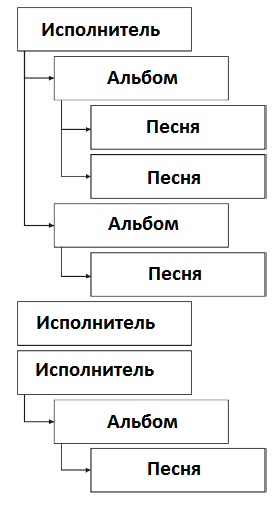
**Лабораторная работа №3. (лекции 7-8)**

Цель: научиться создавать приложение основанное на архитектуре модель-представление Qt. Понять как модель взаимодействует с остальным приложением.

Qt Meta-object system, Qt resource system, Qt Signal-Slot mechanism, Qt model/view architecture, Qt Creator.   
QApplication, QDataStream, QAbstractItemModel, QTreeView, QModelIndex, QHeaderView, QItemDelegate

Продолжительность: 4 часа



Задача:

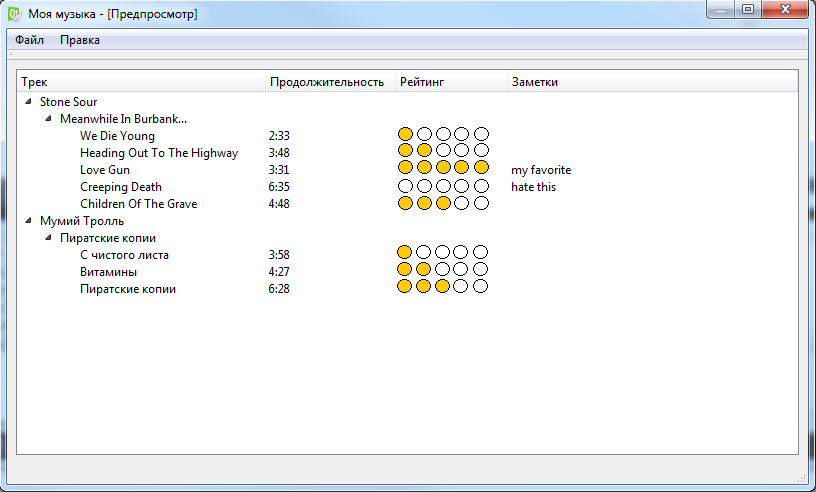
Разработать приложение для хранения информации о музыкальной коллекции. Приложение должно использовать деревянную модель для хранения треков, сгруппированный по альбому и исполнителю. Данная программа сложнее, чем примеры на лекциях, поэтому лучше потратить время на изучения каждого этапа создания приложения и прочитать документацию, чтобы понимать что делается и почему.

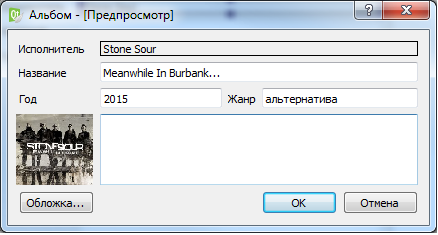
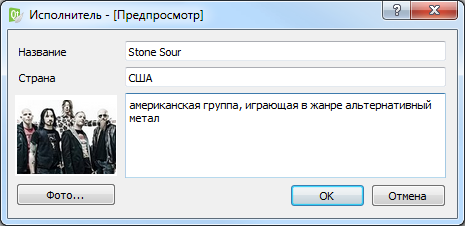
При выполнении ЛР рекомендуется использовать т.н. подход *от частного к общему (снизу - вверх)*. Т.е. сначала создаются небольшие модули из которых потом собирается все приложение. Недостаток этого подхода в том, что нет общей картины при разработке и сложно отследить насколько работа закончена. Это компенсируется возможностью легко протестировать каждый компонент отдельно без взаимодействия с другими частями приложения. Тестирование очень важная часть разработки, когда речь идет об архитектуре модель-представление и тем более о пользовательских моделях, где очень легко сделать ошибку.

Выполнение этой работы подразумевает, что вы уже свободно владеете основами создания Qt приложений. Инструкция больше уделяет внимания задаче, которую нужно сделать, а не тому как ее выполнить.

При использовании подхода *от частного к общему* вам нужно разработать общий план всего проекта. Для выполнения этой ЛР план может быть таким:

1. Разработать структуры для хранения информации о песнях, альбомах и исполнителях с методами записи и чтения их из постоянного хранилища (например жесткий диск)
   1. классы Artist, Album и Song для хранения данных
   2. сохранение и чтение всех данных в файл через QDataStream
2. Реализовать модель данных , которая “оборачивает” структуры в интерфейсы архитектуры модель-представление Qt
   1. класс MusicModel реализующий интерфейс QAbstractItemModel для иерархической модели
   2. модель поддерживает редактирование данных, вставку и удаление строк (исполнителей, альбомов, песен)
3. Реализовать делегаты для отрисовки и редактирования данных
   1. рейтинг песни отображается и редактируется мышью с помощью пользовательского делегата.
   2. Внешний вид рейтинга должен быть таким 
4. Реализовать диалог для удобного управления данными в модели
   1. диалог свойств(характеристик) исполнителя и альбома
5. Реализовать главное окно приложения с меню, панелью инструментов и представлением, которое объединяет все части вместе
   1. представление QTreeView для отображения данных модели
   2. действие для загрузки и сохранения данных в файл(меню, панель инструментов)
   3. действие для добавления/удаления элементов коллекции
   4. пункты меню и кнопки панели инструментов активны когда нужно (например нельзя добавить трек к исполнителю).





**Структура данных для музыкальной коллекции**

Исполнитель описывается следующими свойствами (характеристиками, атрибутами)

|  |  |
| --- | --- |
| **Атрибут (переменная)** | **Тип данных** |
| имя | строка |
| фото | изображение |
| страна | строка |
| комментарий | строка |
| альбомы | список элементов типа “Альбом” |

Альбом имеет характеристики:

|  |  |
| --- | --- |
| **Атрибут (переменная)** | **Тип данных** |
| Имя | строка |
| год выхода | целое |
| обложка | изображение |
| жанр | строка |
| комментарий | строка |
| Песни | список элементов типа “Трек” |
| Исполнитель | Исполнитель |

Трек имеет характеристики

|  |  |
| --- | --- |
| **Атрибут (переменная)** | **Тип данных** |
| название | строка |
| продолжительность | время |
| рейтинг | целое |
| комментарий | строка |
| из альбома | Альбом |

Требования к приложению.

1. Для хранения данных о музыкальной коллекции реализованы классы Artist, Album и Song, являющиеся частью иерархической структуры согласно заданию.
2. Музыкальная коллекция сохраняется и загружается из файла с помощью QDataStream.
3. Реализована иерархическая модель (класс MusicModel) на основе QAbstractItemModel, обеспечивающая доступ к данным в объектах классов Artist, Album и Song.
4. Модель поддерживает редактирование данных о треке (название, длительность, комментарий) с помощью стандартных делегатов
5. Для отображения и редактирования рейтинга трека используется собственный делегат (класс RatingDelegate) на основе класса QStyledItemDelegate.
6. Главное окно приложения содержит представление QTreeView. Отображение модели MusicModel в представлении соответствует структуре коллекции.
7. Реализованы диалоги для отображения и редактирования всех свойств элементов коллекции: исполнитель и альбом. Диалог открывается по контекстному меню “Свойства” у элемента дерева.
8. При закрытии диалога свойств элемента коллекции по кнопке “ОК” измененные данные сохраняются в модель и отображаются в представлении.
9. Приложение позволяет создавать, удалять и редактировать элементы коллекции через меню и кнопки панели инструментов. Пункты меню приложения и кнопки панели управления активны когда ожидается.
10. При добавлении нового элемента коллекции открывается соответствующий диалог свойств элемента для ввода данных об элементе
11. Модель корректно отрабатывает операцию добавления нового элемента в коллекцию и новый элемент появляется в представлении
12. Модель корректно отрабатывает операцию удаления элемента из коллекции и новый элемент исчезает из представления
13. Заголовок главного окна приложение и признак изменения данных коллекции работают как ожидаются.

Оценка

Максимальная оценка за выполнение работы - 25 баллов. Оценка выставляется по результатам проверки работоспособности программы по пунктам требований по правилу:

* 1 балл за пункты 2,4,9,13
* 2 балла за каждый из пунктов 6,7,8,10,11,12
* 3 балла за каждый из пунктов 1,3,5

При выполнении работы на 25 баллов самостоятельно, без использования методического материала, оценка увеличивается на **20%** (т.е. можно получить 30 баллов). Если выполнение работы с помощью методического материала займет более чем неделю, оценка уменьшается на **20%** (т.е. можно получить не более 20 баллов).

**Методические указания по выполнению лабораторной работы №3.**

***1.Структуры данных***

Приложение основано на иерархической(деревянной) структуре представляющей три типа информации: треки, альбомы и исполнителей. Каждый тип информации будет представлен отдельным классом со своими атрибутами и методами.

Исполнитель описывается следующими характеристиками(атрибутами)

|  |  |
| --- | --- |
| **Атрибут (переменная)** | **Тип данных** |
| имя | строка |
| фото | изображение |
| страна | строка |
| комментарий | строка |
| альбомы | список элементов типа “Альбом” |

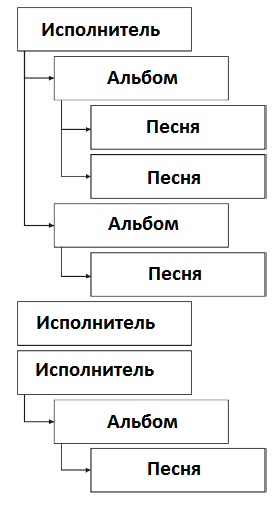
Альбом имеет характеристики:

|  |  |
| --- | --- |
| **Атрибут (переменная)** | **Тип данных** |
| Имя | строка |
| год выхода | целое |
| обложка | изображение |
| жанр | строка |
| комментарий | строка |
| Песни | список элементов типа “Трек” |
| Исполнитель | Исполнитель |

Трек имеет характеристики

|  |  |
| --- | --- |
| **Атрибут (переменная)** | **Тип данных** |
| название | строка |
| продолжительность | время |
| рейтинг | целое |
| комментарий | строка |
| из альбома | Альбом |

Все эти данные собираются вместе в иерархическую структуру, как показано на рисунке:



Все эти элементы являются частью иерархической структуры, и все они основаны на общем классе Item. Общий базовый класс нужен, чтобы операции, свойственные всем трем типам данным реализовать один раз, в частности методы для управления иерархической структурой. Создайте класс Item:

class Item {

public:

Item();

~Item();

Item \*parent() const;

void setParent(Item \*);

void insertChild(Item \*, int position = -1);

Item \*takeChild(int);

Item \*childAt(int) const;

int indexOf(Item\*) const;

int childCount() const;

private:

Item \*m\_parent;

QList<Item\*> m\_children;

QString m\_name;

QString m\_comment;

};

Вот некоторые советы по реализации методов класса Item:

1. конструктор должен создавать пустой объект без родителя
2. в деструкторе объект должен отсоединяться от родителя ( setParent(0)) и удалять все свои дочерние объекты - при этом вызовутся их деструкторы, в которых будут удалены их дети и т.д. Это гарантирует, что все элементы дерева будут последовательно удалены не нарушая его целостности и не будет утечек памяти.
3. метод insertChild() вставляет заданный элемент в указанную позицию в списке детей и так же устанавливает текущий элемент как родителя.
4. метод takeChild() делает обратное, удаляет элемент из заданной позиции из списка детей, отсоединяет его от родителя(от себя) и возвращает указатель на этот “изъятый” элемент.
5. метод setParent() устанавливает родителя для текущего элемента, и также выполняет функции двух предыдущих методов. Он должен присоединять(отсоединять) себя в родительский список детей. Будьте осторожны, чтобы не получился бесконечный цикл ( setParent вызывает insertChild, который вызывает setParent и т.д.)
6. методы childAt() и indexOf() - пара методов для получения элемента по заданной позиции или позиции заданного элемента. Если элемент не находится в списке детей или указанная позиция находится вне границ списка возвращается null или -1 соответственно
7. Для реализации остальных методов особые инструкции не нужны

После реализации класса Item нужно проверить его работу. Для этого напишите простую функцию main() в которой создайте иерархию объектов Item . Попробуйте добавлять элементы в различном порядке, удалять их, добавить один элемент к нескольким родителям и т.д.

Другими словами, убедитесь что все работает до того как продолжать дальше. Если вы сейчас сделаете ошибку, будет сложно отловить ее потом.

Когда ваш класс Item будет работать без ошибок, можно приступать к реализации классов Artist, Album и Song - подклассов Item, в которых будет храниться информация специфичная для каждого типа данных. Когда закончите, тоже протестируйте их подобным образом.

Позже вам понадобится приводить объекты типа Item к типам его подклассов. Можно использовать dynamic\_cast для этого или добавить три виртуальных метода в класс Item

virtual Artist\* toArtist() const;

virtual Album\* toAlbum() const;

virtual Song\* toSong() const;

Убедитесь, что они возвращают null в базовом классе Item и переопределите их в его подклассах, так чтобы они возвращали действительные указатели, например так:

Song\* Song::toSong() const { return this; }

Можно использовать эти методы в реализации Item::setParent() и Item::insertChild() чтобы гарантировать, что только Artist может быть родителем Album, Song детьми Album и т.д. Это будет работать еще лучше если вы эти два метода тоже сделаете виртуальными и переопределите в подклассах.

На данном этапе осталось сделать только методы сериализации. Используете для этого потоковые операторы QDataStream. Определите операторы чтения и записи в поток (<<,>>). Не забудьте про сериализацию дочерних элементов.

QDataStream& operator<<(QDataStream &stream, const Artist &artist) {

stream << “ARTIST”;

stream << artist.name() << artist.photo() << ... ;

// serialize children

int cnt = artist.childCount();

stream << cnt;

for(int i=0; i<cnt; ++i){

Album \*album = artist.childAt(i)->toAlbum();

if(album){

stream << \*album;

}

}

return stream;

}

И еще раз протестируйте код перед тем как продолжать, чтобы убедиться, что сериализация работает правильно - данные сохраняются в файл и правильно читаются из файла, т.е. правильно восстанавливается структура объектов.

***2.Структуры данных***

Теперь у вас есть структуры для хранения данных приложения и нужно предоставить к ним доступ через интерфейс QAbstractItemModel, чтобы отобразить их в представлении. Это значит, что нужен подкласс класса QAbstractItemModel с реализацией всех чисто виртуальных методов. Чтобы сделать нормальную модель нужно переопределить дополнительно еще пару виртуальных методов.

Это общий подход при использовании механизма модель-представление. У вашего приложения есть данные в специальных структурах, а вам нужно представить их как модель, чтобы данные могли быть использованы(отображены) в представлениях Qt.

Первый шаг при создании модели - получить ссылку на ваши структуры данных в модели. В случае с деревьями лучше всего добавить дополнительный “невидимый” элемент, который будет выступать в роль корневого элемента дерева. Он не должен быть доступен снаружи модели (и следовательно не будет отображаться в представлении) и не должен содержать никаких данных. Его единственная цель - обеспечить целостный способ хранения всех элементов в модели (чтобы каждый действительный узел дерева имел действительного родителя- в нашем случае чтобы у всех исполнителей был один общий родитель). Т.е. он будет родителем всех объектов Artist:

class MusicModel : public QAbstractItemModel {

private:

Item \*m\_root;

};

**Построение дерева.**

Первый метод, который нужно переопределить - QAbstractItemModel::index(). На основании номера строки, номера колонки и индекса родителя он должен создать и вернуть индекс элемента модели, т.е. вызвать QAbstractItemModel::createIndex() чтобы создать индекс (экземпляр QModelIndex) для запрашиваемого элемента. Метод createIndex() имеет последним параметром необязательный указатель. Используете его , чтобы хранить указатель на экземпляр Item (т.е. на исполнителя, альбом или песню в структуре данных), соответствующий индексу модели, чтобы можно было получить его позже. Сделайте это используя метод QModelIndex::internalPointer().

Номера строки и столбца недостаточно, чтобы представление могло правильно отобразить экземпляр Item. Ему еще нужно знать кто родитель, т.е. получить индекс родителя. Если вы используете внутренний указатель родителя, то получите указатель на него самого (экземпляр Item родителя). Затем вы можете запросить нужного ребенка на основе номера строки в вызове index().

Если метод вызывается для корня верхнего уровня, индекс родителя будет недействительным и вы не сможете получить внутренний указатель. Вот для чего нужен “невидимый” дополнительный корень. Если вы получите недействительный индекс родителя, используйте “невидимый” элемент для указателя на родительский элемент (Item ).

QModelIndex MusicModel::index(int row,int column,const QModelIndex &parent) const

{

Item \*parentItem = m\_root;

if(parent.isValid())

parentItem = static\_cast<Item\*>(parent.internalPointer());

// ... проверьте, что номер строки и столбца правильные и т.п. ...

return createIndex(row, column, parentItem->childAt(row));

}

Следующим шагом нужно реализовать метод parent(). В качестве шаблона можно использовать index(), но на этот раз трудность скорее не в получении указателя на Item, а в определении номера строки родителя у его родителя. Используйте Item::indexOf() для запроса у родителя родителя(у дедушки) позиции родителя в списке его детей.

Последние два чисто виртуальных метода которые осталось реализовать - rowCount() и columnCount(). Они должны вернуть количество колонок и дочерних строк (т.е. количество дочерних элементов). Используйте ту же информацию, которую вы применяли в реализации методов parent() и index().

Помните, что каждая колонка представляет разную характеристику (кроме первой, с деревом). Глядя на структуру данных вы можете сказать сколько колонок нужно для каждого типа информации (исполнитель, альбом или трек). *При реализации изложенной выше, вы можете определить тип по внутреннему указателю индекса.*

**Получение данных из модели**

Последний чисто виртуальный метод, т.е. который нужно обязательно переопределить - data(). Именно через него модель передает полезную информацию в представление.

Реализация достаточно проста - модель получает индекс (и следовательно его Item, через внутренний указатель) и роль данных. Нужно вернуть осмысленное значение по крайней мере для роли Qt::DisplayRole. Для тех ролей которые не поддерживаются верните недействительный QVariant. *Лучший способ реализации data() использовать switch оператор для выбора колонки и роли.*

Теперь можно протестировать модель. Создайте небольшое приложение с представлением QTreeView и свяжите его с моделью. Добавьте в модель данные. После запуска вы должны увидеть в представлении правильно построенное дерево. Если приложение “падает”, имеет пропущенные или несуществующие узлы, значит вы сделали ошибку при реализации модели. Добейтесь правильного отображения модели перед тем как продолжать.

**Делаем модель редактируемой**

В настоящий момент данные в модели нельзя редактировать. Вы знаете сколько элементов хранится в модели и можете их получить. Но у нас нет возможности ни изменять существующие данные, ни добавлять новые, ни удалять их.

Чтобы добавить возможность редактирования существующих данных нужно решить две задачи. Первая - переопределить метод flags() чтобы сообщить представлению, какие элементы модели можно редактировать. Просто верните флаг Qt::ItemIsEditable дополнительно к флагам по умолчанию для всех индексов, которым нужно добавить возможность редактирования. Чтобы не перечислять все флаги, можно использовать оператор логического ИЛИ для объединения флагов:

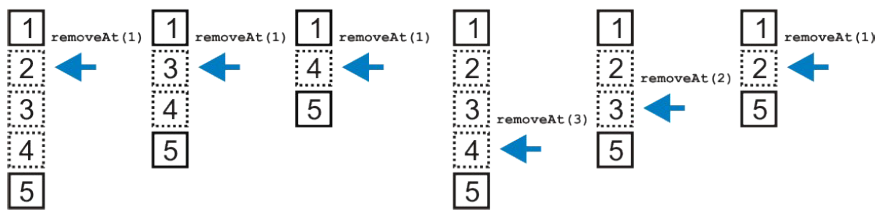
return QAbstractItemModel::flags(index) | Qt::ItemIsEditable;

Вторая задача - переопределить метод setData(). Этот метод обновляет данные в модели после окончания редактирования. Реализуйте его так, чтобы изменял данные модели для ролей Qt::EditRole или Qt::DisplayRole. Не забудьте послать сигнал dataChanged() с индексами измененных элементов перед тем как вернуть true.

Теперь можно запустить и проверить редактирование тестовых данных по двойному клику мыши. Если добавить вывод отладочных сообщений, то можно убедиться, что вызываются правильные методы при вводе новых значений.

Модель почти готова. Осталось только переопределить методы insertRows() и removeRows(). Добавление или удаление строк из модели сделать просто, если для их хранения используется простой список. В этом случае все сводится к вызову методов QList::insert() и QList::removeAt().

Когда вы удаляете строку, то все последующие строки сдвигаются на одну позицию. Посмотрите на рисунок, где показано что происходит. Вам следует реализовать этот алгоритм в коде или удалением и вставкой всех элементов начиная с последнего или пересчетом всех индексов.



При реализации этих двух алгоритмов следует помнить одну деталь. В документации Qt сказано, что при добавлении строк, перед тем как вы физически измените структуру данных, следует вызвать beginInsertRows(). А после того как вы закончите изменение , надо вызвать endInsertRows(). Тоже самое нужно сделать при удалении строк. Сначала вызвать beginRemoveRows(), а в конце endRemoveRows(). Это все необходимо для внутренней работы библиотеки и нужно не забывать их вызывать каждый раз при добавлении или удалении строк из модели. В противном случае представление может рассинхронизироваться с моделью.

**Проверка модели.**

Финальный шаг реализации модели - тестирование, чтобы убедиться, что все работает. Попробуйте использовать специальный класс ModelTest доступный по адресу <http://code.qt.io/cgit/qt/qt.git/tree/tests/auto/modeltest> . При тестировании проверьте все возможные варианты использования модели.

**Пользовательская отрисовка и редактирование элемента**

В классе Song есть переменная Rating. Она используется для хранения рейтинга трека - оценки понравившихся песен. Отображение рейтинга в виде числа возможно, но это не интересно. Лучше отображать рейтинг в виде “звезд”. Мы для простоты будем использовать пять кругов как показано на рисунке



Чтобы это сделать нужно реализовать пользовательский делегат.

Создайте класс делегата RatingDelegate как подкласс QStyledItemDelegate. Сначала рассчитайте размер, который будут занимать пять окружностей при отображении. Предположим один круг имеет диаметр 16 пикселей, расстояние меду соседними кругами 4 пикселя, а расстояние от кругов до любой границы ячейки 2 пикселя. Реализуйте метод sizeHint() и верните вычисленный размер ячейки для отображения рейтинга.

Вторая половина реализации делегата - рисование. Чтобы правильно реализовать метод paint() нужно знать две вещи: что рисовать и где рисовать.

Получить значение рейтинга из модели можно через аргумент index. Используйте QModelIndex::data() чтобы запросить у модели значение рейтинга и преобразуйте его в целое используя QVariant::toInt().

Чтобы определить геометрию ячейки запросите у аргумента option его свойство rect. Методы QRect::left() и QRect::top() помогут вам вычислить положение каждой окружности.

Осталось только в цикле вызвать метод QPainter::drawEllipse() с правильными координатами. Всегда рисуйте пять окружностей, а закрашивайте желтой кистью (Qt::yellow) только те, что отображают значение рейтинга, остальные - прозрачной кистью (Qt::NoBrush). Для рисования самих окружностей используйте черный карандаш.

Теперь используйте QAbstractItemView::setItemDelegateForColumn() чтобы установить делегат только для той колонки в которой должен отображаться рейтинг и проверьте его работу.

const int RatingColumn = 2; // возможно в вашей модели это будет  
 // другой номер колонки

RatingDelegate \*delegate = new RatingDelegate(view);

view->setItemDelegateForColumn(RatingColumn, delegate);

Когда вы запустите приложение, то заметите, что некоторые ячейки содержат пустые круги вместо текста. Это потому, что вы установили делегат для всех ячеек в колонке. Но делегат рейтинга нужно применять только к трекам.

Исправьте класс делегата, чтобы он вызывал реализацию базового класса для всех строк модели кроме строк трека и всех колонок, кроме колонки с рейтингом. Как и раньше, нужно использовать параметр index чтобы определить какие данные представляет элемент.

Допишите класс делетага RatingDelegate для редактирования значения рейтинга. При двойном клике делегат переходит в режим редактирования - пользователь может устанавливать значение рейтинга нажимая на нужную окружность мышью.

Запустите тестовое приложение снова и убедитесь, что все работает правильно. Попробуйте двойной клик мышью на рейтинге и измените его значение, чтобы убедиться, что делегат правильно отрисует новое значение.

**Взаимодействие с пользователем**

К данному этапу мы имеем готовы блоки программы: мы можем загружать, редактировать и сохранять данные. Можем отображать данные и позволяем пользователю взаимодействовать с ними. Теперь нужно собрать все это вместе.

Добавим диалоговые окна для ввода характеристик элементов нашей музыкальной коллекции.

Нам нужен диалог для добавления и редактирования исполнителя, диалог для альбома и диалог для трека (для трека можно обойтись без диалога - все его характеристики и так отображаются и редактируются в представлении). Если внимательно посмотреть и сравнить процессы добавления и редактирования элементов, то можно заметить, что добавление это есть редактирование еще не существующего элемента. Это значит, что для обоих операций можно использовать один диалог. Рассмотрим создание диалога только для одного типа данных - альбома. Для остальных типов данных действия будут аналогичными.

Нам нужно,чтобы диалог был осведомлен о модели, т.е. имел к ней непосредственный доступ. Тогда мы просто укажем диалогу индекс элемента модели и он должен будет уже сам получить по нему данные для элемента из модели и отвечать за сохранение измененных данных в модель.

Рассмотрим реализацию одного диалога, остальные делаются подобным образом.

Добавьте в дизайнере форму (Form class) в проект. Сделайте ее диалогом с кнопками и назовите AlbumDialog. Добавьте виджеты в форму - QLineEdit для ввода названия альбома и жанра, QSpinBox для ввода года выпуска альбома, QPlainTextEdit для ввода комментария и QLabel для изображения обложки. Вставьте кнопку для выбора файла с изображением обложки. Скомпонуйте все виджеты и проверьте что компоновщик работает правильно при изменении размера диалога. (наш диалог по сути является редактором “свойств” альбома. Как правило для настройки свойств используют компоновщик FormLayout)

Кнопка для выбора файла с изображением обложки должна вызвать слот. Слот, в свою очередь, должен использовать QFileDialog для получения пути к файлу и вставлять изображение в QLabel .

Чтобы диалог знал о модели и элементе, который он редактирует, добавьте в класс диалога два метода для получения указателя на модель и ссылки на индекс редактируемого элемента модели:

void setModel(MusicModel\*);

void setModelIndex(const QModelIndex &);

Перед тем как их реализовать добавьте приватную переменную m\_mapper типа QDataWidgetMapper в класс диалога, которая свяжет модель с виджетами диалога. Не делайте ее указателем, тогда не придется заботиться о ее создании и уничтожении.

Реализуйте методы setModel и setModelIndex. В методе setModel вызовете setModel() для объекта m\_mapper.

Метод setModelIndex чуть сложнее. Нужно сообщить мэпперу как об индексе элемента модели, так и его родителе.:

void AlbumDialog::setModelIndex(const QModelIndex &index) {

QModelIndex parent = index.parent();

m\_mapper.setRootIndex(parent);

m\_mapper.setCurrentModelIndex(index);

}

Можно добавить дополнительную проверку, что элемент является альбомом. Объект m\_mapper нужно инициализировать в конструкторе диалога. Следует определить связь между номером колонки в модели и виджетом в форме. Вызовите QDataWidgetMapper::addMapping() для каждой колонки в модели которую вы хотите отобразить в диалоге.

*QDataWidgetMapper можно рассматривать как представление , аналогичное QTableView, которое отображает только одну строку (или колонку) . Только вместо отображения данных в самом себе он использует виджеты.*

На этом этапе можно запустить приложение, чтобы проверить работу диалога. Убедитесь что передаете ему правильную модель и индекс. Вы должны увидеть в диалоге данные из модели и редактировать их. Добавим возможность отменить изменения, если пользователь нажмет “Отмена”.

Установите свойство маппера submitPolicy в ManualSubmit. Будем отправлять изменения в модель, только если пользователь подтвердил их. Переопределите метод accept() диалога следующим образом:

void AlbumDialog::accept() {

m\_mapper.submit();

QDialog::accept();

}

Теперь при закрытии диалога по кнопке “Сохранить” данные сохранятся в модель. Т.е. при вызове диалога не нужно делать дополнительных действий по сохранению данных в модель.

Если диалог будет использоваться для добавления новых элементов в модель, нужно сделать еще кое-что. Новый элемент нужно создать до того как отобразится диалог. Если в диалоге будет выбрано “Отменить”, надо удалить элемент. Это можно сделать внутри одного метода:

bool AlbumDialog::addAlbum(QAbstractItemModel \*model,

const QModelIndex &parent) {

setModel(model);

int row = model->rowCount(parent);

if(!model->insertRow(row, parent))

return false;

QModelIndex index = model->index(row, 0, parent);

setModelIndex(index);

if(!exec()){

model->removeRow(row, parent);

return false;

}

return true;

}

Вы можете написать похожий метод для редактирования альбома, но использовать индекс существующего элемента модели.

Проверьте создание нового альбома в тестовом приложении. Потом таким же образом реализуйте диалоги редактирования исполнителя и песни. Протестируйте их.

**Создаем приложение**

Последний шаг - собрать все части программы вместе. Создайте приложения на основе главного окна QMainWindow. Добавьте в него представление QTreeView как главный виджет окна приложения.

Создайте действия QAction для создания нового файла БД, загрузки и сохранения файла БД, добавления и редактирования элементов модели.

Используйте оставшееся время для добавления полезных “мелочей”, которые сделают пользование приложение более удобным: обработку события закрытия приложения, изменение заголовка для индикации не сохраненного документа и т.д. Добавьте то, что считаете нужным :)