A Multilayered Architecture for Qt Quick

Wednesday, September 18, 2013

The topic of application architecture rarely comes up in the context of Qt, because as Qt developers we tend to be more interested in the classes and objects (trees) than the application as a whole (forest). However, the architecture of a Qt Quick application is the difference between pain and joy. A good architecture makes C++ and QML integration seem natural, while a poor and muddled architecture makes it an exercise in frustration.

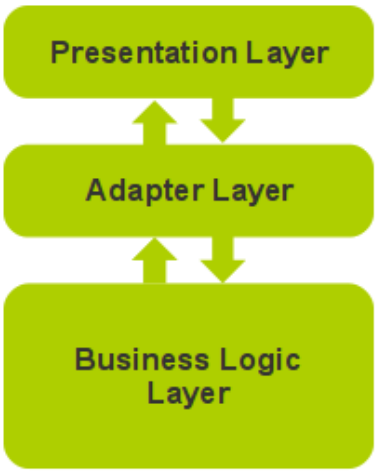
Multilayered Architecture

A multilayered software architecture is one that organizes different responsibilities into layers. Each layer has a broad role within the application, and communicates with other layers through well-defined interfaces. The architecture promotes high cohesion and low coupling, both hallmarks of good software design.

A layer could be a module within an executable, a separate process in a system, or even a physically separate system. The classic layers in a multilayered architecture are:

- Presentation Layer: The visible user interface
- Service Layer: Interprocess communication, frontend/backend bridge, etc.
- Business Logic Layer: Domain knowledge, business rules, etc.
- Data Access Layer: Databases, networks, files, etc.

A slight modification of this provides a good architecture for GUI applications. The Presentation Layer is the actual user interface, what the user can directly see and interact with. The Business Logic Layer is what we sometimes call the "backend". It's where all of our application logic resides. The Adapter Layer sits between the two and acts as a bridge, and might be either a thin interface or a thick service layer.



One benefit of this architecture is that the UI can be easily swapped out. A few years ago I worked on a project that shipped with three different UI frontends (in-process Qt, remote process Qt, remote HTML), with the same business logic backend. The same architecture also allows the backend to be swapped out for a testing harness.

This matches up closely to the classic Qt application model. The Presentation Layer is the UI form class created by Qt Designer, the Adapter Layer is the host QWidget that has ownership of the form class, and the Business Logic Layer is the rest of the application. The host widget controls all communication between the UI frontend and the application backend.

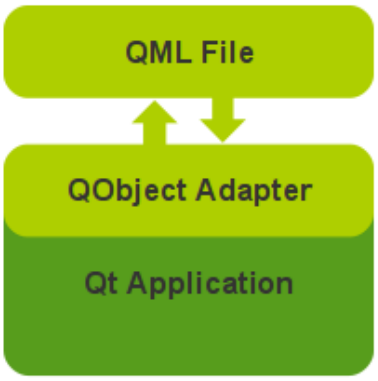
Integrating C++ and QML

Qt Quick provides four main ways to integrate C++ and QML. Each has its own place and purpose.

- **Subclassing QQuickItem**: QQuickItem allows you to write your own visual and non-visual QML items using C++.
- **Registering C++ types with QML**: C++ classes can be registered with the QML type system, allowing them to be instantiated as QML types.
- **Registering Context Properties**: QObjects can be registered with the QML context, allowing their properties to be directly accessed.
- **Accessing QML objects through the QML object tree**: All QML objects reside in a tree hierarchy and can be accessed via the root of the tree.

The last two methods are best suited for multilayered architectures. Registering a context property is best if the QML is pulling data from C++, and accessing the QML objects is best if the C++ is pushing data to QML. Both methods can be used together. For our architecture, we are going to stick with the last method, with the added restriction that C++ objects may only access the QML root object. This is the advice given in the Qt documentation, as the restriction avoids the danger of high coupling when C++ objects refer to QML objects inside the QML object tree. This method also turns out to be surprisingly simple to use.

The architecture will be examined in more detail using an example application. This is a simple desktop calculator application originally written for widgets, but ported to Qt Quick. It may be downloaded from here.



The Main QML File

There will be one main QML file that the C++ side will interact with. It will include the root of the QML object tree, which will provide a sufficient interface for the application to utilize. This includes properties and their implicit signals, other explicit signals and any helper functions. Anything below the root level of the QML tree should be treated as strictly private.

The QML communicates with the C++ by emitting signals. This is anonymous communication, as the QML has no way of knowing who has connected to its signals. It cannot directly communicate with any C++ object because none have been set as context properties.

```
import QtQuick 2.1
import QtQuick.Layouts 1.0

Rectangle {
    id: calculator
```

```
    property alias displayText: display.text
    signal keyClicked(int key)
    signal cutTriggered()
    signal copyTriggered()
    signal pasteTriggered()
    ...
```

In our example, the *calculator* item is the root of the QML object tree. The C++ backend only has access to its standard Rectangle properties (height, width, etc.) and to the *displayText* property, which is an alias for the *text* property of a child TextInput component. Four signals are also defined, and the backend will connect to these.

```
    focus: true

    Keys.onPressed: {
        if (event.modifiers & Qt.ControlModifier) {
            if (event.key === Qt.Key_X) {
                cutTriggered();
                event.accepted = true;
            }
            else if (event.key === Qt.Key_C) {
                copyTriggered();
                event.accepted = true;
            }
            else if (event.key === Qt.Key_V) {
                pasteTriggered();
                event.accepted = true;
            }
        }
    }
```

The root object has the focus and handles keypresses. Whenever one of the Ctrl+X, Ctrl+C, or Ctrl+V commands is pressed, it will emit the appropriate signals from the root item.

```
    GridLayout {
        id: mainLayout
        columns: 4
        anchors.fill: parent
        ...

        Rectangle {
            id: display
            color: "aliceblue"
            border.color: "cadetblue"
            ...

            Text {
                id: displayText
                text: "0"
                font.pixelSize: parent.height - 4*2
                ...
            }
```

```
        }
        ...
```

For simplicity, we will be using the new QtQuick Layouts to lay out the components in the calculator. Unlike a Grid positioner, a GridLayout will both position and size its child items. Note that *text* property of *displayText* was aliased earlier, so that it could be accessed from outside the QML object tree.

```
        Repeater {
            Button {
                id: key
                text: modelData.text

                Layout.preferredWidth: 48
                Layout.preferredHeight: 40
                Layout.fillWidth: true
                Layout.fillHeight: true
                ...

                onClicked: calculator.keyClicked(modelData.key)
            }

            model: [
                { text: "C", key: Qt.Key_Delete },
                { text: "/", key: Qt.Key_Slash },
                { text: "*", key: Qt.Key_Asterisk },
                { text: "-", key: Qt.Key_Minus },
                ...
            ]
        }
    }
}
```

Buttons are created using a repeater. When clicked they will emit the *keyClicked()* signal from the root item.

As should be clear by now, the QML is only concerned with the layout and presentation of the user interface. There is no business logic here, the QML does not perform any calculations on user input.

The QObject Adapter Class

The QObject adapter is the bridge that sits between the QML and the rest of the C++ application. It is the only C++ module allowed to directly communicate with the QML frontend. It may directly access any property of the root QML object, make connections to root's signals, and call any exposed functions. JavaScript functions should be limited in scope to functionality within the user interface itself, such as driving an animation or creating dialogs. They should never intrude into the problem domain, as that is the responsibility of the Business Logic Layer.

```
class Calculator : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString display READ display WRITE setDisplay NOTIFY displayChanged)

public:
```

```
    Calculator(QObject *parent = 0);
    ~Calculator();

    void setRootObject(QObject *root);
    QString display() const { return mDisplay; }

public slots:
    void setDisplay(const QString &display);

    void keyClicked(int key);
    void editCut();
    void editCopy();
    void editPaste();
    ...

signals:
    void displayChanged();

private:
    QObject *mRoot;
    QString mDisplay;
    ...
};
```

In practice, the adapter is very similar to the custom widget classes that were used with Qt Designer UI forms. The connections are typically created in the constructor or initializer, and much of the class is slot implementations that react to signals from the frontend UI.

Note that class defines a *display* property. If an object of this class were registered with the QML context, then the QML could directly access the *display* property. We are not using context properties however.

In this application, the adapter and the business logic reside in the same class, due to the simplicity of the program. In many projects it will make sense to keep them separate.

```
Calculator::Calculator(QObject *parent)
    : QObject(parent), mRoot(0), mDisplay()
{
    connect(this, SIGNAL(displayChanged()), this, SLOT(onDisplayChanged()));
}

void Calculator::setDisplay(const QString &display)
{
    if (mDisplay != display) {
        mDisplay = display;
        emit displayChanged();
    }
}

void Calculator::onDisplayChanged()
{
    // push the new display value to QML
    if (mRoot) mRoot->setProperty("displayText", mDisplay);
```

```
}
```

The constructor sets up a connection so that any changes to the *display* property can be forwarded to the QML frontend. The QObject property system allows us to access any property by name. When the *display* changes, it is pushed to the QML using the setProperty() call.

Note the anonymous nature of the property. The business logic doesn't care if the front is a QML Item or a C++ QWidget. Changing the adapter code slightly, the frontend could even be HTML or other technology. Even sticking to just QML, it makes it easy to change the frontend without touching any backend code.

```
void Calculator::setRootObject(QObject *root)
{
    // disconnect from any previous root
    if (mRoot != 0) mRoot->disconnect(this);

    mRoot = root;

    if (mRoot) {
        // make connections
        connect(root, SIGNAL(keyClicked(int)), this, SLOT(keyClicked(int)));
        connect(root, SIGNAL(cutTriggered()), this, SLOT(editCut()));
        connect(root, SIGNAL(copyTriggered()), this, SLOT(editCopy()));
        connect(root, SIGNAL(pasteTriggered()), this, SLOT(editPaste()));

        // set initial state
        clearAll();
    }
}
```

The main application will set the root object of the QML object tree for the adapter, as we will see later. At this point, the connections are made with the root, and a default state is set.

```
void Calculator::keyClicked(int key)
{
    if (!mRoot) return;

    double operand = mDisplay.toDouble();

    switch (key) {
    case Qt::Key_0:
    case Qt::Key_1:
    case Qt::Key_2:
    case Qt::Key_3:
    case Qt::Key_4:
    case Qt::Key_5:
    case Qt::Key_6:
    case Qt::Key_7:
    case Qt::Key_8:
    case Qt::Key_9: // digit pressed
        if ((mDisplay == "0") && key == Qt::Key_0) return;
```

```
        if (mExpectingOperand) {
            mDisplay.clear();
            mExpectingOperand = false;
        }

        mDisplay += QString::number(key-Qt::Key_0);
        break;
    ...

    emit displayChanged();
}
```

Clicking a button in the QML frontend will emit a keyClicked(int) signal, conveying key information. The application uses the Qt::Key enumeration as a convenience. Different keys will perform different operations that affect the display string. After the operation is complete the *displayChanged()* signal is emitted which will in turn set the *displayText* property of the QML root object.

```
void Calculator::editCut()
{
    // copy and delete
    if (mRoot) {
        editCopy();
        clearAll();
    }
}

void Calculator::editCopy()
{
    if (mRoot) {
        QGuiApplication::clipboard()->setText(mDisplay);
    }
}

void Calculator::editPaste()
{
    if (mRoot) {
        setDisplay(QGuiApplication::clipboard()->text());
    }
}
```

Here are the *editCut()*, *editCopy()*, and *editPaste()* slots. They make use of the system clipboard.

The *main()* Function

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

    // create view window
```

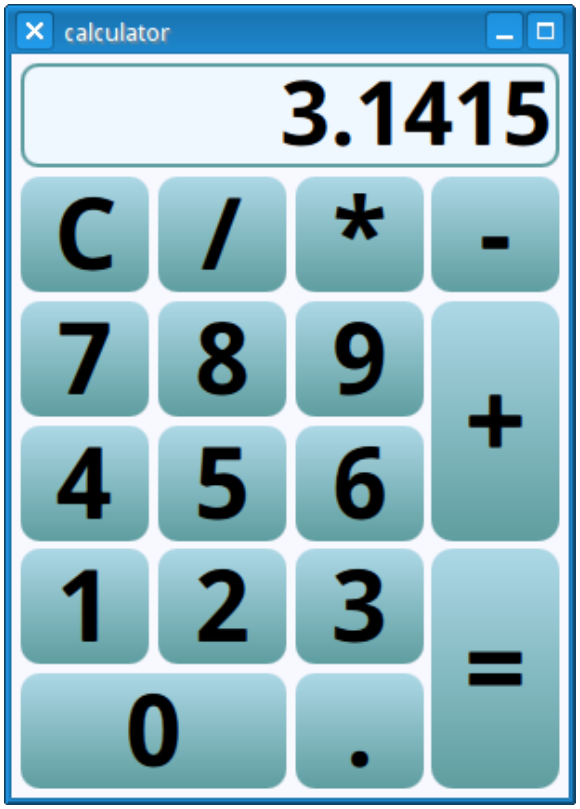
```
    QQuickView *view = new QQuickView;
    view->setSource(QUrl("Calculator.qml"));
    view->setResizeMode(QQuickView::SizeRootObjectToView);
    view->resize(300, 400);

    // create Calculator object
    Calculator calculator;
    calculator.setRootObject(view->rootObject());

    QObject::connect(view->engine(), SIGNAL(quit()), view, SLOT(close()));
    view->show();

    return app.exec();
}
```

The *main()* function of the application is straightforward. It creates a QQuickView for the main *Calculator.qml* file. It then creates a *Calculator* adapter and sets the root object.



Conclusion

The multilayered architecture is a perfect fit for Qt Quick applications. This architecture provides clean separation of responsibilities, and promotes good component design. It's also easy to use and makes integration of QML in C++ applications a simple pleasure rather than a complicated task.

About the author

David Johnson

Bio:

David is a Senior Software Engineer at ICS and certified Qt trainer. David has been actively using Qt since 1998 (with version 1.42). David specializes in desktop applications and data visualization. He has also contributed to the KDE and FreeBSD Open Source projects.

- More articles from the author
- David Johnson is on Google +