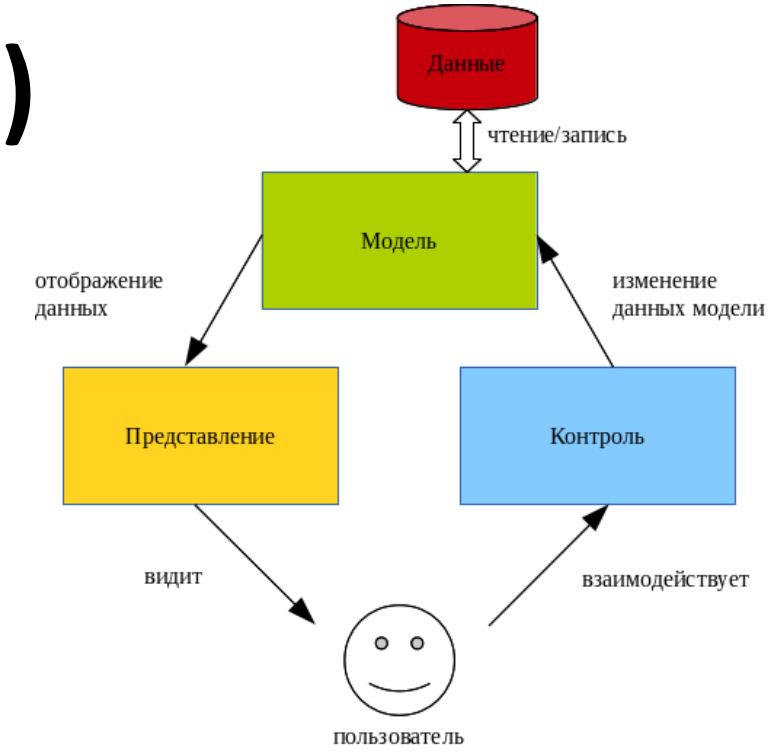
Модель-представление

Лекция + практика

MVC (Model-View-Controller)

MVC позволяет разделить части программы, отвечающие за хранение и доступ к данным, отображение данных и за взаимодействие с пользователем на отдельные слабо связанные модули. Подобное разделение ответственности упрощает структуру программы и позволяет вносить изменения в одну из этих частей не затрагивая остальные.



Этот шаблон разделяет программу на три компонента.

- **Модель**. Отвечает за данные и обеспечивает доступ к ним. Это **обертка** для данных, с которыми оперирует приложение. Сами данные могут храниться как непосредственно в модели, так и в стороннем хранилище (файл, БД…).

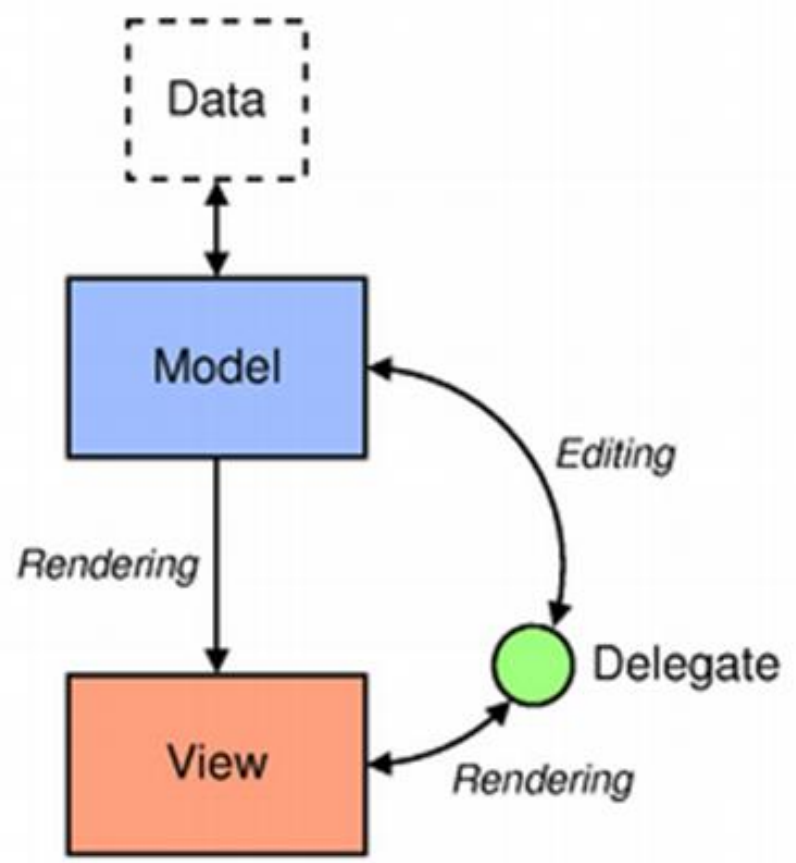
Модель реализует **логику** работы с данными (алгоритмы) и **не** зависит от представления (*view*) и контроллера, **но должна реагировать на запросы**, изменяя свое состояние.

- **Представление** отвечает за отображение данных, полученных из модели в понятном человеку виде.
- **Контроль** отвечает за взаимодействие с пользователем. Может изменять данные в модели.

Модель – представление в Qt

Здесь **модель** отвечает за данные и доступ к ним. Поскольку в графическом интерфейсе за отображение элементов и получение ввода от пользователя нередко отвечают одни и те же элементы, то вполне логична идея объединить **представление** и **контроль**. Именно так и сделано в Qt: **представление** не только отображает данные, но и выполняет функции контроля и отвечает за взаимодействие с пользователем. Но для того, чтобы из-за такого объединения не терять гибкость, было введено понятие **делегата**.

Делегат позволяет определять, как эти данные будут отображаться и как пользователь может их изменять. Представление же, по сути, теперь является контейнером для экземпляров делегата.



Активная модель может оповещать **представления** о том, что данные изменились. В свою очередь, **представление** может отобразить эти изменения. Именно такой вариант считается классической реализацией *MVC*.

В *Qt* применяется *активная модель*. Это дает возможность изменять данные в модели и не беспокоится об отображении обновленных данных — *представление* все сделает само.

Представления и делегат

Представление в *MVC* обеспечивает отображение данных. Это такая часть программы, которая определяет, как будут выглядеть данные и, в конечном итоге, что увидит пользователь.
Реализация *представления* в Qt имеет одну существенную особенность: представление здесь объединено с контролем. Т.к. в графическом интерфейсе нередко одни и те же элементы отвечают за отображение данных и их изменение. В качестве примера можно вспомнить табличный процессор. Каждая ячейка не только отображает данные но и отвечает за их изменение, а значит выполняет функции не только представления, но и контроля. Так что решение объединить их в одном элементе вполне логичное.

Делегат — это компонент, который отображает данные одного элемента модели и обеспечивает их редактирование. Экземпляры делегата создаются для каждого элемента модели и располагаются в представлении, которое по сути является контейнером. Именно делегат решает, как должны отображаться и редактироваться данные конкретного элемента и это дает нам большие возможности кастомизации. В приведенном выше примере с табличным процессором мы можем достаточно просто сделать так, чтобы данные в каждой ячейке редактировались при помощи *спинбокса* или *выпадающего списка*.

Может быть несколько представлений одних и тех же данных: данные могут отображаться по-разному (графическое, таблица, диаграмма…) и по-разному реагировать на действия пользователя.
НО!
Представление не изменяет данные, а только «сообщает» модели: что и как нужно изменить.

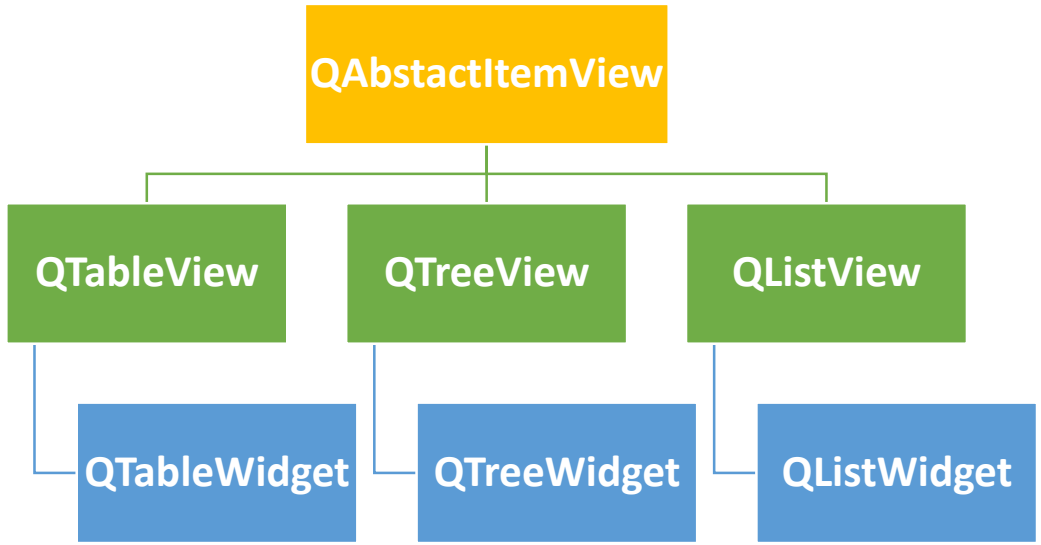
Способы визуализации данных в Qt

в Qt есть два способа визуализации данных:

- *Стандартные виджеты*, которые содержат отображаемые данные непосредственно
- *Представления* (View).

Несмотря на то, что визуализация посредством обоих подходов может выглядеть одинаково, внутренняя реализация имеет принципиальные отличия

Стандартные виджеты Qt	Архитектура Model/View
Содержат отображаемые данные	Оперируют внешними данными
QListWidget	QListView
QTableWidget	QTableView
QTreeWidget	QTreeView



Стандартные виджеты визуализации

Использование стандартных виджетов интуитивно более понятно и проще в реализации.
Стандартные виджеты:

- Содержат данные непосредственно в элементах.
- Внутри используются стандартные классы моделей.

Стандартные виджеты используются в таких задачах, которые:

- содержат единственный набор данных;
- единственное отображение этих данных;
- данные представляются стандартной моделью.

Класс	Элемент данных	Модель
QListWidget	QListWidgetItem	QListModel
QTableWidget	QTableWidgetItem	QTableModel
QTreeWidget	QTreeWidgetItem	QTreeMode

Если требуется несколько разных представлений одних и тех же данных , если каждое представление может данные модифицировать по-разному, то возникают копии данных и, следовательно, **проблемы синхронизации**.

Архитектура Model/View

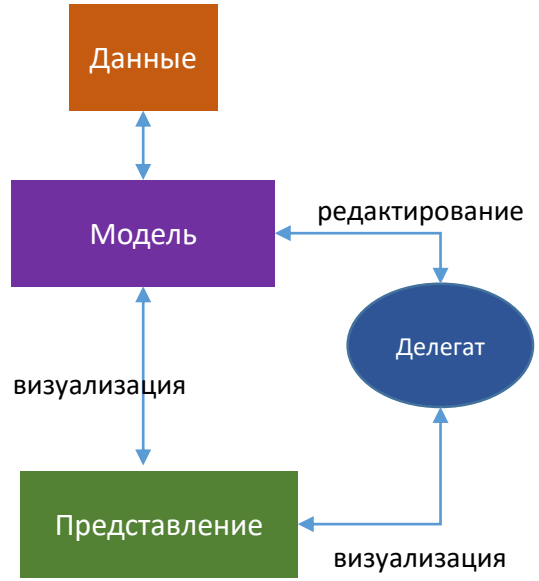
Классы моделей наследуются от	QAbstractItemModel
Классы представления наследуются от	QAbstractItemView
модельные индексы, единообразно представляющих элементы данных (независимо от внутренней организации модели)	QModelIndex
Классы делегатов наследуются от	QAbstractItemDelegate

Классы этой архитектуры разделены на группы:

- модели,
- представления и
- делегаты.

Каждый из этих компонентов описывается абстрактным классом, предоставляющим общий интерфейс и, в некоторых случаях, особенности реализации по умолчанию.

- **Модель** осуществляет соединение с источником данных.
- **Представление** запрашивает данные из модели.
- В стандартных представлениях **делегат** отображает элементы данных.
- При редактировании данных также используется **делегат**.



Классы архитектуры представления

Готовые классы *представлений* Qt :
QListView - отображает список элементов.
QTableView - отображает данные модели в виде таблицы.
QTreeView - отображает элементы модели в виде иерархического списка.

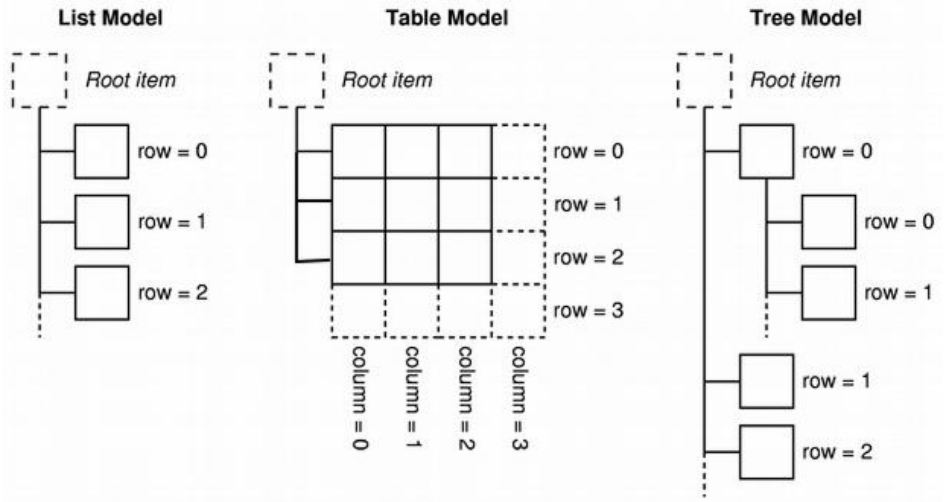
Реализация поведения готовых классов достаточна для большинства приложений. Они предоставляют основные средства редактирования и могут быть настроены для более специфических пользовательских интерфейсов посредством делегатов.

```
QStringList cities;
cities <<"Санкт-Петербург” << "Москва“
<<"Ярославль“ << “Псков”;
QStringListModel   model(cities);
QListView   view;
view.setModel (&listModel);
view.show();
```

Существуют готовые модели
QStringListModel - используется для хранения простого списка элементов QString.
QSqlQueryModel,
QSqlTableModel,
QSqlRelationalTableModel - используются для доступа к реляционным (основанным на таблицах) базам данных.
QStandardItemModel - многоцелевая модель, которая может использоваться для представления различных структур данных в виде списка, таблицы или дерева.
QFileSystemModel предоставляет информацию о файлах и директориях локальной файловой системы. Она не содержит самих элементов данных, а просто представляет файлы и директории локальной файловой системы.

Связь модели и представления задается методом - **setModel()**
void QAbstractItemView::setModel(QAbstractItemModel *model)

Индексы модели

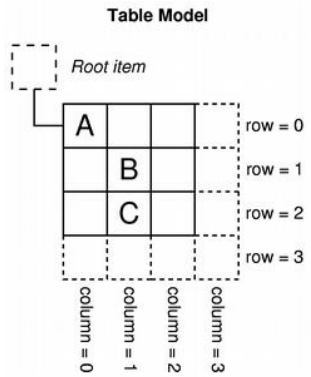


Индекс модели предоставляет универсальный способ доступа к каждому элементу модели, независимо от внутренней организации данных (реализуется посредством класса **QModelIndex**).

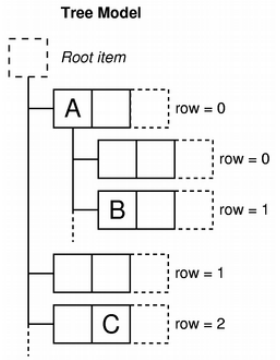
Индекс модели состоит из трех частей: номера строки, номера столбца и внутреннего идентификатора, который позволяет учесть родительский элемент.
Эти три части указываются при получении индекса для конкретной модели.

QModelIndex ind= model()->index(**row, column, parentIndex**);

```
QModelIndex indexA=model->index(0,0, QModelIndex());
QModelIndex indexB=model->index(2,1, QModelIndex());
QModelIndex indexC=model->index(1,0, indexA);
```




```
QModelIndex indexA=model->index(0,0, QModelIndex());
QModelIndex indexB=model->index(1,1, QModelIndex());
QModelIndex indexC=model->index(2,1, QModelIndex());
//индекс родительского элемента задается конструктором по-умолчанию, формируется пустой индекс
```



Специфика индекса модели

Класс **QModelIndex** поддерживает практически любую структуру данных, но для конкретных реализаций имеют смысл:

для списка – индекс строки (*column = 0, parentIndex = QModelIndex()*),
для таблицы – индексы строки и столбца (*parentIndex = QModelIndex()*),
для иерархических структур данных – *индексы относительно родительского элемента*.

Индексы модели могут изменяться в результате действий с данными модели (вставок, удалений, сортировки…), поэтому **запоминать** их для дальнейшего использования **не следует**!
Получить индекс модели для любого элемента данных можно посредством **QAbstractItemModel::index()**
При получении индекса программист может сформировать значения, для которых элемент не существует. При этом возвращается пустой индекс - *QModelIndex()*. проверить существование элемента по возвращенному индексу можно посредством *QModelIndex::isValid()*

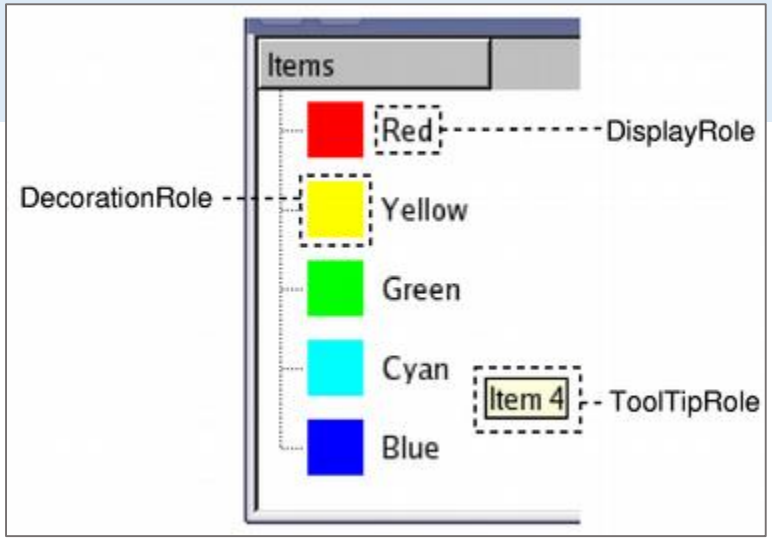
```
// достаем данные по индексу
QModelIndex index = pModel->index( <строка>, <столбец>, QModelIndex());
if(index.isValid()) {
        QVariant val = pModel->data(index);
        int n = val.toInt();
}  else { //если такого элемента нет }
```

Роли данных

Каждый элемент модели описывает какое-то данное (например, для прямоугольника – это координаты вершин, цвет рамки, стиль рамки, толщина рамки и др.). Помимо данных модель может хранить доп. информацию.

Роль используется для:

- указания модели в каком виде представление хочет получить или задать данное (значение цвета или пиктограмма…) в разных ситуациях (при редактировании, при отображении, …)
- задания (получения) сопровождающей это данное информации enum **Qt::ItemDataRole**
- Часть флагов описывают общее назначение ассоциированного с элементом таблицы значения: **Qt::DisplayRole**, **Qt::DecorationRole**, **Qt::EditRole**…
- Роли, описывающие отображение данных: **Qt::FontRole**, **Qt::TextAlignmentRole**…
- Роли специальных возможностей
- Пользовательские роли



Пример.

```
QStringList slist;
slist<<"Russia"<<"USA"<<"Canada"<<"Germany"<<"Great Britain"<<"Japan";
QStandardItemModel* stModel= new QStandardItemModel(slist.size(),1);
for(int i=0; i<stModel->rowCount(); i++) {
        QModelIndex index= stModel->index(i,0);
        QString qstr= slist.at(i);
        stModel->setData(index,qstr,Qt::DisplayRole);
        QPixmap pix(QString("./Flags/%1.png" ).arg(qstr));
        stModel->setData(index,QIcon(pix), Qt::DecorationRole);
        stModel->setData(index,"ToolTip for "+qstr, Qt::ToolTipRole);
}
QListView *listView = new QListView();
listView->setModel(stModel);
listView->setViewMode(QListView::IconMode);
listView->show();
```

Практика

Директория “ModelView” с набором приложений