Release 0.1, 07.04.1994 Copyright © 1995 by Thomas Engel (tomi@shinto.nbg.sub.org)

A Swapping Tutorial

This tutorial tries to give an impression on how to work with the MiscSwapKit and the MiscInspectorKit. These kits were designed to ease the coding and prototyping of mulitpage-views and inspector-style windows. It may not really cover all the technical details.

Intro

This document discusses the ideas behind the MiscSwapKit and MiscInspectorKit. It also shows how some standard swap-situations inside an app can be solved. Right now there is no documentation for individual classes. The header files and source code are fully documented and together with this tutorial and the examples it should be possible to use the objects.

There are only some sample inspectors inside this tutorial because there was not time to write a surrounding app. [Note: The UNIXHaters example app shows use of the new SwapKitPalette.] There is a SwapKit example called *NewAppInspector*. It is a Workspace inspector that tries to inspect .app directories. Hopefully this inspector will be finished soon; for now there is other work to be done. If you want to see how both kits work inside a real program, look at BeakerBoy from the archives. The swap and inspector kits were derived from this chemistry app, so it provides some example code.

The MiscSwapKit

This kit is designed to make it as easy as possible to share one region inside a window to display multiple differnet views. I have tried to ensure that you don't need to leave IB when you just want to build a prototype of a swapping area. The MiscSwapKit consists of a few objects:

MiscSwapView

This is the main object of the whole kit. It is a view subclass that can set its contents to another

view (`setContentView`:). It is kind of stupid because you have to tell it exactly what view to take. There are two different ways of swapping in.

- **Buffered**. Does not allow resizing, uses more memory but it is faster. (It might cause problems when used with double screen systems. Anyone with a ND and a b&w screen around ? There might be problems when moving the view from b&monochrome to color etc..)
  The windows that hold views which will be swapped in should *not* be deferred !

- **Redraw**. Does allow resizing, uses less memory and is slower but should no cause any problems. This is the default behavior and should be used unless not acceptable.

Views that can not be resized are aligned bottom-left.

MiscSwapView_ByContents

This category to the simple SwapView does add a basic mechanism that will enable you to do some kind of automatic swapping. This is easy in most cases because views don't swap without a reason. Mostly swapViews are used to change the UI when the user makes some actions

(pressing a Button, choosing an item of a popUp list, etc.). The user interface element that is used to cause an action is called a *trigger*. The most common situation is that there is one trigger that will cause one view to appear.

To be able to do this we need to know two things:

- Which *trigger* (button, cell,¼) caused the swap request. In general this is the sender of a `swapContentView:` action method.
- Which *view* has to be swapped in. This information is provided by the contentsControllers (see later) that register at the responsible swapView.

The mechanism for finding the right view and contentsController is very simple here. It just compares the real trigger to the triggers the registered contentControllers feel responsible for. If they are the same the found contentController is asked for its view. And ths view will be displayed next.
When working with multiple NIBs (common when working with inspectors) you can't establish real object connection accross those NIBs. So there is no way to set the 'real' trigger for a contentController.
To solve this porblem (and to ease the localization of apps) the swapView can use tag-

comparison too. This is the default behavior and if the trigger has a tag >0 this method will be tried before real object comparison is done. You should never have to deal with disableing tag-comparison.

MiscSwapContentsController

To minimize the effort necessary for building multiview window we have this class. It has all the necessary plugs and methods in there to comunicate with the swapView. This controller knows the view that should be displayed and the trigger that cause the swap. To allow automatic setUps the controller also knows about the swapView so it can register there on its own.

When working with tags-comparison the controllers trigger might be of a completely different class than the real-trigger. Only the tag has to be the same!

The contentsController does provide `ok:`, `revert:` methods by default.
It also has some `will...` and `did...` methods.
The `willSwapIn` is used to trigger a automatic `revert:` before the view will appear. This gives you the chance to update data inside the view. All you have to do is to code the right `revert:` method.

The `willSwapOut` should be used to leave the scene gracefully. Remember to deactivate colorWells when you get swapped out. etc. pp.

MiscSwapViewByPopUp

A subclass of the MiscSwapView that will allow working with popUps. PopUps don't send the cell ID as the sender reference but they pass the popUps ID instead.
This object deals with that problem.
It also enables you to do all the settings inside IB.

MiscSwapViewByMatrix

Another subclass of the MiscSwapView that deals with the problem of matrices being the trigger source. Matrices do not send the cells ID as the swapContentView: action. So we have to find the cells ID ourself.

How it all works together

So before the swapping can work:

- All the triggers have to target the swapView and send the `swapContentView:` action method when they are activated.
  Using a MiscSwapViewByPopUp does only require to connect the swapViews popUpCover outlet to the right popUpCover.

- Every *contentsController* has to add himself to the swapViews controllerList. This is the list of objects that our swapView will ask whether they want to get swapped in by a certain trigger or not.

- Every *contentsController* has to know about his trigger and relating view. You also need a connection to the swapView to allow automatic registration.

All the above settings can be done inside IB and will be stored in your NIB providing all the information needed to enable automatic swapping like this:

- First there is a button, cell etc. that sends the `swapContentView:` message to the swapView.

- The swapView searches the controllerList for a controller that feels responsible for the triggers tag or the trigger itself.

- Having found a controller will cause the swapView to ask for the view to display. If there is no controller it will cover its content area with backgroundGray.

- If it has found a view it will ask itself to `setContentView:`

- The old contentsController will get a `willSwapOut` message to leave the place. Whereas the new contentsController will get a `willSwapIn` message to get a chance to update the views data.

This is the whole story.

A simple example

I have included the pre-pre-alpha version of my NewAppInspector workspace-inspector. It demonstrates the use of the swapKit.
Just open the project and take a look at the NIB while you will try to follow the steps described above.
As you will find the only real code needed is to provide the functionallity desired of the single

contentAreas (revert methods).

....ProjectAdded here in compress fromat!!

The MiscInspectorKit

Inspectors are part of almost every program. They give detailed information on a current selection. The kit is build on top of the swapKit concept and should give you a flexible approach to automatic inspector handling.
The method names are as close to NeXT inspectors as possible. All the other methods were checked for consitency against other inspectorKits. I hope I did choose good ones.
You can build inspectors that handle one categorie (Attribute, Misc, ¼; I will stick to this 'category' meaning in to following text. Do't mess it with Obj-C categories!) for a single or many selection obejcts-types. You might also also create wrappers that appear as mulit-category-inspectors to the outside.
Example: Imaging that you have a *full* set of inspectors for every inspectable category of a selection that is `kindOf:` *animal*.
When you want to inspect a *horse* it may only need *one* new Misc-category inspector.

But all the others will work like they did for every animal.
The NeXTSTEP workspace has something similar (only custom contents-inspectors).

The design of the MiscInspectorKit leaves many possible ways of customization. Changing popUp titles at runtime and on demand for single inspectors might be an example (also I don't like the idea in general). But you have to deal with those things on your own. It might help if you always keep in mind that it is very similar to simple swapping.

MiscInspectorManager

The central point to the whole kit. Here is all the kownledge of how to choose an inspector and how to handle selections. The app has to inform this object of any changes inside its selections. The manager by default has a window with a popUp and a swapView plus a OK and Revert button. The whole stuff is inside the Inspector.nib (see later).
You may hide the okRevertButtons with the given method or you may just remove them from the NIB.

To find the right inspector the manager searches its inspectorList one by one. If an inspector handles the selection it may register its views for the category it likes to.

The last inspectors of the list has the highest priorty. Which means that later ones can override earlier inspectors.

MiscInspector

To control the contents of a single inspector page (it can only control one category!) we need a object. Because the InspectorManager uses the a swapView it is clear that this is a subclass of the MiscSwapContentsController.
The MiscInspector adds methods to a contentsController that enable it to comunicate with the MiscInspectorManager and to find out about the selection.
The methods are fairly simple¼ really NeXT like. The only difference is `doesHandleSelection`. Here you have to find out if you want to handle the current selection or not.
The MiscInspectorManager uses this methods to find out about which inspector to use for inspecting his current selection.

You are responsible for loading the NIB that holds the interface definition. It is not really nice to create one NIB for one category. It will be a mess. See MiscInspecotrWrapper for a solution.

(*Remark*: Nobody forces you to have an inspector only inspect a single object or only a multiple

selection of the same type. You are free to decide whether the selections is good or not. You can have a inspector that can inspect lists of horses, dogs, cats, and birds mixed together in any style.
But for sure. Your code will have to take care of that when doing manipulations to those objects. You get what you asked for¼)

MiscNoSelInspector,
MiscMultiSelInspector

There are some common inspectors that almost every app will need. These inspectors use the predefined views inside the main *Inspector.nib*.
They are included by default through the MiscInspectorManager inside the `addDefaultInspectors` method. If you really don't like this behavior you might just remove (or don't add them) them from the list. But do it OOlike. Subclass MiscInspectorManager and implement your own `addDefaultInspectors`.

These inspector may show you how do write MiscInspectors that work for multiple categories without a wrapper object.

MiscInspectorWrapper

In the real world you will have one inspector for every category (Attributes, Style, Misc) for every type of selectable object (Cone, Spline, Square).
So you will end up with 9 inspectors. But when you take a look at them¼every single inspector a certain object type does use the same `doesHandleSelection` method. This is clear because they are for the same selection type. But every single inspector will need its own NIB to hold the view.

This is not nice and our wrapper solves that problem. It has a `doesHandleSelection` method and looks like a multi-category inspector to the outside. But it does only wrap many single-category inspectors, freeing them form checking the selection on their own, using less space (only one inspector registered at the manager) and allowing us to group all the inspectors inside one single NIB.
All you need to do is to connect the manager outlet of every inspector (MiscInspector) inside the NIB to the filesOwner (MiscInspectorWrapper)

Inspector.nib

This NIB contains the main window, popUp and swapView. It also includes the views for the three deault inspectors.
This NIB will be loaded when a inspectorManager is initialized. This way you might use keyboard commands to make the inspector key or main¼by simply choosing a desired category. You will have to copy this NIB into every project you are using the MiscInspectorManager.

If you are not satisfied with the Inspectors size etc. Just change it. But leave the connections as they are.

The basic feeling

Well if you take a look at the basic problem you will notice that it's always the same:

- You launch your app.
- The MasterInspector (subclass of MiscInspectorManager) does the app-specific settings (like adding the defaultInspectors¼ see the MasterInspector example) and loads the main NIB.
- You make selections inside your program (because we're inside NeXTSTEP they should be objects of some kind)

- You open the Inspector panel.
- You select a topic from the popUp list that sets the category of information you are interested in.
- The InspectorManager has to find the right inspector for a given situation (categroy, selection).
- Every inspector that does handle the selection has the change to establish his connection to the swapView (by adding to the swapViews contentsController list).
- There can be only one contentController for one trigger (category item). So the last inspector overrides all the previous inspectors.
- Like the normal swapping. The inspector gets: `willSwapIn` and can do his `revert:`.

As you can see there is little magic about that.
Remember that the this kit depends on the MiscSwapKit.

A step-by-step example

Sorry. This is not really an example. I have placed some sample objects from my BeakerBoy project here. So maybe you can get an impression.
I will try to give you a step-by-step walktrough. Please look at the samples while you read the

steps.

Actor:

The MasterInspector:	MasterInspector.h ¬MasterInspector.m ¬
The InspectorWrapper:	AtomInspector.h ¬AtomInspector.m ¬
The AtomGraphicsInspector:	AtomGraphicsController.h ¬AtomGraphicsController.m ¬
The AtomInspector NIB:	AtomInspector.nib ¬

Create a MasterInspector. This is needed to install the inspectors. All our custom InspectorManager does is to alloc the single inspectors. In our case we are using wrapper here. But it doesn't make any difference besides the fact that the wrapper does the NIB loading for the related views and not the single inspectors (that are covered inside our NIB)

The app set a selection. By invoking `inspect:`*anObject* the app can ask the manager to find the right inspector for that object. The manager will now start with the NotAppInspecotr and will end with the CameraInspector (which is wrapper).
Now let's assume that the user did choose the *graphics* category and the object to inspect is

some kind of *Atom*.

Using the defaults. as the manager passes its inspectorList it will start with the *NotAppilcableInspector*. This one will apply to every selection and every category available. The next is the *NoSelection* and *MultipleSelection* stuff. They won't do anything because actually we have a single selection at hand.

Using app-specific inspectors is the next step. On its searchpath the manager will find the *AtomInspector* that feels responsible for a single atom too. This is a wrapper as you can see. At the time it finds that there is a selection worth of our attention it will load the NIB which contains the views and objects of all the single inspectors specific to that object-type.

Setting up the Inspectors. The manager asks the wrapper to `addWrappedControllers:`. This will cause every single-inspector to register at the swapView for the trigger they are connected to inside the NIB. As we are using tag-comparsion (multi-NIB force us to) the triggers are simple Buttons here.
With this step we have overridden the NotApplInspectors that did already register for those categories. They will now only remain for the categories that are not covered by a specific atom inspector (like Misc and Info in our case).

Swapping between categories does not need the manager anymore. This is exactly the same as using a simple swapView. So one might say that the MiscInspectorManager is only a class the can automatically set the contentsControllers of a given swapView according the some given selection.

See you later...

There is still some work to be done. Frist all those objects need their own, HeaderViewer proof, documentation. Maybe better examples would be good too.
I'm not very good at writing docs. So any kind help is appreciated.

There are some technical changes that would be nice to have:

- Building palettes.
- Loading inspectors from bundles.
- Enableing various alignments inside the MiscSwapView (Currently only: bottom-left)

These objects helped me a lot. I hope they might be useful for you too. Although this tutorial may

not be perfet it might give a place to start from.

Aloha,

.Unterschrift.tiff ¬