МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Воронежский государственный технический университет»

Кафедра радиоэлектронных устройств и систем

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторных работ № 4-5
для студентов специальности 11.05.01
«Радиоэлектронные системы и комплексы»
очной формы обучения

Воронеж 2024

УДК 681.3.06(07)
ББК 32.97я7

Составитель А. И. Сукачев

Информационные технологии: методические указания к выполнению лабораторных работ №4-5 для студентов специальности 11.05.01 «Радиоэлектронные системы и комплексы» очной формы обучения / ФГБОУ ВО «Воронежский государственный технический университет»; сост.: А. И. Сукачев. – Воронеж: Изд-во ВГТУ, 2024. – 45 с.

В соответствии с рабочими учебными программами дисциплин приведены описания методов измерений и методик выполнения лабораторных работ, изложены теоретические сведения, лежащие в основе программирования на языке С++. По каждой лабораторной работе в описание включены: цель, основные теоретические сведения, порядок подготовки и проведения работы, перечень положений, которые необходимо отразить в выводах.

Предназначены для студентов специальности 11.05.01 «Радиоэлектронные системы и комплексы» очной формы обучения.

Методические указания подготовлены в электронном виде и содержатся в файле МУ_ИТ_ЛР4-5.pdf.

Ил. 24. Библиогр.: 3 назв.

УДК 681.3.06(07)
ББК 32.97я7

Рецензент – А. В. Останков, д-р техн. наук, профессор кафедры радиотехники ВГТУ

Издается по решению редакционно-издательского совета Воронежского государственного технического университета

1. ЛАБОРАТОРНАЯ РАБОТА № 4
КОНЦЕПЦИЯ «МОДЕЛЬ-ПРЕДСТАВЛЕНИЕ»
В ФРЕЙМВОРКЕ QT

1.1. ОБЩИЕ УКАЗАНИЯ

1.1.1. ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является разработка типового интерфейса.

1.1.2. ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Основным содержанием работы является изучение концепции «модель-представление». В ходе работы идёт ознакомление с приведённым примером приложения, производится анализ используемых технологий. На основе полученных знаний выполняется индивидуальное задание.

2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Qt содержит набор классов представления элементов, которые используют архитектуру модель / представление для управления отношениями между данными и способом их представления пользователю. Разделение функциональных возможностей, введенное этой архитектурой, дает разработчикам большую гибкость для настройки представления элементов и обеспечивает стандартный интерфейс модели, позволяющий использовать широкий спектр источников данных с существующими представлениями элементов. В этом документе мы даем краткое введение в парадигму модели/представления, описываем соответствующие концепции и описываем архитектуру системы представления элементов. Каждый из компонентов в архитектуре объясняется, и приведены примеры, которые показывают, как использовать предоставленные классы.

2.1. АРХИТЕКТУРА МОДЕЛИ / ПРЕДСТАВЛЕНИЯ

Model-View-Controller (MVC) - это шаблон проектирования, происходящий из Smalltalk, который часто используется при создании пользовательских интерфейсов. В шаблонах проектирования Gamma et al. писать:

MVC состоит из трех видов объектов. Модель - это объект приложения, представление-его экранное представление, а контроллер определяет способ, которым пользовательский интерфейс реагирует на пользовательский ввод. До MVC дизайн пользовательского интерфейса имел тенденцию объединять эти объекты вместе. MVC отцепляет их для повышения гибкости и повторного использования.

Если представление и объекты контроллера объединены, результатом является архитектура модель / представление. Это по-прежнему отделяет способ хранения данных от способа их представления пользователю, но обеспечивает более простую структуру, основанную на тех же принципах. Это разделение позволяет отображать одни и те же данные в нескольких различных представ-

лениях, а также реализовывать новые типы представлений, не изменяя базовые структуры данных. Чтобы обеспечить гибкую обработку пользовательского ввода, мы вводим понятие *делегата*. Преимущество наличия делегата в этой структуре заключается в том, что он позволяет настраивать способ отображения и редактирования элементов данных.

Как правило, классы model/view можно разделить на три группы, описанные выше: модели, представления и делегаты. Каждый из этих компонентов определяется *абстрактными* классами, которые предоставляют общие интерфейсы и, в некоторых случаях, реализации объектов по умолчанию. Абстрактные классы предназначены для подклассов, чтобы обеспечить полный набор функциональных возможностей, ожидаемых от других компонентов; это также позволяет писать специализированные компоненты.

Модели, представления и делегаты взаимодействуют друг с другом с помощью *сигналов и слотов*:

- Сигналы от модели информируют представление об изменениях данных, хранящихся в источнике данных.
- Сигналы из представления предоставляют информацию о взаимодействии пользователя с отображаемыми элементами.
- Сигналы от делегата используются во время редактирования для сообщения модели и представления о состоянии редактора.

2.2. МОДЕЛИ

Все модели элементов основаны на классе QAbstractItemModel. Этот класс определяет интерфейс, используемый представлениями и делегатами для доступа к данным. Сами данные не обязательно должны храниться в модели; они могут храниться в структуре данных или репозитории, предоставляемом отдельным классом, файлом, базой данных или каким-либо другим компонентом приложения.

Основные понятия, окружающие модели, представлены в разделе о классах моделей.

QAbstractItemModel предоставляет достаточно гибкий интерфейс для обработки представлений, представляющих данные в виде таблиц, списков и деревьев. Однако при реализации новых моделей для структур данных типа списков и таблиц классы QAbstractListModel и QAbstractTableModel являются лучшими начальными точками, поскольку они обеспечивают соответствующие реализации общих функций по умолчанию. Каждый из этих классов может быть разделен на подклассы, чтобы обеспечить модели, которые поддерживают специализированные виды списков и таблиц.

Процесс создания подклассов моделей рассматривается в разделе О создании новых моделей.

Qt предоставляет некоторые готовые модели, которые могут быть использованы для обработки элементов данных:

- QStringListModel используется для хранения простого списка элементов QString.
- QStandardItemModel управляет более сложными древовидными структурами элементов, каждый из которых может содержать произвольные данные.
- QFileSystemModel предоставляет информацию о файлах и каталогах в локальной файловой системе.
- QSqlQueryModel, QSqlTableModel и QSqlRelationalTableModel используются для доступа к базам данных с помощью соглашений модели/представления.

Если эти стандартные модели не соответствуют вашим требованиям , вы можете создать подкласс QAbstractItemModel, QAbstractListModel или QAbstractTableModel для создания собственных пользовательских моделей.

2.3. ДВЕ МОДЕЛИ, ВКЛЮЧЕННЫЕ В QT

Две стандартные модели, предоставляемые Qt, являются QStandardItemModel и QFileSystemModel. QStandardItemModel - это универсальная модель, которая может использоваться для представления различных структур данных, необходимых для представления списка, таблицы и дерева. Эта модель также содержит элементы данных. QFileSystemModel - это модель, которая поддерживает информацию о содержимом каталога. В результате он не содержит никаких элементов данных сам по себе, а просто представляет файлы и каталоги в локальной файловой системе.

QFileSystemModel предоставляет готовую к использованию модель для экспериментов и может быть легко настроена для использования существующих данных. Используя эту модель, мы можем показать, как настроить модель для использования с готовыми представлениями, а также изучить, как управлять данными с помощью индексов модели.

2.4. ИСПОЛЬЗОВАНИЕ ПРЕДСТАВЛЕНИЙ С СУЩЕСТВУЮЩЕЙ МОДЕЛЬЮ

Классы QListView и QTreeView являются наиболее подходящими представлениями для использования с QFileSystemModel . В Примере, представленном ниже, содержимое каталога отображается в виде дерева рядом с той же информацией в виде списка. Представления совместно используют выбор пользователя, так что выбранные элементы выделяются в обоих представлениях.

Мы настраиваем QFileSystemModel так, чтобы он был готов к использованию, и создаем некоторые представления для отображения содержимого каталога. Это показывает самый простой способ использования модели. Построение и использование модели осуществляется изнутри одной main() функции:

```
int main(int argc, char *argv[])
```

```
{
    QApplication app(argc, argv);
    QSplitter *splitter = new QSplitter;

    QFileSystemModel *model = new QFileSystemModel;
    model->setRootPath(QDir::currentPath());
```

Модель настроена на использование данных из определенной файловой системы. Вызов метода setRootPath () указывает модели, какой диск в файловой системе следует предоставить для просмотра.

Мы создаем два представления, чтобы можно было изучить элементы, содержащиеся в модели, двумя различными способами:

```
    QTreeView *tree = new QTreeView(splitter);
    tree->setModel(model);
    tree->setRootIndex(model->index(QDir::currentPath()));

    QListView *list = new QListView(splitter);
    list->setModel(model);
    list->setRootIndex(model->index(QDir::currentPath()));
```

Представления построены таким же образом, как и другие виджеты. Настройка представления для отображения элементов в модели-это просто вызов его функции setModel () с моделью каталога в качестве аргумента. Мы фильтруем данные, предоставляемые моделью, вызывая функцию setRootIndex () на каждом представлении, передавая подходящий *индекс модели* из модели файловой системы для текущего каталога.

index()Функция, используемая в этом случае, уникальна для QFileSystemModel ; мы предоставляем ей каталог, и он возвращает индекс модели. Индексы моделей обсуждаются в классах моделей .

Остальная часть функции просто отображает представления в виджете splitter и запускает цикл событий приложения:

```
    splitter->setWindowTitle("Two views onto the same file system model");
    splitter->show();
    return app.exec();
}
```

В приведенном выше примере мы забыли упомянуть, как обрабатывать выборки элементов. Этот вопрос более подробно рассматривается в разделе об обработке выборок в представлениях номенклатур .

2.5. КЛАСС МОДЕЛЕЙ

Перед изучением того, как обрабатываются выбранные элементы, может оказаться полезным изучить понятия, используемые в рамках модели/представления.

2.5.1. ОСНОВНОЕ ПОНЯТИЕ

В архитектуре модель / представление модель предоставляет стандартный интерфейс, который представления и делегаты используют для доступа к данным. В Qt стандартный интерфейс определяется классом QAbstractItemModel. Независимо от того, как элементы данных хранятся в любой базовой структуре данных, все подклассы QAbstractItemModel представляют данные в виде иерархической структуры, содержащей таблицы элементов. Представления используют это *Соглашение* для доступа к элементам данных в модели, но они не ограничены способом представления этой информации пользователю.

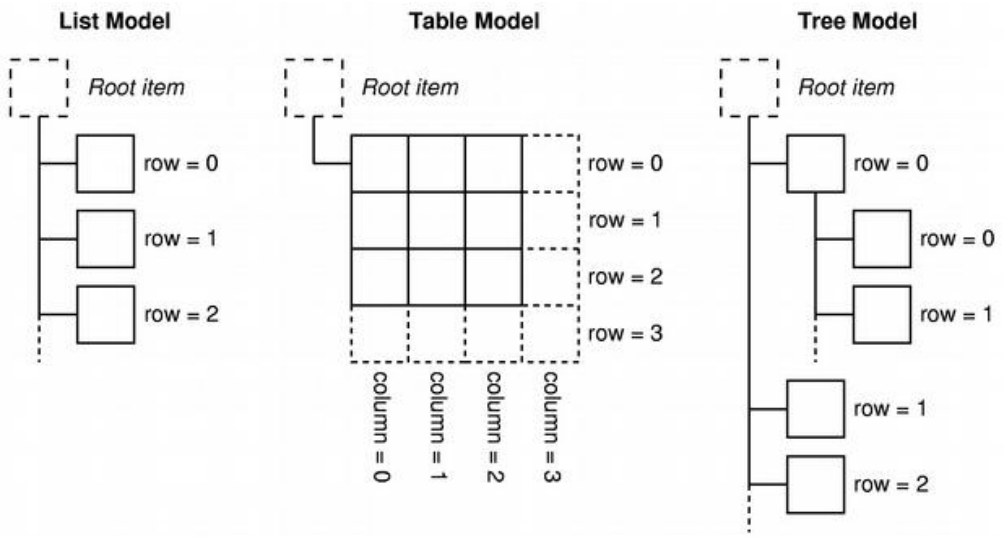

Рис. 1. Различные типы моделей

Модели также уведомляют любые вложенные представления об изменениях данных через механизм сигналов и слотов.

В этом разделе описываются некоторые основные понятия, которые являются центральными для способа доступа к элементам данных другими компонентами через класс модели. Более продвинутые концепции обсуждаются в последующих разделах.

2.5.2. СТРОКИ И СТОЛБЦЫ

В своей самой основной форме модель может быть доступна в виде простой таблицы, в которой элементы расположены по их номерам строк и столбцов. *Это не означает, что базовые части данных хранятся в структуре массива*; использование номеров строк и столбцов - это только соглашение, позволяющее компонентам взаимодействовать друг с другом. Мы можем получить информацию о любом заданном элементе, указав номера его строк и столбцов в модели, и получим индекс, представляющий этот элемент:

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

Модели, которые предоставляют интерфейсы к простым, одноуровневым структурам данных, таким как списки и таблицы, не нуждаются в предоставлении какой-либо другой информации, но, как показывает приведенный выше код, нам нужно предоставить дополнительную информацию при получении индекса модели.

На рис. 2 показано представление базовой табличной модели, в которой каждый элемент расположен по паре номеров строк и столбцов.

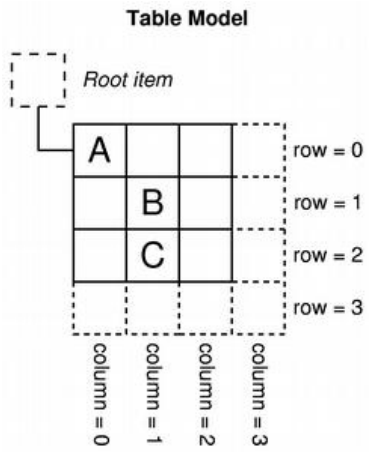

Рис. 2. Базовая табличная модель

Мы получаем индекс модели, который ссылается на элемент данных, передавая соответствующие номера строк и столбцов в модель.

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

На элементы верхнего уровня в модели всегда ссылаются, указывая QModelIndex()в качестве их родительского элемента. Это обсуждается в следующем разделе

2.5.3. РОДИТЕЛИ ПРЕДМЕТОВ

Табличный интерфейс к данным элементов, предоставляемый моделями, идеально подходит при использовании данных в табличном или списковом представлении; система номеров строк и столбцов точно соответствует способу отображения элементов в представлениях. Однако такие структуры, как древовидные представления, требуют от модели предоставления более гибкого интерфейса для входящих в нее элементов. В результате каждый элемент также может быть родительским для другой таблицы элементов, во многом так же, как элемент верхнего уровня в представлении дерева может содержать другой список элементов.

При запросе индекса для элемента модели мы должны предоставить некоторую информацию о родителе элемента. Вне модели, единственный способ ссылаться на элемент - это через индекс модели, поэтому родительский индекс модели также должен быть задан:

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

На рис. 3 показано представление модели дерева, в которой на каждый элемент ссылается родитель, номер строки и номер столбца.

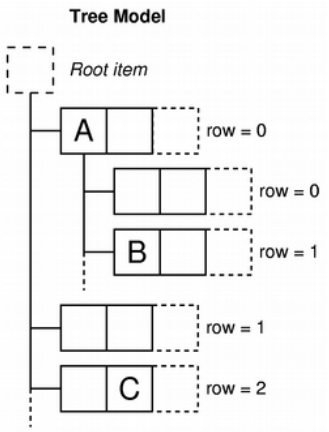

Рис. 3. Модель дерева

Элементы "A" и "c" представлены в модели как родные братья верхнего уровня:

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

Пункт "A" имеет ряд детей. Модельный индекс для пункта "B" получается со следующим кодом:

ModelIndex indexB = model->index(1, 0, indexA);

Краткие сведения:

- Индексы моделей предоставляют представления и делегируют сведения о расположении элементов, предоставляемых моделями таким образом, что это не зависит от каких-либо базовых структур данных.
- Элементы ссылаются на их номера строк и столбцов, а также на индекс модели их родительских элементов.
- Индексы моделей создаются моделями по запросу других компонентов, таких как представления и делегаты.

- Если при запросе индекса с помощью функции index () для родительского элемента указывается допустимый индекс модели , то возвращаемый индекс ссылается на элемент, расположенный под этим родительским элементом в модели. Полученный индекс относится к дочернему элементу этого элемента.
- Если при запросе индекса с помощью функции index () для родительского элемента указывается недопустимый индекс модели , возвращаемый индекс ссылается на элемент верхнего уровня в модели.
- Роль различает различные виды данных, связанных с элементом.

Использование индексов моделей

Чтобы продемонстрировать, как данные могут быть получены из модели, используя индексы модели, мы устанавливаем QFileSystemModel без представления и отображаем имена файлов и каталогов в виджете. Хотя это не показывает обычный способ использования модели, он демонстрирует соглашения, используемые моделями при работе с модельными индексами.

Загрузка QFileSystemModel является асинхронной для минимизации использования системных ресурсов. Мы должны учитывать это при работе с этой моделью.

Мы строим модель файловой системы следующим образом:

```
QFileSystemModel *model = new QFileSystemModel;
   connect(model, &QFileSystemModel::directoryLoaded, [model](const QString &directory) {
      QModelIndex parentIndex = model->index(directory);
      int numRows = model->rowCount(parentIndex);
   });
   model->setRootPath(QDir::currentPath);
```

В этом случае мы начнем с настройки qfilesystemmodel по умолчанию . Мы подключаем его к лямбде, в которой мы получим Родительский индекс, используя конкретную реализацию index (), предоставленную этой моделью. В лямбде мы подсчитываем количество строк в модели, используя функцию rowCount (). Наконец, мы устанавливаем корневой путь QFileSystemModel, поэтому он начинает загрузку данных и запускает лямбду.

Для простоты нас интересуют только элементы в первой колонке модели. Мы изучаем каждую строку по очереди, получая индекс модели для первого элемента в каждой строке, и считываем данные, сохраненные для этого элемента в модели.

```
for (int row = 0; row < numRows; ++row) {
   QModelIndex index = model->index(row, 0, parentIndex);
```

Чтобы получить индекс модели, мы указываем номер строки, номер столбца (ноль для первого столбца) и соответствующий индекс модели для родительского элемента всех элементов, которые мы хотим. Текст, сохраненный в каждом элементе, извлекается с помощью функции Data() модели. Мы задаем индекс модели и роль DisplayRole для получения данных для элемента в виде строки.

```
QString text = model->data(index, Qt::DisplayRole).toString();
    // Display the text in a widget.

}
```

В приведенном выше примере демонстрируются основные принципы, используемые для извлечения данных из модели:

- Размеры модели можно найти с помощью rowCount() и columnCount(). Эти функции обычно требуют указания индекса родительской модели.
- Индексы модели используются для доступа к элементам модели. Для указания элемента необходимы строка, столбец и индекс родительской модели.
- Чтобы получить доступ к элементам верхнего уровня в модели, укажите нулевой индекс модели в качестве родительского индекса C.QModelIndex()
- Элементы содержат данные для различных ролей. Чтобы получить данные для конкретной роли, в модель необходимо ввести как индекс модели, так и роль.

2.6. КЛАСС ПРЕДСТАВЛЕНИЯ

2.6.1. КОНЦЕПЦИЯ

В архитектуре модель/представление получает элементы данных из модели и представляет их пользователю. Способ представления данных не обязательно должен напоминать представление данных, предоставляемое моделью, и может *полностью отличаться* от базовой структуры данных, используемой для хранения элементов данных.

Разделение содержания и представления достигается за счет использования интерфейса стандартной модели , предоставляемого QAbstractItemModel, интерфейса стандартного представления, предоставляемого QAbstractItemView, и использования индексов модели, которые представляют элементы данных в общем виде. Представления обычно управляют общим расположением данных, полученных из моделей. Они могут сами отображать отдельные элементы данных или использовать делегаты для обработки как объектов визуализации, так и объектов редактирования.

Помимо представления данных, представления управляют навигацией между элементами и некоторыми аспектами выбора элементов. В представлениях также реализованы основные функции пользовательского интерфейса, такие как контекстные меню и перетаскивание. Представление может предоставлять средства редактирования по умолчанию для элементов или работать с делегатом для предоставления пользовательского редактора.

Представление может быть построено без модели, но модель должна быть предоставлена, прежде чем она сможет отображать полезную информацию. Представления отслеживают элементы, выбранные пользователем с помощью выборок, которые могут быть сохранены отдельно для каждого представления или совместно использоваться несколькими представлениями.

В некоторых представлениях, таких как QTableView и QTreeView , отображаются заголовки, а также элементы. Они также реализуются классом представления QHeaderView . Заголовки обычно обращаются к той же модели, что и содержащее их представление. Они извлекают данные из модели с помощью функции QAbstractItemModel:: headerData () и обычно отображают информацию заголовка в виде метки. Новые заголовки могут быть разделены на подклассы из класса QHeaderView, чтобы обеспечить более специализированные метки для представлений.

2.6.2. ИСПОЛЬЗОВАНИЕ СУЩЕСТВУЮЩЕГО ПРЕДСТАВЛЕНИЯ

Qt предоставляет три готовых к использованию класса представления, которые представляют данные из моделей способами, знакомыми большинству пользователей. QListView может отображать элементы из модели в виде простого списка или в виде классического представления значков. QTreeView отображает элементы из модели в виде иерархии списков, позволяя глубоко вложенные структуры, которые будут представлены в компактном виде. QTableView представляет элементы из модели в виде таблицы так же, как макет приложения электронной таблицы.

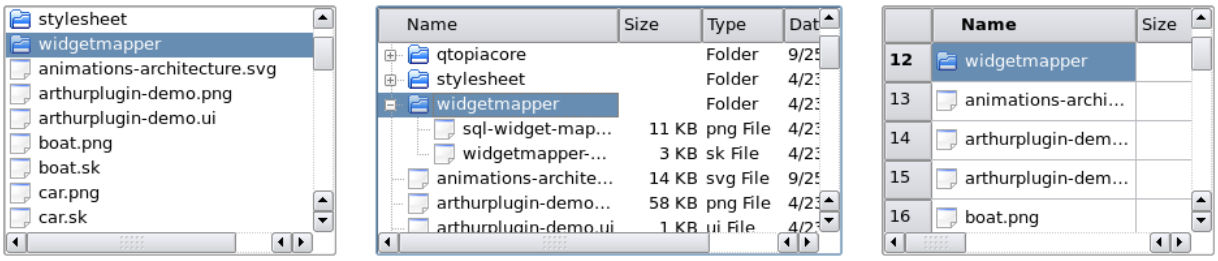

Рис. 4. Окно Qt со списком классов

Поведение по умолчанию стандартных представлений, показанных выше, должно быть достаточным для большинства приложений. Они предоставляют основные средства редактирования и могут быть настроены в соответствии с потребностями более специализированных пользовательских интерфейсов.

2.6.3. ИСПОЛЬЗОВАНИЕ МОДЕЛИ

Мы берем модель строкового списка, которую мы создали в качестве примера модели, устанавливаем ее с некоторыми данными и строим представление для отображения содержимого модели. Все это может быть выполнено в рамках одной функции:

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

// Unindented for quoting purposes:
QStringList numbers;
numbers << "One" << "Two" << "Three" << "Four" << "Five";

QAbstractItemModel *model = new StringListModel(numbers);
```

Обратите внимание, что StringListModel объявлен как QAbstractItemModel. Это позволяет нам использовать абстрактный интерфейс к модели и гарантирует, что код все еще работает, даже если мы заменим модель списка строк другой моделью.

Представление списка, предоставляемое QListView, достаточно для представления элементов в модели строкового списка. Мы построим представление и настроим модель, используя следующие строки кода:

```
QListView *view = new QListView;
view->setModel(model);
```

Вид отображается в обычном режиме:

```
 view->show();
   return app.exec();
}
}
```

Представление отображает содержимое модели, получая доступ к данным через интерфейс модели. Когда пользователь пытается изменить элемент, представление использует делегат по умолчанию для предоставления виджета редактора.

На приведенном ниже рисунке показано, как QListView представляет данные в модели строкового списка.

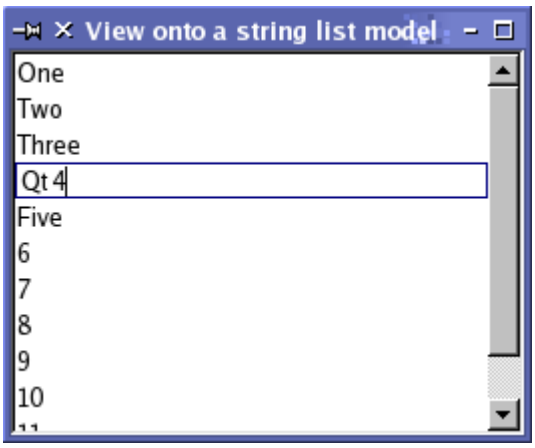

Рис. 5. Представление данных в модели строкового списка

Поскольку модель является редактируемой, представление автоматически позволяет редактировать каждый элемент списка с помощью делегата по умолчанию.

2.7. ДЕЛЕГИРОВАНИЕ КЛАССОВ

2.7.1. КОНЦЕПЦИЯ

В отличие от шаблона Model-View-Controller, дизайн model/view не включает полностью отдельный компонент для управления взаимодействием с пользователем. Как правило, представление отвечает за представление данных модели пользователю, а также за обработку пользовательского ввода. Чтобы обеспечить некоторую гибкость в способах получения этих входных данных, взаимодействие осуществляется делегатами. Эти компоненты предоставляют возможности ввода и также отвечают за отображение отдельных элементов в некоторых представлениях. Стандартный интерфейс для управления делегатами определяется в классе QAbstractItemDelegate.

Ожидается, что делегаты смогут самостоятельно отображать свое содержимое, реализуя функции paint() и sizeHint (). Однако простые делегаты на основе виджетов могут создавать подклассы QStyledItemDelegate вместо QAbstractItemDelegate и использовать преимущества реализаций этих функций по умолчанию.

Редакторы для делегатов могут быть реализованы либо с помощью виджетов для управления процессом редактирования, либо путем непосредственной обработки событий. Первый подход рассматривается далее в этом разделе, и он также показан в Примере делегата Spin Box.

В Примере Pixelator показано, как создать пользовательский делегат, который выполняет специализированную визуализацию для представления таблицы.

2.7.2. ИСПОЛЬЗОВАНИЕ СУЩЕСТВУЮЩЕГО ДЕЛЕГАТА

Стандартные представления, предоставляемые вместе с Qt, используют экземпляры QStyledItemDelegate для предоставления средств редактирования.

Эта стандартная реализация интерфейса делегата отображает элементы в обычном стиле для каждого из стандартных представлений: QListView, QTableView и QTreeView.

Все стандартные роли обрабатываются делегатом по умолчанию, используемым стандартными представлениями. Способ их интерпретации описан в документации QStyledItemDelegate.

Делегат, используемый представлением, возвращается функцией itemDelegate(). Функция setItemDelegate() позволяет установить пользовательский делегат для стандартного представления, и эту функцию необходимо использовать при настройке делегата для пользовательского представления.

2.7.3. ПРОСТОЙ ДЕЛЕГАТ

Делегат, реализованный здесь, использует QSpinBox для предоставления средств редактирования и в основном предназначен для использования с моделями, которые отображают целые числа. Хотя для этой цели мы создали пользовательскую целочисленную табличную модель, вместо нее можно было бы легко использовать QStandardItemModel, поскольку пользовательский делегат управляет вводом данных. Мы создаем табличное представление для отображения содержимого модели, и это будет использовать пользовательский делегат для редактирования.

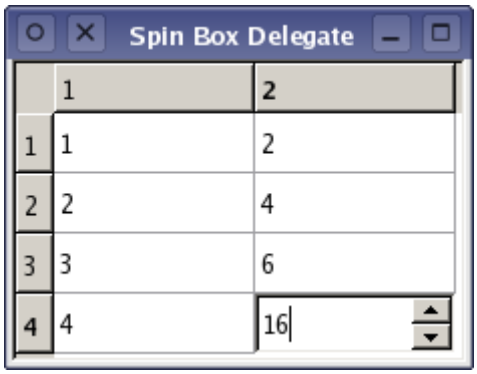

Рис. 6. Табличное представление данных

Мы создаем подкласс делегата от QStyledItemDelegate, потому что не хотим писать пользовательские функции отображения. Однако мы все равно должны предоставить функции для управления виджетом редактора:

```
class SpinBoxDelegate : public QStyledItemDelegate
{
   Q_OBJECT

public:
   SpinBoxDelegate(QObject *parent = nullptr);
```

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

    void setEditorData(QWidget *editor, const QModelIndex &index) const
override;
    void setModelData(QWidget *editor, QAbstractItemModel *model,
                      const QModelIndex &index) const override;

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

Обратите внимание, что никакие виджеты редактора не настраиваются при создании делегата. Мы создаем виджет редактора только тогда, когда это необходимо.

2.7.4. ОТПРАВКА ДАННЫХ В МОДЕЛЬ

Когда пользователь закончит редактирование значения в поле вращений, представление попросит делегата сохранить измененное 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);
}
```

Поскольку представление управляет виджетами редактора для делегата, нам нужно только обновить модель с содержимым предоставленного редактора. В этом случае мы гарантируем, что спин-бокс обновлен, и обновляем модель со значением, которое она содержит, используя указанный индекс.

Стандартный класс QStyledItemDelegate информирует представление о завершении редактирования, посылая сигнал closeEditor (). Представление гарантирует, что виджет редактор будет закрыт и уничтожен. В этом примере мы предоставляем только простые средства редактирования, поэтому нам никогда не нужно излучать этот сигнал.

Все операции с данными выполняются через интерфейс, предоставляемый QAbstractItemModel. Это делает делегат в основном независимым от типа данных, которыми он управляет, но некоторые предположения должны быть сделаны для использования определенных типов виджетов редактора. В этом примере мы предположили, что модель всегда содержит целочисленные значения, но мы все еще можем использовать этот делегат с различными типами моделей, потому что QVariant предоставляет разумные значения по умолчанию для неожиданных данных.

2.8. ЛАБОРАТОРНЫЕ ЗАДАНИЯ И МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ИХ ВЫПОЛНЕНИЮ

Создать объект класса QStandardItemModel под названием modelTree. В заголовочном файле widget.h объявить закрытые переменные QStandardItemModel * modelTree; QTreeView * tree;

Вставить в модель 1 колонкунумеруя с 0, с помощью метода insertColumns, применили модель к объекту tree класса QTreeView

```
hlayl->addStretch();

tree = new QTreeView;
modelTree= new QStandardItemModel;
modelTree->insertColumns(0,1);
tree->setModel(modelTree);
```

Рис. 7. Пример кода

Объявить в заголовочном файле treeobj.h и реализовали открытый метод getStatus в исходном файле treeobj.cpp. Он нужен, чтобы определить создает пользователь ребенка или взрослого.

```
void TreeObj::setFam(QString str1)
{
    lin1->setText(str1);
}
QString TreeObj::getFam()
{
    return lin1->text();
}

void TreeObj::setName(QString str2)
{
    lin2->setText(str2);
}
QString TreeObj::getName()
{
    return lin2->text();
}

void TreeObj::setOtche(QString str3)
{
    lin3->setText(str3);
}
QString TreeObj::getOtche()
{
    return lin3->text();
}

bool TreeObj::getStatus()
{
    if (qcb->currentText()=="parent")
    {
        return 1;
    }
    else return 0;

}
```

Рис. 8. Пример кода

В функции addTree реализовать добавление данных в иерархическое дерево. При этом если ни одна запись не выделена, создать ребёнка невозможно. после нажатия кнопки «+» вызывается модальное окно, в котором пользователь вводит фамилию и выбирает в выпадающем списке пункт «parent» или «child» (программа определяет статус «parent» или «child» посредством метода getStatus(), принадлежащего классу TreeObj). При выборе пункта «parent» в modelTree добавляется строка, куда сохраняются введённые пользователем данные и ID (если пользователь не выделил какой-либо элемент или выделил элемент каталога верхнего уровня, строка добавляется в каталог верхнего уровня; в противном случае она добавляется в каталог более высокого уровня, чем тот, на котором находится выделенный элемент).

При выборе пункта «child» необходимо предварительно выделить какой-либо элемент: тогда для выделенного элемента создаётся подкаталог, куда добавляются строки с введёнными пользователем данными и ID. ID формируется следующим образом: для элементов каталога верхнего уровня он равен количеству строк данного каталога, для дочерних элементов он представляет с собой строку, в которой через дефис записан ID элемента-родителя и количество строк подкаталога. Если в модели нет строк (modelTree–>rowCount()==0), пункт «child» недоступен.

```
void Widget::addTree()
{
    bool ch;
if (tree->currentIndex().row()==-1)
{
  ch = 0;
}
else ch =1;
    TreeObj* add = new TreeObj(ch);
    if (add->exec()==QDialog::Accepted)
    {
        if (add->getStatus()==1)
        {
            modelTree->insertRow(modelTree->rowCount());
            modelTree->setData(modelTree->index(modelTree->rowCount()-1,0), add->getName());
            modelTree->setData(modelTree->index(modelTree->rowCount()-1,1), modelTree->rowCount());
        }
        else
        {
           modelTree->itemFromIndex(tree->currentIndex())->setChild(modelTree->rowCount(tree->currentIndex()),
           QString str= modelTree->data(modelTree->index(tree->currentIndex().row(),1,tree->currentIndex().par
          modelTree->itemFromIndex(tree->currentIndex())->setChild(modelTree->rowCount(tree->currentIndex())-1
        }
        delete add;
    }
else delete add;
}
```

Рис. 9. Пример кода

```
unt(tree->currentIndex()),0, new QStandardItem(add->getName()));
1,tree->currentIndex().parent())).toString()+"-"+QString::number(modelTree->rowCount(tree->currentIndex()));
nt(tree->currentIndex())-1,1, new QStandardItem(str));
```

Рис. 10. Пример кода

Реализоват в исходном файлewidget.cpp открытый метод delTree, который используется для удаления записей из дерева.Объявили в заголовочном файле widget.h.

```
}

void Widget::delTree()
{
    modelTree->removeRow(tree->currentIndex().row(),tree->currentIndex().parent());
}
```

Рис. 11. Пример кода

Дополнили слот editTree:

```
void Widget::editTree()
{
    if (tree->currentIndex().row()!=-1)
    {
    TreeObj* edit = new TreeObj("",modelTree->data(modelTree->index(tree->currentIndex().row(),
        tree->currentIndex().column()),Qt::DisplayRole).toString(),"");
    if (edit->exec()==QDialog::Accepted)
    {
        modelTree->setData(modelTree->index(tree->currentIndex().row(),tree->currentIndex().column(),tree->cur
        delete edit;
    }
    else delete edit;
    }
}
```

Рис. 12. Пример кода

Вид иерархического дерева:

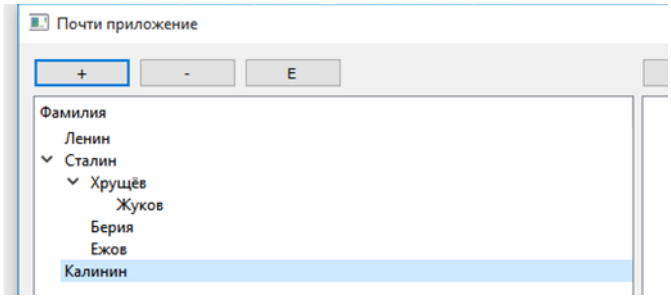

Рис. 13. Пример работы программы

2.9. УКАЗАНИЯ ПО ОФОРМЛЕНИЮ ОТЧЕТА

Отчет должен содержать название и цель работы, краткие теоретические сведения, а также код с комментариями.

3. ЛАБОРАТОРНАЯ РАБОТА № 5
РАЗРАБОТКА СОБСТВЕННОЙ МОДЕЛИ КОНЦЕПЦИИ МОДЕЛЬ-ПРЕДСТАВЛЕНИЕ

3.1. ОБЩИЕ УКАЗАНИЯ

3.1.1. ЦЕЛЬ РАБОТЫ

Целью работы является разработка типового интерфейса.

3.1.2. ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Основным содержанием работы является разработка собственной модели концепции «модель-представление». В ходе работы идёт ознакомление с приведённым примером приложения, производится анализ используемых технологий. На основе полученных знаний выполняется индивидуальное задание.

3.2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

3.2.1. ПРОГРАММИРОВАНИЕ МОДЕЛИ/ПРЕДСТАВЛЕНИЯ

3.2.2. ВВЕДЕНИЕ

Qt содержит набор классов представления элементов, которые используют архитектуру модель / представление для управления отношениями между данными и способом их представления пользователю. Разделение функциональных возможностей, введенное этой архитектурой, дает разработчикам большую гибкость для настройки представления элементов и обеспечивает стандартный интерфейс модели, позволяющий использовать широкий спектр источников данных с существующими представлениями элементов. В этом документе мы даем краткое введение в парадигму модели/представления, описываем соответствующие концепции и описываем архитектуру системы представления элементов. Каждый из компонентов в архитектуре объясняется, и приведены примеры, которые показывают, как использовать предоставленные классы.

3.2.3. АРХИТЕКТУРА МОДЕЛИ/ПРЕДСТАВЛЕНИЯ

Model-View-Controller (MVC) - это шаблон проектирования, происходящий из Smalltalk, который часто используется при создании пользовательских интерфейсов. В шаблонах проектирования Gamma et al. писать:

MVC состоит из трех видов объектов. Модель - это объект приложения, представление-его экранное представление, а контроллер определяет способ, которым пользовательский интерфейс реагирует на пользовательский ввод. До MVC дизайн пользовательского интерфейса имел тенденцию объединять эти объекты вместе. MVC отцепляет их для повышения гибкости и повторного использования.

Если представление и объекты контроллера объединены, результатом является архитектура модель / представление. Это по-прежнему отделяет способ хранения данных от способа их представления пользователю, но обеспечивает более простую структуру, основанную на тех же принципах. Это разделение позволяет отображать одни и те же данные в нескольких различных представлениях, а также реализовывать новые типы представлений, не изменяя базовые структуры данных. Чтобы обеспечить гибкую обработку пользовательского ввода, мы вводим понятие *делегата*. Преимущество наличия делегата в этой структуре заключается в том, что он позволяет настраивать способ отображения и редактирования элементов данных.

Модель взаимодействует с источником данных, обеспечивая *интерфейс* для других компонентов в архитектуре. Характер связи зависит от типа источника данных и способа реализации модели. На рис. 14 представлена схема взаимодействия делегата и модели.

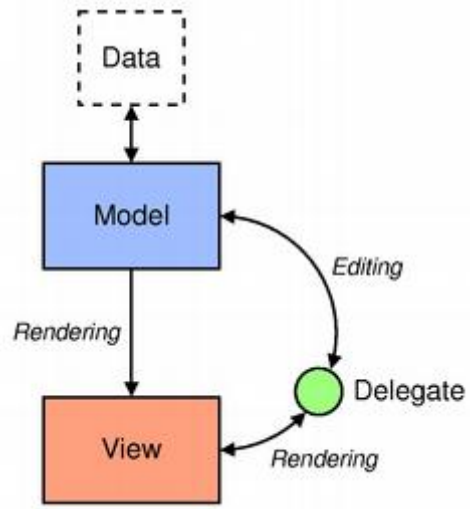

Рис. 14. Схема взаимодействия модели и делегата

Представление получает *индексы модели* из модели; это ссылки на элементы данных. Предоставляя модели индексы, представление может извлекать элементы данных из источника данных.

В стандартных представлениях *делегат* отображает элементы данных. При редактировании элемента делегат взаимодействует с моделью напрямую, используя индексы модели.

Как правило, классы model/view можно разделить на три группы, описанные выше: модели, представления и делегаты. Каждый из этих компонентов определяется *абстрактными* классами, которые предоставляют общие интерфейсы и, в некоторых случаях, реализации объектов по умолчанию. Абстрактные классы предназначены для подклассов, чтобы обеспечить полный набор функциональных возможностей, ожидаемых от других компонентов; это также позволяет писать специализированные компоненты.

Модели, представления и делегаты взаимодействуют друг с другом с помощью *сигналов и слотов*:

- Сигналы от модели информируют представление об изменениях данных, хранящихся в источнике данных.
- Сигналы из представления предоставляют информацию о взаимодействии пользователя с отображаемыми элементами.
- Сигналы от делегата используются во время редактирования для сообщения модели и представления о состоянии редактора.

3.2.4. МОДЕЛИ

Все модели элементов основаны на классе QAbstractItemModel. Этот класс определяет интерфейс, используемый представлениями и делегатами для доступа к данным. Сами данные не обязательно должны храниться в модели; они могут храниться в структуре данных или репозитории, предоставляемом отдельным классом, файлом, базой данных или каким-либо другим компонентом приложения.

Основные понятия, окружающие модели, представлены в разделе о классах моделей.

QAbstractItemModel предоставляет достаточно гибкий интерфейс для обработки представлений, представляющих данные в виде таблиц, списков и деревьев. Однако при реализации новых моделей для структур данных типа списков и таблиц классы QAbstractListModel и QAbstractTableModel являются лучшими начальными точками, поскольку они обеспечивают соответствующие реализации общих функций по умолчанию. Каждый из этих классов может быть разделен на подклассы, чтобы обеспечить модели, которые поддерживают специализированные виды списков и таблиц.

Процесс создания подклассов моделей рассматривается в разделе О создании новых моделей.

Qt предоставляет некоторые готовые модели, которые могут быть использованы для обработки элементов данных:

- QStringListModel используется для хранения простого списка элементов QString.

- QStandardItemModel управляет более сложными древовидными структурами элементов, каждый из которых может содержать произвольные данные.
- QFileSystemModel предоставляет информацию о файлах и каталогах в локальной файловой системе.
- QSqlQueryModel, QSqlTableModel и QSqlRelationalTableModel используются для доступа к базам данных с помощью соглашений модели/представления.

Если эти стандартные модели не соответствуют вашим требованиям , вы можете создать подкласс QAbstractItemModel, QAbstractListModel или QAbstractTableModel для создания собственных пользовательских моделей.

3.2.5. ИСПОЛЬЗОВАНИЕ МОДЕЛЕЙ И ВИДОВ

В следующих разделах объясняется, как использовать шаблон модель/представление в Qt. Каждый раздел содержит пример и сопровождается разделом, показывающим, как создавать новые компоненты.

3.2.6. ДВЕ МОДЕЛИ, ВКЛЮЧЕННЫЕ В QT

Две стандартные модели, предоставляемые Qt, являются QStandardItemModel и QFileSystemModel. QStandardItemModel - это универсальная модель, которая может использоваться для представления различных структур данных, необходимых для представления списка, таблицы и дерева. Эта модель также содержит элементы данных. QFileSystemModel - это модель, которая поддерживает информацию о содержимом каталога. В результате он не содержит никаких элементов данных сам по себе, а просто представляет файлы и каталоги в локальной файловой системе.

QFileSystemModel предоставляет готовую к использованию модель для экспериментов и может быть легко настроена для использования существующих данных. Используя эту модель, мы можем показать, как настроить модель для использования с готовыми представлениями, а также изучить, как управлять данными с помощью индексов модели.

3.2.7. ИСПОЛЬЗОВАНИЕ ПРЕДСТАВЛЕНИЙ С СУЩЕСТВУЮЩЕЙ МОДЕЛЬЮ

Классы QListView и QTreeView являются наиболее подходящими представлениями для использования с QFileSystemModel. В Примере, представленном ниже, содержимое каталога отображается в виде дерева рядом с той же информацией в виде списка. Представления совместно используют выбор пользователя, так что выбранные элементы выделяются в обоих представлениях.

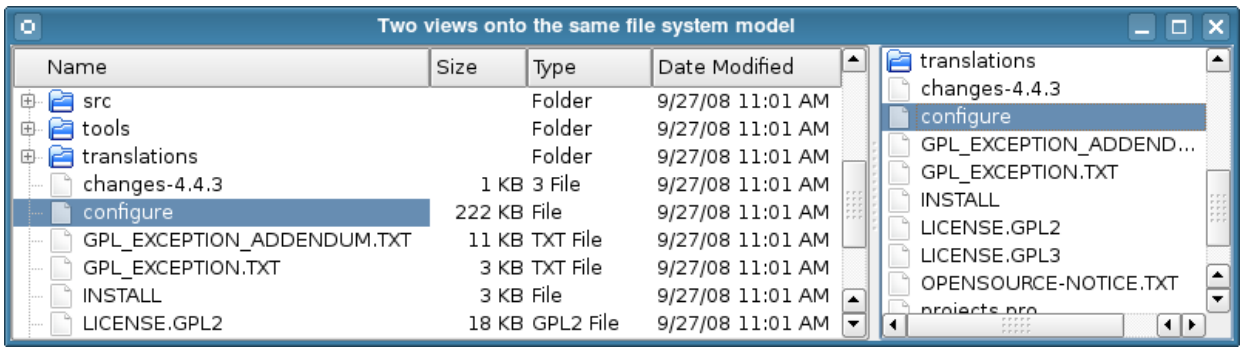

Рис. 15. Содержимое проекта

Мы настраиваем QFileSystemModel так, чтобы он был готов к использованию, и создаем некоторые представления для отображения содержимого каталога. Это показывает самый простой способ использования модели. Построение и использование модели осуществляется изнутри одной main() функции:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QSplitter *splitter = new QSplitter;

    QFileSystemModel *model = new QFileSystemModel;
    model->setRootPath(QDir::currentPath());
```

Модель настроена на использование данных из определенной файловой системы. Вызов метода setRootPath() указывает модели, какой диск в файловой системе следует предоставить для просмотра.

Мы создаем два представления, чтобы можно было изучить элементы, содержащиеся в модели, двумя различными способами:

```
QTreeView *tree = new QTreeView(splitter);
    tree->setModel(model);
    tree->setRootIndex(model->index(QDir::currentPath()));

    QListView *list = new QListView(splitter);
    list->setModel(model);
    list->setRootIndex(model->index(QDir::currentPath()));
```

Представления построены таким же образом, как и другие виджеты. Настройка представления для отображения элементов в модели-это просто вызов его функции setModel() с моделью каталога в качестве аргумента. Мы фильтруем данные, предоставляемые моделью, вызывая функцию setRootIndex() на каждом представлении, передавая подходящий *индекс модели* из модели файловой системы для текущего каталога.

index() - функция, используемая в этом случае, уникальна для QFileSystemModel; мы предоставляем ей каталог, и он возвращает индекс модели. Индексы моделей обсуждаются в классах моделей.

Остальная часть функции просто отображает представления в виджете splitter и запускает цикл событий приложения:

```
splitter->setWindowTitle("Two views onto the same file system model");
   splitter->show();
   return app.exec();
}
```

В приведенном выше примере мы забыли упомянуть, как обрабатывать выборки элементов. Этот вопрос более подробно рассматривается в разделе об обработке выборок в представлениях номенклатур.

3.3. КЛАСС МОДЕЛЕЙ

Перед изучением того, как обрабатываются выбранные элементы, может оказаться полезным изучить понятия, используемые в рамках модели/представления.

3.3.1. ОСНОВНОЕ ПОНЯТИЕ

В архитектуре модель / представление модель предоставляет стандартный интерфейс, который представления и делегаты используют для доступа к данным. В Qt стандартный интерфейс определяется классом QAbstractItemModel. Независимо от того, как элементы данных хранятся в любой базовой структуре данных, все подклассы QAbstractItemModel представляют данные в виде иерархической структуры, содержащей таблицы элементов. Представления используют это *Соглашение* для доступа к элементам данных в модели, но они не ограничены способом представления этой информации пользователю.

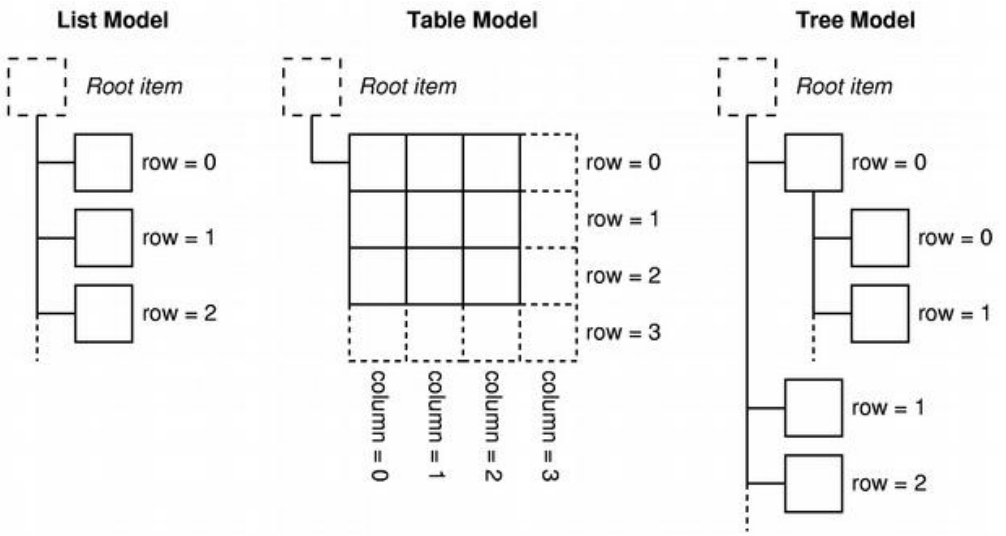

Рис. 16. Схема различных моделей

Модели также уведомляют любые вложенные представления об изменениях данных через механизм сигналов и слотов.

В этом разделе описываются некоторые основные понятия, которые являются центральными для способа доступа к элементам данных другими компонентами через класс модели. Более продвинутые концепции обсуждаются в последующих разделах.

3.3.2. СТРОКИ И СТОЛБЦЫ

В своей самой основной форме модель может быть доступна в виде простой таблицы, в которой элементы расположены по их номерам строк и столбцов. *Это не означает, что базовые части данных хранятся в структуре массива* ; использование номеров строк и столбцов-это только соглашение, позволяющее компонентам взаимодействовать друг с другом. Мы можем получить информацию о любом заданном элементе, указав номера его строк и столбцов в модели, и получим индекс, представляющий этот элемент:

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

Модели, которые предоставляют интерфейсы к простым, одноуровневым структурам данных, таким как списки и таблицы, не нуждаются в предоставлении какой-либо другой информации, но, как показывает приведенный выше код, нам нужно предоставить дополнительную информацию при получении индекса модели.

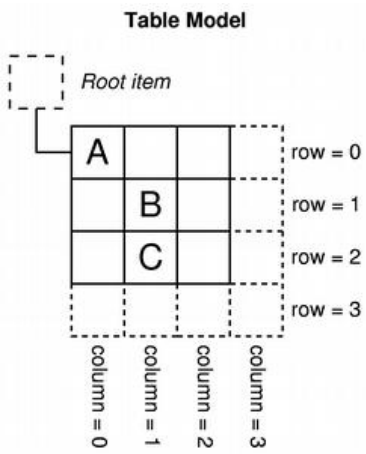

Рис. 17. Модель таблицы

На диаграмме показано представление базовой табличной модели, в которой каждый элемент расположен по паре номеров строк и столбцов. Мы получаем индекс модели, который ссылается на элемент данных, передавая соответствующие номера строк и столбцов в модель.

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

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

На элементы верхнего уровня в модели всегда ссылаются, указывая QModelIndex()в качестве их родительского элемента. Это обсуждается в следующем разделе.

3.3.3. РОДИТЕЛИ ПРЕДМЕТОВ

Табличный интерфейс к данным элементов, предоставляемый моделями, идеально подходит при использовании данных в табличном или списковом представлении; система номеров строк и столбцов точно соответствует способу отображения элементов в представлениях. Однако такие структуры, как древовидные представления, требуют от модели предоставления более гибкого интерфейса для входящих в нее элементов. В результате каждый элемент также может быть родительским для другой таблицы элементов, во многом так же, как элемент верхнего уровня в представлении дерева может содержать другой список элементов.

При запросе индекса для элемента модели мы должны предоставить некоторую информацию о родителе элемента. Вне модели, единственный способ ссылаться на элемент - это через индекс модели, поэтому родительский индекс модели также должен быть задан:

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

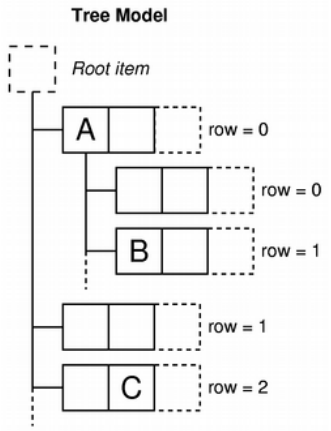

Рис. 18. Модель дерева

На диаграмме показано представление модели дерева, в которой на каждый элемент ссылается родитель, номер строки и номер столбца.

Элементы "A" и "c" представлены в модели как родные братья верхнего уровня:

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

ПУНКТ "A" имеет ряд детей. Модельный индекс для пункта "B" получается со следующим кодом:

```
ModelIndex indexB = model->index(1, 0, indexA);
```

3.3.4. КРАТКИЕ СВЕДЕНИЯ

Индексы моделей предоставляют представления и делегируют сведения о расположении элементов, предоставляемых моделями таким образом, что это не зависит от каких-либо базовых структур данных.

Элементы ссылаются на их номера строк и столбцов, а также на индекс модели их родительских элементов.

Индексы моделей создаются моделями по запросу других компонентов, таких как представления и делегаты.

Если при запросе индекса с помощью функции index () для родительского элемента указывается допустимый индекс модели , то возвращаемый индекс ссылается на элемент, расположенный под этим родительским элементом в модели. Полученный индекс относится к дочернему элементу этого элемента.

Если при запросе индекса с помощью функции index () для родительского элемента указывается недопустимый индекс модели , возвращаемый индекс ссылается на элемент верхнего уровня в модели.

Роль различает различные виды данных, связанных с элементом.

3.3.5. ИСПОЛЬЗОВАНИЕ ИНДЕКСОВ МОДЕЛЕЙ

Чтобы продемонстрировать, как данные могут быть получены из модели, используя индексы модели, мы устанавливаем QFileSystemModel без представления и отображаем имена файлов и каталогов в виджете. Хотя это не показывает обычный способ использования модели, он демонстрирует соглашения, используемые моделями при работе с модельными индексами.

Загрузка QFileSystemModel является асинхронной для минимизации использования системных ресурсов. Мы должны учитывать это при работе с этой моделью.

Мы строим модель файловой системы следующим образом:

```
QFileSystemModel *model = new QFileSystemModel;
   connect(model, &QFileSystemModel::directoryLoaded, [model](const QString &directory) {
      QModelIndex parentIndex = model->index(directory);
      int numRows = model->rowCount(parentIndex);
   });
   model->setRootPath(QDir::currentPath);
```

В этом случае мы начнем с настройки qfilesystemmodel по умолчанию. Мы подключаем его к лямбде, в которой мы получим Родительский индекс, ис-

пользуя конкретную реализацию index(), предоставленную этой моделью. В лямбде мы подсчитываем количество строк в модели, используя функцию rowCount(). Наконец, мы устанавливаем корневой путь QFileSystemModel, поэтому он начинает загрузку данных и запускает лямбду.

Для простоты нас интересуют только элементы в первой колонке модели. Мы изучаем каждую строку по очереди, получая индекс модели для первого элемента в каждой строке, и считываем данные, сохраненные для этого элемента в модели.

```
for (int row = 0; row < numRows; ++row) {
    QModelIndex index = model->index(row, 0, parentIndex);
```

Чтобы получить индекс модели, мы указываем номер строки, номер столбца (ноль для первого столбца) и соответствующий индекс модели для родительского элемента всех элементов, которые мы хотим. Текст, сохраненный в каждом элементе, извлекается с помощью функции Data() модели. Мы задаем индекс модели и роль DisplayRole для получения данных для элемента в виде строки.

```
QString text = model->data(index, Qt::DisplayRole).toString();
    // Display the text in a widget.

}
```

В приведенном выше примере демонстрируются основные принципы, используемые для извлечения данных из модели:

- Размеры модели можно найти с помощью rowCount() и columnCount(). Эти функции обычно требуют указания индекса родительской модели.
- Индексы модели используются для доступа к элементам модели. Для указания элемента необходимы строка, столбец и индекс родительской модели.
- Чтобы получить доступ к элементам верхнего уровня в модели, укажите нулевой индекс модели в качестве родительского индекса C.QModelIndex()
- Элементы содержат данные для различных ролей. Чтобы получить данные для конкретной роли, в модель необходимо ввести как индекс модели, так и роль.

3.4. КЛАСС ПРЕДСТАВЛЕНИЯ

3.4.1. КОНЦЕПЦИЯ

В архитектуре модель/представление представление получает элементы данных из модели и представляет их пользователю. Способ представления дан-

ных не обязательно должен напоминать представление данных, предоставляемое моделью, и может *полностью отличаться* от базовой структуры данных, используемой для хранения элементов данных.

Разделение содержания и представления достигается за счет использования интерфейса стандартной модели , предоставляемого QAbstractItemModel, интерфейса стандартного представления, предоставляемого QAbstractItemView, и использования индексов модели, которые представляют элементы данных в общем виде. Представления обычно управляют общим расположением данных, полученных из моделей. Они могут сами отображать отдельные элементы данных или использовать делегаты для обработки как объектов визуализации, так и объектов редактирования.

Помимо представления данных, представления управляют навигацией между элементами и некоторыми аспектами выбора элементов. В представлениях также реализованы основные функции пользовательского интерфейса, такие как контекстные меню и перетаскивание. Представление может предоставлять средства редактирования по умолчанию для элементов или работать с делегатом для предоставления пользовательского редактора.

Представление может быть построено без модели, но модель должна быть предоставлена, прежде чем она сможет отображать полезную информацию. Представления отслеживают элементы, выбранные пользователем с помощью выборок, которые могут быть сохранены отдельно для каждого представления или совместно использоваться несколькими представлениями.

В некоторых представлениях, таких как QTableView и QTreeView, отображаются заголовки, а также элементы. Они также реализуются классом представления QHeaderView. Заголовки обычно обращаются к той же модели, что и содержащее их представление. Они извлекают данные из модели с помощью функции QAbstractItemModel::headerData () и обычно отображают информацию заголовка в виде метки. Новые заголовки могут быть разделены на подклассы из класса QHeaderView, чтобы обеспечить более специализированные метки для представлений.

3.4.2. ИСПОЛЬЗОВАНИЕ СУЩЕСТВУЮЩЕГО ПРЕДСТАВЛЕНИЯ

Qt предоставляет три готовых к использованию класса представления, которые представляют данные из моделей способами, знакомыми большинству пользователей. QListView может отображать элементы из модели в виде простого списка или в виде классического представления значков. QTreeView отображает элементы из модели в виде иерархии списков, позволяя глубоко вложенные структуры, которые будут представлены в компактном виде. QTableView представляет элементы из модели в виде таблицы так же, как макет приложения электронной таблицы.

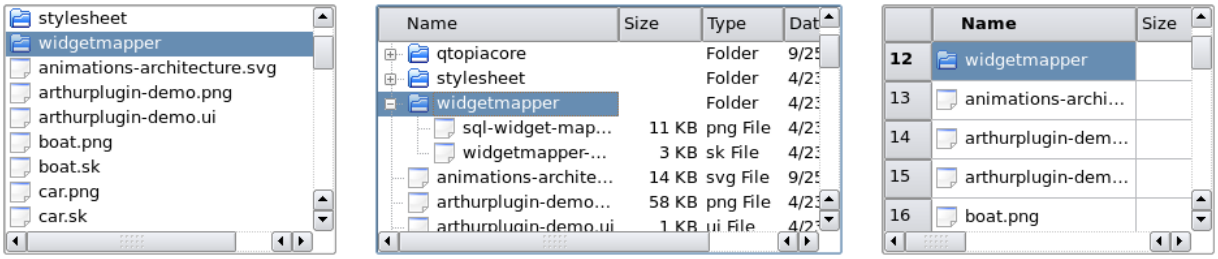

Рис. 19. Список модели

Поведение по умолчанию стандартных представлений, показанных выше, должно быть достаточным для большинства приложений. Они предоставляют основные средства редактирования и могут быть настроены в соответствии с потребностями более специализированных пользовательских интерфейсов.

3.4.3. ИСПОЛЬЗОВАНИЕ МОДЕЛИ

Мы берем модель строкового списка, которую мы создали в качестве примера модели, устанавливаем ее с некоторыми данными и строим представление для отображения содержимого модели. Все это может быть выполнено в рамках одной функции:

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

// Unindented for quoting purposes:
QStringList numbers;
numbers << "One" << "Two" << "Three" << "Four" << "Five";

QAbstractItemModel *model = new StringListModel(numbers);
```

Обратите внимание, что StringListModelon объявлен как QAbstractItemModel. Это позволяет нам использовать абстрактный интерфейс к модели и гарантирует, что код все еще работает, даже если мы заменим модель списка строк другой моделью.

Представление списка, предоставляемое QListView, достаточно для представления элементов в модели строкового списка. Мы построим представление и настроим модель, используя следующие строки кода:

```
QListView *view = new QListView;
view->setModel(model);
```

Вид отображается в обычном режиме:

```
 view->show();
```

```
    return app.exec();
}
}
```

Представление отображает содержимое модели, получая доступ к данным через интерфейс модели. Когда пользователь пытается изменить элемент, представление использует делегат по умолчанию для предоставления виджета редактора.

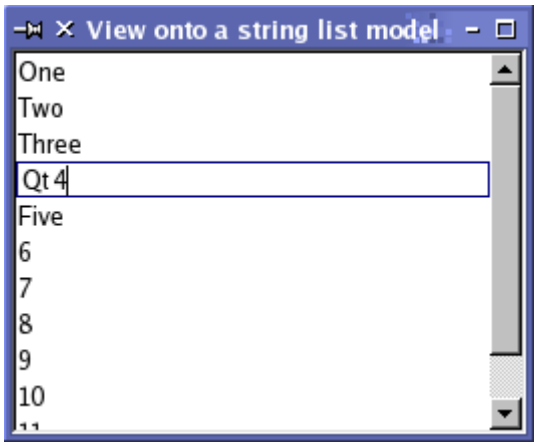

Рис. 20. Представление в виде таблицы

На приведенном выше рисунке показано, как QListView представляет данные в модели строкового списка. Поскольку модель является редактируемой, представление автоматически позволяет редактировать каждый элемент списка с помощью делегата по умолчанию.

3.5. ДЕЛЕГИРОВАНИЕ КЛАССОВ

3.5.1. КОНЦЕПЦИЯ

В отличие от шаблона Model-View-Controller, дизайн model/view не включает полностью отдельный компонент для управления взаимодействием с пользователем. Как правило, представление отвечает за представление данных модели пользователю, а также за обработку пользовательского ввода. Чтобы обеспечить некоторую гибкость в способах получения этих входных данных, взаимодействие осуществляется делегатами. Эти компоненты предоставляют возможности ввода и также отвечают за отображение отдельных элементов в некоторых представлениях. Стандартный интерфейс для управления делегатами определяется в классе QAbstractItemDelegate.

Ожидается, что делегаты смогут самостоятельно отображать свое содержимое, реализуя функции paint() и sizeHint (). Однако простые делегаты на основе виджетов могут создавать подклассы QStyledItemDelegate вместо QAbstractItemDelegate и использовать преимущества реализаций этих функций по умолчанию.

Редакторы для делегатов могут быть реализованы либо с помощью виджетов для управления процессом редактирования, либо путем непосредственной обработки событий. Первый подход рассматривается далее в этом разделе, и он также показан в Примере делегата Spin Box.

В Примере Pixelator показано, как создать пользовательский делегат, который выполняет специализированную визуализацию для представления таблицы.

3.5.2. ИСПОЛЬЗОВАНИЕ СУЩЕСТВУЮЩЕГО ДЕЛЕГАТА

Стандартные представления, предоставляемые вместе с Qt, используют экземпляры QStyledItemDelegate для предоставления средств редактирования. Эта стандартная реализация интерфейса делегата отображает элементы в обычном стиле для каждого из стандартных представлений: QListView, QTableView и QTreeView.

Все стандартные роли обрабатываются делегатом по умолчанию, используемым стандартными представлениями. Способ их интерпретации описан в документации QStyledItemDelegate.

Делегат, используемый представлением, возвращается функцией itemDelegate(). Функция setItemDelegate() позволяет установить пользовательский делегат для стандартного представления, и эту функцию необходимо использовать при настройке делегата для пользовательского представления.

3.5.3. ПРОСТОЙ ДЕЛЕГАТ

Делегат, реализованный здесь, использует QSpinBox для предоставления средств редактирования и в основном предназначен для использования с моделями, которые отображают целые числа. Хотя для этой цели мы создали пользовательскую целочисленную табличную модель, вместо нее можно было бы легко использовать QStandardItemModel, поскольку пользовательский делегат управляет вводом данных. Мы создаем табличное представление для отображения содержимого модели, и это будет использовать пользовательский делегат для редактирования.

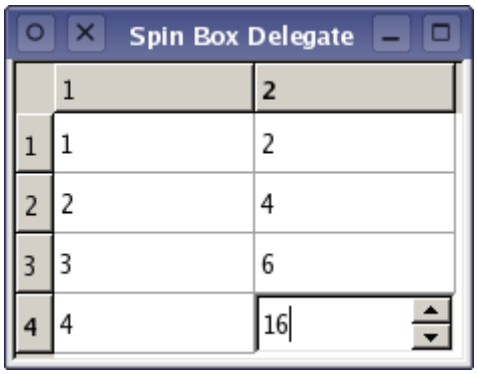

Рис. 21. Представление в виде таблице

Мы подкласс делегата от QStyledItemDelegate, потому что мы не хотим писать пользовательские функции отображения. Однако мы все равно должны предоставить функции для управления виджетом редактора:

```
class SpinBoxDelegate : public QStyledItemDelegate
{
   Q_OBJECT

public:
   SpinBoxDelegate(QObject *parent = nullptr);

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

   void setEditorData(QWidget *editor, const QModelIndex &index) const override;
   void setModelData(QWidget *editor, QAbstractItemModel *model,
                     const QModelIndex &index) const override;

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

Обратите внимание, что никакие виджеты редактора не настраиваются при создании делегата. Мы создаем виджет редактора только тогда, когда это необходимо.

3.5.4. ОТПРАВКА ДАННЫХ В МОДЕЛЬ

Когда пользователь закончит редактирование значения в поле вращений, представление попросит делегата сохранить измененное 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);
}
```

Поскольку представление управляет виджетами редактора для делегата, нам нужно только обновить модель с содержимым предоставленного редактора. В этом случае мы гарантируем, что спин-бокс обновлен, и обновляем модель со значением, которое она содержит, используя указанный индекс.

Стандартный класс QStyledItemDelegate информирует представление о завершении редактирования, посылая сигнал closeEditor(). Представление гарантирует, что виджет редактор будет закрыт и уничтожен. В этом примере мы предоставляем только простые средства редактирования, поэтому нам никогда не нужно излучать этот сигнал.

Все операции с данными выполняются через интерфейс, предоставляемый QAbstractItemMode. Это делает делегат в основном независимым от типа данных, которыми он управляет, но некоторые предположения должны быть сделаны для использования определенных типов виджетов редактора. В этом примере мы предположили, что модель всегда содержит целочисленные значения, но мы все еще можем использовать этот делегат с различными типами моделей, потому что QVariant предоставляет разумные значения по умолчанию для неожиданных данных.

3.5.5. СОЗДАНИЕ НОВЫХ МОДЕЛЕЙ

Разделение функциональных возможностей между компонентами модели / представления позволяет создавать модели, которые могут использовать преимущества существующих представлений. Этот подход позволяет нам представлять данные из различных источников с использованием стандартных компонентов графического пользовательского интерфейса, таких как QListView, QTableView и QTreeView .

Класс QAbstractItemModel предоставляет достаточно гибкий интерфейс для поддержки источников данных, упорядочивающих информацию в иерархических структурах, что позволяет добавлять, удалять, изменять или сортировать данные тем или иным способом. Он также обеспечивает поддержку операций перетаскивания.

Классы QAbstractListModel и QAbstractTableModel обеспечивают поддержку интерфейсов для более простых неиерархических структур данных и легче используются в качестве отправной точки для простых моделей списков и таблиц.

В этом разделе мы создадим простую модель только для чтения, чтобы изучить основные принципы архитектуры model/view. Далее в этом разделе мы адаптируем эту простую модель, чтобы элементы могли быть изменены пользователем.

Пример более сложной модели см. В разделе пример простой модели дерева.

Требования подклассов QAbstractItemModel более подробно описаны в справочном документе модель подклассов.

3.5.6. ПРОЕКТИРОВАНИЕ МОДЕЛИ

При создании новой модели для существующей структуры данных важно рассмотреть, какой тип модели должен использоваться для обеспечения интерфейса с данными. Если структура данных может быть представлена в виде списка или таблицы элементов, можно создать подкласс QAbstractListModel или QAbstractTableModel, поскольку эти классы предоставляют подходящие реализации по умолчанию для многих функций.

Однако, если базовая структура данных может быть представлена только иерархической древовидной структурой, необходимо создать подкласс QAbstractItemModel . Этот подход взят в простом примере модели дерева.

В этом разделе мы реализуем простую модель, основанную на списке строк, поэтому QAbstractListModel предоставляет идеальный базовый класс для построения.

Независимо от того, какую форму принимает базовая структура данных, обычно рекомендуется дополнить стандартный API QAbstractItemModel в специализированных моделях тем, который обеспечивает более естественный доступ к базовой структуре данных. Это упрощает заполнение модели данными, но все же позволяет другим общим компонентам модели/представления взаимодействовать с ней с помощью стандартного API. Модель, описанная ниже, предоставляет пользовательский конструктор именно для этой цели.

3.5.7. ЗАГОЛОВКИ МОДЕЛЕЙ И ДАННЫЕ

Для элементов в представлении мы хотим вернуть строки в списке строк. Функция data() отвечает за возврат элемента данных, соответствующего аргументу index:

```
QVariant StringListModelStringListModel::data((const QModelIndex &index,
int role) ) const
{
   {
   if ((!index.isValid())
      ())
      return QVariant();



   if (index(index.row() () >= stringList.size())
      ())
      return QVariant();



   if (role (role == Qt::DisplayRole)
```

```
        )
        return stringList.at(index(index.row());
    ());
    else
        return QVariant();
}
}
```

Мы возвращаем допустимый QVariant только в том случае, если указанный индекс модели является допустимым, номер строки находится в диапазоне элементов в списке строк, а запрошенная роль является той, которую мы поддерживаем.

Некоторые представления, такие как QTreeView и QTableView, могут отображать заголовки вместе с данными элемента. Если наша модель отображается в виде с заголовками, Мы хотим, чтобы заголовки отображали номера строк и столбцов. Мы можем предоставить информацию о заголовках, создав подкласс функции headerData():

```
QVariant StringListModelStringListModel::headerData((int section,
Qt::Orientation orientation orientation,
                                int role) ) const
{
    {
    if (role (role != Qt::DisplayRole)
        )
        return QVariant();



    if (orientation (orientation == Qt::Horizontal)
        )
        return QStringLiteral("Column %1").arg(section);
    (section);
    else
        return QStringLiteral("Row %1").arg(section);
}(section);
}
```

Мы возвращаем допустимый QVariant только в том случае, если роль является той, которую мы поддерживаем. Ориентация заголовка также учитывается при принятии решения о точном возврате данных.

Не все представления отображают заголовки с данными элемента, а те, которые это делают, могут быть настроены для их скрытия. Тем не менее, ре-

комендуется реализовать функцию headerData() для предоставления соответствующей информации о данных, предоставляемых моделью.

Элемент может иметь несколько ролей, выдавая различные данные в зависимости от указанной роли. Элементы в нашей модели имеют только одну роль DisplayRole, поэтому мы возвращаем данные для элементов независимо от указанной роли. Однако мы можем повторно использовать данные, предоставляемые для роли DisplayRole, в других ролях, таких как ToolTipRole, который представления могут использовать для отображения информации об элементах во всплывающей подсказке.

3.5.8. ВСТАВКА И УДАЛЕНИЕ СТРОК

Можно изменить количество строк и столбцов в модели. В модели списка строк имеет смысл только изменить количество строк, поэтому мы только переопределяем функции для вставки и удаления строк. Они объявлены в определении класса:

```
bool insertRows(bool insertRows(int position, int rows, const QModelIndex &index = QModelIndex()) override;
    bool removeRows( override;
    bool removeRows(int position, int rows, const QModelIndex &index = QModelIndex()) override; override;
```

Поскольку строки в этой модели соответствуют строкам в списке, insertRows()функция вставляет ряд пустых строк в список строк перед указанной позицией. Количество вставленных строк эквивалентно количеству указанных строк.

Родительский индекс обычно используется для определения того, где в модели должны быть добавлены строки. В этом случае у нас есть только один список строк верхнего уровня, поэтому мы просто вставляем пустые строки в этот список.

```
bool StringListModel StringListModel::insertRows((int position, int rows, const QModelIndex &parent)
{
   beginInsertRows()
{
   beginInsertRows(QModelIndex(), position, position+rows-1);



   for ((int row = 0; row  row < rows; ; ++row) {
      stringList) {
      stringList.insert(position(position, "");
```

```
    }

    endInsertRows();

    }

    endInsertRows();
    return true;
}
}
```

Модель сначала вызывает функцию beginInsertRows(), чтобы сообщить другим компонентам, что число строк скоро изменится. Функция задает номера строк первой и последней новых вставляемых строк, а также индекс модели для их родительского элемента. После изменения списка строк он вызывает endInsertRows(), чтобы завершить операцию и сообщить другим компонентам, что размеры модели изменились, возвращая true для указания успеха.

Функция удаления строк из модели также проста в написании. Строки, которые будут удалены из модели, определяются положением и количеством заданных строк. Мы игнорируем Родительский индекс, чтобы упростить нашу реализацию, и просто удаляем соответствующие элементы из списка строк.

```
bool StringListModel StringListModel::removeRows((int position, int rows,
const QModelIndex &parent)
{
    beginRemoveRows()
{
    beginRemoveRows(QModelIndex(), position, position+rows-1);



    for ((int row = 0; row  row < rows; ; ++row) {
        stringList) {
        stringList.removeAt(position);
    }

    endRemoveRows();
    (position);
    }

    endRemoveRows();
    return true;
}
```

}

Функция beginRemoveRows() всегда вызывается до удаления каких-либо базовых данных и указывает первую и последнюю строки, подлежащие удалению. Это позволяет другим компонентам получить доступ к данным до того, как они станут недоступными. После удаления строк модель выводит endRemoveRows(), чтобы завершить операцию и сообщить другим компонентам, что размеры модели изменились.

4. ЛАБОРАТОРНЫЕ ЗАДАНИЯ И МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ИХ ВЫПОЛНЕНИЮ

1. В программе untitled1 добавили новый класс названый newmodel.

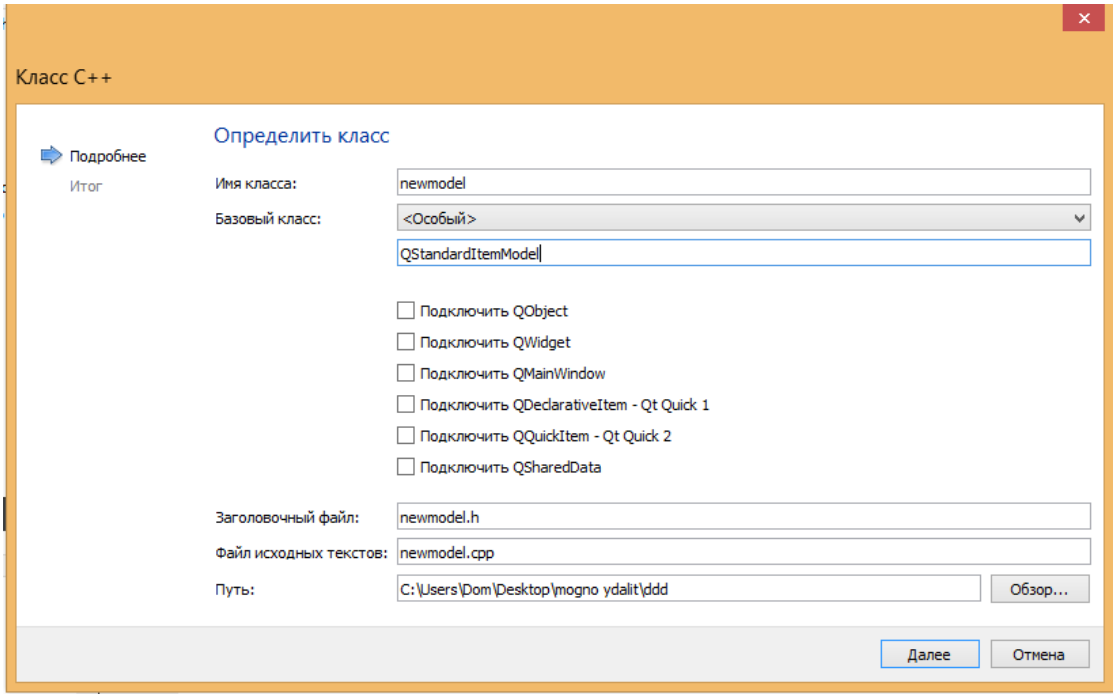

Рис. 22. Добавление нового класса

2. В проекте untitled1 в заголовочном файле newmodel.h через указатель объявили закрытый метод void refresh(); и публичный void update();

```
#ifndef NEWMODEL_H
#define NEWMODEL_H

#include <QStandardItemModel>
#include <QtSql>

class NewModel : public QStandardItemModel
{
    Q_OBJECT

private:
    void refresh();
    //QMultiMap<QString,QString> map;

public:
    explicit NewModel(QStandardItemModel *parent = nullptr);
    void update();

signals:

public slots:
};

#endif // NEWMODEL_H

```

Рис. 23. Код нового класса

3. В проекте untitled1 в исходном файл newmodel.cpp реализовали модель присвоения ID.

```
#include "newmodel.h"
#include <QtSql>

NewModel::NewModel(QStandardItemModel *parent) : QStandardItemModel(parent)
{
    refresh();
}

void NewModel::refresh()
{
    clear();
    insertColumns(0,3);
    QSqlQuery query;
    query.exec("select* from testdb");
    qDebug()<<query.lastError();
    int id;
    QString code;
    QString university;
    QSqlRecord rec = query.record();
    while (query.next())
    {
        id = query.value(rec.indexOf("id")).toInt();
        code = query.value(rec.indexOf("code")).toString();
        university = query.value(rec.indexOf("university")).toString();
        insertRow(rowCount());
        setData(index(rowCount()-1,0),id);
        setData(index(rowCount()-1,1),code);
        setData(index(rowCount()-1,2),university);
    }
}

void NewModel::update()
{

}

```

Рис. 24. Код модели присвоения

4.1. УКАЗАНИЯ ПО ОФОРМЛЕНИЮ ОТЧЕТА

Отчет должен содержать название и цель работы, краткие теоретические сведения, а также код с комментариями.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Саммерфилд М. Qt. Профессиональное программирование. Разработка кроссплатформенных приложений на C++ / М. Саммерфилд – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 560 с.
2. Шлее М. Qt 5.10. Профессиональное программирование на C++ / М. Шлее – СПб: БХВ-Петербург, 2018. – 1072 с.
3. Бланшет Ж. QT 4: программирование GUI на C++ / Ж. Бланшет – М.: КУДИЦ-ПРЕСС, 2007. – 546 с.

ОГЛАВЛЕНИЕ

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторных работ №4-5
для студентов специальности 11.05.01
«Радиоэлектронные системы и комплексы»
очной формы обучения

Составитель
Сукачев Александр Игоревич

Издается в авторской редакции

Подписано к изданию 18.03.2024.
Уч.-изд. л. 2,4.

ФГБОУ ВО «Воронежский государственный технический университет»
394006 Воронеж, ул. 20-летия Октября, 84