Model To View

This chapter and the next two deal with the cycle shown in figure 6-1. Here in chapter 6 we will discuss how the contents of the model are transformed into a view presentation. This presentation is basic to how graphical user interfaces function. In chapter 7 we will look at how changes to the model (Δ model) are carried through to change the view (Δ view). This propagation of changes is key to keeping all parts of the user interface in synch with each other. The model can be changed in a variety of ways and it is essential that view/model consistency is reliably and efficiently maintained. In chapter 8 we will explore how changes to the view (primarily through user interaction are converted into changes to the model. This completes the interactive cycle. The user perceives the view and expresses changes in terms of what they see. We need to interpret those view-centric changes into model changes. This interpretation must be consistent with and integrated with the model-view mappings discussed in this chapter. In future chapters we will look at how the user expresses their changes in terms of the view.

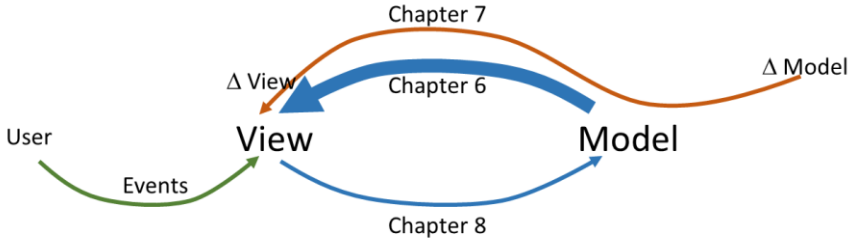
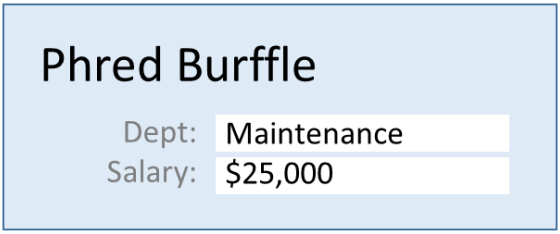


Figure 6-1

Example problems

The following are three example problems of models that we want to present to the user. Each provides examples of particular problems that we need to capture when mapping model information into visual presentations.

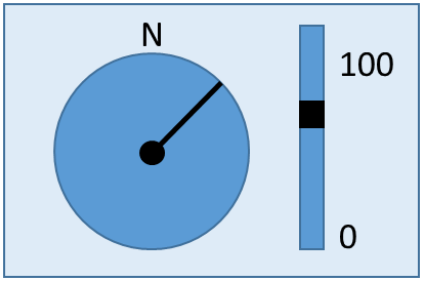


6-2 – An employee form

The first example is a simple contact form, like that shown in figure 6-2. The model might be:

```
Employee{ name: "Phred Burffle",
          department: "Maintenance",
          salary: 25000
    }
```

This example involves background drawing which is static such as the blue rectangle and the labels on the fields. It also involves very simple mappings from the model to the view. There is the slight complication of converting integers into currency upon display.



6-3 – Direction/speed control

The second example is a little more complicated. In figure 6-3 we see a control consisting of a direction control and a speed slider. The model might be:

```
Control{ direction: 45, speed: 75 }
```

In this case there is a more complex relationship between the values in the model and the geometry of the presentation. This is not a hard problem to solve, but too many user interface builders will quickly convert this problem back into a form, with its simpler implementation rather than deal with the potentially superior presentation in 6-3. Note that figure 6-3 might not actually be a more usable solution than a form, but that should be a usability decision based on user performance rather than a reflection of low implementation skills.

Our last example consists of lists of similar presentations. In figure 6-4 we see a list of employee records, each for a different employee. The handling of lists offers the opportunity to define one mode-to-view mapping and then repeated reuse that mapping with different models.

Phred Burffle	$25,000
Kelli Kanfield	$27,000
Crissy Crow	$52,000
Jose Martinez	$53,000

6-4 – List of forms

Paint method

In chapter 2 we introduced the class SOReflect that will create a Spark object that uses reflection to generate a Spark Object that has the attributes of a class. In chapter 3 we introduced the Drawable interface that requires a class to have a paint() method that takes a Graphics object as a parameter. We can combine these together to create view classes that can be placed in the view tree and will render the model information.

Simple content painting

For example, our form in figure 6-2 might be implemented as follows:

```
public class EmployeeView extends SOReflect
       implements Drawable
{
    public Employee model;
    public void paint(Graphics g)
    {   g.setColor(Color.lightBlue);
        g.fillRect(0,0,400,160);
        g.setColor(Color.darkBlue);
        g.drawRect(0,0,400,160);
        g.setFontSize(28);
        g.drawString(40,50,model.name);
        g.setColor(Color.gray);
        g.setTextAlign(RIGHT);
        g.setFontSize(18);
        g.drawString(140,100,"Dept:");
        g.drawString(140,150,"Salary:");
        g.setColor(Color.white);
        g.fillRect(160,80,230,30);
        g.fillRect(160,120,230,30);
        g.setColor(Color.black);
        g.setTextAlign(LEFT);
        g.drawString(180,105,model.department);
        g.drawString(180,140,toCurrency(model.salary));
    }
}
```

6-5 – Paint method

There are a few things worth noting in this example. First is that most of the information in the paint() method consists of styling and laying out the background information for the form. Very little of the effort actually involves retrieving the information from the model. Secondly we used a function toCurrency() that transforms an integer into a string that shows the value in dollars. We will visit such transformations of model data later. Lastly the class EmployeeView can now be embedded into a presentation tree as:

```
. . . . .
EmployeeView{
   model: Employee{
             name: "Phred Burffle",
             department: "Maintenance",
             salary: 25000
          }
   }
. . . . .
```

6-6 – Employee view object

Alternatively we could have given EmployeeView a key reference to the Employee record so that the employee was retrieved from a data store rather than embedded in the presentation tree, such as:

```
EmployeeView{ model: "EMP090080" }
```

Computed geometry painting

The control from figure 6-3 has many similar elements to those in EmployeeView. The painting of the background and labels are pretty much the same. The difference is in the way that the model relates to the geometry of the direction pointer and the position of the slider. Our implementation might be as follows:

```
public class ControlView extends SOReflect
       implements Drawable
{   public Control model;
    public void paint(Graphics g)
    {
        . . . . .
        paintSlider(g);
        paintPointer(g);
        . . . . .
    }
    private void paintSlider(Graphics g)
    {   g.setColor(Color.black);
        double sliderTop = 160-(model.speed/100 * 120);
        g.fillRect(200,sliderTop,20,20);
    }
    private void paintPointer(Graphics g)
    {   g.setColor(Color.black);
        double radians = model.direction/360.0*2*pi();
        double pointX = 80*sine(radians) + 100;
        double pointY = 100-(80*cosine(radians));
        g.drawLine(100,100,pointX,pointY);
    }
}
```

6-7 - ControlView

In this case we used private methods to separate the key geometry problems out of the rest of the paint method. For the slider we needed to convert the speed values of 0-100 into a Y location that took into account the length and position of the slider track as well as the size of the slider itself. For the pointer we needed to convert a direction angle into an (x,y) position for the end of the pointer. These conversions between model information and geometry are quite common and we will develop better mechanisms for dealing with them later in this chapter.

List painting

When just using a paint() method, painting lists of things is fairly straightforward. In this example we will keep track of the top of each form. Later we will take care of this using a layout manager. Our implementation might look like this:

```
public class EmployeeListView extends SOReflect
       implements Drawable
{   public SA model; // Spark array of employees
    public void paint(Graphics g)
    {   int top = 0;
        for (int i=0;i<model.size;i++)
        {   paint(g, top, i);
            top+=50;
        }
    }
    private void paint(Graphics g, int top, int i)
    {   g.setColor(Color.lightBlue);
        g.fillRect(0,top,300,40);
        g.setColor(Color.darkBlue);
        g.drawRect(0,top,300,40);
        g.setFontSize(18);
        g.drawString(15,top+30,model[i].name);
        g.drawString(200,top+30,
            toCurrency(model[i].salary));
    }
}
```

6-8 - EmployeeListView

The painting of lists is obvious when one factors the painting of an individual item into a function. We will want similar structures in our other architectures.

Presentation building method

In the paint() method approach for generating a presentation, we redraw everything directly from the model. This has the distinct advantage that what is drawn always reflects the current state of the model. It also requires minimal structure in the architecture of our user interface system.

The problem with the paint() method approach is that all of the information about what is being drawn is either embedded in code or is discarded right after the drawing calls are executed. As we will see in subsequent chapters, we need this information for other parts of the architecture. An alternative approach is to have a method that builds a presentation tree from the model and then the actual painting is done from the presentation tree. The presentation tree will contain all of the

drawing geometry that can be used by other parts of our architecture. Historically presentation trees took up too much memory. In modern devices, available RAM far exceeds the needs for display.

Appendix A.3 contains the definitions of the Spark presentation tree classes. Similar objects are found in other systems. In Microsoft WPF the presentation tree is represented in XAML [xx]. In HTML5 the presentation tree might be the Domain Object Model (DOM) if the presentation is hypertext or Scaler Vector Graphics (SVG) for a more graphical flavor of the kind we are discussing here. A presentation tree is very common in 3D graphics because much of the interaction consists of changing the viewpoint rather than changing the model. It is more efficient to redraw from the presentation tree with a different viewpoint than to rebuild the tree from the model.

For this discussion we will use the following Spark display classes:

```
Line(x1,y1,x2,y2,lineWidth,drawColor)
Rect(top,left,width,height,lineWidth, drawColor, fillColor)
Ellipse(top,left,width,height,lineWidth, drawColor, fillColor)
Text(x,y,text,textColor)

Group(contents)
```

The Group object contains an array of other display objects (contents). Group is helpful in collecting together objects that relate to a particular model concept.

As a foundation for our presentation tree building approach we will create a Drawable called ModelGroup that will be the superclass of all of our specific view classes. ModelGroup is defined as follows:

```
public class ModelGroup extends SOReflect
       implements Drawable
{   public SA presentation;
    public void paint(Graphics g)
    {   if (presentation==null)
           buildPresentation();
        for ( int i=0;i<presentation.size();i++)
        {   SV e = presentation.get(i);
            if(e.isSO())
            {   SO o = e.getSO();
                if (o instanceof Drawable)
                {   ((Drawable)o).paint(g); }
            }
        }
    }
    abstract void buildPresentation();
}
```

6-9 - ModelGroup

The ModelGroup class check to see if it has a presentation. If it does not, then it uses the method buildPresentation() to create one. The buildPresentation() method is only called once to generate the presentation tree and then everything is drawn from the presentation tree. If the model ever changes, we set presentation to null and the presentation tree will be rebuilt from the model.

The following code shows how we create a view for our simple form:

```
public class EmployeeView extends ModelGroup
{   public Employee model;
    public void buildPresentation()
    {
        SA p = new SA();
        p.add(new Rect(0,0,400,160,1,
            darkBlue,lightBlue) );
        p.add(new Text(40,50,model.name, black));
        p.add(new Text(140,100,"Dept:",gray,RIGHT));
        p.add(new Text(150,150,"Salary:",gray,RIGHT));
        p.add(new Rect(160,80,230,30,white));
        p.add(new Rect(160,120,230,30,white));
        p.add(new Text(180,105,model.department));
        p.add(new Text(180,140,
            toCurrency(model.salary)));
        presentation = p;
    }
}
```

6-10 – EmployeeView using presentation tree

The above code will build a Drawable object that contains a list of all the graphics to be drawn. The code in buildPresentation() looks a lot like the paint() method that we built using the previous architecture. The difference is that the information is captured in objects that we can reuse.

In figure 6-11 we see what EmployeeView will look like after buildPresentation() has been called. The drawing information about an employee has now been exposed in a form that other software can work with. As long as the drawing information is buried in the code of a paint() method, there is little that other software can do. Now that the drawing is exposed as data objects tools such as selection, undo, visual design tools, animation and web collaboration are all readily possible. We will see tools like this in subsequent chapters. They are empowered by exposing the drawing as data objects.

```
EmployeeView {
    model: Employee{ name:”Pherd”,
           department:”Maintenance”,
           salary: 25000
       },
    presentation: [
       Text{x:40,y:50,text:”Pherd”, color:“black”},
       Text{x:140,y:100,text:”Dept:”,color:”gray”,
          align:”RIGHT”},
       Text(x:150,y:150,text:”Salary:”,color:”gray”,
          align:”RIGHT”},
       Rect{left:160,top:80,width:230,height:30,
          fill:”white”},
       Rect(left:160,top:120,width:230,height:30,
          fill:”white”},
       Text{x:180,y:105,text:”Maintenance”},
       Text{x:180,y:140,text:”$25,000”}
    ]
}
```

6-11 – EmployeeView with presentation

Presentation update

In all of our examples above the code to create the presentation contained lots of information that does not change when the model changes. The presentation also contains a lot of geometry information that is not well expressed in code. It would require very careful comparison of the code in the paint() methods against the pictures of the presentations that they are supposed to create. What is helpful is if we can draw a sample presentation using some drawing tool and then make changes to that presentation based on the model information.

Suppose we want to design one of the employee forms from figure 6-4. We can imagine some drawing tool that would let us draw an example of that form. The result might be the following presentation tree:

```
[
       Rect{ left:0, top:0, width:300, height:40,
          lineWidth:1,drawColor:darkBlue,
          fillColor:lightBlue }
       Text{ x:15, y:30, text:”Full Name” }
       Text{ x:200, y:30, text:“Salary” }
]
```

6-12 – Employee presentation objects

Not all of the properties are present, but the above is sufficient to see what is going on. To use this to present a model we can create a class ModelUpdateGroup that is the super class for views that might use this technique.

```
public class ModelUpdateGroup extends SOReflect
       implements Drawable
{   public SA presentation;
    public void paint(Graphics g)
    {   . . . code to paint presentation . . }
    abstract void modelUpdate();
}
```

6-13 - ModelUpdateGroup

We can use this super class to create a different version of EmployeeView. If we assume that the presentation field has been filled with the data in figure 6-12, then the implementation is as follows:

```
public class EmployeeView extends ModelUpdateGroup
{   public Employee model;
    public void modelUpdate()
    {
        presentation[1].text = model.name;
        presentation[2].text = toCurrency(model.salary);
    }
}
```

6-14 – EmployeeView using presentation update technique

The next time this object is painted its presentation tree will use the text that was inserted from the model. This is the basic mechanism used by JavaScript in updating the DOM that was derived from the HTML. Changing the DOM, or in our example the presentation tree, will cause the tree to be redrawn and the model information will appear on the screen. In chapter 7 we will look at the mechanisms that actually force the presentation tree to be repainted. For now we will assume that it magically happens whenever the tree is changed.

An advantage of this architecture is that the presentation can have all the decoration, images, icons, curvy things and texture that a designer might want. The presentation can be drawn in some external tool that designers will like instead of embedded in code which designers hate.

The presentation can be as complex as desired, but the mapping from model to presentation code is quite simple.

One of the problems with the code above is that it depends upon knowing that the Text{} objects to be changed are at indices 1 and 2. In a complex design this might be hard to determine and if the design changes at all these indices will probably change. That makes our modelUpdate() method very fragile. Because our Spark objects can have any attributes that we want, we can add an "id" attribute to each of the Text{} objects and us this to reference the objects. So our new presentation is:

```
[
       Rect{ left:0, top:0, width:300, height:40,
          lineWidth:1,drawColor:darkBlue,
          fillColor:lightBlue }
       Text{ x:15, y:30, text:"Full Name" id:"name"}
       Text{ x:200, y:30, text:"Salary" id:"salary"}
]
```

6-15 – Employee presentation with embedded IDs

We will also change ModelUpdateGroup and add a method called findID(*id*) that accepts a string and the searches the presentation tree for an object that has that id and returns that object. Now we can change our implementation of EmployeeView to a much more robust version:

```
public class EmployeeView extends ModelUpdateGroup
{  public Employee model;
   public void modelUpdate()
   {  SO o = findID("name");
      o.text = model.name;
      o = findID("salary");
      o.text = toCurrency(model.salary);
   }
}
```

6-16 – EmployeeView using presentation IDs

In this implementation the presentation can change as much as it wants and the code will still find the right objects and set the right attributes. Also a tool that draws the presentation design can be modified to allow

the designer to put id names on any objects that they want so that the information will be available for the modelUpdate() method to use.

This technique is widely used in JavaScript and HTML. Web browsers generally provide a global variable called “document” that contains the DOM or presentation tree. A particular object in the presentation can be found using the document.getElementByID() method.

```
. . . . .
<h1>Employee record</h1>
<p>Name: <span id="name"> </span>
<p>Salary: <span id="salary"> </span>
. . . .

var n = document.getElementById("name");
n.innerHTML=document.createTextNode(model.name);
var s = document.getElementById("salary");
s.innerHTML=
    document.createTextNode(toCurrency(model.salary));
```

6-17 – HTML and Javascript use of presentation update.

Figure 6-17 shows a snippet of HTML with tags used to denote where the model data should go. The JavaScript below shows how one would find the appropriate tag and then modify its text contents to match the model. The web browser parses the HTML and produces the DOM. It is the same as our Spark presentations except that the elements are designed for text layout rather than graphical drawing.

Figure 6-18 shows how the JavaScript would change when using JQuery. JQuery provides a richer more concise mechanism for finding the correct parts of the presentation to change, but the idea is fundamentally the same. When comparing figure 6-16 and 6-17 we see the same setting of the HTML contents but the amount of text to accomplish the goal is much less with JQuery.

```
. . . . .
<h1>Employee record</h1>
<p>Name: <span id="name"> </span>
<p>Salary: <span id="salary"> </span>
. . . .

$('#name').text(model.name);
$('#salary').text(toCurrency(model.salary));
```

6-18 – JQuery use of presentation update.

Text macros

Some interactive applications are "presentation-mostly." This is the traditional view of web interaction. The presentation is static with limited interaction. The goal is to present information and the interaction might consist mostly of clicking on links.

If the presentation is described in a text file then we have an opportunity to embed the model-to-view mapping in the text file and then preprocess the text file before it is parsed into a presentation tree. Figure 6-19 shows an example of how this might work in an HTML file.

```
. . . . .
<h1>Employee record</h1>
<p>Name: <span>{{model.name}}</span>
<p>Salary: <span>{{toCurrency(model.salary)}}</span>
. . . . .
```

6-19 – Macro processing of HTML

Figure 6-19 looks very much like figure 6-18 except that the model mapping has been moved into the HTML itself. In this example we have used {{ to mark the beginning of macro processing and }} to mark the end. The algorithm is simple. We process the text until we find {{. We then take everything between the double curly braces and compute whatever should go there. The results of the computation replaces the section enclosed in double curly braces. The name field illustrates a simple substitution of some information from the model. The salary field shows how we might execute some language fragment and take the result of that computation as the substitution text.

Such a text-based system does not need to know the kind of presentation being generated. It is simply processing text and substituting computation results. For example, figure 6-20 is the same as figure 6-19 except that the result is a SON specification for a drawable group rather than HTML. The algorithm is the same, the difference is how the resulting text is used.

```
Group{
    contents: [
        Rect{ left:0, top:0, width:300, height:40,
            lineWidth:1,drawColor:darkBlue,
            fillColor:lightBlue }
        Text{ x:15, y:30, text:"{{model.name}}"}
        Text{ x:200, y:30,
            text:"{{toCurrency(model.salary)}}" }
    ]
}
```

6-20 – Macro processing of SON

This is the primary technique used in PHP. PHP uses a unique syntax for identifying where it starts computing new text. We can recode figure 6-19 into what is shown in figure 6-21.

```
. . . . .
<h1>Employee record</h1>
<p>Name: <span><?=$model.name?></span>
<p>Salary: <span><?=toCurrency($model.salary)?></span>
. . . . .
```

6-21 – PHP processing of HTML

This particular form of PHP is subject to some debate with many people contending that it is bad form. This is not the place for PHP style discussions. However, figure 6-22 shows a slightly more verbose, but politically correct form of the same thing.

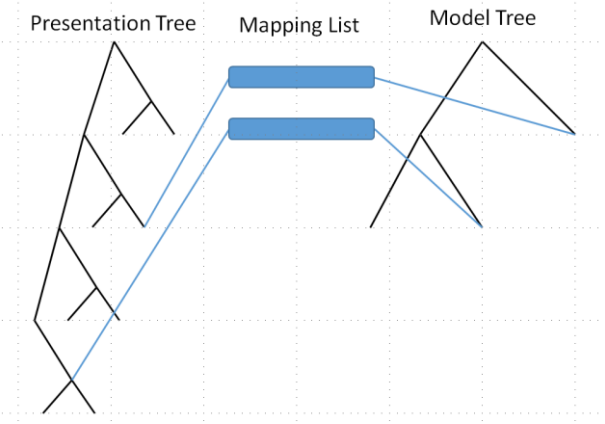
```
. . . . .
<h1>Employee record</h1>
<p>Name: <span><?php echo $model.name?></span>
<p>Salary: <span>
    <?php echo toCurrency($model.salary)?></span>
. . . . .
```

6-22 – PHP processing of HTML

PHP assumes that it is happily copying text from the input to the output until it encounters “<?php”. It then takes the section between there and “?>” and treats it as a small program written in the PHP language. The “echo” command will take the result of the expression to its right and will write it to the output stream where all the other text is going. The advantage of this is that instead of simple text substitution one can write arbitrarily large sections of code that generate large sections of text output. This significantly increases the sophistication of how models can be transformed into presentations. PHP can be thought of as a text processor on steroids.

There is real power in the text-based approach. The downfall will appear in later chapters when we want to interactively work back from the presentation to change the model. It is impossible to work back through arbitrary code.

Tree mapping



6-23 – Mapping objects

Figure 6-23 shows the structure of a technique we will call tree mapping. There are three main parts to this architecture. First and second are the model trees and presentation trees that we have seen before. Third is a list of mapping objects. Mapping objects describe the relationship between the model and the presentation trees. Just as we

pulled the code of a paint() method out into a set of objects in a presentation tree, we are going to pull the code of the modelUpdate() method in figure 6-16 out into objects as well.

A mapping object consists of four parts:

1) a path reference to the presentation object that is to be changed (see chapter 2 for path references,)
2) a path reference to the model object that is the source for the information that is to appear in the presentation,
3) a function that maps the model source into the value needed in the presentation, and
4) parameter values that map the source value to the presentation value.

A path can be represented as an array of selectors (strings for objects, integers for arrays) or can be represented as a string using dots between the selectors such as “context.1.text” for the model name in figure 6-13. We can slightly extend our path descriptions to include string selectors that begin with # to indicate a search for a particular id such as “#name.text” in figures 6-14 and 6-15. This will make our paths more robust as we have already discussed.

We have already shown in figures 6-20 and 6-21 an arbitrary fragment of code that is dynamically compiled can perform such a mapping. In many cases we do not have the compiler readily at hand to perform such operations. In an object oriented language we can overcome this problem by means of a set of classes derived from Map, as shown in figure 6-24.

```
public class Map extends SOReflect
{   SV presentPath;
    SV modelPath;
    protected SV getPath(SV obj, SV path)
    { . . .retrieves a value from obj using path . . }
    protected void setPath(SV obj, SV path, SV val)
    { . . .sets obj.path to val . . . }
    public void fromModel(SV presentation, SV model)
    {   SV v = getPath(model,modelPath);
        setPath(presentation, presentPath, v);
    }
}
```

6-24 – Map class

In figure 6-23 the Mapping List consists of a list of subclasses of Map. The mapping process works by taking each Map object in turn and calling it's fromModel() method using the presentation and model trees as parameters.

We can also embed mapping objects into the presentation tree using a "map" attribute. This is one of the places where having a data model that allows objects to have any attributes we want is an advantage. Those who wrote the presentation objects such as Line and Rect do not need to worry about the mapping process. Figure 6-25 shows how we might do this for our Employee group.

```
Group{
    contents: [
        Rect{ left:0, top:0, width:300, height:40,
            lineWidth:1,drawColor:darkBlue,
            fillColor:lightBlue }
        Text{ x:15, y:30, text:"",
            map: Map{ presentPath:"text",
                modelPath:"name" } }
        Text{ x:200, y:30,text:"",
            map: Currency{ presentPath:"text",
                modelPath:"salary", currency:"USD" }}
    ]
}
```

6-25 – Map objects embedded in the presentation tree.

Mapping functions

The Map class is sufficient to copy the model name into the presentation text but does not account for the currency function that

reformats the salary into a currency specification. For this we define a new Currency class that has additional attributes (currency) and is a subclass of Map. The Currency class simply overrides the fromModel() method to perform the transformation of an integer value into a string representing US Dollars. We can imagine a Currency class that accesses the localization information that transforms into a variety of different currency notations. We could also imagine a variety of string formatting classes that would map information in various ways.

A very powerful map is the Linear class. It has a model and a presentation path as described for Map. In addition it has the attributes modelMin, modelMax, presentMin, presentMax. It would be defined as in figure 6-26.

```
public class Linear extends Map
{   SV presentPath;
    SV modelPath;
    double modelMin;
    double modelMax;
    double presentMin;
    double presentMax;
    public void fromModel(SV presentation, SV model)
    {   SV v = getPath(model,modelPath);
        double mv = v.getDouble();
        double r = (presentMax-presentMin)/
            (modelMax-modelMin);
        pv = (mv-modelMin)*r + presentMin;
        setPath(presentation, presentPath, pv);
    }
}
```

6-26 – Linear Map

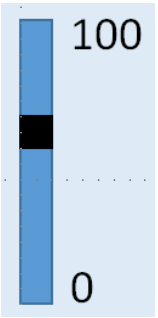
The idea of the linear map is that the model values vary between modelMin and modelMax. The presentation values vary between presentMin and presentMax. These four values are the parameters of the map. We can then compute a linear relationship between the model and presentation values. The relationship is captured in the equation:

$$\frac{Pv - Pmin}{Pmax - Pmin} = \frac{Mv - Mmin}{Mmax - Mmin}$$

By solving for Pv we get the equation:

$$Pv = (Mv - Mmin) * \frac{Pmax - Pmin}{Mmax - Mmin} + Pmin$$

Figure 6-27 shows our original slider problem. Figure 6-28 shows the model and presentation objects for this problem with a Linear mapping object embedded in the presentation. Note in figure 6-28 that presentMin is smaller than presentMax. The use of max and min is to create an association with the corresponding model parameters rather than to actually provide the maximum and the minimum.



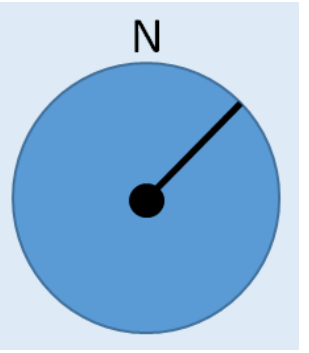
6-27 – A slider

```
Control{ direction: 45, speed: 75 }

Group{ contents: [
    Rect{ top:0, left:0, width:20, height:140,
        lineWidth:1, drawColor:darkBlue, fillColor:blue},
    Text( x:30,y:20, text:"100" },
    Text( x:30,y:140, text:"0" },
    Rect{ top:0, left:0, width:20, height:20,
        fillColor:black,
        map:Linear{ presentPath:"top", modelPath:"speed",
            presentMax:0, presentMin:120,
            modelMax:100, modelMin:0}
        }
    ]
}
```

6-28 – Linear map used to solve the slider presentation

We can also create a Radial mapping object that can resolve the direction dial problem in figure 6-29. The key problem is the position of the end point of the direction arrow. When we look at the parametric equations for ellipses in a later chapter we will see how to use the Linear map for this purpose. For now, however, we will develop a special Radial mapping object.



6-29 – Direction dial

Figure 6-30 shows an implementation for the Radial mapping object. We need to define two sets of parameters: 1) the geometry for the ellipse upon which the point will lie and 2) the angular range that the point will move through. The geometry is easy. We just use the bounding rectangle as with an ellipse. The angular range is in radians, starts on the x axis and goes counterclockwise. We will need to deal with this because our model is defined in degrees and goes clockwise. We also need two separate paths in the presentation for X and Y.

```
public class Radial extends Map
{   SV xPath;
    SV yPath;
    SV modelPath;
    double top,left,width,height; // geometry
    double pMin, pMax; // present angular range
    double mMin, mMax; // model value range
    public void fromModel(SV presentation, SV model)
    {   SV v = getPath(model,modelPath);
        double cx = left+width/2;
        double cy = top+height/2;
        double angle = (v-mMin)*(pMax-pMin)/(mMax-mMin)
            +pMin;
        double x = cx + cos(angle)*width/2;
        double y = cy + sin(angle)*height/2;
        setPath(presentation, xPath, x);
        setPath(presentation, yPath, y);
    }
}
```

6-30 – Radial map implementation

Using our new Radial map object we can now define the presentation for the dial problem in figure 6-29. This is shown in figure 6-31. Note that by setting pMax to a negative number we make the angle move in a clockwise direction based on the degrees coming from the model.

```
Group{ contents:[
       Ellipse{ top:0, left:0, width: 160, height:160,
           lineWidth:1,drawColor:darkBlue,
           fillColor:blue},
       Ellipse{ top:70, left:70, width:20, height:20,
           fillColor:black },
       Line{ x1:80,y1:80,x2:80,y2:0,
           lineWidth:2, drawColor:black,
           map:Radial{ xPath:"x2", yPath:"y2"
               modelPath:"direction",
               top:0, left:0, width: 160, height:160,
               pMin: 1.57075, // PI/2
               pMax: 7.853982, // PI*5/2
               mMin:0, mMax:360}
           }
    ]
}
```

6-31 – Using Radial mapping for the direction dial

There are a variety of other mapping objects that we could define. These are sufficient to show their implementation and how they can be embedded in a presentation. Note also that because the fromModel() method accepts its presentation and model objects as parameters, we can implement a mapping for the list in figure 6-4 using a for loop across the objects in the model and making a copy of the presentation each time. When the copy is made, the map attribute can be used to update the copy according to the corresponding model object.

Summary

In this chapter we have discussed various ways in which we convert a model object into a presentation drawn on the screen. The simplest mechanism was the paint() method that calls the Graphics object's methods to draw the model any time the system needs it drawn. Our next technique was to build a list of presentation objects that capture all of our drawing. We generate this list from the model in much the same way as drawing in the paint() method. The difference is this list is retained and paint() will redraw from the presentation tree whenever needed. The third technique was to get the presentation from some other tool, like a drawing tool and then modify that presentation from the model using a modelUpdate() method, whenever the model is changed. We then looked at using macro processor methods to generate the text of the presentation before it is parsed into data objects. Lastly we looked at creating mapping objects that connect content from the model to content in a presentation tree.

Another chapter

Better list handling

Intermediate models

Sorted and filtered models to be viewed.

Conditional models

Selected models