MiscControllerKit

Preliminary documentation v01.2

Warning

This kit is a work in progress. If you have any ideas/complaints/suggestions, etc. please email me, or better yet, post to the misckit mailing list (I'm not sure what the address is but I think it just changed).

Briefly

The MiscControllerKit is a framework for building any type of applications. There are objects that try to make life easier for dealing with windows, inspectors, controller objects, info and inspector panels. For instance, shouldn't all apps have a default info panel without you even having to do anything? Then if you wanted to create your own, shouldn't it be as easy as telling the app manager which class or bundle to use instead? How about dealing with windows closing and alert panels? Well we take care of this and even more... How much do you think you'd have to pay to have all this? $1000?, $500?, $200? I guess you already know the answer since this is the MiscKit distribution...

The five main classes in the framework are the **MiscNibManager**, responsible for loading nibs, the **MiscWindowManager** which manages an NSWindow, the **MiscControllerManager** (subclass of MiscWindowManager) which manages instances of MiscController, the **MiscController** which is an abstract class for supporting the Controller in the Model-View-Controller pattern, and the **MiscAppManager** which takes care (or will take care of) of Info, Inspector loading and Preferences.
On top of this framework I hope to build the rest of the document framework from the MOKit (it sort of started a migration to the MiscKit but was never fully integrated) that Mike Ferris wrote.

Some Advantages of the framework

It reduces the number of outlets between the various objects that tie a nib's UI objects together. For example, instances of

MiscNibManager knows about all it's window managers. MiscControllerManager instances also know about their controllers. You can conveniently ask the MiscWindowManager class for any window manager in the app as long as you know it's name or it's class (it makes use of NSApp's windows for this). This makes it easy to get from the WindowManager in your document to the WindowManager for the inspector, etc.

There is some logic for deciding when a window should be displaying the broken "X" (document edited) or not. When an edit in the UI takes place the responsible controller tells it's controller manager (subclass of MiscWindowManager) to update the window's document edited status. When a document is reverted or saved the "edited" flag is cleared.

You get an alert panel when you try to close a "edited" window. The title, message, and button titles that appear on the panel can all be easily changed by subclasses of MiscWindowManager.

You get automatic window cascading. (I'm not sure that this works correctly yet)

Nib managers release their controller managers, which release their controllers. You can also configure the nib manager and window manager to release itself and it's window is closed.

App Manager

When the app is about to terminate we close all our open windows and allow the user to save any edited documents.
When the info menu item is clicked the user will at least get a default info panel displaying the app icon, application name, and version. It is also very easy to create a customized NSBundle for the info panel or use a subclass of MiscInfoWindowManager and have it loaded instead of the default info panel. In a later release there'll be similar functionality for inspectors too.

Nib Manager

Manages a nib resource as the nib's owner. It's a very lightweight class that just encapsulates the loading of a nib.
I've recently figured out that loading the interface is merely a detail and shouldn't play too much of a role in the Document framework design. When you've loaded the nib all you care about are the objects in it. Because of this, I've made the main object in the nib (in my case so far it's been the MiscWindowManager) make use of the MiscNibManager to load itself (and the other objects in the nib)

and have no other object in the application know about the MiscNibManager that loaded the nib. I'm starting to think that this is definitely the way it should be. Take a look at the AdderWindowManager's **newAdderWindowManager** method for an example of what I'm talking about.

Window Manager

Manages a window as long as it's the window's delegate. Allows easy setup of auto window saving, window cascading, releasing itself when the window closes, etc. Using the MiscWindowManager class you can also get a list of all current MiscWindowManager instances. You can also have named window managers (for instance "InspectorWindowManager") so others can ask the MiscWindowManager class for a window manager by name. Right now, this class is in a constant state of flux. Some of it's functionality that is just as easy to manipulate by asking for it's window will probably be eliminated.

Controller Manager

A subclass of window manager that manages the controllers that are using the window to display their contents. Keeps track of (and releases) all known controllers. This allows other controllers in the window to message any other controller also in the window (as long as you know it's name or class) without keeping an outlet to it.

Controller

The controller is the same one from the basic Model-View-Controller paradigm. It is for displaying model objects. It can notify the controller manager when it is dirty and the window will be updated appropriately, etc. You can also notify other objects (usually other MiscController instances) using the MiscControllerDidEditNotification.

There's probably other good stuff that these classes take care of for you that I forgot to mention, so take a look through the documentation for each of the classes.

General Notes

You'll notice that some classes end in Controller and some end in Manager. Any object that just helps out or otherwise takes care of another object ends with Manager (ie: WindowManager, ControllerManager). Controller is reserved for classes that fit with the Model-View-Controller sort of outlook. The controller's job is then to move data from a Model object into a View object.

I'll just briefly explain my decisions for the seemingly strange code below:

```
@interface MyObject : NSObject
{
#if IB_HACK
    id window;
    id button;
#endif

@private
    NSWindow* _window;
    NSButton* _button;
}

- (NSWindow*) window;
- (void) setWindow:(NSWindow*)aWindow;
- (NSButton*) button;
- (void) setButton:(NSButton*)aButton;
```

First of all I decided to add "@private" directive to all my ivars so that subclasses couldn't directly access them. Then I found that InterfaceBuilder coudn't parse any instance variables past the @private directive. Strike one. Then I thought it also be a good idea to put underscores before each instance variable name so I wouldn't have the problem of in a method declaring a local variable name the same as my instance variable and getting that annoying "Local declaration hides instance variable."
I also realized that if I have an underscored ivar named _window, IB doesn't look for setWindow: it looks for set_window:. Strike two. Therefore I put in the #if/#endif statement to get around both of the above problems. IB will parse this header and get two outlets

window and button. It will then use setWindow: and setButton: when the nib is loaded which sets the real ivars, _window and _button.