7

Abstract Model Widgets

The use of a toolkit of pre-built widgets has greatly improved the development of user interfaces. Widget toolkits provide consistency to the user interface, more reliable implementation, easier design and more rapid development. However, the widgets we have seen so far are limited in the kinds of interfaces that they can construct. In this chapter we will look at a widget architecture that supports more complex interactions.

Figure 7.1 shows the classic model-view-controller architecture with the View and the Controller implemented together in a single view class and the model as a separate class. In chapter 6 we showed how this separation allows for many kinds of synchronization and other relationships among views.

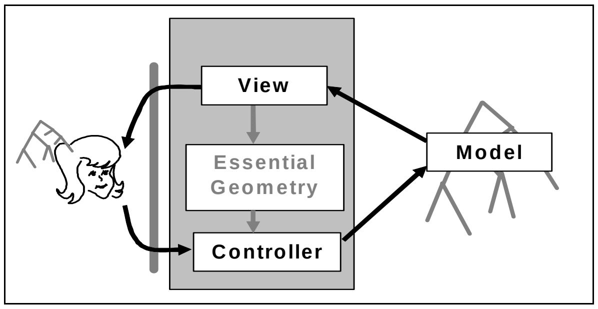


Figure 7.1 – Classic Model-View-Controller

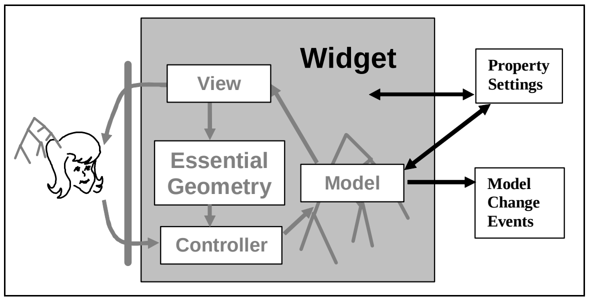


Figure 7.2 – Standard widget architecture

Figure 7.2 shows the standard widget architecture. For the widgets we have looked at so far, the model is simple (integer, string, boolean) and is incorporated into the widget class itself. Other software interacts with the widget by getting and setting properties of the widget and by responding to model change events.

The standard widget architecture works very well for simple models. In addition, we would like to use more complex models. Tables and trees are examples of complex behavior that could be shared across many applications. We want prebuilt widgets for these cases because they occur so often, but the simple model architecture in figure 7.2 does not work on such large models. For these situations we adopt an architecture like that shown in figure 7.3.

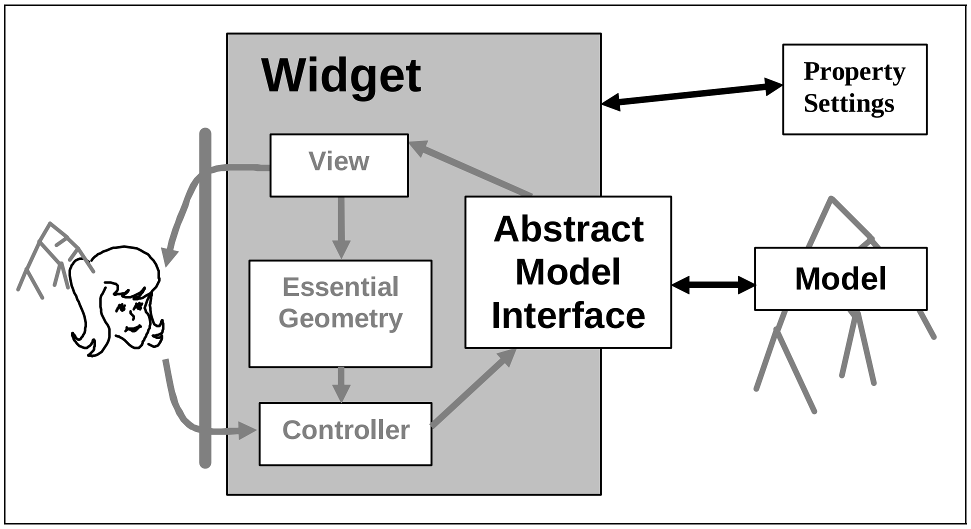


Figure 7.3 – Abstract model architecture

Abstract model widgets have property settings as in the standard widget architecture yet the model is external to the widget as in the classic model-view-controller architecture. The model is represented to the widget as an interface that defines the methods that a model must have. We then can plug any number of models into this widget as long as they conform to the widget's model interface. There are a number of large-scale widgets that can be defined in this way. To understand this architecture, we will review three examples of such widgets and how they relate to their models. They are a tree widget, a table widget and a generalized drawing widget.

In designing abstract model widgets, we look for a common form of interaction and study what is the same about all instances and what is different.

We take the things that are the same and build them into the widget. We take the things that are different and put them either in properties or in the model. We will use this same/difference analysis in each of our three examples.

The architectural key to these kinds of widgets is in the interface to the model. In languages like Java or C# the widget designers create an interface type that specifies all of the methods that the model must provide. A common technique is to also create an *adaptor* class that implements all of the methods and provides default behavior for them. To use these widgets on a given model the model must implement the methods of the interface or must inherit the adaptor class as its super class and then override only those methods that are necessary. In older object-oriented languages such as C++ that do not have interfaces, the widget defines a super class for the abstract model. To use the widget the programmer creates a subclass of the abstract model class.

In many situations the model already exists or must be placed somewhere else in the class inheritance tree. In such situations the programmer creates a *translator* class. The translator class accepts the model in its constructor and then implements all of the widget model methods, passing them on to the appropriate model methods. This forms a translation between the existing model and the needs of the widget's model interface. This architecture is shown in figure 7.4.

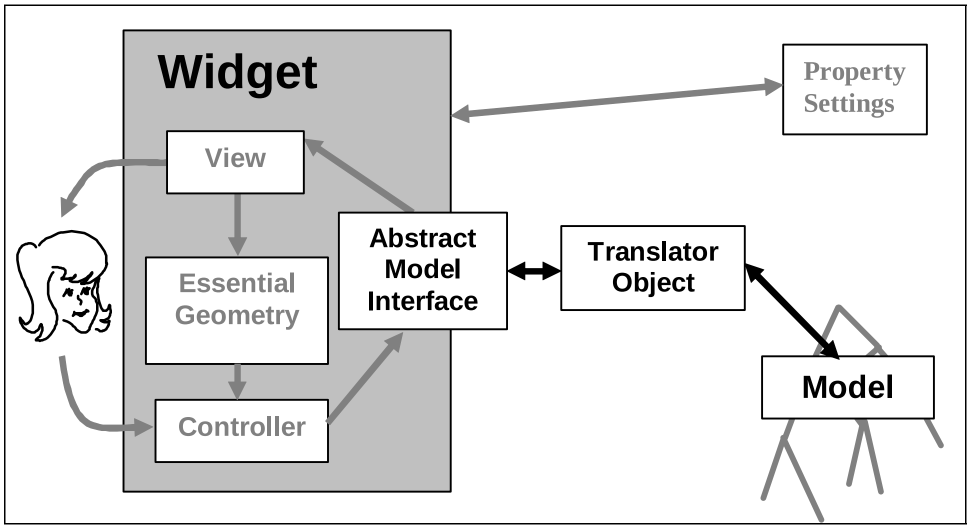


Figure 7.4 – Translator object for widget/model interface

Tree Widget

One of the common ways for people to organize information and objects is in trees. This allows one to group things by similarity, topic, location or a variety of other purposes. Because trees occur so often in applications, we want to do only one implementation of the user interface. Dragging, dropping, opening and closing of tree nodes can all be programmed only one time and provide a consistent interactive behavior for these tasks. To design a tree widget, we first look at several examples of trees in user interfaces.

Figure 7.5 shows trees from four different applications: A) file folders from the Windows file explorer, B) bookmarks tree form the Firefox web brower, C) package explorer from the Eclipse Java development tool, and D) mail boxes from the Eudora email reader.

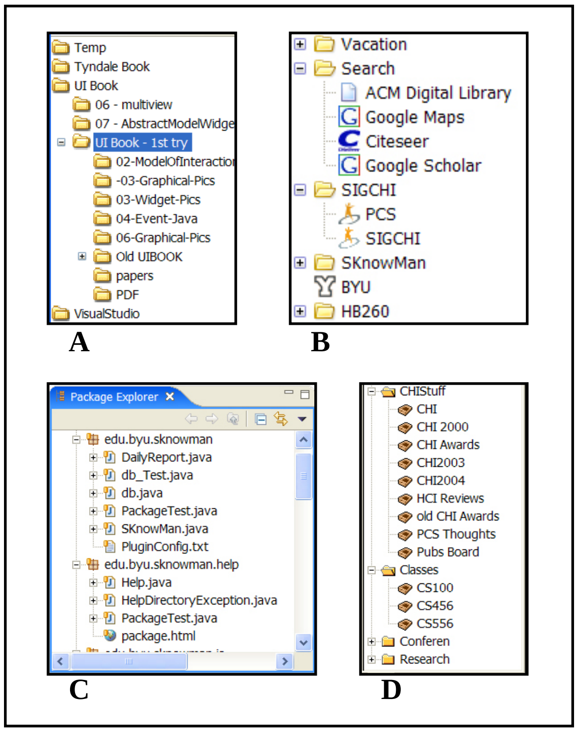


Figure 7.5 – Example tree widgets

We first look at how they are all the same. When there are many things in common, there is an advantage to building a widget that shares and reuses the commonality.

- They all represent a tree as an indented outline using the same layout.
- Every node of the tree has a textual name and one or two icons. Containers have two icons, one for open and one for closed.
- They all use little plus/minus boxes to open/close the container
- Clicking on the name allows the user to change the name of any of the nodes.

We now look at how they are different. The differences are:

- Containers have variable numbers of children arranged in varying depths in the tree.
- Every node has a different name.
- There are different icons for different applications and for different items within those applications.
- Three of the four examples use faint gray lines to indicate which nodes are siblings. One example (A) does not.
- Example B uses a different font and size of font.

The similarities we build into the widget. The differences are built into the properties and into the abstract model. The properties that our widget provides might be:

- Font face, size and style
- boolean to display sibling lines
- number of pixels to indent at each level

The abstract model comes in two parts. The first is the model itself that our widget will manipulate and the second is the notification mechanism for the widget to learn about model changes. The model itself can be defined around the `TreeNode` interface shown in figure 7.6.

```
public interface TreeNode
{
        int numberOfChildren();   // how many children nodes are there. Zero for leaf nodes
        TreeNode getChild(int childIdx);              // get one of this node's children
        Image getIcon();              // gets the primary icon for this node
        Image getOpenIcon();      // if this is a container this returns the icon for when it is open
        String getName()            // gets the name of this node
        void setName(String name);    // changes the name of this node
}
```

Figure 7.6 – Abstract TreeNode model

This interface provides all of the information the widget needs to know about the tree. With this interface the widget can recursively tour the tree. The widget must also know about changes to the tree. Since changes can occur at any point in the tree, we need to notify the widget as to where the change occurred. Since each container provides access to its children by an index, we can locate any node in the tree with a *path* or sequence of indices that leads from the root of the tree to the desired node. We represent this using Java's generic template notation `List<int>`. The `TreeWidget` class needs to implement the methods shown in figure 7.7.

```
public class TreeWidget extends Widget
{
        public void nodeChanged(List<int> pathToChangedNode){ . . . }
        public void nodeToInsert(List<int> insertLocation) { . . . }
        public void nodeToDelete(List<int> deleteLocation) { . . . }
}
```

Figure 7.7 – Widget notification of model change

Our tree model is not complete because we need to provide a mechanism to register the widget with the model. We also need to provide methods that let the widget make changes to the model. The `TreeRoot` class in figure 7.8 provides the necessary methods for this purpose.

```
public interface TreeRoot
{
        public TreeNode root;
        public void addListenerWidget(TreeWidget widget);
        public void removeListenerWidget(TreeWidget widget);
        public void deleteNode(List<int> deleteLocation);
        public void moveNode(List<int> fromLocation, List<int> toLocation);
        public void changeName(List<int> nodeToChange, String newName);
}
```

Figure 7.8 – TreeRoot access

As an example we can use this widget to provide the file explorer interface from figure 7.5(A). The file explorer is an example of a model that we are not allowed to modify. Microsoft provides the API to access the file system and is not going to let us see the source code. For this we will need three translator classes. We can create a FileRoot class that implements TreeRoot. This class can have a constructor that accepts a path name for the root folder. We will need a TreeFolder class and a TreeFile class to interface to folders and files respectively. Both of these must implement the TreeNode interface. A TreeFolder would find out the number of files and folders that it contains and report them as children. Using the file system API the TreeFolder class will create TreeFolder or TreeFile classes for the folders and files that it contains. Creating these classes to access the file system is much simpler than implementing the tree widget. The entire file system is obviously too large to copy into the tree widget model. The abstract interface technique provides us the advantage of pre-built widgets on a signification model.

Varying the widget

Our abstract model architecture also allows us to create many kinds of tree widgets. Figure 7.9 shows Robertson, Mackinlay and Card's cone tree representation[1]. The model interface is the same as what we have shown, but the presentation and manipulation of the tree is very different. Cone trees use 3D presentation to give a spatial sense of the whole tree and its contents.

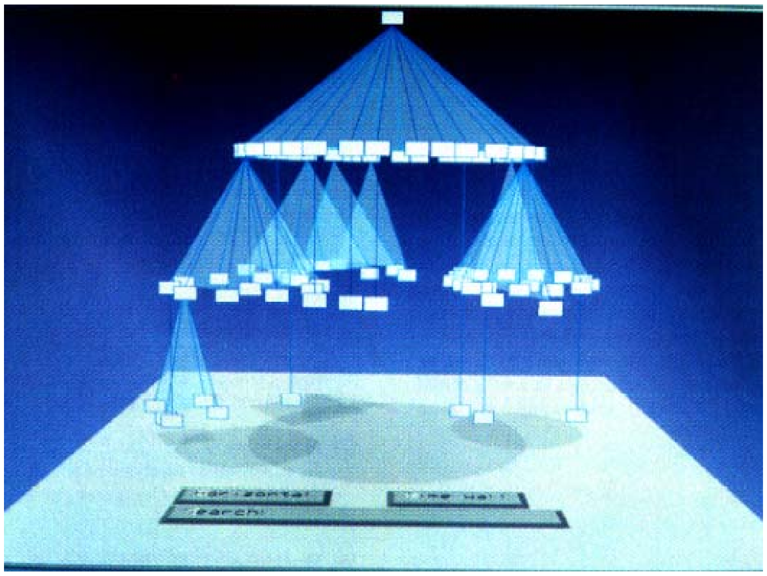


Figure 7.9 – Cone Trees[1]

Figure 7.10 shows hyperbolic trees from Lamping and Rao[2], where the nodes of a tree are laid out on hyperbolic plane. In hyperbolic geometry the circumference of a circle grows exponentially with its radius. This maps naturally to tree layouts where the number of nodes to be displayed grows exponentially with tree depth. The hyperbolic geometry is then mapped back to a unit circle and scaled up to the display size. This allows flexible layout with enough room for more nodes to be displayed.

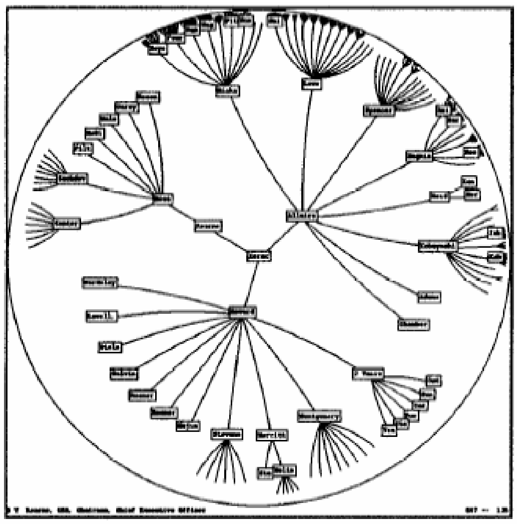


Figure 7.10 – Hyperbolic trees[2]

There are some tree models where properties other than the name are to be presented. There are a variety of displays where a "size" property of each tree node is to be presented. The "size" can be many things. It can be the number of bytes allocated to files and folders, the total sales volume of an organization or just the number of items in the tree. To accommodate such widgets we only need to augment the abstract model from figure 7.6 with a getSize() method. Figure 7.11 shows a tree and a corresponding TreeViz[3] map of its content. The area of items in the map corresponds to their "size".

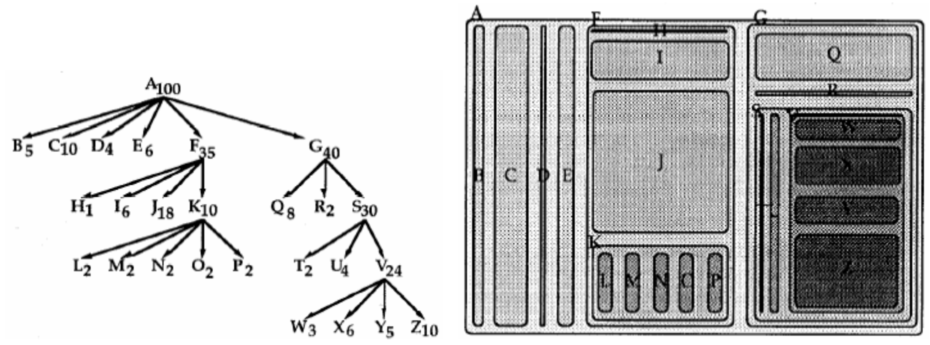


Figure 7.11 – TreeViz tree maps[3]

Maps of trees using size can be further enhanced by mapping model properties to other visual information such as color or texture. Figure 7.12 shows beam trees by van Wijk, van Ham and van de Wetering[4]. Figure 7.13 shows their even more imaginative fruit trees with properties encoded as various shapes of fruit.

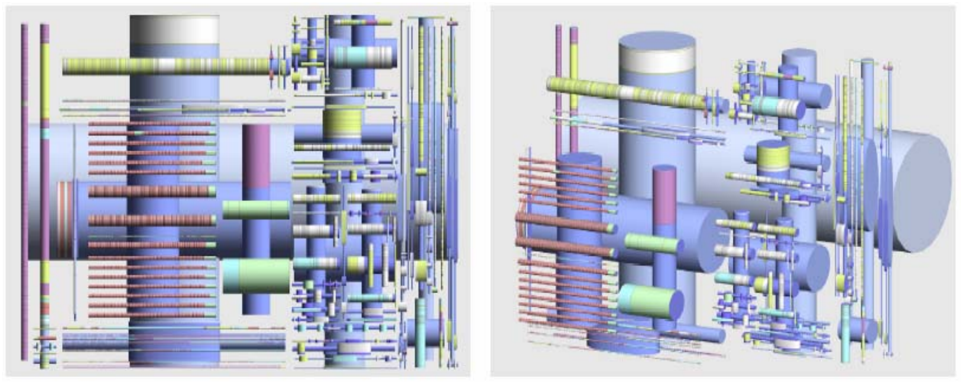


Figure 7.12 – Beam Trees[4]

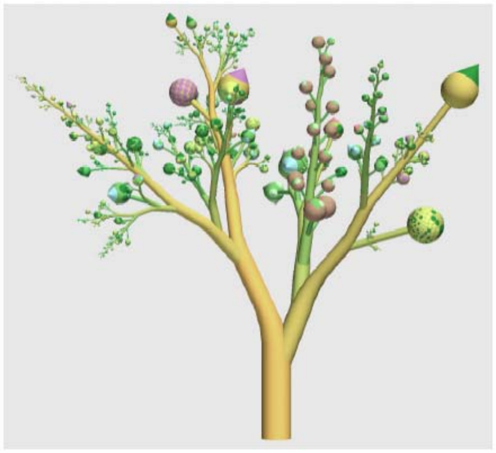


Figure 7.13 – Fruit Trees

Because trees are widely used for organizing information, the size property can be replaced by a “degree of interest” property. The degree of interest is an estimate of how interesting or valuable a particular tree node may be to a set of users. Degree of interest tree widgets attempt to present as much of the “interesting” parts of a tree as possible within a given amount of screen space. Figure 7.14 shows an example of such a tree.

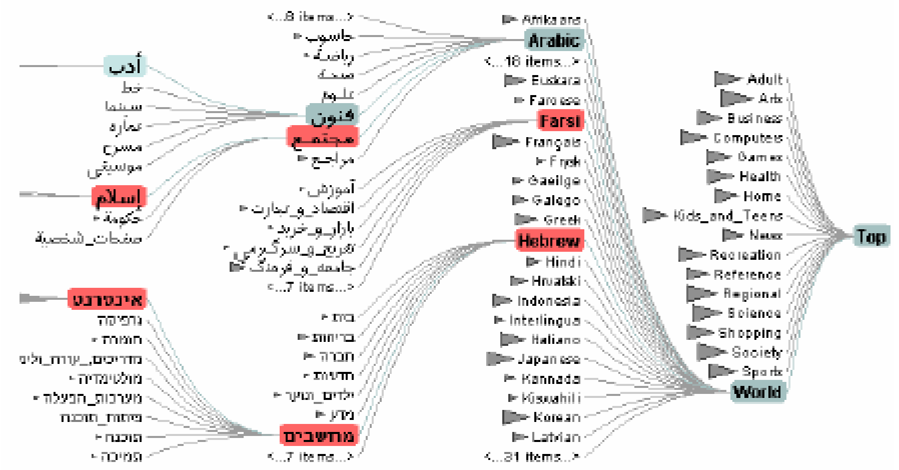
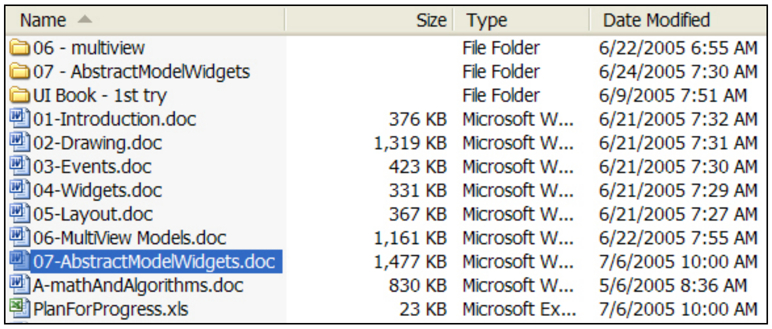


Figure 7.14 – Degree of Interest trees[5]

With a relatively simple abstract model of a tree a connection is forged between a variety of actual data models and tree presentation widgets. Conversely this same abstract model allows a single application model to be presented by a variety of tree forms each with different properties.

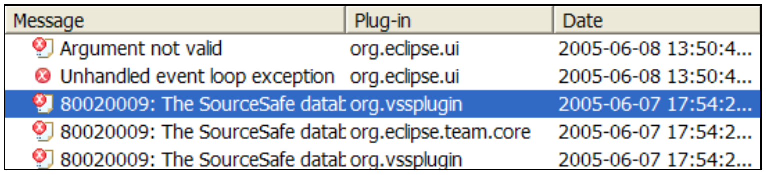
Table Widget

Figures 7.15 through 7.18 show various uses of a table widget. Each is from a different application yet all have a similar presentation and interaction techniques. We can implement all of these using the same design approach and architectural structure that we used for the tree widget.



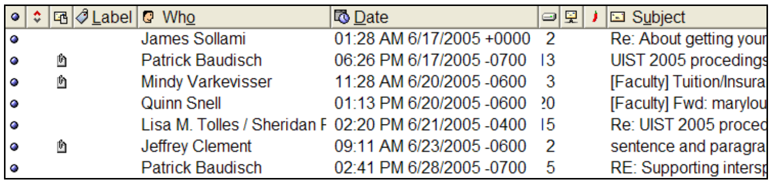
Name	Size	Type	Date Modified
06 - multiview		File Folder	6/22/2005 6:55 AM
07 - AbstractModelWidgets		File Folder	6/24/2005 7:30 AM
UI Book - 1st try		File Folder	6/9/2005 7:51 AM
01-Introduction.doc	376 KB	Microsoft W...	6/21/2005 7:32 AM
02-Drawing.doc	1,319 KB	Microsoft W...	6/21/2005 7:31 AM
03-Events.doc	423 KB	Microsoft W...	6/21/2005 7:30 AM
04-Widgets.doc	331 KB	Microsoft W...	6/21/2005 7:29 AM
05-Layout.doc	367 KB	Microsoft W...	6/21/2005 7:27 AM
06-MultiView Models.doc	1,161 KB	Microsoft W...	6/22/2005 7:55 AM
07-AbstractModelWidgets.doc	1,477 KB	Microsoft W...	7/6/2005 10:00 AM
A-mathAndAlgorithms.doc	830 KB	Microsoft W...	5/6/2005 8:36 AM
PlanForProgress.xls	23 KB	Microsoft Ex...	7/6/2005 10:00 AM

Figure 7.15 – Microsoft file table



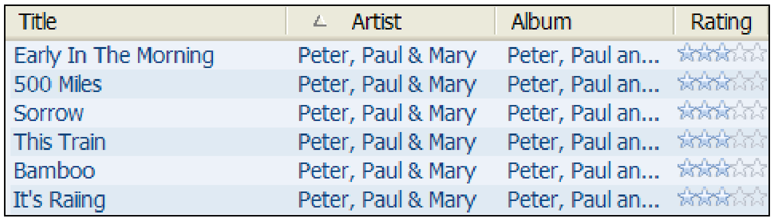
Message	Plug-in	Date
Argument not valid	org.eclipse.ui	2005-06-08 13:50:4...
Unhandled event loop exception	org.eclipse.ui	2005-06-08 13:50:4...
80020009: The SourceSafe datat	org.vssplugin	2005-06-07 17:54:2...
80020009: The SourceSafe datat	org.eclipse.team.core	2005-06-07 17:54:2...
80020009: The SourceSafe datat	org.vssplugin	2005-06-07 17:54:2...

Figure 7.16 – Eclipse error log



			Label	Who	Date				Subject
●				James Sollami	01:28 AM 6/17/2005 +0000	2			Re: About getting your
●		📎		Patrick Baudisch	06:26 PM 6/17/2005 -0700	I3			UIST 2005 procedings
●		📎		Mindy Varkevisser	11:28 AM 6/20/2005 -0600	3			[Faculty] Tuition/Insura
●				Quinn Snell	01:13 PM 6/20/2005 -0600	?0			[Faculty] Fwd: marylou
●				Lisa M. Tolles / Sheridan F	02:20 PM 6/21/2005 -0400	I5			Re: UIST 2005 proced
●		📎		Jeffrey Clement	09:11 AM 6/23/2005 -0600	2			sentence and paragra
●				Patrick Baudisch	02:41 PM 6/28/2005 -0700	5			RE: Supporting intersp

Figure 7.17 – Eudora mail message list



Title	Artist	Album	Rating
Early In The Morning	Peter, Paul & Mary	Peter, Paul an...	★★★☆☆
500 Miles	Peter, Paul & Mary	Peter, Paul an...	★★★☆☆
Sorrow	Peter, Paul & Mary	Peter, Paul an...	★★★☆☆
This Train	Peter, Paul & Mary	Peter, Paul an...	★★★☆☆
Bamboo	Peter, Paul & Mary	Peter, Paul an...	★★★☆☆
It's Raiing	Peter, Paul & Mary	Peter, Paul an...	★★★☆☆

Figure 7.18 – Music list

We begin by identifying the similarities. Without a large degree of similarity there is no point in trying to design a general purpose widget. There are, however, many things that these widgets have in common.

- There are a fixed set of columns each with a title and/or possibly an icon.
- There are an arbitrary number of rows that will probably require scrolling.
- Every cell of the table has a string and/or an icon for its content. If the icon and string occur together the icon is first.
- Rows can be sorted based on any of the column contents.
- Rows can be selected and that selection used elsewhere. The use of selection is not shown in these figures. Selecting a song in figure 7.18 will play that song. Selecting a message in figure 7.17 will display the contents of the message.
- It is possible, though not visible in the figures, to hide or show columns and to resize their width.

There are also differences that we must capture in our abstract model and in the properties.

- Though the number of columns is fixed there are varying numbers of columns between applications.
- Columns have different names, icons and widths.
- The contents of rows and cells are all different.
- There are differences in font and background color.

The key to the table widget is the communication between the widget and the model both to retrieve and change model information as well as to be notified of changes to the model. In most situations the table is used primarily for viewing of information rather than editing it. However, selected rows can be deleted and frequently cell string contents can be edited and changed. More complex manipulations of the table model are generally handled by other widgets and then notifications forwarded to the widget. The model-view-controller architecture automatically handles the cooperation between the table widget and other widgets by means of the shared model.

Table Model

A possible abstract model for a table widget is shown in figure 7.19. As usual we need a mechanism to register listeners. The widget needs to know how many rows there are and needs descriptors for the columns. These descriptors provide information about the name, icon, width and justification of the column. The name and width can be interactively changed, so we provide methods to tell the model when that happens. A column also has a `columnID` that we can use to identify the column to the model when manipulating cells.

```
public interface TableModel
{
    public void addTableListener(TableListener listener);
    public void removeTableListener(TableListener listener);
    public int nRows();   // returns the number of rows
    public TableColumn [] getColumns();      // returns descriptors for all of the columns
    public String getCellString(int rowIndex, int columnID);
    public void setCellString(int rowIndex, int columnID,String newCellValue);
    public Image getCellIcon(int rowIndex, int columnID);
    public void selectRow(int rowIndex);       // notifies the model when a row is selected
    public void deleteRow(int rowIndex);
}
public enum ColumnJustify{LEFT, CENTER, RIGHT}
public interface TableColumn
{
    public int columnID();
    public ColumnJustify justification();
    public String getColumnName();
    public void setColumnName(String newName);
    public Image getColumnIcon();
    public int getColumnWidth();
    public void setColumnWidth(int widthInPixels);
    public boolean isReadOnly();
}
```

Figure 7.19 – Abstract model for Table widget

If a cell has an icon but no string (as in the Rating column of figure 7.18) then a null is returned by getCellString(). If there is string with no icon, as in most of the columns, then the getCellIcon() method returns null.

Sorting of rows can be handled without the assistance of the model by using the string value for the cells. This provides great functionality without any effort in the model. The model simply provides data.

Notification Interface

In figure 7.7 we integrated the notification methods into the TreeWidget class. This simplified the architecture. For TableWidget we will define a TableListener interface for notification. The difference in the approaches lies in whether we believe that there will be other classes of objects that will be interested in receiving notifications. Our TableListener interface is shown in figure 7.20. This interface provides for notification at various levels of granularity. The listeners and models can then decide for themselves how they wish to deal with the changes. A simple model would just call tableChanged() on every change. For small table models this would be just fine. For very large models, this may cause efficiency problems. In a large model, most rows would not be visible and therefore changes to those rows would not affect the presentation.

```
public interface TableListener
{
        public void cellToChange(int rowIndex, int columnID, String newCellValue);
        public void rowChanged(int rowIndex);
        public void rowsToDelete(int firstRowIndex, int nRowsToDelete);
        public void rowsInserted(int firstRowIndex, int nRowsInserted);
        public void columnDescChanged(int columnID);
        publis void tableChanged();
}
```

Figure 7.20 – TableListener interface

Table properties

Our Table widget also needs properties. A possible set of properties would be: header font, cell font, background color, and alternating line background color. The alternating line background color would handle the effect in figure 7.18 that highlights rows with slightly different colors. We may also need column properties. Figure 7.19 shows column justification as part of the model. The column justification generally does not change very often. Its behavior is more like a property where it is set at the initialization of the widget and rarely modified. The justification property was placed in the model because the model already had column descriptors.

Information that varies among widget instances can appear in the model or in the properties. The decision about where they should appear depends on two issues: how frequently the information changes as a natural part of the interaction and whether the information should normally be set in code or by an interface design environment. Information that normally is established at widget initialization should probably be a property. Information that is frequently modified by the end user should be in the model. Information to be specified at design time should probably be a property that is available to the interface design environment.

In the case of our Table widget, the width of a column is changed interactively by the user by dragging column borders. The justification of a column is generally not modified by end users. The justification is generally specified at design time. These facts would lead us to place column justification and possibly column color, border and font in properties. However, as we will see in chapter 9 on interface design environments, single value properties are easier to deal with than multiple complex properties such as multiple column descriptors each with several property values. Ultimately the decision on whether column justification should be part of the model or a property will probably rest upon the capabilities of the target interface design environment. This issue will be revisited in chapter 9.

Varying the Table widget

Because tables are such a common technique for representing lists of information and because they map naturally to the results from relational databases, there have been many extensions and variations. The key idea, however, is that the abstract model interface remains essentially unchanged.

Most of the variations in the table widget attempt to address very large tables where very little of the information in the table will fit on the screen. The very early Table Lens[6] used a focus + context technique. In this technique selected rows, columns or cells are given more prominence than the others. The user is able to select items that they wish to see more clearly. These are expanded and the others are reduced. Figure 7.21 shows the modified table layout when cell G4 and H5 are selected for emphasis.

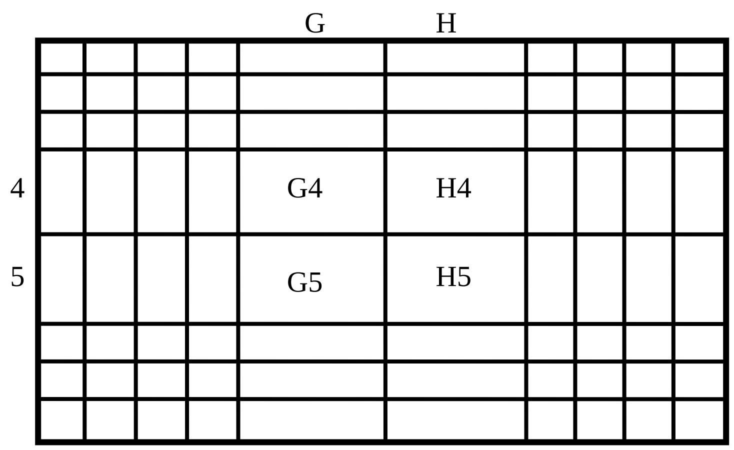


Figure 7.21 – Table Lens focus technique

A major challenge is dealing with cell contents that need to be shrunk down to make room for the focus cells. Rao and Card addressed this by identifying those columns that contained numeric rather than textual information. We can do this by adding a column property that indicates numeric columns. Their brief form of numeric information is a bar indicating the size of the value relative to the maximum value for that column. An alternative presentation is a dot indicating the position of the value in the range of possible values for a column. Shrunken textual cells are simply shown in gray. All of these techniques are shown in figure 7.22 which shows several focus columns and rows. The user gets a sense of the information without actually seeing all of the values.

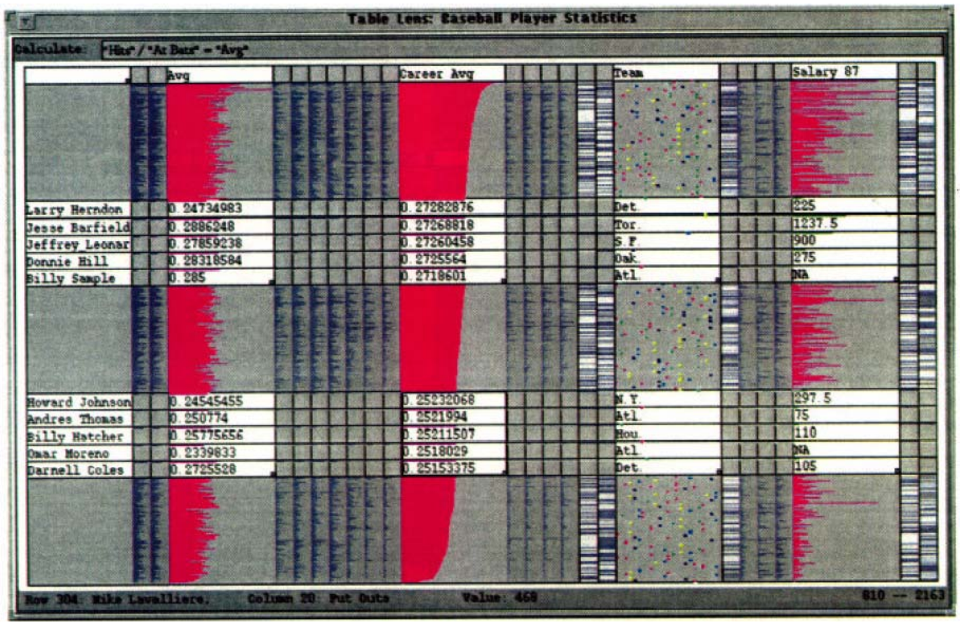


Table Lens: Baseball Player Statistics

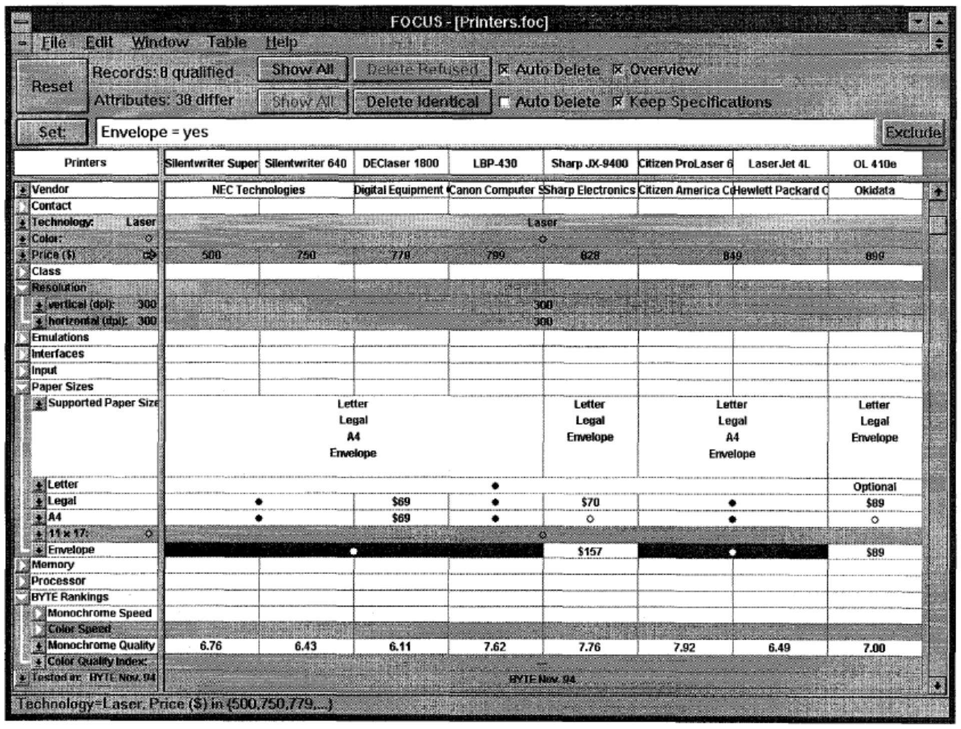
Calculate: "Hits" / "At Bats" = "Avg"

	Avg	Career Avg	Team	Salary 87
Larry Herndon	0.24734983	0.27282876	Det.	225
Jesse Barfield	0.2886248	0.27268818	Tor.	1237.5
Jeffrey_Leonar	0.27859238	0.27260458	S.F.	900
Donnie Hill	0.28318584	0.2725564	Oak.	275
Billy Sample	0.285	0.2718601	Atl.	NA
Howard Johnson	0.24545455	0.25232068	N.Y.	297.5
Andres Thomas	0.250774	0.2521994	Atl.	75
Billy Hatcher	0.25775656	0.25211507	Hou.	110
Omar Moreno	0.2339833	0.2518029	Atl.	NA
Darnell Coles	0.2725528	0.25153375	Det.	105

Row 304: Mike Lavalliere. Column 20: Put Outs Value: 468 810 -- 2163

Figure 7.22 – Table Lens

Another mechanism for handling very large tables is to restrict the rows/columns that are actually shown. The FOCUS[7] table transposes the traditional display and shows attributes as rows rather than columns. Since each row of their table corresponds to an attribute and each in that row contains a value, selecting a cell defines a query of the form "<attribute> == <value>". When a cell or range of cells is selected, the "SET" command is used to convert that selection into a query term. The "EXCLUDE" command will convert the selection into a query term that excludes all columns that match the query. The use of shift selection can add multiple query terms that are then ORed together. Successively created query terms are ANDed together. With this mechanism a user can rapidly change the set of columns that are displayed. There is also an option for retaining unselected columns but showing them reduced and grayed out. A novel approach introduced by the FOCUS table is to merge adjacent cells that share the same value. This is shown in figure 7.23. FOCUS also retained the focus technique and numerical bar technique from Table Lens.



Records: 8 qualified

Attributes: 38 differ

[✓ Auto Delete] [✓ Overview]

[ ] Auto Delete [✓ Keep Specifications]

Set: Envelope = yes

Printers	Silentwriter Super	Silentwriter 640	DEClaser 1800	LBP-430	Sharp JX-9400	Citizen ProLaser 6	LaserJet 4L	OL 410e
Vendor	NEC Technologies		Digital Equipment	Canon Computer S	Sharp Electronics	Citizen America C	Hewlett Packard C	Okidata
Contact								
Technology: Laser	Laser							
Color: ○	○							
Price ($)	500	750	779	799	828	849		899
Class								
Resolution								
vertical (dpi): 300	300							
horizontal (dpi): 300	300							
Emulations								
Interfaces								
Input								
Paper Sizes								
Supported Paper Size	Letter Legal A4 Envelope				Letter Legal Envelope	Letter Legal A4 Envelope		Letter Legal Envelope
Letter	●							Optional
Legal	●		$69	●	$70	●		$89
A4	●		$69	●	○	●		○
11 x 17: ○	○							
Envelope	●				$157	●		$89
Memory								
Processor								
BYTE Rankings								
Monochrome Speed								
Color Speed								
Monochrome Quality	6.76	6.43	6.11	7.62	7.76	7.92	6.49	7.00
Color Quality Index:	--							
Tested in: BYTE Nov. 94	BYTE Nov. 94							

Technology=Laser, Price ($) in {500,750,779,...}

Figure 7.23 – FOCUS table

In the case of tabular data where all of the columns have numeric data a presentation called *parallel axes* can be used. This presentation tends to show when values are correlated in table entries. Each column becomes an axis on which the position of a value can be plotted. A row of the table is displayed as a line linking the values for that object. Figure 7.24 shows how these can display clusters of values. As with trees, the interfaces allow for many models to use a particular widget implementation and for a single model to use many different widgets to present information.

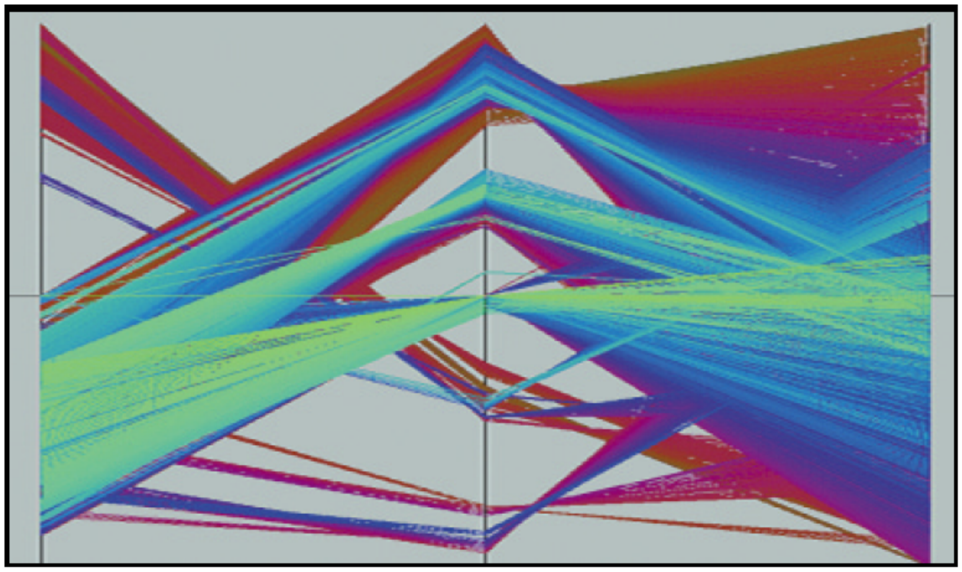


Figure 7.24 – Parallel Axes[8]

Drawing Widget

Trees and tables have relatively simple abstract models and relatively straightforward interaction techniques. We will now consider a more complex widget inspired by the Workspaces architecture[9]. Figure 7.25 shows two applications, one of them is for drawing and the other is for laying out widgets in a user interface design. Though these two applications have very different purposes, they have very similar interactive behavior.

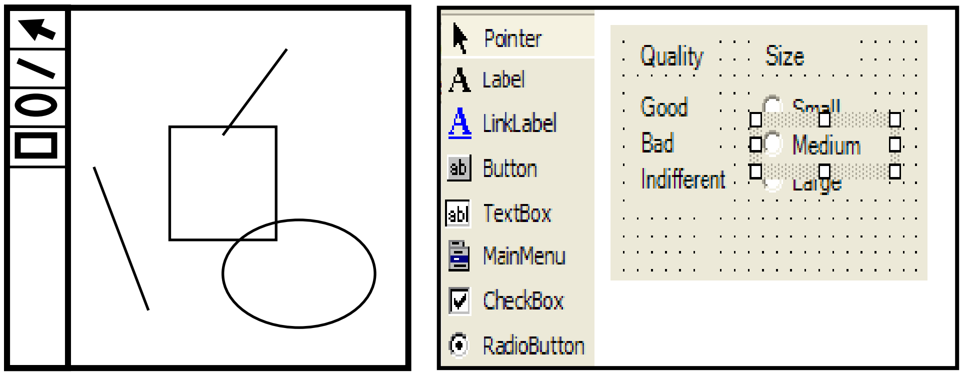


Figure 7.25 – Drawing widgets

First we consider how these two applications are similar. The similarities lie in the way that they interact with objects. This particular widget has rather complicated interactive behavior. Separating the behavior out into a widget simplifies the creation of complex applications.

- The model is fundamentally a list of objects to be drawn on a surface.
- There is a menu of objects that can be selected and placed in the drawing area.
- Creating new objects uses the same event sequence and interactive behavior. There are three forms: click to place an item, down-drag-up to create two control points, and placement of multiple control points.
- Selection works by clicking on objects, shift-clicking to select multiple objects and rubberband rectangles to select a group of objects.
- Objects can be dragged around the draw area.
- Object are manipulated by dragging control points and those control points are displayed in a similar way.
- Deleting objects is the same.

There are obvious differences:

- The set of objects to be created is very different as is their appearance in the menu.
- How each object is drawn and how each relates to its control points is different.
- The geometry for selecting objects differs.

Drawing Model

The model for a drawing object is quite simple. However, the model is special because ultimately the model is the presentation. One of the ways in which objects in the drawing model differ is in the way that they draw themselves and in the way that they handle their geometry. Selecting a circle is very different from selecting a line or selecting a scroll-bar. Because of this fuzzy relationship between the model and the presentation we will spend more time on how the widget itself is constructed.

The model for a drawing is simply a sequence of drawing objects that can be painted in order and a list of possible classes for drawing objects. Figure 7.26 shows a possible interface for a DrawingModel.

```
public interface DrawingModel
{
	public void addDrawingListener( DrawingListener listener);
	public void removeDrawingListener(DrawingListener listener);

	public int nDrawingObjects();
	public DrawingObject getDrawingObject(int index);

	public void deleteDrawingObject(int index);
	public void addDrawingObject(DrawingObject newObject);

	public int nDrawingClasses();
	public DrawingClass getDrawingClass(int index);
}

pubic interface DrawingObject
{
	public void redraw(Graphics g);
	public boolean isSelected(Point mousePoint);
	public Rectangle getBounds();

	public int nControlPoints();
	public Point getControlPoint(int index);
	public void setControlPoint(int index, Point newPoint);
	public void addControlPoint(Point newPoint);
	public void move(int dX, int dY);
}

public enum InputSyntax{SINGLEPOINT, DRAGPOINT, MULTIPOINT}
public interface DrawingClass
{
	public Image getIcon();
	public String getName();
	public DrawingObject createNew();
	public InputSyntax getSyntax();
}
```

Figure 7.26 – Drawing model

The DrawingModel itself consists of only three parts: the registration of listeners, the list of drawing objects with means for deleting and adding them, and a list of object classes that can be created. A DrawingObject needs a means for drawing itself to a Graphics object (redraw), support for selection(isSelected and getBounds) and mechanisms for manipulating control points. The DrawingClass contains the necessary information to create the menu and to create new objects.

There are 6 basic tasks that a widget must perform. These tasks depend on whether the widget is in object creation mode (when one of the object classes is selected in the menu) or in selection mode (when the pointer is selected in the menu). In addition to the model, the most important data item retained by the widget is a list of indices of those objects that are currently selected. Much of the interaction is based on the currently selected object set. For our simple drawing widget, the tasks are:

- Redraw the drawing from the model.
- Create new objects on the drawing surface
- Select objects
- Drag objects around
- Drag control points
- Delete objects

We will address each of these tasks in turn and show how the widget can do them using the abstract model. Redrawing consists of drawing the objects and then drawing the control points of any selected objects. This algorithm is shown in figure 7.27.

```
public class DrawingWidget
{       private DrawingModel myModel;
        private int[] selectedObjects;
        . . . .
        public void redraw(Graphics g)
        {       for (int i=0;i<myModel.nDrawingObjects();i++)
                {       myModel.getDrawingObject(i).redraw(g); }
                for (int i=0;i<selectedObjects.length; i++)
                {
                        DrawingObject so=myModel.getDrawingObject(i);
                        for (int cp=0;cp<so.nControlPoints();cp++)
                        {       Point p=so.getControlPoint(cp);
                                draw control point at point p
                        }
                }
        }
}
```

Figure 7.27 – Redraw for DrawingWidget

The redraw code is not very complicated because the DrawingObject implementations do most of the work.

Creating new drawing objects

Before we can create new objects we need to build the menu of objects that we are offering to the user. The DrawingModel's nDrawingClasses() and getDrawingClass() methods give us the information that we need. For each DrawingClass the getIcon() and getName() methods give us the information that we need to fill the menu.

One of the important things about this abstract model architecture is that it is readily extended. Even if we never reuse our DrawingWidget for more than one application, the architecture still has significant advantages because it allows the design to grow over time. For a company to make money over the long term it must improve its product so that it can sell new versions. This economic drive for a new version will lead to new features. If we built our drawing application with all of the objects hard-coded into the widget, then adding new kinds of objects will be difficult because the code must be changed in so many places. With this architecture, we simply add another class to the model and implement a new DrawingObject. We will see the importance of such extensibility in chapter 8 on interface design environments.

To create a new object the user selects an item from the object menu and then uses the mouse in the drawing area to show where the object is to be created. There are three kinds of input syntax depending on the geometry of the object to be placed. These three types are captured in the InputSyntax enumeration in figure 7.26. The fact that input syntax (the sequence of mouse events) is frequently the same across many widgets was the primary reason that the view and the controller were separated in the original MVC architecture. Myers describes a number of such general syntax models. The three that we will use here are simplifications of some of his interactors[10]. The details of how to design input syntax will be discussed in more detail in chapter 11.

The syntax for these three creation techniques will be spread over the mouse-down, mouse-move and mouse-up event handling methods. Each method must chose its behavior based on the currently selected drawing class's input syntax.

Single Click

Some drawing objects like locations on a map, locating a text string or special marks only have a single point. Figure 7.28 shows how this would be handled. Positioning the object is all done by setting its zero control point. The drawing object is responsible for getting itself drawn in the right position. This syntax allows the user place an object and move it into position in a single motion.

```
DrawingObject newObj;
on mouse down
      DrawingClass dc = object class selected in the menu;
      newObj = dc.createNew();
      newObj.setControlPoint(0, current mouse position );
      myModel.addDrawingObject(do);
on mouse move
      newObj.setControlPoint(0,current mouse position );
on mouse up
      newObj.setContolPoint(0,current mouse position );
```

Figure 7.28 – Single click input syntax

Two Point Drag

Many objects such as lines, rectangles, ellipses, and widgets require two points for their initial geometry. They may have other control points later to allow many ways to resize or manipulate the object, but they all use the mouse-down, drag, mouse-up sequence to specify two points. Figure 7.29 shows how this might be done.

```
DrawingObject newObj;
on mouse down
      DrawingClass dc = object class selected in the menu;
      newObj = dc.createNew();
      newObj.setControlPoint(0, current mouse position );
      newObj.setControlPoint(1, current mouse position );
      myModel.addDrawingObject(do);
on mouse move
      newObj.setControlPoint(1,current mouse position );
on mouse up
      newObj.setContolPoint(1,current mouse position );
```

Figure 7.29 – Two-point drag input syntax

Multi-point input

Multiple input points are required to create polylines with many segments, polygons and curves. For these there is a first click to identify point zero and then successive clicks to create new points. The process terminates when a click occurs on the first point (to close the shape) or on the last point (to end a polyline or curve).

Selecting objects

Using our abstract model, the selection of objects is a simple loop. It is the individual drawing objects that must work out the geometry issues in their isSelected() method. Figure 7.30 shows selection with a single point. Note that the loop operates backwards so that the top item under the mouse (the last one drawn) is selected first.

```
public class DrawingWidget
{       public DrawingModel myModel;
        public int selectObject(Point selectPoint)
        {
                for (int i=myModel.nDrawingObjects()-1;i>=0;i--)
                {
                        DrawingObject do=myModel.getDrawingObject(i);
                        if (do.isSelected(selectPoint) )
                                return i;
                }
                return -1; // no selection
        }
        . . . .
}
```

Figure 7.30 – Single point selection

Similarly figure 7.31 shows selection with a rectangle.

```
public class DrawingWidget
{
        . . .
        public IndexList selectObjects(Rectangle select)
        {
                IndexList result= empty list of object indices;
                for (int i=0;i<myModel.nDrawingObjects();i++)
                {
                        DrawingObject do=myModel.getDrawingObject(i);
                        if (do.getBounds().isInsideOf(select))
                                result.add(i);
                }
                return result;
        }
}
```

Figure 7.31 – Rectangle selection

Other Tasks

Dragging objects is simply a matter of looping through the list of selected objects and calling the move() method for each one. To drag a control point the widget loops through the selected objects and through each of their control points to find the object and point for dragging. Once this is known, dragging is performed by changing that control point each time the mouse moves. Deleting objects involves looping through the selected objects from highest to lowest index and deleting each one. The order is important so that early deletes do not change the index for later selected objects.

Most of these tasks are embedded in the three event methods for mouse down, mouse move and mouse up. Sorting out which is active and when each should be performed can become complicated. It must be thought through carefully. In chapter 11 we will study design techniques for working out these complexities.

Summary

In this chapter we have worked through a software architecture for widgets where the details of the model are not known. We define an abstraction of the model using one or more interface definitions. This interface defines what we need to know about the model in order for the widget to function correctly. Based on the abstract model we can build a powerful widget. To reuse the widget the developer must only implement the model or adapter classes that map the abstract model to the actual model. This takes much less time and is more reliable than building the user interface from scratch. Even in cases where a widget will only be used once, the techniques of abstract models can be used to enhance the extensibility of the application.

[1] Robertson, G. G., Mackinlay, J. D., and Card, S. K. “Cone Trees: Animated 3D Visualizations of Hierarchical Information”. *Human Factors in Computing Systems (CHI '91)*. (1991) 189-194.

[2] Lamping, J. and Rao, R. “Laying Out and Visualizing Large Trees Using a Hyperbolic Space.” *User interface Software and Technology (UIST '94)*. (1994) 13-14.

[3] Johnson, B. “TreeViz: Treemap Visualization of Hierarchically Structured Information.” *Human Factors in Computing Systems (CHI '92)*. (1992) 369-370.

[4] van Wijk, J. J., van Ham, F., and van de Wetering, H. “Rendering Hierarchical Data.” *Commun. ACM* 46, 9 (Sep. 2003), 257-263.

[5] Heer, J. and Card, S. K. “DOITrees Revisited: Scalable, Space-constrained Visualization of Hierarchical Data.” *Working Conference on Advanced Visual interfaces (AVI '04)*. (2004) 421-424.

[6] Rao, R. and Card, S. K. “The Table Lens: Merging Graphical and Symbolic Representations in an Interactive Focus + Context Visualization for Tabular Information”. *Human Factors in Computing Systems* (*CHI '94)*. (1994) 318-322.

[7] Spenke, M., Beilken, C., and Berlage, T., “FOCUS: The Interactive Table for Product Comparison and Selection”. *User interface Software and Technology (UIST '96)*. ACM Press, (1996), 41-50

[8] Keim, D. A. 2001. Visual Exploration of Large Data Sets. *Commun. ACM* 44, 8 (Aug. 2001), 38-44.

[9] Olsen, D. R., McNeill, T. G., and Mitchell, D. C. “Workspaces: An Architecture for Editing Collections of Objects”. *Human Factors in Computing Systems (CHI '92)*. (1992) 267-272.

[10] Myers, B. A. “A New Model for Handling Input.” *ACM Trans. Inf. Syst.* 8, 3 (Jul. 1990), 289-320.