. Таким хранилищем будет переменная `selecting`. Допустимыми значениями этой переменной являются `NoSelect` (начальное состояние), `StartSelect`

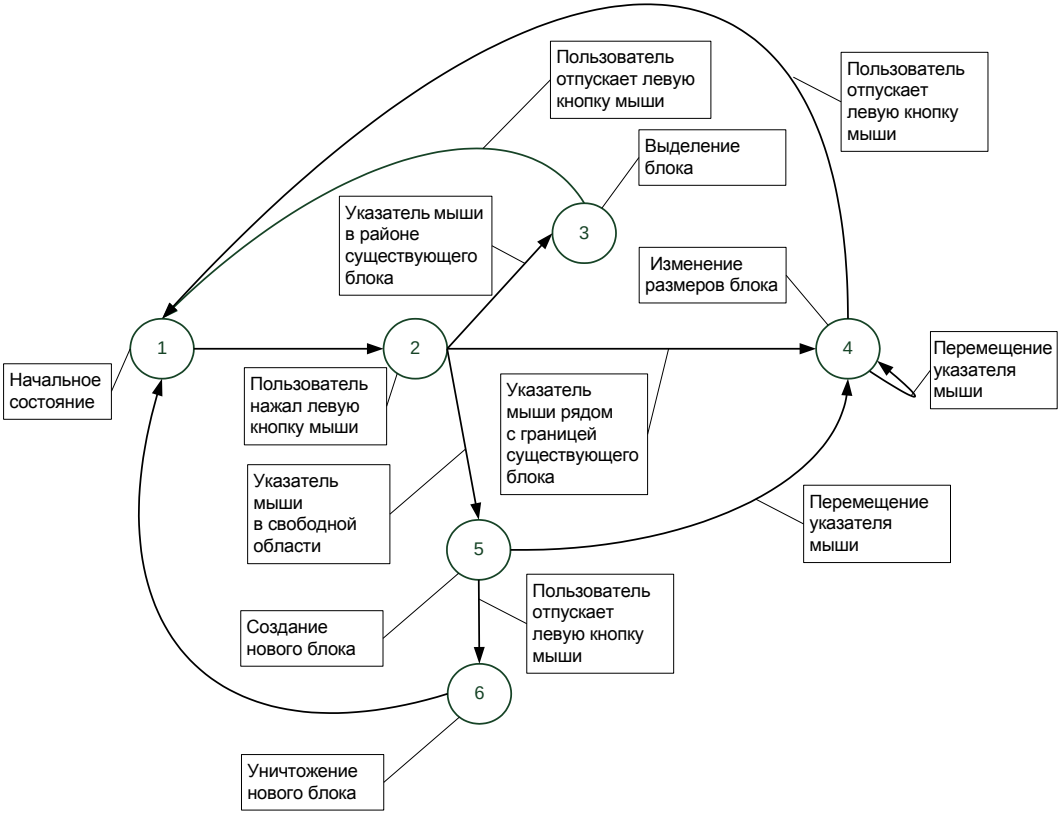


Рис. 10.9. Диаграмма состояний и переходов виджета

(начинается процесс выбора блока), `Selecting` (виджет находится в процессе выбора блока).

Как уже отмечалось, по умолчанию графическая сцена неинтерактивна, но она получает информацию о событиях от объектов, отображающих эту сцену. Для того чтобы сцена реагировала на события, мы, как и в игре "Сокобан", перекрываем ряд методов, ответственных за обработку событий, связанных с устройствами ввода. В нашем случае это методы `mousePressEvent()`, `mouseMoveEvent()`, `mouseReleaseEvent()`, `wheelEvent()`, `keyReleaseEvent()` и `keyPressEvent()`.

Рассмотрим функцию `mousePressEvent()` (листинг 10.21). Этот метод вызывается в том случае, если пользователь нажимает левую кнопку мыши, когда указатель мыши находится в области окна `QGraphicsView`, связанного с объектом класса `GraphicsInput`. Если указатель мыши находится внутри рядом с границей уже существующего блока, виджет переходит в состояние `Selecting`. В этом состоянии перемещение курсора мыши при нажатой левой кнопке приводит к изменению размеров блока. Если в момент нажатия левой кнопки мыши указатель мыши находится в свободной области, то виджет переходит в состояние `StartSelect` и если дальше пользователь начнет перемещать курсор мыши, не отпуская кнопку, будет создан новый блок и виджет перейдет в состояние `Selecting`.

Листинг 10.21. Метод `mousePressEvent()` класса `GraphicsInput`

```
void GraphicsInput::mousePressEvent(QGraphicsSceneMouseEvent *event)
{
    if (!hasImage)
        return;
    if (event->buttons() == Qt::LeftButton) {
        buttonPressed = Qt::LeftButton;
        if (selecting == NoSelect) {
            if ((near_res = nearActiveBorder(event->scenePos().x(),
                    event->scenePos().y())) != 0) {
                m_CurrentBlockRect = m_LastSelected;
                selecting = Selecting;
                blockRect = m_CurrentBlockRect->rect();
            } else {
                selecting = StartSelect;
                blockRect.setLeft(event->lastScenePos().x());
                blockRect.setTop(event->lastScenePos().y());
                blockRect.setWidth(10);
                blockRect.setHeight(10);
            }
        }
    } else
        buttonPressed = Qt::RightButton;
}
```

В начале метода мы проверяем, загружено ли в виджет базовое изображение (если изображение не загружено, то и выделять нечего). Информация об этом хранится в переменной `hasImage`. Затем мы проверяем, какая кнопка мыши была нажата. Если была нажата левая кнопка, необходимо проверить, где в момент нажатия находился указатель мыши (рядом с границей существующего блока или в свободной области). Эту проверку выполняет функция `nearActiveBorder()`, которая возвращает значение 0, если указатель находится не рядом с границей блока, 1 — если указатель находится рядом с левой границей блока, 2 — если указатель находится рядом с верхней границей блока, 3 — если указатель находится около правой границы, и 4 — если указатель находится недалеко от нижней границы. Функции `nearActiveBorder()` необходимо передать координаты указателя мыши, которые хранятся в объекте `event`. Объект `event`, который передается функциям, обрабатывающим события сцены, содержит значения координат в системе координат сцены. Переменная `blockRect` хранит координаты прямоугольника, с которым мы работаем в данный момент. Если виджет переходит в состояние `StartSelect`, мы сохраняем координаты указателя мыши как начальные координаты прямоугольника и задаем начальную ширину и высоту прямоугольника равными 10 пикселам. Имеются в виду пикселы сцены, а не пикселы экрана. Минимальные высота и ширина прямоугольника, который можно выбрать с помощью нашего виджета, соответствуют 12 пикселам. Если вам нужны другие параметры, вы, конечно, можете изменить их.

Минимальные параметры ширины и высоты прямоугольника необходимо задать для того, чтобы отличать простой щелчок мышью в области окна, отображающего сцену, от попытки выбрать прямоугольник. Поясню, в чем дело. Когда пользователю кажется, что он просто щелкнул кнопкой мыши, указатель мыши обычно успевает переместиться на некоторое расстояние, пока кнопка остается нажатой. С точки зрения программы это выглядит так, словно пользователь пытается выбрать прямоугольник. Для того чтобы любой щелчок кнопкой мыши не приводил к выбору прямоугольной области, необходимо задать некий порог чувствительности, за пределами которого прямоугольник не воспринимается как прямоугольник. Рассмотрим теперь метод `mouseReleaseEvent()` (листинг 10.22).

Листинг 10.22. Метод `mouseReleaseEvent()`

```
void GraphicsInput::mouseReleaseEvent(
     QGraphicsSceneMouseEvent *mouseEvent)
{
    if (buttonPressed == Qt::LeftButton) {
        if (selecting == Selecting) {
            selecting = NoSelect;
            if ((blockRect.width() < 12) || (blockRect.height() < 12)) {
                if (m_CurrentBlockRect == m_LastSelected)
                    m_LastSelected = 0;
                deleteBlockRect(m_CurrentBlockRect);
                // Щелчок!
                leftMouseRelease(mouseEvent->scenePos().x(),
                    mouseEvent->scenePos().y());
            } else
                emit blockSelected();
            m_CurrentBlockRect = 0;
        }
        if (selecting == StartSelect) {
            selecting = NoSelect;
            m_CurrentBlockRect = 0;
            leftMouseRelease(mouseEvent->scenePos().x(),
                mouseEvent->scenePos().y());
        }
    }
    if (buttonPressed == Qt::RightButton) {
        this->rightMouseRelease(mouseEvent->scenePos().x(),
            mouseEvent->scenePos().y());
    }
    buttonPressed = Qt::NoButton;
}
```

В этом методе мы тоже сначала проверяем, какая кнопка отпущена. Если отпущена левая кнопка и виджет находился в состоянии `Selecting`, то проверяем размеры выбранного прямоугольника. Если эти размеры оказываются меньше пороговых, то

мы считаем, что пользователь просто щелкнул мышью в окне сцены, и удаляем прямоугольник, который мы уже начали создавать. В противном случае блок добавляется к сцене, а объект `GraphicsInput` эмитирует сигнал `blockSelected()`. Этот сигнал может оказаться полезным для изменения состояния других элементов управления программы, поведение которых зависит от того, был ли выбран нами блок. В любом случае виджет возвращается в начальное состояние. Если виджет находился в состоянии `StartSelect`, мы также считаем, что имел место щелчок мышью.

Теперь рассмотрим метод `mouseMoveEvent()`, самый сложный из всех методов, работающих с мышью (листинг 10.23).

Листинг 10.23. Метод `mouseMoveEvent()`

```
void GraphicsInput::mouseMoveEvent(QGraphicsSceneMouseEvent *mouseEvent)
{
    if (selecting == StartSelect) {
        selecting = Selecting;
        m_CurrentBlockRect = newBlock(blockRect);
    }
    if ((mouseEvent->modifiers() & Qt::ControlModifier) == 0)
        if (mouseEvent->buttons() == Qt::NoButton) {
            near_res = nearActiveBorder(mouseEvent->scenePos().x(),
                mouseEvent->scenePos().y());
            switch (near_res) {
                case 0:
                    m_view->setCursor(Qt::ArrowCursor);
                    break;
                case 1:
                    m_view->setCursor(Qt::SplitHCursor);
                    break;
                case 2:
                    m_view->setCursor(Qt::SplitVCursor);
                    break;
                case 3:
                    m_view->setCursor(Qt::SplitHCursor);
                    break;
                case 4:
                    m_view->setCursor(Qt::SplitVCursor);
                    break;
                default:
                    break;
            }
        }
    QRectF newRect;
    if (near_res && (mouseEvent->buttons()&Qt::LeftButton)) {
        QRectF newRect =
            m_LastSelected->mapRectToScene(m_LastSelected->rect());
```

```
        switch (near_res) {
            case 1:
                newRect.setLeft(mouseEvent->lastScenePos().x());
                break;
            case 2:
                newRect.setTop(mouseEvent->lastScenePos().y());
                break;
            case 3:
                newRect.setRight(mouseEvent->lastScenePos().x());
                break;
            case 4:
                newRect.setBottom(mouseEvent->lastScenePos().y());
                break;
            default:
                break;
        }
        m_CurrentBlockRect = m_LastSelected;
        m_CurrentBlockRect->setRect(
            m_LastSelected->mapRectFromScene(newRect));
        for (int i = 0;
            i < m_CurrentBlockRect->collidingItems().size(); ++i)
            if (m_CurrentBlockRect->collidingItems().at(i)->data(1) ==
                    "block") {
                m_CurrentBlockRect->setRect(
                    m_LastSelected->mapRectFromScene(selBlockRect));
                return;
            }
        selBlockRect = newRect;
        return;
    }
    if (selecting == Selecting) {
        newRect = blockRect;
        if (newRect.left() < mouseEvent->lastScenePos().x())
            newRect.setRight(mouseEvent->lastScenePos().x());
        else
            newRect.setLeft(mouseEvent->lastScenePos().x());
        if (newRect.top() < mouseEvent->lastScenePos().y())
            newRect.setBottom(mouseEvent->lastScenePos().y());
        else
            newRect.setTop(mouseEvent->lastScenePos().y());
        m_CurrentBlockRect->setRect(newRect);
        for (int i = 0;
                i < m_CurrentBlockRect->collidingItems().size(); ++i)
            if (m_CurrentBlockRect->collidingItems().at(i)->data(1) ==
                    "block") {
                m_CurrentBlockRect->setRect(blockRect);
                return;
            }
```

```
        blockRect = newRect;
        emit blockSelected();
        return;
    }
}
```

Во время движения мыши с нажатой левой кнопкой виджет всегда находится в состоянии перетаскивания границы прямоугольника (либо только что созданного, либо существовавшего ранее), и мы хотим, чтобы пиктограмма указателя мыши принимала соответствующий вид. Мы также хотим, чтобы курсор принимал соответствующий вид даже в том случае, если указатель мыши просто окажется рядом с границей выделенного блока. Таким образом, пользователь поймет, что он может перемещать границу блока с помощью мыши. Перетаскивание границ прямоугольника сводится к изменению его координат. Обо всем остальном позаботится Graphics View Framework.

Функции `leftMouseRelease()` и `rightMouseRelease()`, которые мы вызываем из обработчиков событий, выполняют много интересных действий (листинг 10.24).

Листинг 10.24. Методы `leftMouseRelease()` и `rightMouseRelease()`

```
void GraphicsInput::leftMouseRelease(qreal x, qreal y)
{
    QGraphicsItem *it = this->itemAt(x, y);
    if (it) {
        if (it->data(1).toString() == "block") {
            QGraphicsRectItem *r = (QGraphicsRectItem *) it;
            QPen p(Qt::SolidLine);
            QBrush b(Qt::SolidPattern);
            b.setColor(QColor(0, 0, 127, 127));
            p.setColor(QColor(0, 0, 255));
            p.setWidth(2);
            if (r->data(2).toString() == "no") {
                if (m_LastSelected) {
                    m_LastSelected->setPen(p);
                    m_LastSelected->setBrush(b);
                    m_LastSelected->setData(2, "no");
                }
                b.setColor(QColor(127, 0, 0, 127));
                p.setColor(QColor(255, 0, 0));
                r->setData(2, "yes");
                m_LastSelected = r;
                selBlockRect = m_LastSelected->rect();
            } else {
                m_LastSelected = 0;
                r->setData(2, "no");
            }
```

```
            r->setPen(p);
            r->setBrush(b);
        }
    } else
        m_CurrentBlockRect = 0;
    emit leftMouseClicked(m_view->mapFromScene(x, y).x(),
        m_view->mapFromScene(x, y).y(), m_CurrentBlockRect != 0);
}

void GraphicsInput::rightMouseRelease(qreal x, qreal y)
{
    QGraphicsItem *it = this->itemAt(x, y);
    if (it) {
        if (it->data(1).toString() == "block") {
            m_CurrentBlockRect = (QGraphicsRectItem *) it;
        }
    } else
        m_CurrentBlockRect = 0;
    emit rightMouseClicked(m_view->mapFromScene(x, y).x(),
        m_view->mapFromScene(x, y).y(), m_CurrentBlockRect != 0);
}
```

Функция `leftMouseRelease()`, помимо прочего, задает параметры кисти и пера для вывода прямоугольника. Мы пользуемся полупрозрачной кистью, чтобы прямоугольник, отмечающий выделенный блок, не скрывал его содержимое. Еще один интересный момент — генерация сигнала `leftMouseClicked()`. Этот сигнал эмитируется на тот случай, если другим компонентам приложения потребуется обрабатывать щелчок левой кнопки мыши в окне нашего виджета. В списке параметров сигнала передаются координаты указателя мыши в момент щелчка, и тут мы снова сталкиваемся с проблемой перевода значений из одной системы координат в другую. До сих пор нам требовалось переводить координаты окна, отображающего графическую информацию, в координаты сцены, для чего мы пользовались методом `mapToScene()` класса `QGraphicsView`. Обработчики событий, приведенные выше, также имеют дело с координатами в системе координат сцены. Но для внешних компонентов, которым адресован сигнал, координаты в системе сцены, скорее всего, просто не имеют смысла. Им нужнее координаты в системе координат окна отображения сцены. Для этого мы пользуемся методом `mapFromScene()` класса `QGraphicsView`, который выполняет преобразование координат точки, обратное преобразованию, выполняемому методом `mapToScene()`. Аналогично функция `rightMouseRelease()` генерирует сигнал `rightMouseClicked()`. Помимо координат указателя мыши в системе координат окна оба означенных сигнала передают третий параметр типа `bool`. Этот параметр указывает, находился ли указатель мыши в момент щелчка внутри выделенного блока (значение `true`) или в свободной области. Эта информация тоже может быть полезна для внешних компонентов. Например, если в ответ на щелчок правой кнопкой мыши ваша программа выводит кон-

текстное меню, набор команд этого меню может меняться в зависимости от того, находился указатель мыши в момент щелчка внутри блока или нет.

Кроме прочего, наш визуальный компонент умеет масштабировать изображение (как уже отмечалось, в масштабе не более 1:1). Это ограничение связано с изначальным применением компонента — для просмотра отсканированных изображений и выделения фрагментов. Изображения обычно сканируются с высоким разрешением, так что у пользователя вряд ли возникнет потребность просматривать их с разрешением, большим, чем 1:1. Однако в процессе создания виджета возникла интересная задача. Требовалось, чтобы изменение масштаба выполнялось при прокрутке колесика мыши (или при нажатии комбинации клавиш <Ctrl>+<+> или <Ctrl>+<–>). Иными словами, изменение масштаба должно выполняться достаточно быстро. Оказалось, что Graphics View Framework плохо справляется с масштабированием изображений.

Разработчики Graphics View Framework утверждают, что их система способна без труда обрабатывать сцены, состоящие из тысяч небольших графических примитивов, и это правда. Делается это просто: с помощью хорошо известного алгоритма двоичного разбиения пространства (BSP) система очень быстро определяет, какие именно графические примитивы видит пользователь в данный момент (редко бывает так, чтобы пользователь видел сразу все тысячи примитивов, а если бывает, то эта задача требует отдельного решения) и рисует только их.

Но в нашей ситуации сцена состоит из небольшого количества очень крупных примитивов (растрового изображения и блоков), содержащих десятки тысяч пикселов. И если с масштабированием блоков проблем не возникает (в конце концов, это просто однородно закрашенные прямоугольники), то масштабирование растрового изображения действительно может занять много времени. В некоторых ситуациях решением может стать использование мощных ресурсов ускорителя графической карты. Проще говоря, OpenGL. Добавить поддержку OpenGL в Graphics View Framework очень просто, требуется всего лишь несколько строк кода. Но у этого решения есть серьезные недостатки. Не на всех видеоустройствах драйверы поддерживают OpenGL, а там, где поддерживают, максимальный размер растрового изображения, с которым OpenGL может работать напрямую, ограничен не очень большим числом (например, 4096 пикселов по высоте и ширине), с чем мы уже сталкивались.

Причем особенности Graphics View Framework таковы, что при масштабировании сцены может уменьшиться скорость выполнения других операций, например рисования на сцене новых блоков-прямоугольников. С другой стороны, если сцена обрабатывается в масштабе 1:1, проблем с производительностью практически не возникает даже при работе с очень большими изображениями. Происходит так потому, что если графические примитивы отмасштабированы, операция масштабирования каждого примитива (в том числе растрового изображения) выполняется при каждом изменении сцены. Сравните: вы выделяете блок, "ведя" мышью прямоугольник за один из его углов (так, как это делается во всех подобных программах). В обычном случае системе приходилось бы масштабировать растровое изображение всякий раз, когда вы меняете размеры прямоугольника. На практике это приво-

дит к тому, что менять размеры выделенного прямоугольника с помощью мыши в реальном времени просто невозможно.

Правильное понимание проблемы фактически подсказало и решение: сделать так, чтобы при разных масштабах отображения растрового изображения система работала в масштабе 1:1. Самый простой способ сделать это — заранее создать несколько копий базового изображения, соответствующих масштабам, используемым наиболее часто. При таком подходе при изменении масштаба сцены само растровое изображение не масштабируется, только подставляется его заранее созданная копия. Все остальные примитивы (прямоугольники и т. д.) масштабируются обычным способом, что не требует много времени. Таким образом, в нашей системе масштабирование выполняется один раз, при загрузке, а когда пользователь меняет размеры того же прямоугольника, системе достаточно перерисовывать этот блок и изображение непосредственно под ним, с чем она вполне справляется и в реальном времени.

Достоинство этого подхода заключается в том, что система обрабатывает изображение с высокой скоростью даже на самой скромной аппаратной конфигурации и на любой поддерживаемой Qt платформе, включая режим удаленного рабочего стола Linux.

Недостатки предложенного решения тоже очевидны. Прежде всего, оно исключает произвольное масштабирование изображения. Масштабировать можно только в соответствии с заранее заданными значениями.

При этом стоит отметить, что в современных графических системах нет прямой зависимости между коэффициентом масштабирования и временем, которое на него затрачивается, так что вариант "задать несколько фиксированных масштабов, а потом "домасштабировать" ближайший подходящий вариант до выбранного пользователем значения" сводит на нет весь выигрыш в скорости.

Второй недостаток — потребление памяти кэшированными копиями. Оно, естественно, будет тем больше, чем больше предустановленных масштабов будет задано. В классическом противостоянии "занятый объем памяти — быстродействие" мы делаем выбор в пользу быстродействия.

Поскольку, независимо от масштаба изображения, пользователь хочет получать содержимое выделенных блоков в масштабе 1:1, копирование фрагмента для пользователя производится из оригинального изображения. При этом координаты блоков пересчитываются в координаты оригинального изображения.

Посмотрим, как все это реализуется на практике. Создание кэширующих копий выполняет метод `loadImage()`, на который возложена задача загрузки исходного растрового изображения (листинг 10.25).

Листинг 10.25. Метод `loadImage()`

```
bool GraphicsInput::loadImage(const QPixmap &image, bool clearBlocks)
{
    if (clearBlocks || (!hasImage)) {
        real_rotate = 0;
```

```
        real_scale = 1;
        this->clear();
        this->items().clear();
        m_LastSelected = 0;
        m_CurrentBlockRect = 0;
    }
    if ((!clearBlocks) && hasImage) {
        this->removeItem(m_image);
        real_scale = 1;
    }
    old_pixmap = image;
    pm2 = image.scaledToWidth(image.width() / 2);
    pm4 = pm2.scaledToWidth(pm2.width() / 2);
    pm8 = pm4.scaledToWidth(pm4.width() / 2);
    pm16 = pm8.scaledToWidth(pm8.width() / 2);
    this->setSceneRect(image.rect());
    m_image = this->addPixmap(image);
    m_realImage = this->addPixmap(image);
    m_realImage->setData(1, "image");
    m_realImage->hide();
    this->setFocus();
    m_image->setFocus();
    m_image->setAcceptedMouseButtons(Qt::LeftButton | Qt::RightButton |
        Qt::MidButton);
    m_image->setAcceptHoverEvents(true);
    m_image->setData(1, "image");
    if (m_view) {
        m_view->centerOn(0, 0);
        m_view->show();
        hasImage = true;
        return true;
    } else
        return false;
}
```

Само кэширование тоже занимает некоторое время, но при умелом подборе алгоритма его можно сделать практически незаметным для пользователя. Поскольку в нашем случае изображение нужно только уменьшать по сравнению с оригиналом, все кэширующие копии будут меньше оригинала. Этот факт можно использовать для ускорения самого кэширования (что мы и делаем). Допустим, что мы решили хранить изображения в размерах, составляющих 1/2, 1/4, 1/8 и 1/16 оригинала. В этом случае каждую следующую копию можно получить уменьшением в два раза размеров предыдущей копии, что занимает меньше времени, чем если бы все копии получались из исходного изображения, т. к. каждый раз мы работаем с изображением с меньшим количеством пикселов. Кэширующие копии хранятся в переменных pm2, pm4, pm8, pm16 (1/2, 1/4, 1/8 и 1/16 оригинала соответственно). Ориги-

нальное изображение, из которого извлекаются выделенные пользователем фрагменты, хранится в объекте `m_realImage`.

Само масштабирование выполняется методом `setViewScale()`, который просто подставляет изображение нужного размера из кэша. При этом все прямоугольники, отмечающие выделенные блоки, масштабируются "по-настоящему".

Функция `extractPixmap()` используется различными методами класса `GraphicsInput`, которые возвращают выделенные блоки (листинг 10.26).

Листинг 10.26. Метод `extractPixmap()`

```
QPixmap GraphicsInput::extractPixmap(QGraphicsRectItem *item)
{
    if (!hasImage) {
        return QPixmap(0, 0);
    }
    QRectF rect(0, 0, 0, 0);
    if (item == 0) {
        foreach(QGraphicsItem * it, m_view->scene()->items()) {
            if (it->data(1).toString() == "block") {
                rect = it->mapRectToScene(it->boundingRect());
                break;
            }
        }
    } else
        rect = item->mapRectToScene(item->rect());
    return m_realImage->pixmap().copy(rect.left() / real_scale,
       rect.top() / real_scale,
       (rect.right() — rect.left()) / real_scale,
       (rect.bottom() — rect.top()) / real_scale);
}
```

Если в виджете выделено несколько блоков, и один из них выбран (помечен красным цветом), функция возвращает объект `QPixmap`, соответствующий этому блоку. Если ни один из блоков не выбран, функция возвращает первый выделенный блок. Если ни один блок не выбран, результатом выполнения функции становится пустой объект `QPixmap` (для которого метод `isNull()` возвращает значение `true`). Когда мы говорим, что функция возвращает блок, мы имеем в виду, что она возвращает фрагмент изображения, соответствующий этому блоку. Определить, какой именно фрагмент соответствует блоку, не так уж и трудно, но для этого опять-таки необходимо помнить, в какой системе координат выполняется та или иная операция. В данном случае мы имеем дело с тремя системами координат: системой координат прямоугольника, которым помечен выделенный блок (напомню, что прямоугольник, как и любой графический примитив Graphics View Framework, обладает собственной системой координат), системой координат растрового изображения, которое с точки зрения Graphics View Framework тоже является графическим примитивом,

и системой координат сцены. Поскольку начало координат растрового изображения совпадает с началом координат сцены, преобразование между этими системами координат выполняется тривиально. Для преобразования координат прямоугольника в координаты сцены мы используем функцию `mapRectToScene()` класса `QGraphicsScene()` (эта функция была введена в Qt 4.5).

Для получения требуемого блока необходимо преобразовать прямоугольники из того масштаба, в котором отображается сцена, в масштаб 1:1. С этой целью используется переменная `real_scale`, которая содержит коэффициент масштабирования примитивов сцены относительно оригинального изображения. После получения правильных координат прямоугольника мы, с помощью метода `copy()` объекта класса `QPixmap`, извлекаем соответствующий фрагмент из объекта `m_realImage` и создаем новый объект `QPixmap`, который и возвращается в качестве результата функции. Возвращенный графический фрагмент можно передать другим компонентам для обработки, сохранить на диске или скопировать в буфер обмена (рис. 10.10).

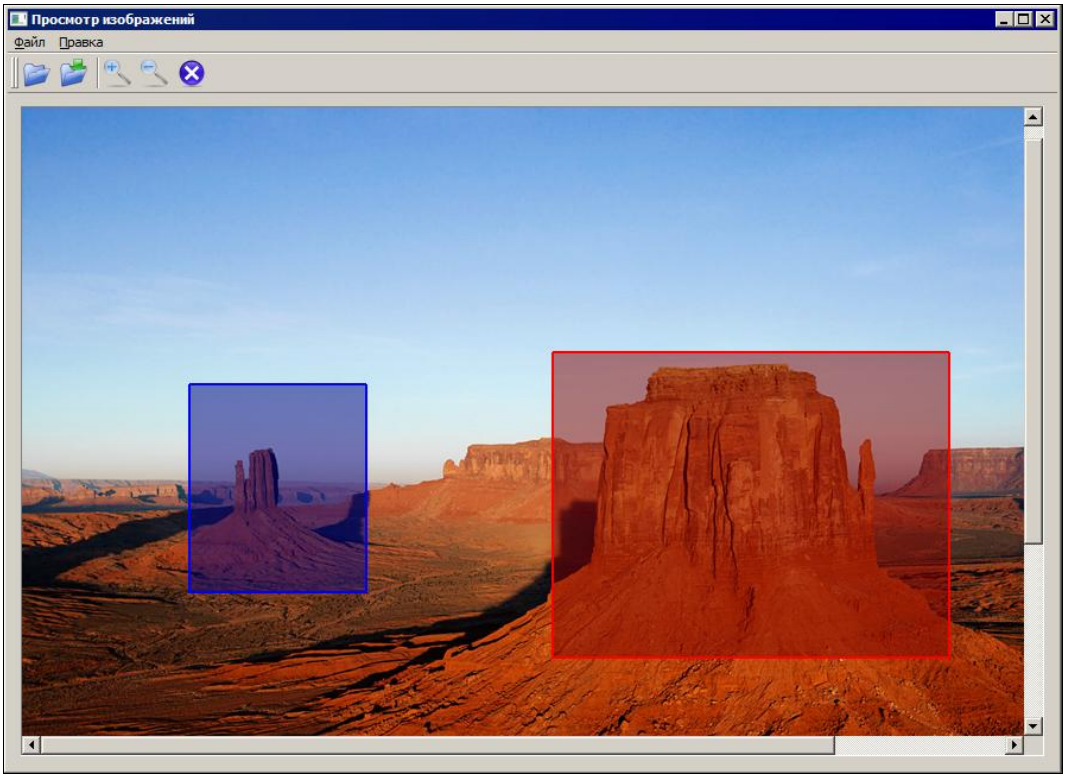


Рис. 10.10. Выделение нескольких фрагментов изображения

Думаю, вы уже убедились, насколько полезна система Graphics View Framework при создании сложных сцен и новых сложных виджетов. Но это далеко не все, что можно сказать о ней.

Graphics View Framework и OpenGL

На протяжении этой книги мы уже неоднократно использовали аппаратную поддержку OpenGL для ускорения вывода двухмерной графики. Система Graphics View Framework позволяет делать это так же легко. У нас есть два пути, простой и сложный. Простой путь заключается в том, чтобы назначить объект класса `QGLWidget` в качестве порта вывода виджета `QGraphicsView` (с помощью метода `setViewPort()`, что подробно описано в документации). При этом из всех режимов обновления порта вывода придется выбрать самый неоптимальный `FullViewportUpdate`, поскольку объект `QGLWidget` может выполнять функцию порта вывода только в этом режиме. Подробности можно найти по адресу **http://doc.trolltech.com/qq/qq26-openglcanvas.html#turbocharginggraphicsview**. Более сложный путь предоставляет нам больше возможностей в плане управления выводом сцены (иначе не было бы смысла о нем говорить). Он заключается в использовании объекта класса `QGLWidget` для непосредственного вывода содержимого объекта `QGraphicsScene`, не задействуя объект `QGraphicsView`. Это возможно благодаря методу `render()` класса `QGraphicsScene`, который позволяет отобразить всю сцену или ее прямоугольную область в некоторую прямоугольную область объекта, поддерживающего вывод графики (например, объекта класса `QGLWidget`). Для этого достаточно перекрыть метод `paintGL()` класса `QGLWidget` (листинг 10.27)

Листинг 10.27. Метод `paintGL()`, в котором выводится содержимое графической сцены

```
void QXtGLWidget::paintGL()
{
  QPainter painter(this);
  graphicsScene->render(&painter, QRectF(this->rect()),
    graphicsScene->sceneRect());
  painter.end();
  swapBuffers();
}
```

Это простейший вариант метода `paintGL()`, который просто перерисовывает сцену `graphicsScene` в прямоугольную область объекта класса-потомка `QGLWidget`. Но возможны и более интересные варианты. Для ускорения рисования сцены иногда приходится жертвовать малозначительными деталями. Перекрыв метод `paintGL()`, мы можем отключать отрисовку отдельных элементов сцены или временно заменять их на более простые.

Рассмотрим в качестве примера нашу реализацию игры "Сокобан". В принципе, малозначительных деталей в сцене игры нет, да и сама сцена настолько проста, что для ее перерисовки никакое ускорение не требуется. Тем не менее эта сцена может продемонстрировать общий принцип. Допустим, что при перерисовке сцены "Сокобан" мы решили пожертвовать ящиками (листинг 10.28).

Листинг 10.28. Перерисовка сцены "Сокобан" без ящиков

```
void QXtGLWidget::paintGL()
{
    foreach(QGraphicsItem * item, graphicsScene->items()) {
        if (item->data(0) == "Box")
            item->hide();
    }
  QPainter painter(this);
  graphicsScene->render(&painter, QRectF(this->rect()),
    graphicsScene->sceneRect());
  painter.end();
  swapBuffers();
  foreach(QGraphicsItem * item, graphicsScene->items()) {
      if (item->data(0) == "Box")
          item->show();
  }
}
```

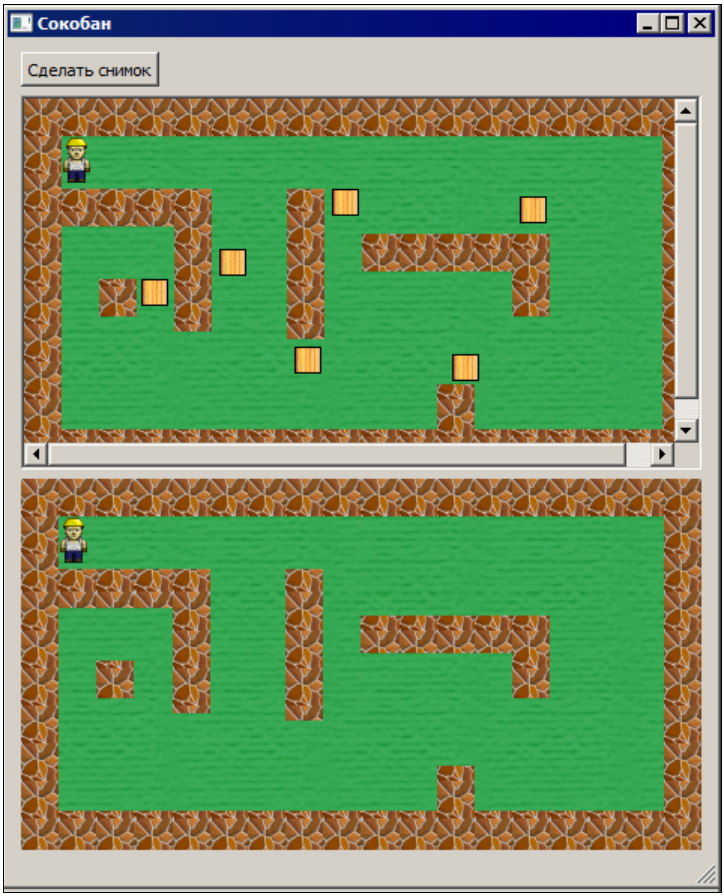


Рис. 10.11. Два способа вывода сцены игры "Сокобан"

Для того чтобы графический примитив не перерисовывался, нам достаточно сделать его невидимым, вызвав метод `hide()` соответствующего объекта `QGraphicsItem`. Затем видимость объекта можно восстановить, вызвав метод `show()`. В результате у нас появится "альтернативная" сцена игры "Сокобан", не содержащая ящиков (рис. 10.11).

Нижняя часть сцены, нарисованная с помощью OpenGL, не содержит ящиков.

Между прочим, у класса `QGraphicsItem` тоже есть метод `paint()`, позволяющий отобразить объект соответствующего класса-потомка (сам класс `QGraphicsItem` является абстрактным, т. е. содержит нереализованные виртуальные методы) на любой поверхности, которую можно связать с объектом `QPainter`. Однако если вы попытаетесь заменить метод `render()` класса `QGraphicsScene` на примерно такой код:

```
foreach(QGraphicsItem * item, graphicsScene->items())
  item->paint(...);
```

то вас будет ожидать разочарование. Дело в том, что объект класса-потомка `QGraphicsItem` перерисовывает свое содержимое в своей собственной системе координат, и это разумно, т. к. упрощает работу создателям классов-потомков `QGraphicsItem`. Но для нас это означает, что сам объект класса-потомка `QGraphicsItem` нарисует себя без учета таких преобразований как перенос, масштабирование или поворот. Иначе говоря, все объекты сцены будут расположены беспорядочно.

Виджет в стиле браузера Opera

Вернемся к созданию интересных и полезных виджетов, использующих возможности Graphics View Framework. На этот раз мы создадим виджет, отражающий другую популярную тенденцию в области построения графических интерфейсов: создание закладок (посещенных страниц, открытых окон и приложений и т. п.) в виде уменьшенных копий, причем иногда это не просто снимки соответствующих окон в определенный момент времени, а "живые" закладки, вид которых меняется вместе с изменением вида окна.

Мне трудно сказать, кто первым придумал и реализовал эту идею. Кажется, она увидела свет в "активных каналах" Windows 98 (теперь эта идея весьма широко применяется в Windows Vista и Windows 7), или в браузере Opera. Я буду называть такие виджеты "виджетами в стиле Opera", потому что именно в этой программе я чаще всего их вижу.

От предыдущих виджетов, созданных на основе Graphics View Framework, этот виджет будет отличаться тем, что в нем мы воспользуемся графическими элементами Graphics View Framework, специально предназначенными для использования в качестве виджетов, — классами `QGraphicsWidget` и `QGraphicsLayout`.

Как уже отмечалось, класс `QGraphicsWidget` нельзя считать настоящим виджетом Qt, поскольку он не происходит от класса `QWidget`. Тем не менее, учитывая некоторые тенденции в развитии Qt, можно ожидать, что в следующих версиях пользовательский интерфейс Qt будет строиться именно на основе таких классов, как

QGraphicsWidget. Будучи потомком QObject, класс QGraphicsWidget умеет работать с сигналами, слотами и свойствами. Являясь также потомком QGraphicsItem, этот класс способен встраиваться в графическую сцену и встраивать в себя дочерние графические элементы, включая виджеты. Кроме того, и этим класс QGraphicsWidget и его потомки похожи на классы-потомки QWidget, объекты класса QGraphicsWidget и классов-потомков этого класса могут использовать менеджеры компоновки. Эти менеджеры (потомки класса QGraphicsLayout) отличаются от менеджеров компоновки, которые применяют стандартные виджеты, тем не менее, они решают те задачи, над решением которых мы бились при создании собственного виджета, встраиваемого в графическую схему, — зафиксировать положение графических виджетов относительно друг друга или относительно окна просмотра. Последнюю задачу решает виджет QGraphicsAnchorLayout, который появился в Qt 4.6. Можно было бы ожидать, что корневым объектом, к которому привязываются менеджеры компоновки, будет объект класса QGraphicsScene или QGraphicsView, но это не так. Корнем иерархии менеджеров компоновки должен быть объект класса-потомка QGraphicsLayoutItem, к каковым относятся объекты классов QGraphicsLayout и QGraphicsWidget. Таким образом, при создании иерархии менеджеров компоновки мы сначала добавляем в графическую сцену объект класса QGraphicsWidget (в отличие от QGraphicsLayout это не абстрактный класс, и его объекты создавать можно), а затем делаем его корнем иерархии менеджеров компоновки.

Мы используем все эти функциональные возможности QGraphicsWidget. Наша задача — создать виджет, который будет содержать уменьшенную копию другого виджета, окруженную рамкой окна стандартного размера.

Для начала мы создадим класс Applet (листинг 10.29), который наследует классу QGraphicsWidget и решает несколько задач: заполняет свою рабочую область содержимым дочернего графического примитива или виджета так, чтобы область была заполнена максимально, но при этом сохранились исходные пропорции дочернего элемента. Для этого применяется масштабирование дочернего элемента с помощью метода setScale().

> ***Примечание***
>
> В общем случае геометрия дочернего графического примитива никак не ограничивается размерами примитива-родителя, но в случае с виджетом Applet мы хотим, чтобы дочерний примитив, который составляет содержимое окна Applet, не выходил за пределы родительского окна.

Затем вокруг рабочей области создается обрамление стандартного диалогового окна. Кроме того, класс Applet реагирует на некоторые действия пользователя, эмитируя соответствующие сигналы. Класс позволяет также менять стандартное поведение элемента графической сцены на поведение, более характерное для виджета.

Листинг 10.29. Класс Applet

```
class Applet : public QGraphicsWidget
{
    Q_OBJECT
```

```
public:
    Applet(QGraphicsItem * parent = 0);
    ~Applet();
    void addChildItem(QGraphicsItem * item);
    void addChildWidget(QWidget * w);
    void setGeometry(const QRectF &rect);
    void setGeometry(qreal x, qreal y, qreal w, qreal h);
    void setCanClose(bool b);
    bool canClose();
    void fixCurrentSize();
    void setMovable(bool b);
    bool movable();
    int VMargingHint();
    int HMargingHint();
signals:
    void doubleClicked();
    void closed();
protected:
    void mouseDoubleClickEvent(QGraphicsSceneMouseEvent * event);
    void closeEvent ( QCloseEvent * event );
    QVariant itemChange(GraphicsItemChange change,
        const QVariant & value);
private:
    void scaleToFit();
    QWidget * widget;
    bool m_canClose;
    bool m_movable;
    int mcounter;
};
```

Для добавления в виджет элемента, который должен заполнить его рабочую область, служат методы `addChildItem()` и `addChildWidget()`. Эти методы устроены так, чтобы у объекта `Applet` всегда был только один дочерний элемент. В ответ на двойной щелчок мышью объект эмитирует сигнал `doubleClicked()`, а в ответ на попытку закрыть окно виджета с помощью соответствующей кнопки (не забывайте, что у виджета `Applet` есть заголовок настоящего диалогового окна) — сигнал `closed()`. Этот сигнал эмитируется только в том случае, если окно виджета действительно можно закрыть (если метод `canClose()` возвращает значение `true`). Изменить реакцию виджета `Applet` на щелчок по кнопке **Закрыть** можно с помощью метода `setCanClose()`. Каждый раз, когда размеры виджета меняются, его содержимое, т. е. дочерний элемент, тоже меняет свои размеры таким образом, чтобы оставаться внутри виджета и при этом занимать максимально возможную часть его рабочей области, не нарушая при этом пропорций. Мы добиваемся этого за счет перекрытия методов `setGeometry()`. Эти методы, которые вызываются всякий раз, когда пользователь изменяет размер виджета, вызывают метод `scaleToFit()`, задача которого

как раз и заключается в том, чтобы изменить размер дочернего элемента, соблюдая его пропорции (даже если пропорции виджета `Applet` изменятся).

Для того чтобы зафиксировать виджеты `Applet` на странице, мы воспользуемся менеджером компоновки `QGraphicsGridLayout`, который можно рассматривать как аналог `QGridLayout` для объектов `QGraphicsWidget`.

Менеджер компоновки `QGraphicsGridLayout` работает не так, как менеджеры компоновки, предназначенные для работы с отдельными виджетами. Этот менеджер позволяет перемещать виджеты и произвольно изменять их размеры, чего обычные менеджеры компоновки делать не позволяют. Справедливости ради надо сказать, что размеры и положение обычных элементов `QGraphicsItem` нельзя изменить интерактивно, — это делается только программными методами.

Возможно, вы захотите сделать так, чтобы позиции, занимаемые закладками, и размеры самих закладок оставались неизменными. На первый взгляд может показаться, что для этого достаточно перекрыть методы `moveEvent()` и `resizeEvent()` точно так же, как мы это сделали с методом `closeEvent()`, но в данном случае этот прием не сработает. Методы `moveEvent()` и `resizeEvent()` вызываются постфактум, после того как соответствующие преобразования уже завершены. Здесь мы пойдем другим путем. Посмотрите на код метода `fixCurrentSize()`, который не позволит изменять размеры окна закладки, "ухватившись" за край окна мышью (листинг 10.30).

Листинг 10.30. Метод `fixCurrentSize()`

```
void Applet::fixCurrentSize()
{
    setMinimumSize(size());
    setMaximumSize(size());
}
```

С блокированием перемещений окна дело обстоит сложнее. Виджет с заголовком в стиле диалогового окна можно перемещать, "ухватившись мышью за заголовок", и попытка присвоить атрибуту `QGraphicsItem::ItemIsMovable` значение `false` ничего не изменит. Нам придется воспользоваться тем же приемом, что и в примере extwidgets, где мы блокировали масштабирование и поворот. Как вы, вероятно, помните, вызов метода `setFlag()` с параметрами `QGraphicsItem::ItemSendsGeometryChanges` и `true` соответственно приводит к тому, что перед каждым геометрическим преобразованием, применяемым к примитиву, будет вызываться метод `itemChange()`, с помощью которого можно отменить данное преобразование. Мы устанавливаем соответствующее значение флага в конструкторе класса `Applet` и вводим в класс перекрытый метод `itemChange()`(листинг 10.31).

Листинг 10.31. Метод `itemChange()` класса `Applet`

```
QVariant Applet::itemChange(GraphicsItemChange change,
    const QVariant &value)
```

```
{
    if (change == QGraphicsItem::ItemPositionChange) {
        if ((mcounter < 1) || m_movable) {
            mcounter ++;
            return value;
        }
        QPointF p = pos();
        return QVariant(p);
    } else
        return value;
}
```

Переменная `m_movable`, значением которой управляет метод `setMovable()` (по умолчанию это значение равно `true`), определяет, сможет ли пользователь перемещать окно виджета. Обратите внимание на переменную `mcounter`, которой в конструкторе класса присваивается значение 0. Очевидно, что представленная реализация метода `itemChange()` разрешает переместить виджет один раз (это должен сделать менеджер компоновки, иначе все виджеты займут одно и то же положение и вместо четырех виджетов мы увидим только один). Дальнейшие перемещения виджета запрещаются. По этой причине, если вы хотите, чтобы положение виджетов `Applet` относительно друг друга было зафиксировано, необходимо сначала заполнить менеджер компоновки всеми объектами `Applet`, а уже затем для каждого из этих объектов вызывать метод `setMoveable()` с параметром `false`. Если добавлять в менеджер компоновки новые объекты после вызова этого метода, менеджеру может понадобиться переместить объекты `Applet` для сохранения параметров компоновки, а сделать этого он уже не сможет.

В качестве "наполнителя" виджета `Applet` мы используем объект класса `QGraphicsWebView`. Этот класс, являющийся потомком `QGraphicsWidget`, можно рассматривать как аналог класса `QWebView`. Как и `QWebView`, класс `QGraphicsWebView` умеет отображать содержимое Web-страниц, на которые ему передана ссылка, позволяет пользователю переходить по ссылкам и выполнять другие действия, стандартные для Web-браузера. Поскольку класс `QGraphicsWebView` является потомком `QGraphicsItem`, к нему можно применять те же преобразования, что и к другим объектам `QGraphicsItem`.

Посмотрим теперь на виджет `Applet` в работе (листинг 10.32).

Листинг 10.32. Создание менеджера компоновки виджетов `Applet`

```
QGraphicsScene * gs = new QGraphicsScene(this);
ui->graphicsView->setScene(gs);
QGraphicsWidget * mainForm = new QGraphicsWidget;
QGraphicsGridLayout * layout = new QGraphicsGridLayout(0);
layout->setGeometry(QRectF(0, 0,900,900));
layout->setColumnMinimumWidth(0, 400);
layout->setRowMinimumHeight(0, 300);
layout->setColumnMinimumWidth(1, 400);
```

```
layout->setRowMinimumHeight(1, 300);
mainForm->setLayout(layout);
gs->addItem(mainForm);

Applet * applet1 = new Applet();
applet1->setGeometry(0,0,400,300);
applet1->setCanClose(false);
applet1->fixCurrentSize();
gs->addItem(applet1);
QGraphicsWebView * wv = new QGraphicsWebView();
wv->setUrl(QUrl("http://www.sat24.com/"));
wv->setGeometry(QRectF(0, 0, 800, 600));
applet1->addChildItem(wv);
applet1->setWindowTitle("www.sat24.com");
layout->addItem(applet1, 0, 0);

layout->setRowSpacing(0, applet1->VMargingHint()+2);
layout->setColumnSpacing(0, applet1->HMargingHint()+2);
layout->setRowSpacing(1, applet1->VMargingHint()+2);
layout->setColumnSpacing(1, applet1->HMargingHint()+2);

Applet * applet2 = new Applet();
applet2->setGeometry(0,0,400,300);
applet2->setCanClose(false);
applet2->fixCurrentSize();
gs->addItem(applet2);
wv = new QGraphicsWebView();
wv->setUrl(QUrl("http://www.yandex.ru/"));
wv->setGeometry(QRectF(0, 0, 800, 600));
applet2->addChildItem(wv);
applet2->setWindowTitle("yandex.ru");
layout->addItem(applet2, 0, 1);
...
applet1->setMoveable(false);
applet2->setMoveable(false);
applet3->setMoveable(false);
applet4->setMoveable(false);
```

Листинг представляет собой сокращенный вариант конструктора главного класса приложения. В приложении operalike, которое вы найдете в папке Ch10, мы создаем не два, а четыре виджета. Обратите внимание на то, что метод `setMoveable()` мы вызываем после того, как менеджер компоновки заполняется виджетами. Следует также учитывать, что менеджер компоновки `QGraphicsGridLayout` при определении геометрии виджета почему-то не учитывает стандартное обрамление диалогового окна, которое мы придаем виджету `Applet` путем передачи базовому конструктору `QGraphicsWidget` параметра `Qt::Dialog`. В результате, если мы не предпримем никаких специальных мер, виджеты `Applet`, размещенные с помощью менеджера компоновки `QGraphicsGridLayout`, будут наезжать друг на друга. Специально для решения этой проблемы в класс `Applet` были введены методы `VMargingHint()` и

`HMargingHint()`, которые возвращают значение соответственно вертикального и горизонтального отступа (spacing) менеджера компоновки, при котором виджеты не будут накладываться друг на друга.

Методы `VMargingHint()` и `HMargingHint()` используют объекты класса `QStyle`, которые возвращают размеры декораций окон в пикселах (листинг 10.33). При масштабировании виджета эти параметры следует умножать на соответствующий коэффициент. Учтите также, что в системах, основанных на X11, например Linux, значения, возвращенные этими методами, могут быть неточными.

Листинг 10.33. Создание менеджера компоновки виджетов `Applet`

```
int Applet::VMargingHint()
{
    return style()->pixelMetric(QStyle::PM_TitleBarHeight) +
        style()->pixelMetric(QStyle::PM_MdiSubWindowFrameWidth);
}

int Applet::HMargingHint()
{
    return style()->pixelMetric(QStyle::PM_MdiSubWindowFrameWidth)*2;
}
```

Рис. 10.12. Виджет в стиле Opera

В результате мы получим интерфейс, напоминающий интерфейс заглавной страницы браузера Opera (рис. 10.12).

Создаем графический виджет

В этом разделе мы рассмотрим создание класса-потомка класса `QGraphicsWidget`, обладающего всеми необходимыми атрибутами графического виджета. Наш виджет, который мы назовем `EllipseWidget` (листинг 10.34), выводит эллипс с обрамляющей линией и полупрозрачным заполнением заданного цвета. Класс `EllipseWidget` обрабатывает события мыши и эмитирует соответствующие сигналы, чего его аналог, класс `QgraphicsEllipseItem`, делать не умеет.

Листинг 10.34. Класс `EllipseWidget`

```
class EllipseWidget : public QGraphicsWidget
{
    Q_OBJECT
    Q_PROPERTY(qreal rx READ rx WRITE setRx)
    Q_PROPERTY(qreal ry READ ry WRITE setRy)
public:
    EllipseWidget(QGraphicsWidget *parent = 0);
    qreal rx();
    void setRx(qreal value);
    qreal ry();
    void setRy(qreal value);
    void setColor(QColor aColor);
    QColor getColor();
    QRectF boundingRect () const;
signals:
    void doubleClicked();
    void mouseDown();
    void mouseUp();
    void focusIn();
    void focusOut();
protected:
    void focusInEvent(QFocusEvent *event);
    void focusOutEvent(QFocusEvent *event);
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
        QWidget *widget);
    void mouseDoubleClickEvent ( QGraphicsSceneMouseEvent * event );
    void mousePressEvent (QGraphicsSceneMouseEvent *event);
    void mouseReleaseEvent (QGraphicsSceneMouseEvent *event);
private:
    bool belongs(const QPointF &p);
    qreal mRx, mRy;
    QColor color;
};
```

Полуоси эллипса, которые и определяют размеры виджета, задаются методами `setRx()` и `setRy()`. Цвет заливки виджета задается методом `setColor()`.

Для того чтобы создать работоспособный класс-потомок класса `QGraphicsWidget`, необходимо выполнить, как минимум, два действия: определить метод `paint()`, который собственно и будет выводить виджет на экран (или другое устройство вывода) и определить метод `boundingRect()`, который будет выводить обрамляющий прямоугольник для виджета (сам виджет может иметь любую форму). Обрамляющий прямоугольник необходим для того, чтобы система Graphics View Framework "знала", когда виджет становится видимым и его следует перерисовать. Методы `paint()` и `boundingRect()` тесно связаны друг с другом. И тот, и другой работают в локальной системе координат виджета. И тот, и другой должны задавать параметры виджета одинаково относительно этой системы координат. Наиболее рациональными являются два варианта, представленные на рис. 10.13.

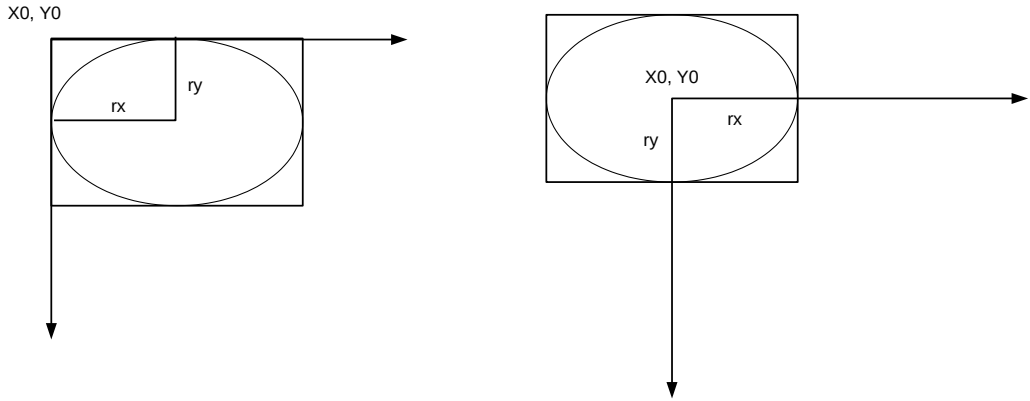


Рис. 10.13. Два варианта вывода эллипса и обрамляющего прямоугольника

В верхнем варианте начало локальной системы координат совпадает с верхним левым углом обрамляющего прямоугольника. В этом случае обрамляющий прямоугольник можно описать следующим образом: `QRectF(0, 0, 2*rx, 2*ry)`, где `rx` и `ry` — полуоси эллипса. При этом эллипс нужно нарисовать так, чтобы он вписывался в прямоугольник, т. е. чтобы его центр имел координаты `rx`, `ry`. Во втором варианте, показанном в нижней части рисунка, центр эллипса и обрамляющего прямоугольника совпадают с началом координат. В этой ситуации обрамляющий прямоугольник можно описать так: `QRectF(-rx, -ry, 2*rx, 2*ry)`. Соответственно должен быть нарисован и эллипс. С точки зрения Graphics View Framework никакого существенного различия между этими двумя способами нет, но необходимо, чтобы один и тот же способ применялся в методах `paint()` и `boundingRect()`. Как и в других случаях, когда мы работаем с локальной системой координат, метод `boundingRect()` (так же, как и метод `paint()`) ничего не должен "знать" о геометрических преобразованиях, которым подвергается виджет в геометрической сцене. Все необходимые поправки будут сделаны соответствующим объектом `QGraphicsScene`.

В нашем виджете (листинг 10.35) применяется первый метод.

Листинг 10.35. Методы `paint()` и `boundingRect()` класса `EllipseWidget`

```
void EllipseWidget::paint(QPainter *painter,
    const QStyleOptionGraphicsItem *option, QWidget *widget)
{
    QBrush oldBrush = painter->brush();
    QPen oldPen = painter->pen();
    QPen pen(this->color);
    color.setAlpha(100);
    QBrush brush(color);
    pen.setWidth(2);
    painter->setPen(pen);
    painter->setBrush(brush);
    painter->drawEllipse(QPointF(mRx,mRy), mRx, mRy);
    painter->setBrush(oldBrush);
    painter->setPen(oldPen);
}

QRectF EllipseWidget::boundingRect() const
{
    const int w = 2*const_cast<EllipseWidget*>(this)->rx();
    const int h = 2*const_cast<EllipseWidget*>(this)->ry();
    return QRectF(0, 0, w, h);
}
```

Помимо прочего, метод `paint()` получает указатель на объект `QPainter`, который он и использует для вывода изображения. Обратите внимание, что при вызове метода `drawEllipse()` в качестве центра эллипса указывается точка с координатами `rx`, `ry` (соответствующие значения хранятся в переменных `mRx` и `mRy`). Теперь наш виджет умеет правильно отображать себя.

Для того чтобы виджет мог реагировать на события мыши, мы должны перекрыть соответствующие методы: `mouseDoubleClickEvent()`, `mousePressEvent()` и `mouseReleaseEvent()`. Здесь надо учесть одну особенность работы системы. Когда пользователь щелкает мышью в окне `QGraphicsView`, система отображает координаты указателя мыши в момент щелчка в координаты объекта `QGraphicsScene`. Затем системе необходимо определить, принадлежит ли эта точка какому-либо из виджетов. Система делает это, проверяя принадлежность точки прямоугольникам виджетов. Но для проверки используется не прямоугольник, который возвращает метод `boundingRect()`, как этого можно было бы ожидать, а прямоугольник, который возвращает метод `geometry()`. Эти два прямоугольника не зависят друг от друга. В результате, если не принять никаких специальных мер, может возникнуть ситуация, когда щелчок мыши по изображению, созданному виджетом, не будет приводить к передаче виджету соответствующего события.

Мы решаем эту проблему, обновляя виджет каждый раз, когда программа обновляет значение полуосей эллипса (листинг 10.36).

Листинг 10.36. Методы `setRx()` и `setRy()` класса `EllipseWidget`

```
void EllipseWidget::setRx(qreal value)
{
    mRx = value;
    setGeometry(pos().x(), pos().y(), boundingRect().width(),
         boundingRect().height());
}

void EllipseWidget::setRy(qreal value)
{
    mRy = value;
    setGeometry(pos().x(), pos().y(), boundingRect().width(),
        boundingRect().height());
}
```

Обратите внимание на то, что прямоугольник, который возвращает метод `geometry()` (и который мы создаем с помощью метода `setGeometry()`), содержит координаты виджета на плоскости сцены. Этим он отличается от прямоугольника, который возвращает метод `boundingRect()`.

Перейдем теперь к методам-обработчикам событий мыши (листинг 10.37).

Листинг 10.37. Методы-обработчики событий мыши

```
void EllipseWidget::mouseDoubleClickEvent(
    QGraphicsSceneMouseEvent *event)
{
    if (belongs(event->pos())) {
        emit doubleClicked();
        event->accept();
    }
}

void EllipseWidget::mousePressEvent(QGraphicsSceneMouseEvent *event)
{
    if (belongs(event->pos())) {
        emit mouseDown();
        event->accept();
    }
}

void EllipseWidget::mouseReleaseEvent(QGraphicsSceneMouseEvent *event)
{
```

```
    if (belongs(event->pos())) {
        emit mouseUp();
        event->accept();
    }
}
```

Мы обрабатываем такие события как двойной щелчок мышью, нажатие и отпускание кнопки мыши. Система Graphics View Framework не сообщает нам об одинарных щелчках мыши. Разумеется, получаемой виджетом информации достаточно для того, чтобы эмулировать обработку одинарных щелчков различными способами, но мы этого делать не будем. Обработав событие мыши, виджет эмитирует соответствующий сигнал, так же, как это делает, например, виджет `QPushButton`. Этот сигнал может быть связан с сигналом или слотом любого другого объекта программы, так что эти объекты могут взаимодействовать с графическими элементами напрямую, минуя Graphics View Framework.

В обработчиках событий мы решаем еще одну задачу. Очевидно, что виджет должен реагировать только на те щелчки мышью, которые происходят, когда указатель мыши находится внутри эллипса. Но, как мы уже знаем, наш виджет будет получать информацию о событиях мыши, которые происходят, когда указатель мыши находится в любой точке прямоугольника, возвращаемого методом `geometry()`. Иначе говоря, если не принять специальных мер, виджет `EllipseWidget` будет реагировать на события мыши, происходящие в тот момент, когда указатель мыши находится внутри прямоугольника, в который вписан эллипс, а не внутри самого эллипса. Для того чтобы виджет более точно реагировал на действия мыши, область, в которой обрабатываются события, нужно уточнить. Эту задачу можно решить несколькими способами, мы решаем ее с помощью функции `belongs()`, которая возвращает `true`, если координаты точки, переданные ей в качестве аргумента, находятся внутри эллипса, и `false` в противном случае. Если `belongs()` возвращает

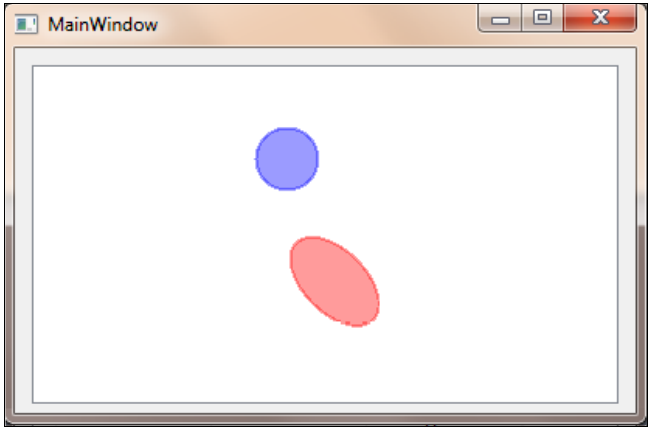


Рис. 10.14. Исходный виджет и результат его преобразования

`true`, эмитируется соответствующий сигнал и вызывается метод `accept()` объекта `event`.

Мы уже видели, что для правильной обработки событий виджета необходимо учитывать его координаты в системе координат графической сцены. А что будет, если виджет подвергнется другим преобразованиям, например, вращению? Для этого случая не надо принимать никаких специальных мер. Все наши методы будут работать корректно независимо от вращения и масштабирования виджета. Таким образом, у нас появляется виджет в виде эллиптической кнопки, которой не хватает только анимации (рис. 10.14).

ЧАСТЬ III

Дополнительный материал

Глава 11. Конечные автоматы и анимация

Глава 12. Сценарии для программ Qt

Глава 13. Динамические расширения программ Qt

Глава 14. Консольные приложения Qt

Глава 15. Язык QML

ГЛАВА 11

Конечные автоматы и анимация

В Qt 4.6 появилась совершенно новая подсистема — подсистема конечных автоматов (state machine framework). Я думаю, нет смысла лишний раз упоминать о том, какую роль автоматы играют в программировании. Работу многих программ без особых затруднений можно описать в терминах конечного автомата. Когда я пишу "без особых затруднений", я имею в виду, что диаграмма такого автомата будет понятна человеку. Вообще говоря, практически все программы могут быть представлены как конечные автоматы.

Мы рассмотрим ситуацию, когда представление работы программы в виде детерминированного конечного автомата имеет практический смысл. Мы уже написали такую программу. Это программа-калькулятор из второй главы. Работа этой программы, точнее работа класса `Calculator`, может быть представлена в виде сравнительно простого конечного автомата (рис. 11.1).

Состояния калькулятора (табл. 11.1) описываются тремя переменными: `Rf` — "регистр переднего плана" (число, которое видит пользователь калькулятора на экране), `Rb` — "регистр заднего плана" (второй операнд операции) и `Op` — переменная, в которой сохраняется знак операции. Будем обозначать первый операнд `x`, второй — `y`, а результат операции — `r`.

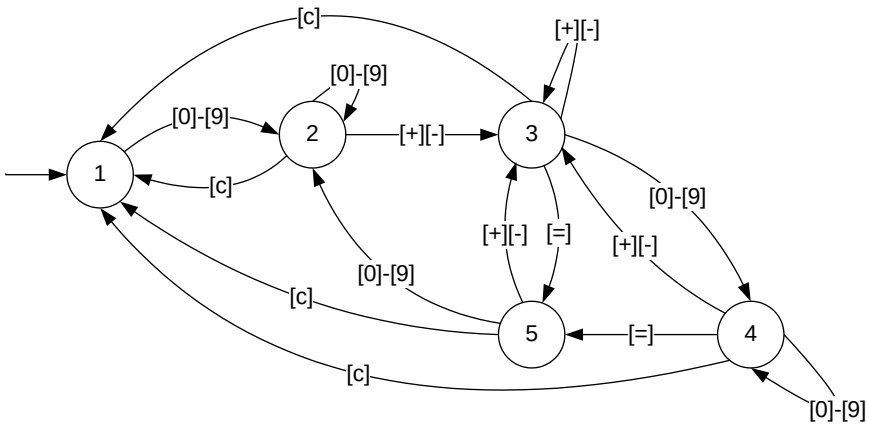


Рис. 11.1. Диаграмма конечного автомата класса `Calculator`

Таблица 11.1. *Состояния конечного автомата класса* `Calculator`

Состояние	Переменные
1	`Rf = 0, Rb = 0, Op = NULL`
2	`Rf = x, Rb = 0, Op = NULL`
3	`Rf = x, Rb = Rf, Op = [+|-]`
4	`Rf = y, Rb = x, Op = [+|-]`
5	`Rf = r, Rb = 0, Op = NULL`

При выполнении перехода `1->2` регистр `Rf` заполняется числом, вводимым пользователем. Остальные регистры остаются неизменными. При выполнении перехода `2->3` содержимое `Rf` копируется в `Rb`, а в переменную `Op` записывается знак операции. В процессе перехода `3->4` в регистр `Rf` записывается второй операнд, вводимый пользователем, а содержимое остальных регистров не меняется. В процессе перехода `4->5` выполняется операция над операндами `Rb` и `Rf` (соответственно либо `Rb + Rf`, либо `Rb - Rf`) и ее результат записывается в регистр `Rf`, а регистры `Rb` и `Op` становятся пустыми. Переход `5->2` эквивалентен переходу `1->2` (регистры `Rb` и `Op` при этом очищаются), переход `5->3` эквивалентен переходу `2->3`. В результате перехода `4->3` выполняется операция над текущими значениями `Rb` и `Rf`, ее результат записывается в `Rf`, в регистр `Rb` записывается содержимое `Rf`, а в регистр `Op` — новый символ операции. В результате перехода `3->3` над операндами `Rb` и `Rf` выполняется операция, символ которой в данный момент хранится в регистре `Op`, результат записывается в регистры `Rf` и `Rb`, а в регистр `Op` записывается символ новой операции. Переходы `2->1`, `3->1`, `4->1` и `5->1` опустошают все регистры калькулятора.

Для реализации конечного автомата нам понадобятся объекты трех классов: `QStateMachine`, `QState` и `QSignalTransition`. Думаю, тут все очевидно. Класс `QStateMachine` представляет автомат в целом. Класс `QState` представляет различные состояния автомата, а класс `QSignalTransition` — переходы между этими состояниями, инициируемые сигналами. С помощью перечисленных классов мы легко сможем построить диаграмму, показанную на рис. 11.1. Но для работы программы этого недостаточно. Наш автомат должен не только переходить из одного состояния в другое, он также должен выполнять различные операции. Рассмотрим, как все это выполняется на практике (листинг 11.1).

Листинг 11.1. Создание конечного автомата

```
s1 = new QState();
s2 = new QState();
s3 = new QState();
s4 = new QState();
s5 = new QState();
s1->addTransition(this, SIGNAL(digitButtonPressed()), s2);
s2->addTransition(this, SIGNAL(cancelButtonPressed()), s1);
```

```
s2->addTransition(this, SIGNAL(digitButtonPressed()), s2);
s2->addTransition(this, SIGNAL(operationButtonPressed()), s3);
s3->addTransition(this, SIGNAL(cancelButtonPressed()), s1);
s3->addTransition(this, SIGNAL(operationButtonPressed()), s3);
s3->addTransition(this, SIGNAL(digitButtonPressed()), s4);
s3->addTransition(this, SIGNAL(equalButtonPressed()), s5);
s4->addTransition(this, SIGNAL(cancelButtonPressed()), s1);
s4->addTransition(this, SIGNAL(digitButtonPressed()), s4);
s4->addTransition(this, SIGNAL(operationButtonPressed()), s3);
s4->addTransition(this, SIGNAL(equalButtonPressed()), s5);
s5->addTransition(this, SIGNAL(cancelButtonPressed()), s1);
s5->addTransition(this, SIGNAL(digitButtonPressed()), s2);
s5->addTransition(this, SIGNAL(operationButtonPressed()), s3);
connect (s1, SIGNAL(entered()), this, SLOT(s1Entered()));
connect (s1, SIGNAL(exited()), this, SLOT(s1Exited()));
connect (s2, SIGNAL(entered()), this, SLOT(s2Entered()));
connect (s2, SIGNAL(exited()), this, SLOT(s2Exited()));
connect (s3, SIGNAL(entered()), this, SLOT(s3Entered()));
connect (s3, SIGNAL(exited()), this, SLOT(s3Exited()));
connect (s4, SIGNAL(entered()), this, SLOT(s4Entered()));
connect (s4, SIGNAL(exited()), this, SLOT(s4Exited()));
connect (s5, SIGNAL(entered()), this, SLOT(s5Entered()));
connect (s5, SIGNAL(exited()), this, SLOT(s5Exited()));
machine = new QStateMachine(0);
machine->addState(s1);
machine->addState(s2);
machine->addState(s3);
machine->addState(s4);
machine->addState(s5);
machine->setInitialState(s1);
machine->start();
```

Переменные `s1-s5` представляют собой, как нетрудно догадаться, состояния нашего автомата. Для того чтобы установить переход между двумя состояниями, мы вызываем метод `addTransition()` объекта класса `QState`. Как уже было сказано, в нашем случае переходы инициируются сигналами (есть еще вариант, в котором переходы инициируются событиями Qt), поэтому первым аргументом метода `addTransition()` должен быть указатель на объект-источник сигнала, вторым аргументом — имя сигнала, а третьим аргументом — целевое состояние перехода. Таким образом, строка

```
s1->addTransition(this, SIGNAL(digitButtonPressed()), s2);
```

устанавливает переход между состояниями `s1` и `s2`, инициируемый сигналом `digitButtonPressed()` объекта, определяемого указателем `this`. Между двумя состояниями можно установить несколько переходов, инициируемых разными событиями. Сигналов, которые могут инициировать переходы, у нас четыре: `digitButtonPressed()` (была нажата цифровая клавиша); `operationButtonPressed()`

(была нажата клавиша операции); `cancelButtonPressed()` (была нажата клавиша отмены) и `equalButtonPressed()` (была нажата клавиша "равно"). Каждый вызов метода `addTransition()` создает новый объект класса `QSignalTransition` (или `QEventTransition`, в зависимости от того, какой из перегруженных вариантов метода был выбран). Указатели на этот объект возвращаются методами `addTransition()`, но поскольку нам сейчас нечего добавить к объекту, описывающему переход, мы ими не пользуемся. Дальнейшая установка переходов между состояниями в комментариях, я думаю, не нуждается.

Каждый раз, когда автомат переходит из одного состояния в другое, генерируются два сигнала. Состояние, из которого осуществляется переход, эмитирует сигнал `exited()`, тогда как целевое состояние эмитирует сигнал `entered()`. Благодаря этим сигналам мы можем связать с переходами определенные действия. Этот важный момент необходимо четко понимать: действия выполняются, когда автомат переходит из одного состояния в другое. Когда автомат находится в некотором состоянии, никакие действия не выполняются. Состояние просто указывает на то, что произойдет, когда на входе автомата появится очередное событие.

Теперь, когда все состояния связаны между собой переходами и назначены слоты для сигналов `entered()` и `exited()` состояний, мы создаем объект класса `QStateMachine`. Для того чтобы создать конечный автомат, мы добавляем состояния в объект класса `QStateMachine` с помощью метода `addState()` этого класса. Необходимо назначить начальное состояние, что мы и делаем с помощью метода `setInitialState()`. Согласно теории, у автомата должны быть (или, по крайней мере, желательно, чтобы были) допускающие (конечные) состояния. Для описания такого состояния в системе конечных автоматов Qt предусмотрен специальный класс `QFinalState`. Когда автомат переходит в это состояние, он эмитирует сигнал `finished()`. В то время как начальное состояние у автомата может быть только одно, конечных состояний (в полном соответствии с теорией) может быть несколько.

У автомата, описывающего работу класса `Calculator`, нет конечных состояний, и, как мы увидим, их нет у многих других классов, для которых предназначены конечные автоматы Qt. После того как автомат создан, мы вызываем метод `start()` объекта `QStateMachine` и автомат начинает выполнять свою работу.

Как уже отмечалось, полезные действия выполняются в слотах, связанных с сигналами `entered()` и `exited()`. Рассмотрим их (листинг 11.2). Полный текст программы вы найдете в папке Ch11/calculator.

Листинг 11.12 Слоты, обрабатывающие сигналы автомата

```
void Calculator::s1Entered()
{
    Rf = 0;
    Rb = 0;
    Op =  opNone;
    emit valueChanged(Rf);
}
```

```
void Calculator::s2Entered()
{
    if (Rf < 9999999) {
        Rf = Rf*10 + transitionButton;
        emit valueChanged(Rf);
    }
}

void Calculator::s3Entered()
{
    if (Rb != 0) {
        doOp(Op);
        emit valueChanged(Rf);
    }
    Rb = Rf;
    Op = transitionButton;
}

void Calculator::s3Exited()
{
    if (transitionButton > 9) {
        doOp(Op);
        Rb = 0;
        Op = transitionButton;
        emit valueChanged(Rf);
    } else  {
        Rf = 0;
    }
}

void Calculator::s4Entered()
{
    s2Entered();
}

void Calculator::s5Entered()
{
    doOp(Op);
    Op = opNone;
    emit valueChanged(Rf);
}

void Calculator::s5Exited()
{
    if (transitionButton <= 9) {
        Rb = 0;
        Rf = 0;
    }
}
```

Содержимое этих методов соответствует описанию переходов автомата, представленному в начале главы. Метод `doOp()` выполняет арифметическую операцию, символ которой передан ему в качестве аргумента. Операндами операции являются переменные `Rf` и `Rb`.

Конечные автоматы и минимизация повторяющегося кода

Реализация класса `Calculator` в виде конечного автомата выглядит сложнее, чем его исходная реализация. Это один из тех случаев, когда простой "книжный" пример не может продемонстрировать всех преимуществ технологии именно в силу своей простоты. Тем не менее вы можете представить себе ситуации, когда использование конечных автоматов существенно упрощает работу. Это особенно справедливо для тех случаев, когда сама постановка задачи включает в себя описание конечного автомата. Впрочем, конечные автоматы Qt обладают рядом специфических возможностей, которые делают их чрезвычайно полезными при создании новых виджетов или анимированных сцен Graphics View Framework.

Отметим также, что для реализации конечного автомата нам понадобилось меньше кода, чем можно было бы ожидать. Например, для всех переходов в состояние `s1` должно выполняться одно и то же действие. Поэтому мы просто записываем соответствующий код в слот `s1Entered()`. То же самое относится к состоянию `s5`. При переходе в это состояние должны выполняться одни и те же действия, независимо от источника перехода (а вот на выходе из этого состояния действия могут быть разными).

Вместе с тем тот факт, что для одного перехода вызываются два слота — один на выходе, другой на входе — может стать источником путаницы и ошибок при реализации конечного автомата. Иногда разумнее иметь один метод, полностью описывающий действия, связанные с конкретным переходом. Еще одно неудобство, присущее системе сигналов `exited()` и `entered()`, связано с тем, что в слотах, обрабатывающих эти сигналы, трудно определить, куда мы "уходим" и, соответственно, откуда мы "приходим". В нашем примере мы определяем это по косвенным признакам — состояниям переменных `Rb`, `Rf`, `Op` и специальной переменной `transitionButton`, которая содержит код клавиши, нажатие которой вызвало переход. Все это наталкивает на мысль об альтернативной реализации, в которой сигналы эмитировались бы не объектами, описывающими состояния, а объектами, описывающими переходы.

Готовых объектов для решения этой задачи в Qt нет, но ничто не мешает нам написать свои собственные. Заодно мы узнаем кое-что новое о системе сигналов и слотов Qt. Создание собственной системы объектов мы начнем с класса `QXtTransition`, который, как следует из его названия, является расширенной версией класса, описывающего переходы между состояниями (листинг 11.3).

Листинг 11.3. Класс `QXtTransition`

```
class QXtTransition : public QSignalTransition
{
    Q_OBJECT
public:
    explicit QXtTransition(QObject * sender, const char * signal,
        QObject * reciever, const char * slot,
        QState * sourceState = 0);
protected:
    void onTransition ( QEvent * event );
signals:
    void transiting(QState * from, QAbstractState * to, QString label);
};
```

Каждый раз, когда конечный автомат выполняет переход из одного состояния в другое, в объекте, описывающем переход, вызывается метод `onTransition()`, объявленный в разделе `protected:`. В классе `QXtTransition` мы перекрываем метод `onTransition()` для того, чтобы он эмитировал сигнал `transiting()`. У этого сигнала три параметра: параметр `from` указывает на объект-состояние, из которого выполняется переход. Параметр `to` указывает на целевое состояние перехода, а параметр `label` идентифицирует, какой именно переход `QXtTransition` стал источником сигнала (это сделано на тот случай, если один слот связан с несколькими сигналами `transiting()`). В переменную `label` записывается либо имя сигнала, вызвавшего переход, либо имя объекта `QXtTransition`, если таковое присвоено ему с помощью метода `setObjectName()`. Перейдем к реализации конструктора `QXtTransition()` и метода `onTransition()` (листинг 11.4).

Листинг 11.4. Реализация методов класса `QXtTransition`

```
QXtTransition::QXtTransition(QObject *sender, const char *signal,
        QObject *reciever, const char *slot, QState *sourceState):
    QSignalTransition(sender, signal, sourceState)
{
    int offset = (*slot == '0'+QSLOT_CODE) ? 1 : 0;
    const QMetaObject *meta = reciever->metaObject();
    int slotIndex;
    if ((slotIndex = meta->indexOfSlot(slot+offset)) == -1) {
        if ((slotIndex = meta->indexOfSlot(
            QMetaObject::normalizedSignature(slot+offset))) == -1)
        {
            qWarning("QXtTransition: no such slot %s::%s",
                meta->className(), slot+offset);
            return;
        }
    }
```

```
        offset = (*signal == '0'+QSIGNAL_CODE) ? 1 : 0;
    int signalIndex =
        this->metaObject()->indexOfSignal(
        "transiting(QState*,QAbstractState*,QString)");
    if (signalIndex == -1) {
        qWarning() << "QXtTransition: failed to find signal";
        return;
    }
    if (!meta->connect(this, signalIndex, reciever, slotIndex))
        qWarning() << "QXtTransition: failed to connect signal and slot";
}

void QXtTransition::onTransition(QEvent *e)
{
    QString label = objectName() ==
        "" ? QString::fromLatin1(signal().data(), signal().size())
        : objectName();
    emit transiting(sourceState(), targetState(), label);
    QSignalTransition::onTransition(e);
}
```

Самое интересное происходит в конструкторе. Для удобства мы хотим указывать слот, который будет связан с сигналом `transiting()`, прямо в конструкторе. Разумеется, ничто не мешает связать сигнал и слот традиционным способом, с помощью `connect()`. Итак, первый параметр конструктора — указатель на объект-источник сигнала, вызывающего переход (не забываем, что класс `QXtTransition` основан на классе `QSignalTransition`). Второй параметр — имя сигнала. За ним идет указатель на объект, которому принадлежит слот, с которым нужно связать сигнал `transiting()`. Следующий параметр — имя слота. В последнем параметре конструктора мы передаем указатель на объект, описывающий состояние-источник перехода.

Для решения задачи связывания сигнала и слота в конструкторе нам фактически придется сделать то, что делает функция `connect()` класса `QObject`. Мы хотим, чтобы в конструкторе класса слот, который будет связан с сигналом `transiting()`, передавался так же, как в методе `connect()`, т. е. с помощью макроса `SLOT()`. Этот макрос преобразует переданное ему имя слота в строку символов `char` с нулевым окончанием, имеющую определенный формат. Фактически наш конструктор получает указатель на объект и имя метода этого объекта. Чтобы взаимодействовать с этим методом, нам нужен его индекс, т. е. номер в описании методов объекта. Для каждого объекта Qt, наследующего `QObject`, можно получить метаобъект, который, в свою очередь, позволит получить об объекте данные, которые обычно доступны только во время компиляции программы. Константный указатель на метаобъект объекта `foo` можно получить, вызвав метод `foo.metaObject()`. Имея метаобъект для объекта `foo`, мы можем выполнить интроспекцию объекта `foo`, в том числе узнать, какими свойствами и методами обладает объект `foo`, и вызвать эти методы. Но сей-

час нам нужно другое. Индекс слота объекта можно найти с помощью метода `indexOfSlot()` соответствующего метаобъекта. Метод `indexOfSlot()` гораздо более строг к формату имени слота, чем макрос `SLOT()` (и это естественно, ведь `indexOfSlot()` предназначен для внутреннего употребления, многие программисты Qt никогда не имеют с ним дела). По этой причине нам приходится выполнять некоторые дополнительные операции, чтобы быть уверенными, что имя слота приведено к каноническому виду. Иначе метод `indexOfSlot()` не найдет этот слот, даже если он определен в соответствующем классе (в этом случае `indexOfSlot()` возвращает значение –1). Затем нужно найти индекс сигнала `transiting()` класса `QXtTransition`. Это делается практически так же, как в случае поиска индекса слота, за исключением того, что имя слота, которое задает программист, использующий наш класс, нам неизвестно, а имя сигнала мы знаем. Далее мы связываем сигнал и слот с помощью метода `connect()` метаобъекта. Этот метод отличается от одноименного метода класса `QObject`. Вместо имен сигнала и слота ему передаются индексы (именно для этого мы их и искали). Если связывание прошло успешно, метод возвращает значение `true` и мы можем себя поздравить.

В приведенном примере мы создавали объекты `QSignalTransition` неявно, при вызове метода `addTransition()`. Было бы неплохо создавать объекты `QXtTransition` таким же неявным способом. Но класс `QState`, естественно, этого сделать не может. Нам понадобится собственный класс-потомок класса `QState` с методом `addTransition()`, умеющим добавлять переходы `QXtTransition` (листинг 11.5).

Листинг 11.5. Класс `QXtState`

```
class QXtState : public QState
{
    Q_OBJECT
public:
    explicit QXtState(QState *parent = 0);
    void addTransition(QAbstractTransition * transition);
    QSignalTransition * addTransition(QObject * sender,
        const char * signal, QAbstractState * target);
    QAbstractTransition * addTransition(QAbstractState * target);
    QXtTransition * addTransition(QObject * sender, const char * signal,
        QObject * reciever, const char * slot, QState * target);
public slots:
protected:
};
```

Поскольку метод, объявленный в классе-потомке, делает невидимыми все перегруженные одноименные методы, мы не только добавляем новый метод `addTransition()`, но и перекрываем старые, чтобы наш класс, в случае необходимости, можно было использовать как обычный `QState`. С точки зрения реализации этот класс проще, чем предыдущий (листинг 11.6).

Листинг 11.6. Методы класса `QXtState`

```
QXtState::QXtState(QState *parent) :
    QState(parent)
{
}

void QXtState::addTransition(QAbstractTransition * transition)
{
    QState::addTransition(transition);
}

QSignalTransition * QXtState::addTransition(QObject * sender,
    const char * signal, QAbstractState * target)
{
    return QState::addTransition(sender, signal, target);
}

QAbstractTransition * QXtState::addTransition(QAbstractState * target)
{
    return QState::addTransition(target);
}

QXtTransition * QXtState::addTransition(QObject * sender,
    const char * signal, QObject * reciever, const char * slot,
    QState * target)
{
    QXtTransition * transition =
        new QXtTransition(sender, signal, reciever, slot, this);
    transition->setTargetState(target);
    addTransition(transition);
    return transition;
}
```

В методах `addTransition()`, повторяющих методы базового класса, мы просто передаем параметры методам базового класса. Интересен последний метод `addTransition()`, ради которого мы все и делали. В этом методе мы создаем объект класса `QXtTransition`. Поскольку в конструкторе класса необходим указатель на объект-приемник и слот, который будет связан с сигналом `transiting()`, их тоже необходимо передать в параметрах метода `addTransition()`. В конструкторе передается только указатель на объект, описывающий исходное состояние автомата (точно так же устроены конструкторы и других классов-потомков `QAbstractTransition`). Объект, описывающий переход, должен знать и целевое состояние, которое мы задаем с помощью метода `setTargetState()`. Далее мы вызываем унаследованный метод `addTransition()`, которому передаем указатель на созданный объект.

Рассмотрим работу описанных классов на примере простейшего конечного автомата (рис. 11.2). Автомат имеет два состояния и два перехода между ними. Создать реализацию этого конечного автомата в Qt очень просто (листинг 11.7).

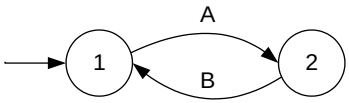


Рис. 11.2. Простейший конечный автомат

Листинг 11.7. Класс `QXtGraphicsView`

```
QStateMachine * machine1 = new QStateMachine(0);
    QXtState * xs1 = new QXtState();
    xs1->setObjectName("1");
    QXtState * xs2 = new QXtState();
    xs2->setObjectName("2");
    QXtTransition * t;
    xt = xs1->addTransition(ui->pushButton, SIGNAL(clicked()), this,
        SLOT(doTransiting(QState*,QAbstractState*,QString)), xs2);
    xt->setObjectName("A");
    xt = xs2->addTransition(ui->pushButton, SIGNAL(clicked()), this,
        SLOT(doTransiting(QState*,QAbstractState*,QString)),xs1);
    xt->setObjectName("B");
    machine1->addState(xs1);
    machine1->addState(xs2);
    machine1->setInitialState(xs1);
    machine1->start();
```

Переход из состояния 1 в состояние 2, так же как и обратный переход, инициируется сигналом `clicked()` объекта `pushButton`. С сигналами `transiting()` обоих объектов, описывающих переходы между состояниями, связывается один и тот же слот `doTransiting()` (листинг 11.8).

Листинг 11.8. Слот, обрабатывающий сигналы переходов

```
void Dialog::DoTransiting(QState *from, QAbstractState *to,
    QString label)
{
    qWarning() << "transit" << from->objectName() << to->objectName()
        << label;
}
```

В результате, щелкая по кнопке `pushButton`, мы получим на консоли примерно следующие строчки:

```
transit "1" "2" "A"
transit "2" "1" "B"
transit "1" "2" "A"
...
```

В таком режиме работа конечного автомата представляется более наглядно.

Конечные автоматы и анимация

Анимации, о которых тут пойдет речь, не имеют отношения к показу видеороликов. Анимации Qt предназначены для того, чтобы придать виджетам, меняющим свое состояние, более изящный вид. Попросту говоря, объект анимации Qt управляет тем, как меняется значение некоторой переменной в течение некоторого промежутка времени. Предположим, что мы хотим, чтобы некоторое окно нашей программы открывалось и закрывалось не мгновенно, а плавно. Мы можем решить эту задачу с помощью таймера. Через определенные промежутки времени таймер будет изменять значения свойства `size` окна, пока оно не достигнет некоего окончательного значения. Классы-потомки класса `QAbstractAnimation` значительно упрощают этот процесс.

Класс `QVariantAnimation` управляет значением переменной типа `QVariant`. С помощью различных методов этого класса можно указать начальное и конечное значение переменной (конкретные типы начальной и конечной переменной должны быть, разумеется, одинаковыми). Вы можете также задать время, в течение которого переменная должна перейти из начального значения в конечное, или наоборот. Соответственно задается и направление изменений. Кроме того, можно указать траекторию, по которой значение переменной будет меняться во времени (изменения могут происходить с постоянной скоростью, замедлением или ускорением, рассчитываемым по различным формулам). Изменение значений переменной можно также сделать циклическим.

Потомок класса `QVariantAnimation` класс `QPropertyAnimation` позволяет "анимировать" значения свойств объектов Qt. Здесь слово "анимация" уже приближается к своему общепринятому значению. Если с помощью объекта `QPropertyAnimation` управлять значениями таких свойств виджета как `pos`, `size` или `windowOpacity`, то виджет будет выглядеть анимированным в самом точном смысле этого слова. Иногда возникает необходимость анимировать сразу несколько объектов, причем мы должны быть уверены, что анимации выполняются синхронно. В этом нам помогут специальные классы, управляющие группами анимаций. Например, класс `QParallelAnimationGroup` гарантирует, что все управляемые им анимации будут выполняться параллельно. Анимации добавляются в группу методом `addAnimation()`. После этого группу можно запустить, так же как и отдельную анимацию, с помощью метода `start()`. Если мы хотим, чтобы несколько анимаций выполнялись одна за другой, следует воспользоваться классом `QSequentialAnimationGroup`.

Конечные автоматы Qt тесно связаны с анимацией. Каждый объект, описывающий переход из одного состояния в другое, можно связать с несколькими объектами,

контролирующими анимацию. В результате при выполнении перехода будут выполнены и соответствующие анимации. Хотя выполнением анимаций можно управлять и явным образом, с помощью методов-слотов `start()`, `stop()` и `setPaused()`, введенных в классе `QAbstractAnimation`, в случае использования анимаций совместно с конечными автоматами, запуском и остановкой анимаций должны управлять автоматы. Это тот случай, когда сигналы, которые генерируются соответствующими элементами конечного автомата, можно не связывать ни с какими слотами, но автомат все равно будет выполнять полезную работу.

В качестве примера мы рассмотрим управление виджетом с помощью простейшего конечного автомата, описанного ранее. В ответ на щелчок по кнопке наш виджет плавно открывает (или закрывает) окно календаря. Формирование конечного автомата для этого виджета представлено в листинге 11.9.

Листинг 11.9. Создание конечного автомата, управляющего анимацией

```
QXtState * xs1 = new QXtState();
QXtState * xs2 = new QXtState();
animation = new QPropertyAnimation(this);
animation->setTargetObject(ui->scrollArea);
animation->setPropertyName("size");
animation->setStartValue(QVariant(QSize(250,60)));
animation->setEndValue(QVariant(QSize(250,190)));
animation->setEasingCurve(QEasingCurve::OutQuad);
animation->setDuration(2500);
QXtTransition * t;
t = xs1->addTransition(ui->pushButton, SIGNAL(clicked()), this,
    SLOT(doTransiting(QState*)),xs2);
t->setObjectName("A");
t->addAnimation(animation);
t = xs2->addTransition(ui->pushButton, SIGNAL(clicked()), this,
    SLOT(doTransiting(QState*)),xs1);
t->setObjectName("B");
t->addAnimation(animation);
machine = new QStateMachine(this);
machine->addState(xs1);
machine->addState(xs2);
machine->setInitialState(xs1);
machine->start();
```

В этом фрагменте мы создаем объект класса `QPropertyAnimation`. В отличие от объектов, описывающих переходы и состояния, объект, управляющий анимацией, не становится собственностью конечного автомата, и мы должны самостоятельно контролировать его жизненный цикл. Как уже отмечалось, объект класса `QPropertyAnimation` предназначен для того, чтобы управлять изменением свойств объектов Qt. Метод `setTargetObject()` позволяет назначить объект, а метод

`setPropertyName()` — указать имя свойства, значением которого будет управлять объект `animation`. Объект класса `QPropertyAnimation` может управлять не только свойствами простых типов, таких как `int` или `qreal`, но и свойствами типов `QLine`, `QLineF`, `QPoint`, `QPointF`, `QRect`, `QRectF`, `QSize` и `QSizeF`. Естественно, анимированными могут быть только те свойства, которые не предназначены только для чтения (т. е. те, у которых в описании есть функция-сеттер). Метод `setStartValue()` задает начальное значение свойства, а метод `setEndValue()` — конечное значение. Продолжительность анимации в миллисекундах можно задать с помощью метода `setDuration()`, а траекторию величины во времени — с помощью метода `setEasingCurve()`. По умолчанию величина, контролируемая объектом-аниматором, меняется с постоянной скоростью. Мы передаем методу `setEasingCurve()` значение `QEasingCurve::OutQuad`. В этом режиме скорость изменения контролируемой величины линейно зависит от прошедшего времени.

Мы назначаем один и тот же объект-аниматор обоим переходам нашего конечного автомата. Это возможно потому, что объект-аниматор не является собственностью объектов конечного автомата. Заметим, что в процессе выполнения разных переходов анимация должна выполняться в разных направлениях. Изменить направление выполнения анимации, как уже было сказано, можно, но для этого необходимо знать, какой именно переход выполняется в данный момент. В этом нам поможет слот `doTransiting()`, который теперь объявлен так:

```
void doTransiting(QState* from);
```

Иначе говоря, из всех аргументов, которые получает слот, нас интересует только начальное состояние перехода.

> ***Примечание***
>
> Мы, конечно, могли бы воспользоваться сигналами `exiting()` объектов, описывающих состояния, но тогда нам понадобились бы два слота вместо одного.

Слот `doTransiting()` меняет направление анимации в зависимости от того, в каком направлении выполняется переход (листинг 11.10).

Листинг 11.10. Метод `mousePressEvent()` класса `GraphicsInput`

```
void MainWnd::doTransiting(QState *from)
{
    if (from->objectName() == "1")
        animation->setDirection(QAbstractAnimation::Backward);
    else
        animation->setDirection(QAbstractAnimation::Forward);
}
```

Значение `QAbstractAnimation::Forward` задает направление анимации от значения, заданного `setStartValue()`, к значению, заданному `setEndValue()`, тогда как значение `QAbstractAnimation::Backward` заставляет анимацию выполняться в противоположном направлении.

Примечание

Класс `QPropertyAnimation` опирается на механизм свойств, которые могут быть только у классов-потомков `QObject`. Для классов-потомков `QGraphicsItem`, которые таковыми не являются, существует специальный класс `QGraphicsItemAnimation`. Он был введен в Qt значительно раньше, чем остальные классы-аниматоры, и он не является потомком класса `QAbstractAnimation`.

В результате мы получим окно, в котором последовательные щелчки по кнопке будут приводить к закрытию и открытию окна календаря (рис. 11.3). Полный исходный текст программы вы можете найти в папке Ch11/animation.

Рис. 11.3. Программа с анимированным виджетом

Еще один вариант, при котором конечный автомат Qt может выполнять полезную работу, не связываясь с сигналами и слотами, предполагает использование метода `assignProperty()` класса `QState`. Этот метод позволяет сделать так, что при переходе автомата в определенное состояние заданное свойство указанного объекта Qt примет соответствующее значение. С одним состоянием можно связать несколько свойств объектов. Таким образом, переходы между состояниями могут управлять объектами Qt, минуя механизм сигналов и слотов. Главный недостаток этого метода заключается в том, что, как и в случае с сигналом `entered()`, мы не знаем, из какого состояния автомат перешел в данное состояние, а это может быть важно, если в данное состояние можно попасть разными путями.

ГЛАВА 12

Сценарии для программ Qt

Хорошая программа должна быть максимально настраиваемой. Это не означает, что пользователю придется выполнять сложную настройку при каждой установке программы, но разработчик должен обеспечить пользователю возможность, в случае необходимости или желания, максимально настроить программу "под себя" и расширить функциональность программы, хотя бы в некоторых пределах. Применение языков сценариев является одной из реализаций принципа разделения политики (набора правил поведения программы) и ее движка. Языки сценариев используются во множестве программ, начиная с офисных приложений и заканчивая играми. Некоторые языки программирования (например, Lua или ECMAScript) специально создавались для того, чтобы быть языками сценариев.

Языки сценариев стали интегрированной частью Qt, начиная с версии 4.3. За работу со сценариями отвечает модуль QtScript. Мы продемонстрируем работу QtScript на примере системы конфигурации программы. Главное преимущество системы сценариев (скриптов) по сравнению с обычными конфигурационными файлами заключается в гибкости сценариев. Используя систему сценариев как средство конфигурации программы, вы "бесплатно" получаете многие возможности, которые в противном случае пришлось бы реализовывать самостоятельно. Впрочем, в этой свободе таится и опасность. Иногда возможности системы сценариев оказываются слишком широкими, благодаря чему пользователь может намеренно или неумышленно нарушить безопасность программы.

В качестве примера мы напишем программу-часы (рис. 12.1), без кукушки, но зато с мощной системой настройки.

Модуль QtScript использует язык программирования ECMAScript, на котором основаны также языки JScript и JavaScript. Рассматривать подробно синтаксис языка

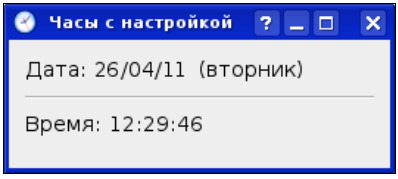


Рис. 12.1. Настраиваемые часы

ECMAScript мы не будем. Если вы имели дело с JavaScript, можно сказать, что вы этот язык уже знаете, остальным же я рекомендую познакомиться с описанием языка на сайте Qt. С точки зрения программы Qt центральным элементом системы сценариев является класс `QScriptEngine`, который, собственно, и инкапсулирует движок выполнения сценариев. Для того чтобы программа могла поддерживать сценарии, необходимо собрать ее с подключением модуля QtScript и создать хотя бы один объект класса `QScriptEngine`. Рабочая лошадка класса `QScriptEngine` — метод `evaluate()`. Именно этот метод выполняет сценарии.

Если вы просто хотите убедиться в том, что система сценариев работает, можете добавить в свою Qt-программу три строки, приведенные в листинге 12.1.

Листинг 12.1. Проверка работы системы QtScript

```
engine = new QScriptEngine();
qint32 result = engine->evaluate("2 + 2").toInt32();
qWarning() << "Result" << result;
```

В файл `pro` следует добавить директиву

```
QT += script
```

В результате выполнения программы в переменную `result` будет записано значение 4 (неожиданный результат, не правда ли?). Главным аргументом метода `evaluate()` (у него есть и другие аргументы, со значениями, заданными по умолчанию) является строка, содержащая текст сценария. Кроме того, начиная с версии 4.7, в Qt появился перегруженный вариант метода `evaluate()`, о котором будет сказано позже.

> ***Примечание***
>
> Из приведенного примера становится очевидным еще одно возможное применение сценариев Qt — создание калькуляторов. Только не таких калькуляторов, как многократно рассмотренная нами программа-калькулятор, а калькуляторов, способных вычислять сложные выражения, содержащие скобки и математические функции. Использование сценариев Qt для этой цели — не самый эффективный способ, т. к. система сценариев компилирует сценарий перед тем как его выполнить, но если калькулятор нужно сделать быстро и просто, система сценариев хорошо подойдет для этой цели.

Даже такое, предельно упрощенное, применение сценариев может оказаться полезным. С помощью сценариев вы можете встроить в свою программу калькулятор для вычисления математических выражений, заданных пользователем, и это будет калькулятор с очень большими возможностями, ведь за ним будет стоять вся мощь языка ECMAScript. Мы, однако, займемся более интересными вещами.

Для того чтобы оптимально использовать возможности системы сценариев в своей программе, вы должны понимать некоторые ключевые особенности работы языков встроенных сценариев (возможно, вы все это уже знаете, но и повторить, наверное, не помешает). Какими бы широкими возможностями ни обладал язык, он может быть полезен только в том случае, если сценарию доступны хотя бы некоторые

объекты приложения. Программа, использующая встроенные сценарии, должна создать среду окружения сценария, состоящую из объектов, функций и других элементов, необходимых сценарию. Например, система сценариев текстового редактора должна предоставлять сценариям доступ к объектам, с помощью которых сценарии смогут манипулировать редактируемым документом и вызывать команды редактора. Каким образом выполняется передача данных между приложением и сценарием? Для ответа на этот вопрос следует вспомнить, что переменные в языках сценариев, как правило, полиморфны (и язык ECMAScript не является исключением из этого правила). В зависимости от контекста одна и та же переменная языка сценариев может предоставлять доступ к своему значению как к числу, строке или объекту. Полиморфизм переменных означает, что все переменные следует рассматривать как принадлежащие одному полиморфному типу. Это свойство переменных важно не при написании самих сценариев. Значение любой переменной сценария QtScript может быть представлено в виде объекта класса `QScriptValue`. В зависимости от контекста вы можете извлечь из этого объекта значение того или иного простого (или сложного) типа. Объекты `QScriptValue` — своего рода посредники между сценарием и приложением. Все, что вы хотите передать из приложения в сценарий и обратно, может быть передано с их помощью. Например, метод `evaluate()` возвращает результат выполнения сценария в виде объекта `QScriptValue`. Этот объект может содержать значение простого типа (как в рассмотренном примере), объект с информацией об ошибке, если таковая имела место в процессе выполнения сценария, или пустое значение. Если последний пункт вам не очень понятен, можете провести аналогию с языком C, в котором функция может возвращать "значение" `void`. Информацию об ошибке, возникшей в ходе выполнения сценария, можно вывести, например, так, как показано в листинге 12.2.

Листинг 12.2 Выполнение сценария с обработкой возможных ошибок

```
QScriptValue result = engine->evaluate(script);
if (result.isError())
        qDebug() << "Script error:" << result.toString();
```

Статический метод `checkSyntax()` класса `QScriptEngine` позволяет проверить сценарий на предмет синтаксических ошибок, не выполняя его.

Передача ссылок на объекты Qt в сценарии

Вернемся к вопросу о создании окружения сценария. Для того чтобы передать в среду окружения сценария объект приложения, нужно сначала "упаковать" этот объект в объект `QScriptValue`, а затем присвоить ему имя в среде сценария. Для объектов, принадлежащих классам-потомкам `QObject`, это делается просто. Например, объект, реализующий главное окно программы, передается среде сценария так, как показано в листинге 12.3. Фрагмент взят из конструктора объекта главного окна.

Листинг 12.3. Передача объекта Qt в сценарий

```
QScriptValue objectWnd = engine->newQObject(this);
engine->globalObject().setProperty("MainWindow", objectWnd);
```

Методу `newQObject()` передается указатель на объект, который нужно упаковать в объект `QScriptValue` (у метода есть и другие аргументы, которые мы оставляем со значениями, заданными по умолчанию). Обратите внимание на то, что метод `newQObject()` предназначен для работы с объектами классов-потомков `QObject`, что и отражено в его названии. Во второй строке мы присваиваем имя новому объекту в среде сценария и назначаем ему глобальную область видимости. В результате объект, реализующий главное окно, будет доступен в сценарии под именем `MainWindow`.

Для тех, кто хочет добраться до сути вещей, приведу более строгое объяснение того, что происходит в последней строке примера. Язык ECMAScript является полностью объектно-ориентированным. Иначе говоря, весь код сценария выполняется в контексте некоего глобального объекта. Глобальный объект создается автоматически движком сценария и доступен в Qt-приложении через свойство `globalObject()` объекта `QScriptEngine` (как и все, что связано со сценариями, глобальный объект среды сценариев представлен в Qt-приложении объектом класса `QScriptValue`). Метод `setProperty()` создает новое свойство глобального объекта. В нашем примере это свойство имеет имя `MainWindow` и хранит ссылку объект, реализующий главное окно Qt-приложения.

Передача объекта приложения в среду окружения сценария предполагает, что программа-сценарий сможет получить доступ к элементам этого объекта. Возможно, это покажется странным, но не все методы переданного сценарию объекта Qt доступны в контексте сценария. Причина заключается в том, что сценарий может получить доступ только к тем элементам объекта, описания которых доступны во время выполнения программы, тогда как в языке C++ информация о полях и методах объекта доступна только на этапе компиляции. У этой проблемы существует несколько решений. Самое простое из них заключается в использовании информации о типах, доступной во время выполнения программы (RTTI), т. е. через уже знакомую нам систему метаобъектов, которая реализована для объектов классов, производных от класса `QObject`. С помощью этого механизма во время работы программы можно получить информацию о сигналах, слотах и свойствах объектов Qt неизвестного типа. Таким образом, сценарий может получить доступ к сигналам, слотам и свойствам объекта класса-потомка `QObject`, но не к обычным полям и методам этого объекта. Если мы создаем новый класс, объекты которого должны быть переданы сценарию, мы должны позаботиться о том, чтобы нужные элементы класса были видимы в сценарии. Рассмотрим объявление класса `Clock`, который реализует главное окно программы-часов (листинг 12.4). Полный текст программы вы найдете в папке Ch12/clock.

Листинг 12.4. Класс `Clock`

```
class Clock : public QDialog, public Ui::MainForm
{
    Q_OBJECT
public:
       Clock(QWidget *parent = 0);
       virtual ~Clock();
       Q_PROPERTY(QString timeFormat READ getTimeFormat
            WRITE setTimeFormat)
       Q_PROPERTY(QString dateFormat READ getDateFormat
            WRITE setDateFormat)
       Q_PROPERTY(QString currentTime READ getCurrentTime)
protected slots:
       void onTimeout();
       void beep();
private:
       QTimer * timer;
       QString m_timeFormat;
       QString m_dateFormat;
       void setTimeFormat(const QString & format)
       {
              m_timeFormat = format;
       };
       QString getTimeFormat()
       {
              return m_timeFormat;
       };
       void setDateFormat(const QString & format)
       {
       m_dateFormat = format;
       };
       QString getDateFormat()
       {
              return m_timeFormat;
       };
       QString getCurrentTime()
       {
              return QTime::currentTime().toString(m_timeFormat);
       };
       QScriptEngine * engine;
};
```

В этом классе мы объявляем три свойства: `timeFormat`, `dateFormat` и `currentTime`. Первые два свойства позволяют установить формат отображения даты и времени на наших часах. Свойство `currentTime`, доступное только для чтения, позволяет по-

лучить текущее значение времени. Обратите также внимание на слот `beep()`. Этот слот позволяет послать звуковой сигнал, используя статический метод `QApplication::beep()`. Благодаря этому программу-часы можно использовать и как будильник. Еще один способ сделать метод объекта видимым в сценарии — использовать в объявлении метода макрос `Q_INVOKABLE`:

```
Q_INVOKABLE void visibleMethod();
```

Самое интересное, как это часто бывает, происходит в конструкторе класса `Clock` (листинг 12.5).

Листинг 12.5. Конструктор класса `Clock`

```
Clock::Clock(QWidget *parent) : QDialog(parent)
{
        setupUi(this);
        m_timeFormat = "hh:mm:ss";
        m_dateFormat = "yyyy-MM-dd (dddd)";
        QFile scriptFile("clock.config");
        timer = new QTimer(this);
        timer->setObjectName("timer");
        timer->setInterval(1000);
        connect(timer, SIGNAL(timeout()), this, SLOT(onTimeout()));

        QString script;
        if (scriptFile.exists())
        {
                scriptFile.open(QIODevice::ReadOnly);
                script.append(scriptFile.readAll());
                scriptFile.close();
        }
        engine = new QScriptEngine();

        QScriptValue objectWnd = engine->newQObject(this);
        engine->globalObject().setProperty("MainWindow", objectWnd);
        QScriptValue func = engine->newFunction(toUnicode);
        engine->globalObject().setProperty("toUnicode", func);
        QScriptValue func2 = engine->newFunction(newTextInputDialog);
        engine->globalObject().setProperty("newTextInputDialog", func2);

        QScriptValue result = engine->evaluate(script);
        if (result.isError())
                qDebug() << "Script error:" << result.toString();

        timer->start();
}
MainWindow.timeFormat = "hh:mm:ss";
MainWindow.dateFormat = "MM/dd/yy (dddd)";
```

Теперь, когда мы знаем, как передать сценарию объект, рассмотрим текст простейшего сценария (листинг 12.6). Программа clock читает сценарии конфигурации из файла clock.config. Пользователи Microsoft Visual Studio могут изменить расширение файла на какое-нибудь другое, т. к. в Visual Studio файлы с расширением config имеют особый смысл. В принципе, программы на языках сценариев, как и любые другие ресурсы программ Qt, можно сделать частью исполнимого файла программы или отдельного скомпилированного модуля ресурсов. Но в этом случае теряется одно из главных преимуществ сценариев — возможность модифицировать их с помощью простого текстового редактора без повторной сборки программы.

Листинг 12.6. Простейший сценарий

```
MainWindow.timeFormat = "hh:mm:ss";
MainWindow.dateFormat = "MM/dd/yy (dddd)";
MainWindow.windowTitle = "Часы с настройкой";
```

Первые две строчки не должны вызывать вопросов. Мы используем определенные нами свойства `timeFormat` и `dateFormat` для указания формата вывода даты и времени с помощью принятых в Qt символов-спецификаторов. Поскольку в файлах сценариев ваши пользователи наверняка захотят использовать кириллицу, уместно затронуть вопрос о кодировках. "Родной" кодировкой движка сценариев Qt является UTF-16, но мы будем пользоваться, как обычно, UTF-8, поскольку эта кодировка доступна на максимальном количестве платформ. Это означает, между прочим, что при работе с символами кириллицы в текстах программ-сценариев нам понадобится поменять кодировку текста с UTF-8 на UTF-16, точно так же, как мы делаем это в исходных текстах программ Qt. Для этой цели создадим функцию `toUnicode()`, о которой поговорим позже.

Обратите внимание на последнюю строчку примера. Мы не определяли свойство `windowTitle`. Это свойство унаследовано классом `Clock` от его предка. Таким образом, в нашем сценарии настройки программы мы получили дополнительную возможность, о которой заранее не думали. Как уже отмечалось, свобода, которую предоставляют сценарии, таит в себе потенциальный риск.

Если добавить в сценарий строчку

```
MainWindow.close();
```

сценарий завершит работу нашей программы. Метод `close()` класса `Clock` является слотом, а значит, будет доступен из сценария, как метод объекта `MainWindow`. Оказывается, сценарий получает доступ к гораздо большему количеству элементов объекта класса `Clock`, чем мы предполагали. Но и это еще не все. Посмотрите на следующую конструкцию:

```
MainWindow.dateText.text = "Date:";
```

Что такое свойство `dateText` объекта `MainWindow`? Имя `dateText` присвоено в нашей программе объекту `QLabel`, который выводит пояснение к строке с текущей датой (под именем объекта подразумевается не имя переменной, в которой хранится ука-

затель на объект, а значение свойства `name()` данного объекта). В приведенной строке мы присваиваем новое значение свойству `text` объекта `dateText`. В результате внешний вид окна программы изменится. Здесь необходимо небольшое замечание относительно синтаксиса. Если мы говорим о свойствах в контексте Qt, то ставим скобки после имени свойства, (например, `name()`), что соответствует синтаксису Qt. Когда же речь заходит о свойствах в контексте сценария, имена свойств пишутся без конечных скобок, в соответствии с синтаксисом языка сценария.

Но и это еще не все. С помощью нашей системы конфигурации смышленый пользователь сможет изменить внешний вид программы до неузнаваемости (рис. 12.2). Сделать это совсем несложно (листинг 12.7).

Листинг 12.7. Сценарий, модифицирующий внешний вид окна программы

```
MainWindow.dateLabel.hide();
MainWindow.dateText.hide();
MainWindow.setStyleSheet("QDialog { background: yellow }");
MainWindow.timeLabel.setStyleSheet("QLabel {border: 2px solid green;}");
MainWindow.timeText.setStyleSheet("QLabel {border: 2px solid red;}");
```

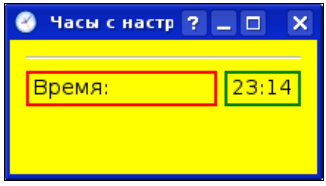


Рис. 12.2. Модифицированное окно программы-часов

Может быть, такая модификация программы и не противоречит нашим намерениям, но мы и не предоставляли эту возможность явным образом.

Таким образом, мы видим, что сценарий может получить доступ не только ко всем свойствам, сигналам и слотам переданного ему объекта, но и ко всем именованным (т. е. с непустым свойством `name()`) дочерним объектам данного объекта. В некоторых случаях это полезно, в других — нет. Например, если программа открывает диалоговое окно, а сценарий имеет доступ к соответствующему объекту, сценарий может "щелкнуть" одну из кнопок (вызвав сигнал `clicked()`) вместо пользователя. Из всего этого следует, что вообще говоря, для обмена данными со сценарием лучше всего создавать специальный объект, предоставляющий только те свойства, сигналы и слоты, которые должны быть доступны сценарию.

Доступ сценария к элементам объекта можно ограничить и другим способом. Третий аргумент метода `newQObject()` позволяет определить, какие из элементов объекта будут доступны в сценарии (листинг 12.8).

Листинг 12.8. Ограничение доступа сценария к объектам программы

```
QScriptValue objectWnd = engine->newQObject(this,
    QScriptEngine::QtOwnership, QScriptEngine::ExcludeSuperClassMethods |
    QScriptEngine::ExcludeSuperClassProperties);
```

После этого вызов метода

```
MainWindow.close();
```

приведет к сообщению об ошибке

```
ReferenceError: close is not defined
(Ошибка вызова метода: close не определен)
```

Добавление в третий аргумент константы `QScriptEngine::ExcludeChildObjects` приведет к тому, что программа-сценарий не сможет получить доступ к дочерним объектам данного объекта.

Подчеркну один важный аспект работы с объектами в сценариях. По умолчанию сценарий получает ссылку на объект приложения. При этом приложение остается владельцем объекта и оно же контролирует время его жизни. Далее мы рассмотрим процесс создания объектов, которые целиком принадлежат сценарию.

Обработка сигналов в сценарии

Как мы уже видели, сценарий может вызывать любой сигнал или слот любого доступного ему объекта. Не менее интересна и другая возможность — использование функции, определенной в сценарии, в качестве обработчика сигнала объекта Qt. Рассмотрим фрагмент сценария настройки нашей программы (листинг 12.9).

Листинг 12.9. Функция-слот в сценарии

```
function onTimer()
{
var time = MainWindow.currentTime;
if (time.substring(0,5) == "12:00")
  MainWindow.beep();
}
```

Функция `onTimer()` должна стать обработчиком сигнала `timeout`, который периодически генерируется объектом-таймером в нашей программе. С помощью свойства `currentTime` функция `onTimer()` получает текущее значение времени в виде строки (по умолчанию в формате ЧЧ:ММ:СС), извлекает первые пять символов (часы и минуты) значения времени и ровно в полдень запускает звуковой сигнал, который будет активен минуту.

После того как функция-обработчик объявлена, ее следует связать с сигналом таймера:

```
MainWindow.timer.timeout.connect(onTimer);
```

В Qt-программе объекту-таймеру присвоено имя `timer`, под которым он и доступен сценарию, как одно из свойств объекта `MainWindow`. Сигнал `timeout` объекта `timer` представляется в сценарии как свойство-объект, у которого есть методы `connect()`

и `disconnect()`. Первый метод выполняет связывание обработчика и сигнала, второй метод разрывает эту связь. Необходимо понимать разницу между обращениями:

```
MainWindow.timer.timeout()
```

и

```
MainWindow.timer.timeout
```

В первом случае мы вызываем сигнал `timeout` таймера, что приведет к вызову связанных с ним слотов. Во втором случае мы получаем доступ к свойству-объекту, которое позволяет нам управлять параметрами обработки сигнала.

Благодаря возможности назначать функции, объявленные в сценарии, в качестве обработчиков сигнала таймера мы превращаем нашу программу-часы в будильник с весьма широкими функциями настройки.

> ***Примечание***
>
> А не проще ли в самой программе сделать окно для указания времени срабатывания будильника? Конечно проще. Но это та самая простота, которая хуже воровства. Действуя таким путем, мы действительно обкрадываем пользователя, лишая его возможности задавать более сложные условия срабатывания будильника, которые можно установить в сценарии и нельзя установить в диалоговом окне. Впрочем, сказанное не отменяет пользы простых будильников.

В случае с обработкой сигналов в сценарии функция-обработчик вызывается (что вполне логично) и после выхода из функции `evaluate()`. При этом возникает еще один интересный вопрос: сколько времени существует сценарий? Ответ на него тоже вполне логичен: сценарий существует столько же, сколько существует его контекст, в том числе и глобальный объект. По умолчанию время жизни контекста сценария совпадает со временем существования объекта `QScriptEngine`.

А как эмитировать сигналы Qt из сценария? На вопрос "зачем это нужно?" отвечает наш пример с будильником. Пусть мы хотим, чтобы при наступлении момента времени, установленного пользователем через сценарий, будильник не подавал звуковые сигналы, а посылал программе сигнал Qt. Если в объекте, переданном контексту сценария, объявлены сигналы, сценарий может заставить объект эмитировать эти сигналы, вызвав сигнал, как простую функцию. Например, если в классе `Clock` объявить сигнал:

```
void alarm(int reason);
```

то для вызова сигнала из сценария достаточно будет написать:

```
MainWindow.alarm(1234);
```

Если сигнал `alarm()` связан с каким-либо слотом, то будет вызван этот слот и ему будет передан соответствующий параметр (значение 1234).

Использование функций приложения в сценарии

Qt-приложение может передавать в контекст сценария не только ссылки на объекты, но и ссылки на функции, не являющиеся методами объектов. Для передачи ссылки на функцию в контекст сценария необходимо выполнить те же действия, что и для передачи ссылки на объект — упаковать указатель на функцию в объект `QScriptValue` и назначить функции имя и область видимости в контексте сценария. Функция приложения, предназначенная для использования в сценарии, выглядит по-разному с точки зрения сценария и приложения. Поскольку прототип функции в сценарии отсутствует, при вызове в сценарии этой функции можно передать любое количество аргументов любого типа. При определении функции в приложении у нее должно быть, как минимум, два параметра: указатели на объекты `QScriptContext` и `QScriptEngine`.

В качестве примера напишем функцию `toUnicode()`, преобразующую текст сценария из кодировки (UTF-8) в кодировку UTF-16, которой пользуется Qt. То же самое можно сделать с помощью функций языка ECMAScript, но нам интереснее реализовать собственную функцию (листинг 12.10).

Листинг 12.10. Функция `toUnicode()`

```
QScriptValue toUnicode(QScriptContext *context, QScriptEngine *engine)
{
    QString s = context->argument(0).toString();
    return QScriptValue(engine, QString::fromUtf8(s.toAscii().data()));
}
```

Когда сценарий вызовет нашу функцию, все переданные функции аргументы будут сохранены в специальной коллекции объекта `context` (общее количество переданных аргументов можно получить с помощью метода `argumentCount()` объекта `context`). Все переданные сценарием аргументы, как и возвращаемое функцией значение, являются, естественно, объектами `QScriptValue`. Для присваивания функции имени в сценарии мы вызываем следующие методы объекта `QScriptEngine`:

```
QScriptValue func = engine->newFunction(toUnicode);
engine->globalObject().setProperty("toUnicode", func);
```

Метод `newFunction()` класса `QScriptEngine` создает объект `QScriptValue`, содержащий указатель на функцию. Вторая строка нам уже знакома — мы делаем новую функцию свойством глобального объекта сценария. Теперь мы можем использовать в сценарии функцию `toUnicode()`, например:

```
MainWindow.windowTitle = toUnicode("Часы с настройкой");
```

Поскольку никаких ограничений на количество и тип аргументов при вызове функции в сценарии не существует, функция, определенная в приложении, должна выполнять проверку на соответствие переданного списка аргументов ожидаемому.

Хотя такое и маловероятно, тщательно подобранные злоумышленником аргументы при отсутствии проверки могут вызвать исключительную ситуацию в приложении, выполняющем сценарий.

Создание объектов в сценарии

Сценарии Qt могут не только пользоваться объектами, предоставленными им приложением, но и создавать свои собственные объекты. Сценарии могут создавать объекты на основе описаний типов, хранимых в самих сценариях, а также создавать экземпляры объектов того приложения, в котором они выполняются. Последняя задача гораздо важнее с практической точки зрения, поэтому мы сосредоточимся на ней.

Есть два пути создания объектов в сценарии — простой и сложный. Простой путь заключается в создании в приложении-хозяине сценария функции-фабрики объектов. Когда сценарию требуется создать объект, он вызывает функцию-фабрику, которая возвращает ему готовый объект. Этот метод настолько прост и удобен, что во многих случаях вам, кроме него, ничего не понадобится.

Второй способ сложнее. Суть его заключается в том, что мы вводим в контекст программы-сценария новый тип данных, соответствующий тому объекту прложения, с которым должен работать сценарий. В результате сценарий получает более широкие возможности в работе с объектом. Рассмотрим, например, вопрос интроспекции. Как вы уже знаете, программа-сценарий может получить доступ к публичным слотам и свойствам объекта, используя механизм метаобъектов. Но это возможно только в том случае, если объект принадлежит классу, который является потомком `QObject`. Второй же способ, благодаря которому мы фактически создаем для сценария полноценный тип данных, отображающий в контексте сценария класс C++, позволяет, в принципе, использовать в сценарии объекты классов C++, не являющихся потомками `QObject`.

Мы рассмотрим сначала более простой способ. В демонстрационном приложении clock определен класс `TextInputDialog`, который создает диалоговое окно со строкой для ввода текста, пояснением к строке и кнопками **OK** и **Отмена**. Это диалоговое окно можно использовать в сценариях настройки программы clock для ввода текста во время выполнения сценария. Для того чтобы сценарий мог создавать объект класса `TextInputDialog` тогда, когда ему нужно, в приложении clock мы объявляем функцию-фабрику (листинг 12.11).

Листинг 12.11. Функция `newTextInputDialog()`

```
QScriptValue newTextInputDialog(QScriptContext *context,
    QScriptEngine *engine)
{
        return engine->newQObject(new TextInputDialog(),
            QScriptEngine::ScriptOwnership);
}
```

Список параметров у этой функции такой же, как и у любой другой функции, предназначенной для вызова из сценария. В теле функции мы создаем объект класса `QScriptValue`, инкапсулирующий объект класса `TextInputDialog`. Этот объект `QScriptValue` и возвращается в качестве результата функции. Обратите внимание на второй аргумент метода `newQObject()`. Константа `QScriptEngine::ScriptOwnership` указывает, что создаваемый объект должен принадлежать сценарию (еще один важный момент: объект, создаваемый специально для сценария, не должен иметь родительского объекта в приложении, т. е. его свойство `parent` должно содержать значение `NULL`). Мы делаем функцию `newTextInputDialog` доступной сценарию так же, как и в случае функции `ToUnicode()`. Теперь в файле сценария можно работать с простым диалоговым окном (листинг 12.12).

Листинг 12.12. Создание диалогового окна в файле сценария

```
var Dlg = newTextInputDialog();
var Text;
function onAccept()
{
        Text = Dlg.lineEdit.text;
        print(Text);
}
Dlg.accepted.connect(onAccept);
Dlg.modal = true;
Dlg.show();
```

Мы создаем объект `TextInputDialog` и сохраняем ссылку на него в переменной `Dlg` (поскольку адресное пространство сценарию недоступно, уместно говорить именно о ссылках, а не об указателях). Сигнал `accepted` диалогового окна (этот сигнал эмитируется, если пользователь щелкнул кнопку **OK**) мы связываем с обработчиком `onAccept`. В обработчике мы считываем строку, введенную пользователем (свойство `text` объекта `lineEdit`), и выводим ее в окно консоли с помощью функции `print()`.

В чем разница между передачей объектов в сценарий и созданием объектов в ходе выполнения сценария? В первом случае сценарий получает объект как некую данность. Во втором случае сценарий сам решает, когда объект должен быть создан. В связи с этим может возникнуть вопрос: создавать объекты мы научились, но почему не предусмотрен метод для их удаления? Все дело в том, что такой метод просто не нужен. Среда выполнения сценария использует механизм управления памятью, известный как "сборка мусора". Динамически созданный объект будет удален автоматически, когда все видимые переменные перестанут ссылаться на него.

Создание новых типов данных в сценарии

Как вы уже поняли, объекты, принадлежащие классам, производным от `QObject`, и объекты, таковым классам не принадлежащие, интегрируются в систему сценариев Qt совершенно по-разному. Для объекта, среди предков класса которого есть

QObject, все просто. Достаточно ввести этот объект в контекст сценария с помощью фабрики объектов, и множество его методов и свойств (иногда даже больше, чем хотелось бы) становится доступно в программе-сценарии. С объектами, которые не ведут свою родословную от QObject, дело обстоит иначе. По умолчанию для сценария эти объекты представляют собой совершенно непрозрачные структуры данных. Для их интеграции в сценарии придется затратить дополнительные усилия. Мы можем сравнить оба подхода на примере создания сценариев для приложения, использующего Graphics View Framework. Для иллюстрации первого подхода воспользуемся классом EllipseWidget, который мы создали ранее. Для иллюстрации второго подхода мы интегрируем в систему сценариев класс QColor, который не является потомком QObject и потому не содержит свойств, сигналов и слотов, т. е. всего того, что система сценариев Qt может увидеть по умолчанию.

Объекты EllipseWidget добавляются в контекст сценария достаточно просто (листинг 12.13).

Листинг 12.13. Добавление объекта класса EllipseWidget в контекст сценария

```
QGraphicsScene * scene = new QGraphicsScene();
QScriptEngine * engine = new QScriptEngine();
EllipseWidget * ew = new EllipseWidget();
scene->addItem(ew);
engine->globalObject().setProperty("Ellipse1", engine->newQObject(ew));
```

В результате в контексте сценария появится объект Ellipse1, для которого будут доступны, помимо прочего, свойства x, y, rx, ry и метод setColor(). Если со свойствами x, y, rx и ry не возникнет никаких проблем, т. к. их значениями являются простые числа, то вызвать метод setColor() просто так не получится, ведь ему необходимо передать значение типа QColor, а о таком типе данных язык сценариев не знает.

Для решения этой проблемы вспомним еще раз особенности представления переменных в языке сценариев Qt. Как уже было сказано, переменные языка сценариев полиморфны, т. е. одна и та же переменная может, в зависимости от контекста, принимать значения разных типов. Это возможно потому, что в языке сценариев все переменные имеют один тип Object, который можно сравнить с типом QVariant. Собственно, типу Object языка сценариев и соответствует тип QScriptValue, который, в некотором смысле, более полиморфен, чем QVariant. Когда из программы-сценария вызывается метод объекта, класс которого определен в Qt, и которому требуется передать параметр, выполняется "переупаковка" данных. Переменная сценария, имеющая тип Object, приводится (если это возможно) к типу параметра объекта Qt.

Таким образом, нам нужно научиться создавать переменные языка сценариев, которые можно было бы привести к типу QColor. Желательно, чтобы синтаксис выражения для создания переменной соответствовал синтаксису языка сценариев, например:

```
var color = new QColor(255, 255, 0);
```

Решение задачи оказывается неожиданно простым. Это делается с помощью функции-фабрики объектов. Напишем функцию `QColor_ctor` (листинг 12.14).

Листинг 12.14. Функция-фабрика для объектов `QColor`

```
Q_DECLARE_METATYPE(QColor)
Q_DECLARE_METATYPE(QColor*)
QScriptValue QColor_ctor(QScriptContext *context, QScriptEngine *engine)
 {
     int r = context->argument(0).toInt32();
     int g = context->argument(1).toInt32();
     int b = context->argument(2).toInt32();
    return engine->toScriptValue(QColor(r, g, b, 80));
}
...
engine->globalObject().setProperty("QColor",
engine->newFunction(QColor_ctor))
```

Эта функция возвращает значение типа `QScriptValue`, которое в контексте сценария трансформируется в значение типа `Object`. Важно здесь то, что система знает, что данное значение может быть приведено к типу `QColor`, а значит, его можно передавать в качестве параметра функциям, которым необходим параметр типа `QColor`. Теперь в файле сценария можно написать:

```
var color = new QColor(r, g, b);
Ellipse1.setColor(color);
```

Фактически мы просто создали в контексте сценария функцию `QColor()`. Однако обратите внимание на различие между функцией `QColor()` и функцией `newTextInputDialog()` (листинг 12.11). Функцию `newTextInputDialog()` нельзя использовать совместно с оператором `new`, а функцию `QColor()` — можно. Таким образом, функция `QColor_ctor()` фактически создает "обертку" для объекта `QColor`, которой можно пользоваться в сценарии. Система сумеет, при необходимости, извлечь объект `QColor` из этой "обертки".

Теперь мы можем создавать в сценарии переменные типа `QColor` и передавать их методам классов и функциям, но ни для чего другого эти переменные не годятся. С точки зрения языка сценариев такая переменная содержит что-то неизвестное и никаких иных операций выполнять с ней нельзя. Вместе с тем тип Qt `QColor` позволяет манипулировать составляющими цвета уже после того, как переменная создана. Неплохо было бы добавить к реализации объекта `QColor` в языке сценариев свойства `red`, `green` и `blue`, которые бы позволяли получать и модифицировать, соответственно, красную, зеленую и синюю составляющие цвета. Можно добавить еще и свойство `alpha` для канала прозрачности, но мы не будем этого делать, чтобы не усложнять листинги без необходимости. Если бы класс `QColor` был потомком `QObject`, мы могли бы определить соответствующие свойства очень просто, с помощью методов класса (сеттеров и геттеров). Для придания свойств типу `QColor` в

языке сценариев мы поступим аналогично, только воспользуемся не методами, а функциями (листинг 12.15). Полные тексты вы найдете в папке Ch12/QColorMetatype на сайте **http://symmetrica.net/qt47book/**.

Листинг 12.15. Добавление свойств к типу `QColor`

```
Q_DECLARE_METATYPE(QColor)
Q_DECLARE_METATYPE(QColor*)

QScriptValue QColor_prototype_red(QScriptContext *context,
    QScriptEngine *engine)
{
    QColor color = qscriptvalue_cast<QColor>(context->thisObject());
    return color.red();
}

QScriptValue QColor_prototype_setRed(QScriptContext *context,
    QScriptEngine *engine)
{
    QColor * color = qscriptvalue_cast<QColor *>(context->thisObject());
    if (!color)
        return context->throwError(QScriptContext::TypeError,
            "This object is not a QColor");
     color->setRed(context->argument(0).toInt32());
     return engine->undefinedValue();
 }

QScriptValue QColor_prototype_green(QScriptContext *context,
    QScriptEngine *engine)
{
    QColor color = qscriptvalue_cast<QColor>(context->thisObject());
    return color.green();
}

QScriptValue QColor_prototype_setGreen(QScriptContext *context,
    QScriptEngine *engine)
{
     QColor * color = qscriptvalue_cast<QColor *>(context->thisObject());
     if (!color)
         return context->throwError(QScriptContext::TypeError,
            "This object is not a QColor");
     color->setGreen(context->argument(0).toInt32());
     return engine->undefinedValue();
}

QScriptValue QColor_prototype_blue(QScriptContext *context,
    QScriptEngine *engine)
```

```
{
    QColor color = qscriptvalue_cast<QColor>(context->thisObject());
    return color.blue();
}

QScriptValue QColor_prototype_setBlue(QScriptContext *context,
    QScriptEngine *engine)
{
     QColor * color = qscriptvalue_cast<QColor *>(context->thisObject());
     if (!color)
         return context->throwError(QScriptContext::TypeError,
            "This object is not a QColor");
     color->setBlue(context->argument(0).toInt32());
     return engine->undefinedValue();
}

QScriptValue QColor_ctor(QScriptContext *context, QScriptEngine *engine)
 {
     int r = context->argument(0).toInt32();
     int g = context->argument(1).toInt32();
     int b = context->argument(2).toInt32();
     QScriptValue res = engine->toScriptValue(QColor(r, g, b, 80));
     res.setProperty("red", engine->newFunction(QColor_prototype_setRed),
        QScriptValue::PropertySetter);
     res.setProperty("red", engine->newFunction(QColor_prototype_red),
         QScriptValue::PropertyGetter);
     res.setProperty("green",
        engine->newFunction(QColor_prototype_setGreen),
        QScriptValue::PropertySetter);
     res.setProperty("green",
        engine->newFunction(QColor_prototype_green),
        QScriptValue::PropertyGetter);
     res.setProperty("blue",
        engine->newFunction(QColor_prototype_setBlue),
        QScriptValue::PropertySetter);
     res.setProperty("blue",
        engine->newFunction(QColor_prototype_blue),
        QScriptValue::PropertyGetter);
     return res;
 }
```

Как нетрудно догадаться, функции `QColor_prototype_*` представляют собой сеттеры и геттеры для свойств `QColor`. Как и в случае с объектами Qt, геттеры и сеттеры объектов языка сценариев являются методами соответствующего объекта, только получить к ним доступ напрямую нельзя. При вызове функции-метода в сценарии, соответствующем прототипу функции на языке C++, становится доступен объект `thisObject` (получить его можно с помощью метода `thisObject()` объекта класса `QScriptContext`). Этот дополнительный параметр аналогичен параметру `this` мето-

дов классов C++. Поскольку переданный функциям объект `thisObject` представляет собой "обертку" для объекта `QColor`, мы можем извлечь этот объект (или указатель на него) из "обертки" с помощью функтора `qscriptvalue_cast`. Обратите внимание на то, что в функциях-геттерах мы извлекаем объект `QColor` по значению, т. е. получаем копию того объекта, который "завернут" в `thisObject`. В функциях-сеттерах, где нам необходимо изменить тот самый объект, который содержится в `thisObject`, мы получаем указатель на `QColor`. Ничто не мешает нам получать указатель на объект и в функциях-геттерах. Это полезно, например, в том случае, когда создание копии объекта сопряжено с большими затратами памяти.

В функции-конструкторе `QColor_ctor()` мы назначаем свойства объекту `res` (который является "представителем" объекта `Qt QColor` в сценарии) точно так же, как мы назначали свойства глобальному объекту в примерах, рассмотренных ранее. Разница в том, что при создании нового объекта-функции в методе `setProperty()` мы указываем флаг `QScriptValue::PropertySetter` или `QScriptValue::PropertyGetter`. В результате, например, вызов `res.setProperty("red", ...)` создает вместо метода с именем `red()` свойство `red`, с которым связаны две функции: сеттер и геттер. Саму функцию `QColor_ctor()` мы, как и в приведенном ранее примере, делаем методом глобального объекта. В результате у переменной, содержащей значения `QColor`, появляются свойства `red`, `green` и `blue`, к которым можно обращаться в тексте сценария (листинг 12.16).

Листинг 12.16. Обращение к свойствам переменной со значением типа `QColor`

```
var color = new QColor(255,255,255);
color.red = 0;
print(color.blue);
Ellipse1.setColor(color);
```

Между прочим, описанный выше способ добавления свойств и методов в функции-конструкторе позволяет добавить объекту сценария `QColor` такие методы и свойства, которые не предусмотрены в его прототипе (листинг 12.17).

Листинг 12.17. Обращение к свойствам переменной со значением типа `QColor`

```
QScriptValue QColor_toString(QScriptContext *context,
    QScriptEngine *engine)
{
    QColor color = qscriptvalue_cast<QColor>(context->thisObject());
    return QString("{%1, %2, %3}")
        .arg(color.red()).arg(color.green()).arg(color.blue());
}

QScriptValue QColor_ctor(QScriptContext *context, QScriptEngine *engine)
 {
     int r = context->argument(0).toInt32();
```

```
    int g = context->argument(1).toInt32();
    int b = context->argument(2).toInt32();
    QScriptValue res = engine->toScriptValue(QColor(r, g, b, 80));
    ...
    res.setProperty("toString", engine->newFunction(QColor_toString));
    return res;
}
```

В результате у объекта `QColor` в сценарии появится метод `toString()`, преобразующий значение объекта в строку. Теперь в сценарии можно написать:

```
var color = new QColor(0, 255, 255);
print(color);
```

Дело в том, что функция `print()` ищет у переданного ей аргумента метод `toString()` и выводит то, что возвращает этот метод. Несмотря на то, что у типа `QColor` в языке сценариев появился новый метод, он по-прежнему приводим к типу `QColor` и может использоваться там, где требуется этот тип (например, как аргумент метода `setColor()`).

Отдельного разговора заслуживает вопрос о взаимодействии объекта `QColor` и сборщика мусора. У объекта `QColor` нет ресурсов, динамически созданных в конструкторе, значит, и удаление этого объекта не представляет проблем. Однако система сборки мусора не сможет вызвать деструктор класса, объявленный явным образом, если класс не является потомком `QObject`. Мы могли бы вызвать деструктор в программе Qt, но в таком случае мы должны знать, когда объект выйдет из области видимости в программе-сценарии. В этом нам поможет использование класса `QSharedPointer` в качестве указателя на объект. Соответствующая методика описана в статье, доступной по ссылке **http://doc.qt.nokia.com/4.7-snapshot/scripting.html**. Из этой статьи можно почерпнуть множество других полезных приемов при работе со сценариями.

Новшества в системе сценариев Qt 4.7

Перед выполнением программы-сценария система сценариев Qt компилирует программу, преобразуя ее в специальный промежуточный код. В обычном режиме компиляция выполняется перед каждым запуском сценария. Если сценарий выполняется только один раз, это не имеет значения, но если один и тот же сценарий выполняется множество раз, затраты времени на компиляцию могут оказаться значительными. Использование класса `QScriptProgram` позволяет сократить эти расходы. Если создать для текста программы-сценария объект `QScriptProgram`, а потом передать этот объект методу `evaluate()` объекта `QScriptEngine`, количество компиляций программы будет сведено к минимуму.

ГЛАВА 13

Динамические расширения программ Qt

Если рассматривать встроенные языки сценариев как средство расширения функциональности программы, следует признать за ними одно существенное достоинство и один, не менее существенный, недостаток. Достоинство языков сценариев заключается в том, что писать сценарии, расширяющие функциональность, легко. Для этого не требуются специальные средства разработки, достаточно текстового редактора, а сами языки, как правило, просты для освоения. Недостаток языков сценариев заключается в том, что возможности расширения функциональности, предоставляемые ими, весьма ограничены.

> ***Примечание***
>
> Я говорю об этом как о недостатке, хотя в некоторых случаях и эту особенность сценариев можно рассматривать как достоинство.

Если для расширения возможностей программы вам требуется нечто большее, чем языки сценариев, к вашим услугам система модулей расширений (plugins) программ Qt. Модули расширения программ Qt представляют собой динамические разделяемые библиотеки, которые написаны по определенным правилам.

Если вы пишете программу с закрытыми исходными текстами, то можете использовать механизм расширений для того, чтобы сторонние разработчики могли адаптировать программу для своих нужд. На первый взгляд может показаться, что для тех, кто пишет программы с открытыми исходными текстами, модули расширения не нужны, ведь необходимый код всегда можно добавить в исходный текст самой программы. Однако этот взгляд ошибочен (не в том смысле, что исходный текст открытой программы нельзя модифицировать, а в том, что модули расширения при разработке открытых программ бесполезны). Во-первых, расширения, хранящиеся в отдельных библиотеках, позволяют лучше организовать ваши исходные тексты, следуя принципу разделения различных элементов программы, о котором уже много говорилось в этой книге. А во-вторых, модули расширения сделают вашу программу более адаптивной. Представьте себе две программы: одна — огромный монолит, в который включено все, что только может понадобиться пользователю, другая — компактное приложение с набором модулей расширений, из которых каждый пользователь может выбрать те, которые нужны лично ему. Я не стану доказывать, что второй вариант лучше. Если вы не согласны с этим, можете просто пропустить данную главу.

Центральную роль в системе расширений Qt играет концепция интерфейсов. В первом приближении интерфейс — это описание набора методов класса. В этом смысле можно сказать, что у каждого класса есть интерфейс. Но концепция интерфейсов сложнее. Интерфейсом можно считать некое подмножество множества методов класса. Например, класс `QGraphicsObject` является потомком классов `QObject` и `QGraphicsItem`. Можно сказать, что объект класса-потомка `QGraphicsObject` реализует два интерфейса: интерфейс `QObject` и интерфейс `QGraphicsItem`. Объект такого класса можно привести к одному из этих типов и рассматривать его исключительно как объект соответствующего типа.

Расширения Qt основаны на объектах. С точки зрения самой библиотеки расширений каждое расширение представляет собой класс, реализующий один или несколько интерфейсов. С точки зрения приложения интерфейс представляет собой объект, который принадлежит классу-наследнику `QObject` и абстрактным классам, реализующим интерфейсы расширений.

Мы рассмотрим использование модулей расширений на примере программы, воспроизводящей звук из различных источников. Работа с источниками звука выполняется во внешних расширениях. Поскольку центральной концепцией в приложениях, использующих динамические расширения, является интерфейс, именно с его описания мы и начнем (листинг 13.1)

Листинг 13.1. Класс `AudioSourceInterface`

```
class AudioSourceInterface
{
public:
    virtual ~AudioSourceInterface() {};
    virtual void setSourceName(const QString &name) = 0;
    virtual unsigned short getChannels() = 0;
    virtual unsigned short getBPS() = 0;
    virtual unsigned int getSampleRate() = 0;
    virtual unsigned short getFrameSize() = 0;
    virtual long int getFramesCount() = 0;
    virtual int getRawData(char * data, int numFrames) = 0;
    virtual QString getLastError() const = 0;
    virtual void clearError() = 0;
    virtual void release() = 0;
};
```

Как нетрудно заметить, этот класс имеет определенную специфику. Все его методы являются виртуальными и объявлены в разделе `public`. У всех методов, кроме деструктора, отсутствует реализация (деструктор с пустым телом нужен только для того, чтобы некоторые компиляторы C++ не выдавали предупреждающих сообщений). С конструкторами и деструкторами связана одна из проблем интерфейсов. В определении интерфейса не может быть конструктора или деструктора, выполняющего какие-то осмысленные действия, поскольку одна из задач интерфейса — абстрагироваться от конкретной реализации. Это значит, что функции инициализа-

ции объекта и освобождения ресурсов, которые обычно выполняют конструкторы и деструкторы, должны взять на себя другие методы. В рассматриваемом примере эти роли играют методы `setSourceName()` и `release()`. Наш класс представляет источник аудиоданных настолько абстрактно, насколько это возможно. В результате за интерфейсом `AudioSourceInterface` может стоять практически любой источник данных: программный генератор, сетевой поток, физическое устройство или файл. Предполагается, что детали работы с конкретным источником реализованы в модуле расширения. Все что видит программа — это единообразный интерфейс.

Функция `setSourceName()` предназначена для того, чтобы задать имя конкретного ресурса аудиоданных. В зависимости от типа источника это может быть имя файла, устройства, сетевой адрес или какой-либо объект, "понимающий" конкретный источник данных. Эта функция инициализирует источник данных, и она должна вызываться прежде любых других функций интерфейса. Функция `getChannels()` возвращает количество каналов (1 — моно, 2 — стерео и т. д.). Функция `getBPS()` возвращает значение разрядности сэмплов в битах (возможные значения — 8, 16, 24, 32). Хотя некоторые источники аудиоданных могут возвращать сэмплы в виде чисел с плавающей точкой, наш интерфейс предполагает, что сэмплы — целые числа. Если источник данных, с которым работает реализация интерфейса, возвращает данные в формате с плавающей точкой, модуль расширения должен конвертировать их в один из целочисленных форматов. Функция `getSampleRate()` возвращает частоту дискретизации аудиосигнала в герцах. С помощью функции `getFrameSize()` можно узнать размер одного аудиофрейма (по сути, это произведение количества байт, занимаемых одним сэмплом, на количество каналов) в байтах. Например, если разрядность сэмплов сигнала составляет 16 бит, а количество каналов — 2, то размер фрейма составит 4 байта. Если объем передаваемых источником аудиоданных известен заранее, то функция `getFramesCount()` возвращает общее количество фреймов. В противном случае эта функция возвращает значение –1.

Функция `getRawData()` позволяет получить очередную порцию аудиоданных. Эта функция является блокирующей. Поскольку данные лучше всего считывать целыми фреймами, функции передается количество фреймов (а не количество байтов), которое мы хотим прочитать. Первый аргумент функции — указатель на буфер, в который будут записаны данные. Буфер должен иметь размер, достаточный для того, чтобы записать в него затребованное количество фреймов, хотя функция может записать и меньше фреймов, чем затребовано. Функция возвращает количество записанных в буфер байтов (а не фреймов). Если это число больше нуля, значит, было прочитано целое количество фреймов. Если `getRawData()` возвращает значение 0, его не следует рассматривать как признак окончания передачи данных. Когда передача данных окончена, функция возвращает значение –1 и продолжает возвращать это значение при всех последующих вызовах вплоть до следующего вызова `setSourceName()`.

Функция `getLastError()` возвращает текстовое описание последней ошибки, возникшей при работе с источником данных. Если ошибок не было, функция возвращает пустую строку. Сбросить значение последней ошибки можно с помощью функции `clearError()`.

Функция `release()` освобождает ресурсы, используемые источником данных. Эта функция предназначена для досрочного завершения работы с источником аудиоданных. Если функция `getRawData()` вернула значение –1, вызывать функцию `release()` не требуется.

> ***Примечание***
>
> В качестве примера мы рассмотрим только один модуль расширений, реализующий интерфейс `AudioSourceInterface`. Разумеется, режим работы этого модуля соответствует описанию интерфейса, приведенному ранее, но не реализует все аспекты поведения этого интерфейса.

Подробное описание интерфейса `AudioSourceInterface` приводится для того, чтобы продемонстрировать, насколько важно при проектировании интерфейса предусмотреть (и четко описать) все возможные ситуации, которые могут возникнуть при его работе. Интерфейс является единственным связующим звеном между разработчиком основной программы и разработчиком расширений, и его описание должно быть составлено так, чтобы исключить недопонимание с обеих сторон.

Поскольку у класса `AudioSourceInterface` отсутствует реализация, все, что связано с этим классом, удобно разместить в одном заголовочном файле (в нашем примере это файл audiosourceinterface.h). Данный заголовочный файл будет использоваться как в проекте программы, так и в проекте модуля расширения. Помимо объявления класса в заголовочный файл следует добавить строку:

```
Q_DECLARE_INTERFACE(AudioSourceInterface,
"net.symmetrica.qt47book.ch13.AudioSourceInterface/1.0")
```

Макрос `Q_DECLARE_INTERFACE()` объявляет о том, что класс, имя которого передано ему в качестве первого параметра, является интерфейсом. Помимо прочего, этот макрос придает интерфейсу уникальное имя. Само имя класса, описывающего интерфейс, не может служить для этой цели хотя бы потому, что один интерфейс может существовать в разных версиях. Для создания уникального имени интерфейса используется та же система, что и при именовании пакетов Java: доменное имя с элементами в обратном порядке, имя интерфейса и номер версии. Теперь наш интерфейс готов к использованию, как в модулях расширения, так и в программе, которая будет загружать эти модули.

Мы начнем разработку демонстрационных проектов с модуля расширения, т. к. без него демонстрационная программа работать просто не сможет. Модуль расширения, который будет рассмотрен далее, использует в качестве источника аудиоданных файлы с расширением wav. Мы назовем этот модуль waveplugin. Как уже отмечалось, модуль расширения является разделяемой библиотекой. Файл проекта Qt должен выглядеть соответствующим образом (листинг 13.2).

Листинг 13.2. Файл waveplugin.pro

```
QT -= gui
TARGET = waveplugin
TEMPLATE = lib
```

```
CONFIG += plugin
DEFINES += WAVEPLUGIN_LIBRARY
SOURCES += waveplugin.cpp
HEADERS += waveplugin.h\
        waveplugin_global.h \
    audiosourceinterface.h
```

Мы удаляем из проекта модуль gui. Хотя модули расширения могут содержать элементы графического интерфейса, нашему модулю они не требуются. Значение `lib`, присвоенное переменной `TEMPLATE`, указывает на то, что мы создаем библиотеку, а не файл образа программы. Значение `plugin`, добавленное к значениям переменной `CONFIG`, указывает не то, что создаваемая библиотека будет модулем расширения программы Qt. К значениям переменной `DEFINES` (эта переменная устанавливает директивы препроцессора C++) мы добавляем значение `WAVEPLUGIN_LIBRARY`, которое будет управлять директивами экспорта и импорта символов из библиотеки.

По листингу 13.2 видно, что модуль расширения не так уж и сложен. Он состоит из одного файла исходных текстов и трех заголовочных файлов, один из которых мы подробно рассмотрели ранее. Файл waveplugin_global.h был сгенерирован автоматически, он содержит директивы препроцессора, управляющие экспортом символов (листинг 13.3).

Листинг 13.3. Файл waveplugin_global.h

```
#ifndef WAVEPLUGIN_GLOBAL_H
#define WAVEPLUGIN_GLOBAL_H

#include <QtCore/qglobal.h>

#if defined(WAVEPLUGIN_LIBRARY)
#  define WAVEPLUGINSHARED_EXPORT Q_DECL_EXPORT
#else
#  define WAVEPLUGINSHARED_EXPORT Q_DECL_IMPORT
#endif
```

Макрос `Q_DECL_EXPORT` указывает Qt, что соответствующий символ должен быть сделан экспортируемым. Макрос `Q_DECL_IMPORT` указывает на то, что соответствующий символ должен быть импортирован из внешней библиотеки (предполагается, что библиотека и программа используют один и тот же заголовочный файл, но с разными директивами). В случае модуля расширения Qt используются другие заголовочные файлы и все эти директивы в принципе не нужны.

Самое интересное, т. е. реализация интерфейса `AudioSourceInterface`, сосредоточено в файлах waveplugin.h и waveplugin.cpp. Прежде всего мы должны объявить класс, реализующий интерфейс (листинг 13.4).

Листинг 12.4. Класс `WavePlugin`

```
class WAVEPLUGINSHARED_EXPORT WavePlugin : public QObject,
    public AudioSourceInterface
{
    Q_OBJECT
    Q_INTERFACES(AudioSourceInterface)
public:
    void setSourceName(const QString &name);
    unsigned short getChannels();
    unsigned short getBPS();
    unsigned int getSampleRate();
    unsigned short getFrameSize();
    long int getFramesCount();
    int getRawData(char * data, int numFrames);
    QString getLastError() const;
    void clearError();
    void release();
private:
    QString lastError;
    QFile * device;
    unsigned int m_size;
    unsigned short bps;
    unsigned int sr;
    unsigned short channels;
};
```

Класс `WavePlugin` наследует одновременно классам `QObject` и `AudioSourceInterface`. Поскольку класс, реализующий интерфейс, является потоком `QObject`, ему доступны возможности, предоставляемые классом `QObject`, например, сигналы и слоты. Но тут есть одна тонкость. Определения сигналов и слотов не должны быть частью интерфейса. Интерфейс — это набор обычных виртуальных методов C++ и ничего более. Если вы хотите, чтобы класс, реализующий интерфейс, эмитировал сигналы, вы должны объявить эти сигналы в объявлении класса (в нашем примере это должен быть класс `WavePlugin`), но не в определении интерфейса. Программа, загрузившая модуль расширения и создавшая объект класса расширения, сможет связать свои слоты с сигналами, которые эмитирует этот класс.

Примечание

Возможно, у вас возникает вопрос, откуда же программа узнает о сигналах, эмитируемых классом, если они не определены в интерфейсе. Следует понимать, что интерфейс играет роль на этапе компиляции программы, тогда как связывание сигналов и слотов происходит во время выполнения, с использованием информации, предоставленной метаобъектом соответствующего объекта. На этапе компиляции программа не проверяет, действительно ли переданный ей объект обладает определенными сигналами или слотами. Именно поэтому мы можем передать функции `connect()` указа-

тель на объект любого класса-потомка QObject, как указатель на QObject, и при этом выполнить связывание с любыми сигналами и слотами, определенными в потомке. Другое дело, что если интерфейс класса является единственным источником информации о возможностях класса, о передаваемых классом сигналах ничего не узнает программист. Но задача информирования программистов лежит за пределами задач библиотеки Qt.

Наш класс расширения не использует сигналы и слоты, и перечень функций в его разделе public полностью совпадает с перечнем функций интерфейса. Помимо традиционного макроса Q_OBJECT(), в объявлении класса WavePlugin мы используем еще один макрос — Q_INTERFACES(). Этот макрос необходим системе, чтобы сообщить, что создаваемый объект класса WavePlugin может быть приведен к типу AudioSourceInterface. У макроса Q_INTERFACES() может быть несколько аргументов, т. е. один и тот же класс может экспортировать несколько интерфейсов. Более того, один и тот же объект, созданный на основе такого класса, может использоваться как носитель нескольких совершенно разных интерфейсов. Главное, чтобы они не мешали друг другу.

В реализации методов класса WavePlugin нет ничего необычного (листинг 13.5).

Листинг 13.5. Реализация методов класса WavePlugin

```
void WavePlugin::setSourceName(const QString &name)
{
    device = new QFile(0);
    if (device->isOpen())
        device->close();
    device->setFileName(name);
    if (!device->open(QIODevice::ReadOnly)) {
        lastError = QString::fromUtf8("Невозможно открыть источник");
        return;
    }
    QString str;
    QByteArray ba;
    ba = device->read(4);
    if (QString::fromLatin1(ba.data(), 4) != "RIFF") {
        device->close();
        lastError = QString::fromUtf8("Неизвестный формат источника");
        return;
    }
    device->read(4); // Пропускаем полный размер файла
    ba = device->read(4);
    if (QString::fromLatin1(ba.data(), 4) != "WAVE") {
        device->close();
        lastError = QString::fromUtf8("Неизвестный формат источника");
        return;
    }
    device->read(4); // Пропускаем тег fmt
    device->read(4); // и его размер,
```

```
    device->read(2); // и его формат (предполагаем WAVE_FORMAT_PCM).
    device->read((char *)&channels, 2);
    device->read((char *)&sr, 4);
    device->read(4); // Пропускаем avg_bytes_sec
    device->read(2); // и nblock_align.
    device->read((char *)&bps, 2);
    device->read(4); //Пропускаем тег data.
    device->read((char *)&m_size, 4);
    device->seek(44);
}

void WavePlugin::release()
{
    delete device;
    device = 0;
}

QString WavePlugin::getLastError() const
{
     return lastError;
}

void WavePlugin::clearError()
{
    lastError = "";
}

unsigned short WavePlugin::getChannels()
{
    return channels;
}

unsigned short WavePlugin::getBPS()
{
    return bps;
}

unsigned int WavePlugin::getSampleRate()
{
    return sr;
}

unsigned short WavePlugin::getFrameSize()
{
    return channels*bps/8;
}
```

```
long int WavePlugin::getFramesCount()
{
    return m_size/getFrameSize();
}

int WavePlugin::getRawData(char * data, int numFrames)
{
    if (!device) return -1;
    if (device->pos() >= m_size + 44) {
        release();
        return -1; // EOF
    }
    int numBytes = numFrames*getFrameSize();
    if (device->pos() + numBytes > m_size + 44)
        numBytes = m_size + 44 — device->pos();
    return device->read(data, numBytes);
}

Q_EXPORT_PLUGIN2(waveplugin, WavePlugin)
```

Поскольку класс `WavePlugin` использует в качестве источника данных звуковые файлы, методу `setSourceName()` передается имя файла, с которым класс работает обычным для Qt образом — с помощью объекта `QFile`. Первая задача, которую должен решить метод `setSourceName()` — прочитать информацию об аудиопотоке из заголовка WAV-файла. Вообще-то файлы с расширением wav могут содержать данные в самых разных форматах, даже в формате MP3. Для упрощения примера мы упрощаем себе задачу. Наш модуль расширения может воспроизводить только файлы, которые хранят данные в формате `WAVE_FORMAT_PCM`, и в заголовке которых нет других разделов, кроме раздела `fmt`. Вам не придется разбираться в этом. Вместе с исходными текстами примера вы найдете файл affirmative.wav, который соответствует всем требованиям. Поскольку формат `WAVE_FORMAT_PCM` весьма распространен, многие другие файлы, загруженные из Интернета, тоже можно будет воспроизвести.

Обратите внимание на макрос `Q_EXPORT_PLUGIN2()`. Этот макрос — последнее, что мы должны вызвать, для того чтобы оформить класс `WavePlugin` как расширение программы Qt. Первый аргумент макроса — имя модуля, он совпадает со значением переменной `TARGET` в файле проекта Qt. Второй аргумент — имя класса.

Теперь мы можем скомпилировать модуль расширения. В результате у нас появится файл разделяемой библиотеки — waveplugin.dll для Windows и libwaveplugin.so для Linux. Можно переходить к разработке приложения, использующего созданный нами модуль расширений.

Программа plugindemo загружает модуль расширений waveplugin и воспроизводит выбранный файл. В Qt существуют две возможности загрузки модулей расширений, которые соответствуют двум способам загрузки разделяемых библиотек: статическому и динамическому. При статическом способе приложение связывается

с модулем еще на этапе сборки. Программа не запустится, если соответствующий модуль не будет найден. При динамическом способе загрузки имя модуля выбирается во время работы программы. Его может выбрать пользователь, с помощью файла конфигурации или в специальном диалоговом окне, как это происходит в программе plugindemo. Статическая загрузка лучше подходит для обычных разделяемых библиотек, без которых программа все равно не сможет работать. В случае с модулями расширения гораздо интереснее динамическая загрузка. Ее мы и рассмотрим (листинг 13.6).

Листинг 13.6. Загрузка модуля расширений

```
void MainWindow::on_pushButton_2_clicked()
{
    QString pluginName = QFileDialog::getOpenFileName(this,
        trUtf8("Зарузить библиотеку расширений"), "",
        trUtf8("Библиотеки (*.dll *.so)"));
    if (pluginName != "") {
        QPluginLoader loader(pluginName);
        audioSource = loader.instance();
        if (!audioSource) {
            QMessageBox::critical(this, QString::fromUtf8("Ошибка"),
               QString::fromUtf8("Невозможно загрузить расширение."));
            return;
        }
        audioSourceInterface =
            qobject_cast<AudioSourceInterface *>(audioSource);
        if (!audioSourceInterface)
            QMessageBox::critical(this, QString::fromUtf8("Ошибка"),
               QString::fromUtf8("Невозможно создать интерфейс."));
    }
}
```

Загрузкой модулей расширений управляет объект класса `QPluginLoader`. Если программе требуются несколько модулей расширения, неважно, одного типа или разных, для их динамической загрузки необходимо использовать несколько объектов `QPluginLoader`, так чтобы каждый модуль загружался своим объектом. Имя загружаемого модуля можно передать объекту в параметре конструктора, что мы и делаем. Метод `instance()` создает и возвращает объект класса, экспортируемого модулем расширения, в виде указателя на объект `QObject`. Если по каким-либо причинам создать объект не удалось, метод возвращает значение 0. Метод `isLoaded()` позволяет проверить, удалось ли загрузить выбранный модуль расширения. Метод `errorString()` возвращает текстовое описание последней ошибки, возникшей в процессе загрузки модуля. Динамически загруженный модуль можно динамически выгрузить с помощью метода `unload()`.

В соответствии со сказанным выше переменная `audioSource` имеет тип `QObject *`. Для того чтобы работать с расширением, нам, естественно, необходимо привести

тип объекта к `AudioSourceInterface`. Это делается с помощью уже знакомой нам функции `qobject_cast()`.

После того как мы убедились, что интерфейс нами получен, мы можем приступить к воспроизведению звука. Для этой цели мы используем объект класса `QAudioOutput`. Класс `QAudioOutput` представляет собой простое и удобное средство для кросс-платформенного воспроизведения "сырых" аудиоданных (т. е. аудиоданных, не хранящихся в файле в каком-либо формате).

> ***Примечание***
>
> Форматы хранения аудиоданных предусматривают хранение, помимо самих данных, таких параметров аудиосигнала, как частота дискретизации, разрядность отсчетов, количество каналов, длительность записи и т. п. Например, WAV-файлы хранят эту информацию в заголовочной части (откуда мы ее и извлекаем). Поскольку класс `QAudioOutput` предназначен для работы с "сырыми" данными (и этим он отличается от других классов Qt, работающих с мультимедийными данными), все эти параметры надо передавать ему отдельно.

Для того чтобы программа могла использовать класс `QAudioOutput`, в файл проекта Qt надо добавить строку:

```
QT += multimedia
```

Для настройки объекта `QAudioOutput` на воспроизведение мы пользуемся методами интерфейса `AudioSourceInterface` (листинг 13.7).

Листинг 13.7. Настройка объекта `QAudioOutput`

```
if (!audioSourceInterface) {
        QMessageBox::critical(this, QString::fromUtf8("Ошибка"),
            QString::fromUtf8("Расширение не загружено."));
        return;
    }
    audioSourceInterface->clearError();
    QString fileName = QFileDialog::getOpenFileName(this,
         trUtf8("Открфть файл"), "", trUtf8("Файлы WAV (*.wav)"));
    if (fileName.isEmpty())
        return;
    audioSourceInterface->setSourceName(fileName);
    if (!audioSourceInterface->getLastError().isEmpty()) {
        QMessageBox::critical(this, QString::fromUtf8("Ошибка"),
            audioSourceInterface->getLastError());
        return;
    }
    ui->pushButton->setEnabled(false);
    QAudioFormat format;
    format.setFrequency(audioSourceInterface->getSampleRate());
    format.setChannels(audioSourceInterface->getChannels());
    format.setSampleSize(audioSourceInterface->getBPS());
    format.setCodec("audio/pcm");
```

```
format.setByteOrder(QAudioFormat::LittleEndian);
format.setSampleType(QAudioFormat::UnSignedInt);
QAudioDeviceInfo info(QAudioDeviceInfo::defaultOutputDevice());
if (!info.isFormatSupported(format)) {
    qWarning() << QString::fromUtf8("Формат не поддерживается.");
    return;
}
audio = new QAudioOutput(format, this);
audio->setBufferSize(audioSourceInterface->getFrameSize()*frames);
connect(audio, SIGNAL(stateChanged(QAudio::State)), this,
    SLOT(finished(QAudio::State)));
audioOut = audio->start();
```

Параметры аудиопотока объединены классом `QAudioFormat`. Мы заполняем объект `format` этого класса, используя соответствующие методы интерфейса `AudioSourceInterface`. После того как объект `format` заполнен данными, мы проверяем, поддерживает ли устройство аудиовывода, заданное в системе по умолчанию, данную конфигурацию. Это делается с помощью метода `isFormatSupported()` класса `QAudioDeviceInfo`.

> ***Примечание***
>
> Использовать устройство, заданное по умолчанию, вовсе не обязательно. Классы, которыми мы пользуемся в этом примере, могут работать с любым аудиоустройством, доступным системе.

Если заданная нами конфигурация поддерживается, мы создаем объект класса `QAudioOutput`, настраиваем размер аудиобуфера с помощью метода `setBufferSize()` и запускаем устройство с помощью метода `start()`. Будучи вызван без параметров, этот метод возвращает указатель на объект класса `QIODevice`, в который можно записывать аудиофреймы точно так же, как при работе с файлом.

Применение абстрактных классов в качестве экспортируемых интерфейсов, создание уникальных имен интерфейсов, возможность приведения типа объекта к одному из нескольких интерфейсов не являются новинками в мире программирования. Многие прикладные интерфейсы Windows опираются на похожие принципы. Вы можете реализовать похожую систему модулей расширения для своего приложения, вообще не прибегая к помощи механизмов Qt. Возникает вопрос: что же дают нам эти механизмы? Во-первых, механизмы Qt позволяют использовать в наших интерфейсах специфические типы данных Qt, в том числе виджеты. Во-вторых, система модулей расширения Qt будет работать одинаково на всех платформах, которые поддерживает Qt. Хотя методы динамической загрузки разделяемых библиотек в Windows, Linux и, например, AIX, довольно сильно различаются, при работе с динамическими модулями Qt мы можем не думать об этих различиях. Система берет на себя заботу о реализации загрузки на каждой платформе. Наконец, система модулей расширений Qt весьма широко распространена. Большинство разработчиков, пишущих программы с использованием Qt, не изобретает велосипед, а берет готовую систему модулей расширений, которую предоставляет Qt. Это зна-

чит, что если вы захотите создать модуль расширения для программы, написанной на Qt вашим коллегой, вам, скорее всего, придется использовать эту систему. И если вы хотите, чтобы другие программисты писали модули расширений для вашей программы, вам также лучше использовать стандартные средства расширений Qt, потому что другие разработчики, скорее всего, знакомы с этими средствами.

Класс *QLibrary*

В конце этой главы речь пойдет об одном классе, который весьма тесно, хотя и не очень заметно, связан с системой модулей расширений. Это класс `QLibrary`. Когда мы рассматривали программу, проверяющую орфографию, мы, помимо прочего, познакомились с системой статического связывания программ Qt и разделяемых библиотек. Класс `QLibrary` помогает приложению выполнять динамическую загрузку разделяемых библиотек (которые вовсе необязательно являются модулями расширений). О различиях между статической и динамической загрузкой мы уже говорили.

У каждой операционной системы, которой знакома концепция разделяемых библиотек, есть средства их динамической загрузки. Главное преимущество класса `QLibrary` заключается в кросс-платформенности. Если ваша программа загружает библиотеки с помощью объекта класса `QLibrary`, вам, скорее всего, не придется менять ни одной строчки кода при переносе программы с одной платформы на другую.

Например, при работе с библиотеками с помощью объекта `QLibrary` необязательно указывать расширение имени файла, в котором хранится библиотека. Это означает, что один и тот же код в программе может динамически загружать библиотеки при работе под управлением разных операционных систем (в которых файлы библиотек имеют разные расширения). Этот механизм будет работать, разумеется, только при условии, что базовые части имен библиотеки в разных системах совпадают.

В случае, если файл библиотеки задан только своим именем, без указания пути, класс `QLibrary` постарается найти этот файл, последовательно перебирая несколько наиболее вероятных директорий. Например, если программа собирается под Linux, где разделяемые библиотеки обычно хранятся в специальной директории, объект `QLibrary` сначала выполнит поиск в директориях, перечисленных в переменной `LD_LIBRARY_PATH`, потом в рабочей директории программы и т. д.

Обычно динамическая загрузка библиотеки выполняется для того, чтобы можно было получить адреса каких-либо объектов библиотеки, к которым программа обращается по именам. Каждая платформа предоставляет свой собственный интерфейс для решения этой задачи: функции `LoadLibrary()`, `FreeLibrary()`, `GetProcAddress()` на платформе Windows, функции `dlopen()`, `dlclose()`, `dlsym()` — на платформе Linux и т. д. Если вы используете класс `QLibrary`, вам не нужно беспокоиться об этих деталях. Метод `resolve()` позволит получить адрес экспортируемого объекта на любой поддерживаемой платформе.

ГЛАВА 14

Консольные приложения Qt

Разумеется, чаще всего библиотека Qt используется для создания приложений с графическим интерфейсом. Однако с помощью Qt можно создавать и консольные программы. Есть ли смысл использовать Qt для создания консольных программ? Я уверен, что есть. Как вы уже могли убедиться, Qt — это не просто набор виджетов. Помимо компонентов графического интерфейса Qt предоставляет множество удобных классов для работы со строками, файлами, потоками, XML-документами, базами данных, сетевыми протоколами, разделяемыми библиотеками и многими другими компонентами, которые могут понадобиться и в консольных программах. Если у вас уже есть опыт работы с Qt, то почему бы не использовать этот опыт для написания кросс-платформенного консольного приложения?

> ***Примечание***
>
> Программы, которые я называю консольными, отличаются от графических программ тем, что выводят свои данные в текстовом виде — в окне командной строки Windows или в окне терминала Linux. Программы этого типа можно условно разделить на две категории. Одни программы работают в интерактивном режиме. Примерами этой категории программ могут служить файловый менеджер Far или Midnight Commander. Программы второй категории используют интерфейс командной строки. Примером может служить утилита ping. Программы этой категории получают все команды пользователя в виде ключей командной строки и больше не ждут от пользователя никакого ввода (кроме, возможно, команды досрочного завершения). Существуют также программы, которые комбинируют оба подхода.

Обработка событий в консольной программе

Консольная программа Qt может обрабатывать события точно так же, как и программа с графическим интерфейсом (правда, это делается с помощью другого класса). Главное отличие заключается в том, что консольная программа получает события гораздо реже, чем графическая, или даже не получает вообще. Рассмотрим простейшую консольную программу Qt (листинг 14.1).

Листинг 14.1. Простейшая консольная программа Qt

```
#include <QtCore/QCoreApplication>

int main(int argc, char *argv[])
{
      QCoreApplication a(argc, argv);
      return a.exec();
}
```

Внешне эта программа похожа на функцию `main()` графического приложения Qt. Вместо класса `QApplication` в качестве главного класса приложения используется `QCoreApplication`, конструктору которого передаются значения параметров функции `main()`. Далее мы запускаем цикл обработки событий с помощью метода `exec()`. Однако в таком виде программа не сможет сделать ничего полезного. И тут мы сталкиваемся с главным концептуальным отличием консольных программ Qt от графических. В графической программе мы создали бы главный виджет, который получал бы события из цикла обработки, и либо обрабатывал их сам, либо передавал бы дочерним виджетам. В результате, с помощью слотов и перекрытых функций обработчиков событий наше графическое приложение получало бы информацию, которую можно было бы обработать. Но для консольной программы такого виджета у нас нет.

Помимо прочего, для того чтобы приложение стало по-настоящему консольным, из него необходимо удалить модуль gui. Для этого в файл проекта Qt добавим такие строки:

```
QT -= gui
QT += core
CONFIG += console
```

Причина наших трудностей кроется в том, что Qt ориентирована на программы, управляемые событиями. Графические программы очень хорошо вписываются в эту модель. В нее также хорошо вписываются некоторые типы консольных программ, например, сетевые серверы. Но в большинстве консольных программ модель приложения, управляемого событиями, можно реализовать только путем дополнительных ухищрений.

Ввод и вывод данных на консоль

В стандартном C++ для ввода и вывода данных на консоль принято использовать потоки `cin` и `cout`. Разумеется, эти средства можно использовать и в программах Qt, но их недостаток заключается в том, что они не поддерживают работу с типами данных Qt, такими как `QString`. Можно, конечно, выполнять преобразования типов данных Qt в стандартные типы данных C++ перед обращением к потокам `cin` и `cout`, но можно поступить и иначе, а именно, создать собственные аналоги этих потоков. Это можно сделать с помощью класса `QTextStream` (листинг 14.2).

Листинг 14.2 Консольный ввод/вывод

```
#include <QtCore/QCoreApplication>
#include <QTextStream>
#include <cstdio>
#include <QString>

int main(int argc, char *argv[])
{
    ConsoleApplication a(argc, argv);
    QTextStream qtin(stdin, QIODevice::ReadOnly);
    QTextStream qtout(stdout, QIODevice::WriteOnly);
    QString line = "";
    while (line != "q") {
        qtin >> line;
        qtout << line << endl;
    }
    return a.exec();
}
```

Класс `QTextStream` является потомком класса `QIODevice` и предназначен специально для работы с текстовыми данными. Для того чтобы получить собственные потоки консольного ввода/вывода, мы создаем два объекта `QTextStream` — один для файла `stdin`, второй — для файла `stdout`.

Представленная в листинге 14.2 программа повторяет ввод пользователя до тех пор, пока пользователь не введет строку, состоящую из символа `q`. После этого программа, по идее, должна завершить свою работу, но цикл обработки событий продолжает выполняться. Он будет выполняться бесконечно, т. е. до тех пор, пока пользователь не завершит программу принудительным способом. Можно ли вообще отказаться от вызова цикла обработки событий? В принципе, можно. Но лучше этого не делать. Некоторые компоненты Qt работают, исходя из твердого предположения, что цикл существует. Отсутствие цикла обработки событий может привести к непредсказуемому поведению этих компонентов.

Так как же завершить консольную программу, у которой есть цикл обработки событий? У класса `QCoreApplication` есть статический слот `quit()`, который позволяет завершить программу. Но для того, чтобы этот слот сработал, он должен быть вызван откуда-то изнутри цикла обработки. Когда этот слот вызывается из другого слота или из перекрытого метода какого-либо виджета, все так и происходит, поскольку, в конечном счете, все слоты и методы виджетов вызываются функцией цикла обработки событий. Однако если вызвать этот слот из функции `main()`, ничего не произойдет. Разные программисты решают эту проблему по-разному. Некоторые создают специальные экземпляры классов `QThread` и `QTimer` только для того, чтобы вызывать метод `quit()`. Мы пойдем другим путем, более сложным, но и более элегантным.

У класса `QCoreApplication` есть статический метод `postEvent()`, который позволяет передать циклу обработки событие откуда угодно, в том числе и из функции `main()`. В Qt нет события, которое бы приводило к завершению работы программы (событие `QCloseEvent`, которое закрывает виджеты, не в счет). Но мы можем создать такое событие сами (листинг 14.3).

Листинг 14.3. Консольная программа с корректным завершением

```
#include <QtCore/QCoreApplication>
#include <QTextStream>
#include <cstdio>
#include <QString>
#include <QEvent>
const QEvent::Type AppExitEventType =
     static_cast<QEvent::Type>(QEvent::User + 1);

class AppExitEvent : public QEvent
{
public:
    ExitEvent() : QEvent(AppExitEventType) {}
};

class ConsoleApplication : public QCoreApplication
{
public:
    ConsoleApplication(int argc, char ** argv)
        :QCoreApplication(argc, argv) {}
protected:
    void customEvent ( QEvent * event )
    {
        if (event->type() == AppExitEventType)
            QCoreApplication::quit();
        else
            QCoreApplication::customEvent(event);
    }
};

int main(int argc, char *argv[])
{
    ConsoleApplication a(argc, argv);
    QTextStream qtin(stdin, QIODevice::ReadOnly);
    QTextStream qtout(stdout, QIODevice::WriteOnly);
    QString line = "";
    while (line != "q") {
        qtin >> line;
        qtout << line << endl;
    }
```

```
    AppExitEvent * e = new ExitEvent();
    QCoreApplication::postEvent(&a, e);
    return a.exec();
}
```

Наша задача состоит из двух частей: создать класс, соответствующий новому событию, и научить цикл обработки событий приложения его обрабатывать. Первую часть задачи решает класс `ExitEvent`. Напомню, что у базового класса всех событий `QEvent` есть метод `type()`, который возвращает значение перечислимого типа, соответствующее типу события. На основании значения, возвращенного методом `type()`, функция `event()`, получившая событие, принимает решение о том, что делать с этим событием дальше. При создании нового класса события нам достаточно, чтобы метод `type()` этого класса возвращал нестандартное значение. В Qt предусмотрено создание программистами собственных классов событий. Для типов этих событий зарезервированы значения в диапазоне, начинающемся со значения `QEvent::User` и заканчивающемся значением `QEvent::MaxUser` (всего в этом диапазоне лежит 55 535 значений, так что нам хватит). Константа `AppExitEventType` хранит значение для типа нового события `AppExitEvent`. Сам класс `AppExitEvent` отличается от базового класса `QEvent` только тем, что в конструкторе передает значение `AppExitEventType`. В принципе, можно обойтись вовсе без специального класса и вместо строки

```
AppExitEvent * e = new ExitEvent();
```

написать

```
QEvent * e = new QEvent(AppExitEventType);
```

Обратите внимание на то, что объект события нельзя создавать как локальную переменную. Напомню, что система обработки событий Qt рассматривает переданные ей объекты событий как свою собственность и уничтожает их тогда, когда сочтет нужным.

При решении второй части стоящей перед нами задачи без нового класса обойтись не удастся. Для обработки события нового типа можно воспользоваться объектом-фильтром событий, но мы поступим иначе. Каждый раз, когда цикл обработки событий приложения получает событие нестандартного типа, он вызывает виртуальную функцию `customEvent()`. Мы создаем класс-потомок `QCoreApplication`, в котором перекрываем эту функцию. Поскольку функция `customEvent()` вызывается из цикла обработки событий, вызов `QCoreApplication::quit()` из этой функции сделает то, что нам нужно, т. е. завершит программу. Теперь у нас есть способ "цивилизованного" завершения цикла обработки событий: достаточно послать событие `AppExitEvent`.

Механизм текстового ввода/вывода, который мы рассмотрели выше, традиционен для консольных программ, написанных на C++. Главная функция программы только тем и занимается, что обрабатывает пользовательский ввод. Возможно, что для ваших целей это будет неудобно. Например, вы захотите, чтобы главная функция

программы делала что-то еще, а ввод пользователя, если таковой появится, обрабатывался отдельно, например, в специальном слоте какого-либо класса. Решить эту проблему можно разными способами, но все они требуют использования дополнительного потока для чтения данных, введенных пользователем (листинг 14.4).

Листинг 14.4. Чтение данных в слоте

```
#include <QtCore/QCoreApplication>
#include <QTextStream>
#include <cstdio>
#include <QString>
#include <QEvent>
#include <QThread>
#include <QSocketNotifier>

const QEvent::Type AppExitEventType =
   static_cast<QEvent::Type>(QEvent::User + 1);

class ConsoleApplication : public QCoreApplication
{
public:
    ConsoleApplication(int argc, char ** argv)
        :QCoreApplication(argc, argv) {}
protected:
    void customEvent ( QEvent * event )
    {
        if (event->type() == AppExitEventType)
            QCoreApplication::quit();
        else
            QCoreApplication::customEvent(event);
    }
};

QTextStream qtin(stdin, QIODevice::ReadOnly);
QTextStream qtout(stdout, QIODevice::WriteOnly);

class IOThread : public QThread
{
Q_OBJECT
public:
    IOThread() : QThread(0) {}
    public slots:
        void newData(const QString &data)
        {
            qtout << data << endl;
        }
```

```
        void threadFinished()
        {
            QEvent * e = new QEvent(AppExitEventType);
            QCoreApplication::postEvent(QCoreApplication::instance(), e);
        }

signals:
        void haveData(const QString &data);
protected:
    void run()
    {
        connect(this, SIGNAL(haveData(const QString &)), this,
            SLOT(newData(const QString &)), Qt::QueuedConnection);
        connect(this, SIGNAL(finished()), this,
            SLOT(threadFinished()), Qt::QueuedConnection);
        QString line = "";
        while(line != "q") {
            qtin >> line;
            emit haveData(line);
        }
    }
};

int main(int argc, char *argv[])
{
    ConsoleApplication a(argc, argv);
    IOThread thread;
    thread.start();
    return a.exec();
}

#include "main.moc"
```

В этом варианте функция потока `IOThread::run()` фактически играет роль обработчика событий. Как только пользователь вводит очередную строку, функция эмитирует сигнал `haveData()` и передает введенную строку в качестве аргумента. В слоте, который связан с сигналом `haveData()`, мы можем сосредоточиться на обработке введенных данных. Слот вовсе не обязательно должен быть методом класса потока. Я сделал так просто для того, чтобы не создавать еще один класс. Обратите внимание на то, что поскольку при связывании сигнала и слота использовалось значение `Qt::QueuedConnection`, слот выполняется в контексте главного потока, в том же, в котором выполняется функция `main()`. Вы можете убедиться в этом, если создадите в слоте объект какого-нибудь класса-потомка `QObject`, а в качестве параметра `parent` передадите значение `this`, т. е. указатель на объект класса `IOThread`. Напомню, что объект `IOThread` существует в контексте потока функции `main()`, а значит, указывать его в качестве родителя нового объекта можно только в том случае, если

новый объект тоже создается в контексте потока функции `main()`. Как же слот может вызываться в контексте функции `main()`, если в это время функция `main()` выполняет цикл обработки событий? Напомню, что при выполнении отложенных слотов создается объект-событие, которое передается обработчику событий приложения. Иначе говоря, в этом примере цикл обработки событий занят делом, а не крутится вхолостую. Тот факт, что чтение данных и их обработка выполняются в разных потоках, означает, между прочим, что программа может делать что-то полезное в то время, когда пользователь размышляет над вводом очередной строки.

Последнее, на что я хотел бы обратить ваше внимание в этом примере, это строка исходного текста:

```
#include "main.moc"
```

Поскольку вся наша программа состоит из одного файла main.cpp и не имеет заголовочных файлов, метакомпилятор создаст файл main.moc, который необходимо явно включить в текст программы. Если бы у классов из этого примера не было сигналов и слотов, не потребовалось бы включать файл main.moc (да и сам файл не был бы создан).

Службы и демоны

Особый класс приложений, не имеющих графического интерфейса, составляют приложения, которые выполняются в фоновом режиме, вообще без регулярного взаимодействия с пользователем. Очевидно, что такие программы в принципе не могут обладать графическим интерфейсом. В Windows программы этого типа именуются службами, в UNIX-системах — демонами.

В базовый дистрибутив Qt не входят классы, предназначенные для создания служб и демонов. Однако такие классы есть в дополнительном пакете Qt Solutions, который распространяется на тех же условиях двойного лицензирования, что и сама библиотека Qt. Qt Solutions представляет собой набор классов, которые предназначены для решения самых разнообразных задач (от создания серверных приложений до разработки интерфейсов ввода данных с встроенной проверкой орфографии). Отдельные части Qt Solutions не связаны между собой, так что вы можете использовать только то, что вам нужно, и ничего лишнего.

Мы рассмотрим пакет qtservice (архив qtservice-2.6_1-opensource.zip вы можете найти по адресу **http://symmetrica.net/qt47book/**). Этот пакет включает в себя класс `QtService`, а также ряд вспомогательных классов. Все эти классы предназначены для создания служб на платформе Windows и демонов на платформе Linux. Для того чтобы понять, какую большую работу проделали разработчики пакета, следует отметить некоторые специфические особенности служб и демонов. Службы и демоны — это не просто программы, не имеющие графического интерфейса. Когда они работают в нормальном режиме, пользователь не видит их вообще. Об их присутствии свидетельствуют только специальные утилиты управления и, разумеется, выполняемые ими функции. Работа служб и демонов должна подчиняться строгим правилам. Например, службы Windows должны взаимодействовать с менеджером

служб (Service Control Manager) и корректно обрабатывать команды, которые этот менеджер им посылает. Службы и демоны должны записывать информацию о событиях, возникающих в процессе их работы, в специальные системные журналы. Существует необходимый минимум команд, который службы и демоны должны поддерживать. Несмотря на то, что службы и демоны очень похожи с точки зрения предъявляемых к ним требований и выполняют, по сути, одинаковые функции, реализации служб на платформе Windows и демонов на UNIX-платформах очень сильно отличаются. Тем не менее разработчикам пакета qtservice удалось создать кросс-платформенную систему, благодаря чему один и тот же исходный код можно использовать как для создания службы, так и для написания демона.

Помимо класса `QtService` в состав пакета входит класс `QtServiceController`, который упрощает написание приложений, управляющих службами и демонами. Следует отметить разницу в подходах к управлению службами и демонами между Windows и UNIX-системами. В Windows наиболее предпочтительным является управление службами с помощью единообразных средств Service Control Manager, и специальные утилиты управления службами могут быть полезны, скорее, во время отладки или установки служб, тогда как в UNIX специализированные утилиты управления демонами распространены довольно широко.

Задача класса `QtService` заключается в том, чтобы связать между собой три компонента: объект, реализующий базовые функции приложения (например, объект класса `QCoreApplication`), объект, реализующий "сердце" службы — ее целевую функциональность (например, Web-сервер), и объект класса `QtServiceBase`, который отвечает за реализацию тех функций приложения, благодаря которым оно становится полноценной службой или демоном.

Класс `QtServiceBase` интегрируется в этот триплет путем наследования (`QtService` — потомок `QtServiceBase`). С классом, реализующим базовые функции приложения Qt, все гораздо интереснее. Дело в том, что `QtService` — это шаблон. В качестве параметра шаблона указывается класс `QCoreApplication` или другой подобный класс. Наконец, чтобы добавить в класс `QtService` объект, реализующий функциональность службы, необходимо создать потомка этого класса (листинг 14.5).

Листинг 14.5. Класс службы сервера HTTP

```
class HttpService : public QtService<QCoreApplication>
{
public:
    HttpService(int argc, char **argv)
    : QtService<QCoreApplication>(argc, argv, "Qt HTTP Daemon")
    {
        setServiceDescription(
            "A dummy HTTP service implemented with Qt");
        setServiceFlags(QtServiceBase::CanBeSuspended);
    }
```

```
protected:
    void start()
    {
        QCoreApplication *app = application();

        quint16 port = (app->argc() > 1) ?
                QString::fromLocal8Bit(app->argv()[1]).toUShort() : 8080;
        daemon = new HttpDaemon(port, app);

        if (!daemon->isListening()) {
            logMessage(QString("Failed to bind to port %1")
                .arg(daemon->serverPort()), QtServiceBase::Error);
            app->quit();
        }
    }

    void pause()
    {
        daemon->pause();
    }

    void resume()
    {
        daemon->resume();
    }

private:
    HttpDaemon * daemon;
};
```

Здесь мы видим все три элемента службы в работе. Класс `HttpService` наследует классу `QtService`, который является потомком класса `QtServiceBase`. В качестве параметра шаблона передается класс `QCoreApplication`. Класс `HttpDaemon` реализует функциональность сервера HTTP. Метод `application()` возвращает указатель на объект класса-приложения. Поскольку в параметре шаблона был передан класс `QCoreApplication`, это будет указатель на объект `QCoreApplication`. Поскольку класс `HttpService` полностью включает в себя класс `QCoreApplication`, его можно использовать в качестве главного класса приложения (листинг 14.6).

Листинг 14.6. Класс `HttpService` как главный класс приложения

```
int main(int argc, char **argv)
{
    HttpService service(argc, argv);
    return service.exec();
}
```

Вот так выглядит главная функция приложения, реализующего простой, но отвечающий всем стандартным требованиям Web-сервер (рис. 14.1). Разумеется, для того чтобы приложение работало, класс, реализующий функциональность сервера, должен быть источником событий (в рассмотренном примере класс `HttpDaemon` является потомком класса `QTcpServer`, который генерирует события в ответ на запрос на подключение к серверному сокету).

Рис. 14.1. Приветствие от сервера, основанного на классе `QtService`

ГЛАВА 15

Язык QML

В последних версиях библиотеки Qt разработчики решили, по-видимому, закрепить принципы разделения движка и интерфейса программы "на законодательном уровне", заставив их буквально говорить на разных языках. Если основным инструментом реализации движка по-прежнему остается C++ и все библиотеки и вспомогательные средства Qt library, то для описания интерфейса приложения предложен самостоятельный язык QML (Qt Meta-Object Language, язык метаобъектов Qt)).

QML — декларативный язык, основанный на JavaScript. Он позволяет описывать, что именно должен видеть пользователь на экране, в частности внешний вид и расположение элементов управления. Как вам нравится, например, идея заменить стандартный суховатый QDial на такой фотореалистичный "спидометр" (рис. 15.1)?

Рис. 15.1. Графический элемент управления, созданный с помощью QML

Помимо декларативных элементов язык включает в себя полный набор императивных элементов, близкий к JavaScript. Так что, в принципе, с помощью QML можно не только описывать графические интерфейсы, но и создавать программы, а Qt использовать лишь в качестве интерпретатора. Собственно говоря, в состав новейших дистрибутивов Qt входит такой интерпретатор — утилита qmlviewer, которая, помимо прочего, может казаться весьма полезной при отладке модулей QML. Вот простейшая программа — "Hello World" мира QML (листинг 15.1). Этот пример взят из официальной документации Qt.

Листинг 15.1. Простейшая программа на QML

```
import QtQuick 1.0
 Rectangle {
 id: page
 width: 500; height: 200
 color: "lightgray"
 Text {
 id: helloText
 text: "Hello world!"
 y: 30
 anchors.horizontalCenter: page.horizontalCenter
 font.pointSize: 24; font.bold: true
}
```

Прежде чем приступить к обсуждению примера, следует упомянуть об изменениях в синтаксисе. При переходе от Qt 4.7 к Qt 4.7.1 синтаксис QML изменился (и, по-видимому продолжит незначительно меняться в следующих версиях). Например, в версии 4.7 в начале программы вместо `import QtQuick 1.0` мы должны писать `import Qt 4.7`. Поскольку я верю, что изменения, произошедшие при переходе от Qt 4.7 к Qt 4.7.1, отражают определенную тенденцию, я буду пользоваться синтаксисом Qt 4.7.1. А вот в самой документации по Qt все еще можно встретить примеры, использующие старый синтаксис. Будьте внимательны!

Итак, первая строка программы QML указывает, где хранятся определения структур, применяемых в программе. Если вы пользуетесь Qt 4.7, эта строка должна выглядеть иначе, о чем было уже сказано. Затем следует корневой объект нашей программы. Как и в HTML, и в XML элементы QML образуют иерархические структуры (листинг 15.2).

Листинг 15.2. Схема иерархической структуры программы QML

```
ОБЪЕКТ {
ОБЪЕКТ {
...
}
...
}
```

В рассмотренном примере объект `Rectangle` (корень иерархии) включает в себя дочерний объект `Text`, предназначенный для вывода текста. Но это не все. Помимо дочерних объектов у элементов языка QML (так же, как у тегов HTML) имеются параметры (иногда называемые свойствами). Из листинга 15.1 видно, что для элемента `Rectangle` задаются такие параметры как ширина, высота и цвет. Кроме того, у объекта `Rectangle` есть параметр `id`, который позволяет программе QML отличать один объект `Rectangle` от другого. Этот параметр не является обязательным, и его

можно опустить, если другие части программы не ссылаются на данный конкретный объект, однако в нашем случае он необходим. Рассмотрим строку:

```
anchors.horizontalCenter: page.horizontalCenter
```

Эта строка устанавливает горизонтальное расположение элемента `Text` внутри прямоугольника. В общем, нетрудно догадаться, что мы хотим, чтобы центр строки текста по длине совпадал с центром прямоугольника. Для ссылки на прямоугольник нам требуется его индивидуальный идентификатор, заданный в параметре `id`.

На язык QML можно взглянуть и иначе, а именно, как на язык сценариев для `QGraphicsView`. Действительно, на стороне программы Qt важную роль в работе QML играет класс `QDeclarativeView`, который является потомком класса `QGraphicsView`. Таким образом, мы имеем развитие технологии Graphics View Framework, о которой мы уже знаем, что одно из ее предназначений — построение сложных графических интерфейсов. QML можно использовать и для написания сценариев для объектов `QGraphicsScene`.

А можно ли выполнить простейшую программу QML из нашей собственной программы, написанной в Qt? Разумеется! Ведь программа qmlviewer, которая считается "официальным" интерпретатором QML, написана на Qt, и все, что может сделать она, может сделать и любое другое приложение Qt. Вообще надо отметить, что на данном этапе разработчики языка QML сосредоточились, в основном, на описании самого языка и в Интернете уже предостаточно ресурсов на эту тему. Вопросы взаимодействия QML и Qt освещаются не так подробно (скорее всего, у разработчиков пока не дошли руки до этого). Но мы попробуем немного исправить положение дел, ведь главный документ — исходные тексты инструментов Qt, предназначенных для работы с QML, у нас есть.

Допустим, у нас есть окно `Dialog`, которое является потомком класса `QDialog`. Добавить в него виджет, способный отобразить вывод представленной ранее программы QML, очень просто (листинг 15.3).

Листинг 15.3. Подготовка диалогового окна к отображению вывода программы QML

```
Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
    ui->setupUi(this);
    QDeclarativeView *qmlView = new QDeclarativeView;
     qmlView->setSource(QUrl("qrc:/helloworld.qml"));
     QVBoxLayout *layout = new QVBoxLayout(this);
     layout->addWidget(qmlView);
     qmlView->show();
}
```

Как уже было сказано, простейшее средство отображения QML в Qt — класс `QDeclarativeView`, экземпляр которого мы и создаем в первую очередь (объект

`qmlView`). Затем нужно загрузить файл программы QML в созданный объект, для этого служит метод `setSource()`, которому ссылка на файл программы QML передается в виде объекта `QUrl`. Из этого факта вы можете сделать вывод, что в перспективе мы будем загружать QML-обертки наших программ Qt из Интернета. Обратите внимание на то, что файл модуля QML загружается из системы ресурсов приложения (при этом вместо обычного двоеточия необходимо использовать префикс `qrc`). В данном демонстрационном приложении это сделано для наглядности. Файл QML становится частью приложения и нам не нужно задумываться о том, где он будет находиться при сборке программы на разных платформах с помощью разных средств сборки. Однако во многих случаях размещение файлов QML в модулях ресурсов (встроенных в приложение или разделяемых) — не лучшая идея. Подробнее об этом будет сказано далее.

Далее виджет `qmlView` добавляется в окно обычным для программно созданных виджетов способом. В результате в окне нашей программы появится то же, что вы уже видели в окне, созданном утилитой qmlviewer (разумеется, путь к файлу qml, указанный в аргументе метода `setSource()`, вам надо заменить на свой собственный).

Для сборки программы в файл проекта необходимо добавить такую строку:

```
QT += declarative
```

Эта строка укажет, что в проект следует включить ссылки на библиотеки Qt, ответственные за работу с QML. Описание класса `QDeclarativeView` (единственного класса, предназначенного для QML, который нам пока что понадобился) хранится в заголовочном файле `<QtDeclarative/QDeclarativeView>`.

Наша программа выводит картинку, созданную с помощью простейшей программы QML, и вы можете поэкспериментировать с кодом файла helloworld.qml и убедиться в том, что даже существенно измененные файлы будут отображаться в нашей программе точно так же (иначе говоря, мы действительно добились разделения движка и интерфейса). Однако было бы преждевременно думать, что мы создали полноценный браузер QML. Хотя наша программа умеет показывать графические сцены, созданные с помощью QML (честно говоря, далеко не все), эти картинки и сама программа никак не взаимодействуют между собой, а ведь в этом взаимодействии и заключается смысл использования QML.

Наша следующая задача заключается в том, чтобы наладить взаимодействие между кодом Qt и QML, а это, в свою очередь, удобнее сделать, если код QML интерактивен, т. е. может реагировать на действия пользователя. Начнем с программы на QML (листинг 15.4), благо она не намного сложнее предыдущей. Этот файл называется interactive.qml. Полные исходные тексты этого примера вы сможете найти по адресу **http://symmetrica.net/qt47book/ch15**.

Листинг 15.4. Файл interactie.qml

```
import QtQuick 1.0
 Rectangle {
     width: 120
```

```
    height: 40
    color: palette.background

    TextInput {
      id: ti
        anchors.centerIn: parent
        color: palette.foreground
        text: "blue"
    }

    MouseArea {
        anchors.fill: parent
        onClicked: {
             palette.background = ti.text;
             ti.focus = true;
        }
    }
}
```

В этой программе мы встречаем два новых объекта QML: `MouseArea` и `TextInput`. Первый, `MouseArea`, как и подсказывает название, — это область, которая реагирует на события мыши. Строка `anchors.fill: parent` в теле элемента `MouseArea` означает, что данный элемент должен заполнить все пространство своего родительского элемента, в данном случае — элемента `Rectangle`. Кроме того, у элемента `MouseArea` есть ряд событий, похожих на события объекта Qt. Обработчик события `onClicked` вызывается всякий раз, когда пользователь щелкает мышью в области элемента `MouseArea`, т. е. в нашем случае — в области прямоугольника `Rectangle`. Прежде чем обсуждать обработчик события, рассмотрим другой новый элемент программы — объект `TextInput`. Этот элемент предназначен для ввода строки текста (таким образом, в нашей программе целых два объекта принимают данные, введенные пользователем). У объекта `TextInput` есть идентификатор `ti`, который используется в обработчике события `onClicked`. Благодаря строке `ti.focus = true;` элемент `TextInput` получает фокус ввода всякий раз, когда мы щелкаем мышью в прямоугольнике. Параметр `text` элемента `TextInput` содержит введенный пользователем текст, и его можно использовать для начального задания текста, который будет отображаться по умолчанию. Осталось изучить объект `palette`, который, судя по всему, управляет цветом прямоугольника (свойство `palette.background`) и текста строки ввода (свойство `palette.foreground`).

Изначально элемент ввода содержит текст `"blue"` и именно это значение будет присвоено свойству `palette.background` при щелчке мышью. В результате прямоугольник станет синим, а текст — желтым (желтый цвет дополняет синий до белого). Так работает объект `palette`. Если в строке текста вы введете `"yellow"`, то после щелчка мышью увидите, как текст и прямоугольник "поменяются" цветами. Можете поэкспериментировать и с другими сочетаниями, учтя при этом, что нужно вводить названия цветов, принятые в QML. А откуда взялся этот объект `palette`? Он не явля-

ется частью QML, а создан программой Qt. В Qt-программе этому объекту соответствует класс `CustomPalette`, объявленный в файле palette.h (листинг 15.5).

Листинг 15.5. Объявление класса `CustomPalette`

```
class CustomPalette : public QObject
 {
     Q_OBJECT
     Q_PROPERTY(QColor background READ background WRITE setBackground
         NOTIFY backgroundChanged)
     Q_PROPERTY(QColor foreground READ foreground WRITE setForeground
         NOTIFY foregroundChanged)
 public:
     CustomPalette();
     QColor background() const;
     void setBackground(const QColor &c);
     QColor foreground() const;
     void setForeground(const QColor &c);
signals:
     void foregroundChanged();
     void backgroundChanged();
 private:
     QColor m_background;
     QColor m_foreground;
 };
```

Примечание

Вы можете относиться к классу `CustomPalette` как к Qt-прототипу объекта `palette`, используемого в модуле QML, по аналогии с тем, как мы создавали в программах Qt прототипы объектов, которые затем использовались в сценариях QtScript.

Как мы видим, этот класс объявляет два свойства: `background` и `foreground`. Свойства, объявленные с помощью макроса `Q_PROPERTY()`, не имеют особого смысла для программы, написанной на C++, зато они очень полезны, когда объекты Qt взаимодействуют с кодом, написанным на других языках, в том числе и на QML. Что происходит в программе Qt, когда мы присваиваем значение свойству `background`? Вызывается метод `setBackground()`, который, как вы можете убедиться, ознакомившись с реализацией класса (листинг 15.6), присваивает новое значение переменной `m_background`, так что при чтении свойства `background`, за которое отвечает метод `background()`, оно будет возвращено. Далее метод `setBackground()` инвертирует полученный цвет и присваивает инвертированное значение переменной `m_foreground`, так что теперь свойство `foreground` вернет значение, дополняющее `background` до белого. В конце метод `setBackground()` эмитирует два сигнала: `backgroundChanged()` и `foregroundChanged()`. Обратите внимание, что в самой программе Qt обработчики для этих сигналов не предусмотрены. Дело в том, что эти сигналы предназначены для программы QML. Они должны оповестить соответствующие элементы программы о том, что цвета изменились. Именно поэтому имена этих сигналов стоят

после ключевого слова NOTIFY макроса Q_PROPERTY(). Чтобы понять, как это работает, вернемся к описанию объекта Rectangle. Строка color: palette.background указывает на то, что значение цвета этот объект должен получать из свойства palette.background, и, благодаря наличию сигнала backgroundChanged(), это будет происходить автоматически всякий раз, когда свойству palette.background будет присвоено новое значение. Точно так же будет вести себя и свойство palette.foreground.

Листинг 15.6. Реализация класса CustomPalette

```
CustomPalette::CustomPalette() : m_background(Qt::white),
                                 m_foreground(Qt::black) {}

QColor CustomPalette::background() const { return m_background; }

void CustomPalette::setBackground(const QColor &c) {
         if (c != m_background) {
             m_background = c;
             QRgb fg = c.rgb();
             fg = 0xFFFFFF - fg;
             m_foreground.setRgb(fg);
             emit backgroundChanged();
             emit foregroundChanged();
         }
}

QColor CustomPalette::foreground() const { return m_foreground; }
void CustomPalette::setForeground(const QColor &c) {
         if (c != m_foreground) {
             m_foreground = c;
             emit foregroundChanged();
         }
}
```

Обратите внимание также на то, что, хотя в программе QML мы используем текстовые обозначения цветов, свойства класса CustomPalette оперируют значениями QColor. Впрочем, класс QColor обладает средствами для конвертации текстовых обозначений цветов в другие их представления, так что ничего особенного тут нет. Вот, собственно говоря, и вся магия нашей программы. Осталось связать объект класса CustomPalette с объектом palette программы QML. Для начала загрузим программу в объект qmlView. Тут для нас нет ничего нового, за исключением того, что нам необходимо предоставить программе QML доступ к объекту класса CustomPalette.

Для понимания того, как это происходит, необходимо немного углубиться в иерархию классов Qt, отвечающих за взаимодействие с QML. Как уже было сказано, программа на QML представляет собой иерархию объектов, таких как Rectangle или TextInput. При загрузке программы в объект QDeclarativeView для каждого

объекта QML создается его "представитель" — объект `QDeclarativeContext`. Эти объекты образуют иерархию, соответствующую иерархии объектов QML, и предназначаются, вы правильно поняли, для обмена данными между программами (точнее, между соответствующими объектами) QML и Qt. Корневому объекту программы QML (в нашем случае объекту `Rectangle`) соответствует объект `QDeclarativeContext`, указатель на который возвращает метод `rootContext()` объекта `qmlView` класса `QDeclarativeView`. Данные, которые мы передаем этому объекту, становятся видны корневому компоненту программы QML и, по умолчанию, всем его дочерним компонентам.

> ***Примечание***
> Если вы думаете, что между классами `QDeclarativeContext` и `QScriptContext` существует функциональное подобие, вы недалеки от истины.

Таким образом, строчка:

```
qmlView->rootContext()->setContextProperty("palette", new CustomPalette);
```

из файла qmltest.cpp задает значение свойства (параметра) `palette` объекта `Rectangle`. Иначе этот параметр просто не имел бы смысла. Обратите внимание, что дочерние элементы наследуют свойство `palette`. Чтобы связать все вышесказанное в единое целое, рассмотрим текст конструктора класса `qmltest` (листинг 15.7).

Листинг 15.7. Конструктор класса `qmltest`

```
qmltest::qmltest(QWidget *parent, Qt::WFlags flags)
       : QDialog(parent, flags)
{
    ui.setupUi(this);
    QDeclarativeView *qmlView = new QDeclarativeView;
    qmlView->setSource(QUrl("qrc:/interactive.qml"));
    qmlView->rootContext()->setContextProperty("palette",
       new CustomPalette);
    QVBoxLayout *layout = new QVBoxLayout(this);
    layout->addWidget(qmlView);
}
```

Остается взглянуть на нашу интерактивную программу в работе (рис. 15.2).

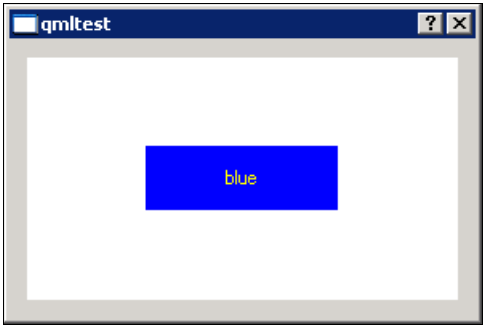


Рис. 15.2. Интерактивный модуль QML

Виджет QML в программе Qt

Пришла пора воспользоваться QML для создания "настоящего" виджета, который можно будет использовать в программе на Qt. Этот виджет представляет собой переработанный пример QML Dial из дистрибутива Qt 4.7.1. Как и пример из предыдущего раздела, рассмотренный здесь пример будет работать с версиями Qt 4.7.1, 4.7.2, 4.7.3 и, надеюсь, с более поздними, но не с более ранними.

Демонстрационная программа QML Dial из дистрибутива Qt — это самостоятельная программа, написанная на QML (я уже упоминал о том, что QML можно использовать как самостоятельный язык программирования). Нас же сейчас больше интересует применение QML для расширения возможностей интерфейсов программ Qt. Впрочем, преобразовать программу QML в виджет совсем несложно. Сам объект QML Dial представляет собой почти фотореалистичную имитацию стрелочного индикатора, такого как спидометр или индикатор давления жидкости в трубе. Мы не просто "перетащим" этот индикатор в свою программу Qt, но и дополним его некоторыми элементами.

Мы уже знаем, что важнейшей задачей QML является создание красивых графических интерфейсов, а это, как вы понимаете, невозможно без мощных средств работы с растровой графикой (рисовать красивые интерфейсы программным образом с помощью векторных функций несколько утомительно). Для работы с изображениями, хранимыми в файлах, в языке QML есть объект `Image{}`. Этот объект подобен тегу HTML `<img...>`, за исключением того, что он может гораздо больше. В простейшем варианте использование объекта `Image{}` выглядит так:

```
Image { source: "background.png" }
```

Эта конструкция загружает фон нашего индикатора (описание внешнего вида индикатора хранится в файле Dial.qml, расположенном в директории Dial и сопутствующих файлах с расширением png). На первый взгляд, тут все понятно без особых пояснений. Мы загружаем изображение из файла background.png. Вы можете поинтересоваться, куда именно мы загружаем изображение и где оно будет расположено. Вспомним, что визуальная часть QML основана на Qt Graphics View Framework, а эта система по умолчанию стремится расположить графические элементы так, чтобы геометрический центр сцены совпадал с центром окна вывода. Так что по умолчанию изображение просто займет место в центре окна нашего виджета (что нам и требуется).

Помимо собственных свойств, к каковым относится, например, свойство `source`, объект `Image{}` унаследовал от базовых объектов QML такие свойства как `x`, `y`, `scale`, `rotation`, `transform`, `anchors` и многие другие. Сам движок, выполняющий вывод изображения на экран, обладает многими полезными возможностями. Например, если загружаемый формат поддерживает альфа-канал, то при выводе изображения учитывается уровень прозрачности его элементов.

Рассмотрим некоторые свойства объекта `Image{}`. Если мы хотим выложить сцену фоновым рисунком как плиткой, надо добавить свойство `fillMode: Image.Tile`. Свойства `scale`, `rotation` и `transform`, как вы уже догадались, позволяют выполнять

преобразования изображения, такие как масштабирование и вращение. Например, можно вращать тень вслед за стрелкой нашего индикатора (листинг 15.8).

Листинг 15.8. Реализация вращения

```
transform: Rotation {
            origin.x: 9; origin.y: 67
            angle: needleRotation.angle
        }
```

Вся эта конструкция находится в теле объекта `image`, отвечающего за вывод тени под стрелкой (на нашей фотореалистической картинке тень должна двигаться вместе со стрелкой). В объекте `Rotation` мы задаем координаты центра вращения и угол поворота (как вы, вероятно, догадались, в этом замечательном языке двоеточие является оператором присваивания). Вращение самой стрелки реализовано еще интереснее (листинг 15.9).

Листинг 15.9. Вращение стрелки

```
Image {
        id: needle
        x: 98; y: 33
        smooth: true
        source: "needle.png"
        transform: Rotation {
            id: needleRotation
            origin.x: 5; origin.y: 65
            //! [needle angle]
            angle: root.angle
            Behavior on angle {
                SpringAnimation {
                    spring: 1.4
                    damping: .15
                }
            }
            //! [needle angle]
        }
    }
```

Бóльшая часть этого описания должна быть вам уже понятна. Мы устанавливаем координаты стрелки, присваивание значения `true` свойству `smooth` приводит к тому, что при выполнении преобразований изображения стрелки (в нашем случае это вращение) выполняется специальное фильтрование, сглаживающее эффекты "лесенки", которые могут в ходе этих преобразований возникнуть. Сравните координаты стрелки и координаты тени. Между прочим, поскольку на изображении (файл needle.png) стрелка смотрит вверх, наш индикатор по умолчанию указывает на зна-

чение в середине диапазона. Можно было бы изменить это, повернув стрелку на картинке или выполнив необходимое вращение при инициализации, но мы оставим все как есть, потому что так интереснее (рис. 15.3).

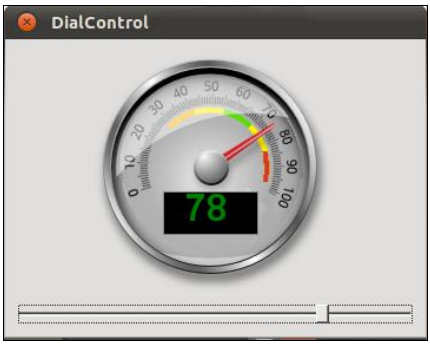


Рис. 15.3. Фотореалистический индикатор в программе Qt

Описание объекта `Rotation` начинается так же, как и в случае с тенью, однако дальше появляются различия. Конструкция `Behavior on angle {}` указывает, что должна делать стрелка, когда ее угол поворота меняется. Объект `SpringAnimation` создает специальный анимационный эффект при движении стрелки — эффект затухающих колебаний. Свойство `spring` указывает, насколько велика должна быть начальная амплитуда колебаний, а свойство `damping` определяет скорость их затухания. В результате стрелка нашего индикатора будет двигаться как настоящая стрелка на пружине. С помощью свойства `visible`, которым обладают все объекты QML, в том числе и объект `image`, мы можем управлять видимостью этих объектов.

Возможно, вас удивляет, что свойство `angle` объекта `needlrRotation` меняется всякий раз, когда меняется свойство `root.angle`. В таких языках как C++ одна операция присваивания означает одно изменение значение, но, кажется, в QML операция присваивания продолжает работать постоянно, и изменение значения свойства в правой части выражения присваивания приводит к изменению значения свойства в левой части, когда бы оно ни случилось. И это действительно так. В QML этот механизм назван связыванием свойств (property binding). Строка `angle: root.angle` связывает между собой свойства `root.angle` и `angle`, так что изменение значения первого свойства всегда будет приводить к изменению значения второго. Связывание свойств — это особая форма присваивания, которая используется всегда, когда слева от оператора присваивания указано свойство объекта, а справа — любое синтаксически корректное выражение языка JavaScript (свойство, функция и т. д.). То есть фактически связывание свойств является стандартным методом присваивания в QML, при котором свойство, стоящее слева от оператора присваивания, становится псевдонимом выражения, стоящего справа (вот почему связывать можно не только свойства, но и такие "пассивные" объекты как функции, которые сами не могут ничего инициировать). Связывание свойств — это тот механизм, который позволяет объектам QML обмениваться данными друг с другом, подобно тому, как объекты Qt обмениваются данными друг с другом с помощью сигналов и слотов. Точно так же, как в случае сигналов и слотов, связывание уже связанных свойств можно изменить с помощью элемента QML `PropertyChanges {}`.

Примечание

На мой взгляд, связывание свойств — очень полезная функциональная возможность, и если бы я проектировал новый объектно-ориентированный язык программирования, я обязательно включил бы в него этот механизм.

Далее в исходном примере на основание стрелки накладываются колпачок и стекло с бликом, изображения которых хранятся в файле overlay.png. Обратите внимание, что практически везде мы используем способность формата PNG сохранять частично прозрачные изображения, иначе такой сложный элемент управления у нас просто не получился бы. Обратите внимание также на то, что порядок расположения изображений поверх друг друга соответствует порядку следования представляющих их объектов в тексте программы QML. В общем-то, это естественно, если учесть, что виджет создается по мере выполнения программы.

Ко всему этому великолепию я добавил совсем немного. С объектом `Rectangle` мы уже встречались (листинг 15.10).

Листинг 15.10. Элемент `Rectangle` и `Text`

```
Rectangle {
x: 61
y: 118
width: 80
height: 36
color: "black"
border.color: "#888888"
}

Text {
  color: "green"
 text: root.angle/2 + 50
 x: 80
 y: 114
 font.pointSize: 24; font.bold: true
 style: Text.Raised
 styleColor: "black"
}
```

Новое здесь — свойство `border.color`, которое позволяет задать цвет границы прямоугольника. Объект `Text` дублирует значение, которое указывает индикатор. При анализе кода, отвечающего за вращение стрелки, следует учитывать, что по умолчанию стрелка индикатора смотрит вверх, а значит, фактические углы поворота следует указывать относительно этого положения, причем повернутая на некоторый угол стрелка при новом повороте всегда интерпретирует новый угол так, как если бы она начинала вращаться из исходного положения.

Переходим теперь ко второй части нашей работы — превращению модуля QML в виджет Qt. С основными операциями, которые необходимы для этого, мы уже

знакомы. Однако сейчас нам есть что добавить. Представителем виджета QML в нашей программе Qt является класс `Dial` (листинг 15.11).

Листинг 15.11. Класс `Dial`

```
class Dial : public QObject
 {
     Q_OBJECT
     Q_PROPERTY(int angle READ angle WRITE setAngle NOTIFY angleChanged)
 public:
     Dial();
     int angle();
     void setAngle(int a);
 signals:
     void angleChanged();
 private:
     int m_angle;
 };
```

В определении этого класса нет ничего сложного. Класс экспортирует единственное свойство `angle`. Напомню только, что метод `setAngle()` должен явным образом эмитировать сигнал `angleChanged()`, иначе виджет QML никогда не узнает, что угол изменился. Возможно, вас удивляет, что в сигнале `angleChanged()` не передается новое значение угла поворота. Если бы сигнал предназначался для других классов Qt, я бы так и сделал, но сигнал предназначен для виджета QML, а соответствующий объект этого виджета все равно прочитает значение свойства `angle` с помощью метода `angle()` (указанного после ключевого слова `READ` в макросе `Q_PROPERTY`). Так что передавать какой-либо параметр в сигнале Qt просто нет необходимости. Создание виджета в окне Qt не несет в себе ничего принципиально нового (листинг 15.12).

Листинг 15.12. Создание виджета в окне программы Qt

```
QDeclarativeView *qmlView = new QDeclarativeView;
dial = new Dial();
qmlView->rootContext()->setContextProperty("Dial", dial);
qmlView->setSource(QUrl("qrc:/Dial/Dial.qml"));
QVBoxLayout *layout = new QVBoxLayout(this);
layout->addWidget(qmlView);
```

Теперь уместно вернуться к вопросу о том, где лучше хранить тексты модулей QML и связанные с ними графические ресурсы. Движок Qt QML умеет работать с пространством ресурсов приложения (и с другими пространствами URL) как с локальной файловой системой. Так что если включенному в модуль ресурсов модулю QML понадобится файл, тоже включенный в модуль ресурсов, модуль QML смо-

жет без проблем загрузить его. При этом важно, чтобы ссылки на файлы были относительными, а не абсолютными. Технических проблем при хранении всего модуля QML в составе модуля ресурсов не возникает. С одной стороны, это упрощает установку программы: вам не нужно думать о том, где должны быть размещены файлы QML в соответствии со стандартом XDG, и если вы пишете кроссплатформенное приложение, ваша жизнь упрощается еще больше. С другой стороны, при таком подходе теряется одно из важнейших преимуществ QML как средства описания интерфейсов программ Qt: возможность радикально сменить интерфейс без повторной сборки приложения. В случае расположения виджета QML целиком в собственных файлах дизайнеру для изменения интерфейса будет достаточно обычного текстового редактора и редактора Photoshop (или GIMP). В общем, наиболее разумный выбор зависит от конкретных целей, в частности от того, хотите ли вы, чтобы каждый пользователь, освоивший QML и растровую графику, мог "сшить новую одежку" для вашей программы.

До сих пор мы использовали метод `setContextProperty()` для передачи указателя на созданные нами объекты со свойствами. Но его возможности гораздо шире. Прежде всего, отметим, что существует и другой вариант метода `setContextProperty()`:

```
void setContextProperty(const QString & name, const QVariant & value)
```

То есть со свойством контекста QML можно связать не только объект, но и любое значение, приводимое к типу `QVariant`. Например:

```
qmlView->rootContext()->setContextProperty("HelloText", trUtf8("Hello World!");
```

Объекты, которые мы передаем контекстам, могут экспортировать в QML не только свойства, но и методы. Для того чтобы модуль QML увидел метод объекта, этот метод должен быть объявлен в разделе `public slots`. Например, если добавить в класс `Dial` публичный слот `int angleToPos(int angle)`, то в файле Dial.qml можно написать так:

```
property int angleToPos : dial.angleToPos(angle)
```

Поскольку наша сложная конструкция QML может нуждаться в отладке, в программе предусмотрен способ вывода информации об ошибках, которые возникают в ходе выполнения программы на QML. Вообще-то, в ОС Linux все сообщения об ошибках QML выводятся в стандартный поток вывода той консоли, с которой была запущена программа. В ОС Windows по неизвестным причинам Qt этого не делает: даже если запустить программу из окна командной строки Windows, никаких сообщений об ошибках мы не увидим (независимо от того, есть ли ошибки). В любом случае, мы пишем графическое приложение, не обязательно запускаемое из окна консоли, так что неплохо обзавестись собственным методом вывода информации об ошибках, основанном на графическом интерфейсе.

Объект класса `QDeclarativeError` имеет несколько свойств и методов, из которых мы воспользуемся методом `toString()`. Этот метод возвращает информацию об ошибке на русском языке в переменной `QString` (листинг 15.13), где `errors` — объект указанного ранее класса, производного от `QList`.

Листинг 15.13. Вывод сообщений об ошибках в модуле QML

```
errors = qmlView->errors();
for(int i = 0; i < errors.count(); i++)
MessageBox::critical(this, QString::fromUtf8("Ошибка QML:"),
    errors.at(i).toString());
```

Вы, наверное, обратили внимание, что я написал, что метод `errors()` возвращает информацию об актуальных ошибках, т. е. о тех, которые существуют на момент его вызова. Тому, кто привык к компилируемым языкам программирования, это может показаться необычным, но в динамически исполняемых программах QML ошибки могут возникать и исчезать динамически и не каждая ошибка приводит к аварийному завершению программы.

В связи с этим полезно сравнить вывод программы на консоль Linux (куда поступает информация обо всех ошибках) с нашим выводом, полученным с помощью класса `MessageBox`. Таким образом, например, я узнал, что связывать корневой контекст с объектом `Dial` лучше до загрузки текста программы QML, ведь выполнение программы начинается немедленно после вызова метода `setSource()`. Если сразу после вызова `setSource()` программа на QML не сможет инициализировать свой объект `Dial`, это вызовет ошибку, которая однако будет устранена, как только мы укажем программе, чем именно следует инициализировать этот объект. Иначе говоря, инициализировать корневой контекст можно и до вызова `setSource()`, и после этого вызова, но первый способ работает корректнее.

И, наконец, для того, чтобы наш виджет выглядел, как любой другой виджет Qt, мы удалим белый фон, заботливо созданный для нас объектом `QDeclarativeView` (не забывайте, что это потомок `QGraphicsView`). Мы сделаем это так:

```
qmlView->setBackgroundRole(QPalette::Background);
```

Поскольку наш виджет только показывает значения, но не позволяет их вводить, нам следует добавить в окно программы еще один виджет, предназначенный для управления виджетом QML. Как и в исходном примере программы QML, мы воспользуемся для этого ползунком, только в нашем случае это будет объект класса `QSlider`. Наш виджет будет реагировать на сигнал `sliderChanged()` этого объекта. Можно было бы добавить в класс `Dial` слот и связать этот слот с сигналом `sliderChanged()` напрямую, я оставляю вам это в качестве домашнего задания.

Самостоятельная программа на языке QML

Самостоятельная программа на QML не так уж сильно отличается от модулей QML, используемых в приложениях Qt. Фактически модуль всегда можно превратить в программу и наоборот, в чем мы только что убедились. Существенное отличие — файл с расширением qmlproject. Само расширение говорит о том, что задача этого файла — описание проекта QML. Внутри файл описания проекта представляет собой типичный файл модуля, написанного на QML (листинг 15.14).

Листинг 15.14. Файл проекта QML

```
Project {
    /* Include .qml, .js, and image files
       from current directory and subdirectories */
    QmlFiles {
        directory: "."
    }
    JavaScriptFiles {
        directory: "."
    }
    ImageFiles {
        directory: "."
    }
    /* List of plugin directories passed to QML runtime */
    // importPaths: [ " ../exampleplugin " ]
}
```

Структура файла с расширением qmlproject похожа на структуру модуля QML, хотя модулем QML он, строго говоря, не является. Как и практически любой файл QML, файл описания проекта начинается с директивы `import`. Файл описания проекта импортирует пакет `QmlProject` с обязательным, как и во всех подобных случаях, указанием версии — 1.0. Далее идет ключевое слово `Project`, за которым в фигурных скобках следует сам модуль. Внешне все выглядит так, словно в QML имеется элемент "`Project{}`", аналогичный элементу `Rectangle{}` или `MouseArea{}`. Но в официальном перечне элементов QML такой элемент не упомянут, поэтому я воздержусь от того, чтобы называть его элементом. Вообще, на момент работы над этой книгой было очень мало официальной документации по файлу описания проекта. Вероятно, эта часть QML еще будет меняться в следующих версиях. На самом деле файл с расширением qmlproject и не является обязательным для запуска программы QML. Файлы описания проекта требуются, прежде всего, программе Qt Creator, которая берет из этих файлов информацию о проекте. Qt Creator умеет выполнять синтаксический анализ файлов QML, находить в них ошибки, давать подсказки и даже выполнять автоматическое завершение идентификаторов, все как в C++ (рис. 15.4).

Контекстная справка по QML работает так же хорошо, как и справка по Qt (только в связи с быстрыми изменениями языка написанное в ней не всегда отражает реальность). При запуске программы QML на выполнение из Qt Creator вызывается утилита qmviewer. Позаботьтесь только о том, чтобы Qt Creator был настроен на работу с той версией Qt, которая поддерживает новейшую версию QML.

Вернемся к файлу с расширением qmlproject. Имена структур `QmlFiles{}`, `JavaScriptFiles{}` и `ImageFiles{}` говорят сами за себя. У каждой из этих структур есть свойство `directory`, позволяющее указать директорию, в которой хранятся файлы QML, JavaScript и графики соответственно. В нашем примере все эти файлы хранятся в той же директории, что и файл описания проекта. Свойство `importPaths`,

вероятно, должно хранить списки директорий, из которых программа загружает расширения, хотя для каждого конкретного расширения в каждом модуле, который его загружает, можно указать ту же информацию.

Более того, если у нас есть директория, которая содержит файлы JavaScript, QML и графические файлы, составляющие некий вспомогательный модуль программы, мы можем указать в главном модуле QML имя этой директории (в кавычках, после ключевого слова `import`) и система QML сама найдет все, что нужно.

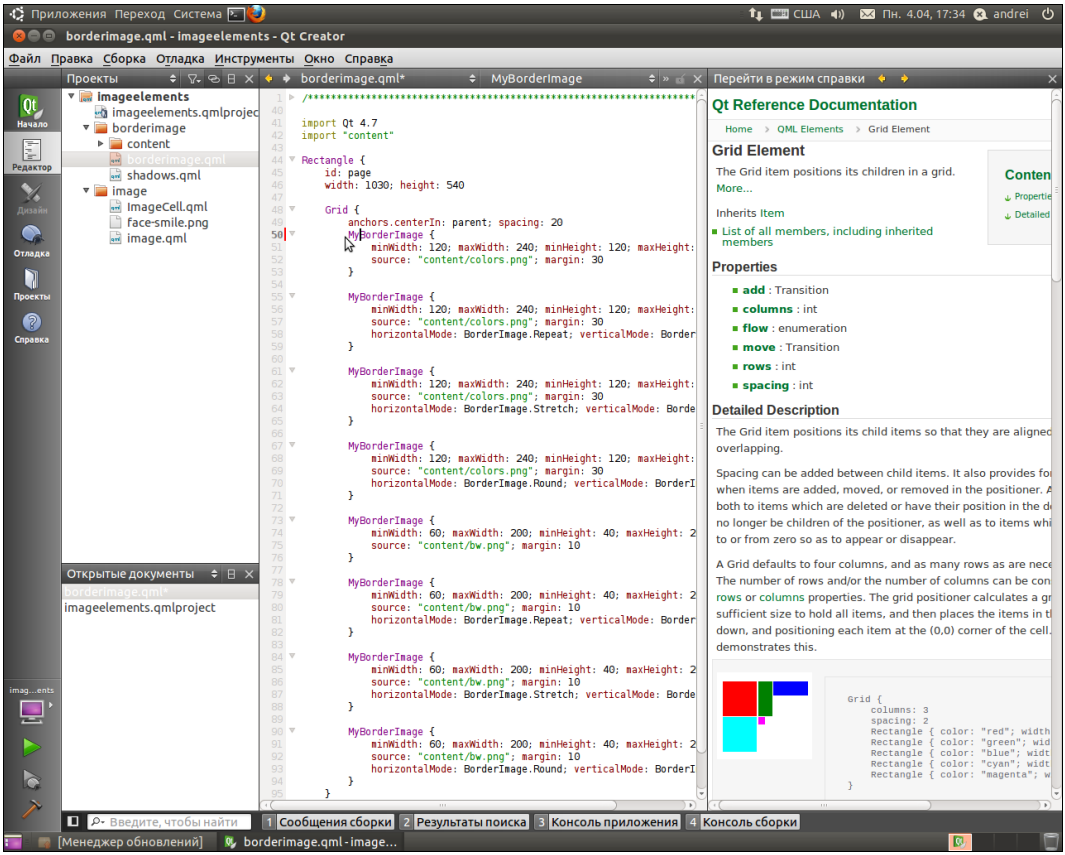


Рис. 15.4. Программа QML в Qt Creator

Изменения и дополнения

Как известно, один из самых простых способов начать изучение нового языка — взять текст несложной программы на этом языке и путем небольших изменений превратить ее в то, что нужно программисту. При этом важно, чтобы на каждом этапе трансформации программа работала или четко давала понять, что именно программист сделал не так. Применительно к современному QML, однако, подобный подход приводит к специфическим трудностям, связанным с тем, что сам язык QML еще не обрел окончательную форму и даже очень простые примеры про-

грамм, взятые не из той версии, могут не работать просто в силу несовместимости версий. Разумеется, все системы с годами меняются, если, конечно, разработчики не забросили их. Но сейчас изменения в QML вносятся слишком часто. То, что работало в версии Qt 4.7, может не работать (и даже, скорее всего, не будет работать) в Qt 4.7.1, не говоря уже о коде из Qt 4.8. Вот, например, некоторые изменения, которые претерпел QML за последнее время:

- элемент `LineEdit` теперь называется `TextInput`, а элемент `MouseRegion` — `MouseArea`;
- раньше индивидуальный идентификатор (значение свойства `id`) элемента QML мог начинаться с заглавной буквы, а теперь не может;
- в прежних версиях, если вы объявляли свойство с именем, аналогичным имени унаследованного свойства элемента QML, унаследованное свойство перекрывало ваше, а теперь все наоборот;
- раньше QML имел гораздо более мягкую систему типов, в которой любое значение изначально рассматривалось как строка, а уже затем приводилось к нужному типу.

Программа clocks

А теперь рассмотрим саму программу на QML. Я создал ее на основе демонстрационной программы clocks, которая входит в дистрибутив Qt. Исходная программа clocks показывает текущее время в трех разных городах Земли (рис. 15.5), используя время, установленное в компьютере, и "зашитые" в код названия городов и смещения по времени.

Рис. 15.5. Исходная программа clocks

Мы дополним эту программу элементами интерактивности. В результате пользователь сам сможет выбирать имена городов и смещения времени относительно Гринвича, а также выбирать цвет фона окна программы с помощью специальной цветовой панели (рис. 15.6).

С точки зрения изучения языка QML эта программа интересна для нас, прежде всего, тем, что в ней мы впервые встречаемся с элементами, которые управляют рас-

положением нескольких виджетов. В нашей программе присутствуют три ряда виджетов: верхний ряд с дисками часов, ниже ряд для ввода имен городов и смещений, еще ниже — ряд для выбора цвета фона. Для создания ряда виджетов нам потребуется элемент `Row{}`. Задача этого элемента — расположить дочерние элементы в один горизонтальный ряд. Для того чтобы создать ряд, например, прямоугольников, строим иерархическую конструкцию (листинг 15.15).

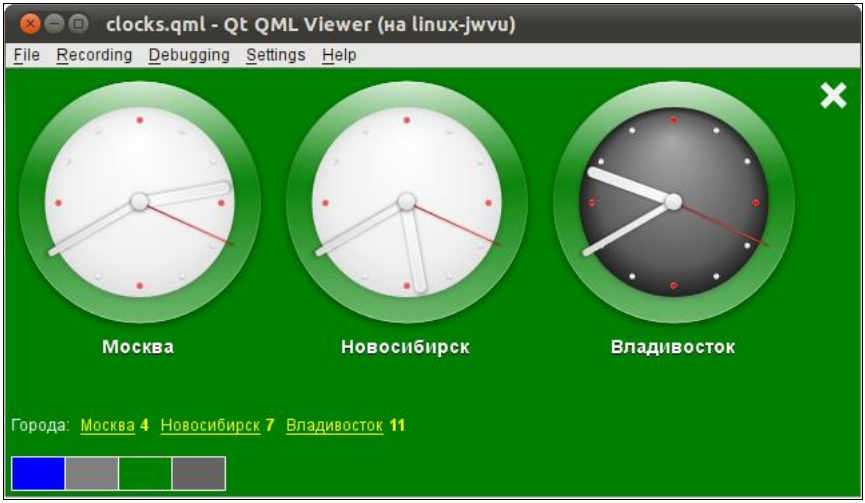


Рис. 15.6. Модифицированная программа clocks

Листинг 15.15. Описания ряда, содержащего прямоугольники

```
Row {
Rectangle {
...
}
Rectangle {
id : leftCell
...
}
Rectangle {
anchors.left : leftCell.right
...
}
}
```

Если вам требуется более сложное расположение (с разными интервалами между элементами ряда, особым расположением элементов, различающихся по высоте, и т. п.), можно поиграть со свойствами `anchors`, а также `left` и `right`. При этом, например, свойство `anchors.left` элемента может быть связано со свойством `right` соседа слева, если, конечно, у соседа слева инициализировано свойство `id`.

Многие другие элементы в рассматриваемой программе должны быть вам уже знакомы. Обратите внимание на то, что, когда вы вводите другое название города, подпись под соответствующими часами меняется непосредственно в процессе ввода. Так работает связывание свойств `text` элементов `TextInput` и `Text` (первый предназначен для ввода текста, второй — для вывода его под часами). Для изменения цвета фона окна достаточно щелкнуть мышью по соответствующему прямоугольнику. Благодаря тому, что для вывода некоторых элементов часов используется полупрозрачная растровая графика, цвет этих элементов так же будет меняться вместе с цветом фона.

Мы уже использовали конечные автоматы для проектирования виджетов, поэтому вас не должно удивлять, что в языке QML используются две важных концепции — состояния и переходы. Что это такое, легко объяснить на примере кнопки. У кнопки может быть два состояния — обычное и нажатое (под нажатым состоянием понимается состояние кнопки, когда на нее "надавили" указателем мыши). Между этими двумя состояниями существуют два перехода — из обычного состояния в нажатое и обратно. Переходы определяют графические эффекты, которыми сопровождается изменение состояния виджета. В самом простом случае без переходов можно вообще обойтись, достаточно просто перерисовать виджет в новом состоянии, но наличие переходов делает поведение виджетов гораздо более привлекательным и натуралистичным (или наоборот, необычным и привлекательным). Использование состояний и переходов позволяет заставить цветные прямоугольники вести себя как настоящие кнопки. В листинге 15.16 приведен код элемента `Item`, использующего состояния и переходы.

Листинг 15.16. Элемент `Item`, использующий состояния и переходы

```
Item {
id: cell1; width: 40; height: 25
Rectangle {
id : rect1; color: "blue"; border.color: "white"; anchors.fill: parent
     }
MouseArea {
 id : mouseArea; anchors.fill: parent; onClicked: root.color = "blue"
}
states: State {
        name: "down"; when: mouseArea.pressed == true
             PropertyChanges { target: rect1; border.color : "#000088";
                 border.width : 2 }
        }
transitions: Transition {
from: ""; to: "down"; reversible: true
             }
 }
```

Свойству `states` элемента `Item` (в нашем примере этот элемент имеет идентификатор `cell1`) можно присвоить список дополнительных состояний. Мы описываем

состояние "down", которое должен принимать элемент cell1, когда кнопка мыши нажата в области этого элемента. Структура PropertyChanges описывает само состояние. Целью состояния является прямоугольник rect1, у которого должны измениться толщина и цвет обрамляющей линии. Вообще говоря, "описать новое состояние элемента" означает в QML "задать новые значения различных свойств элемента".

Аналогичным образом, свойство transitions содержит список переходов. Мы описываем один переход — из состояния по умолчанию в состояние "down". Обратите внимание, что свойству reversible присвоено значение true, которое указывает, что элемент должен переходить из состояния "down" в исходное, как только исчезнет условие, вызывающее состояние "down". Причем этот переход будет зеркальным отражением перехода из состояния по умолчанию в состояние "down". Если состояния определяют новые значения свойств элемента, то переходы определяют, как именно исходные значения свойств должны трансформироваться в новые. Например, с помощью переходов можно определить скорость изменения значения числового свойства. Для свойства типа color, определяющего цвет, можно также указать, с какой скоростью исходный цвет должен трансформироваться в конечный, и возможно, задать промежуточные цвета. Например, мы можем добавить в описание перехода анимационный эффект (листинг 15.17).

Листинг 15.17. Анимация перехода

```
ParallelAnimation {
ColorAnimation { duration:500}
NumberAnimation { duration:500}
}
```

Это описание указывает, что в процессе перехода должны одновременно меняться значения свойств численных типов и свойств, определяющих цвет, причем оба изменения должны занимать полсекунды. Начальные и конечные значения свойств при этом заданы состояниями, между которыми выполняется переход, а промежуточные значения в данном случае QML выбирает по умолчанию. В модифицированном примере clocks такая анимация применена к кнопке выбора синего цвета, что вызывает несколько необычное поведение этой кнопки.

Утилита qmlviewer

Я уже писал о том, что утилита qmlviewer не представляет собой ничего особенного в том смысле, что вы и сами легко сможете воспроизвести ее функциональность. Это, конечно, верно. Тем не менее стоит ли изобретать велосипед, когда утилита, причем весьма удобная, у нас уже есть?

Что может qmlviewer? Прежде всего, утилита может загружать модули и программы QML и выполнять их. Если программа QML содержит элементы анимации, в целях отладки скорость анимации можно снизить. Но этим возможности qmlviewer не исчерпываются. В отличие от Linux, в Windows нельзя увидеть сооб-

щения об ошибках, которые система интерпретации QML выводит на консоль. Впрочем, в Windows это трудно сделать чисто технически, поскольку, в отличие от Linux, графическая программа Windows тут же теряет связь с консолью, из-под которой она запущена. Программа qmlviewer исправляет этот недостаток (рис. 15.7). Команда меню **Debugging** (Отладка) | **Show warnings...** (Показывать предупреждения) открывает специальное окно, в котором можно увидеть подробное описание проблемы и номер строки исходного текста, в которой она возникла. Самое удобное в утилите qmlviewer — возможность быстро перезагрузить модуль QML после внесения изменений. Для этого достаточно нажать клавишу <F5>.

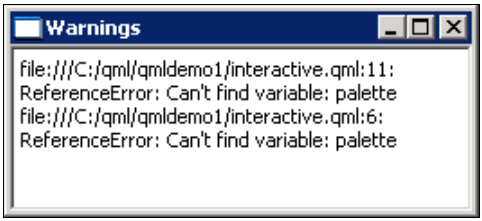


Рис. 15.7. Предупреждения программы qmlviewer

Единственный, на мой взгляд, недостаток программы qmlviewer заключается в том, что она не интегрирована с текстовым редактором, в котором можно было бы сразу отредактировать обнаруженные недочеты. Впрочем, наличие Qt Creator с поддержкой QML делает этот недостаток не таким уж существенным. Можно было бы ожидать, что утилита qmlviewer будет загружать файлы с расширением qmlproject, но это не так. Если программа состоит из нескольких файлов QML, загрузить нужно главный из них. А для того чтобы понять, какой файл главный, нужно разобраться в структуре программы. Впрочем, обычно это несложно. Кроме того, утилита qmlviewer умеет загружать файлы QML из Интернета по протоколу HTTP и даже позволяет настроить HTTP-прокси.

На этом мы завершаем разговор о языке QML, хотя, если вы захотите использовать QML для создания собственных виджетов Qt, ваше знакомство с ним только начинается.

Заключение

На протяжении всей этой книги неоднократно говорилось о том, что даже в рамках перечня рассмотренных тем изложение не претендует на то, чтобы охватить все аспекты Qt. Многие технологии, реализованные в библиотеке Qt, могли бы стать темами отдельных книг. И, скорее всего, в будущем это произойдет, например, с такой технологией как QML.

Я, однако, надеюсь, что мне удалось выполнить поставленную задачу — познакомить читателя с практическими аспектами Qt и показать, как эта библиотека может быть адаптирована для решения различных нетривиальных задач.

В отличие от технологий разработки Borland-Embarcadero, перспективы которых представляются мне туманными, библиотека Qt не только выдержала проверку временем, разными платформами разработки и самыми жесткими требованиями к производительности и стабильности приложений, но и продолжает активно развиваться. Важную роль в развитии и поддержке Qt играет сообщество разработчиков программ с открытыми исходными текстами (open source software). Это означает, что как бы ни изменились в будущем приоритеты нынешнего владельца Qt, компании Nokia, и что бы ни случилось с самой компанией, Qt не умрет. В некотором смысле различия между VCL и MFC с одной стороны и Qt с другой стороны похожи на различия между языками Delphi Language и C++. В то время как первое (в обеих парах сравнения) является исключительной собственностью одной компании, от которой целиком зависит его дальнейшая судьба, второе не является уже чьей-то исключительной собственностью (в случае с Qt это не так очевидно, и тем не менее это так). А значит, у C++ и Qt больше шансов на будущее, чем у Delphi, VCL или MFC. Из этого, в свою очередь, следует, что изучение библиотеки Qt является одной из самых надежных "интеллектуальных инвестиций" в мире прикладного программирования, а, стало быть, и программистам, выбирающим Qt, должна сопутствовать удача.

В заключение я хотел бы сказать, что буду благодарен за любые отзывы, пожелания и исправления ошибок (неизбежных, но, как я надеюсь, немногочисленных). Вы можете связаться со мной с помощью электронной почты **anb@symmetrica.net**. Исправления и дополнения к книге, если таковые появятся, вы сможете найти там же, где и тексты примеров для книги, на сайте **http://symmetrica.net/qt47book/**.

Список литературы

1. Арлоу Д., Нейштадт И. UML 2 и Унифицированный процесс. Практический объектно-ориентированный анализ и проектирование, 2-е издание. — СПб.: Символ-Плюс, 2007.
2. Макконнелл С. Совершенный код. Практическое руководство по разработке программного обеспечения, — СПб.: Питер, 2005.
3. Ульман Д., Уидем Д. Основы реляционных баз данных. — М.: Лори, 2006.
4. Фаулер М. Рефакторинг. Улучшение существующего кода. — СПб.: Символ-Плюс, 2003.
5. Эндрюс Г. Основы многопоточного, параллельного и распределенного программирования. — М.: Вильямс, 2002.
6. Qt Online Reference Documentation (**http://doc.qt.nokia.com**).
7. Qt, справочная документация (**http://doc.crossplatform.ru**).

Предметный указатель

A

accept(), метод 68, 266
addDatabase(), метод 235
addMapping(), метод 263
addPixmap(), метод 356
addState(), метод 406
addTransition(), метод 405
addWidget(), метод 35, 360
Arthur 271
Aspell 318

C

checkSyntax(), метод 420
ChooseFontLayout, класс 35
CMake 19, 57, 92
collidingItems(), метод 359
cppcheck 133

D

database(), метод 237
deleteLater(), метод 73
Drag and Drop 66
dragEnterEvent(), метод 66
dropEvent(), метод 66

E

Eclipse 17
ECMAScript 418
event(), метод 64
eventFilter(), метод 73

G

GDB 125
getDisplayValue(), метод 249
GLSL 281
GNU build system 18
Graphics View Framework 340

I

installEventFilter(), метод 73
installTranslator(), метод 28
Interview Framework 96, 235
itemAt(), метод 359

K

keyPressEvent(), метод 63

L

lrelease, утилита 27
lupdate, утилита 25

M

MainWindow, класс 66
map(), метод 226
mapToItem(), метод 358
mapToScene(), метод 358
Microsoft Visual Studio 15
mpg123 107

N

newFunction(), метод 428
newQObject(), метод 421
notify(), метод 65
NULL 242

O

ODBC 235
OpenCL 293

OpenGL 272
Opera 388

P

postEvent(), метод 69
Process Monitor, утилита 231
processEvents(), метод 72

Q

QAbstractAnimation, класс 414
QAbstractItemDelegate, класс 96, 251
QAbstractItemModel, класс 96
QAbstractItemView, класс 96
QAction, класс 41, 240, 327
QApplication, класс 21
QAtomicInt, класс 181
QAtomicPointer, класс 206
QAudioDeviceInfo, класс 448
QAudioFormat, класс 448
QAudioOutput, класс 447
QBuffer, класс 307
QCLContext, класс 297
QComboBox, класс 35, 269
QCoreApplication, класс 451
QDataWidgetMapper, класс 263
QDeclarativeView, класс 463
QDialog, класс 32, 264
QEvent, класс 64, 454
QFile, класс 226
QFileDialog, класс 47
QFinalState, класс 406
QFormLayout, класс 31
QGLBuilder, класс 290
QGLContext, класс 276
QGLDome, класс 290
QGLFormat, класс 276
QGLInfo, класс 292
QGLMaterial, класс 291
QGLPainter, класс 286
QGLSceneNode, класс 289
QGLShader, класс 280
QGLShaderProgram, класс 280
QGLTexture2D, класс 289
QGLView, класс 287, 288
QGLWidget, класс 272, 276
QGraphicsAnchorLayout, класс 389
QGraphicsGridLayout, класс 391
QGraphicsItem, класс 341, 356
QGraphicsLayout, класс 389
QGraphicsObject, класс 342
QGraphicsPixmapItem, класс 356
QGraphicsProxyWidget, класс 361
QGraphicsScene, класс 341, 359, 360
QGraphicsView, класс 341, 380
QGraphicsWebView, класс 392
QGraphicsWidget, класс 342, 388, 389, 395
QGrid, класс 31
QHBox, класс 31
QImage, класс 299
QImageReader, класс 301
QIODevice, класс 307, 448
QLabel, класс 300
QLibrary, класс 449
QLineEdit, класс 61, 257
QMainWindow, класс 274
qmake, утилита 22
QMenu, класс 327
QML 461
qmlviewer, утилита 461, 481
QModelIndex, класс 239
QMutex, класс 189
QMutexLocker, класс 189
qmviewer, утилита 476
QObject, класс 28
QPaintEngine, класс 271
QPainter, класс 273, 337
QParallelAnimationGroup, класс 414
QPicture, класс 299
QPixmap, класс 274, 299, 302
QPlane3D, класс 287
QPluginLoader, класс 446
QPoint, класс 327
QPointer, класс 105
QPropertyAnimation, класс 414
QRay3D, класс 287
QRect, класс 301
QRegExp, класс 347
QResource, класс 52
QRunnable, класс 218
QScopedPointer, класс 106
QScriptEngine, класс 419, 420, 428
QScriptProgram, класс 436
QScriptValue, класс 420
QScrollArea, класс 300
QSemaphore, класс 192
QSequentialAnimationGroup, класс 414
QSharedMemory, класс 189
QSignalMapper, класс 82
QSignalTransition, класс 404
QSizePolicy, класс 36
QSpinBox, класс 35
QSqlDatabase, класс 235

QSqlRelationalTableModel, класс 103, 242, 269
QSqlTableModel, класс 243, 249
QState, класс 404
QStateMachine, класс 404
QStaticText, класс 335
QString, класс 349
QStyle, класс 253
QStyledItemDelegate, класс 251
QStyleOptionViewItemV4, класс 253
QSysInfo, класс 93
Qt Creator 14
Qt Designer 37
Qt Framework 16
Qt Linguist 25
Qt SDK 14
Qt Solutions 457
Qt/3D 286
QTableView, класс 238
QTextBlock, класс 317
QTextCharFormat, класс 323
QTextCodec, класс 28
QTextDocument, класс 316
QTextDocumentWriter, класс 328
QTextEdit, класс 317
QTextFrame, класс 317
QTextLayout, класс 332
QTextLine, класс 334
QTextList, класс 317
QTextStream, класс 451
QTextTable, класс 317
QTextTableCell, класс 317
QThread, класс 169
QThreadPool, класс 219
QThreadStorage, класс 188
QTranslator, класс 27
QTriangle3D, класс 287
QtScript 418
QVariantAnimation, класс 414
QVBox, класс 31
QVBoxLayout, класс 35
QWaitCondition, класс 193
QWidget, класс 365
QXmlStreamReader, класс 353

R

rcc, утилита 52
releaseThread(), метод 219
reserveThread(), метод 219
run(), метод 168

S

Scribe 316
sendEvent(), метод 70
Service Control Manager 458
setData(), метод 356
setInitialState(), метод 406
setItemDelegate(), метод 266
setRenderHint(), метод 273
SQLite 98, 235
sqlite3, утилита 241
Subversion 145
svn 148
svnadmin, утилита 145

T

TortoiseSVN 159
tr(), метод 28
trUtf8(), метод 28
Tulip 57

U

uic, утилита 43
Unit Test Library 116
unmap(), метод 226

V

Visual Assist X 135

X

XML 346

А

Автоматическое завершение строки 61
Алгоритм Бойера—Мура 229
Атомарная операция 180

Б

База данных, пример 98
Бойера—Мура, алгоритм 229
Большое изображение 298
Буфер
◊ создание 204
◊ чтение данных 204

В

Ввод/вывод 451
Взаимная блокировка 172
Видеокарта 271
Встраивание виджетов 360
Выделение метода 135

Г

Графическая система 271
Графический
◊ виджет 395
◊ интерфейс 461

Д

Двоичное разбиение пространства 381
Двухмерная графика 340
Делегат 238, 250
Демон 457
Добавление изображения 329

З

Завершение программы 452
Заголовок таблицы 261

И

Извлечение изображения 330
Изменение размеров элемента 36
Интернационализация 24
Интерфейс 438

К

Класс
◊ ChooseFontLayout 35
◊ MainWindow 66
◊ QAbstractAnimation 414
◊ QAbstractItemDelegate 96, 251
◊ QAbstractItemModel 96
◊ QAbstractItemView 96
◊ QAction 41, 240, 327
◊ QApplication 21
◊ QAtomicInt 181
◊ QAtomicPointer 206
◊ QAudioDeviceInfo 448
◊ QAudioFormat 448
◊ QAudioOutput 447
◊ QBuffer 307
◊ QCLContext 297
◊ QComboBox 35, 269
◊ QCoreApplication 451
◊ QDataWidgetMapper 263
◊ QDeclarativeView 463
◊ QDialog 32, 264
◊ QEvent 64, 454
◊ QFile 226
◊ QFileDialog 47
◊ QFinalState 406
◊ QFormLayout 31
◊ QGLBuilder 290
◊ QGLContext 276
◊ QGLDome 290
◊ QGLFormat 276
◊ QGLInfo 292
◊ QGLMaterial 291
◊ QGLPainter 286
◊ QGLSceneNode 289
◊ QGLShader 280
◊ QGLShaderProgram 280
◊ QGLTexture2D 289
◊ QGLView 287, 288
◊ QGLWidget 272, 276
◊ QGraphicsAnchorLayout 389
◊ QGraphicsGridLayout 391
◊ QGraphicsItem 341, 356
◊ QGraphicsLayout 389
◊ QGraphicsObject 342
◊ QGraphicsPixmapItem 356
◊ QGraphicsProxyWidget 361
◊ QGraphicsScene 341, 359, 360

◊ QGraphicsView 341, 380
◊ QGraphicsWebView 392
◊ QGraphicsWidget 342, 388, 389, 395
◊ QGrid 31
◊ QHBox 31
◊ QImage 299
◊ QImageReader 301
◊ QIODevice 307, 448
◊ QLabel 300
◊ QLibrary 449
◊ QLineEdit 61, 257
◊ QMainWindow 274
◊ QMenu 327
◊ QModelIndex 239
◊ QMutex 189
◊ QMutexLocker 189
◊ QObject 28
◊ QPaintEngine 271
◊ QPainter 273, 337
◊ QParallelAnimationGroup 414
◊ QPicture 299
◊ QPixmap 274, 299, 302
◊ QPlane3D 287
◊ QPluginLoader 446
◊ QPoint 327
◊ QPointer 105
◊ QPropertyAnimation 414
◊ QRay3D 287
◊ QRect 301
◊ QRegExp 347
◊ QResource 52
◊ QRunnable 218
◊ QScopedPointer 106
◊ QScriptEngine 419, 420, 428
◊ QScriptProgram 436
◊ QScriptValue 420
◊ QScrollArea 300
◊ QSemaphore 192
◊ QSequentialAnimationGroup 414
◊ QSharedMemory 189
◊ QSignalMapper 82
◊ QSignalTransition 404
◊ QSizePolicy 36
◊ QSpinBox 35
◊ QSqlDatabase 235
◊ QSqlRelationalTableModel 103, 242, 269
◊ QSqlTableModel 243, 249
◊ QState 404
◊ QStateMachine 404
◊ QStaticText 335
◊ QString 349
◊ QStyle 253
◊ QStyledItemDelegate 251
◊ QStyleOptionViewItemV4 253
◊ QSysInfo 93
◊ QTableView 238
◊ QTextBlock 317
◊ QTextCharFormat 323
◊ QTextCodec 28
◊ QTextDocument 316
◊ QTextDocumentWriter 328
◊ QTextEdit 317
◊ QTextFrame 317
◊ QTextLayout 332
◊ QTextLine 334
◊ QTextList 317
◊ QTextStream 451
◊ QTextTable 317
◊ QTextTableCell 317
◊ QThread 169
◊ QThreadPool 219
◊ QThreadStorage 188
◊ QTranslator 27
◊ QTriangle3D 287
◊ QVariantAnimation 414
◊ QVBox 31
◊ QVBoxLayout 35
◊ QWaitCondition 193
◊ QWidget 365
◊ QXmlStreamReader 353
Кодировка 90
Кольцевой буфер 193
Компоновка визуальных элементов управления 31
Конечный автомат 403
Консольная программа 450
Контейнер 57
Контекстное меню 325
Координаты 343, 358
Критическая область 173

Л

Лицензия 13
Локализация 24

М

Менеджер компоновки 31, 389
Менеджер потоков 220

Метод
◊ accept() 68, 266
◊ addDatabase() 235
◊ addMapping() 263
◊ addPixmap() 356
◊ addState() 406
◊ addTransition() 405
◊ addWidget() 35, 360
◊ checkSyntax() 420
◊ collidingItems() 359
◊ database() 237
◊ deleteLater() 73
◊ dragEnterEvent() 66
◊ dropEvent() 66
◊ event() 64
◊ eventFilter() 73
◊ getDisplayValue() 249
◊ installEventFilter() 73
◊ installTranslator() 28
◊ itemAt() 359
◊ keyPressEvent() 63
◊ map() 226
◊ mapToItem() 358
◊ mapToScene() 358
◊ newFunction() 428
◊ newQObject() 421
◊ notify() 65
◊ postEvent() 69
◊ processEvents() 72
◊ releaseThread() 219
◊ reserveThread() 219
◊ run() 168
◊ sendEvent() 70
◊ setData() 356
◊ setInitialState() 406
◊ setItemDelegate() 266
◊ setRenderHint() 273
◊ tr() 28
◊ trUtf8() 28
◊ unmap() 226
Многопоточность 167
Модель данных 238, 341
Модуль расширения 437
Модульное тестирование 114
Модульность 86
Мьютекс 189

Н

Настройка программы 88

О

Обработчик события 60
Отладчик GDB 125
◊ значение переменной 130
◊ справка 127
◊ стек вызовов 132
◊ точка останова 129
Отображение файлов в память процесса 225
Очередь 205

П

Переход между элементами 54
Платформа, определение 91
Поиск файла 221
Политика изменения размера 36
Поток 72, 167, 218
◊ завершение 170
◊ локальная память 187
◊ передача данных 188
Потоковая безопасность 174
Производительность 123

Р

Растровая графика 299
Регулярные выражения 347
Редактирование партнеров 54
Редактор действий 41
Редактор ресурсов 49
Реентерабельность 173
Репозиторий 145
Рефакторинг 134
◊ Eclipse 138
◊ Visual Assist X 135
◊ выделение метода 135
Роль 239

С

Сборка приложения 57
Связный список
◊ добавление элемента 206
◊ создание 205
◊ удаление элемента 210
Связывание свойств 471
Семафор 192
Сжатие событий 70
Сигнал 75
Синтаксический разбор 346

Система контроля версий 143
Слот 75
Служба 457
Событие 60, 69
Статический анализ 132
Стек вызовов 132
Сценарий 418
◊ обработка сигнала 426
◊ передача объекта 420
◊ создание объекта 429
◊ функция 428

Т

Таблица стилей 55
Таймер 274
Текст 316
Тестирование модульное 114
Точка останова 129
Трехмерная графика 286

У

Удаление объекта 73, 106
Указатель
◊ интеллектуальный 105
◊ размер 89
Утилита
◊ lrelease 27
◊ lupdate 25
◊ Process Monitor 231
◊ qmake 22
◊ qmlviewer 461, 481
◊ qmviewer 476
◊ Qt Linguist 25
◊ rcc 52
◊ sqlite3 241
◊ svnadmin 145
◊ uic 43

Ф

Файл проекта 22
Фактор растяжения 36
Фильтрация событий 73
Форма 262
Формат файла 345
Фрагмент изображения 371
Фрейм 317

Ц

Цвет 465

Ш

Шейдер 280