Lab 4 – The Model View Framework

Aim: This lab will take you through the steps required to build a Qt application based on the model-view architecture. The focus is to understand the interface between the model and the rest of the application.

Duration: 3-4 h

Introduction

The goal of this lab is to explore the model view framework. You will have to deal with more complex situations than those from the exercises. Make sure to take your time to study each step and consult the reference documentation until you are sure you know what you are doing and why.

The task is to develop an application for storing information about music. The application will be built around a tree model storing songs structured in albums by artists.

This lab will be implemented according to the so called *bottom-up* approach. This means that you will start by creating building blocks that you later assemble into the complete application.

The drawback of this technique is that we do not have the complete picture of the program from the very beginning, so it is hard to track where you are going. It is compensated by the ability to easily test each component separately without interference from other parts of the final application. Testing is a very important aspect when it comes to the model view framework as custom model implementations can be error prone to implement.

In this lab it is assumed that you have already are familiar with the basics of creating Qt applications. This means that the instructions will not focus on things such as creating windows, setting up tool bars or adding slots to classes. Instead they focus on the specific task at hand.

When using the bottom-up approach, you need to have an overall plan for the final application. For this application you will need the following pieces:

- Data structures to hold information about songs, albums and artists with methods to store and restore them from permanent storage (i.e. disk)
- A data model that wraps the structures into Qt's model-view interface
- Delegates for custom rendering and editing of data
- Dialogs to conveniently manipulate data from the model
- A main application window with menus, tool bars and views that seamlessly brings all the other pieces together

The data structures

The application will be based around a tree of songs, albums and artists. Each of these information types will be represented by a class of its own with operations and attributes.

Artists are described by the following attributes:

Field name	Data type
Name	string
Photo	image
Country of origin	string
Comment	string
Albums	list of type 'Album'

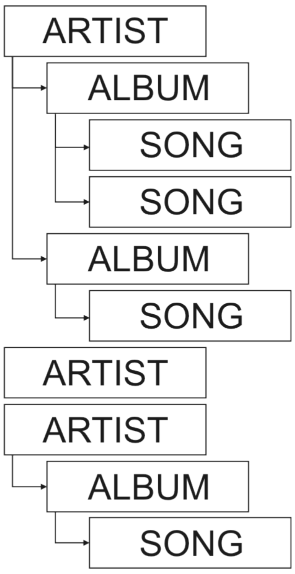
Each album record looks like this:

Field name	Data type
Name	string
Year	integer
Cover	image
Genre	string
Comment	string
Songs	list of type 'Song'
createdBy	Artist

A song entry has the following fields:

Field name	Data type
Title	string
Length	time
Rating	integer
Comment	string
inAlbum	Album

All these items are put together in a tree as the one shown below:



As all items will be a part of the tree structure, each of the items will be based on a common base class: `Item`. Create the `Item` class now.

```
class Item {
    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;
};
```

Here are some guidelines for implementing the the methods of the `Item` class.

- The constructor should create an empty, parentless object.
- The destructor should detach the object from its parent (by calling `setParent(null)`) and delete all its children. This will ensure that the consistency of the whole tree is kept intact as items in the tree are deleted, and that there are no memory leaks.
- The `insertChild()` method inserts the given item at a given position in the child list. It also sets the current item as the parent of the given item.
- The `takeChild()` method does the opposite, it removes the item at a given position from the list of children, detaches the item from the current object and returns the detached item.
- The `setParent()` method sets the parent of the item, but should also complement the functionality of the previous two methods. It should attach or detach the item from the parent's list of children. Be very careful to avoid infinite loops (`setParent` calls `insertChild` which calls `setParent`, etc.).
- The `childAt()` and `indexOf()` are a pair of methods for retrieving items at a given position or

retrieving the position of a given item. If the item is not in the list of children or the position is out of bounds of the child list, return `null` or -1 respectively.

- The remaining methods should be possible to implement without specific instructions.

After implementing the `Item` class, you need to test your work. To do that, write a simple `main()` function that creates a hierarchy of `Item` instances. Try adding items in different orders, removing them, adding the same items to multiple parents, etc.

In other words, make sure everything works before continuing with the next step of this lab. If you make a mistake here you will face a tricky debugging situation later on.

When your `Item` class works, the time has come to implement `Artist`, `Album` and `Song` classes as subclasses of the `Item` class. When you are done, test them in a similar fashion as before.

Later on you will need to be able to cast `Item` to its specific subclasses. You can either use `dynamic_cast` for that, or provide the following three virtual methods in the Item class:

```
    virtual Artist* toArtist() const;
    virtual Album* toAlbum() const;
    virtual Song* toSong() const;
```

Make sure that they return `null` from the item class and then re-implement the appropriate method in each subclass to return a real pointer when a downcast is possible, i.e. like this:

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

You can use these methods in the implementation of `Item::setParent()` and `Item::insertChild()` to make sure that you are only allowing Artists to be parents of Albums, Songs to be children of Albums, etc. This will work even better if you make those two methods virtual and re-implement them in the subclasses.

What is left regarding the data structure, is to provide serialization and deserialization methods. The idea is to implement streaming operators for a `QDataStream`. Remember to serialize and deserialize the children of each item in this operation.

```
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;
}
```

Again, write a simple program to make sure that the serialization and deserialization works before continuing.

Implementing the tree model

Now that you have a data structure for the application, you will have to expose it through the `QAbstractItemModel` interface. This means subclassing the `QAbstractItemModel` class and implementing all its pure virtual methods. In order to create a well behaved model, you will have to re-implement a couple of virtual methods as well.

This is a common case when using the model-view framework. You have a data structure specific to your application and you need to expose the contents as a model in order to be able to use the views provided by Qt.

The first step in creating your model is to have a reference to your data structure in your model. In case of trees, it is best to add an extra “invisible” item to the tree to serve as the root of the tree. It will not be accessible from outside the model (so it won't be seen in the view) and will not carry any data. Its only purpose is to provide a consistent way to handle all items in the model (so that we can assume that each valid node in the tree has a valid parent).

```
class MusicModel : public QAbstractItemModel {
private:
    Item *m_root;
};
```

Model traversal

The first method to implement is `QAbstractItemModel::index()`. Based on the row number, column number and index of the parent, it has to call `QAbstractItemModel::createIndex()` to create a `QModelIndex` instance for the item in question. The `createIndex` method accepts an optional pointer or number as its last parameter. Use it to associate a pointer to the `Item` instance corresponding to the model index so that you can retrieve it later. You do that using the `QModelIndex::internalPointer()` method.

The row and column numbers are not enough to decide which `Item` instance is the right one. You also have to consider the parent index. If you acquire the parent's internal pointer, you will have a pointer to its `Item` instance. You can then retrieve its proper child based on the row argument of the `index()` call.

If the method is called for a top-level item, the parent model index will be invalid and you can't query its internal pointer. This is where the extra root item proves useful. If you encounter an invalid parent model index, use the root item as its parent `Item` pointer.

```
QModelIndex MusicModel::index(int row,
                              int column,
                              const QModelIndex &parent) const {
  Item *parentItem = m_root;
  if(parent.isValid())
      parentItem = static_cast<Item*>(parent.internalPointer());
  // ... make sure row and column numbers are valid, etc. ...
  return createIndex(row, column, parentItem->childAt(row));
}
```

The next step is to implement the `parent()` method. The implementation pattern is similar to the one of `index()`, but this time the difficulty is not to get the `Item` pointer, but rather the row number of the parent. Use `Item::indexOf()` to ask the parent's parent about the position of the parent in its child list.

Two last pure virtual methods that you need to implement are `rowCount()` and `columnCount()`. They return the number of child rows and number of columns for each child. Using the information you used when implementing `parent()` and `index()` you will be able to implement them by yourself.

Remember that each column represents a different piece of data (excluding the bits that form the parent-child relationship). Looking at the data structure, you can tell that the number of columns depend on the type of item listed (artist, album or song).

If you implemented the casting methods suggested earlier you can use them now on the internal pointer of an index to determine the type of data structure it represents.

Exposing data

The last method that is pure virtual, i.e. has to be reimplemented by all subclasses, is the `data()` method. Without it your model will not be able return any useful information to the view.

The implementation is straightforward – you are given an index (and hence its `Item` instance, through the internal pointer) and a data role. You should return meaningful values at least for `Qt::DisplayRole`. For all roles that do not apply, return an invalid `QVariant`.

It is best to implement `data()` using `switch` statements operating on columns and roles. This leads to clean, readable and easily maintainable code.

At this point you can test your model class. Create a little application that sets up a `QTreeView` and assign your model to it. Do this in the constructor of the model load or hardcode some data.

When you run the application, you should be able to traverse your model as expected. If you application crashes, misses nodes or has ghost nodes, it means you made some errors in the model implementation. Make sure that the model works as expected before continuing.

Making the model editable

At this point your model is read-only. You know how many items your model holds and you can fetch its data. However there are no facilities to modify the existing data nor to add or delete items.

In order to enable editing of existing items, there are two tasks that need to be performed. First you have to re-implement `flags()` to inform the view that the model's data is editable. Simply return the `Qt::ItemIsEditable` flag along with the default flags for any index that is to be edit-enabled. Instead of enumerating all the flags from the beginning you can just use the logical OR operator to join the flags together:

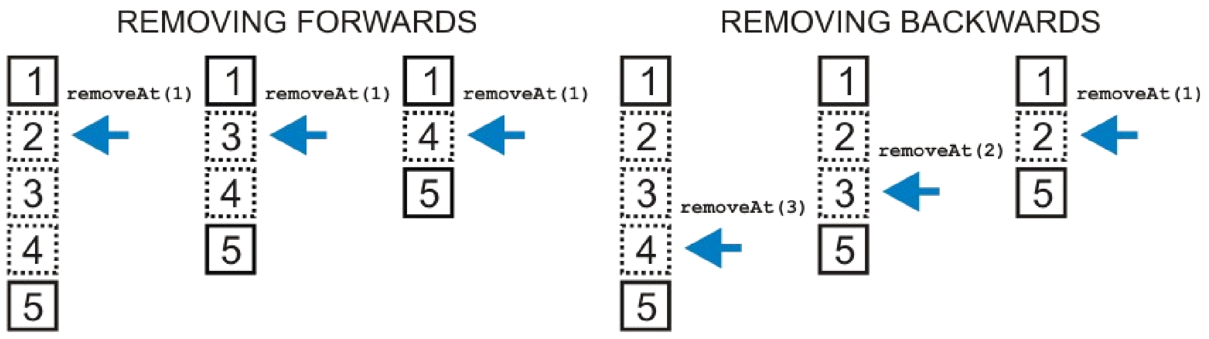
```
    return QAbstractItemModel::flags(index) | Qt::ItemIsEditable;
```

The second task to attend to is to re-implement the `setData()` method. This method actually updates the contents of the model. Implement this method so that it changes the data kept in the model when a `Qt::EditRole` or a `Qt::DisplayRole` is passed to it. Make sure you emit the `dataChanged()` signal for changed indexes before returning `true` from the method.

Now you can run your test application and try editing your test items by double-clicking cells in the model and altering the values. By adding debug output in the actual data structure, you can ensure that the correct setter method is called.

The model is now almost complete. The only issue left is to re-implement `insertRows()` and `removeRows()`. Adding or removing rows from the data backend itself is easy as you keep your data in a list. It all comes down to calling `QList::insert()` or `QList::removeAt()`.

When you remove a row, all rows after it are moved one cell. Look at the picture below, you can see what happens. You need to counter the effect in your code either by removing and recreating all items starting from the last item or by adjusting the indexes.



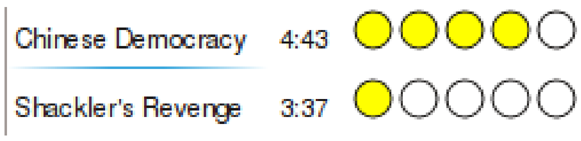
When implementing these two methods you need to pay attention to one more detail. The Qt reference manual states that when adding rows, before you physically change the underlying data structure, you should call `beginInsertRows()`. When you are finished with your modifications, call `endInsertRows()`. The same applies when removing rows. First call `beginRemoveRows()` and then `endRemoveRows()`. This is all required for Qt's internal bookkeeping and you must not forget to call those methods whenever you add or remove rows from your model. Otherwise the view might get out of sync with the model.

Model Test

The final moment when implementing a model is to test it to ensure that it does not break. Try to use the the monitoring and testing class, `ModelTest`, available from Gitorious here: *http://qt.gitorious.org/qt/qt/trees/4.7/tests/auto/modeltest*. When testing the model, try to interact with it to follow all of your main code paths.

Custom Rendering

In the `Song` class there is a field called `Rating`. It is used for keeping track of which songs the user likes. Showing it as an integer is possible, but boring. A better approach is to show it as a set of up to five circles, as shown below.



To customize the way a particular table cell is shown, you need to implement a custom delegate.

Create a delegate class called `RatingDelegate` based on `QStyledItemDelegate`. First calculate the size needed to render five circles. Assume a circle diameter of 16 pixels with horizontal spacing between circles of 4 pixels and a 2 pixel margin to all sides. Implement the `sizeHint()` method and return the calculated size.

In this situation you set an absolute size on each circle but you can make it relative to the font or cell size used by the view. The `QStyleOptionViewItem` argument passed to `sizeHint()` contains `fontMetrics` and `rect` fields that contain information about font geometry and cell height. You can use them to calculate the circle diameter for the rating.

The other half of the delegate is the rendering. To implement the `paint()` method properly you need to know two things: what to draw and where to draw.

You can get the rating value from the model using the provided `index` argument. Use `QModelIndex::data()` to ask the model for the value and convert it to integer using `QVariant::toInt()`.

To determine the geometry of the cell ask the `option` argument for its `rect` field. `QRect::left()` and `QRect::top()` will help you calculate coordinates for each circle.

What remains is to make a loop that will call `QPainter::drawEllipse()` with proper coordinates. Always draw five circles and fill the ones that represent the rating value with yellow color (`Qt::yellow`). Use a transparent brush (`Qt::NoBrush`) for the remaining ones. In both cases paint the outline using a black pen.

Now you can use `QAbstractItemView::setItemDelegateForColumn()` to set your newly developed delegate to the model column corresponding to the rating value and see if it works.

```
const int RatingColumn = 2; // adjust it to your model
RatingDelegate *delegate = new RatingDelegate(view);
view->setItemDelegateForColumn(RatingColumn, delegate);
```

When you run your test application you will notice that some of the cells contain empty circles instead of text. This is because you have set the delegate for all cells in a column. The delegate only applies to songs.

Correct your delegate class so that it falls back to the base class implementation for entities other than the song, or columns other than the one containing the rating. As before, the `index` parameter will help you determine which data structure lays beneath.

Run your test application again and see if your corrections work. Try double-clicking some of the rating cells and change ratings to see if the delegate updates the rendering.

Interacting with the user

Having come this far, the building blocks of the application are complete. You can load, edit and save data. You can visualize it and let the user interact with it. Now you need to build an application from these pieces.

The first step is to provide dialog windows to allow the application to force the user to focus on a single activity.

You will need dialogs for adding and modifying artists, other dialogs for albums and event more for songs. If you look closely and compare the process of addition and modification of item, you will notice that addition is similar to modification but operates on an item which doesn't yet exist. This means you can use the same dialog for both. The instructions will cover only a dialog for one type of data – the albums, the remaining ones can be implemented in the same fashion.

To save some work, you can make the dialogs model-aware. This means that the dialog will be smart enough to operate directly on the data in the model. You will simply tell the dialog which model index to work on and it will be the dialog's responsibility to fetch the data from the model and to commit changes to it afterwards.

We will now look at implementing one of the dialogs. You will then be able to extrapolate this information to the remaining dialogs.

Start by adding a Qt Designer Form Class to your project. Make it a dialog with buttons at the bottom and call the class `AlbumDialog`. Then add widgets to the form – line edits for the name and genre, a spin box for the year, a plain text edit for the comment and a label for the cover image. In addition to this, add a push button for changing the cover image. Place all the widgets in layouts and ensure that the dialog behaves as expected when resized.

> The dialog you are designing is meant to work on what we can call “properties” of an album. The `FormLayout` class is usually a good candidate for managing properties-like dialogs.

The push button for changing the cover image should invoke a slot. The slot, in turn, should use `QFileDialog` to get a file path to an image and load it into the label.

To make the dialog model-aware, you need to be able to tell it what the model is and which index you want it to operate on. To do that declare the following two methods in your dialog class header:

- `void setModel(MusicModel*)`
- `void setModelIndex(const QModelIndex &)`

Before implementing the new methods, add a private member variable that will do the magic for us: a `QDataWidgetMapper` class member object called `m_mapper`. Do not make it a pointer but a regular object, this way you won't have to worry about creating or deleting it.

Now implement the two methods declared earlier with code. Make `setModel()` call `setModel()` on the mapper object. The `setModelIndex` method is slightly more complicated. You need to inform the mapper both about the model index as well as its parent:

```
    void AlbumDialog::setModelIndex(const QModelIndex &index) {
        QModelIndex parent = index.parent();
        m_mapper.setRootIndex(parent);
        m_mapper.setCurrentModelIndex(index);
    }
```

You can add an additional check in the above method to make sure the index passed to the dialog is really an index representing an album. To do that, check the internal pointer of the index.

The widget mapper object needs to be initialized in the dialog's constructor. You need to define the mapping between column numbers in the model and widgets on the form. Call `QDataWidgetMapper::addMapping()` for every column in the model that you wish to show in the dialog.

`QDataWidgetMapper` can be treated as a view similar to `QTableView` which shows only one row (or column) at a time. But instead of showing the data inside itself it uses external widgets.

At this point you should be able to run the dialog in a testing application. Make sure to set it up with a proper model and model index. This should let you see the data from the model and edit it. However, the editing takes place live in the model. This is not the desired behavior, for instance the user might want to be able to cancel the dialog and revert the changes.

To fix that, you can set the `submitPolicy` property of the data widget mapper to `ManualSubmit`. Then commit the changes to the model when the dialog is accepted. To do that re-implement the `accept()` method of the dialog as follows.

```
    void AlbumDialog::accept() {
        m_mapper.submit();
        QDialog::accept();
    }
```

When the dialog returns, the changes are already submitted to the model. This means that the caller of the dialog does not have to take any action to update the model.

If the dialog is to be reused for adding new items to the model, an additional step is required. A new item has to be created before the dialog is shown. If the dialog is rejected, the newly created item needs to be removed. We can wrap it all up into a single method.

```
    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;
    }
```

You can implement a similar method for editing an album, but then you use an existing item.

An alternative to removing the newly created row after an unsuccessful addition is to re-implement `submit()` and `revert()` for the model. This lets you simply call `revert()` after the dialog is rejected. However, this makes the model more complex as you would have to track all changes to the data between consecutive calls to `submit()` and `revert()`. The approach you have taken is not as elegant but much easier to implement.

Extend your test application now to see if the dialog works properly. Then implement similar dialogs for artists and songs. Make sure to test them too.

Implementing the music database application

The last task is to put all the pieces together. Base your application around a main window. Add a `QTreeView` to it as its central widget.

Create actions in the window for creating new files, loading files, saving files and editing entries of the model.

Use the remaining time to add additional features to the application. Interesting features could be handling close events, document modification state, etc. Feel free to add what you feel is relevant.