Model Change Propagation

In the previous chapter we talked about software architecture for establishing how the presentation should be drawn, based on the model. In an interactive setting the model regularly changes. Sometimes it changes because of an interaction in the view, sometimes from another view and sometimes from a completely different program running on a different computer. In this chapter we are not concerned with where the changes came from, we are, as shown in figure 7-1, interested in how to translate a change to the model into a corresponding change to the view presentation.

This chapter has four parts. The first is to establish a basic listener mechanism by which view presentations can listen to models and be notified of any changes. The second is a discussion about the granularity of listening that we should use and techniques for implementation. The third is the mechanism taking presentation changes and notifying the windowing system that some portion of the display needs to be redrawn. Lastly we will take each of the mode to presentation mapping techniques discussed in chapter 6 and show how model changes are handled using that technique.

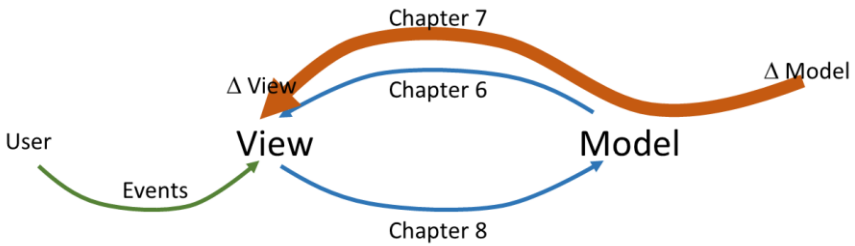


Figure 7-1

Basic Listener

At the foundation of virtually all of our change propagation techniques is the ability for one object to register with another object so that it will

be notified whenever changes occur. We will start with two Java Interface definitions to capture the ideas and then show how similar capabilities can be used in other languages. These two definitions are shown in figure 7-2.

```
interface ChangeNotifier
{   void addListener(ChangeListener listener);
    void removeListener(ChangeListener listener);
    void changeNotify(SO changeDescriptor);
}

interface ChangeListener
{   void changeNotify(SV changedObj,
           SO changeDescriptor);
}
```

7-2 – Change notification definitions

Every class that implements ChangeNotifier must have a list of listeners that it maintains. The addListener() and removeListener() methods maintain this list. Whenever ChangeNotifier.changeNotify() is called, it calls changeNotify() on each of the listeners in the list.

The changeDescriptor that is passed as a parameter can be null (we just are saying that the model has changed) or may contain a change descriptor like those shown in figures 2-6 or 2-7. The change descriptors can provide a more detailed representation of what changed and allow the presentation to be more precise about what gets updated. On many modern devices simply causing the entire view to be repainted is fast enough and we will use the null changeDescriptor. However, there are cases where this is excessive, so we will also discuss the more precise techniques.

As an example, consider our Employee and EmployeeView from chapter 6. The Employee class would implement the ChangeNotifier interface from figure 7-2. The EmployeeView class would implement ChangeListener. When the EmployeeView object is constructed and placed into the presentation, it will add itself as a listener to the Employee object using the addListener() method. Whenever the Employee is changed, its changeNotify() method should be called which

will call changeNotify() on all of its listeners (including our EmployeeView). When the EmployeeView is no longer needed as part of the user interface, it would call removeListener() on the Employee object to take itself off of the listener list. The Employee class would need to implement the addListener() and removeListener() methods that would manage a list of listeners.

A variation on the architecture in figure 7-2 is to define a different listener interface for each class of model object that offers change listening. The value of this is that the listener interface for a specific class can contain specific methods for various kinds of changes. These specific change methods are an alternative to a changeDescriptor. This is frequently useful in strongly typed languages such as Java or C++ which do not have a generic data model. An example of this would be to change the ChangeListener interface to EmployeeChangeListener with the method being employeeChangeNotify(Employee e). The advantage of this is that the listener can have different methods for different types of objects that it may be listening to.

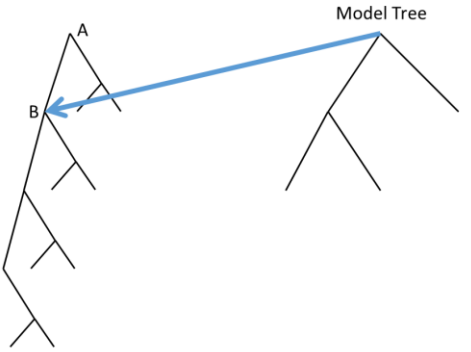
This simple listener architecture allows any number of presentations to register as listeners to a given model object and then to be notified. There are many cases where a given set of information is viewed in multiple ways. It is also the case that a persistent data store, such as a database or web service will want to be notified of a model change. They too can listen to the model and update their files based on the changes.

Any class that wants to allow listeners to register can implement the ChangeNotifier methods. Any presentation, database or other interested object can register themselves by calling addListener() on the object they would like to listen to.

Garbage collection issues with listeners

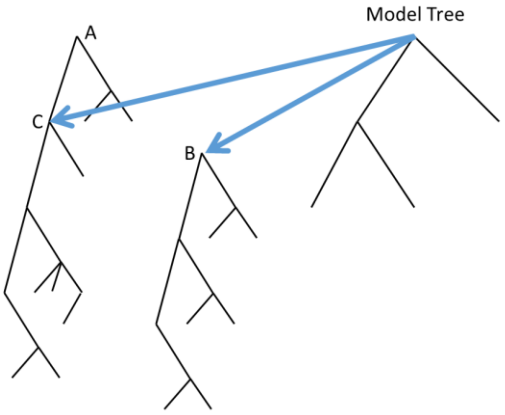
Most modern user interface languages support automatic garbage collection of data. This has been a great boon to UI development. There is a garbage collection problem that users of the listener architecture

should be aware of. Figure 7-3 shows a situation where object B in presentation tree A has registered itself as a listener. When the registration occurred, the model tree kept a pointer to object B so that it could notify object B of any changes.



7-3 – Listening to the model tree.

In figure 7-4 we see that for whatever reason object A has replaced object B with object C, thus changing the presentation. People familiar with garbage collection would assume that breaking the connection between A and B would cause B to be collected and reused. However, unless B has been removed as a listener, the listener pointer still exists and therefore B will not be garbage collected. This can lead to a lot of useless storage if the presentation is changed many times. The fix is to always remove unneeded listeners.



7-4 – Not removing a listener

Delegates in C#

The Java listener mechanism described above has a number of small problems. The first is that the code for adding and removing listeners as well as notifying all listeners must be written for every new model situation. Some use of superclasses can make the code reusable, but frequently that is not possible. Second, is that all notifications from all sources (and there may be several) all pass through the same changeNotify() method in the ChangeListener interface. Frequently a presentation or other object will listen to several sources for different purposes. Having them all go through the same method is an awkward design.

In C# and similar languages there is the concept of a *delegate*. The delegate types and delegate variables make all of this very simple. The code in figure 7-5 is an example:

```
delegate void ChangeListener(SO object,
    SV changeDescriptor);

public class MyModel
{
    public ChangeListener notifyChange;
    . . . .
    void someMethod()
    {   . . . .
        notifyChange(this, null);
    }
}

public class MyView
{   . . .
    public MyView(MyModel model)
    {   . . .
        model.notifyChange+=notifyThisView;
    }
    public notifyThisView(SO model, SV changeDescriptor)
    {   update the presentation }
}
```

7-5 – Use of delegates in C#

At the top of figure 7-5 we see a declaration of the delegate type ChangeListener. The MyModel class then declares the variable notifyChange to be a ChangeListener. The notifyChange variable now contains everything necessary to manage a list of listeners. This was automatically generated by the compiler rather than written by the programmer. Inside of someMethod(), the notifyChange delegate is called. This implicitly forwards the call to every registered listener. In our MyView class we want to register as a listener to the model. The += operator performs the same function as addListener(). There is also a -= operator to perform removeListener(). These were also automatically generated by the compiler. The MyView constructor has told the model to call notifyThisView() whenever changes occur. If MyView is listening to several different sources they each can be given a different method to be called.

There are two distinct differences between the delegate architecture and the listener architecture. The first is that with delegates, the compiler generates most of the code, which greatly simplifies

development. The second is that notifications are on the granularity of methods rather than whole objects with a single method. Instead of registering a listening object, we register a listening method. This simplifies hooking up listener mechanisms with more clarity in the code.

In the MyView class of figure 7-5 the notifyThisView() method was added to the notifyChange delegate. In actuality, what is added is {this,notifyView()}. Not only is the method added, but the MyView object being referenced at the time of adding the delegate. When notifyThisView() is later called, what really happens is this.notifyThisView(). The view object is implicitly the target object for the notification.

Function values in JavaScript, Python and others

With the advent of C it became quite common to pass pointers to functions and then call those functions as part of a notification architecture. The problem with the C mechanism is that type checking and other controls were absent. It was very easy to do the wrong thing and then have your program explode in a very obscure manner.

Languages like JavaScript, Python, Ruby and others all have the notion of a function as a data object that can be passed as a parameter and stored in data structures. This is exactly what we need. There are two advantages to the functions-as-data approach. The first is that when errors occur they are clearly reported and the second is that the scope of variables at function creation time is more powerfully defined. These languages are not statically type checked, like Java or C#, but they are not unchecked, like C. Figure 7-6 shows a simple example of how functions can be passed as data in JavaScript.

```
var notify = function(msg) { alert(msg); };

var tryThis = notify;

tryThis("hello");
```

7-6 – Function data objects

In figure 7-6 a simple function is created and assigned to the variable notify. It is a data value just like any other and can be reassigned to something else. In the last line tryThis() is invoked as a function and calls the function that was originally assigned to notify. This is not the place to discuss all of the nuances of function definition in these languages. The example in figure 7-6 illustrates what we need.

Figure 7-7 shows how we can use this to create a simple presentation notification system in JavaScript.

```
var model = { name:"Jose Shwartz", salary:250000 };
var presentation = { display objects };

presentation.updateView=function(model)
    { code that will update the view from the model };

model.listener=presentation.updateView;
```

7-7 – Simple view update using function objects

In figure 7-7, whenever there are any changes to the model the code that performed the change should call listener(), which is assigned the presentation's updateView() function. Thus changes in the model are propagated into the view.

There is a serious problem with the architecture of figure 7-7. It only allows one listener on the model. Figure 7-8 shows a more robust approach.

```
function ListenerList() { this.listeners = []; }
ListenerList.prototype.listen=function(listener)
   { add listener to the list of this.listeners };
ListenerList.prototype.remove=function(listener)
   { remove from this.listeners };
ListenerList.prototype.notify=function()
   {  for (var listener in this.listeners)
      {  listener.notify(); }
   };

function MyModel()
{  whatever is needed for my model
};
MyModel.prototype=new ListenerList();
   // Weird way JavaScript does superclasses

function MyPresentation(model)
{  this.model = model;
   this.notify=function()
      { code to update the presentation };
   model.listen(this);
   . . . .
}
```

7-9 – List of listeners

In JavaScript, “classes” are created by functions used with the new operator. To add methods to the “class” we assign functions to attributes of the function’s prototype object. There is no need to declare an interface because a listener can be any object that has a notify() method. In figure 7-9 we have also factored the listener code into a ListenerList class so that we do not need to reimplement it for every model class that we create. The use of prototypes for method inheritance is a little cumbersome in JavaScript, but it is usable.

Granularity of Listening

The granularity issue is illustrated by comparing figures 7-10 and 7-11. In figure 7-10 each individual text item registers itself as a listener on the corresponding data item in the Employee model. When any item in the model changes, the corresponding item in the presentation tree is notified and updated. This approach efficiently changes only the presentation fragments that need to be changed.

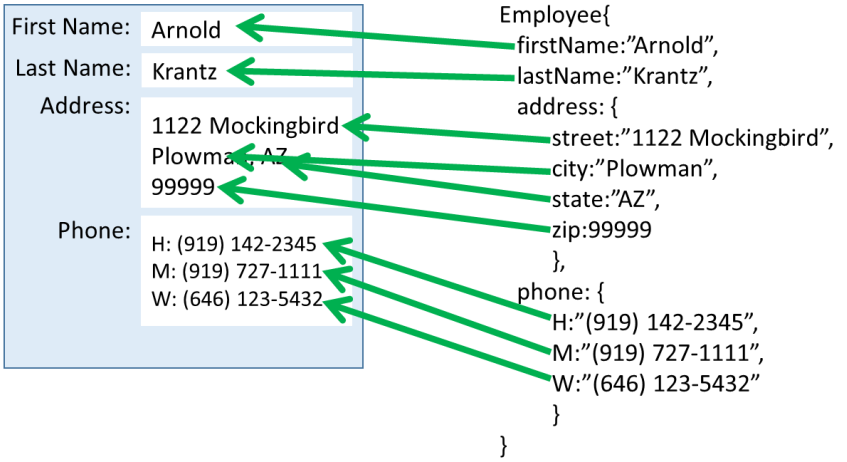
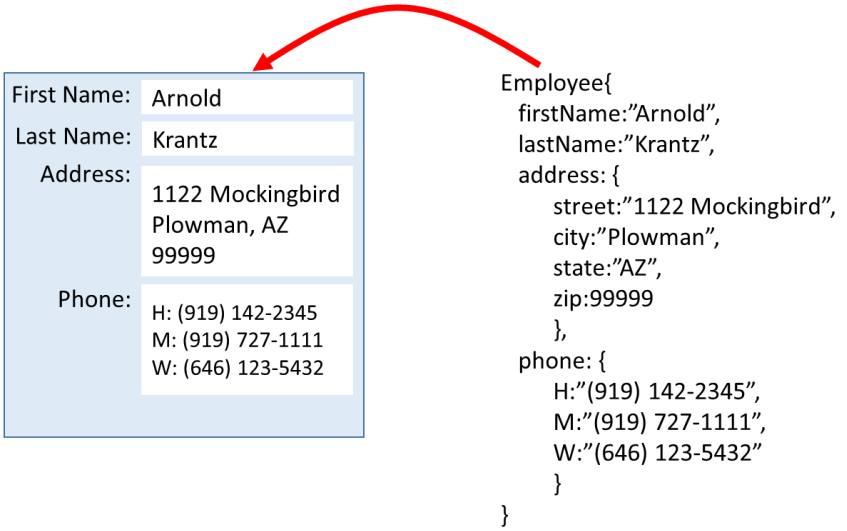


Figure 7-10 – Listener per data item



7-11 – Listener per major object

In figure 7-11 the whole view listens only to changes to the Employee model. If anything changes in the model, the entire presentation must be rebuilt. The problem lies in the building up and tearing down of all the listener relationships in figure 7-10. Not only is space required for all

of the listener registrations, but every change to the presentation entails changes the listener relationships. The code to setup and tear down the listening starts to become quite cumbersome. Provided the presentation is not too extensive, the extra effort of redrawing the entire presentation in response to any change is worth the simplification of the code.

The remaining problem is how the Employee object learns of changes down inside of the address object. This particular example is only two levels deep, but in many cases the tree may be deeper. The simplest approach is to have the programmer call Employee.notify() whenever changes have been made anywhere in the Employee model tree. While this may be simple, it imposes the burden of remembering to call notify() on the right objects whenever necessary. This is a source of semi-obscure bugs in the code.

The next approach is to have getter() and setter() methods on every value in the model tree. If one only sets firstName by calling model.setFirstName() then we can put the call to notify() into setFirstName(). In this way every time the firstName is changed, the notify() method is called and the presentation gets updated. For this to work reliably, we make the actual data values private and the getter() and setter() methods public. Thus there is no way to modify the data without notify() being called. This works for strongly typed languages such as Java or C# but not for languages such as JavaScript or Python. Using function closures it is possible to create private information in JavaScript and Python, but it is rather cumbersome and not at all intuitive. The technique will work in these languages, but it is not as failsafe. In addition to setting of attributes, there is also insertion and deletion of items in a list or array. All methods that change the data must invoke notify when they are done.

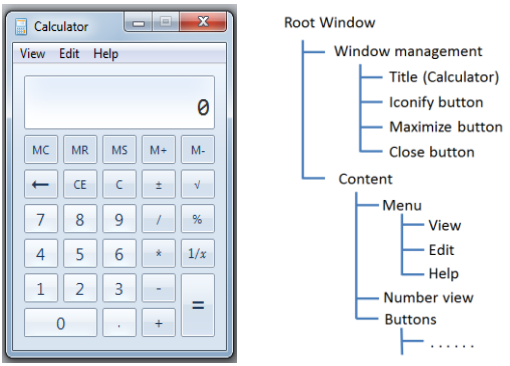
This technique does not solve all of our problems in figure 7-11. For example if model.address.setStreet() is called, only the address object is notified and nobody is listening to it. In this case the presentation will not get updated. The answer lies in the fact that we are using trees

throughout rather than more complex data structures. Every node in a tree has at most one parent. We can improve the notify() method so that it first notifies all of its listeners and it then calls notify() on its parent object. In figure 7-11, address has no listeners so model.notify() is called immediately which will then call presentation.notify() so that the presentation can get updated. By always propagating notify() up the model tree, a presentation can attach a listener to whatever objects it wants at whatever level of granularity in the model tree.

Updating the display

With the exception of HTML and some other presentation tools, notifying the presentation that a change has occurred is insufficient. The windowing system must be notified of the change so that it can call paint() again. When a change notification arrives, we do not want to immediately call paint() for a variety of reasons. The first is that the presentation may not currently be visible. The window may be iconified or hidden behind other windows. Only the windowing system knows for sure. The second is that a single interactive event may generate multiple changes. For example changing the zip code in our address object may automatically change the city and state. What looks like one change becomes three and each generates a call to notify().

When first discussing the windowing system in chapter 3 we had a calculator like that in figure 7-12 that has a corresponding window tree.



7-12 Calculator with window tree

Each window in the tree controls a rectangular region of the screen into which information can be drawn. In addition, each window can receive input events and dispatch them to application code (as we will see later). A window also has access to application code to redraw the rectangular region. In this text we have been using the paint() method as that mechanism, as discussed in chapters 3 and 6.

This combination of window, paint() method and event handling is called many things in many systems. In Java it is a Component or JComponent. In most Microsoft products it is called a Control. In Android they are called Views. In historical systems they were called Widgets, which is the term this text will use.

For the purposes of this discussion every Widget has a damage() method. You call damage() when you want to tell the system that the rectangular region corresponding to this Widget no longer contains an up to date presentation of the contents. This method also is called many things including update(), repaint(), dirty() and others. This is a key method for an interactive system so find it early when learning a new system. The purpose is to tell the windowing system that this window needs to be redrawn.

In most systems there is a version of damage() that accepts a rectangle so that an application can be more precise about which part of the window needs to be redrawn. This can improve the efficiency of the application by not redrawing portions of the screen that have not changed. The majority of the time performing damage() on the whole window is fast enough for the user not to see the difference. In some cases of large, complex widgets, painting the whole window may be too slow for interactive responsiveness. In those cases you should try damaging only the portions of the window that have actually changed. However, avoid optimizing too soon. If damage to the whole window is fast enough, do not complicate your code by adding trying to figure out a smaller damage rectangle.

When the windowing system receives an input event, some method on the Widget is called (as will be discussed later). This generally leads to some change to the model, which will cause various views to be notified of the changes. When notified each presentation view should call damage() to warn the windowing systems of the needed changes. If an event makes many model changes, the notifications will fire many times and damage will be called many times. However, the windowing system simply collects all of the calls until the input event has been completely processed. It then discards any damaged regions that can't be seen for various reasons. It may trim some down because they are only partially visible. It may combined several of them because they overlap substantially. It will then call paint() on the regions that remain. This is much more efficient than calling paint() redundantly on every notification. The windowing system will do all of this for you automatically.

Propagating change

In review, we now have a mechanism whereby a presentation can register to listen for any changes to a model. We also have a mechanism for change notification to work its way up the model tree so that presentations can be notified of model changes. We also have a mechanism whereby presentations can call damage() or notifications can propagate up the presentation tree so that damage() is eventually called. Once damage() has been called the windowing system takes over and paint() will be called at the appropriate time to get the screen updated. The missing piece in all of this is to take all of our techniques from chapter 6 for converting model information into presentation information and show how they use model change notifications to generate appropriate presentation changes.

Simple content painting

Our first technique was to just implement a paint() method. In this technique, which is the basic technique for virtually all interactive systems, a notify() call on the presentation translates directly into a

damage() call on the widget. Eventually the paint() method is called and the item is repainted.

Presentation tree rebuilding

In chapter 6, we represented the presentation as a presentation tree. Whenever the presentation tree changes it is discarded and a buildPresentation() method is called (figure 6-9). The buildPresentation() method will retrieve information from the model and use it to build a new presentation tree. In this architecture, model change notifications are only received at the widget level, similar to figure 7-11. When a change notification is received, damage() is called and the presentation tree is set to null. As shown in figure 6-9, when paint() is called again, the null value will cause buildPresentation() to be called so the presentation tree can be rebuilt and then painted. This can be optimized slightly so that damage() is only called when the presentation tree is not null. It is an easy check that will save the windowing system some work.

Presentation tree change

Let us assume that in figure 7-13 the entire employee presentation is a single widget. Let us also assume that the user has changed the zip code in the employee's address. Let us also assume that we are using a presentation tree as part of our presentation system and the text object containing the zip code has been changed.

7-13 – A presentation tree

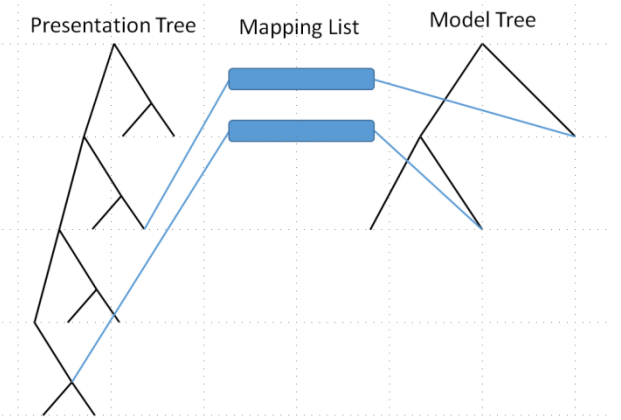
In this architecture, the paint() method is not called by the application code. Instead each presentation has a modelUpdate() method, as described in chapter 6. This method retrieves information from the model and makes changes to the presentation tree. In our example the text object containing the "99999" is changes, but it is not a widget and does not have a damage() method.

Our presentations are trees and each node has one parent. We can give each drawable object a notify() method. When any drawable object is changed, it calls notify() whose default implementation is to call parent.notify(). For drawable objects that correspond to widgets, their notify() method will call damage() and will not propagate the notification further up the tree. Thus any change to the presentation tree will produce a damage() call at the appropriate level in the tree. The windowing system can then take over and eventually call paint() to get the pixels restored to their proper presentation. Because modelUpdate() may make many changes to the presentation tree, damage() may be called may times. As we discussed previously, these multiple calls to damage() are combined by the windowing system so that paint() is only called once.

Another variation on this techniques is when a drawable object is changed, it computes its former rectangular bounds and its new rectangular bounds and sends these in the notify message to its parent. When the widget receives them it calls damage() using these two rectangles. This damages only the pixels that have changed. By incorporating the bounds checks into the drawable objects themselves, the code is implemented once into the presentation system and requires no programmer effort after that. Thus a more efficient paint() is provided with no additional programmer effort.

Tree mapping

The tree mapping techniques is shown in figure 7-14. A list of mapping objects provides the connection between the model and the presentation tree. Any of the previous presentation tree techniques can be used here. Setting the presentation tree to null can cause the presentation to be rebuilt. A more efficient technique is to reevaluate each of the model mapping objects so that they make their changes to the presentation tree. Thus the presentation update because a variation on the presentation tree change technique described previously.



7-14 – Mapping objects

Another possible variation is to have model changes enhanced beyond simple propagation of notify() up the model tree. In chapter 2 we discussed the representation of change on trees. Figures 2-6 and 2-7

showed examples of how changes can be represented. A change descriptor consists of the change to be made and a path describing where in the tree the change should occur. We can modify our notify() method to receive a change descriptor. Before passing the change descriptor on to its parent it adds its own part of the path onto the front of the change descriptor's path. Thus when the notification reaches a level where there is a listener, the change descriptor can be sent with the listener's notification. Each mapping object can then compare its own path against the change descriptor's path to see if it applies. This eliminates any changes to the model that do not change the presentation. We will use change descriptors later for other purposes besides this. Having the model mapping object explicitly describe their change locations as paths makes this technique possible.

Summary

We started with basic mechanisms for how one object can "listen" for changes on another object. Listener architectures are based on what language features are available. We looked at the basic Java mechanism that can be used in strongly typed languages. There is the delegate mechanism of C# that has the compiler generate most of the mechanism for you and then function objects like those found in JavaScript and Python.

We next talked about the granularity of listening. In particular, we discussed how one might want to listen to a single higher level part of the model tree rather than in detail on every value.

We then talked about mechanisms for how change to the presentation is translated into updating the screen. The damage()/paint() mechanism was described. Lastly we worked our way through each of the model to view mapping techniques from chapter 6 and showed how each can be used to update the display.

Exercises