Глава 4

Базовые элементы управления

4.1. Класс QAbstractButton и производные компоненты

Мы проделали большую работу, рассмотрев один из ключевых классов Qt, `QWidget`. Описание класса `QWidget` было лишено примеров, поскольку он не предназначен для создания экземпляров, а служит только прототипом для всех остальных, в том числе и ваших собственных, элементов управления.

Теперь мы займемся рассмотрением классов, производных от `QWidget`, и в первую очередь обратимся к классу `QAbstractButton`, заменившему класс `QButton` в версии Qt 3. Кнопки в различных видах — очень употребимый и полезный элемент любых прикладных приложений. Даже пример программы `Hello World` состоит из одной кнопки, так что отчасти мы уже видели кнопку в действии.

Класс `QAbstractButton`, как теперь ясно следует из названия, также является абстрактным, поэтому мы рассмотрим его в первую очередь, а затем уж перейдем к его потомкам: классам `QCheckBox`, `QPushButton`, `QRadioButton` и `QToolButton`.

Итак, что отличает кнопки в общем? Кнопка состоит из рамки, текста надписи и иконки (пиктограммы). Главным элементом кнопки является текст надписи — обычно нам встречаются кнопки без рисунков, если не считать инструментальные кнопки на панелях управления.

Для установки текста надписи кнопки служит метод `QAbstractButton::setText(const QString &)`, а для получения этого текста — метод и одноименное свойство `QAbstractButton::text`. Обратите внимание на то, что, в отличие от некоторых других систем, для кнопок в Qt не предполагается надписи по умолчанию, при этом метод `QButton::text()` возвращает значение `QString::null`.

Точно так же, как и надпись, для кнопки может быть установлена пиктограмма, для чего служит метод `setIcon(const QIcon &)`. Этот метод заменил аналогичный `iconSet()` в версии Qt 3: теперь метод читается правильно, как "установить иконку", а не как "набор иконок". Кроме того, Qt 4 не использует возможность установки изображения методом `setPixmap(const QPixmap &)` — для вставки в кнопку изображений вы должны использовать метод `setIcon()`.

Если текст подписи содержит амперсанд "&", то символ после амперсанда автоматически устанавливается в качестве "горячей клавиши", переопределяя ранее заданное значение. Вы можете определить собственный акселератор для кнопки с

помощью метода setShortcut(const QKeySequence &), метод и свойство shortcut возвращают значение последовательности "горячих клавиш" для данной кнопки. Обратите внимание на отличие от Qt 3: там комбинации "горячих клавиш" задавались методом setAccel(), который больше не рекомендуется использовать.

Коды QKeySequence могут быть заданы как числовыми значениями символов, так и строками специального формата, которые могут восприниматься в Qt как допустимые. Например, <Ctrl+G> является корректным сочетанием в качестве комбинации "горячих клавиш".

Кроме визуальных атрибутов, с кнопкой связаны функциональные свойства. Одним из таких свойств является тип кнопки. Различают кнопки с состоянием (toggle button)[1] и без состояния. В терминах Qt 4, в отличие от Qt3, все типы кнопок с состоянием задаются методом setCheckable(); в Qt 3 ту же роль выполнял метод toggleType(). Со свойством checkable связано свойство autoExclusive: оно определяет, будут ли кнопки в группе поддерживать культуру "не больше одной нажатой", как это реализовано в радиокнопках, или же каждая кнопка может быть выбрана независимо, как это "принято" у кнопок-флажков (check box). Естественно, вы можете реализовать то же самое поведение вручную, вызывая в обработчике щелчка на кнопке метод setChecked(false) для остальных. Кстати, узнать, к какой группе относится та или иная кнопка, можно, опросив свойство group.

Главным событием для кнопки является щелчок на ней пользователем. Собственно, существует несколько событий, связанных с этим действием. Событием высокого уровня, обозначающим, что кнопка была "нажата", является сигнал clicked(). Этот "логический" сигнал срабатывает не только от щелчков мыши в области кнопки, но также и при нажатии "горячей клавиши", а также при вызове методов click() и animateClick(). Последний метод анимирует щелчок на кнопке: она будет отрисована в "нажатом", а через указанное время — в "отжатом" состоянии. Такой эффект часто реализуется в демонстрационных целях, когда нужно эмулировать работу пользователя.

Если вы хотите как-то обрабатывать состояние в промежутке между нажатием кнопки и ее освобождением, то для этого служат два низкоуровневых события pressed() и released(). Наконец, при переключении кнопки из одного состояния в другое (если кнопка поддерживает состояния) будет вызван сигнал toggled(). В качестве параметра будет передано новое состояние. Обратите внимание, что когда кнопка находится в пассивном состоянии (disabled), никакие сигналы от нее не поступают.

4.1.1. Классы QCheckBox, QPushButton, QRadioButton, QToolButton

Наконец-то мы добрались до конкретных представителей класса QWidget. Вы можете — и, фактически, будете — создавать реальные экземпляры этих классов в ваших приложениях.

---

[1] Переключатель, т.е. элемент управления интерфейса, имеющий два фиксированных состояния. — *Примеч. ред.*

Начнем с флажков. Задача объекта типа QCheckBox — возвращать значение логического типа: да-нет, включено-выключено и т.д. Флажки являются типичными представителями кнопок с состоянием. Однако обратите внимание: в Qt 4 состояние не представляет собой однозначное логическое значение; на самом деле здесь используется перечислимый тип Qt::ChackState, который предусматривает три значения: Unchecked, PartiallyChecked, Checked. Значение PartiallyChecked ("частично отмечен") применяется в иерархических списках для узла, некоторые (но не все) дочерние узлы которого отмечены. Вероятно, когда-то, когда компьютеры будут широко использовать нечеткую логику, переключатель будет хранить вещественное значение.

Собственно установка и проверка состояния объекта типа QCheckBox производятся методами setCheckState() и checkState() соответственно. Кроме этого, каждое изменение состояния сопровождается сигналом stateChanged().

В общем случае флажки могут находиться в трех положениях: "включено", "выключено" и "состояние не определено". Для переключения двухпозиционного флажка в трехпозиционный достаточно использовать метод QCheckBox::setTristate(). Получить установленное с его помощью значение можно через соответствующее свойство tristate. Обычно флажки не объединяются в группу, а представляют собой независимый набор значений, хотя иногда их "заставляют" работать в качестве радиокнопок.

Самым распространенным, или, по крайней мере, самым *нажимаемым*, типом кнопок являются командные кнопки, представленные в Qt классом QPushButton. Такие кнопки содержат текст или изображение и обычно выглядят как горизонтальный (в MacOS — с закругленными углами) прямоугольник. С точки зрения управления класс QPushButton практически не привносит ничего нового по сравнению с классом QAbstractButton: главным, а часто и единственным интересным событием является clicked(). Специфика кнопок действия состоит в следующем: одна из них может быть кнопкой по умолчанию, т.е. генерировать сигнал активизации во время нажатия клавиши <Enter> на форме. Обычно такое поведение используется в окнах диалога. Для установки/проверки свойства "кнопка по умолчанию" используются методы QPushButton::setDefault() и isDefault() соответственно.

С кнопками действия в Qt связано еще несколько свойств. Прежде всего, Qt расширяет функциональность кнопок действия, позволяя связать с ними выпадающее меню. Меню появляется в момент щелчка на кнопке, так же как это происходит с кнопкой Пуск в Windows. Установить выпадающее меню можно методом setMenu(QMenu *), а получить — через свойство QMenu *menu. Эти методы заменят своих "коллег" из Qt 3 (setPopup(), popup() и isMenuButton()), которые используют устаревший класс QPopupMenu. Вместо последнего метода isMenuButton() в новом коде просто проверяйте условие (!pb.menu()). Вместо слота openPopup() для принудительного вызова меню в Qt 4 нужно вызывать метод showMenu(). При этом не забывайте, что всплывающие меню по историческим причинам (первоначально — для обеспечения производительности) являются модальными: вы не получите управления, пока пользователь так или иначе не закроет выпадающее меню.

Как и любой потомок класса QAbstractButton, командные кнопки можно сделать "залипающими", вызвав метод setCheckable(). Включить режим автоповтора можно с помощью метода setAutoRepeat() — кнопка будет генерировать последовательность нажатий до тех пор, пока пользователь не отпустит ее. Документация Qt 4 не рекомендует использовать эти свойства для обычных командных кнопок, хотя для инструментальных кнопок типа QToolButton это распространенные свойства.

Следующий рассматриваемый нами потомок класса QAbstractButton тоже знаком вам — это радиокнопки, основная задача которых — возвращать одно значение (один вариант) из нескольких. Основная специфика этих кнопок состоит в том, что они всегда объединяются в группы, так что только одна из этих кнопок может быть выделена. На уровне кода это выглядит как установленное по умолчанию свойство autoExclusive.

Поскольку речь идет об устойчивом выделении, то эти кнопки характеризуются состоянием, поэтому свойство checkable тоже всегда установлено по умолчанию. Проверка или установка состояний для QRadioButton-объектов производится парой унаследованных методов isChecked() и setChecked(), хотя в реальной жизни вы чаще всего будете обрабатывать радиокнопки через их контейнер — группу. В остальном радиокнопки не добавляют, по сравнению с классом QAbstractButton, никаких специфических свойств и методов.

Наконец, последний тип предопределенных кнопок — QToolButton, предоставляющий доступ к "быстрым командам", обычно в составе объекта типа QToolBar. Эти кнопки, как правило, не воспринимают фокус ввода, так что щелчок на них не приводит к потере фокуса текущим элементом управления. Также по соглашению эти кнопки не имеют текста надписи, а только пиктограмму для отображения состояний. Инструментальные кнопки часто являются "залипающими", с установленным свойством checkable — например, переключатель наклонного шрифта. Некоторые из них генерируют повторные сигналы при удержании в нажатом состоянии — обычно это касается различных "перемоток" на пультах управления мультимедиа или при просмотре записей баз данных.

В отличие от других кнопок, объект класса QToolBar особым образом отзывается на наведение курсора мыши: при этом эмулируется "приподнятое" состояние кнопки. Установить или отменить этот режим можно через свойство QToolButton::setAutoRaise(). Это свойство автоматически включается, если кнопка помещается на QToolBar-панель.

4.1.2. Немного "практических" кнопок и групп

Вернемся к практике и проиллюстрируем работу различных предопределенных типов кнопок, в том числе и в составе групп. Этот пример вы можете найти среди поставки Qt, он называется Widgets/Group Box. Собственно сам main-код выглядит тривиально — создается один сложный компонент управления, содержащий все остальные кнопки и их группы. На всякий случай приведу полностью текст файла main.cpp.

```
#include <QApplication>
#include "window.h"
int main (int argc, char **argv) {
   QApplication app(argc, argv);
   Window window;
   window.show;
   return app.exec();
}
```

Как видите, создается и отображается один элемент управления, после чего приложение входит в главный цикл асинхронной обработки событий. Вся реальная работа, которой здесь немного, реализована в классе `Window`. Обратите внимание на новый формат директивы `#include` — начиная с версии Qt 4 стандартные модули следует подключать так, как вы видите в первой строке.

Описание этого класса имеет следующий вид.

```
#include <QWidget>
class Window:public QWidget {
Q_OBJECT
public:
   Window(QWidget *parent=0);
private:
   QGroupBox *createFirstExclusiveGroup();
   QGroupBox *createSecondExclusiveGroup();
   QGroupBox *createNonExclusiveGroup();
   QGroupBox *createPushButtonGroup();
}
```

Как видите, описание нашего "большого" класса не отличается изяществом — экспортируемый интерфейс включает только один конструктор, причем только с одним параметром родительского контейнера. Конечно, такой класс не обладает гибкостью и не предназначен для повторного использования. Также имена методов в реальной жизни, вероятно, не будут такими бессмысленно мнемоническими. Как показано выше, при создании своих компонентов в среде Qt не забывайте указывать макрос `Q_OBJECT`.

Мы не будем рассматривать весь текст, поскольку там все достаточно просто, рассмотрим только один характерный фрагмент.

```
#include <QtGui>
#include "window.h"
Window::Window(QWidget *parent):QWidget(parent) {
   QGridLayout *grid = new QGridLayout;
   grid->addWidget(createFirstExclusiveGroup(), 0, 0);
   grid->addWidget(createSecondExclusiveGroup(), 1, 0);
   grid->addWidget(createNonExclusiveGroup(), 0, 1);
   grid->addWidget(createPushButtonGroup(), 1, 1);
   setLayout(grid);
   setWindowTitle(tr("Group Box"));
   resize(480, 320);
}
...
QGroupBox *Window::createSecondExclusiveGroup() {
QGroupBox *groupBox = new QGroupBox(tr("E&xclusive Radio Buttons"));
groupBox->setCheckable(true);
groupBox->setChecked(false);
```

```
QRadioButton *radio1 = new QRadioButton(tr("Rad&io button 1"));
QRadioButton *radio2 = new QRadioButton(tr("Radi&o button 2"));
QRadioButton *radio3 = new QRadioButton(tr("Radio &button 3"));
radio1->setChecked(true);
QCheckBox *checkBox = new QCheckBox(tr("Ind&ependent checkbox"));
checkBox->setChecked(true);
QVBoxLayout *vbox = new QVBoxLayout;
vbox->addWidget(radio1);
vbox->addWidget(radio2);
vbox->addWidget(radio3);
vbox->addWidget(checkBox);
vbox->addStretch(1);
groupBox->setLayout(vbox);
return groupBox;
}
```

Как видите, текст не отличается сложностью — он приведен только для того, чтобы вы могли с чего-то начать свои эксперименты. Есть пару интересных моментов, на которые нужно обратить внимание. Во-первых, повторюсь, это новый формат директивы `#include` — вы указываете модуль, а не файл. Во-вторых, (важно!) обратите внимание на то, что все строковые константы заключены в блок `tr()`. Это указание исполнительной системе Qt получать эти строки из файлов национальных настроек. Сами такие файлы создаются с помощью приложения Qt Linguist, которое рассмотрим в приложении B. Тем не менее если вы даже в отдаленном будущем планируете интернационализацию своей программы, с самого начала заключайте переводимые фразы в блок `tr()`.

Что касается самих кнопок — то тут нет ничего неожиданного. Обратите внимание на то, как формируется экранное представление: сначала создается менеджер размещения, в него добавляются вложенные компоненты, после чего менеджер располагается в контейнере, например, в объекте типа `QGroupBox` или `QWidget`. Другими словами, группа напрямую не является владельцем кнопок, но тем не менее это не мешает группе руководить их поведением, реализуя `autoExclusive`-логику. Для переключателя, находящегося в группе, свойство `autoExclusive` по умолчанию не установлено, поэтому на него не влияет состояние других элементов управления.

И последнее. К группе, как целому, также применимы методы `setCheckable()` и `setChecked()`, что визуально выглядит как кнопка-флажок (check box) в заголовке группы. В результате "выключения" группы все вложенные элементы становятся недоступными (рис. 4.1). Этот сравнительно новый подход реализован в обеих версиях: Qt 3 и Qt 4.

4.2. Ввод и отображение текста: классы QLabel, QLineEdit, QTextEdit

Самый простой и в то же время самый незаменимый элемент, позволяющий отображать фрагмент текста, представляется объектом класса `QLabel`. На самом деле это *не элемент управления*, поскольку вы не управляете с его помощью приложением — скорее, он управляет вами, снабжая полезной информацией. Он не происходит от класса `QWidget`, а как последний, происходит от класса `QFrame`, т.е.

является просто “рамкой”. В результате “текстовая метка” имеет почти те же визуальные свойства, что и `QWidget`-объект, а именно размеры, подсказки по размерам, палитры и т.д. — но не имеет никаких слотов, сигналов и других “оживляющих” свойств. Так что, в отличие от некоторых других “псевдометок” (таких, как в Delphi), которые на самом деле все “видят и чувствуют” и поэтому могут реагировать на ввод пользователя, метки в Qt являются действительно декорациями на празднике жизни. Как следствие — это очень “производительные” (хотя и ничего не производящие) элементы интерфейса, активно используемые как строительный материал для создания более сложных компонентов.

Рис. 4.1. *Так выглядят группы кнопок в Qt. Обратите внимание на флажок, "выключающий" целую группу*

Справедливости ради нужно признать, что у этой простоты `QLabel`-объектов есть двойное дно: метки `QLabel` могут содержать не только строки, но и экземпляры типа `QMovie`, `QPicture`, `QPixmap`. Правда, от этого они не становятся более “разговорчивыми”, но часто именно метка визуально является центральным объектом формы.

Еще одна неочевидная возможность — переадресация фокуса ввода “дружественному” элементу управления (buddy). При нажатии на клавиатуре горячей клавиши, связанной с меткой, она не получает управления, но передает его другому элементу формы, для которого фокус ввода имеет смысл. Это работает только в случае, если метка — текстовая, с установленной для нее “горячей клавишей”, т.е. в строке попросту должен встречаться амперсанд (“&”). При установке другого, нетекстового, значения, свойство `QLabel::buddy` сбрасывается. То же происходит и еще при некоторых операциях, например после вызова метода `QLabel::setNum()`, описанного ниже. Этим механизмом пользуются метки-спутники всех остальных элементов управления — на самом деле текстовая метка всегда является экземпляром класса `QLabel`, вы не можете вывести текст другим способом, не считая рисования.

Существует даже возможность связать метку одного элемента с другим — это может понадобиться в сложных элементах управления, состоящих из нескольких компонентов, хотя того же эффекта можно достичь и более естественным образом, добавив еще одну "независимую" текстовую метку.

Еще один небольшой сервис класса `QLabel` связан с выводом чисел — для этого не нужно пользоваться явным преобразованием, просто воспользуйтесь перегруженным методом `setNum()`, воспринимающим как целые, так вещественные числа. Кстати, Qt сравнивает новое значение метки со старым и перерисовывает метку только в случае, если ее "начертания" в новом и старом виде отличаются.

С переходом на Qt 4 метки класса `Qlabel`, в основном, сохранили свои свойства. Имели место лишь два небольших изменения: был модифицирован метод `setAlignment()`, так что теперь его параметр имеет перечислимый тип `Qt::Alignment`, а не является целым числом. Также был выключен режим `setWrap` по умолчанию для форматированного текста, поскольку обычно от меток не ждут такого поведения.

Более "настоящим" элементом управления, на этот раз — прямым потомком класса `QWidget`, является класс `QLineEdit`. Как следует из названия, это поле редактирования, лишенное таких "радостей", как перенос строк, полосы прокрутки и возможности гипертекста. Зато присутствуют: режимы отмены и "отмены отмены" (undo и redo), функции вырезки, копирования и вставки, а также перетаскивание текста в поле редактирования и из него.

Как принято в различных пользовательских интерфейсах, поля редактирования опционально подлежат проверке: с текстовым полем можно связать проверку ввода через свойство `inputMask` и метод `setValidator()`. Максимальную длину можно ограничить с помощью свойства `maxLength()` (рис. 4.2).

Самым сложным и интересным из перечисленных способов является установка "валидатора". "Валидатор" — это экземпляр класса `QValidator`, а точнее — одного из таких его потомков, как `QIntValidator`, `QDoubleValidator` и `QRegExpValidator`. В последнем случае в регулярное выражение не нужно вставлять символы начала и конца строки (`^` и `$`) — и так ясно, что речь идет об одной строке. Вот как выглядит техника подключения "валидатора" с регулярным выражением:

```
QRegExp rx( "-?\\d{1,3}" );
QValidator* validator = new QRegExpValidator(rx, this);
QLineEdit* edit = new QLineEdit( this );
edit->setValidator(validator);
```

В данном случае регулярное выражение задает необязательный минус и от одной до трех цифр в поле ввода. Пока поле не пройдет "валидацию" — оно не генерирует сигнал `returnPressed()`.

Также в поле ввода можно подавить вывод текста, задав соответствующий режим с помощью метода `echoMode()`, — это полезно для полей ввода паролей или в случае, когда вы хотите "фильтровать" (преобразовывать) вводимые пользователем символы.

***Рис. 4.2.** Типизированный ввод, шаблоны и средства контроля значений позволяют выполнить большинство технических проверок вводимых данных*

Естественно, что вы по-прежнему можете программно "руководить" элементами управления. Текст в поле редактирования можно задать или изменить методами `setText()` или `insert()`. Получить его можно с помощью методов `text()` и `displayText()` — эти значения могут отличаться, если вы подавили эхо, как сказано выше.

Двумя главными сигналами, которые генерирует текстовое поле, являются `textChanged()` и `returnPressed()`. Мы не будем рассматривать все комбинации, например копирование или вырезание, обрабатываемые полем ввода, они, скажем, самые обычные. Заметим только, что элемент текстового ввода может быть создан пустым, с заданным текстом, а также с заданным текстом и маской ввода — для этого существует три перегруженных конструктора.

Наконец, последний рассматриваемый нами класс для ввода текста — очень мощный и порой незаменимый элемент ввода гипертекста, `QTextEdit`. Это настоящий текстовый редактор гипертекста, работающий в режиме WYSIWYG[2] и обрабатывающий данные в HTML-подобном формате. Основными единицами форматирования текста являются параграфы и отдельные символы. Кроме того, в тексте поддерживаются рисунки, списки и таблицы.

[2] WYSIWYG — сокр. от "What You See Is What You Get" — что видишь на экране, то и получишь при печати (известный принцип построения экранного редактора текстов). — *Примеч. ред.*

Будьте бдительны: в Qt режим RichText[3] обозначает собственно формат RTF (Rich Text Format — расширенный текстовый формат), но внутренне он представлен в формате HTML (HyperText Markup Language — язык, используемый для создания страниц WWW). Разметка осуществляется тегами, и они же сохраняются в файле на диске — так что результат сохранения объекта типа `QTextEdit` с любым расширением , в том числе `*.rtf`, всегда будет представлен документом HTML, что в некоторых версиях Linux даже приводит к срабатыванию системы антивирусной безопасности: "тип файла не совпадает с расширением". Так что лучше сохраняйте RichText-текст в `htm`-файлах — по крайней мере вы сможете легко открывать их в браузере.

Программист имеет полный доступ к тексту элемента управления типа `QTextEdit` — причем не только как к целому объекту, но и как к отдельным параграфам и символам. Документ состоит из одного или большего числа параграфов. Символы нумеруются начиная с нуля для каждого параграфа. К параграфам, как объектам, применимы такие операции, как выравнивание. Для символов применимы операции форматирования текста (например, жирное начертание или цветовое выделение). Класс `QTextEdit` поддерживает понятие курсора ввода (текущее положение каретки I-beam[4]) и, как следствие, текущего параграфа и текущего формата, а также текущего фрагмента выделения.

Немного остановимся на жизненном цикле поля ввода гипертекста. Вначале вы можете инициализировать поле ввода любым текстом, вызывая метод `setDocument()`. При этом текст, находящийся до этого в поле ввода, например введенный в приложении Designer, удаляется. Если вы вводите текст с разметкой HTML, то, возможно, считанный впоследствии методом `toHtml()` код может отличаться в конкретных тегах, хотя визуально будет выглядеть так же, как и при вводе.

Дополнительные возможности программной модификации представляют слоты `insertHTML()`, `insertPlainText()`, `append()` и т.д. Текст, добавляемый методом `append()`, не попадает в буфер отката, так что именно этот метод рекомендован для добавления длинных фрагментов.

Аналогично методы `cut()`, `copy()`, `clear()` программно выполняют операции "за пользователя". Фактически эти методы эмулируют соответствующие клавиатурные комбинации, а точнее, они связаны с теми же обработчиками.

В отличие от некоторых других объектных моделей, компонент `QTextEdit` самостоятельно не занимается сохранением данных — для этого используются дополнительные операции. Сам текст извлекается и обновляется через вызовы методов `toHtml()` или `toPlainText()`.

В поле текстового ввода переносы контролируются методом (и свойством) `setWordWrapMode()`. Возможные варианты, кроме тривиальных "по ширине элемента" и "не переносить вообще", включают еще несколько, например "переносить

---

[3] Rich Text означает "обогащенный текст". Имеется в виду текстовая информация, сохраненная в формате, доступном для ее чтения и интерпретации множественными приложениями. — *Примеч. ред.*

[4] I-beam — курсор в форме двутавровой балки, определяющий место вставки вводимых с клавиатуры символов. — *Примеч. ред.*

в любом подходящем месте, не обязательно на границе слова". Ширина зоны принудительного переноса в пикселях или символах задается свойством `lineWrapColumnOrWidth`. Если вы запрещаете перенос или требуете перенос по словам, но слово не помещается, то ситуацию "спасает" горизонтальная полоса прокрутки.

Мы не будем подробно останавливаться на всех методах и свойствах класса `QTextEdit` — тем более что их около сотни (я насчитал одних методов 60, а в Qt 3 их было еще раза в два больше). Пока достаточно представить, что вы имеете дело с некоторым текстовым редактором "средней продвинутости", таким как KWord или MS Wordpad, где доступны различные выделения символов: цвет, жирность, наклон, подчеркивание и различные шрифты. Доступны уже упоминавшиеся операции вырезки и вставки в буфер обмена, отмена и "отмена отмены". Также можно пользоваться операциями поиска по тексту, выделения фрагмента и такими функциями, как увеличение и уменьшение текста (zoom in/out). Все операции доступны программно, а многие — и интерактивно, т.е. с помощью горячих клавиш. Имена методов, как всегда в Qt, являются крайне интуитивными, так что найти нужное — довольно легко.

Важно отметить, что, несмотря на общую преемственность класса `QTextEdit` в версии Qt 4, многие свойства тем не менее были модифицированы. Новый механизм отображения и редактирования текста, Scribe, основан на открытой технологии отрисовки Unicode-шрифтов.

Главное отличие — это представление текста не в виде строки с гипертекстовой разметкой типа `QString`, а в виде документа класса `QTextDocument`. К документу возможен доступ и как к последовательному линейному буферу "в духе" класса `QString`, и как к иерархии объектов в стиле DOM[5]. В качестве элементов документа выступают абзацы, таблицы, списки и другие абстрактные области любой степени вложения. На уровне блоков текст представляется в виде отдельных символов: каждый параметр отображения может быть задан на уровне блока или символа. Блокам соответствуют более общие параметры, такие как выравнивание, цвет фона абзаца или отступы. В отличие от DOM-интерфейса, управление документом не производится напрямую. Вместо этого используется интерфейс класса `QTextCursor`.

Этот интерфейс (экземпляр класса) может быть создан как явно, так и путем получения интерфейса к классу `QTextEdit`:

```
QTextCursor cursor(edit->document());
```

Все операции, которые вы производите с курсором текста, тут же будут отображаться в визуальных представлениях, в частности в объекте типа `QTextEdit`. В любом случае вы, т.е. программа, и пользователь работаете одновременно с одними и теми же структурами, и вам доступны одни и те же операции, такие как перемещение абзаца, изменение формата текста и т.д. Например, вы и пользователь обладаете равными правами по выделению текста и операциям с выделением. Разумеется, посредством переназначения клавиатурного ввода и операций мыши

[5] DOM — сокр. от Document Object Model, т.е. стандарт консорциума WWW, определяющий способы манипулирования объектами и изображениями на одной Web-странице. — *Примеч. ред.*

пользователь сам получает такой же программный интерфейс к визуальному представлению — откуда и следует общность операций. Для формирования динамической "заготовки" в поле ввода служит весьма прямолинейный метод `QTextCursor::insertText()`.

Несколько иное впечатление производят возможности произвольного форматирования класса `QTextCursor` — пример из документации буквально разочаровывает, вместо операции "заполнить траекторию текстом", что доступно во многих системах, предлагается расчет положения и длины каждой строки. Вы, конечно, легко сможете реализовать любые эффекты даже с помощью существующего интерфейса, однако некоторые системы предлагают более высокоуровневые возможности. Главное, однако, что рендеринг шрифтов теперь полностью отделен от операционной системы (или, вернее, сделан системонезависимым), так что вы можете оперировать "глифами" как траекториями так же, как это в свое время было сделано в Windows NT.

4.3. Команды класса QAction, меню и панели инструментов

Главный объект, связанный с меню, горячими клавишами и панелями инструментов, — это "независимые" команды `QAction`. Смысл состоит в следующем. Вы один раз создайте такую команду, как, скажем, открытие файла, связываете (встраиваете) ее по собственному желанию в меню, панель инструментов или привязываете к горячим клавишам. Теперь любое из событий (например, выбор пункта меню или горячей клавиши) будет приводить к выполнению одного и того же кода — раньше в этом не было никакой уверенности.

Это удобно и логично. Однако `QAction`-команды представляют собой нечто большее, чем просто обработчики тех или иных событий, — для этого не нужен объект, достаточно точного указателя на функцию. Экземпляр класса `QAction` хранит всю связанную с командой информацию (например, такую, как пиктограмма, текст, всплывающая подсказка и т.д.), а также текущее состояние (например, такое, как "отмечено" или "недоступно"). Куда бы ни была прикреплена команда — ее подпись, пиктограмма или всплывающая подсказка будут одинаковыми.

Наконец, экземпляры распространяют оповещения об изменении собственного состояния. В результате, если пользователь, скажем, сделает команду недоступной, изменит подпись или пиктограмму, связанную с командой, то все связанные элементы меню и панели инструментов тут же изменят свое состояние согласно новым изменениям.

Существует две возможности создать `QAction`-объект. Во-первых, можно явно создать экземпляр, дополнительно настроить его с помощью таких нужных set-методов, как `setCheckable()` или `setIconText()`, а затем прикрепить его к одному или нескольким контейнерам. Во-вторых (это упрощенный вариант), можно создать команду непосредственно при создании меню, а именно методом `QMenu::addAction()`. В принципе команда может быть прикреплена к любому потомку класса `QWidget` через метод `addAction()`.

Обратите внимание на то, что в прошлой версии 3.3 экземпляр класса QAction сам прикреплялся к контейнеру методом addTo() и откреплялся методом removeFrom(). Эта техника уже не актуальна в версии Qt 4. Современный код должен выглядеть примерно так:

```
fileOpenAction = new QAction( QIcon("open.png"), tr("&Open..."),
↳ this);
fileOpenAction->setShortcut(tr("Ctrl+O"));
fileOpenAction->setActionTip(tr("Open file"));
connect(fileOpenAction, SIGNAL(triggered()),this,SLOT(open()));
/* Qt 3 connect(fileOpenAction,SIGNAL(activated()),this,
SLOT(open())); */
QToolBar *fileTools=new QToolBar(this, "file operations");
fileTools->addAction(fileOpenAction);
/* Qt 3 fileSaveAction->addTo( fileTools ); */
QPopupMenu *file=new QPopupMenu( this );
file->addAction(fileOpenAction);
/* Qt 3 fileSaveAction->addTo( file ); */
```

Важно не забыть привязать с помощью метода connect() активный сигнал команды к слоту-обработчику, поскольку это единственный шанс сделать что-то полезное в ответ на действия пользователя. Также обратите внимание, что в версии 3 QAction-команда генерировала два сигнала: activated() и toggled(). В версии Qt 4 сигнал activated() заменен на triggered() и, кроме того, добавлены сигналы changed() и hovered(). Сигнал activated() оставлен только для совместимости и вызывает сигнал triggered(), так что использовать его нет никакого смысла.

Обращайте внимание на принадлежность экземпляров QAction: вы можете прикрепить команду в качестве дочерней к любому компоненту, но обычно команды имеют глобальную область действия, так что в качестве родительского контейнера используйте главную или подчиненную форму.

Теперь рассмотрим оборотную сторону медали, т.е. те контейнеры, к которым прикрепляются команды. Начнем с панелей управления (или панелей инструментов): они были значительно изменены по сравнению с третьей версией Qt.

Раньше класс QToolBar происходил от QDockWidget, что приводило к их одинаковой обработке и одинаковому поведению. Теперь это два совершенно различных класса, и при негоциации (распределении) рабочей поверхности они обрабатываются по-разному. Грубо говоря, панели инструментов имеют преимущество перед другими компонентами при размещении вдоль рамок главного окна: панели всегда будут парковаться ближе к рамке или к другой панели, причем так, как если бы это была неклиентская область. Экземпляры класса QDocWidget при парковке занимают более удаленные от края зоны в клиентской области.

Еще одно отличие состоит в следующем. Класс QDockArea, как подконтейнер, для размещения панелей управления больше не используется: этот класс вообще больше не используется. Вместо этого само главное окно приложения типа QMainWidget с помощью специального менеджера размещения определяет расположение инструментальных панелей. При размещении элементов типа QDockWidget и QToolBar в главном окне используются достаточно сложные вычисления на уровне

средней школы, хорошо описанные в документации по классу QMainWindow, но мы не будем сейчас на этом особо останавливаться.

В общем, для создания инструментального меню и управления им вам не надо ни о чем особо заботиться: главное окно само отвечает за многие полезные функции панелей инструментов. Вы можете создавать экземпляры класса QAction непосредственно встроенными в панель инструментов с помощью метода addAction(). Несколько перегрузок этого метода позволяют создать сразу кнопку с текстом, пиктограммой и т.д. Метод возвращает только что созданный экземпляр. В качестве родительского контейнера для него будет установлена сама панель, а значит, и время жизни у них будет совпадать — имейте это в виду, поскольку после разрушения панели вы перестанете получать и QAction-команды. Кроме того, как только вы получили указатель на созданный QAction-объект, не откладывая, вызовите необходимый метод connect().

Наконец, пару слов о меню. В отличие от Qt3, где меню строились на основе вспомогательного класса QMenuData, в Qt4 остались только два интерфейсных класса для построения любых типов меню. Класс QMenuBar служит для создания главного и других горизонтальных меню, а класс QMenu подходит для всех типов выпадающих меню, в том числе прикрепленных к главному меню, а также свободных контекстно-зависимых выпадающих меню. Это упрощает многие вещи. Кроме того, как уже было сказано, сейчас не QAction-команда решает, к какому меню прикрепляться, а напротив, само меню располагает методом addAction() для создания команды и автоматического его связывания с пунктом меню.

Класс QMenuBar был модифицирован по сравнению с Qt3, хотя само имя класса осталось без изменений. В результате класс имеет как бы два независимых интерфейса: новый, упрощенный, которым вы должны пользоваться, и старый, служащий целям совместимости, но не рекомендованный к применению. К первой категории относится всего несколько таких мощных методов, как addMenu() и addAction(), а также addSeparator(). Такой простой набор методов помогает разработчику “не сходить с пути прямого” и создавать только стандартные горизонтальные меню.

Выпадающие меню, теперь называемые экземплярами класса QMenu, к всеобщей радости, тоже были значительно упрощены: в прошлом остались многочисленные перегрузки метода addItem(), и в общей сложности около пяти десятков (!) методов и свойств были преданы истории. Главными из оставшихся методов являются все те же addAction() и addMenu() — для создания отдельных команд и вложенных выпадающих меню. Кроме, собственно, команд и подменю, существуют еще и разделители, добавляемые с помощью метода addSeparator().

При выборе пункта меню соответствующая команда получает сигнал QAction::triggered(), точно так же как и при щелчке на кнопке панели инструментов. Обычно вы будете иметь по отдельному обработчику для каждой команды. Если, однако, вы захотите обрабатывать любой щелчок или подсветку любого пункта меню в одной функции, то вам, скорее, придется обрабатывать сигналы QMenu::triggered() и QMenu::hovered().

В целом можно сказать, что в Qt 4 значительно упростилась работа с командами, меню и панелями управления, хотя классы по-прежнему перегружены многими совместимыми методами.

4.4. Классы QAbstractSlider, QAbstractSpinBox и их производные

В практике вычислений часто приходится выбирать значения из некоторого диапазона, а также отображать такие значения. Первичным, по историческим причинам, элементом управления такого типа является полоса прокрутки, сопровождающая многие окна верхнего уровня и области редактирования текста.

Библиотека Qt 4 предоставляет базовую функциональность для такого типа значений в виде класса `QAbstractSlider`, который заменил собой использовавшийся в Qt 3 класс `QRangeControl`. Основной недостаток `QRangeControl` заключался в том, что он не являлся потомком класса `QWidget`, так что производные от него классы должны были использовать двойное наследование. В двойном наследовании нет ничего плохого, но в данном случае абстрактный "диапазон", не имеющий визуального представления, не делал никакой существенной работы.

Основными характеристиками `QAbstractSlider`-экземпляра являются максимальное, минимальное и текущее значения. Для установки и опроса этих значений определены парные методы, например для минимального значения — это `minimum()`/`setMinimum()` (аналогичные пары методов существуют для максимального и текущего значений).

В классе `QAbstractSlider` используются два значения, на которые могут дискретно изменяться значения "большого шага", или *страничного*, соответствующего, например, нажатию клавиши <PgUp>, и малого, или *строчного*, соответствующего, например, нажатию клавиши <↑>. Эти величины хранятся в свойствах `singleStep` и `pageStep` соответственно, а для их установки можно использовать один из set-методов. Вы можете программно прокручивать слайдер (бегунок) на одно "строчное" значение или на одну страницу, а также устанавливать произвольное, находящееся между максимальным и минимальным, значение свойства `value`. Наличие двух величин прокрутки объясняется происхождением всех элементов этой категории от полосы прокрутки окна, которая изначально "понимала" прокрутку на строку и страницу.

Кроме того, есть определенная тонкость в получении текущего значения бегунка. Существует два свойства: `sliderPosition` и `value`. Есть ли между ними разница — зависит от значения свойства `tracking`. Если трекинг-слежение включено (по умолчанию), то при перемещении бегунка пользователем значение `value` сразу будет изменяться в соответствии со значением свойства `sliderPosition`. Иногда имеет смысл изменить этот режим так, чтобы не передавать в программу лишние промежуточные значения.

У самого класса `QAbstractSlider` существует довольно тонкая "нервная система" в виде нескольких сигналов, таких как `rangeChanged()` или `sliderMoved()`. Эти сигналы возникают как реакция на действия пользователя. Интересным

сигналом высокого уровня является actionTriggered(): события возникают при изменении значения бегунка, достижении граничных значений и т.д.

При реализации собственных потомков класса QAbstractSlider вы должны переопределить виртуальный метод sliderChange() — этот метод вызывается при любом значительном событии, начиная от изменения значения и заканчивая таким "потрясением", как QAbstractSlider::SliderOrientationChange. Сами события, как вы уже поняли, описаны константами.

Теперь рассмотрим самый интересный, по моему мнению, представитель семейства бегунков и полос прокрутки — класс QDial. Внешне элемент управления представляет собой нечто среднее между спидометром и ручкой управления стереосистемой. В любом случае это обычный бегунок с радиальной шкалой. Он имеет такие же, унаследованные от класса QAbstractSlider значения maximum и miminun, как и другие бегунки и полосы прокрутки (в версии Qt 3 они назывались maxValue и minValue).

Для класса QDial существует несколько особых свойств, не предусмотренных в его родителе QAbstractSlider. Например, на радиальной шкале рисуются засечки (notches), с которыми связано несколько дополнительных свойств и методов. Кроме того, свойство wrapping позволяет QDial-"ручке" вращаться все время в одном направлении по замкнутому кругу. Графически это отображается замкнутой круговой шкалой.

Из сигналов можно выделить только valueChanged() — этот сигнал генерируется при изменении значения элемента управления, а его дискретность ("аккуратность") зависит от установленного режима setTracking(). Другой унаследованный сигнал, QAbstractSlider::dialMoved(), генерируется даже при отключенном трекинге.

Раз уж мы рассмотрели класс QDial, то не будет преувеличением сказать, что мы рассмотрели и класс QSlider. Фактически при отключенном режиме wrapping эти слайдеры полностью эквивалентны, имеют одинаковые свойства и генерируют одинаковые сигналы. Естественно, что QSlider-объект выглядит совсем по-другому, представляя собой линейный градуированный отрезок с бегунком и делениями, а не круговую шкалу.

Еще один класс происходит от QAbstractSlider — собственно полоса прокрутки QScrollBar. Как правило, это не самостоятельный элемент управления, а одна из составляющих в таких сложных элементах управления, как поле редактирования текста или окно (объект класса QScrollView). Созданная независимо, полоса прокрутки программно не отличается от QSlider- или QDial-объекта.

Обратите внимание на отличие от Qt 3: раньше класс QRangeControl также являлся суперклассом для QSpinBox. Теперь ситуация изменилась — для различных окошек счетчика (spin box)[6] существует отдельный класс QAbstractSpinBox.

---

[6] Элемент обрамления окна, который представляет собой редактируемое текстовое поле, снабженное двумя стрелочными кнопками — "вперед" и "назад", — и отображает родственные, но взаимоисключающие варианты выбора, — например, дни недели или номера записей в словарной базе данных. — *Примеч. ред.*

Базовый класс QAbstractSpinBox является достаточно сложным: он включает однострочное поле редактирования текста, текстовую надпись, рамку, две кнопки приращений значения и еще несколько параметров, например упоминавшийся уже параметр wrapping, реализующий возможность циклического изменения значения.

Безусловно, самым важным "проблематичным" отличием от слайдеров является поле ввода, куда пользователь может попытаться ввести некорректные, в терминах слайдера, значения — например, буквы. Кроме этого, нужно контролировать сами величины вводимых значений, пустой ввод и т.д. По сути, класс QAbstractSpinBox не решает этих вопросов, но создает абстрактные виртуальные методы (например, validate()) и предоставляет возможность потомкам самим заниматься проверкой ввода.

Есть также такое "удобство", как элемент specialValueText, который представляет собой специальную строку (подобную "Auto", "Default" или т.п.), заменяющую минимальное значение. Программа в результате ее использования получит просто минимальное значение, а не специальный текст. Выглядит это так:

```
QSpinBox sb(-1,20,1,this);
mb.setSpecialValueText("Default");
```

От класса QAbstractSpinBox происходит класс QSpinBox — такой же класс был и в Qt 3, но это — полностью переписанный компонент с совершенно другим происхождением. Класс QSpinBox немного усложнен по сравнению со своим абстрактным суперклассом: в поле ввода находятся три текстовых поля — prefix, cleanText и suffix. Для редактирования пользователем доступно только поле cleanText, а поля префикса и суффикса являются декоративными элементами, например в качестве префикса может использоваться текст "$", а в качестве суффикса — "грн.".

В принципе, сам класс QSpinBox поддерживает только числовые значения, однако если вы переопределите в своем подклассе методы validate(), textFromValue() и valueFromText(), то пользователь сможет вводить в поле ввода произвольные строковые значения, например, "раз", "два", "три". При этом значением QSpinBox все равно всегда будет целое число.

Небольшой вариацией на ту же тему является класс QDoubleSpinBox, позволяющий вводить вещественные числа типа double. Естественно, что все параметры (например, minimum, maximum, value и singleStep) имеют тип double. Также обратите внимание на свойство decimals, которое содержит число отображаемых десятичных знаков. Это значение может быть в диапазоне от нуля до четырнадцати знаков после запятой: вы, конечно, должны установить его так, чтобы было видно каждое изменение значения на величину singleStep. Например, если значение singleStep равно 0,01, то decimals должно быть не меньше двух. По умолчанию значение singleStep равно единице, а decimals — двум знакам после запятой.

Наконец, еще одним вариантом на тему SpinBox является класс QDateTimeEdit. Этот новый класс заменяет все классы редактирования даты и времени, существующие в версии Qt 3, хотя старые варианты также сохранены для

совместимости. Редактор QDateTimeEdit представляет собой целую группу окошек счетчика (spin box), разделенных символами форматирования. Для управления максимальными и минимальными значениями служат setMinimumDate и setMaximumDate, и аналогично для времени.

Дата и время представляются в формате displayFormat, например "hh:mm:ss" обозначает привычный формат отображения времени. Формат не только служит внешней маской — установка формата также ограничивает возвращаемые значения типа QDateTime. Так, установка формата "hh:mm:ss" всегда будет возвращать установленную дату, независимо от указанных граничных условий:

```
QDateTimeEdit dte(QDate(2005,6,18));
dte.setDateRange(QDate(2001,1,1),QDate(2003,12,31));
dte.setDisplayFormat("hh:mm");
```

Области формата, такие как "часы" или "минуты", называются в терминах QDateTimeEdit секциями. Каждый тип секции закодирован однобитовым флагом типа QDateTimeEdit::Sections. В результате вы можете вызвать метод currentSection() для определения, в какой области ввода находится пользователь. Метод displayedSections() выводит весь список отображаемых секций в виде битовой OR-маски.

Как легко догадаться, именно ограничением отображаемых секций и занимаются теперь уже "вторичные" классы совместимости QDateEdit и QTimeEdit.

4.5. Страничные отображения с закладками

Страничное отображение, также известное как диалог с закладками, является популярным средством "упаковки" множества элементов управления на ограниченной площади. Qt 4 предлагает вам две возможности для реализации закладок. Инструмент высокого уровня, расположенный в палитре Designer, реализован классом QTabWidget. Если вы не хотите выходить за рамки стандартного поведения, то этого вполне достаточно.

Класс QTabWidget имеет несколько "страниц", представленных закладками. Форма и расположение закладок задаются методами setTabPosition() и setTabShape().

Вы создаете закладку с надписью и расположенными на ней элементами управления (обычно — одним, а именно фреймом, с соответствующим менеджером размещения, который уже служит контейнером для остальных). Добавление новой закладки происходит в одно действие, путем вызова метода addTab(). Для данной страницы можно задать такие параметры, как текст надписи (setTabText), иконка (setTabIcon), режим доступности (setTabEnabled) и т.д. Все эти методы принимают в качестве первого параметра индекс страницы.

Получить текущую страницу можно, вызвав метод currentIndex() (в версии Qt 3 аналогичный метод назывался currentTabIndex()). Соответственно, текущий элемент управления доступен через метод currentWidget(). Обработка элементов управления производится обычным образом, а именно так, как если бы они

были расположены на одной большой форме. Никакой "страничной" специфики, кроме визуального эффекта, не существует.

Если вы не удовлетворены стандартным элементом управления, то Qt 4 предоставляет вам "конструктор" из двух вспомогательных компонентов. Один из них, создаваемый на базе класса `QTabBar`, представляет собой саму линейку, в которой расположены закладки. Для каждой закладки, добавляемой с помощью метода `QTabBar::addTab`, определены текстовая надпись, пиктограмма, режим доступности и т.д. В том числе с закладкой (с помощью метода `setTabData()`) вы можете связать произвольные данные типа `QVariant`. Иногда это бывает полезным, если вы формируете строку закладок в одном месте, а используете в другом. Главным и единственным событием полосы закладок является `currentChanged()`.

Еще одним вспомогательным классом, доступным в палитре Designer и отдельно от класса `QTabWidget`, является класс `QStackedWidget` (соответствует классу `QWidgetStack` в Qt 3). Это класс-контейнер для списка элементов управления. В каждый конкретный момент времени отображается только один элемент из списка (`currentWidget`), расположенный в списке под индексом `currentIndex`. Две миниатюрные кнопки в верхнем правом углу, наблюдаемые в `QTabWidget`-объекте, являются частью `QStackedWidget`-объекта.

Собственно, класс `QStackedWidget` обладает очень простым программным интерфейсом: кроме добавления новых элементов методом `addWidget()` и установки текущего методом `setCurrentIndex()`, вам вряд ли понадобится как-то взаимодействовать с ним. Однако обратите внимание на то, что этот элемент происходит от класса `QFrame`, который в свою очередь является представителем класса `QWidget`, так что их потомок `QStackedWidget` наследует все свойства и методы фреймов.

В заключение вспомним удобный, но очень "очевидный" класс `QTabDialog`, существовавший в Qt3. Этот класс признан морально устаревшим, хотя и сохранен в библиотеке совместимости в виде класса `Q3TabDialog`. В свете последних веяний моды диалоги с закладками рекомендуется строить самостоятельно, располагая `QTabWidget`-объект на `QDialog`-форме (рис. 4.3).

4.6. Стандартные и пользовательские диалоги

Раз уж мы упомянули класс `QDialog`, то заодно рассмотрим и остальные диалоги, связанные с этим классом. Окна диалога — это элементы управления верхнего уровня, которые тем не менее имеют родительский контейнер. Диалоги могут быть как *модальными*, блокирующими работу приложения до возвращения управления, так и *немодальными*, которые могут быть переведены на задний план (и часто там теряются). Обычно диалоги возвращают значения, а некоторые могут быть расширены за счет новых компонентов. Как правило, диалоги имеют фиксированный размер, но иногда могут его изменять (если вызван метод `setSizeGripEnabled(true)`).

Самый общий тип диалогов — модальные, с возвращаемым значением целого типа, которое чаще всего обозначает активную кнопку, вроде Да, Нет, Не знаю. Обычным способом работы с модальными диалогами является вызов метода

exec(), который блокирует работу приложения (визуального потока) до закрытия окна диалога и возвращает значение. Класс QDialog имеет три слота закрытия диалога — accept(), reject() и, более общий, done(), который воспринимает константы Accepted или Rejected. Указанное значение и будет возвращено в приложение.

Рис. 4.3. *Диалоговое окно с закладками создается в среде Qt Designer в одно действие, хотя программно теперь это выглядит как "диалог + закладки"*

Методы возврата управления из диалога могут повлечь за собой нечто большее, чем простое сокрытие окна, что является действием по умолчанию. Поскольку все они вызывают метод QWidget::close(), это может привести даже к разрушению объекта диалогового окна, если оно создано с ключом Qt::WA_DeleteOnClose. Если это главное окно приложения — тогда также завершится все приложение. Если это было последнее закрытое окно из множества окон верхнего уровня, то приложение получит сигнал QApplication::lastWindowClosed(), что тоже обычно вызывает завершение приложения. Обратите внимание на то, что если вам нужны значения, встроенные в диалог (например, такие, как код возврата метода result()), то вы не должны разрушать окно диалога.

В случае немодальных диалогов, которые отображаются независимо от главного окна приложения (как диалоги поиска и замены), они активизируются методом show(), который немедленно возвращает управление (рис. 4.4).

Среди классов кнопок мы уже встречали кнопки по умолчанию — кнопка по умолчанию выделяется более широкой рамкой и срабатывает при нажатии <Enter>. Для задания главной кнопки вы вызываете метод QPushButton::setDefault() — обычно это кнопка OK, но в диалогах вроде "Вы действительно желаете отформатировать весь диск?" это кнопка Cancel. Аналогично клавише <Enter> существует

специальная обработка нажатия клавиши <Esc> — это немедленно вызывает выполнение кода возврата со значением `Rejected`.

Рис. 4.4. *Диалоговые окна штатно поддерживают "потайные" области расширения, где отображаются редко используемые элементы управления*

Существует также такая возможность, как расширенные диалоги. Диалог отображается в краткой форме, но при щелчке, например, на кнопке More диалог расширяется в горизонтальном или вертикальном направлении. Методы `setExtension()`, `setOrientation()` и `showExtension()` как раз и занимаются реализацией этого эффекта. Метод `setExtension()` вызывается в момент, когда окно скрыто, и добавляет к нему элемент управления (фрейм), который и будет расширять данное окно дополнительными компонентами.

В качестве демонстрации окна диалога выберем пример немодального диалога поиска, поскольку немодальные окна вызываются немного сложнее, чем простым вызовом метода `exec()`.

```
void EditorWindow::find() {
   if (!findDialog) {
      findDialog=new FindDialog(this);
      connect(findDialog,SIGNAL(findNext()),this,SLOT(filndNext()));
   }
findDialog->show();
findDialog->raise();
findDialog->activateWindow();
}
```

Как видите, кроме `show()`, надо вызвать еще пару методов, выдвигающих наше окно на передний план и передающих ему управление. Дело в том, что окно после метода `show()` будет отображено в том состоянии, в котором оно было скрыто — а это не всегда передний план, а, скорее, наоборот.

Относительно сигналов — диалоговое окно воспринимается как обычный элемент управления, без каких-либо особенностей. Любой такой контейнер, как окно, диалог или фрейм, должен генерировать содержательные сигналы для оповещения об изменении своего состояния.

Теперь кратко рассмотрим производные окна диалога, которые выполняют распространенные операции, например открытие файлов или настройка принтера.

Их иерархия значительно претерпела изменения по сравнению с Qt 3: классов и уровней стало больше (за счет абстрактных уровней), а методов и свойств — значительно меньше.

Общий подход для всех стандартных окон диалога — это отказ от явного создания экземпляра и вызова метода `exec()`. Вместо этого вызывается статический get-метод, который создает окно диалога одним из двух способов, в зависимости от системы, и поставляет запрошенные данные. Эти статические методы работают поразному под управлением Linux и Windows/Mac OS X. В первом случае будет создан экземпляр `QDialog` нужного типа, который и будет отображен. В случае MS Windows или MacOS X будет вызван стандартный диалог, встроенный в оконную систему. Это редкий случай, когда Qt использует готовые элементы управления (как известно, потомки класса `QWidget` имеют совершенно независимые от операционной системы события, свойства и жизненный цикл).

Таким образом работают все стандартные диалоги, которые поставляют любые настройки: от имени открываемого файла до цвета в формате RGB или настроек шрифта. Большинство параметров диалога задается в конструкторе (в роли "заместителя" которого часто выступают статические методы, о которых шла речь выше) — именно благодаря этому так уменьшилось количество set-методов в Qt 4.

В качестве самого популярного примера рассмотрим диалог на базе класса `QFileDialog` — все остальные похожи на него. Этот диалог служит для выбора существующего файла или каталога, а также для создания имени нового файла, например, при выборе команды `SaveAs`. Главным статическим методом, являющимся "внештатным" конструктором, "назначен" один из get-методов: `getExistingDirectory()`, `getOpenFileName()`, `getOpenFileNames()` или `getSaveFileName()`. Эти методы получают "увесистый" список параметров, определяющих, например, фильтрацию файлов по расширению, начальный каталог поиска, а также строки надписей, появляющиеся в окне диалога.

Вот, например, как можно открыть окно диалога для получения имени одного файла.

```
QString s=QFileDialog::getOpenFileName(
   this,
   "Выберите файл",
   "/home",
   "Изображения (*.gif *.png *.jpg)");
```

Альтернативно можно создать реальный экземпляр класса `QFileDialog`, потомка классов `QDialog` и `QWidget`. В этом случае вы можете делать с окном диалога все, что доступно для отдельного элемента управления:

```
QFileDialog fd=new QFileDialog(this);
fd->setMode(QFileDialog::AnyFile);
fd->setDirectory("/home");
fd->setFilter("Изображения (*.gif *.png *.jpg)");
fd->setViewMode(QFileDialog::Datail);
QStringList files;
if (fd->exec())
   files=fd->selectedFiles(); // в документации здесь опечатка
```

Мы не будем особо углубляться во все настройки, использованные в этом фрагменте, поскольку они очень просты и интуитивны.

Нерассмотренными остались совсем простые уж элементы управления.

Как всегда, мы упустили несколько (около дюжины) второстепенных элементов управления и вспомогательных классов. Из визуально броских это, например, класс `QLCDNumber`, предназначенный для отображения цифр в виде жидкокристаллического дисплея наручных часов. К сожалению (а фактически — ко всеобщей радости), объем этой книги и наша жизнь не позволяют мне отвлекать вас такими второстепенными темами — но, освоив *метод* Qt, вы несомненно легко поймете назначение и принцип работы любого существующего компонента пользовательского интерфейса, а также создадите и новые.

4.7. Оболочка "модель-представление", класс QAbstractItemView и его подклассы

Обратите внимание на факт, который на первый взгляд может показаться незначительным: класс `QListBox` в Qt 4 уже вышел из употребления. Вместо этого современные списки нужно создавать на основе классов `QListWidget` или `QListView`. Также предан забвению и сложный класс `QIconView`, занимавший в Qt 3 целый модуль. Аналогично стал историей и класс `QTable` — вместо него нужно использовать класс `QTableWidget`. Похоже на очередное переименование в целях маркетинга.

На самом деле все значительно сложнее — затронуты самые основы объектной модели. В Qt 4 в корне пересмотрено построение всех таких "групповых" элементов управления, как списки и таблицы, которые могут обрабатывать сотни и тысячи элементов данных, визуально представленных как строки или иконки (пиктограммы), а на уровне программной логики соответствовать файлам, записям базы данных и любым другим сущностям предметной области.

Механизм Qt 4, реализующий такие "большие" элементы управления, называется оболочкой "модель-представление" (model-view framework). В материальном исчислении данная оболочка представлена достаточно полной и гибкой иерархией классов, начиная, как это принято в Qt, с самых общих и абстрактных.

Для начала обратимся к базовому классу отображения всех списков, таблиц и деревьев в Qt 4 — классу `QAbstractItemView`. Конструктив `QAbstractItemView` служит основой для всех классов, в которых обрабатывается множество элементов. Перечислим его предков (в порядке иерархии): `QWidget`, `QFrame`, `QAbstractScrollArea`. Даже и не погружаясь в рассмотрение этих классов, становится ясно, что класс `QAbstractItemView`, как минимум, является фреймом с возможностями включения полос прокрутки.

Основная концепция Qt 4, выраженная в оболочке "модель-представление", как нельзя лучше демонстрируется на примере списков. Сам класс `QAbstractItemView` не хранит отображаемые элементы и не занимается их обработкой — его задача обеспечить потомкам такие визуальные эффекты, как прокрутка изображения,

а также стандартные действия по вводу текста, редактированию, слежению за текущим положением "курсора" (будь то выделенная строка списка или активный элемент древовидного представления), реализуя при этом возможности по выделению одного или нескольких элементов.

Поведение объекта класса `QAbstractItemView` зависит от того, в каком режиме он находится. Сами режимы, описанные `State`-константами (`NoState`, `DraggingState`, `EditingState` и т.д.), определяют, как представление будет реагировать на действия мыши и клавиатурный ввод. Понятно, что при редактировании данных реакция будет иной, чем при перетаскивании. Каждый потомок класса `QAbstractItemView` должен исследовать эти режимы и действовать соответственно.

Например, если пользователь не занят никаким "особым" действием, например редактированием, то он находится в режиме навигации. В этом режиме он перемещается между элементами согласно режиму `CursorAction`. Напротив, другой битовый вектор, `EditTriggers`, являющийся комбинацией флагов `EditTrigger`, указывает, какие действия пользователя приводят к переходу в режим редактирования элемента, начиная от `NoEditTriggers` (нулевой псевдофлаг; используется только отдельно) и до, например, `DoubleClicked` (т.е. для начала редактирования нужно дважды щелкнуть на элементе) или `CurrentChanged` (редактирование начинается сразу после перехода на новый элемент).

Мы не будем детально исследовать класс `QAbstractItemView` — надеюсь, вы проделаете это сами. Здесь только важно определиться, на каком уровне абстракции находится этот класс и за что он отвечает, а именно: за такие общие принципы внешнего интерфейса, как отображение и взаимодействие с пользователем.

Еще один абстрактный класс, `QAbstractItemModel`, является прототипом для построения хранилищ элементов. Точнее, модель не обязательно хранит данные, но всегда знает, где их получить: из собственных запасов, у другого объекта, из файла, базы данных, или их можно сгенерировать на основе формулы, не суть важно. Элементы в общем случае организованы в виде иерархии плоских таблиц — предполагается, что таким образом можно выразить все такие современные схемы представления данных, как списки, таблицы и деревья. Если вы не нуждаетесь в иерархическом представлении, то можете считать, что хранилище имеет вид обыкновенной таблицы или даже списка — это, так сказать, частные, вырожденные случаи.

В качестве указателя на отдельный элемент модели служит экземпляр особого класса — `QModelIndex`, который возвращается в ответ на обращение к методу `QAbstractItemModel::index()` с заданными строкой и колонкой. В этом классе описано все, что нужно для нахождения элемента, вплоть до указателя на модель, в которой он размещен. Характеристики индекса — строка, колонка и родительский индекс, возвращаемые методами `QModelIndex::row()`, `column()` и `parent()` соответственно. Элементы верхнего уровня не имеют родительского узла и поэтому возвращают недействительный элемент, для которого метод `isValid()` возвращает значение `FASLE`. Кроме родительского узла, можно также получить соседний (сестринский) и дочерний, опросив методы `sibling()` и `child()`.

Собственно элемент данных возвращается методом `data()` — это первый метод, который должен переопределить потомок для хранения данных определенного

типа. Хорошая модель также переопределяет метаданные строки или столбца (свойство headerData) и еще, как минимум, количество строк (с помощью метода rowCount()).

Важно: каждая ячейка обычно хранит целый набор данных (величин), каждое из которых сопоставлено с одной из ролей, которые кодируются целыми константами. Вы запрашиваете не только адрес данных, но и нужный тип (роль) возвращаемых данных. Наглядный пример: с ячейкой в электронной таблице могут быть связаны константа, формула, строка форматирования, данные об используемом шрифте, режим выравнивания и тип данных по умолчанию. При расчете таблицы вас интересует формула, а при выводе на экран — значение и, скажем, цвет символов. Можно использовать как предопределенные роли (например, как самую важную DisplayRole), так и собственные.

Итак, класс QAbstractItemModel хранит данные, предоставляя указатель на отдельный элемент с помощью QModelIndex-объекта, а QAbstractItemView-экземпляр отвечает за интерактивный интерфейс модели, которую возвращает его метод model(). В принципе, модель и представление не должны быть сильно зависимы: каждый из них должен воспринимать экземпляр абстрактного типа. Разумеется, некоторые представления специально созданы и лучше подходят для определенных моделей. В дальнейшем мы рассмотрим конкретные реализации этого механизма.

4.7.1. Классы QListView и QListWidget

Начнем с условно самой простой модели и комплиментарного представления — одномерного списка типа QListView. Современный QListView-список, построенный на основе новой технологии model-view, заменил классы QListBox и QIconView из Qt 3.

Собственно, класс QListView может отображать элементы в двух режимах, задаваемых свойством viewMode, — в виде текстового списка и пиктограмм. Другие возможные параметры описываются остальными свойствами. Так, например, свойство flow отвечает за размещение элементов — слева направо или сверху вниз. Свойство movement указывает, могут ли перемещаться отдельные элементы и должны ли они при этом выравниваться по сетке. Значение resizeMode форсирует или запрещает перестроение элементов после изменения размера контейнера и т.д. Большинство остальных свойств и методов (например, iconSize, editTriggers или setTextElideMode()) класс QListView наследует от своего абстрактного предка QAbstractItemView.

Параллельно существуют две “урезанные” до одного измерения модели, основанные на классах QAbstractListModel и QStringListModel. Обе представляют собой реализацию одномерных списков, в последнем случае — списка текстовых строк. Именно с этой моделью чаще всего и работает класс QListView.

На основе класса QListView построен элемент управления еще более высокого уровня — QListWidget. Этот элемент создан для сокрытия всех сложностей явной работы с моделью — модель сокрыта внутри этого класса, и вы общаетесь с ней косвенно, посредством методов класса QListWidget. Для добавления элементов,

имеющих специальный тип QListWidgetItem, служит метод QListWidget::insertItem(). Однако еще проще сразу создавать элементы, принадлежащие данному списку:

```
new QListWidgetItem(tr("Element 1"), listWidget);
```

Посчитать элементы в списке позволяет метод count(), а удалить — метод removeItem(). Текущая строка хранится в свойстве currentRow. В общем, все это напоминает старый класс QListBox, за исключением того, что в основе лежат более сложные и общие механизмы.

4.7.2. Класс QComboBox

Раз уж мы затронули класс QListView, то по ходу упомянем и основанный на нем класс QComboBox. Это очень популярный элемент управления, состоящий из строки ввода текста, а также выпадающего меню, из которого пользователь может выбрать вариант ввода. Для полноты изложения нужно сказать, что, кроме выпадающего списка и текстового поля, класс QComboBox содержит еще два элемента — надпись, представленную элементом Label, и мини-кнопку, при щелчке на которой, собственно, и выпадает список.

В интерфейсе класса QComboBox присутствует настройка всех параметров, соответствующих полю текстового ввода, — по сути, это более всего напоминает поле ввода QLineEdit, поскольку в результате значением поля ввода типа QComboBox всегда является одна строка текста. В частности, вы можете задать шрифт, длину поля ввода, валидатор или даже получить доступ напрямую к дочернему полю ввода через метод QComboBox::lineEdit(). Впрочем, также вы можете получить и дочерний список (фактически — не только список) типа QAbstractItemView через метод view(). Если помните, раньше для выпадающего списка использовался экземпляр класса QListBox, доступный через метод listBox(). Можно даже установить новые объектные значения для этих свойств, хотя вам это понадобится только при создании собственных компонентов.

Обратите внимание на то, что, в отличие от элемента типа QLineEdit, QComboBox-объект возвращает выбранную или введенную пользователем строку методом QComboBox::currentText(), а не QComboBox::text() — последний метод вообще не существует. Если пользователь выбрал один из пунктов меню, то получить его индекс можно методом currentIndex(). Не полагайтесь на постоянство списка и не храните это значение: индекс заданной строки может измениться.

В области свойств, методов и сигналов встроенного списка класс QComboBox мало чем отличается от QListView-списков, при этом все предыдущие замечания по классу QAbstractItemView также остаются в силе.

4.7.3. Классы QTableView и QTableWidget

Совершенно аналогично одномерным классам ведут себя двухмерные: QTableView и QTableWidget. Как легко догадаться, оба эти класса служат для отображения табличных данных. Раньше, в Qt 3, то же поведение было

реализовано с помощью класса QTable, но новые классы имеют совсем другое происхождение.

Если говорить об отличиях от одномерных вариантов, то первое и самое заметное состоит в наличии горизонтальных и вертикальных заголовков, доступных через методы horizontalHeader() и verticalHeader() соответственно. Поскольку заголовки сами по себе являются списками типа QHeaderView (потомками класса QAbstractItemView), можно сказать, что класс QTableView содержит целых три представления: одно собственно табличное и два экземпляра заголовков. Класс QHeaderView (который заменяет бывший ранее в употреблении класс QHeader) используется не только при отображении таблиц, но и при отображении деревьев. К заголовкам можно применять такие привычные действия, как изменение размера (ширины колонок и высоты строк), изменение порядка строк или колонок или задание сортировки (для этого достаточно щелкнуть на нужном заголовке). Встроенные заголовки автоматически связаны сигналами с контейнером, так что вам не нужно заботиться об этом. Например, сигнал QHeaderView::sectionMoved() автоматически связан с таблицей, так что данные сразу же последуют за перемещением "своего" заголовка.

Из других внешних отличий стоит подчеркнуть наличие сетки, разделяющей отдельные ячейки. Тип этой сетки определяется свойством gridStyle. Подобно одномерным спискам, таблица умеет скрывать свои строки и столбцы, а потом снова их отображать (рис. 4.5).

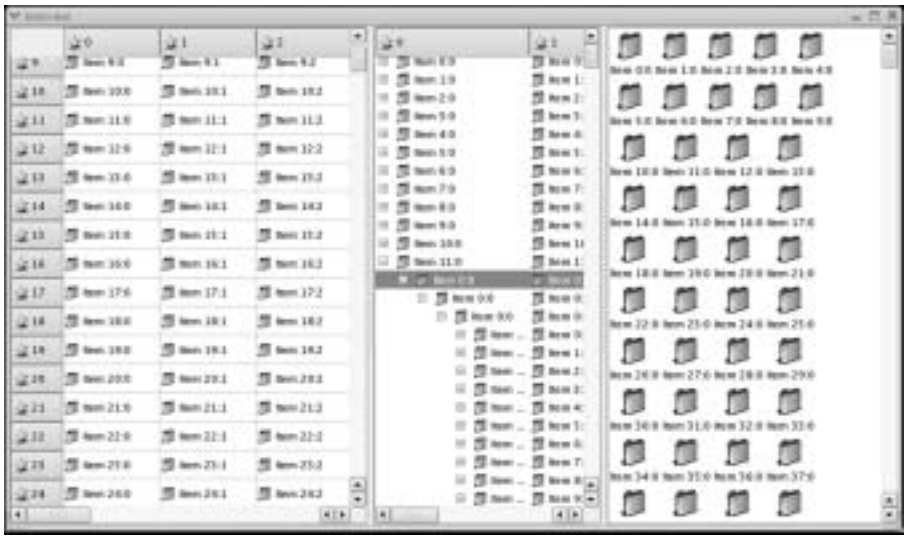

Рис. 4.5. *Табличное представление заменило в Qt 4 класс IconView: теперь самые сложные конструкции реализуются как деревья таблиц, а обычные таблицы и списки являются их частными случаями*

Располагающийся в палитре Designer инструмент типа QTableWidget представляет собой пользовательский элемент управления высокого уровня, упрощающий доступ к табличной модели. Например, метод QTableWidget::setItem()

напрямую устанавливает значение определенной ячейки, а метод `QTableWidget::setHorizontalHeaderItem()`, опять-таки напрямую, явно не обращаясь к экземпляру класса `QHeaderView`, устанавливает заголовок определенного столбца и т.д. Если ваша задача — обычное прикладное приложение, то вы, скорее всего, будете использовать только класс `QTableWidget`.

Так же, как и со списками, существует специальный тип `QTableWidgetItem`, который обычно содержит текст, пиктограмму или флаг выделения. Дополнительно каждый экземпляр класса `QTableWidgetItem` может иметь собственное цветовое и шрифтовое оформление. Дополнительные флаги `flags()` определяют, может ли ячейка быть выбрана или модифицирована.

4.7.4. Классы QTreeView и QTreeWidget

Вот мы и добрались до самых сложных элементов, представленных в Qt-оболочке "модель-представление". Скорее, в древовидных представлениях не так много сложностей, как в древовидных моделях, которые действительно представляют собой иерархические таблицы.

Из визуальных особенностей этого представления сразу можно вспомнить свойство `expandable`, которое указывает, может ли пользователь раскрывать отдельные пункты меню, чтобы показывать или скрывать дочерние элементы. Как правило, древовидные представления не позволяют редактировать элементы данных, но это скорее соглашение, чем правило. Метод `editItem()` начинает редактирование заданного элемента, если он еще раньше не вошел в режим редактирования на основе существующих правил (рис. 4.6).

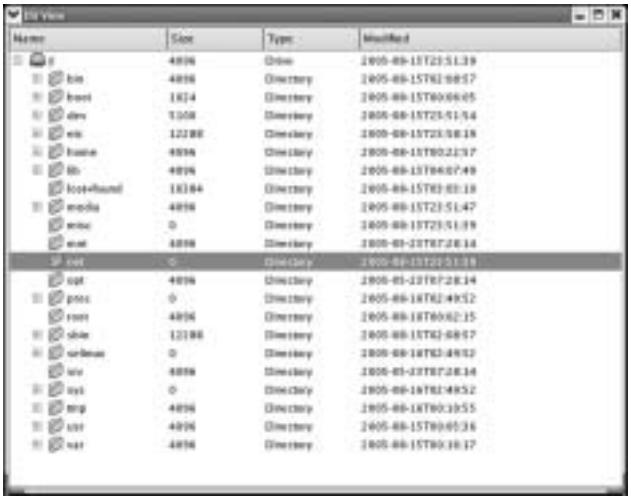

Рис. 4.6. *Реализация древовидного представления немного отличается от привычного: фактически, это дерево и таблица одновременно, т.е. "два в одном"*

Конечно, вы уже догадались, для чего предназначен элемент типа `QTreeWidget`. Действительно, это элемент управления высокого уровня для отображения деревьев. Тип используемых элементов — `QTreeWidgetItem`. С древовидными представлениями связана строка заголовка `QHeaderView`, и это немного отличает `QTreeWidget`-представления от тех, что встречаются в MS Windows. Для того чтобы элементы сразу могли иметь комментарии, пояснения и так далее, сразу после создания `QTreeWidget`-объекта определите количество колонок с помощью метода `setColumnCount()` — предполагается, что их количество будет постоянным на протяжении всей жизни элемента управления (рис. 4.7).

В остальном класс `QTreeWidget` очень похож на `QTableWidget`: он так же позволяет напрямую работать со встроенной в него моделью, строкой заголовка и, разумеется, с параметрами представления (с помощью методов, унаследованных от класса `QAbstractItemView`).

***Рис. 4.7.** В Qt сложно провести черту между древовидным представлением и таблицей*

4.7.5. Список файлов и каталогов типа QDirModel

Модель `QDirView` можно охарактеризовать как “одиноко стоящую”, не имеющую особой привязки к какому-то представлению и в качестве данных получающую список файлов вашего компьютера (а также и соседних, если они “вмонтированы” в вашу файловую систему). Приведем пример гибкости предложенной системы. Во-первых, как уже было сказано, данные являются внешними, получаемыми из операционной системы. Во-вторых, модель вводит большое количество собственных специфических свойств и методов. Например, по заданному индексу `QModeIndex` можно получить данные о файле с помощью следующих методов с такими “красноречивыми” именами: `fileIcon()`, `fileInfo()`, `fileName()` и `filePath()`. Сюда даже встроены такие свойства, как `lazyChildCount` (“ставить на каталогах значок расширения, не проверяя фактического наличия там файлов”) или `resolveSylinks` (“разрешать символические ссылки”).

Модель QDirView хорошо "рифмуется" с древовидным представлением — классический пример на эту тему выглядит так.

```
#include <QtGui>
int main(int argc, char *argv[]) {
   QApplication app(argc, argv);
   QDirModel *model = new QDirModel;
   QTreeView *tree = new QTreeView;
   tree->setModel(model);
   tree->setWindowTitle(QObject::tr("Dir View"));
   tree->resize(640, 480);
   tree->show();
   return app.exec();
}
```

Если вы замените в приведенном выше примере класс QDirView на QListView, то результат также будет вполне удовлетворительным. Это прекрасный пример гибкости и полиморфизма на содержательном уровне.

4.7.6. Последние замечания по теме "модель-представление"

Вы можете вполне успешно писать программы, используя старую технику и готовые классы из категории *Widget. Однако со временем вы обязательно захотите создать собственное представление для существующей модели или даже собственную модель данных. В таком случае вам понадобится дополнительная информация.

Во-первых, обратите внимание на класс QStandardItemModel. Эта модель выступает как хранилище (т.е. данные хранятся в самой модели) для произвольных данных типа QVariant. Если вы не хотите получать данные из более сложных источников, то, возможно, — это все, что вам нужно, например, для реализации электронных таблиц с особыми свойствами. Фактически, этот класс просто реализует интерфейс QAbstractItemModel на минимальном (для создания экземпляра) уровне. В любом более сложном случае, например для получения внешних данных, вы, вероятно, захотите наследовать сам класс QAbstractItemModel и реализовать получение данных собственным путем.

Во-вторых, вам может пригодиться класс QProxyModel. Эта модель не хранит данные и даже не получает их из внешних, по отношению к программе, источников. Вместо этого она получает данные из другой модели, устанавливаемой методом QProxyModel::setModel() и возвращаемой методом model(). Основная задача класса QProxyModel — фильтрация и сортировка получаемых данных перед их отображением в представлении. Вы также, если все сделаете правильно, сможете воспользоваться этим сервисом.

Наконец, иногда вам придется сталкиваться с обработкой нестандартных данных или, напротив, стандартных, но нестандартным образом. Например, вы можете захотеть для ввода целых чисел в таблице использовать экземпляры класса QSlider (рис. 4.8).

В таком случае вы должны заинтересоваться классами-делегатами, происходящими от QAbstractItemDelegate. Эти небольшие объекты как раз и отвечают за

представление информации (предоставляя методы `paint()` и `sizeHint()` для заданного типа данных), а также за создание объектов редактирования по требованию и двусторонний обмен данными "модель-редактор". Вы можете как полностью переписать свой класс, отталкиваясь от класса `QAbstractItemDelegate`, так и (в случае стандартных данных) взять за основу класс `QItemDelegate`.

Рис. 4.8. *Пример встраивания нестандартного элемента управления в табличное представление*

Теперь вы знаете, что собой предоставляет технология "модель-представление", а всю остальную информацию можно найти в разделе документации "Model/View Programming" и в описаниях соответствующих классов.