Введение в Model-View на примере таблиц в Qt4. Материалы семинаров.

Самарев Роман Станиславович, samarev [ ] acm.org
канд. техн. наук, доцент каф. «Компьютерные системы и сети», МГТУ им. Н.Э.Баумана, 2013.

В Qt4 существует два принципиально различающихся подхода к использованию интерфейсных элементов: таблиц, списков и деревьев. Первый подход - «элементно-базированные» классы – QTableWidget, QListWidget, QTreeWidget, реализующие по-элементную модель доступа. Такой же подход использовался в Qt3. Указанные классы реализованы на основе новых классов QTableView, QListView, QTreeView, реализующих принцип Model-View-Delegate.

Пример:

```
QStringList strlist;
strlist << "Name" << "Description" << "Sum";
QTableWidget *simple_table = new QTableWidget(this);
simple_table->setColumnCount( 3 );
simple_table->setHorizontalHeaderLabels( strlist );
simple_table->setRowCount( 10 );
simple_table->setSelectionBehavior(QAbstractItemView::SelectRows);
connect(simple_table, SIGNAL(itemSelectionChanged()),
        this, SLOT(selectionChanged()));

// модель данных стандартная, элементы по умолчанию не созданы
QTableWidgetItem *item;
for (int i=0; i<10; i++){
    item = new QTableWidgetItem();
     // Ограничиваем возможности первого столбца только отображением.
     // Устанавливаем только требуемые флаги.
    item->setFlags(Qt::ItemIsSelectable | Qt::ItemIsEnabled);
    item->setText( QString("String %1").arg(i+1));
    simple_table->setItem(i,HEADER_NAME,item);

    item = new QTableWidgetItem();
    item->setText( QString().setNum((i+1) * 10 ));
    simple_table->setItem(i,HEADER_SUM,item);
}
```

Константы HEADER_NAME, HEADER_SUM имеют значение 0 и 2, соответственно.

После выполнения указанного выше фрагмента будет создана таблица, содержащая 3 колонки и 10 строк, причём первая колонка будет содержать текст и не будет позволять себя редактировать, последняя – будет содержать числа с возможностью ввода любого значения, поскольку каких-либо проверок не предусмотрено, а вторая колонка будет содержать пустые ячейки, которые, однако, можно редактировать и считывать.

Если потребуется переопределить внешний вид элемента, цвет заливки и пр., необходимо реализовать свои классы-элементы на основе `QTableWidgetItem`. Однако, забегая вперед, упомянем, что также имеется возможность установить на строки или колонки делегат, изменяющий режим редактирования и способ проверки данных:

```
m_simple_table->setItemDelegateForColumn(
HEADER_SUM, new ComboBoxDelegate(m_simple_table));
```

Иной подход – модель-представление. См. гл. 12 Шлее. Qt4.5[1]. Этот подход позволяет отделить данные от непосредственного отображения, что исключает необходимость дублирования данных. Модель может быть привязана к внешним данным, которые расположены в базе данных или в файле. Более подробно см. документацию Qt4[2] в разделе Model/View Programming.

Базовый принцип Model View заключается в том, что:

- Существует модель, задачей которой является обеспечить доступ к данным, а также предоставить информацию о том, с какими атрибутами/флагами необходимо эти данные отобразить. Например указать цвет фона, указать вид данных: рисунок, текст, селектор и пр. Отметим, что имеются предопределенные модели QStandardItemModel, QStringListModel, QDirModel, QFileSystemModel.
- Существует схема представления (View). Обеспечивает каркас для отображения таблицы, списка или дерева (стандартные классы QListView, QTreeView, QTableView). Представление также может быть переопределено.
- Существуют делегаты, задачей которых является взаимодействие с пользователем: получение данных, их передача в модель данных, а также получение из модели данных и передача на отображение. Если нет необходимости изменять стандартный ввод или вывод, делегаты можно явно не использовать.

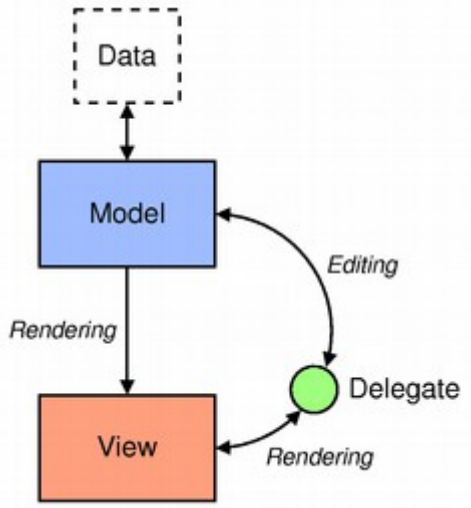

Рисунок 1 Принцип Model-View (из документации Qt4)

Существуют так называемые прокси-модели, которые используются например для того, чтобы обеспечить сортировку данных при их отображении без изменения физического расположения этих данных.

Рассмотрим приложение, внешний вид которого представлен на рисунке 2. Таблица в верхней части окна реализована классом QTableWidget, таблица в нижней части окна - QTableView. Фрагмент кода, обеспечивающий отображение верхней таблицы представлен ранее. Выпадающий список реализован при помощи делегата `ComboBoxDelegate`, код которого будет рассмотрен позже.

Нижняя таблица реализована с помощью принципа Model View, поэтому первое, что необходимо реализовать – модель данных. Эта модель будет сохранять данные, занесенные в таблицу, в файле. Обратите внимание на то, что правая колонка содержит в качестве элемента редактирования выпадающий список, причём в том случае, если значение в ячейке равно 10000, будет отрисован ромб.

[1] М.Шлее. Qt4.5. Профессиональное программирование на C++. БХВ-Петербург.-2010
[2] Qt Assistant

Нижняя отдельная строка в форме предназначена для вывода выбранных в данный момент значений таблицы.

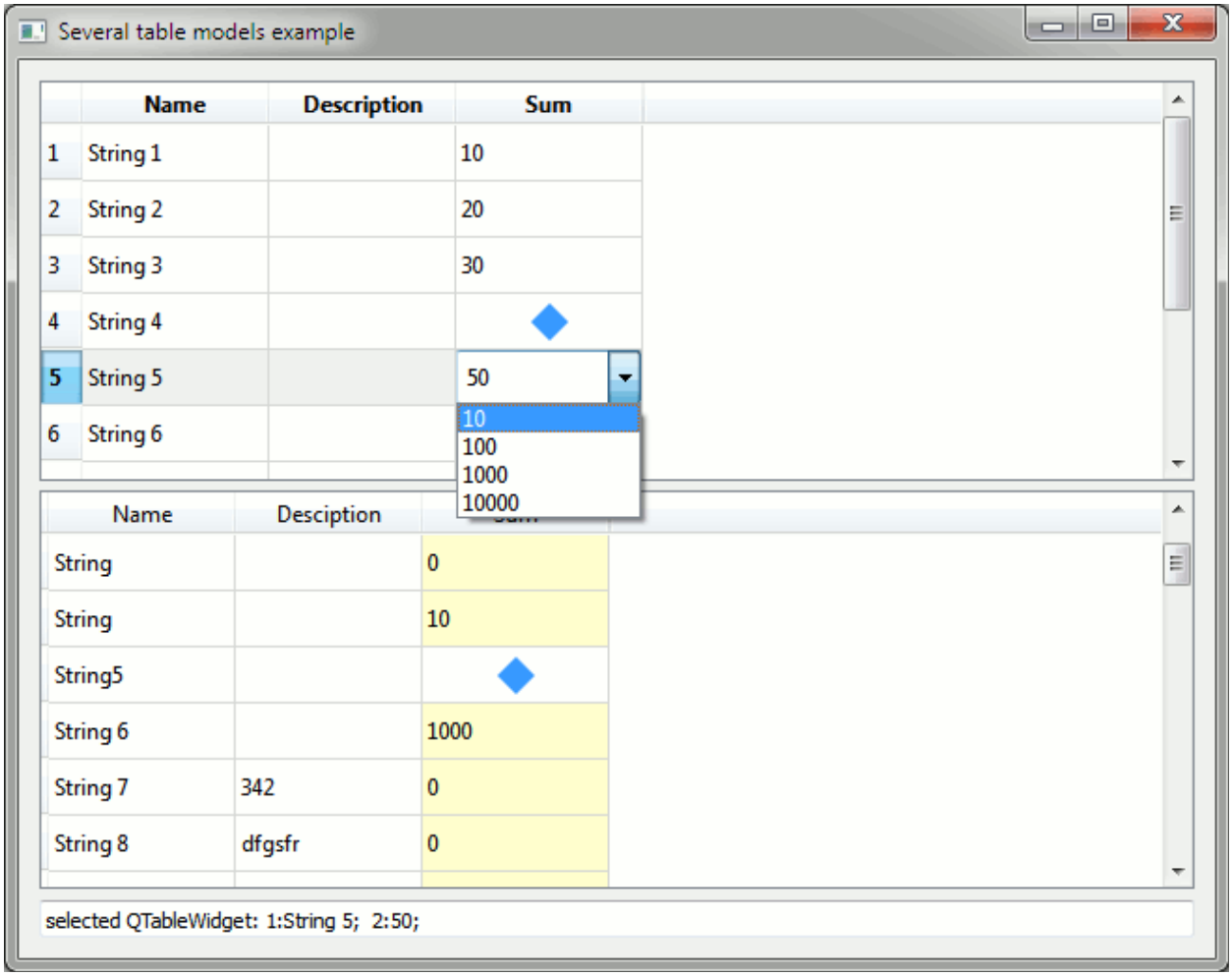

Рисунок 2 Окно приложения с несколькими вариантами таблиц

Для реализации модели необходимо реализовать класс-потомок от абстрактного класса `QAbstractTableModel`.

```
class TableModel: public QAbstractTableModel
```

Необходимо определить следующие методы:

```
/// количество строк. Устанавливаем так, чтобы скроллер отображался корректно
virtual int rowCount(const QModelIndex &) const {return m_recList.size();};

/// устанавливаем количество столбцов.
virtual int columnCount(const QModelIndex &) const {return 3;};

/// функция для передачи данных пользователю
virtual QVariant data(const QModelIndex &,int) const;

/// Функция для приёма данных от пользователя
virtual bool setData(const QModelIndex &index, const QVariant &value, int
role);
```

Для того чтобы модель обеспечила отображение названий столбцов, переопределяем метод:

```
virtual QVariant headerData( int section, Qt::Orientation orientation,
```

```
                    int role) const;
```

Отметим, что класс `QModelIndex` предназначен для указания координат элемента таблицы, к которому производится обращения.

Ключевой особенностью является наличие методов data и setData, именно через которые представление (View) взаимодействует с моделью.

Обратим внимание на метод:

```
QVariant TableModel::data(const QModelIndex & index,int role) const
```

Представление вызывает этот метод для каждого конкретного элемента таблицы, координаты которого заданы через index. Причём под ролью здесь понимается причина вызова этого метода, среди которых: отобразить значение как текст, отобразить картинку, элемент ввода с запоминанием состояния (CheckBox), указать цвет и пр. См. документацию Qt4. При запросе представления данных метод обеспечивает их преобразование из физической модели хранения в то, что должно быть отображено по соответствующим координатам таблицы. Пример реализации метода data:

```
QVariant TableModel::data(const QModelIndex & index,int role) const
{
      // выводим в консоль текущие значения параметров и считаем, сколько
      // раз вызывается метод TableModel::data и для каких ролей
      qDebug() << "data" << index << role;

    QVariant result;
    // закрасим колонку суммы
    if (role == Qt::BackgroundRole && index.column() == HEADER_SUM)
        return QColor(255,255,204);

    // Если необходимо отобразить картинку - ловим роль Qt::DecorationRole
    // Если хотим отобразить CheckBox, используем роль Qt::CheckStateRole

    // если текущий вызов не относится к роли отображения, завершаем
    if (!index.isValid() || role != Qt::DisplayRole)
        return result;

    // устанавливаем соответствие между номером столбца и полем записи
    const TableModel::Record &rec = m_recList.at(index.row());
    switch( index.column() ) {
    case HEADER_NAME:
        result = rec.name;
        break;
    case HEADER_DESC:
        result = rec.descr;
        break;
    case HEADER_SUM:
        result = rec.sum;
        break;
    };

    return result;
}
```

Метод возвращает результат типа QVariant, поскольку он может содержать текст, изображения и пр. в зависимости от заявленной роли и требуемого способа отображения.

Используя этот метод, можем указать цвет фона (роль `Qt::BackgroundRole`) или с лёгкостью изменить ячейку для ввода строки на ячейку для ввода селектора (роль `Qt::CheckStateRole`).

Симметричный метод, предназначенный для ввода данных в модель, который также вызывается представлением, также вызывается для каждого изменённого элемента, и также содержит указание на роль, с которой осуществляется вызов. Пример метода setData:

```
bool TableModel::setData(     const QModelIndex &index,
                        const QVariant &value,
                        int role)
{
      qDebug() << "setData" << index << value.toString() << role;
      // нас интересует только роль, сообщающая об изменении
      if (index.isValid() && role == Qt::EditRole) {
        // строка таблицы однозначно связана с номером в массиве
        TableModel::Record &rec = m_recList[index.row()];

        // определяем соответствие между столбцами и полями структуры
        switch(index.column()) {
        case HEADER_NAME:
            rec.name = value.toString();
            break;

        case HEADER_DESC:
            rec.descr = value.toString();
            break;

        case HEADER_SUM:
            rec.sum = value.toInt();
            break;

        // обратились непонятно к чему
        default:
            return false;
        }
        // оповещаем об изменении данных
        emit(dataChanged(index, index));

        // данные приняты
        return true;
    }

    return false;
}
```

Метод возвращает true или false как факт того, удалось ли записать данные в модель. Задачей этого метода является преобразование координат таблицы в конкретное физическое расположение данных.

Возможность редактирования элемента определяется результатом, возвращаемым методом `flags()`. Если необходимо заблокировать редактирование, этот метод должен быть также переопределён.

```
Qt::ItemFlags TableModel::flags(const QModelIndex &index) const
{
      Qt::ItemFlags result;
      if (!index.isValid())
            return Qt::ItemIsEnabled;

      // разрешаем редактирование всего, кроме первого столбца
      result = QAbstractTableModel::flags(index);
      if (index.column() != HEADER_NAME)
```

```
            result |= Qt::ItemIsEditable;
      return result;
}
```

Для того, чтобы реализовать элемент ввода в виде выпадающего списка вместо стандартной строки ввода, необходимо использовать механизм делегатов. Для этого реализуем класс `ComboBoxDelegate` как поток класса `QItemDelegate` и переопределяем следующие методы:

- Метод для создания конкретного элемента для подстановки в редактируемое поле. Именно в этом методе должен быть создан QComboBox(parent).

```
virtual QWidget *createEditor(QWidget *parent,
      const QStyleOptionViewItem &option,
      const QModelIndex &index) const;
```

- Метод, в который обеспечивает подстановку данных из модели в поле редактирования. Для того, чтобы забрать данные для конкретного элемента из модели, метод вызвыается с указанием индекса index.

```
virtual void setEditorData(QWidget *editor,
                  const QModelIndex &index) const;
```

- Метод, предназначенный для помещения данных в модель model с индексом index после окончания редактирования элемента editor.

```
virtual void setModelData(QWidget *editor, QAbstractItemModel *model,
                     const QModelIndex &index) const;
```

- Метод, обеспечивающий корректное отображение элемента редактирования в ячейку таблицы.

```
virtual void updateEditorGeometry(QWidget *editor,
                  const QStyleOptionViewItem &option,
            const QModelIndex &index) const;
```

- Для того, чтобы изменить отображение не редактируемой в данный момент ячейки (на рисунке 2 ромбом отображаются поля, имеющие значение 10000), необходимо переопределить метод paint, причем если нестандартное отображение требуется не во всех случаях, то для них необходимо явно вызывать метод paint предка - QItemDelegate::paint(painter, option, index).

```
virtual void paint(QPainter *painter, const QStyleOptionViewItem &option,
               const QModelIndex &index) const;
```

После объявления и описания всех необходимых методов можем перейти к рассмотрению кода, создающего таблицу по варианту Model-View. Создание таблицы выглядит следующим образом:

```
    TableModel *table_model = new TableModel(this);
    QTableView *custom_table = new QTableView(this);
    custom_table->setModel(table_model);
    custom_table->setSelectionBehavior(QAbstractItemView::SelectRows);

    custom_table->setItemDelegateForColumn(
            HEADER_SUM, new ComboBoxDelegate(m_custom_table));
```

Следует обратить внимание на то, как получить выбранные строки (то, что выбраны могут быть только строки, определяет вызов метода `setSelectionBehavior()` со значением `QAbstractItemView::SelectRows`).

Для того, чтобы обеспечить обработку строк по мере их выделения, необходимо связать генерируемый моделью сигнал `selectionChanged` со слотом, который определям самостоятельно. Связывание сигнала со слотом выглядит следующим образом:

```
    connect( custom_table->selectionModel(),
SIGNAL(selectionChanged(const QItemSelection &, const QItemSelection &)),
                SLOT(selectionChanged())
                );
```

Слот для обработки изменения выделенных строк должен обрабатывать как верхнюю, так и нижнюю таблицы и выглядит следующим образом:

```
void TablesWidget::selectionChanged ()
{
    QTableView *tbl = (QTableView *)sender();
    QString str;
    if ( tbl == m_simple_table ) {
        str += "selected QTableWidget: ";

// Внимание! данный способ получения элементов строки можно использовать
// только в том случае, если все элементы были явно созданы. Иначе
// количество элементов в строке может оказаться меньше, чем количество
//  столбцов. В случае, если используется режим выделения нескольких строк,
// то список будет содержать все выделенные элементы
        QList<QTableWidgetItem *> list = m_simple_table->selectedItems ();
        for (int i = 0; i < list.size(); ++i) {
            str += QString().setNum(i+1) + ":" + list.at(i)->text() + ";  ";
        }
    } else {
// Через индексы можем получить список выделенных ячеек в любом случае
// для любого потомка QTableView, включая QTableWidget
        QModelIndexList indexes =
                m_custom_table->selectionModel()->selection().indexes();

// Но только для своей модели можем определить метод,
// который соберёт данные в строку. Поскольку выбрано поведение
// модели выбора SelectRows, то необходимы только индексы строк
// Если требуется лишь первая строка, то достаточно взять 0-й элемент
        if (indexes.size())
            str = m_table_model->getRow(indexes.at(0));
    }
    m_text->setText( str );
};
```

В реализации этого метода следует обратить внимание на то, что `QTableWidget::selectedItems()` возвращает список адресов тех элементов, которые в данный момент выбраны. Их индексы в массиве могут НЕ СООТВЕТСТВОВАТЬ индексам таблицы, поскольку в списке хранятся только существующие элементы без пропусков (если выбрана строка), а, кроме того, могут содержаться произвольные выбранные элементы, если это допустимо моделью выбора (например выделение прямоугольником).

Цепочка вызовов `QTableView::selectionModel()->selection().indexes()` позволяет получить список индексов выбранных элементов аналогично первому случаю. Поскольку моделью выбора была установлена строка, то для получения данных достаточно знать индекс строки любого из возвращённых индексов, например `indexes.at(0).row()`.

Аналогичные принципы используются при работе с QListView/QTreeView.

Литература

- М.Шлее. Qt4.5. Профессиональное программирование на C++. БХВ-Петербург.-2010
- http://qt-project.org/doc/qt-4.8/model-view-programming.html

Приложение 1. Исходные тексты программы.

Файл qt_mvt.pro

```
#тип проекта - приложение
TEMPLATE = app

#определяем имя исполняемого файла
TARGET = mv_table_example

#Подключаем зависимости только из текущей директории
DEPENDPATH += .
INCLUDEPATH += .

CONFIG += qt

# подключаем консоль для того, чтобы видеть отладочный вывод в Windows
debug {
 CONFIG += console
}

# Подключаем все файлы проекта
HEADERS += mvt.h combo.h
SOURCES += main.cpp mvt.cpp combo.cpp
```

Файл main.cpp

```
#include <QApplication>
#include "mvt.h"

//«Элементно-базированные» классы – QTableWidget, QListWidget, QTreeWidget
//Модель доступа по-элементная. Классы реализованы на основе
Q...Table/List/Tree...View.
//
//Иной  подход – модель-представление. См. гл. 12 Шлее. Qt4.5.
//Позволяет отделить данные от непосредственного отображения, что исключает
необходимость
//дублировании данных. Модель может быть привязана к внешним данным, расположенным в
БД или в файле.
int main(int argc, char **argv)
{
    QApplication app(argc, argv);

    TablesWidget widget;
    widget.setWindowTitle(widget.tr("Several table models example"));
    widget.resize(640, 480);
    widget.show();

    return app.exec();
}
```

Файл mvt.h

```
#ifndef _MVC_H_09052012_
#define _MVC_H_09052012_
#include <QWidget>
#include <QAbstractTableModel>
#include <QList>
#include <QDataStream>
```

```
// декларируем классы, определение которых не подключено
class QTableWidget;
class QTableView;
class QLineEdit;

// декларируем класс, который будет объявлен ниже
class TableModel;

/// Основное окно
class TablesWidget : public QWidget
{
    Q_OBJECT
public:
    TablesWidget();

protected:
    QTableWidget *m_simple_table; //< Верхняя таблица

      QTableView *m_custom_table;   //< Нижняя таблица
    TableModel *m_table_model;

      QLineEdit *m_text; //< Поле для вывода выделенных элементов

public slots:
    /// слот для определения момента изменения выделения
    void selectionChanged ();
};

/// класс, определяющий модель таблицы
/// Дополнительной функцией будет сохранение данных в файл.
class TableModel: public QAbstractTableModel
{
    Q_OBJECT
public:
    TableModel(QObject * parent = 0);
    virtual ~TableModel();

    // Обязательные методы
      /// количество строк. Устанавливаем так, чтобы скроллер отображался корректно
    virtual int rowCount(const QModelIndex &) const {return m_recList.size();};
      /// устанавливаем количество столбцов. Поскольку они не меняются - указываем
константу
    virtual int columnCount(const QModelIndex &) const {return 3;};
      /// функция для передачи данных пользователю
      virtual QVariant data(const QModelIndex &,int) const;
    /// Функция для приёма данных от пользователя
    virtual bool setData(const QModelIndex &index, const QVariant &value, int role);

    /// Описание заголовков таблицы
    virtual QVariant headerData(int section, Qt::Orientation orientation, int role)
const;

      /// Определение возможностей ячейки таблицы (отображение, редактируемость...)
    virtual Qt::ItemFlags flags(const QModelIndex &index) const;

    /// Возвращаем строку, соответствующую указанному столбцу
    QString getRow(const QModelIndex &index);

protected:
    /// Стуктура для хранения записей
    struct Record {
        QString name;
        QString descr;
        quint32 sum;
    };

    QList<Record> m_recList; ///< Список для хранения записей
    QString m_filename; ///< Имя файла с данными

    // Операторы для сериализации структуры Record
    friend QDataStream &operator<<(QDataStream &dataStream, const Record& rec);
```

```
    friend QDataStream &operator>>(QDataStream &dataStream, Record& rec);
};
#endif
```

Файл combo.h

```
#ifndef _COMBO_H_120517_
#define _COMBO_H_120517_
#include <QItemDelegate>

/// Делегат.
/// Обеспечивает создание выпадающего списка при входе в редактирование.
/// Обеспечивает отображение ромба, если ячейка соответствует условию.
class ComboBoxDelegate : public QItemDelegate
{
    Q_OBJECT
public:
    ComboBoxDelegate(QObject *parent = 0):QItemDelegate(parent){};

       // ***** Переопределяем методы, необходимые для создания нестандартного ********
       // ****************   элемента ввода и редактирования ***************************

       /// Создаём элемент для подстановки в редактируемое поле
    virtual QWidget *createEditor(QWidget *parent, const QStyleOptionViewItem &option,
                          const QModelIndex &index) const;

       /// Определяем как передать данные из модели с указанным индексом index
       /// в созданный элемент editor
    virtual void setEditorData(QWidget *editor, const QModelIndex &index) const;
       /// Определяем как забрать данные из editor и поместить их в model с индексом
index
    virtual void setModelData(QWidget *editor, QAbstractItemModel *model,
                      const QModelIndex &index) const;
       /// Обновляем размеры editor в соответствии с имеющимися размерами поля таблицы
    virtual void updateEditorGeometry(QWidget *editor,
                    const QStyleOptionViewItem &option, const QModelIndex &index)
const;
    // ********************************************************************************

       /// переопределяем функцию рисования для замены отображения
       /// не редактируемого в данный момент элемента
    virtual void paint(QPainter *painter, const QStyleOptionViewItem &option,
                const QModelIndex &index) const;
};
#endif
```

Файл mvt.cpp

```
#include <QTableWidget>
#include <QTableView>
#include <QVBoxLayout>
#include <QSplitter>
#include <QLineEdit>
#include <QFile>
#include <qDebug>

#include "mvt.h"
#include "combo.h"

/// Идентификаторы столбцов таблицы
enum {
       HEADER_NAME,
       HEADER_DESC,
       HEADER_SUM
};

TablesWidget::TablesWidget()
:QWidget(NULL)
```

```
{
	QVBoxLayout *layout = new QVBoxLayout();
	setLayout( layout );

	QSplitter *splitter = new QSplitter(Qt::Vertical, this);

	// ************** Создаём первую таблицу ********************
	QStringList strlist;
	// фукнция tr() - заготовка под будущую интернационализацию
	strlist << tr("Name") << tr("Description") << tr("Sum");
	m_simple_table = new QTableWidget(this);
	m_simple_table->setColumnCount( 3 );
	m_simple_table->setHorizontalHeaderLabels( strlist );
	m_simple_table->setRowCount( 10 );
	m_simple_table->setSelectionBehavior(QAbstractItemView::SelectRows);
	connect(m_simple_table, SIGNAL(itemSelectionChanged()),
		this, SLOT(selectionChanged()));

	// модель данных стандартная, элементы по умолчанию не созданы
	QTableWidgetItem *item;
	for (int i=0; i<10; i++) {
		item = new QTableWidgetItem();
		// Ограничиваем возможности первого столбца только отображением
		item->setFlags( Qt::ItemIsSelectable | Qt::ItemIsEnabled );
		item->setText( QString("String %1").arg(i+1));
		m_simple_table->setItem(i,HEADER_NAME,item);

		item = new QTableWidgetItem();
		item->setText( QString().setNum((i+1) * 10 ));
		m_simple_table->setItem(i,HEADER_SUM,item);
	}
	m_simple_table->setItemDelegateForColumn(
		HEADER_SUM, new ComboBoxDelegate(m_simple_table));
	// ************* конец создания первой таблицы **********************


	// ****** Создаем другую таблицу с явно определённой моделью *********
	m_table_model = new TableModel(this);
	m_custom_table = new QTableView(this);
	m_custom_table->setModel(m_table_model);
	m_custom_table->setSelectionBehavior(QAbstractItemView::SelectRows);
	connect(
		m_custom_table->selectionModel(),
		SIGNAL(selectionChanged(const QItemSelection &, const QItemSelection &)),
		SLOT(selectionChanged())
		);

	m_custom_table->setItemDelegateForColumn(
		HEADER_SUM, new ComboBoxDelegate(m_custom_table));
	// ************* конец создания второй таблицы **********************


	m_text = new QLineEdit(this);
	m_text->setReadOnly(true);

	splitter->addWidget( m_simple_table );
	splitter->addWidget( m_custom_table );

	layout->addWidget( splitter );
	layout->addWidget( m_text );
}

void TablesWidget::selectionChanged ()
{
	QString str;
	QTableView *tbl = (QTableView *)sender();
	if ( tbl == m_simple_table ) {
		str += "selected QTableWidget: ";

		// Внимание! данный способ получения элеметов строки можно использовать
		// только в том случае, если все элементы были явно созданы. Иначе
```

```
                // количество элементов в строке может оказаться меньше, чем количество
                // столбцов.
                QList<QTableWidgetItem *> &list = m_simple_table->selectedItems ();
                for (int i = 0; i < list.size(); ++i) {
                        str += QString().setNum(i+1) + ":" + list.at(i)->text() + ";  ";
                }
        } else {
                // Через индексы можем получить список выделенных ячеек в любом случае
                QModelIndexList indexes =
                        m_custom_table->selectionModel()->selection().indexes();

                // Но только для своей модели можем определить метод,
                // который соберёт данные в строку. Поскольку выбрано поведение
                // модели выбора SelectRows, то необходимы только индексы строк
                // Если требуется лишь первая строка, то достаточно взять 0-й элемент
                if (indexes.size())
                        str = m_table_model->getRow(indexes.at(0));
        }
        // добавляем текст в соответствующее поле
        m_text->setText( str );
};

//******************************************************************************

TableModel::TableModel(QObject * parent)
: QAbstractTableModel(parent)
{
        m_filename = "test.dat";
        QFile file(m_filename);
        if (file.open(QIODevice::ReadOnly)) {
                // используем стандартный метод сериализации
                QDataStream in(&file);
                in.setVersion(QDataStream::Qt_4_0);
                // непосредственно заполняем QList
                in >> m_recList;
                file.close();
        };

        // Поскольку ввод новых строк не предсмотрен, сделаем добавление новой строки
        // при каждом запуске программы.
        TableModel::Record rec;
        rec.name = "String " + QString().setNum(m_recList.size()+1);
        rec.sum = 0;
        m_recList.push_back(rec);
}

TableModel::~TableModel()
{
        // открываем файл на запись
        QFile file(m_filename);
        if (file.open(QIODevice::WriteOnly)) {
                // используем стандартный метод сериализации
                QDataStream out(&file);   // write the data
                out.setVersion(QDataStream::Qt_4_0);
                // записываем QList со всем содержимым в файл
                out << m_recList;
                file.close();
        }
}

QVariant TableModel::data(const QModelIndex & index,int role) const
{
        // выводим в консоль текущие значения параметров и считаем, сколько
        // раз вызывается метод TableModel::data и для каких ролей
        qDebug() << "data" << index << role;

        QVariant result;

        // закрасим колонку суммы
        if (role == Qt::BackgroundRole && index.column() == HEADER_SUM)
                return QColor(255,255,204);
```

```
	// Если необходимо отобразить картинку - ловим роль Qt::DecorationRole
	// Если хотим отобразить CheckBox, используем роль Qt::CheckStateRole

	// если текущий вызов не относится к роли отображения, завершаем
	if (!index.isValid() || role != Qt::DisplayRole)
		return result;

	// устанавливаем соответствие между номером столбца и полем записи
	const TableModel::Record &rec = m_recList.at(index.row());
	switch( index.column() ) {
	case HEADER_NAME:
		result = rec.name;
		break;
	case HEADER_DESC:
		result = rec.descr;
		break;
	case HEADER_SUM:
		result = rec.sum;
		break;
	};

	return result;
}

QVariant TableModel::headerData(int section, Qt::Orientation orientation, int role)
const
{
	// Для любой роли, кроме запроса на отображение, прекращаем обработку
	if (role != Qt::DisplayRole)
		return QVariant();

	// формируем заголовки по номуру столбца
	if (orientation == Qt::Horizontal) {
		switch (section) {
		case HEADER_NAME:
			return tr("Name");

		case HEADER_DESC:
			return tr("Desciption");

		case HEADER_SUM:
			return tr("Sum");
		}
	}
	return QVariant();
};

Qt::ItemFlags TableModel::flags(const QModelIndex &index) const
{
	Qt::ItemFlags result;
	if (!index.isValid())
		return Qt::ItemIsEnabled;

	// разрешаем редактирование всего, кроме первого столбца
	result = QAbstractTableModel::flags(index);
	if (index.column() != HEADER_NAME)
		result |= Qt::ItemIsEditable;
	return result;
}

bool TableModel::setData(const QModelIndex &index, const QVariant &value, int role)
{
	qDebug() << "setData" << index << value.toString() << role;

	if (index.isValid() && role == Qt::EditRole) {
		TableModel::Record &rec = m_recList[index.row()];

		// определяем соответствие между столбцами и полями структуры
		switch(index.column()) {
		case HEADER_NAME:
```

```
                rec.name = value.toString();
                break;

            case HEADER_DESC:
                rec.descr = value.toString();
                break;

            case HEADER_SUM:
                rec.sum = value.toInt();
                break;

            default:
                return false;
            }
            emit(dataChanged(index, index));

            return true;
        }

        return false;
}

QString TableModel::getRow(const QModelIndex &index)
{
        QString str;
        if (index.isValid()) {
            str = "Selected custom data model: ";
            const TableModel::Record &rec = m_recList.at(index.row());
            str +=
                "1:" + rec.name +
                "; 2:" + rec.descr +
                "; 3:" + QString().setNum( rec.sum);
        }
        return str;
}

// Операторы перенаправления ввода и вывода для реализации сериализации
QDataStream & operator<<(QDataStream &dataStream, const TableModel::Record& rec)
{
        // определяем порядок заполнения в файл
        dataStream << rec.name;
        dataStream << rec.descr;
        dataStream << rec.sum;
        return dataStream;
};
QDataStream & operator>>(QDataStream &dataStream, TableModel::Record& rec)
{
        // в том же порядке читаем из файла
        dataStream >> rec.name;
        dataStream >> rec.descr;
        dataStream >> rec.sum;
        return dataStream;
};
```

Файл combo.cpp

```
#include <QComboBox>
#include <QPainter>
#include "combo.h"

QWidget *ComboBoxDelegate::createEditor(QWidget *parent,
                                        const QStyleOptionViewItem &/* option */,
                                        const QModelIndex &/* index */) const
{
        // создаём выпадающий список (далее Редактор)
    QComboBox *editor = new QComboBox(parent);
        // разрешаем непосредственный ввод значений, а не только выбор из списка
    editor->setEditable(true);
        // добавляем несколько типовых значений
    editor->addItem("10");
```

```
    editor->addItem("100");
    editor->addItem("1000");
    editor->addItem("10000");

    return editor;
}

void ComboBoxDelegate::setEditorData(QWidget *editor,
                                     const QModelIndex &index) const
{
       // Получаем данные из модели и вставляем их так, как нужно, в поле Редактора
    QString value = index.model()->data(index, Qt::DisplayRole).toString();
    QComboBox *combo = static_cast<QComboBox*>(editor);
    combo->setEditText(value);
}
void ComboBoxDelegate::setModelData(QWidget *editor, QAbstractItemModel *model,
                                    const QModelIndex &index) const
{
       // получаем введенное в Редакторе значение и записываем его в модель
    QComboBox *combo = static_cast<QComboBox*>(editor);
    model->setData(index, combo->currentText(), Qt::EditRole);
}
void ComboBoxDelegate::updateEditorGeometry(QWidget *editor,
                                            const QStyleOptionViewItem &option,
                                            const QModelIndex &/* index */) const
{
       // вписываем Редактор в размеры ячейки таблицы
    editor->setGeometry(option.rect);
}

//-----------------------------------------------------

void ComboBoxDelegate::paint(QPainter *painter, const QStyleOptionViewItem &option,
                             const QModelIndex &index) const
{
    QString val;
    if (qVariantCanConvert<QString>(index.data()))
        val = qVariantValue<QString>(index.data());
    // Если в поле значение 10000 - рисуем ромб
    // Дополнительные примеры см в qt/examples/itemviews/stardelegate/
    if ( val == "10000") {
        const int PaintingScaleFactor = 100;
        QPolygonF diamondPolygon;
        painter->save();

        painter->setRenderHint(QPainter::Antialiasing, true);
        painter->setPen(Qt::NoPen);

        painter->setBrush( option.palette.highlight() );

        int yOffset = (option.rect.height() - PaintingScaleFactor) / 2;
        painter->translate(option.rect.x(), option.rect.y() + yOffset);
        painter->scale(PaintingScaleFactor, PaintingScaleFactor);

        // создание ромба следует вынести из этой функции в конструктор.
        // Оставлено только с целью иллюстрации
        diamondPolygon << QPointF(0.4, 0.5) << QPointF(0.5, 0.4)
                       << QPointF(0.6, 0.5) << QPointF(0.5, 0.6)
                       << QPointF(0.4, 0.5);
        painter->drawPolygon(diamondPolygon, Qt::WindingFill);
        painter->translate(1.0, 0.0);
        painter->restore();
    } else {
             // Для всех отстальных случаев выполняем действие по умолчанию
        QItemDelegate::paint(painter, option, index);
    }
}
```