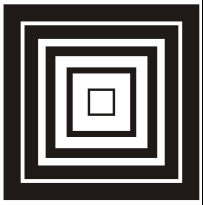
Глава 35



Создание приложений

Приступай к решению любой задачи,
имея в виду решить ее наилучшим образом.

Одна из трех заповедей IBM

Класс главного окна *QMainWindow*

`QMainWindow` — это очень важный класс, который реализует главное окно (рис. 35.1), содержащее в себе типовые виджеты, необходимые большинству приложений, такие как меню *(см. гл. 31)*, секции для панелей инструментов *(см. гл. 34)*, рабочую область, строки состояния *(см. гл. 34)*. В этом классе внешний вид уже подготовлен и его виджеты не нуждаются в дополнительном размещении, так как они уже находятся в нужных местах.

Окно приложения, изображенное на рис. 35.1, имеет рамку, область заголовка для отображения имени и три кнопки, управляющие окном. Кроме этого, окно приложения имеет меню, которое располагается ниже области заголовка окна. Панель инструментов расположена под меню. Под рабочей областью расположена строка состояния.

Указатель на виджет меню можно получить вызовом метода `QMainWindow::menuBar()` и установить в нем нужные всплывающие меню:

```
QMenu* pmnuFile = new QMenu(“&File”);
pmnuFile->addAction(“&Save”);
...
menuBar()->addMenu(pmnuFile);
```

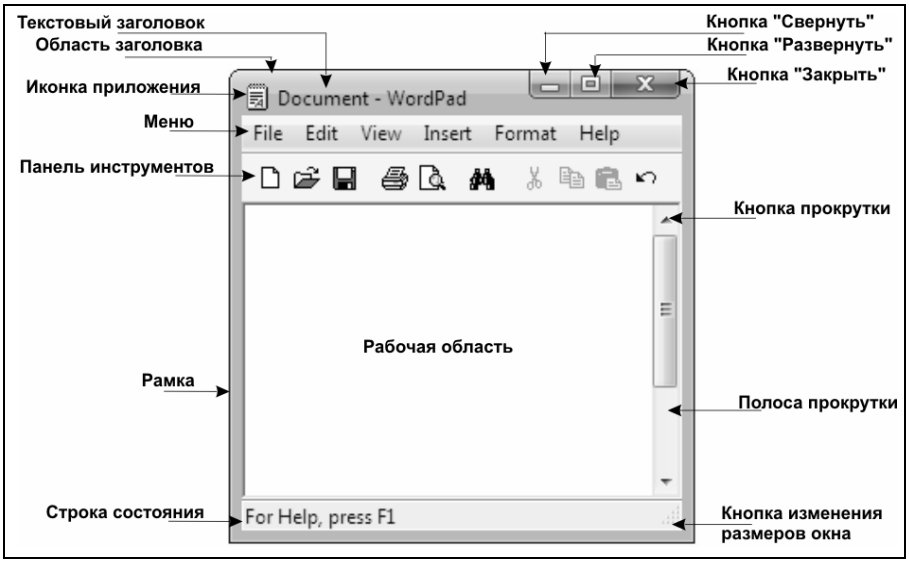


Рис. 35.1. Анатомия главного окна приложения

Как правило, устанавливаются следующие всплывающие меню:

- **File** (Файл) — содержит основные операции для работы с файлами: **New** (Создать) **Open** (Открыть), **Save** (Сохранить), **Print** (Печать) и **Quit** (Выход);
- **Edit** (Правка) — содержит команды общего редактирования: **Cut** (Вырезать), **Copy** (Копировать), **Paste** (Вставить), **Undo** (Отменить), **Redo** (Повторить), **Find** (Найти), **Replace** (Заменить) и **Delete** (Очистить);
- **View** (Вид) — содержит команды, изменяющие представление данных окна. Например, команда **Zoom** (Масштаб) масштабирует отображение документа. В это меню можно включать и те команды, которые управляют отображением элементов интерфейса приложения, например: панели инструментов и строки состояния;
- **Help** (Справка) — необходима для предоставления помощи пользователю, при освоении приложения. А также, обычно, включает в себя информацию об авторских правах приложения. Например, при выборе команды **About** (О программе) появится окно, отображающее имя приложения, его версию, информацию об авторских правах.

Формат: Список

Чтобы получить указатель на рабочую область, следует вызвать метод `QMainWindow::centralWidget()`, который вернет указатель на `QWidget`. Для установки виджета рабочей области потребуется вызвать метод `QMainWindows::setCentralWidget()` и передать в него указатель на этот виджет.

Метод `QMainWindow::statusBar()` возвращает указатель на виджет строки состояния. Кнопка изменения размеров окна, расположенная в нижнем правом углу строки состояния (рис. 35.1), является всего лишь подсказкой для пользователя, сообщающей о том, что размеры главного окна могут быть изменены. Этот виджет реализован в классе `QSizeGrip`. Получить указатель на него из класса главного окна (`QMainWindow`) невозможно, так как он находится под контролем виджета строки состояния.

Предшествующее окно

При запуске многие приложения показывают, так называемое, *предшествующее окно* (Splash Screen). Это окно отображается на время, необходимое для инициализации приложения, и информирует о ходе запуска приложения. Зачастую такое окно используют для маскировки длительного процесса старта программы.

Рис. 35.2. Предшествующее окно

В библиотеке Qt это окно реализовано в классе `QSplashScreen`. Объект этого класса создается в функции `main()` до вызова метода `exec()` объекта приложения. Программа, приведенная в листинге 35.1, отображает перед запуском предшествующее окно, производящее отсчет прогресса инициализации в процентах (рис. 35.2).

Листинг 35.1. Файл main.cpp

```
#include <QtGui>

// ------------------------------------------------------------------------
void loadModules(QSplashScreen* psplash)
{
    QTime time;
    time.start();

    for (int i = 0; i < 100; ) {
        if (time.elapsed() > 40) {
            time.start();
            ++i;
        }

        psplash->showMessage("Loading modules: "
                             + QString::number(i) + "%",
                             Qt::AlignHCenter | Qt::AlignBottom,
                             Qt::black
                            );
    }
}

// ------------------------------------------------------------------------
int main (int argc, char** argv)
{
    QApplication  app(argc, argv);
    QSplashScreen splash(QPixmap("splash.png"));
```

```
    splash.show();

    QLabel lbl("<H1><CENTER>My Application<BR>"
               "Is Ready!</CENTER></H1>"
              );

    loadModules(&splash);

    splash.finish(&lbl);

    lbl.resize(250, 250);
    lbl.show();

    return app.exec();
}
```

В листинге 35.1 объект предшествующего окна создается после объекта приложения. В конструктор предшествующего окна передается растровое изображение, которое будет отображаться после вызова метода `show()`. Виджет `QLabel` представляет, в данном примере, само приложение, которое должно быть запущено. Функция `loadModules()` является эмуляцией загрузки модулей программы, в нее передается адрес объекта предшествующего окна, чтобы функция была в состоянии отображать информацию прогресса загрузки. Объект класса `QTime` *(см. гл. 37)* используется для того, чтобы значение переменной `i` увеличивалось только по истечении 40 миллисекунд. Отображение информации производится при помощи метода `showMessage()`, в который первым параметром передается текст, вторым — расположение текста (см. табл. 7.1), а третьим — цвет текста (см. табл. 17.1). Вызов метода `finish()` производит закрытие предшествующего окна. Если его не закрывать, то оно останется видимым до тех пор, пока пользователь не щелкнет на нем мышью.

Класс действия *QAction*

Действие `QAction` — это очень мощный концепт, ускоряющий разработку приложений. Если бы его не было, то вам нужно было бы создавать команды

меню, соединять их со слотами, затем создавать панели инструментов и соединять их с теми же слотами и т. д. Это могло бы привести к дублированию программного кода и вызвать проблемы при синхронизации элементов пользовательского интерфейса. Например, представьте себе, что вам нужно было бы сделать одну из команд меню недоступной, а для этого вам нужно сделать ее недоступной в меню, а потом проделать то же самое и с кнопкой панели инструментов этой команды.

Объекты класса `QAction` предоставляют решение этой проблемы. Этот концепт значительно упрощает программирование. Например, команда меню **File | New** (Файл | Создать) дублируется кнопкой на панели инструментов, и для них можно создать один объект действия. Тем самым, если вдруг команду потребуется сделать недоступной, мы будем иметь дело только с одним объектом.

Класс `QAction` объединяет следующие элементы интерфейса пользователя:

Формат: Список

- текст для всплывающего меню;
- текст для всплывающей подсказки;
- текст подсказки "Что это";
- "горячие" клавиши;
- ассоциированные пиктограммы;
- шрифт;
- строку состояния.

Для установки каждого из вышеперечисленных элементов в объекте `QAction` существует свой метод. Например:

```
QAction* pactSave = new QAction("file save action", 0);
pactSave->setText("&Save");
pactSave->setShortcut(QKeySequence("CTRL+S"));
pactSave->setToolTip("Save Document");
pactSave->setStatusTip("Save the file to disk");
pactSave->setWhatsThis("Save the file to disk");
pactSave->setIcon(QPixmap(img4_xpm));
connect(pactSave, SIGNAL(triggered()), SLOT(slotSave()));
QMenu* pmnuFile = new QMenu("&File");
pmnu->addAction(pactSave);
QToolBar* ptb = new QToolBar("Linker ToolBar");
ptb->addAction(pactSave);
```

Метод `addAction()` позволяет внести объект действия в нужный виджет. В нашем примере это виджет всплывающего меню `QMenu` (указатель `pmnuFile`) и панель инструментов `QToolBar` (указатель `ptb`).

Создание SDI- и MDI-приложений

Существует два типа приложений, базирующихся на документах. Первый тип — это SDI (Single Document Interface, однодокументный интерфейс), второй — MDI (Multiple Document Interface, многодокументный интерфейс). В SDI-приложениях рабочая область одновременно является окном приложения, а это значит, что невозможно открыть в одном и том же приложении сразу два документа. MDI-приложение предоставляет рабочую область (класса `QWorkSpace`), способную размещать в себе окна виджетов, что дает возможность одновременной работы с большим количеством документов.

SDI-приложение

Типичным примером SDI-приложения является программа ОС Windows Notepad (Блокнот). Пример, приведенный в листинге 35.2, реализует упрощенный вариант этой программы, представляющей собой текстовый редактор. Результат показан на рис. 35.3.

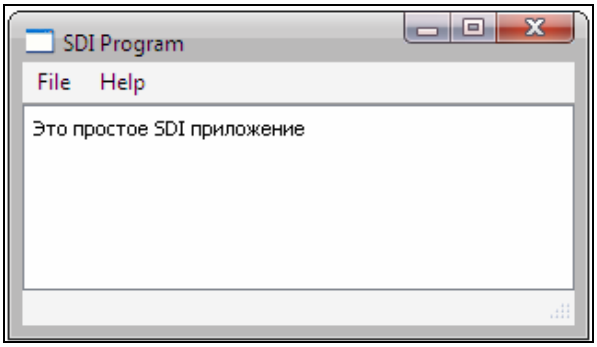


Рис. 35.3. SDI-приложение

Листинг 35.2. Файл DocWindow.h

```
#ifndef _DocWindow_h_
#define _DocWindow_h_

#include <QTextEdit>

// ============================================================================
class DocWindow: public QTextEdit {
Q_OBJECT
private:
    QString m_strFileName;

public:
    DocWindow(QWidget* pwgt = 0);

signals:
    void changeWindowTitle(const QString&);

public slots:
    void slotLoad  ();
    void slotSave  ();
    void slotSaveAs();
};
#endif //_DocWindow_h_
```

Класс `DocWindow`, унаследованный от класса `QTextEdit`, представляет собой окно для редактирования (листинг 35.2). В его определении содержится атрибут `m_strFileName`, в котором хранится имя изменяемого файла. Сигнал `changeWindowTitle()` предназначен для информирования о том, что текстовая область заголовка должна быть изменена. Слоты `slotLoad()`, `slotSave()` и `slotSaveAs()` необходимы для проведения операций чтения и записи файлов.

Листинг 35.3. Конструктор класса `DocWindow`. Файл DocWindow.cpp

```
DocWindow::DocWindow(QWidget* pwgt/*=0*/) : QTextEdit(pwgt)
{
}
```

В конструктор класса, приведенный в листинге 35.3, передается указатель на виджет предка.

Листинг 35.4. Метод `slotLoad()`. Файл DocWindow.cpp

```
void DocWindow::slotLoad()
{
    QString str = QFileDialog::getOpenFileName();
    if (str.isEmpty()) {
        return;
    }

    QFile file(str);
    if (file.open(QIODevice::ReadOnly)) {
        QTextStream stream(&file);
        setPlainText(stream.readAll());
        file.close();

        m_strFileName = str;
        emit changeWindowTitle(m_strFileName);
    }
}
```

Метод `slotLoad()`, приведенный в листинге 35.4, отображает диалоговое окно открытия файла вызовом статического метода `QFileDialog::getOpenFileName()`, с помощью которого пользователь выбирает файл для чтения. В том случае, если пользователь отменит выбор, нажав на кнопку **Cancel** (Отмена), этот метод вернет пустую строку. В нашем примере это проверяется с помощью метода `QString::isEmpty()`. Если метод `getOpenFileName()` возвратит непустую строку, то будет создан объект класса `QFile`, который будет проинициализирован этой строкой. Передача `QIODevice::ReadOnly` в метод `QFile::open()` говорит о том, что файл открывается только для чтения. В случае успешного открытия файла создается объект потока `stream`, который в нашем примере используется для чтения текста из файла. Чтение всего содержимого файла выполняется при помощи метода `QTextStream::readAll()`, который возвращает его в объекте строкового типа `QString`. Текст устанавливается в виджете методом `setPlainText()`. После этого файл закрывается методом `close()`, а местонахождения и имя файла опове-

щаются высылкой сигнала changeWindowTitle() для того, чтобы использующее виджет DocWindow приложение было в состоянии отобразить эту информацию путем изменения заголовка окна.

Листинг 35.5. Метод slotSaveAs(). Файл DocWindow.cpp

```
void DocWindow::slotSaveAs()
{
    QString str = QFileDialog::getSaveFileName(0, m_strFileName);
    if (!str.isEmpty()) {
        m_strFileName = str;
        slotSave();
    }
}
```

В листинге 35.5 слот slotSaveAs() отображает диалоговое окно сохранения файла с помощью статического метода QFileDialog::getSaveFileName(). Если это окно не было отменено пользователем (метод вернул непустую строку), то в атрибут m_strFileName записывается имя файла, указанное пользователем в диалоговом окне, и вызывается слот slotSave().

Листинг 35.6. Метод slotSave(). Файл DocWindow.cpp

```
void DocWindow::slotSave()
{
    if (m_strFileName.isEmpty()) {
        slotSaveAs();
        return;
    }

    QFile file(m_strFileName);
    if (file.open(QIODevice::WriteOnly)) {
        QTextStream(&file) << toPlainText();
        file.close();
        emit changeWindowTitle(m_strFileName);
    }
}
```

Запись в файл представляет собой более серьезный процесс, чем считывание, так как связана с рядом обстоятельств, которые могут сделать ее невозможной. Например, на диске не хватит места или он будет недоступен для записи. Для записи в файл нужно создать объект класса `QFile` и передать в него строку с именем файла. Затем нужно вызвать метод `open()`, передав в него значение `QIODevice::WriteOnly` (флаг, говорящий о том, что будет выполняться запись в файл). В том случае, если файл с таким именем на диске не существует, он будет создан, если существует — то он будет открыт для записи. В случае успешного открытия создается промежуточный объект потока, в который посредством оператора << передается текст виджета с помощью метода `toPlainText()`. После этого файл закрывается методом `QFile::close()` и высылается сигнал с новым именем и местонахождением файла. Это делается для того, чтобы эту информацию могло отобразить приложение, использующее наш виджет `DocWindow`.

Листинг 35.7. Файл SDIProgram.h

```
#ifndef _SDIProgram_h_
#define _SDIProgram_h_

#include <QtGui>
#include "DocWindow.h"
#include "SDIProgram.h"

// ============================================================================
class SDIProgram : public QMainWindow {
Q_OBJECT
public:
    SDIProgram(QWidget* pwgt = 0) : QMainWindow(pwgt)
    {
        QMenu*     pmnuFile = new QMenu("&File");
        QMenu*     pmnuHelp = new QMenu("&Help");
        DocWindow* pdoc     = new DocWindow;

        pmnuFile->addAction("&Open...",
                            pdoc,
                            SLOT(slotLoad()),
```

```
                        QKeySequence("CTRL+O")
                       );
        pmnuFile->addAction("&Save",
                        pdoc,
                        SLOT(slotSave()),
                        QKeySequence("CTRL+S")
                       );
        pmnuFile->addAction("S&ave As...",
                        pdoc,
                        SLOT(slotSaveAs())
                       );
        pmnuFile->addSeparator();
        pmnuFile->addAction("&Quit",
                        qApp,
                        SLOT(quit()),
                        QKeySequence("CTRL+Q")
                       );
        pmnuHelp->addAction("&About",
                        this,
                        SLOT(slotAbout()),
                        Qt::Key_F1
                       );

        menuBar()->addMenu(pmnuFile);
        menuBar()->addMenu(pmnuHelp);

        setCentralWidget(pdoc);
        connect(pdoc,
                SIGNAL(changeWindowTitle(const QString&)),
                SLOT(slotChangeWindowTitle(const QString&))
               );

        statusBar()->showMessage("Ready", 2000);
    }
public slots:
```

```
    void slotAbout()
    {
        QMessageBox::about(this, "Application", "SDI Example");
    }

    void slotChangeWindowTitle(const QString& str)
    {
        setWindowTitle(str);
    }
};
#endif  //_SDIProgram_h_
```

Класс `SDIProgram` унаследован от класса `QMainWindow`. В его конструкторе создаются три виджета — всплывающие меню **File** (Файл) (указатель `pmnuFile`), **Help** (Помощь) (указатель `pmnuHelp`) и виджет созданного нами окна редактирования (указатель `pdoc`). Затем производится с помощью метода `addAction()` неявное создание объектов действий и одновременное их добавление для команд меню. Третьим параметром указывают слот, с которым должна быть соединена команда, во втором параметре указан сам объект, который содержит этот слот. Таким образом, команда **Open...** (Открыть...) соединяется со слотом `slotLoad()`, команда **Save** (Сохранить) — со слотом `slotSave()`, а команда **Save As...** (Сохранить как...) — со слотом `slotSaveAs()`. Все эти слоты реализованы в классе `DocWindow`. Команда **About** (О программе) соединяется со слотом `slotAbout()`, предоставляемым классом `SDIProgram`. Вызов метода `menuBar()` возвращает указатель на виджет меню верхнего уровня, а вызов методов `addMenu()` добавляет созданные всплывающие меню **File** (Файл) и **Help** (Помощь). Вызов метода `setCentralWidget()` делает окно редактирования центральным виджетом, то есть рабочей областью нашей программы. Для изменения текстового заголовка программы, после загрузки файла, сигнал `changeWindowTitle()`, высылаемый виджетом окна редактирования, соединяется со слотом `slotChangeWindowTitle()`. Метод `showMessage()`, вызываемый из виджета строки состояния, отображает надпись "Ready" на время, установленное во втором параметре (в нашем примере оно соответствует двум секундам).

MDI-приложение

MDI-приложение позволяет пользователю работать с несколькими открытыми документами. По своей сути оно очень напоминает обычный рабочий стол, только в виртуальном исполнении. Пользователь может разложить в его области несколько окон документов, или свернуть их. Окна документов могут перекрываться друг другом, а также могут быть развернуты во всю рабочую область.

Рабочая область, внутри которой размещаются окна документов, реализуется классом `QWorkspace` (рис. 35.4). Виджет этого класса производит "закулисное" управление динамически создаваемыми окнами документов. При помощи слотов `title()` и `cascade()`, определенных в этом классе, производится упорядочивание окон. Метод `QWorkspace::windowList()` возвращает список всех содержащихся в нем виджетов.

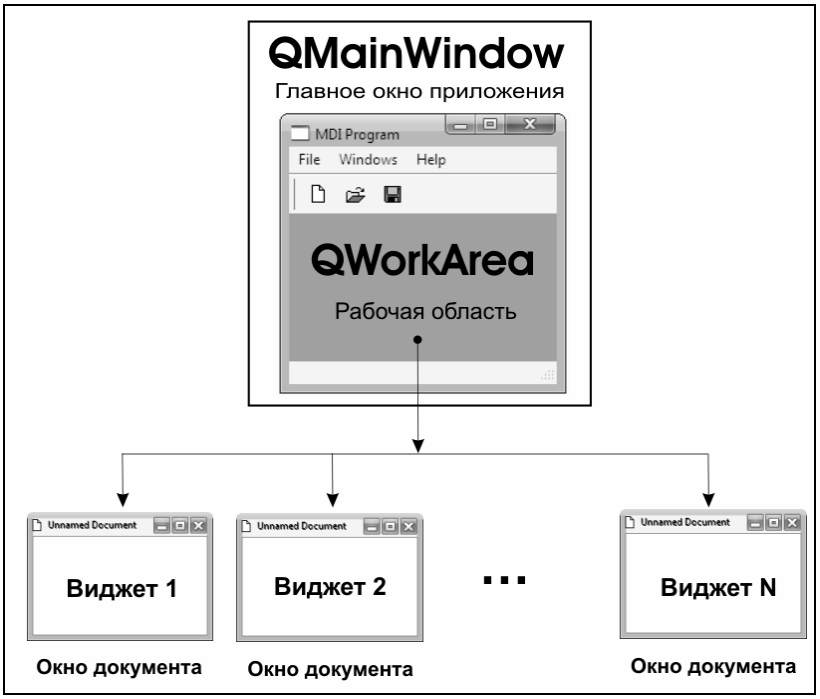


Рис. 35.4. Структура MDI-приложения

Программа, окно которой показано на рис. 35.5, реализует основные функции, присущие MDI-приложению. В качестве класса окна документа применяется класс `DocWindow`, использованный при реализации SDI-приложения.

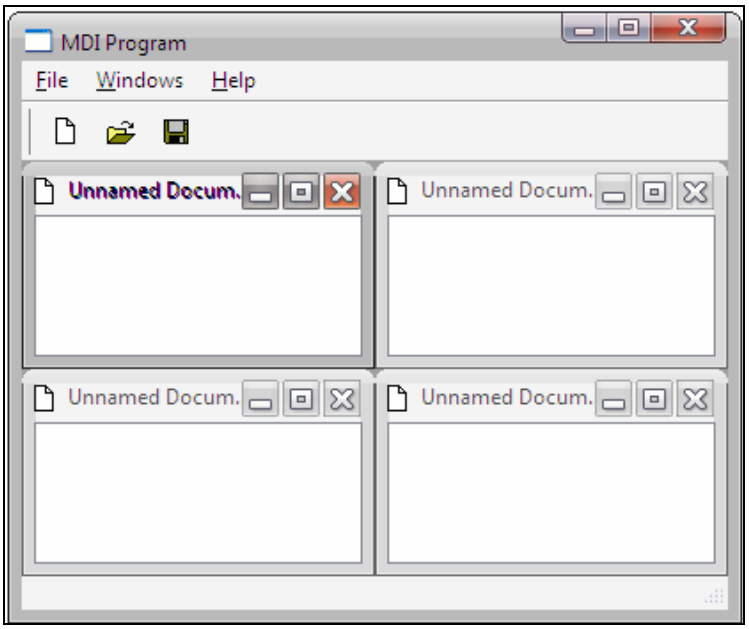


Рис. 35.5. MDI-приложение

Листинг 35.8. Файл MDIProgram.h

```
#ifndef _MDIProgram_h_
#define _MDIProgram_h_

#include <QMainWindow>

class QMenu;
class QWorkspace;
class QSignalMapper;
```

```
class DocWindow;
// ========================================================================
class MDIProgram : public QMainWindow {
    Q_OBJECT
private:
    QWorkspace*    m_pws;
    QMenu*         m_pmnuWindows;
    QSignalMapper* m_psigMapper;

    DocWindow* MDIProgram::createNewDoc();

public:
    MDIProgram(QWidget* pwgt = 0);

private slots:
    void slotChangeWindowTitle(const QString&);

private slots:
    void slotNewDoc ();
    void slotLoad   ();
    void slotSave   ();
    void slotSaveAs ();
    void slotAbout  ();
    void slotWindows();
};
#endif //_MDIProgram_h_
```

Определение класса `MDIProgram`, приведенное в листинге 35.8, содержит атрибуты, хранящие указатели на виджет рабочей области (`m_pws`), на всплывающее меню **Windows** (Окна) (`m_pmnuWindows`) и на сопоставителя сигналов (`m_psigMapper`).

> ***Примечание***
>
> Объекты класса `QAction` в состоянии высылать сигналы `triggered()` только с булевыми значениями. Для нас этого недостаточно, так как мы намерева-

емся высылать указатели виджетов окон редактирования. Поэтому мы и прибегли к использованию класса сопоставления сигналов `QSignalMapper`.

В классе `MDIProgram` определен слот, предназначенный для работы с окнами документов `slotWindows()`, загрузки и сохранения файлов (`slotLoad()`, `slotSave()` и `slotSaveAs()`), создания новых документов (`slotNewDoc()`), а также предоставления информации о самом приложении.

Листинг 35.9. Конструктор `MDIProgram`. Файл MDIProgram.cpp

```
MDIProgram::MDIProgram(QWidget* pwgt/*=0*/) : QMainWindow(pwgt)
{
    QAction* pactNew = new QAction("New File", 0);
    pactNew->setText("&New");
    pactNew->setShortcut(QKeySequence("CTRL+N"));
    pactNew->setToolTip("New Document");
    pactNew->setStatusTip("Create a new file");
    pactNew->setWhatsThis("Create a new file");
    pactNew->setIcon(QPixmap(filenew));
    connect(pactNew, SIGNAL(triggered()), SLOT(slotNewDoc()));

    QAction* pactOpen = new QAction("Open File", 0);
    pactOpen->setText("&Open...");
    pactOpen->setShortcut(QKeySequence("CTRL+O"));
    pactOpen->setToolTip("Open Document");
    pactOpen->setStatusTip("Open an existing file");
    pactOpen->setWhatsThis("Open an existing file");
    pactOpen->setIcon(QPixmap(fileopen));
    connect(pactOpen, SIGNAL(triggered()), SLOT(slotLoad()));

    QAction* pactSave = new QAction("Save File", 0);
    pactSave->setText("&Save");
    pactSave->setShortcut(QKeySequence("CTRL+S"));
    pactSave->setToolTip("Save Document");
    pactSave->setStatusTip("Save the file to disk");
    pactSave->setWhatsThis("Save the file to disk");
```

```
pactSave->setIcon(QPixmap(filesave));
connect(pactSave, SIGNAL(triggered()), SLOT(slotSave()));
QToolBar* ptbFile = new QToolBar("File Operations");
ptbFile->addAction(pactNew);
ptbFile->addAction(pactOpen);
ptbFile->addAction(pactSave);
addToolBar(Qt::TopToolBarArea, ptbFile);

QMenu* pmnuFile = new QMenu("&File");
pmnuFile->addAction(pactNew);
pmnuFile->addAction(pactOpen);
pmnuFile->addAction(pactSave);
pmnuFile->addAction("Save &As...", this, SLOT(slotSaveAs()));
pmnuFile->addSeparator();
pmnuFile->addAction("&Quit",
                    qApp,
                    SLOT(closeAllWindows()),
                    QKeySequence("CTRL+Q")
                   );
menuBar()->addMenu(pmnuFile);

m_pmnuWindows = new QMenu("&Windows");
menuBar()->addMenu(m_pmnuWindows);
connect(m_pmnuWindows, SIGNAL(aboutToShow()), SLOT(slotWindows()));
menuBar()->addSeparator();

QMenu* pmnuHelp = new QMenu("&Help");
pmnuHelp->addAction("&About", this, SLOT(slotAbout()), Qt::Key_F1);
menuBar()->addMenu(pmnuHelp);

m_pws = new QWorkspace;
m_pws->setScrollBarsEnabled(true);
setCentralWidget(m_pws);

m_psigMapper = new QSignalMapper(this);
```

```
    connect(m_psigMapper,
            SIGNAL(mapped(QWidget*)),
            m_pws,
            SLOT(setActiveWindow(QWidget*))
           );

    statusBar()->showMessage("Ready", 3000);
}
```

В конструкторе класса `MDIProgram` создаются три объекта действий для команд создания, открытия и сохранения документов — указатели `pactNew`, `pactOpen` и `pactSave` соответственно (листинг 35.9). Сигнал объектов `triggered()` соединяется со слотами класса `MDIProgram`. Вызовами методов `addAction()` объекты действий добавляются к панели инструментов и в меню.

Команда меню **File | Quit** (Файл | Выход) соединяется со слотом объекта приложения `closeAllWindows()`, который производит закрытие всех окон приложения.

Для создания рабочей области MDI-приложения необходимо создать виджет `QWorkspace`. Чтобы содержимое рабочей области можно было прокручивать, вызывается метод `setScrollBarEnabled()`, в который передается `true`. Установка рабочей области в главном окне виджета производится методом `setCentralWidget()`.

После создания объекта сопоставителя сигналов (указатель `m_psigMapper`), его сигнал `mapped()` соединяется со слотом виджета рабочей области `setActiveWindow()`. Это позволит нам высылать, вместе с сигналами, указатели на виджеты, которые будет обрабатывать слот `setActiveWindow()`.

Метод `showMessage()`, вызываемый из виджета строки состояния, отображает надпись "Ready" в течение трех секунд.

Листинг 35.10. Метод `slotNewDoc()`. Файл MDIProgram.cpp

```
void MDIProgram::slotNewDoc()
{
    createNewDoc()->show();
}
```

Слот `slotNewDoc()`, приведенный в листинге 35.10, создает новое окно документа и делает его видимым.

Листинг 35.11. Метод `createNewDoc()`. Файл MDIProgram.cpp

```
DocWindow* MDIProgram::createNewDoc()
{
    DocWindow* pdoc = new DocWindow;
    m_pws->addWindow(pdoc);
    pdoc->setAttribute(Qt::WA_DeleteOnClose);
    pdoc->setWindowTitle("Unnamed Document");
    pdoc->setWindowIcon(QPixmap(filenew));
    connect(pdoc,
            SIGNAL(changeWindowTitle(const QString&)),
            SLOT(slotChangeWindowTitle(const QString&))
           );

    return pdoc;
}
```

В листинге 35.11, в `createNewDoc()` создается виджет класса `DocWindow` и добавляется вызовом метода `addWindow()` в рабочую область приложения. В метод `setAttribute()` передается значение `Qt::WADeleteOnClose`, сообщающее виджету о том, что он должен быть уничтожен при закрытии своего окна.

Метод `setWindowTitle()` устанавливает заголовок окна. Небольшое растровое изображение в области заголовка устанавливается методом `setWindowIcon()`. Для изменения заголовка окна виджета, а также и окна программы, если виджет развернут, сигнал `changeWindowTitle()` соединяется со слотом `slotChangeWindowTitle()`.

Листинг 35.12. Метод `slotChangeWindowTitle()`. Файл MDIProgram.cpp

```
void MDIProgram::slotChangeWindowTitle(const QString& str)
{
    qobject_cast<DocWindow*>(sender())->setWindowTitle(str);
}
```

Слот `slotChangeWindowTitle()` определен как `private`, и не может быть вызван извне. Поэтому мы проигнорировали проверку успешности приведения к типу `DocWindow` и сразу вызвали, из виджета окна редактирования, метод `setWindowTitle()`, установив в нем имя и местонахождение ассоциированного с ним файла (листинг 35.12).

Листинг 35.13. Метод `slotLoad()`. Файл MDIProgram.cpp

```
void MDIProgram::slotLoad()
{
    DocWindow* pdoc = createNewDoc();
    pdoc->slotLoad();
    pdoc->show();
}
```

Внутри слота `slotLoad()` вызывается метод `createNewDoc()`, который создает новый виджет документа и возвращает его указатель, после чего производится делегирование операции считывания далее, к виджету созданного документа (листинг 35.13).

Листинг 35.14. Метод `slotSave()`. Файл MDIProgram.cpp

```
void MDIProgram::slotSave()
{
    DocWindow* pdoc = qobject_cast<DocWindow*>(m_pws->activeWindow());
    if (pdoc) {
        pdoc->slotSave();
    }
}
```

Слот `slotSave()` получает указатель на текущее окно документа при помощи виджета рабочей области и, в случае если приведение к типу `DocWindow` было успешным, производит делегирование операции сохранения (листинг 35.14).

Листинг 35.15. Метод `slotSaveAs()`. Файл MDIProgram.cpp

```
void MDIProgram::slotSaveAs()
{
```

```
    DocWindow* pdoc = qobject_cast<DocWindow*>(m_pws->activeWindow());
    if (pdoc) {
       pdoc->slotSaveAs();
    }
}
```

Действия слота `slotSaveAs()` аналогичны действиям `slotSave()` (листинг 35.14), только с делегированием метода `slotSaveAs()`(листинг 35.15).

Листинг 35.16. Метод `slotAbout()`. Файл MDIProgram.cpp

```
void MDIProgram::slotAbout()
{
    QMessageBox::about(this, "Application", "MDI Example");
}
```

Слот `slotAbout()` отображает окно сообщения с информацией о приложении (листинг 35.16).

Листинг 35.17. Метод `slotWindows()`. Файл MDIProgram.cpp

```
void MDIProgram::slotWindows()
{
    m_pmnuWindows->clear();

    QAction* pact =
        m_pmnuWindows->addAction("&Cascade", m_pws, SLOT(cascade()));
    pact->setEnabled(!m_pws->windowList().isEmpty());

    pact = m_pmnuWindows->addAction("&Tile", m_pws, SLOT(tile()));
    pact->setEnabled(!m_pws->windowList().isEmpty());

    m_pmnuWindows->addSeparator();

    QList<QWidget*> listDoc = m_pws->windowList();
    m_pmnuWindows->setEnabled(!listDoc.isEmpty());
```

```
    for (int i = 0; i < listDoc.size(); ++i) {
        pact = m_pmnuWindows->addAction(listDoc.at(i)->windowTitle());
        pact->setCheckable(true);
        pact->setChecked(m_pws->activeWindow() == listDoc.at(i));
        connect(pact, SIGNAL(triggered()), m_psigMapper, SLOT(map()));
        m_psigMapper->setMapping(pact, listDoc.at(i));
    }
}
```

Еще одним отличием MDI- от SDI-приложения является наличие всплывающего меню **Windows** (Окна), назначение которого — управление окнами документов, находящимися в рабочей области. Для отображения актуальной информации, перед показом, это меню необходимо очистить методом `clear()` (листинг 35.17). Первыми в меню **Windows** (Окна) добавляются команды меню **Cascade** (Каскад) и **Tile** (Мозаика). В зависимости от наличия в рабочей области окон (опрашивается методом `windowList()`), эти две команды при помощи вызова метода `setEnabled()` становятся доступными или недоступным. В цикле `for` производится размещение команд с названиями окон документов в меню. Каждая команда меню, вызовом метода `setCheckable()` со значением `true`, получает возможность оснащения флажком, который устанавливается только у команды, ассоциирующейся с активным окном. Определение того, является ли окно активным, а также установка статуса активности, достигается сравнением, производимым в методе `setChecked()`.

Далее, сигнал `triggered()` объекта действия (указатель `pact`) соединяется со слотом `map()` сопоставителя сигналов `m_psigMapper`. Благодаря этому, при активации команды меню будет выслан сигнал `mapped()` (см. листинг 35.9), с указателем на виджет окна редактирования, возвращаемый методом `at()`, который устанавливается методом `setMapper()`.

Резюме

Существует два типа приложений — SDI (Single Document Interface, однодокументный интерфейс) и MDI (Multiple Document Interface, многодокументный интерфейс). Главное отличие MDI- от SDI-приложения состоит в том, что SDI-приложение содержит только одно окно документа, а MDI-

приложение способно содержать несколько таких окон, что дает пользователю возможность параллельной работы с несколькими документами.

Класс `QMainWindow` предоставляет уже готовый лейаут, размещающий в себе виджеты, необходимые большинству приложений. В центре размещена рабочая область, которая может содержать только один виджет. При помощи класса `QWorkspace` в этой области можно размещать сразу несколько виджетов, что позволяет реализовывать MDI-приложения. Виджеты находятся в рабочей области, в виде отдельных окон, которые можно перемещать, изменять размеры, сворачивать, разворачивать, упорядочивать их и т. д.

Класс действия `QAction` предоставляет возможность централизации всех элементов интерфейса, связанных с конкретной командой в одном объекте. Это позволяет значительно сократить время разработки программы, а также уменьшить объем исходного кода.