

## Тема - Концепция программирования MVC ("модель-вид-контроллер")

Читать в книге Шлее (QT 5.3): гл. 12

### 1. Общие принципы MVC

При создании сложных приложений существенно увеличивается трудоёмкость разработки кода. Основная концепция идеологии MVC – разделить части программы, отвечающие за хранение и доступ к данным, их отображение и взаимодействие с пользователем на три отдельных модуля, разработка которых относительно независима.

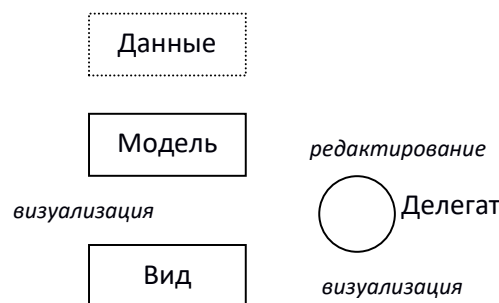
Такой подход:

- увеличивает степень повторного использования кода (например, одна модель может быть применена к нескольким различным между собой представлениям данных);
- облегчает модификацию программы (в идеале можно менять модель, вид или контроллер отдельно);
- становится выгоден при разработке приложений в архитектуре "клиент-сервер", приложений СУБД, использующих хранилища данных, или для перенаправления однопоточного GUI в многопоточные обработчики.

Итак, MVC включает в себя 3 типа объектов:

- **Модель** – осуществляет соединение с источником данных, служит их логической моделью, предоставляет интерфейс другим компонентам архитектуры. Модель представляет знания о моделируемом объекте, но не должна содержать информации о том, как эти знания визуализировать;
- **Вид (представление)** – обеспечивает конечное (например, экранное) представление данных для пользователя. Вид получает из модели модельные индексы, являющиеся ссылками на элементы данных. Сообщая модельные индексы модели, вид может получить элементы из источника данных. Существенно то, что к модели можно применить несколько видов, не изменяя её. Например, одни и те же данные можно показать в виде таблицы, графика или круговой диаграммы;
- **Контроллер** – отвечает за пользовательский интерфейс. Он обеспечивает связь между пользователем и системой: контролирует ввод данных пользователем и использует модель и представление для реализации необходимой реакции.

Существуют различные реализации MVC, во многих из них базовый шаблон MVC модифицируется. Так, в QT применяется следующая разновидность архитектуры "модель-представление":



В стандартных представлениях, уже имеющихся в QT, делегат отображает элементы данных. Если элемент редактируемый, делегат связывается с моделью, непосредственно используя модельные индексы.

И модель, и вид, и делегат представляют собой абстрактные классы, подразумевающие, что мы будем писать для них классы-наследники, предоставляющие полный набор функциональности, ожидаемой другими компонентами.

Модели, представления и делегаты взаимодействуют между собой, используя сигналы и слоты:

- сигналы от модели информируют вид об изменениях данных в источнике данных;
- сигналы от представления предоставляют делегату информацию о пользовательском взаимодействии с отображаемыми пользователю элементами интерфейса;
- сигналы от делегата используются во время редактирования, они сообщают модели и представлению о состоянии редактора.

Все модели элементов основаны на классе `QAbstractItemModel`. Он определяет интерфейс, используемый видами и делегатами для доступа к данным. Сами данные непосредственно в модели не хранятся, но могут содержаться, например, в написанном нами классе-потомке стандартной модели. Также данные могут находиться во внешнем файле, файле ресурсов или ином хранилище, представляемом отдельным классом.

**QAbstractItemModel** удобен для обработки данных, представимых в виде таблиц, списков или деревьев. При создании собственных моделей списков или таблиц можно также использовать использовать классы **QAbstractListModel** или **QAbstractTableModel**.

В QT уже есть несколько готовых моделей для решения типовых задач:

- **QStringListModel** – работает с простым односвязным списком из элементов **QString**;
- **QStandardItemModel** – работает с произвольным деревом элементов, каждый из которых может содержать произвольные данные;
- **QFileSystemModel** – модель для работы с файлами и папками на локальном компьютере;
- **QSqlQueryModel**, **QSqlTableModel**, **QSqlRelationalTableModel** предназначены для доступа к базам данных приложений архитектуры MVC.

QT предоставляет полный функционал следующих стандартных представлений (видов):

- **QListView** – отображение данных в виде списка элементов;
- **QTableView** – отображение данных в виде таблицы;
- **QTreeView** – отображение данных в виде дерева (иерархического списка).

Каждый из этих видов – потомок класса **QAbstractItemView**. В панели Item Widgets вкладки "Дизайн" есть готовые виджеты для применения данных представлений. Правда, они не предназначены для создания подклассов, и не рассчитаны на работу с произвольными моделями.

Существуют и базовые классы для разработки собственных делегатов:

- **QItemDelegate** – предоставляет базовые возможности для редактирования элемента;
- **QStyledItemDelegate** – отличается тем, что позволяет пользовательскую настройку стилей, поэтому, предпочтительней при нестандартных визуализациях редактора.

**Пример 1.** Используем встроенную модель **QFileSystemModel** и покажем её в двух разных видах. Приложение-виджет состоит только из файла `main.cpp`:

**//Попробуйте реализовать данный пример.**

```
#include <QApplication>
#include <QFileSystemModel>
#include <QDir>
#include <QTreeView>
#include <QListView>

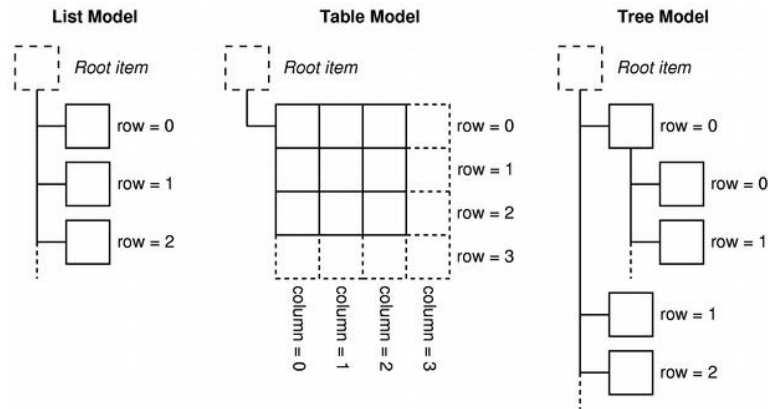
int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    //Готовая модель файловой системы:
    QFileSystemModel *model = new QFileSystemModel;
    model->setRootPath(QDir::currentPath());
    //Показать как дерево, пользуясь готовым видом:
    QTreeView *tree = new QTreeView();
    tree->setModel(model);
    tree->setRootIndex(model->index(QDir::currentPath()));
    //берем данные только через модель!
    tree->show();
    //Показать как список, пользуясь готовым видом:
    QListView *list = new QListView();
    list->setModel(model);
    list->setRootIndex(model->index(QDir::currentPath()));
    list->show();
    //так как не размещаем компоненты, будет в новом окне

    return a.exec();
}
```

В данном случае функция `index` – это функция именно модели **QFileSystemModel**, вообще же модельные индексы у каждого класса моделей свои.

В целом, представления создаются также, как другие виджеты. Настройка представления для отображения элементов модели производится вызовом функции **setModel()** с моделью текущей папки, переданной в качестве аргумента. Мы фильтруем предоставляемые моделью данные вызывая функцию **setRootIndex()** на каждом представлении, передавая соответствующий модельный индекс из файловой системы для текущей папки.

Реально подклассы **QAbstractItemModel** представляют данные в виде иерархической структуры, содержащей таблицы элементов. Представления используют это соглашение для доступа к элементам данных модели, но они не ограничены в способах предоставления информации пользователю.



Понятие *модельного индекса* призвано гарантировать, что представления данных отделены от способа доступа к ним. Представления и делегаты используют эти индексы для запроса отображаемых данных. Только модели требуется знать, как хранятся данные. Модельные индексы содержат указатель на модель, которая их создала.

```
const QAbstractItemModel *model = index.model();
```

Модельные индексы предоставляют временные ссылки на части информации и могут использоваться для получения или изменения данных посредством модели. В связи с тем, что модели могут время реорганизовывать свою структуру, модельные индексы могут стать недействительными и не должны храниться. Если требуется "долгоживущая" ссылка на часть данных, должен быть создан постоянный модельный индекс. Он предоставляет ссылку на данные, которая поддерживается актуальной. Обычные "временные" модельные индексы предоставляются классом **QModelIndex**, а постоянные модельные индексы – классом **QPersistentModelIndex**.

Для получения модельного индекса, соответствующего элементу данных, у модели должны быть заданы три свойства: номер строки, номер столбца и модельный индекс родительского элемента.

```
QModelIndex index = model->index(row, column, ...);
```

Элементы верхнего уровня модели всегда имеют в качестве родителя специфический **QModelIndex()**.

```
QModelIndex indexA = model->index(0, 0, QModelIndex());
```

Элемент, не являющийся корневым, должен знать модельный индекс элемента-родителя:

```
QModelIndex indexA = model->index(0, 0, QModelIndex()); //корневой элемент
```

```
QModelIndex indexB = model->index(1, 0, indexA); //B - дочерний элемент элемента A
```

Элементы модели могут выполнять различные роли для других компонентов, позволяя в различных ситуациях получать различные виды данных.

Например, роль **Qt::DisplayRole** используется для доступа к строке, которая может отображаться в представлении как текст. Как правило, элементы содержат информацию для нескольких различных ролей, а стандартные роли определяются с помощью перечисления **Qt::ItemDataRole**.

Мы можем запросить у модели информацию об элементе, передав ей модельный индекс, соответствующий элементу и задав роль для получения данных желаемого типа:

```
QVariant value = model->data(index, role);
```

По сути, роль указывает модели, данные какого типа будут переданы.

**Пример 2.** Демонстрация получения данных с помощью модельных индексов

```
#include <QApplication>
#include <QtWidgets>
#include <QFileSystemModel>
#include <QDir>
#include <QModelIndex>

int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    QLabel *label1 = new QLabel("");

    QFileSystemModel *model = new QFileSystemModel;
    model->setRootPath(QDir::rootPath());
    QModelIndex parentIndex = model->index(QDir::rootPath());
    int numRows = model->rowCount(parentIndex);
```

```

for (int row = 0; row < numRows; ++row) {
    QModelIndex index = model->index(row, 0, parentIndex);
    QString text = model->data(index, Qt::DisplayRole).toString();
    label1->setText(label1->text()+text+"\n");
}
label1->show();
return a.exec();
}

```

Этот код ещё не показывает изменение данных в модели и не отобразит данные из-за применения виртуальных методов. Здесь мы лишь демонстрируем основные принципы, используемые для восстановления данных из модели:

- Измерения модели могут быть получены с помощью **rowCount()** и **columnCount()**. Эти функции обычно требуют, чтобы был определен родительский модельный индекс.
- Модельные индексы используются для доступа к элементам модели. Для задания элемента требуются номер строки, номер столбца и модельный индекс родительского элемента.
- Для доступа к элементам модели верхнего уровня, в качестве модельного индекса родительского элемента, с помощью **QModelIndex()**, задается нулевой модельный индекс.
- Элементы содержат данные для различных ролей. Для получения данных определенной роли, модели должны быть сообщены модельный индекс и роль.

## 2. Создание новых моделей

Создавая новые модели, использующие преимущества существующих представлений, мы можем представлять данные из разнообразных источников, используя стандартные графические компоненты пользовательского интерфейса, такие как **QListView**, **QTableView** и **QTreeView**.

- Класс **QAbstractItemModel** предоставляет интерфейс, который достаточно гибок для поддержки источников данных, хранящих информацию в иерархических структурах, позволяющих вставлять, удалять и изменять данные, или сортировать их различными способами. Поддерживаются также операции перетаскивания элементов "drag and drop", реализацию которых мы пока оставим "за кадром".
- Классы **QAbstractListModel** и **QAbstractTableModel** предоставляют поддержку интерфейсов для более простых неиерархических структур данных, и более легки для использования в качестве отправной точки для простых моделей списков и таблиц.

**Пример 3. Модель списка строк.** Создадим простую модель списка строк для исследования основных принципов архитектуры "модель-представление".

Важно решить, какой тип модели лучше всего подходит для обеспечения интерфейса к данным. Так как у нас список строк, подойдёт класс **QAbstractListModel**, для которого мы создадим класс-потомок. Сами данные будут храниться в приватном свойстве, имеющем тип **QStringList**. Для минимальной модели необходимы реализации нескольких абстрактных функций, имеющих в классе **QAbstractListModel**:

- конструктор модели - с целью обеспечить стандартное создание объектов класса;
- метод **rowCount()** - определение количества строк в модели;
- метод **data()** - позволяет вернуть или изменить элемент данных, соответствующий определенному модельному индексу;
- метод **headerData()** - служит для получения представлениями деревьев и таблиц сведений, которые будут отображаться в их заголовках;
- метод **flags()** - определяет свойства элементов, в частности, возможность их редактирования;
- метод **setData()** - служит для изменения данных;
- метод **insertRows()** - вставка строк;
- метод **removeRows()** - удаление строк.

Если наша модель иерархическая, мы также должны реализовать функции **index()** и **parent()**. Так как наша модель неиерархическая, мы можем игнорировать модельный индекс, соответствующий родительскому элементу. По умолчанию модели, производные от **QAbstractListModel**, содержат только один столбец, поэтому нам не нужно переопределять функцию **columnCount()**.

Вот интерфейс класса:

```

#ifndef STRINGLISTMODEL_H
#define STRINGLISTMODEL_H

```

```

#include <QAbstractListModel>

class StringListModel : public QAbstractListModel {
    Q_OBJECT

public:
    StringListModel(const QStringList &strings, QObject *parent = 0)
        : QAbstractListModel(parent), stringList(strings) {}

    int rowCount(const QModelIndex &parent = QModelIndex()) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation,
                        int role = Qt::DisplayRole) const;
    Qt::ItemFlags flags(const QModelIndex &index) const;
    bool setData(const QModelIndex &index, const QVariant &value,
                int role = Qt::EditRole);
    bool insertRows(int position, int rows, const QModelIndex &index = QModelIndex());
    bool removeRows(int position, int rows, const QModelIndex &index = QModelIndex());
private:
    QStringList stringList;
};

#endif // STRINGLISTMODEL_H

```

Теперь займёмся реализацией класса в файле stringmodel.cpp. Конструктор нам нужен только для того, чтобы создавать объекты класса стандартным способом, его тело останется пустым и уже встроено в .h файл.

Количество строк в модели у нас равно количеству элементов в списке строк, поэтому соответствующий метод будет очень прост:

```

int StringListModel::rowCount(const QModelIndex &parent) const {
    return stringList.count();
}

```

Функция data() отвечает за возвращение элемента данных, соответствующего запрошенному модельному индексу. Она вернёт непустое значение типа QVariant только если запрошенный модельный индекс валиден и требуемая роль поддерживается:

```

QVariant StringListModel::data(const QModelIndex &index, int role) const {
    if (!index.isValid()) return QVariant();
    if (index.row() >= stringList.size()) return QVariant();
    if (role == Qt::DisplayRole || role == Qt::EditRole)
        return stringList.at(index.row());
    else return QVariant();
}

```

Элементы нашей модели имеют только одну роль, Qt::DisplayRole, так что мы возвращаем данные элемента независимо от указанной роли. Однако, данные, предоставляемые для роли DisplayRole, мы можем повторно использовать в других ролях, таких как ToolTipRole, которую представления могут использовать для отображения информации об элементе во всплывающей подсказке.

Мы можем отдать информацию о заголовках, реализовав в модели функцию headerData():

```

QVariant StringListModel::headerData(int section, Qt::Orientation orientation,
                                     int role) const {
    if (role != Qt::DisplayRole) return QVariant();
    if (orientation == Qt::Horizontal) return QString("Column %1").arg(section);
    else return QString("Row %1").arg(section);
}

```

Конечно, заголовки можно сделать толковей, чем "Column" и "Row".

При редактировании данных элемент списка строк, соответствующий модельному индексу, заменяется на предоставленное значение. Однако прежде, чем мы изменим список строк, мы должны убедиться, что индекс валиден, элемент имеет корректный тип, а роль поддерживается. В соответствии с соглашениями, мы утверждаем, что роль — это Qt::EditRole, используемая стандартными делегатами представлений. После того, как данные установлены, модель должна дать знать представлениям, что некоторые данные изменены. Модель делает это, испуская сигнал dataChanged(). Так как у нас изменился только один элемент данных, указанный в сигнале диапазон элементов данных ограничен одним модельным индексом:

```

bool StringListModel::setData(const QModelIndex &index,

```

```

        const QVariant &value, int role) {
    if (index.isValid() && role == Qt::EditRole) {
        stringList.replace(index.row(), value.toString());
        emit dataChanged(index, index);
        return true;
    }
    return false;
}

```

Важно, что нам не надо знать, как именно делегат выполняет процесс редактирования. Мы лишь предоставляем ему способ занести данные в модель. Однако перед редактированием делегат запрашивает, является ли элемент редактируемым. Ответ должна дать функция `flags()`, в нашем случае она разрешает выбор и редактирование для всех элементов:

```

Qt::ItemFlags StringListModel::flags(const QModelIndex &index) const {
    if (!index.isValid()) return Qt::ItemIsEnabled;
    return QAbstractItemModel::flags(index) | Qt::ItemIsEditable;
}

```

Вставка и удаление строк также легко реализуются в нашей модели переопределением соответствующих функций.

Позиция, куда вставляются строки, обычно зависит от родительского индекса. В нашем случае все элементы-строки относятся к верхнему уровню, поэтому мы просто вставляем пустые строки, не заботясь об их элементах-родителях.

Все вызовы функций с именами на `begin...` и `end...` обязательны при реализации вставки и удаления строк для любой модели.

```

bool StringListModel::insertRows(int position, int rows, const QModelIndex &parent) {
    beginInsertRows(QModelIndex(), position, position+rows-1);
    for (int row = 0; row < rows; ++row) {
        stringList.insert(position, "");
    }
    endInsertRows();
    return true;
}

```

```

bool StringListModel::removeRows(int position, int rows, const QModelIndex &parent) {
    beginRemoveRows(QModelIndex(), position, position+rows-1);
    for (int row = 0; row < rows; ++row) {
        stringList.removeAt(position);
    }
    endRemoveRows();
    return true;
}

```

Основные действия с простой моделью запрограммированы, демонстрация полученного кода может быть, например, такой:

```

#include <QApplication>
#include <QtWidgets>
#include <QAbstractListModel>
#include <QModelIndex>
#include "stringlistmodel.h"

```

```

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QStringList numbers;
    numbers << "One" << "Two" << "Three" << "Four" << "Five";

```

```

    QAbstractItemModel *model = new StringListModel(numbers);
    QListView *view = new QListView;
    view->setModel(model);
    view->show();
    QTableView *tableView = new QTableView;
    tableView->setModel(model);
    tableView->show();
    tableView->setSelectionModel(view->selectionModel()); //модели работают синхронно!
    return app.exec();
}

```



Обработка **выбора** элементов в представлении предоставляется классом `QItemSelectionModel`. Все стандартные представления создают свои собственные модели выбора по умолчанию и взаимодействуют с ними стандартным образом. Модель выбора, используемая представлением, может быть получена с помощью функции `selectionModel()`, а заменить модель выбора можно с помощью функции `setSelectionModel()`. Возможность управлять моделью выбора, используемой представлением, полезна, если вы хотите предоставить множество представлений для одних и тех же данных модели.

### 3. Классы делегатов.

Реализованная в QT архитектура "модель-представление" не включает в себя полностью независимые компоненты для управления взаимодействием с пользователем. Как правило, представление отвечает за представление пользователю данных модели и за обработку пользовательского ввода. Чтобы придать немного гибкости способу, которым этот ввод получается, взаимодействие осуществляется с помощью делегатов (delegates). Эти компоненты предоставляют возможности ввода, а также отвечают за отрисовку индивидуальных элементов в некоторых представлениях. Стандартный интерфейс управления делегатами определен в классе `QAbstractItemDelegate`.

Ожидается, что делегаты способны самостоятельно отрисовывать свое содержимое, реализовав функции `paint()` и `sizeHint()`. Однако, простые, основанные на виджетах, делегаты могут быть созданы как подкласс `QItemDelegate` вместо `QAbstractItemDelegate`, и получить преимущества реализации этих функций по умолчанию.

Редакторы для делегатов могут быть реализованы либо с использованием виджетов для управления процессом редактирования либо непосредственной обработкой событий.

Стандартные представления, поставляемые вместе с Qt, используют экземпляры класса `QItemDelegate` для предоставления средств редактирования. Эта реализация по умолчанию интерфейса делегата отображает элементы в обычном стиле для каждого из стандартных представлений: `QListView`, `QTableView` и `QTreeView`.

Стандартные представления используют делегат по умолчанию, обрабатывающий все стандартные роли. Способ их интерпретации описывается в документации к `QItemDelegate`.

- Функция `itemDelegate()` возвращает делегат, используемый представлением.
- Функция `setItemDelegate()` позволяет вам установить пользовательский делегат для стандартного представления и важно использовать эту функцию когда устанавливается делегат для пользовательского представления.

**Пример 4. Простой делегат.** Этот делегат использует `QSpinBox` (числовое поле ввода со стрелками вверх и вниз) и предназначен для использования с моделями, которые отображают целые числа. Мы создадим представление таблицы для отображения содержимого модели, и оно будет использовать пользовательский делегат для редактирования.

Мы создаем класс делегата как потомка от `QItemDelegate` поскольку не хотим писать функции пользовательского вывода на экран. При этом мы все равно должны предоставить функции для управления виджетом-редактором. Файл `spinboxdelegate.h` будет выглядеть так:

```
#ifndef SPINBOXDELEGATE_H
#define SPINBOXDELEGATE_H

#include <QItemDelegate>
#include <QModelIndex>
#include <QObject>
#include <QSize>
#include <QSpinBox>

class SpinBoxDelegate : public QItemDelegate {
    Q_OBJECT

public:
    SpinBoxDelegate (QObject *parent = 0);

    QWidget *createEditor(QWidget *parent, const QStyleOptionViewItem &option,
        const QModelIndex &index) const;
    void setEditorData(QWidget *editor, const QModelIndex &index) const;
    void setModelData(QWidget *editor, QAbstractItemModel *model,
        const QModelIndex &index) const;
    void updateEditorGeometry(QWidget *editor,
        const QStyleOptionViewItem &option, const QModelIndex &index) const;
};
```

```
#endif // SPINBOXDELEGATE_H
```

При создании делегата виджеты-редакторы не создаются. Мы создаем виджет-редактор только когда это необходимо. Если представление таблицы нуждается в редакторе, оно просит делегата предоставить виджет-редактор, соответствующий модифицируемому элементу. Функцию `createEditor()` снабжаем данными, нужными делегату для установки соответствующего виджета. При этом мы не нуждаемся в хранении указателя на виджет-редактор, так как представление берет на себя ответственность за его уничтожение, когда он станет не нужен. Вот начало файла `spinboxdelegate.cpp`:

```
#include "spinboxdelegate.h"
```

```
SpinBoxDelegate::SpinBoxDelegate(QObject *parent) : QItemDelegate(parent) {}
```

```
QWidget *SpinBoxDelegate::createEditor(QWidget *parent,
    const QStyleOptionViewItem &option, const QModelIndex &index) const {
    QSpinBox *editor = new QSpinBox(parent);
    editor->setMinimum(0);
    editor->setMaximum(100);
    return editor;
}
```

Здесь нижняя и верхняя границы допустимого в `QSpinBox` целого значения произвольно установлены в значения 0 и 100.

Теоретически мы можем создать различные редакторы, зависящие от сообщаемого представлением модельного индекса. Например, если у нас есть столбец целых чисел и столбец строк мы можем вернуть либо `QSpinBox` либо `QLineEdit`, в зависимости от того, какой столбец редактируется.

Делегат должен предоставить функцию копирования данных модели в редактор. В этом примере мы читаем данные, хранящиеся в роли отображения (`display role`), и соответственно устанавливаем значение в окошке счетчика.

```
void SpinBoxDelegate::setEditorData(QWidget *editor, const QModelIndex &index) const {
    int value = index.model()->data(index, Qt::EditRole).toInt();
    QSpinBox *spinBox = static_cast<QSpinBox*>(editor);
    spinBox->setValue(value);
}
```

В этом примере мы знаем, что виджет-редактор — это окошко счетчика, но мы можем предоставлять различные редакторы для разных типов данных в модели, в этом случае мы должны привести виджет к соответствующему типу до обращения к его функциям-членам.

Когда пользователь завершает редактирование значения в окошке счетчика, представление просит делегата сохранить отредактированное значение в модели, вызвав функцию `setModelData()`:

```
void SpinBoxDelegate::setModelData(QWidget *editor, QAbstractItemModel *model,
    const QModelIndex &index) const {
    QSpinBox *spinBox = static_cast<QSpinBox*>(editor);
    spinBox->interpretText();
    int value = spinBox->value();
    model->setData(index, value, Qt::EditRole);
}
```

Так как представление управляет виджетами-редакторами делегата, мы должны только обновить модель с предоставленным редактором содержимым. Мы убеждаемся в том, что окошко счётчика обновилось и обновляем содержащую значение модель, используя заданный индекс.

Стандартный класс `QItemDelegate` сообщает представлению когда он завершил редактирование испуская сигнал `closeEditor()`. Представление проверяет, что виджет-редактор закрыт и уничтожен. В этом примере мы предоставили простейшие средства редактирования, поэтому нам никогда не понадобится испускать этот сигнал.

Все операции над данными выполняются через интерфейс, предоставляемый `QAbstractItemModel`. Это делает делегата в значительной степени независимым от типа данных, с которыми он манипулирует, но должны быть сделаны некоторые допущения в порядке использования определенных типов виджетов-редакторов. В этом примере мы предполагали, что модель всегда содержит целые числа, но мы можем использовать этот делегат с другими видами моделей, поскольку `QVariant` предоставляет удобные значения по умолчанию для непредвиденных данных.

За управление геометрией редактора отвечает делегат. Геометрия должна быть установлена при создании редактора, при изменении размеров или положения элемента в представлении. Представление предоставляет всю необходимую информацию о геометрии внутри объекта.



```
void SpinBoxDelegate::updateEditorGeometry(QWidget *editor, void
SpinBoxDelegate::updateEditorGeometry(QWidget *editor,
    const QStyleOptionViewItem &option, const QModelIndex &index) const {
    editor->setGeometry(option.rect);
}
```

В данном случае мы используем только информацию, предоставляемую опцией представления в прямоугольнике элемента. Делегат, который отображает элементы с несколькими примитивами, не использует прямоугольник элемента непосредственно. Он поместит редактор в зависимости от других примитивов в элементе.

Демонстрация полученного кода может быть такой:

```
#include <QApplication>
#include <QHeaderView>
#include <QItemSelectionModel>
#include <QStandardItemModel>
#include <QTableView>

#include "spinboxdelegate.h"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QStandardItemModel model(4, 2);
    QTableView tableView;
    tableView.setModel(&model);

    SpinBoxDelegate delegate;
    tableView.setItemDelegate(&delegate);

    tableView.horizontalHeader()->setStretchLastSection(true);

    for (int row = 0; row < 4; ++row) {
        for (int column = 0; column < 2; ++column) {
            QModelIndex index = model.index(row, column, QModelIndex());
            model.setData(index, QVariant((row+1) * (column+1)));
        }
    }
    tableView.setWindowTitle(QObject::tr("Spin Box Delegate"));
    tableView.show();
    return app.exec();
}
```

Этот и другие проекты-примеры предполагают, что файл проекта .pro содержит секции, необходимые для сборки приложения с GUI, например, такие:

```
QT      += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = QTree
TEMPLATE = app

HEADERS += \
    spinboxdelegate.h

SOURCES += \
    spinboxdelegate.cpp \
    main.cpp
```

#### 4. Обработка выбора элементов.

Информация о выбранных элементах представления хранится в экземпляре класса QItemSelectionModel. Этот класс содержит свои собственные модельные индексы элементов независимо от представления. В связи с тем, что возможно использование нескольких представлений с одной моделью, можно разделить одну модель выбора между несколькими представлениями, позволив тем самым приложению отобразить одни и те же данные различными способами.

Выбор состоит из выбранных диапазонов. Эти диапазоны компактно хранят информацию о множестве выбранных элементов, запоминая только начальный и конечный модельные индексы для каждого диапазона выбранных элементов. Для описания выбора несмежных элементов используется более одного диапазона выделения.

Выбор применяется к набору модельных индексов, хранящихся в модели выбора. Последний выбор элементов известен как текущее выделение. Свойства этого набора могут быть изменены с помощью некоторых команд управления выбором. Они обсуждаются позже в этом разделе.

В представлении всегда имеется **текущий** элемент и **выделенный** элемент – два независимых состояния. Элемент может быть текущим элементом и выделенным элементом одновременно. Представление отвечает за обеспечение того, чтобы всегда имелся текущий элемент поскольку для перемещения с помощью клавиатуры, например, требуется текущий элемент.

Таблица ниже подчеркивает разницу между текущим элементом и выделенными элементами.

Текущий элемент	Выбранные элементы
Может быть только один текущий элемент.	Может быть много выбранных элементов.
Текущий элемент изменяется при перемещении с помощью клавиатуры или щелчков кнопок мыши.	Состояние выделения элементов устанавливается или снимается, зависящий от нескольких предопределенных режимов – например, выделения единственного элемента, множественного выделения, и т.д. – когда пользователь взаимодействует с элементами.
Текущий элемент будет редактироваться, если нажата клавиша редактирования, F2, или по элементу дважды щелкнули мышью (в том случае, если редактирование разрешено).	Текущий элемент может быть использован вместе с указателем, определяющим диапазон, который будет выбранным или невыбранным (или сочетанием этих двух).
Текущий элемент обозначается прямоугольником фокуса ввода.	Выбранные элементы обозначаются с помощью прямоугольника выделения.

При манипулировании выбором полезно думать о `QItemSelectionModel` как о наборе состояний выбора всех элементов в модели. Как только модель выбора установлена, наборы элементов могут быть выбраны, их выбор может быть отменен или изменен без необходимости знать, какие конкретно элементы выбраны. Индексы всех выбранных элементов могут быть получены в любое время, а с помощью механизма сигналов и слотов другие компоненты могут быть оповещены об изменениях в модели выбора.

Стандартные классы представлений по умолчанию предоставляют **модели выбора**, которые могут использоваться в большинстве приложений. Модель выбора, принадлежащая одному представлению, может быть получена с помощью функции представления `selectionModel()` и разделена между несколькими представлениями с помощью функции `setSelectionModel()`, так что создание новых моделей выбора вообще не требуется.

Выбор создают, задавая модель и пару модельных индексов в `QItemSelection`. Он использует индексы для ссылки на элементы данной модели и интерпретируются как левый верхний и правый нижний элементы в блоке выбранных элементов. Для того, чтобы выбор повлиял на модель выбора, его следует применить к ней; это может быть сделано различными способами, и каждый из них будет иметь различное влияние на уже выбранные элементы модели.

Для демонстрации основных возможностей выбора, мы создаем экземпляр пользовательской модели таблицы, содержащей всего 32 элемента, и для его данных устанавливаем табличное представление:

```
TableModel *model = new TableModel(8, 4, &app);
QTableView *table = new QTableView();
table->setModel(model);
QItemSelectionModel *selectionModel = table->selectionModel();
```

Для дальнейшего использования получаем модель выбора табличного представления по умолчанию. Мы не изменяем в модели никаких элементов, но вместо этого выбираем несколько элементов, которые представление покажет в левом верхнем углу таблицы. Чтобы сделать это, мы должны получить модельные индексы левого верхнего и правого нижнего элементов блока, который должен быть выбран:

```
QModelIndex topLeft;
QModelIndex bottomRight;
topLeft = model->index(0, 0, QModelIndex());
bottomRight = model->index(5, 2, QModelIndex());
```

Чтобы сделать эти элементы в модели выбранными и увидеть соответствующие изменения в табличном представлении, мы должны создать объект выбора и применить его к модели выбора:

```
QItemSelection selection(topLeft, bottomRight);
selectionModel->select(selection, QItemSelectionModel::Select);
```

Выбор применяется к модели выбора с помощью команды, заданной комбинацией флагов выбора. В данном случае, используемые флаги указывают, что объект выбора должен быть представлен в модели независимо от предыдущего состояния выбора. Результирующий выбор отображается представлением.

Выбор элементов может быть изменен с помощью различных операций, определяемых флагами выбора. Выбор, образующийся в результате этих действий, может иметь сложную структуру, но будет эффективно представлен моделью выбора. Использование различных флагов для изменения выбора элементов будет показано при описании обновления выбора.

Модельные индексы, хранящиеся в модели выбора, могут быть **прочитаны** с помощью функции `selectedIndexes()`. Она возвращает несортированный список модельных индексов, который мы можем перебирать не задумываясь о том, к какой модели они относятся:

```
QModelIndexList indexes = selectionModel->selectedIndexes();
QModelIndex index;
foreach(index, indexes) {
    QString text = QString("(%1,%2)").arg(index.row()).arg(index.column());
    model->setData(index, text);
}
```

В вышеприведенном коде Qt для перебора индексов, возвращаемых моделью выбора, и изменения соответствующих им элементов используется удобная конструкция `foreach`.

Модель выбора испускает сигналы, сообщающие об изменениях в выборе. Они уведомляют другие компоненты как об изменениях модели выбора в целом, так и об изменениях элемента модели, имеющего в данный момент фокус. Мы можем соединить сигнал `selectionChanged()` со слотом и исследовать вновь выбранные элементы или элементы, которые при изменении выбора сделаны невыбранными. Слот вызывается с двумя объектами `QItemSelection`: один содержит список индексов, соответствующих выбранным элементам; другой содержит список индексов, соответствующих элементам, выбор которых был отменен.

В следующем коде мы предоставим слот, который получает сигнал `selectionChanged()`, заполняет строку выделенными элементами и очищает содержимое элементов, выбор которых отменен.

```
void MainWindow::updateSelection(const QItemSelection &selected,
    const QItemSelection &deselected) {
    QModelIndex index;
    QModelIndexList items = selected.indexes();
    foreach (index, items) {
        QString text = QString("(%1,%2)").arg(index.row()).arg(index.column());
        model->setData(index, text);
    }
    items = deselected.indexes();
    foreach (index, items)
        model->setData(index, "");
}
```

Мы можем отследить изменение фокуса элементов внутри модели, соединив сигнал `currentChanged()` со слотом, вызываемым с двумя модельными индексами. Эти индексы соответствуют элементу, имевшему фокус раньше, и элементу, имеющему фокус в данный момент.

В следующем коде мы предоставляем слот, получающий сигнал `currentChanged()` и использующий информацию, им предоставляемую, для обновления панели состояния `QMainWindow`:

```
void MainWindow::changeCurrent(const QModelIndex &current,
    const QModelIndex &previous) {
    statusBar()->showMessage(
        tr("Moved from (%1,%2) to (%3,%4)")
        .arg(previous.row()).arg(previous.column())
        .arg(current.row()).arg(current.column()));
}
```

С помощью этих сигналов контролируется выбор, сделанный пользователем, но мы можем изменять модель выбора непосредственно.

Команды выбора специфицируются комбинацией флагов выбора, определенной как `QItemSelectionModel::SelectionFlag`. Каждый из флагов выбора указывает модели выбора, как изменять внутренний набор уже выбранных элементов при каждом вызове функции `select()`. Наиболее часто используется флаг `Select`, который указывает модели выбора запомнить выбранные элементы именно так, как они были выбраны. Флаг `Toggle` указывает модели выбора инвертировать состояние выбора всех указанных элементов, т.е. невыбранные элементы

сделать выбранными, а выбранные сделать невыбранными. Флаг Deselect делает все указанные элементы невыбранными.

Отдельные элементы модели выбора **обновляются** с помощью создания выбора элементов и применения его к модели выбора. В следующем коде мы применяем второй выбор элементов к показанной выше табличной модели, используя команду Toggle для инвертирования состояния выбора указанных элементов.

```
QItemSelection toggleSelection;
topLeft = model->index(2, 1, QModelIndex());
bottomRight = model->index(7, 3, QModelIndex());
toggleSelection.select(topLeft, bottomRight);
selectionModel->select(toggleSelection, QItemSelectionModel::Toggle);
```

По умолчанию команды выбора работают только с элементами заданными модельными индексами. Однако флаг, используемый для описания команды выбора, может использоваться совместно с дополнительными флагами для изменения строк и столбцов. Например, если вы вызываете select() только с одним индексом, но с командой, которая является комбинацией флагов Select и Rows, то будет выбрана строка, содержащая элемент, на который ссылается модельный индекс. Следующий код показывает использование флагов Rows и Columns:

```
QItemSelection columnSelection;
topLeft = model->index(0, 1, QModelIndex());
bottomRight = model->index(0, 2, QModelIndex());
columnSelection.select(topLeft, bottomRight);
selectionModel->select(columnSelection,
    QItemSelectionModel::Select | QItemSelectionModel::Columns);
QItemSelection rowSelection;
topLeft = model->index(0, 0, QModelIndex());
bottomRight = model->index(1, 0, QModelIndex());
rowSelection.select(topLeft, bottomRight);
selectionModel->select(rowSelection,
    QItemSelectionModel::Select | QItemSelectionModel::Rows);
```

Для **замены** текущего выбора новым, комбинируйте флаги выбора с флагом Current. Команда, использующая этот флаг, указывает, что модель выбора должна заменить свой текущий набор модельных индексов на тот, что указан при вызове select().

Для **удаления** прежнего выбора прежде, чем вы сделаете новый, используйте другие флаги выбора совместно с флагом Clear. Это приведет к переустановке набора модельных индексов модели выбора.

Для **выбора всех элементов** модели, необходимо создать выбор для каждого из уровней модели, охватывающий все элементы этого уровня. Мы делаем это, восстанавливая индексы соответствующие левому верхнему и правому нижнему элементам заданного родительского индекса:

```
QModelIndex topLeft = model->index(0, 0, parent);
QModelIndex bottomRight = model->index(model->rowCount(parent)-1,
    model->columnCount(parent)-1, parent);
```

Выбор создается с этими индексами и моделью. При этом будут выбраны соответствующие элементы модели выбора:

```
QItemSelection selection(topLeft, bottomRight);
selectionModel->select(selection, QItemSelectionModel::Select);
```

Это должно быть сделано для всех уровней модели. Родительский индекс для элементов верхнего уровня мы определяем обычным образом:

```
QModelIndex parent = QModelIndex();
```

В иерархических моделях для определения, является ли заданный элемент родительским для элементов более низкого уровня используется функция hasChildren().

## 5. Готовые классы представления элементов.

В Qt входят стандартные виджеты, реализующие классические, основанные на элементах контейнеры. Основанные на элементах виджеты имеют имена, отражающие их назначение: QListWidget предоставляет список элементов, QTreeWidget отображает многоуровневую древовидную структуру, QTableWidget предоставляет таблицу с ячейками. Каждый из этих классов наследует поведение класса QAbstractItemView, который реализует общее поведение выбора элементов и управления заголовками.

**Одноуровневые списки элементов** обычно отображаются с помощью QListWidget и множества QListWidgetItem. Виджет-список создается тем же образом, что и любой другой виджет:

```
QListWidget *listWidget = new QListWidget(this);
```

Элементы списка могут быть добавлены в виджет-список непосредственно при создании:

```
new QListWidgetItem(tr("Sycamore"), listWidget);
new QListWidgetItem(tr("Chestnut"), listWidget);
new QListWidgetItem(tr("Mahogany"), listWidget);
```

Также они могут быть созданы без родительского виджета-списка и добавлены в список позже:

```
QListWidgetItem *newItem = new QListWidgetItem;
newItem->setText(itemText);
listWidget->insertItem(row, newItem);
```

Каждый элемент списка может отображать текст и пиктограмму. При настройке внешнего вида элемента можно изменить цвет и шрифт, используемые для отрисовки текста. Всплывающие подсказки, текст панели состояния и подсказки "What's This?" легко настраиваются для того, чтобы гарантировать, что список органично впишется в приложение.

```
newItem->setToolTip(tooltipText);
newItem->setStatusTip(tooltipText);
newItem->setWhatsThis(whatsThisText);
```

По умолчанию элементы списка отображаются в порядке их добавления в список. Списки элементов могут быть отсортированы по алфавиту или в обратном порядке в соответствии с критерием, указанным в `Qt::SortOrder`:

```
listWidget->sortItems(Qt::AscendingOrder);
listWidget->sortItems(Qt::DescendingOrder);
```

**Виджеты-деревья** или иерархические списки элементов предоставляются с помощью классов `QTreeWidget` и `QTreeWidgetItem`. Каждый элемент виджета-дерева может иметь свои собственные дочерние элементы и отображать данные в нескольких столбцах. Виджеты-деревья создаются также, как и другие виджеты:

```
QTreeWidget *treeWidget = new QTreeWidget(this);
```

Прежде, чем добавлять элементы в виджет-дерево, должно быть установлено количество столбцов. Например, мы можем определить два столбца и создать текстовые метки для отображения заголовков наверху каждого столбца:

```
treeWidget->setColumnCount(2);
QStringList headers;
headers << tr("Subject") << tr("Default");
treeWidget->setHeaderLabels(headers);
```

Самый легкий способ настроить надписи для каждой секции состоит в создании списка строк. Для более сложных заголовков вы можете создать элемент дерева, оформить его и использовать в качестве заголовка виджета-дерева.

Элементы верхнего уровня виджета-дерева создаются с виджетом-деревом в качестве родителя. Они могут быть вставлены в произвольном порядке, или вы можете определить порядок, указав при создании элемента предшествующий элемент:

```
QTreeWidgetItem *cities = new QTreeWidgetItem(treeWidget);
cities->setText(0, tr("Cities"));
QTreeWidgetItem *osloItem = new QTreeWidgetItem(cities);
osloItem->setText(0, tr("Oslo"));
osloItem->setText(1, tr("Yes"));
QTreeWidgetItem *planets = new QTreeWidgetItem(treeWidget, cities);
```

Виджеты-деревья с элементами верхнего уровня обращаются несколько иначе, чем с элементами, находящимися в глубине иерархии. Элементы верхнего уровня могут быть удалены из дерева с помощью вызова функции виджета-дерева `takeTopLevelItem()`, а элементы более низких уровней удаляются с помощью вызова функции `takeChild()` их родительского элемента. На верхний уровень древовидной структуры элементы могут быть вставлены с помощью функции `insertTopLevelItem()`. На более низких уровнях используется функция `insertChild()` родительского элемента.

Элементы легко перемещаются между самым верхним и более низкими уровнями дерева. Нам требуется лишь проверить, являются ли эти элементы элементами верхнего уровня или нет. Эту информацию можно получить с помощью функции `parent()` элемента. Например, мы можем удалить текущий элемент виджета-дерева вне зависимости от его расположения:

```
QTreeWidgetItem *parent = currentItem->parent();
int index;
if (parent) {
    index = parent->indexOfChild(treeWidget->currentItem());
    delete parent->takeChild(index);
}
```

```

}
else {
    index = treeWidget->indexOfTopLevelItem(treeWidget->currentItem());
    delete treeWidget->takeTopLevelItem(index);
}

```

Вставка элемента в виджет-дерево осуществляется подобным образом:

```

QTreeWidgetItem *parent = currentItem->parent();
QTreeWidgetItem *newItem;
if (parent)
    newItem = new QTreeWidgetItem(parent, treeWidget->currentItem());
else
    newItem = new QTreeWidgetItem(treeWidget, treeWidget->currentItem());

```

**Виджеты-таблицы**, подобные тем которые предоставляются табличными редакторами, создаются с помощью `QTableWidget` и `QTableWidgetItem`. Они предоставляют прокручиваемую таблицу с заголовками и элементами в ней.

Таблицы могут создаваться сразу с нужным количеством строк и столбцов, или строки и столбцы могут добавляться по необходимости.

```

QTableWidget *tableWidget;
tableWidget = new QTableWidget(12, 3, this);

```

Элементы создаются вне таблицы, а затем размещаются в нужном месте:

```

QTableWidgetItem *newItem = new QTableWidgetItem(tr("%1").arg(pow(row, column+1)));
tableWidget->setItem(row, column, newItem);

```

Горизонтальные и вертикальные заголовки можно добавить в таблицу, создав их как элементы вне таблицы, а затем установив их в качестве заголовков:

```

QTableWidgetItem *valuesHeaderItem = new QTableWidgetItem(tr("Values"));
tableWidget->setHorizontalHeaderItem(0, valuesHeaderItem);

```

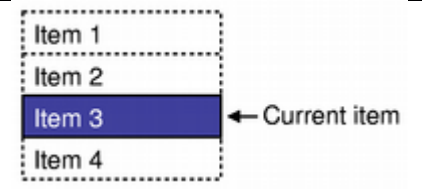
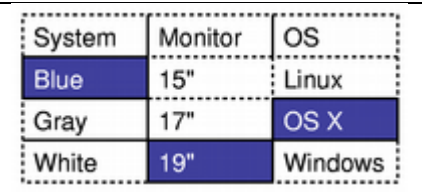
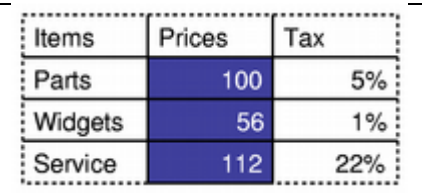
Обратите внимание на то, что нумерация строк и столбцов в таблице начинается с нуля.

## 6. Другие типовые действия с виджетами.

Иногда в виджете представления элементов полезно **скрывать** элементы, вместо того чтобы удалять их. Элементы всех виджетов могут быть скрыты, а потом вновь показаны. Определить, скрыт ли элемент, можно с помощью функции `isItemHidden()`, а скрыть элементы с помощью функции `setItemHidden()`.

Так как эта операция воздействует на элементы, она доступна во всех трех вспомогательных классах представлений.

Способ, которым **выделяются** элементы, управляется режимом выделения виджета (`QAbstractItemView::SelectionMode`). Данное свойство указывает, может ли пользователь выбирать один или несколько элементов, и, если пользователь может выбирать несколько, должен ли это быть непрерывный диапазон. Режим выбора работает одинаково для всех виджетов.

	<p>Выделение единственного элемента: Если пользователю нужно выбрать единственный элемент виджета, то режим по умолчанию - <code>SingleSelection</code> - наиболее подходящий. В этом режиме текущий элемент и выделенный элемент совпадают.</p>
	<p>Выделение нескольких элементов: В этом режиме, пользователь может изменять состояние выделения любого элемента в виджете без изменения существующего выделения, подобно тому как независимо друг от друга можно переключать флажки (checkboxes).</p>
	<p>Расширенное выделение: виджетам, которым часто требуется выбирать множество смежных элементов, например, электронные таблицы, требуется режим <code>ExtendedSelection</code>. В этом режиме непрерывные диапазоны элементов могут быть выделены с помощью мыши или клавиатуры. Сложные выделения, включающие множество несмежных друг-другу элементов, могут быть сделаны с помощью клавиш-модификаторов.</p> <p>Если пользователь выделяет элемент без использования клавиш-модификаторов, существующее выделение очищается.</p>



Выбранные в виджете элементы можно **прочитать** с помощью функции `selectedItems()`, возвращающей список соответствующих элементов, по которому можно перемещаться. Например, с помощью следующего кода мы можем найти сумму всех числовых значений выделенных элементов:

```
QList<QTableWidgetItem *> selected = tableWidget->selectedItems();
QTableWidgetItem *item;
int number = 0;
double total = 0;
foreach (item, selected) {
    bool ok;
    double value = item->text().toDouble(&ok);
    if (ok && !item->text().isEmpty()) {
        total += value;
        number++;
    }
}
```

В режиме выделения единственного элемента, текущий элемент будет находиться в выделении. В режимах выделения нескольких элементов и расширенного выделения текущий элемент может не находиться в выделении, это зависит от способа, которым пользователь осуществляет выделение.

Обычно полезно иметь возможность **найти элементы** внутри представления или для разработчика, или в качестве возможности, предоставляемой пользователю. Все три вспомогательных класса представления элементов предоставляют такую возможность с помощью функции `findItems()`.

Элементы ищутся по тексту, который они содержат, в соответствии с критериями, заданными набором значений `Qt::MatchFlags`. С помощью функции `findItems()` можно получить список соответствующих элементов:

```
QTreeWidgetItem *item;
QList<QTreeWidgetItem *> found = treeWidget->findItems(
    itemText, Qt::MatchWildcard);
foreach (item, found) {
    treeWidget->setItemSelected(item, true);
    // Показать для каждого элемента item->text(0).
}
```

Вышеприведенный код выделяет элементы виджета-дерева, если их текст содержит заданную строку поиска. Этот пример можно использовать также в виджетах-списках и виджетах-таблицах.