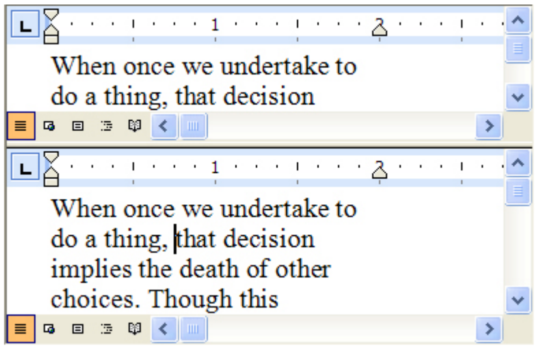
6

Multiple View Models

In chapters 3 and 4 we looked at how the controller makes changes to the model, the model notifies the view and the view notifies the windowing system of damaged regions. The windowing system calls the view to redraw damaged portions of the display image from the current state of the model. We looked at the model-view-controller architecture from the point of view of a single model, single view and single controller. However, the MVC architecture is more general than that.

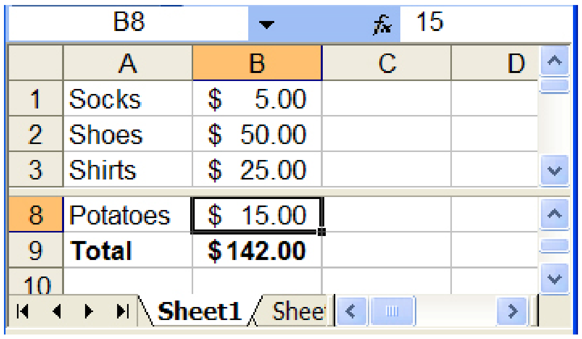
In many situations there are many different kinds of views associated with a given model. Figure 6.1 shows the split screen feature in Microsoft Word. Any edits in the lower window where the selection is shown, will also be seen in the upper window. Many times this split screen feature is used to present different parts of a document. In such a case the changes in one view will not be seen in the other. However, the user is allowed to position the split views any way they want and our software must work correctly regardless of the positioning.


When once we undertake to
do a thing, that decision

When once we undertake to
do a thing, that decision
implies the death of other
choices. Though this

Figure 6.1 – Word with split view

Figure 6.2 shows an Excel spreadsheet with a split view. This allows items in a shopping list to be viewed above while the total is shown in the window below. Though these two views are presenting different parts of the spreadsheet, changing the price of socks in the upper view will change the total in the lower view. This example also demonstrates the sharing of selection among the views. A cell in the lower view is selected, but the column selection is shared between the upper and lower views. Figure 6.2 also demonstrates small helper views that are tied to the same model. The "B8" in the upper left is an independent view of the currently selected cell. It must be updated whenever a different cell or range of cells is selected. The upper right of figure 6.2 shows the actual contents of the selected cell. This might include formulas or other information. This is a different view of the same cell contents.



B8 fx 15

	A	B	C	D
1	Socks	$ 5.00		
2	Shoes	$ 50.00		
3	Shirts	$ 25.00		
8	Potatoes	$ 15.00		
9	**Total**	**$142.00**		
10				

Sheet1 / Shee

Figure 6.2 – Excel split windows

Figure 6.3 shows two views from Adobe Photoshop of a middle-aged woman's cheek. The view on the left is the original image with a region of the cheek selected. The view on the right is showing a filter of that region and how it will look after the filter has been applied.

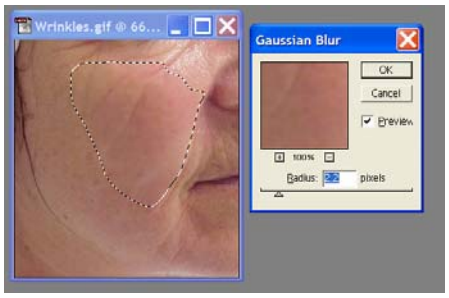


Figure 6.3 – Photoshop filter views

A more extensive use of multiple views is shown in figure 6.4. This shows an art piece being developed in Adobe Photoshop. The image is higher resolution and size than the display screen and it consists of many different layers used to construct the final image. The artist must be able to manipulate all of these facilities while retaining a sense of the final result.

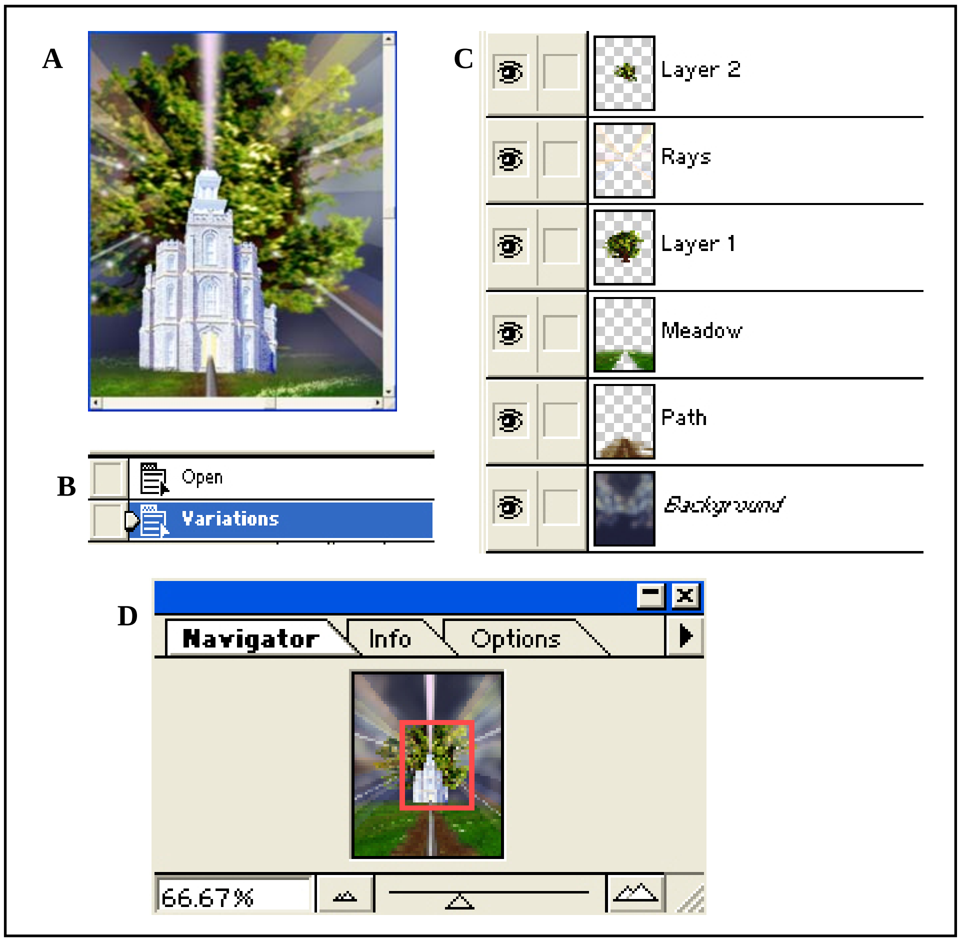


Figure 6.4 – Photoshop image model windows

View A shows the resulting image with all of the current settings. The user may paint directly in this view. View B shows a history of all of the changes to the model. By selecting items in this history, the artist can change the appearance of the picture to reflect only the changes made up to that point. The change history is also a part of the model. View C shows all of the layers from which the picture is composed. Each layer shows a thumbnail of that layer's image. These thumbnails are fully live views of their respective portions of the picture's model.

Clicking on the eyeball icon of a layer in view C will cause the contents of that layer to disappear from the composite image in view A. Selecting a layer from view C will dictate the layer in view A where paint should be placed. Lastly view D shows a navigator view that displays the entire picture as well as a rectangle showing the current pan and zoom used by view A. Moving the rectangle in D will change the image in A.

Review of Model-View-Controller

In each of these examples, there are complex relationships among multiple views of the same model. What we need is a software architecture that makes all of these complexities work together without programmers needing to deal with all of the interrelationships. The model-view-controller architecture provides us with just that mechanism.

Chapter 3 describes the following steps for implementing model-view-controller:

- Implement the model as a class with public methods for the controller to change the model and for the view to get access to model information.
- Define a listener interface with methods for reporting all model changes to any other object that may need to be aware.
- Implement addlistener() and removelistener() methods in the model so that listeners can register themselves.
- For each method *M* in the listener interface, implement a notify*M*() method in the model that will loop through all listener objects *L* and call the corresponding *L.M*() method on each.
- Have the view implement the listener interface and call damage() on any screen regions affected by each change method.
- Register the view with the model using the addlistener() method.

Most of what is necessary to implement interfaces like those in figures 6-1 through 6.4 are found in these six steps. These examples point to three different techniques in the use of multiple views: 1) differing views for multiple purposes, 2) multiple views with differing view controls, and 3) views with synchronized selection. Most of the differences in software architecture lie in deciding what information goes into the model and what information is retained in the view/controller.

The differing views such as in figure 6.4 are handled by several view classes that implement the model's listener interface and are all registered with the model. Each makes its own changes to the model and responds to changes made by it or other views. Note that to make the relationships in figure 6.4 work correctly the model must contain more than an image. It must contain an image change history, the current pan/zoom rectangle and the image layers. All of this information is shared among the various views with each view/controller manipulating and presenting various parts of that information.

Figure 6.2 points out a special problem that is easily handled by model-view-controller. When a change is made to cell B2 the spreadsheet model will recalculate and cause cell B9 to change value. B2 and B9 are displayed by different views. Attempting to synchronize directly across the two split views will create a tangled web of notifications. However, having the model report changes on any cell that it changes during recomputation creates a relatively simple solution that always works correctly. Both the top and the bottom views will receive change notifications for cells B2 and B9. The top view will damage a small rectangle for B2 and ignore the change from B9. The bottom view will damage the rectangle associated with B9 and ignore the notification from B2. However, if the top view was larger so that B9 appeared there also, it would damage both B2 and B9 because it receives both notifications.

Multiple views with differing view controls

Figure 6.1 shows the split window feature of Microsoft Word. The splitter pane is a type of widget with a horizontal or vertical line across the middle the widget controller allows the user to drag this line to adjust the relative size of the split windows. In figure 6.1 each of the views are an identical class. Each of them is registered as a listener on the same document model as shown in figure 6.5. At a given time, at most one of these widgets can have the key focus and receive keyboard input. Each widget keeps a flag indicating whether it has key focus and where the text insertion point should be. This information is independent of the model. When one of the windows is selected, it requests the key focus and the previous window is notified that it has lost key focus. In this example the focus and selection point are maintained independently in each view, only the content of the document is shared in the model.

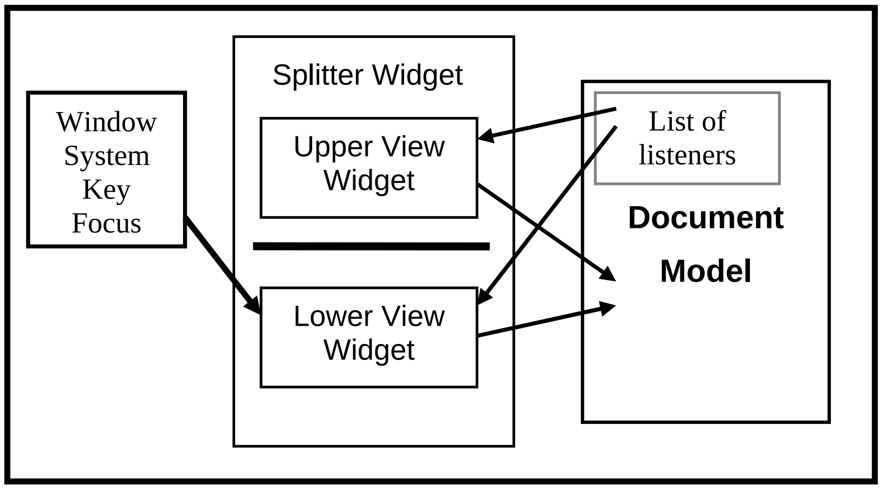


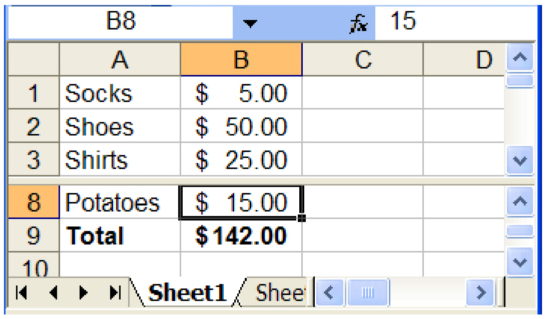
Figure 6.5 – Multiple view listening

Each of the document views in figure 6.1 also has its own scroll position. This allows them to be positioned independently. By not sharing scroll position in the model, the user can work in two places in the same document at the same time.

In the figure 6.1 example a splitter widget is used to organize the two views. This is not necessary in general. If the application would allow the document to be opened in more than one window, the architecture from figure 6.5 would allow virtually any application to edit and view in multiple places at the same time. However, the MVC model listener architecture must be built in from the beginning. Many applications, including Word, do not allow an arbitrary number of windows to be open on the same model, and thus a useful feature is lost.

Synchronized selection

There are many situations where similar views should work together. Figure 6.6 again shows the split windows of an Excel spreadsheet. In this application the selected cell is shared by both views in the model. The horizontal scroll position is also shared by both views. Each view has its own vertical scroll position.



B8 15

	A	B	C	D
1	Socks	$ 5.00		
2	Shoes	$ 50.00		
3	Shirts	$ 25.00		
8	Potatoes	$ 15.00		
9	**Total**	**$ 142.00**		
10				

Figure 6.6 – Synchronized spreadsheet views

By placing the selected cell or range of cells into the model, both views share the selection information. Thus the upper view, which has the column headers, can display the selected column for both views and the lower view can display the selected row as well as the actual selected cell. Sharing the horizontal scroll position in the model allows both views to display aligned columns. For a spreadsheet application this makes a lot of sense because people tend to organize their information into columns.

A similar situation occurs in figure 6.7 with the Photoshop drawing and navigator views. Both of these views share the same drawing model and both draw in virtually the same fashion. In addition, the model for the drawing contains the current pan/zoom region of the drawing being viewed. However, each view interprets this information differently. The drawing (left) uses the pan/zoom information of the model to draw only that region of the model at the specified resolution. The navigator view (right) always draws the entire drawing and then shows the zoom information as the red rectangle on the drawing and in the zoom-factor widgets at the bottom. Dragging the rectangle in the navigator will scroll the drawing and scrolling the drawing will move the rectangle in the navigator. This works because the rectangle information is stored in the model that they both share and thus both are notified of changes to the viewed region.

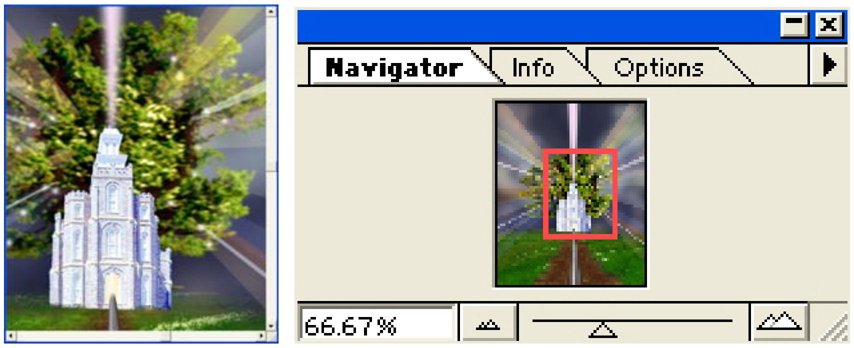
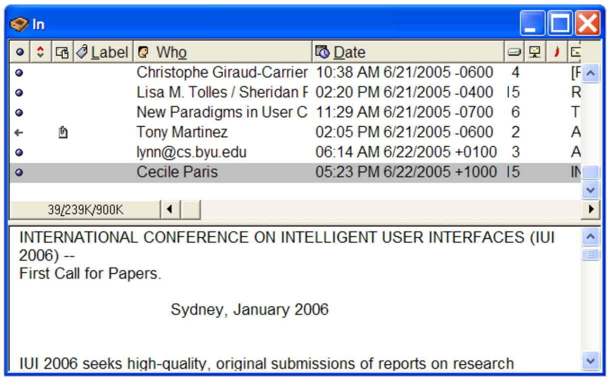


Figure 6.7 – Shared zoom and navigation

Figure 6.8 shows a common list/item pattern for synchronized selection. The email reader has a list of email messages from a mail folder. One of the messages is selected. When one is selected, the message view shows the content to the selected message. Placing the selected message information in the model allows both views to synchronize on what is to be displayed.


In

Label	Who	Date		
	Christophe Giraud-Carrier	10:38 AM 6/21/2005 -0600	4	[F
	Lisa M. Tolles / Sheridan F	02:20 PM 6/21/2005 -0400	I5	R
	New Paradigms in User C	11:29 AM 6/21/2005 -0700	6	T
	Tony Martinez	02:05 PM 6/21/2005 -0600	2	A
	lynn@cs.byu.edu	06:14 AM 6/22/2005 +0100	3	A
	Cecile Paris	05:23 PM 6/22/2005 +1000	I5	IN

39/239K/900K

INTERNATIONAL CONFERENCE ON INTELLIGENT USER INTERFACES (IUI 2006) --
First Call for Papers.

Sydney, January 2006

IUI 2006 seeks high-quality, original submissions of reports on research

Figure 6.8 – Collection/Item synchronization

Figure 6.9 shows a widget arrangement that is common in 3D applications. A single model object is shown in all three views. Each view shows a different top, front or side perspective on the object. Simultaneously showing all views aids the users in getting a sense of the 3D shape.

The user is pointing at a location in the front view using the mouse. This specifies X and Y coordinates, but not a Z value. However, the selection point is shared by all three views. The other two views show lines rather than points because that is the projection of the user's selection into each of the other views. Each view renders the selection in their own way, but they all share the same selection information.

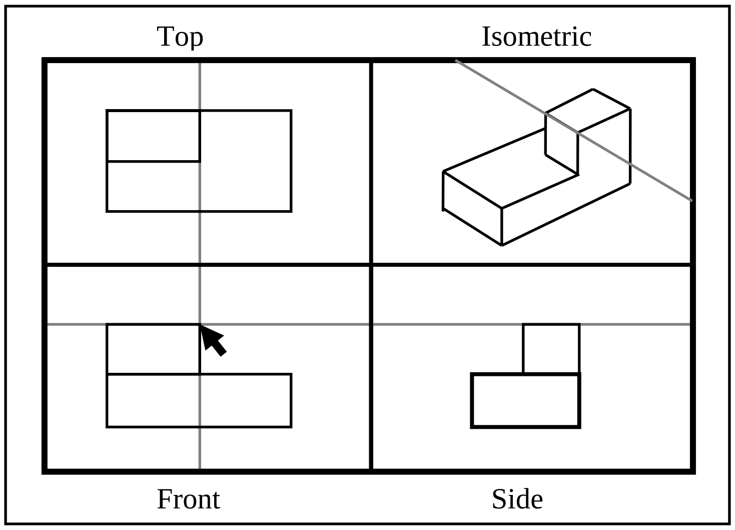


Figure 6.9 – Synchronized 3D position

In the upper right of figure 6.9 is an isometric view of the same model. We now have a design decision. Do we synchronize the selection information of all four views or do we allow the isometric view to handle selection in its own way? Suppose for some reason that we want the selection of the top and isometric views to be synchronized and the front and side views to be synchronized independently. We cannot do this by putting the selection information into the model because there are four views and two sets of selection information.

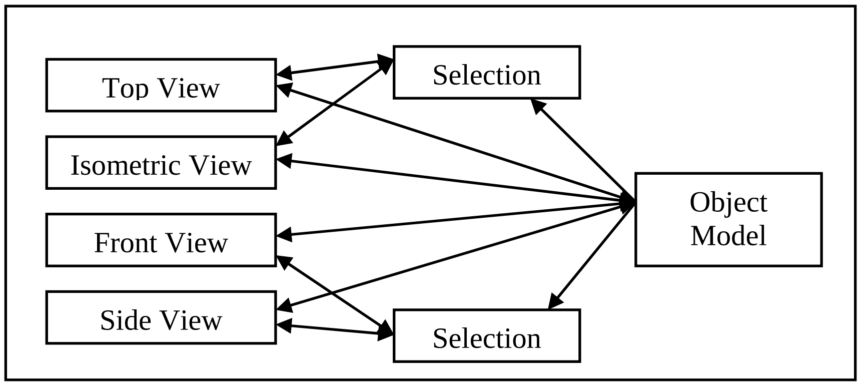


Figure 6.10 – Separate selection models

What we need is an architecture like that in figure 6.10. We have separated selection from the model into selection objects. We treat these selection objects like models in that they have public methods for the controller and they have listeners. Each view is now listening to both a model and a selection model for changes. Note that the selection objects also listen to the object model. Deletion or addition of objects to the model may change the selection and the selection objects must update themselves accordingly.

A pass-through variation on the selection/model architecture is shown in figure 6.11. In this case the views only listen to the selections. The selection objects are responsible for propagating model changes through to the views. This simplifies the interconnections and the definition of what is actually shared between views.

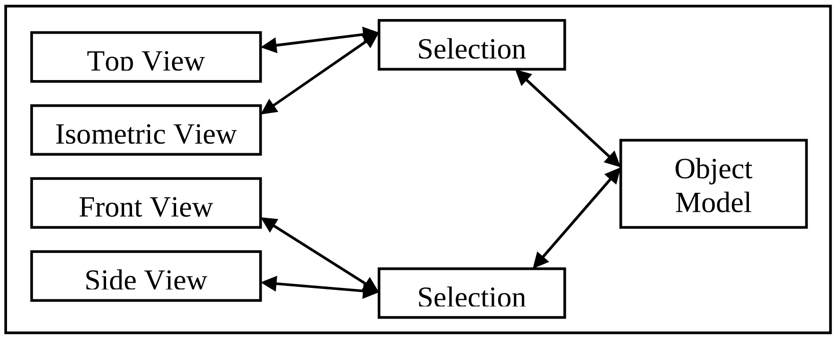


Figure 6.11 – Selection pass-through model access

In the pass-through architecture the selection listener interface should extend the model listener interface. This allows selection listeners to receive model changes in addition to the selection changes. The selection objects in general receive model changes and propagate them directly to their own listeners. Thus the model changes come straight through. In some cases the selection may want to modify itself in response to model changes. In those cases the selection is modified and both the selection change and model change are sent on. The selection object also mirrors and forwards the model change method calls from the controller.

Managing Model Persistence

A last issue is the problem of saving the model when changes have been made. One may open a second view on a document to make changes to a different part of the document and then close that view. The original view is still open. It would be very annoying for the model to ask the user to save the contents every time a view is closed. One way to handle this is to add a method to the model to report the number of listeners registered with the model. When a view is closed and this is the last listener, a dialog can be displayed to ask if the model should be saved. The problem with this is that there are many kinds of listeners. In figure 6.2 there are four views listening to the model. Closing that window will close all of them as a group.

One solution is to have only the container window registered as a listener on the model and have it forward all of the change messages to each of its views. In this case the approach of watching the listener count will work just fine. The architecture for this is shown in figure 6.12.

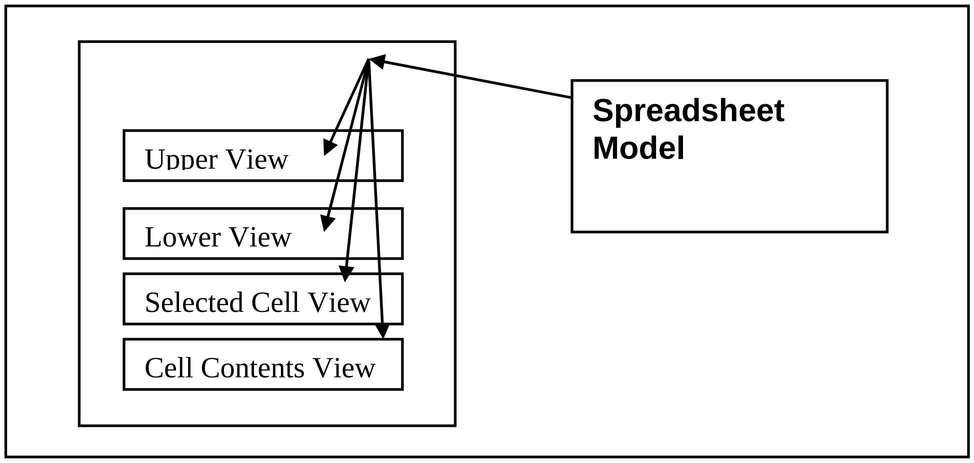


Figure 6.12 – Window dispatch of change messages

A different solution is to add the code in figure 6.13 to the model. When a window (which may have a group of views) is opened and references the model, it will call openWindow(). When that window closes it will call closeWindow() but will only actually close if the result is true. This allows the user to select CANCEL in response to the request to save the model.

```
public class MyModel
{
        private boolean unsavedChanges=false;

        . . .
                 many other fields and methods
                any methods that change the model will set unsavedChanges to true
                any methods that save the model will set unsavedChanges to false
        . . .

        private int windowCount=0;
        public void openWindow()
        {       windowCount ++; }
        public boolean closeWindow()
        {
                if (!unsavedChanges)
                        return true;
                int rslt = pop up model save dialog box with options (Save,Discard,Cancel);
                switch(rslt)
                {
                case SAVE:
                        save the model
                        return true;
                case DISCARD:      return true;
                case CANCEL:       return false;
                }
        }
}
```

Figure 6.13 – Model saving

Summary

There are a variety of ways in which views need to synchronize with each other. The listener mechanism on a model that is shared by those views is the primary mechanism for such synchronization. It allows each view to be implemented independently and yet stay consistent with each other. A key design decision is whether or not viewing and selection information is to be shared. For independent view/selection, the information goes in the view class. For shared view/selection, such information goes in the model. For more complex relationships the selection information can be separated from the model with its own listener/notification mechanism that is either independent of the model or passes the model information through to its own listeners.