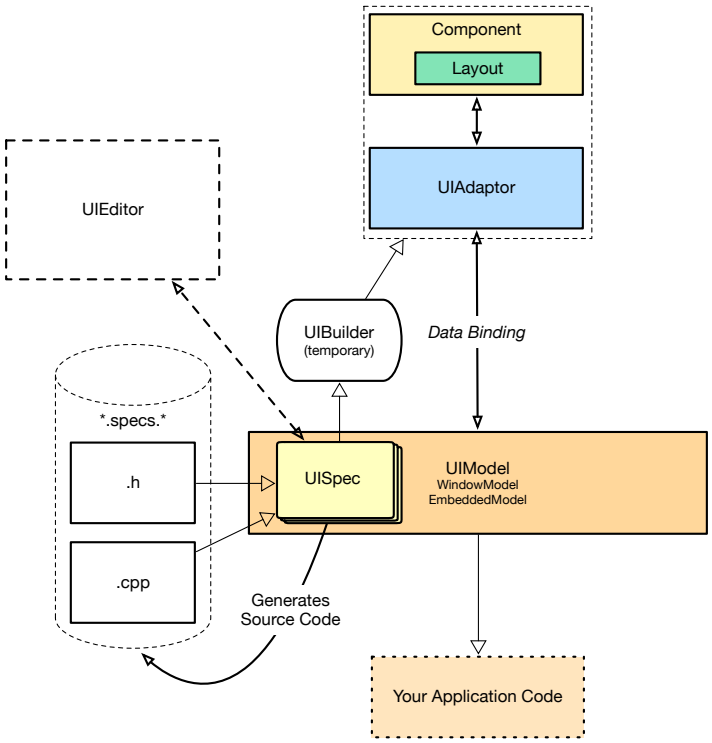
Experimental UI Framework

A proof-of-concept for a declarative UI framework based on generated code

Declarative UI frameworks are often based on XML, JSON or other representations, with concepts leaning on current web technology, like CSS, responsiveness and scripting. This experimental architecture is meant as a proof-of-concept for a possible alternative that uses generated C++ code instead. It is based on the Juce library.

The job of any UI framework is to separate application code from design and execution of the visual appearance (UI). This is achieved by making the UI an editable data structure. This proposed architecture leans on MVC terminology, albeit it only touches Model and View, while leaving the Controller part to the Juce Component hierarchy that it builds on.

In this architecture, any portion of application code that requires a UI derives from UIModel:



Structure

A typical user interface of an application would use many different UIModels behind the scenes. For example, one could have a dozen tabs and inside those more tabs, each of which are populated by a specific UIModel that runs some application logic and interfaces with a small part of the overall UI. Something like the Juce ColourSelector, for example, could be such a UIModel. It could even be composed of multiple models (e.g. RGB, HSV).

Of course, as a consequence, a granular subdivision like this entails some verbosity, both in the number of folders/files and the routine boilerplate required to manage the structure.

Ownership and Lifespan

Components are sort of "in the cloud": You don't keep references to them in your code anymore. In fact, it is not possible to access a Component from UIModel. This may seem odd and uncomfortable to many, but there could be multiple views or windows currently showing the same data and the only way to update them is to propagate a change, in response to which the affected components will update.

What you don't see doesn't exist

Components are created only when a UI of a UIModel becomes visible, e.g. when a tab is selected. As soon as another tab is selected, they are disposed again. This means there are no hidden components that drag on CPU with updates nobody can see.

Data Bindings

In order to have a generalised communication protocol between a model and its views (unless you want to treat every component type as an individual case), there needs to be a predefined "vocabulary" for

- Getting and setting contents, labels, or selection state
- Getting current enablement & visibility of a component
- Asking the model to populate a component with a UISpec
- Asking the model to build a custom component, or (re)configure a component

These are modeled as "Purposes" in the proposed architecture. Of course, not all purposes make sense for all Juce components. For instance, a Label has no selection state.

Highlights

- Layout is declarative and based on supplying components with a Positioner that responds to *resized()* at the top window level. It is prepared to support **FlexBox** and **LayoutFrame** (a relative rectangle with offsets and constraints), or any alternative scheme a Positioner can take care of. FlexBox is not implemented yet.
- **UISpec generates and compiles itself.** If you tell a UISpec to generate its own source code, it will overwrite its *.specs.cpp and *.specs.h header file. Next time you compile and run your app, it will continue from the last saved state.
- **UIEditor can edit UISpecs at runtime** and save changes by generating updated source code. This is achieved by including UIEditor with a development build. UIEditor is still a rudimentary skeleton only, though, and the whole cycle hasn't been tested in practice.
- UIModel and other classes make use of **class-side relection** (Metaclasses, "static virtual" methods) to provide access to the class hierarchy at runtime.

Performance

Building a UISpec from code is a one time effort. The native representation of the spec is cached for repeated access. Data bindings, once established, work through pointers to member functions. A possibly serious performance bottleneck is data transfer however, which in some cases involves copying the data for seemingly unavoidable technical reasons (see "Discussion" below for details).

Models, Dependents, Aspects

The **model** is your application code, components are its **dependents** and **aspects** are used to notify dependents about *what* has changed with your model, so they can respond selectively. Usually there are multiple dependents associated with an aspect, e.g. an aspect 'AudioSettings' might update an entire page of a dialog window.

Symbols

Traditionally, aspects are 'symbols' (unique strings, not unlike Identifier, however pooled at compile time already). Since C++ can't do that, this framework uses a class Symbol that mimicks the behaviour to some extent.

Symbols are globally unique names, pooled at application startup in SymbolTable. From then on they are basically integers, with all advantages of that primitive type. In a way, Symbols are not unlike a global *enum* that codes itself at startup and allows for its values to be converted to strings and back.

UISpec Example

The main model of UIEditor, for example, looks like this:

```
WeakReference<UISpec> UIEditor::MainWindowSpec = new UISpec
(
 UIEditor::Class::instance(),
 "MainWindowSpec",
 []()
 {
    auto window = new WindowSpec ();
    window->setLayout (LayoutFrame("50% - 600", "50% - 350", "0", "0").withFixedWidth(1200).withFixedHeight(700));
    window->setLabel ("UIEditor");
    window->addBinding (Bind::GetLabel (MEMBER (&UIEditor::getWindowTitle), SpecSelection));
    {
        auto content = new CompositeSpec ("content");
        {
            auto mainTabs = new TabsSpec ("mainTabs", TabContents);
            mainTabs->setOrientation (TabbedButtonBar::Orientation::TabsAtTop);
            mainTabs->pages.addPage (UIEditor::TabSpecModel);
            mainTabs->pages.addPage (UIEditor::TabSpecLayout);
            mainTabs->pages.addPage (UIEditor::TabSpecGraphics);
            mainTabs->pages.addPage (UIEditor::TabSpecResources);
            mainTabs->pages.addPage (UIEditor::TabSpecCode);
            mainTabs->addBinding (Bind::GetSelection (MEMBER (&UIEditor::getMainTabSelection), TabSelection));
            mainTabs->addBinding (Bind::SetSelection (MEMBER (&UIEditor::setMainTabSelection)));
            content->addComponent (mainTabs);
        }
        window->addComponent (content);
    }
    return window;
},
true
);
```

In the code snippet above, SpecSelection, TabSelection and TabContents are aspects. For more examples, please have a look at classes WidgetsExample and CustomExample in the source repository.

Discussion & Issues

Bindings

Bindings currently support a choice of data types and parameter passing modes, which are detected from the model's method signatures by some heavy template machinery. I doubt this is ideal. A better implementation might look for a way to do away with discrete 'types' and 'passing modes' and adapt to whatever method signature exists at both ends (or work with lambdas, a completely different approach).

Performance

Because C++ can't override member functions by return type (only by arguments), getting a value from a component requires the Binding to construct a default object and pass a reference to it to the component to be filled in. For small primitive types, this is not an issue. For non-trivial copy-by-value types, performance takes a hit.

This might be avoidable through && or ** or some convoluted mechanism, but I didn't bother diving into this, given my limited time and inexperience with advanced meta-programming.

Smart Pointers

Standard library smart pointers should be used instead of raw pointers and Juce equivalents. This change will trigger a cascade of dependent changes and add significant verbosity, which I avoided so far.

Philosophy

The MVC thing isn't probably for everyone. While elegant and consistent, the granular subdivision of a UI and the boilerplate it entails will likely appeal most to those who have worked with similar systems before. For the average audio plug-in or desktop app, sticking with hard-wired Juce components may be the preferred choice for most users.

General Limitations

Due to the template machinery, error messages issued by the compiler are sometimes confusing.

After all, C++ doesn't support reflection and this costs us dearly when it comes to anything 'declarative'. Any attempt at mimicking it can only go so far. This experimental approach went to great lengths to circumvent and/or hide some of the rigid static-ness. It is still ugly, requiring macros, templates and lots of dynamic casting, only to fake a little bit of the flexibility that dynamic languages provide out of the box. I found this quite frustrating and this doesn't bode well for me eventually using a C++ based GUI for major products.

Outlook & Future

In its current state, the proposed architecture is an experimental proof-of-concept, still incomplete and buggy.

While working on this for a couple weeks, I realized that doing a complex framework like this right requires more than a few iterations of the implement-get-stuck-redesign loop, which is a very time consuming endeavour (let alone the documentation). Realistically, it might take a man/year or two to bring this to production maturity. Since making libraries and frameworks is not my primary profession, it is unclear if and when the time will be available for pursuing this further. For the same reason, it is unclear whether it will make it into my products.

So please don't hold your breath. Developers looking for a framework to use in their current projects (which includes myself) may rather want to pick a third-party solution that is already tested, proven and supported.

Code

You'll find the current state of this on GitHub:

https://github.com/me-ans/UIModelFramework

Happy Coding,

Andre