# HOTDRAW

BY

JOHN MICHAEL BRANT

B.S., University of Illinois, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

# Table of Contents

**Chapter**

# List of Figures

# Chapter 1

# Introduction

HotDraw is a two-dimensional graphics framework for structured drawing editors that is written in Smalltalk [2]. It can be used in many different applications from a simple painting program shown in Figure 1.1 to a visual drawing inspector shown in Figure 1.2.

A HotDraw application edits drawings that are made up of figures. Figures are graphics elements such as lines, ellipses, and text, and they can represent other objects. For example, the drawing inspector has figures that represent Smalltalk objects and lines that represent the link between an instance variable of an object and the object it references. A finite state machine editor has figures that represent states and transitions between states, and the accounts editor has figures that represent accounts and transactions.

A drawing editor built from HotDraw contains a set of tools that are used to manipulate the drawing. There are tools to create new figures and tools to manipulate existing figures. The standard HotDraw drawing editor shown in Figure 1.3 has many tools. Each tool is represented by an icon shown to the left of the drawing. Only one tool can be active at a time, and this tool is used to manipulate the drawing based on user input from either the keyboard or mouse. The tools contained in the standard drawing editor are (from top to bottom) the selection, bring-to-front, send-to-back, deletion, line, arrow, bezier, spline, rectangle, ellipse, arc, text, image, and constraint tools. Other drawing editors created using HotDraw can have different sets of tools.

When a figure is selected by the selection tool, it presents a set of handles. Manipulating a handle changes some property of its figure or performs some action. For example, in Figure 1.1

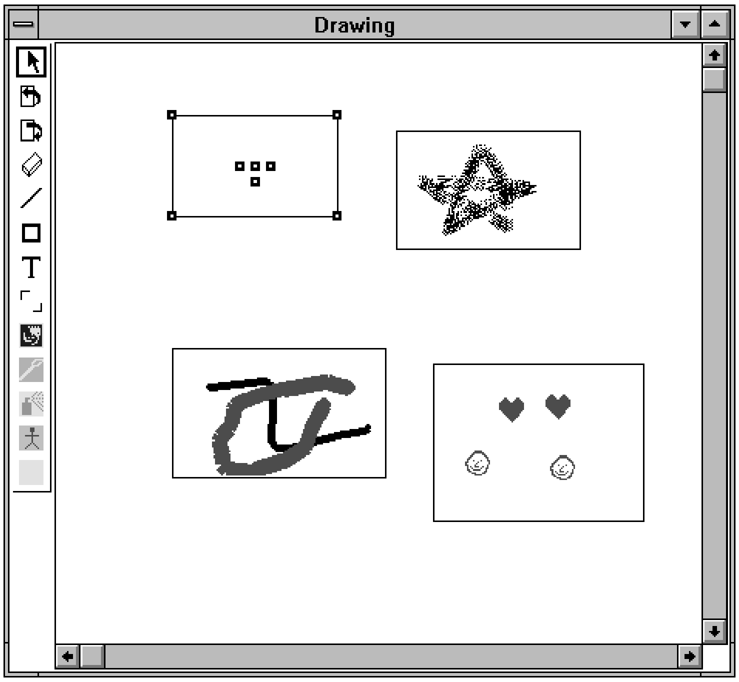


**Figure 1.1**: HotPaint painting program

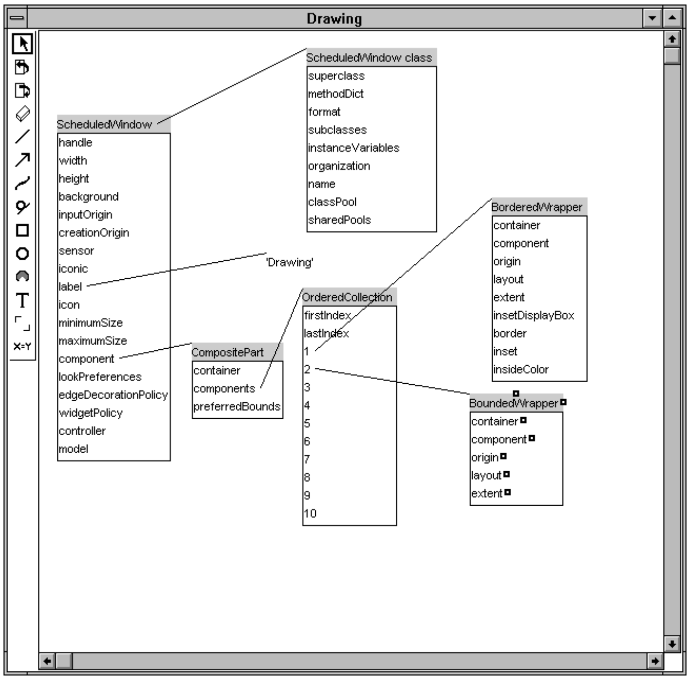


**Figure 1.2**: DrawingInspector inspecting a window object

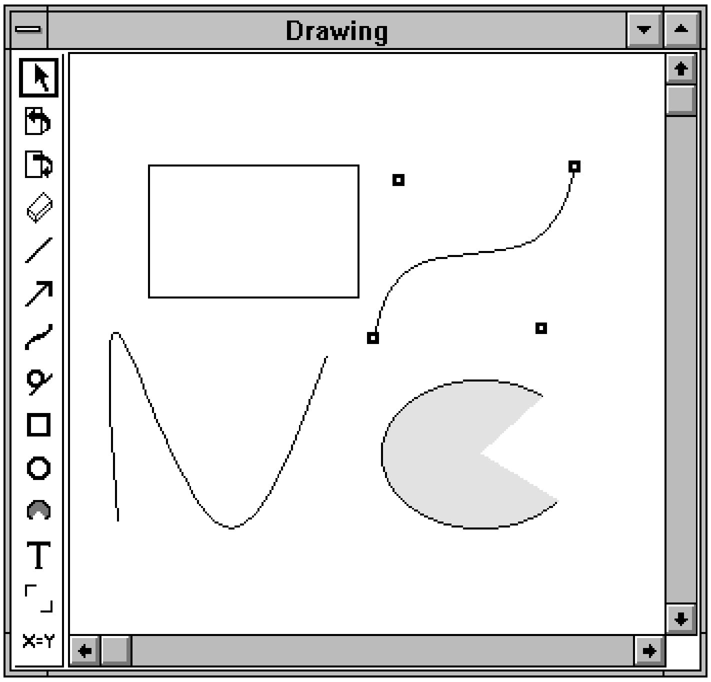


**Figure 1.3**: Standard HotDraw drawing editor

eight handles are displayed on the rectangle figure. The four handles on the external bounding box are used to change the size of the figure. The center handle is used to connect the rectangle to another figure, and the other three are used to change the inside and border colors and the border width of the rectangle. In Figure 1.2, the selected figure has seven handles, each one representing another object. Clicking on a handle draws a line to the figure representing the handle's object, creating the figure if necessary. Although handles are normally rectangular, they can have any shape.

Figures can also have constraints. Constraints can make sure that two figures are connected, are the same size, or have the same color. Constraints can work with objects that are not figures, too. For example, in the PERTChart program the ending date of each PERTEvent is constrained to be the maximum ending date of all of its inputs plus its duration. Constraints are important since they make figures more compositional. Also, they make it easier to define new kinds of figures by composing and placing constraints between simpler existing figures.

This thesis describes the design of HotDraw. It describes various pieces: Drawing, Figure, Tools, Handles, and Constraints and how these pieces fit together. It also describes the design decisions and tradeoffs made for current implementation of HotDraw.

# Chapter 2

# Model-View-Controller

There are two basic functions that all programs perform, input and output, and HotDraw is no exception. It accepts input from the mouse or keyboard and displays it in a window.

The standard design in Objectworks\Smalltalk for input and output is Model-View-Controller (MVC) [3]. MVC has three basic kinds of objects. The model contains data to be displayed, the view displays the model in a window, and the controller accepts user input and changes the model accordingly. HotDraw represents these three objects by the classes DrawingEditor, DrawingView, and DrawingController, respectively. MVC provides benefits to HotDraw and places constraints on it. The main benefit is that views can be composed with other views to make complex user interfaces. Thus, HotDraw can easily be a part of a larger application. A constraint is that a general input/output design such as MVC is not necessarily the best design for HotDraw, but this is also a benefit since users already familiar with the MVC framework can learn HotDraw more easily.

## 2.1 DrawingController

The DrawingController, like most controllers in Smalltalk, handles input from the user. This input can be either mouse or keyboard based. The controller handles input from the operate (middle) or window (right) mouse buttons the same, but it handles input from the select (left) mouse button and the keyboard differently depending on which tool is selected. For example, a click and drag operation results in different actions being performed. If the current tool is the selection tool then the operation should select a set of figures, but if the current tool is the rectangle creation tool then the click and drag should create a new rectangle figure. The DrawingController is responsible for interpreting all actions appropriately in the context of the current tool. This is different from the typical controller that has a well defined method for handling input. A list controller, for example, has only one action that is performed for each mouse operation, it finds the element clicked on and selects that element. As a result the DrawingController is more complex than normal Smalltalk controllers.

Most programs created using HotDraw have different sets of tools. For example, the PERTChart program does not contain the tools that allow you to create new rectangles or lines, it only contains the selection tool, a text editing tool, and a PERTEvent creation tool. On the other hand, the DrawingInspector program contains all of the standard tools that are in the default drawing editor. As a result, the DrawingController should make it easy to add or remove tools from a program.

One way to create a system that allows for different sets of tools is to subclass the DrawingController so that each application has its own specific controller. Each application would add the new actions for its tools to its controller. The controller would then decide at run-time which action to perform. This design makes it difficult to mix new actions from different controllers without multiple-inheritance or cutting and pasting of existing code. It would also require a "switch" statement that checks to see which tool is currently selected and performs that action.

Instead, the DrawingController relies on Tool objects to handle much of the user input. The DrawingController passes any left-button mouse click or keyboard press operation that you are performing to the current tool. The Tool handles these operations in the press and processKeyboard methods, respectively. Since the middle and right-button clicks are handled the same way for all tools, they are handled by the controller. Consequently, the DrawingController is completely general and seldom needs to be subclassed. You can create a new manipulation action, such as creating a new figure, just by creating a new tool that handles that input.

## 2.2 DrawingView

Like other views in Smalltalk, the DrawingView is responsible for displaying its model in a window. There are two things that the DrawingView must do to display the drawing in a

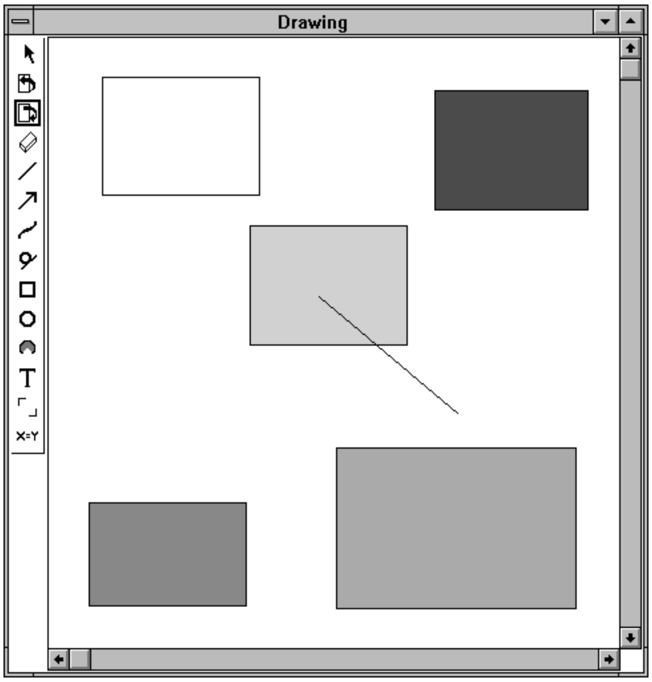


**Figure 2.1**: An ordinary drawing

pleasing manner. The first is that it should be consistent with the drawing. The DrawingView maintains consistency with the drawing by making itself a dependent of the drawing. This is the standard way to maintain consistency between models and views in MVC. When the drawing updates the view, the view damages the area so that it is redrawn in the next display update.

Views also need to make sure that the screen does not "flash" during display updates. Displaying directly on the screen causes the screen to flash. For example, if you have a drawing like that shown in Figure 2.1 then moving the selected line figure causes an intermediate display like that shown in Figure 2.2 where part of the rectangle has been erased. Since the drawing is displayed directly on the screen, the original location of the line is erased (drawn in white), causing part of the rectangle to disappear from the screen. The rectangle is then displayed after the area has been erased. Even though the drawing is eventually displayed properly, this flashing is annoying. Furthermore, since other figures are being redrawn, you might believe that the action of moving the line figure is affecting other parts of the drawing such as the rectangle figure.

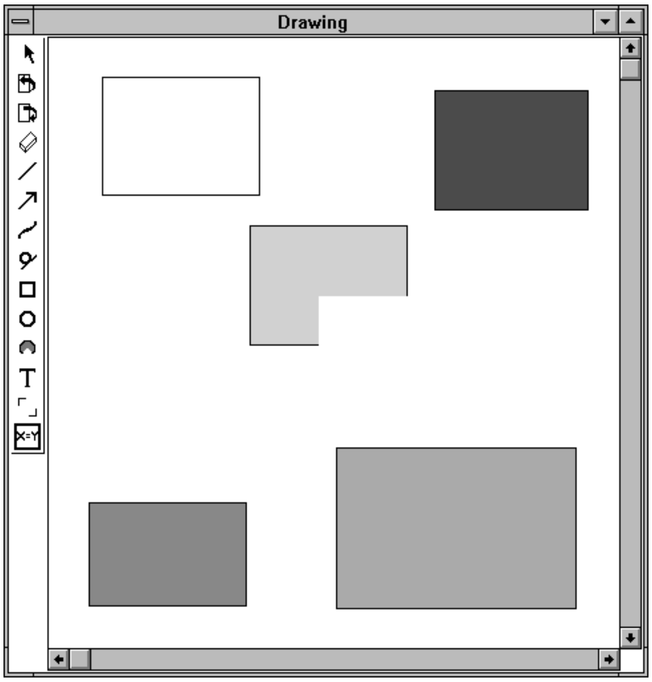


**Figure 2.2**: Drawing without double buffering during display update

The way to eliminate the flashing effect is to use a graphics technique called double buffering. Using this technique, the drawing displays itself on an offscreen pixmap that is then copied to the window. This method eliminates the annoying flashing, since at no time is the user able to see the drawing where not all figures are displayed.

## 2.3 DrawingEditor

The DrawingEditor is the model of the system and as such, has many responsibilities. You can start an application by sending the open message to the drawing editor for that application. The openOn:withLabel: class method creates and opens a window for the application and is responsible for composing the views in the window. If you want to change the look of an application, you must change that method. Since the DrawingEditor opens the window, it must also know the tools for the tool palette view and the type of Drawing to create. These are the responsibility of the class methods defaultTools, which returns the collection of tools, and drawingClass, which returns the class of the Drawing. Programs that have an animated drawing need to redefine the drawingClass method so that the animated drawing is created instead of the normal drawing. As a result, most programs create a subclass of DrawingEditor and redefine some of these methods. In particular, most programs redefine the **defaultTools** method to return the specific tools for that application.

The DrawingEditor is also responsible for the menu that is displayed when the user clicks the operate (middle) mouse button on the background of the drawing. This menu is similar to the **menu** method for figures shown in Chapter 3 and is used for operations on the drawing, such as loading and saving a drawing.

Instead of the DrawingEditor containing the figures in the drawing, it relies on a Drawing to contain them. The Drawing also has the methods to add and remove the figures as well as animate the figures. Since HotDraw is an interactive program, figures change appearance (position, color, etc.). If a figure changes its appearance then the Drawing must notify its dependents (normally DrawingViews) so that they can stay consistent with the Drawing. This technique allows many views to display the same Drawing.

# Chapter 3

# Figures

Figures are the main elements in every drawing. Every program that uses HotDraw will use some type of figure. Figures can represent a variety of different things from a transaction in an accounting system to a Smalltalk object that is being inspected, or they can be used just to make a nice picture that has no other meaning, as in the HotPaint program. If you manipulate a figure, you can implicitly change the underlying structure of the problem. For example, if you connect the quantity field of a figure representing a sales transaction to the quantity-on-hand field of a figure representing the inventory journal in an accounting program, you can specify that the quantity of every sales transaction is posted to the quantity-on-hand of the inventory journal, or if you change the duration of a PERTEvent in the PERTChart program, you automatically change the ending date of that event and all events that depend on the its ending date.

## 3.1 Basic properties

Figures are visual elements in the drawing and as such must be able to display themselves. To do so, they must implement the two methods displayOn: and displayShapeOn:, which draw the figure onto a graphics context. Figure 3.1 shows the displayOn: method for an ellipse figure. The code first draws a filled ellipse if the ellipse figure is not transparent and then draws the border of the ellipse. The displayShapeOn: method is similar to the displayOn:, but it is designed to use a mask's graphics context instead of a window's graphics context, therefore, it does not set the paint color, instead it just uses graphics primitives to record where the figure is opaque. More

**displayOn: aGraphicsContext**

*"Display an ellipse on aGraphicsContext. If the ellipse figure is not transparent then display a filled ellipse using the current fillColor."*

```
transparent value
    ifFalse:
        [aGraphicsContext paint: (self colorAtIndex: fillColor value).
        aGraphicsContext
            displayWedgeBoundedBy: self displayBox
            startAngle: self startAngle
            sweepAngle: self sweepAngle].
width value > 0
    ifTrue:
        [aGraphicsContext lineWidth: width value.
        aGraphicsContext paint: (self colorAtIndex: color value).
        aGraphicsContext
            displayArcBoundedBy: self displayBox
            startAngle: self startAngle
            sweepAngle: self sweepAngle]
```

**Figure 3.1**: EllipseFigure displayOn: method

information on using graphics contexts to construct a displayOn: or displayShapeOn: method can be found in Chapter 19 of [4].

In order for the drawing to display the figures, it needs to know their location. It then redraws only those figures that are in the region to be redisplayed. As a result, figures must define methods that compute their location. Instead of having each figure redefine one method that returns its location, each figure must have a method that returns its origin and one that returns its extent. These methods are used by the displayBox method to compute the bounds of the figure and by methods that specify default behavior of picking operations such as the figureAt: method.

Figures can also define a handles method and a menu method. The handles method returns a set of handles to be displayed when the figure is selected. Each figure can have a different set of handles and can even change its handles between selections. You can manipulate the handles to perform special action on the figure or drawing. These actions can vary widely from changing the size or color of a figure to connecting two figures with a line figure. The menu method returns a PopUpMenu for the figure. Menus are another way for you to manipulate a figure. They are displayed when you select a figure with the operate (middle) mouse button, and they

```
menu
    ^PopUpMenu
        labels: 'Line Width\Line Color\Fill Color\other' withCRs
        values: (Array
            with: (PopUpMenu
                labels: '1 pixel\2 pixels\3 pixels\4 pixels\5 pixels' withCRs
                values: #(#border1 #border2 #border3 #border4 #border5 ))
            with: (PopUpMenu
                labelArray: ColorValue constantNames
                values: (ColorValue constantNames collect: [:aName |
                    ('l' , aName) asSymbol]))
            with: (PopUpMenu
                labelArray: ColorValue constantNames
                values: (ColorValue constantNames collect: [:aName |
                    ('f' , aName) asSymbol]))
            with: super menu)
```

**Figure 3.2**: RectangleFigure's menu method

display short lists of items from which you can choose. The menu is displayed no matter what tool is being used. Menu selections except for cut, copy, and paste are passed to the figure for processing. In addition to the standard edit operations such as cut, copy, and paste, figures often put border and fill color information in menus. Figure 3.2 shows a RectangleFigure's menu. This menu can be used to change the rectangle's border and fill colors, to change the border width, and to perform normal edit operations that are contained in the RectangleFigure superclass' menu. As an example, if you select "2 pixels" under the "Line Width" submenu, the border2 message is sent to the rectangle. More information on constructing PopUpMenus can be found in [4].

Since HotDraw is an interactive environment, most figures change over the life of the drawing. When a figure changes, the drawing needs to redraw both its old area and its new area. The old area must be redrawn to erase the figure from its old location. Redrawing only one of these areas leaves the drawing with either two copies of the figure, both the old and new copy, or no copy. For these reasons, the figure must notify its container, a Drawing, before and after it changes. This notification is done by the figure sending a willChange message to itself before it changes and a changed message after. Chapter 6 shows how these messages can be eliminated by using value holders, but figures that perform changes without using value holders still must send these messages. When these two messages are sent, the container is notified

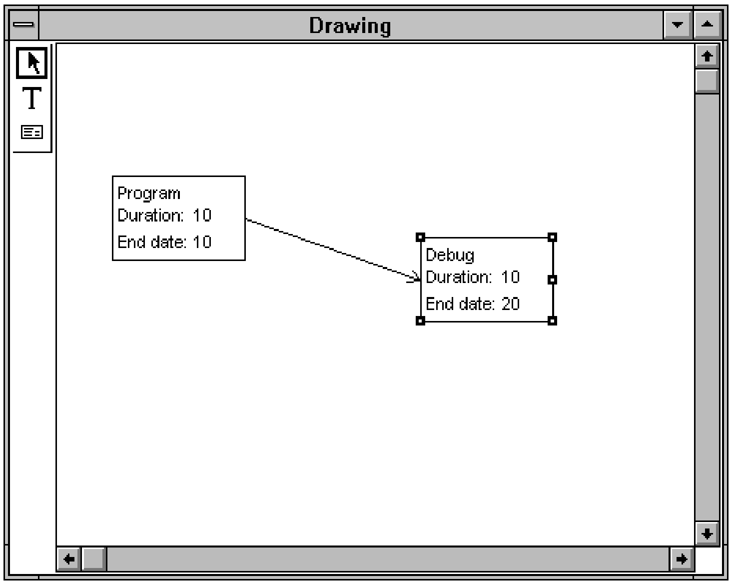


**Figure 3.3**: PERTChart program with PERTEvent figure selected

of the change and takes appropriate action. Normally, Drawings forwards this information to their dependents, which are DrawingViews. The DrawingView then damages the area so that it is redrawn in the next display update. If you have a DrawingView that is not redisplaying properly after a change, most likely you have forgotten to send one or both of these messages.

## 3.2 Composite figures

HotDraw contains many types of figures. Most graphics object in Smalltalk are already figures in HotDraw. These include lines, bezier curves, splines, rectangles, ellipses, arcs, and text. Many times, though, an application represents an external object with a group of figures. For example, the PERTChart program shown in Figure 3.3 has two PERTEvents, each of which consists of five text figures plus a rectangle bounding box. Other objects have different representations that use other figures.

One way to build such a representation is to add all of the figures that make up the representation to the drawing separately. A PERTEvent would add the five text figures and the rectangle bounding box to the drawing. Although this gives a correct initial picture, it causes problems when you start to manipulate the drawing. For example, if you try to move one of the figures, you expect the others to move too, but instead only one of the figures moves. Furthermore, if you delete one of the figures, all figures corresponding to that PERTEvent should also be removed. You need to be able to manipulate all figures as a single figure.

Another way is to make a new figure that displays all of the component figures for the new figure. This new figure is called a CompositeFigure. A CompositeFigure is like a normal figure such as a line figure; it can display itself, knows its origin and corner, etc. Unlike other figures, it is made from a collection of other HotDraw figures. The CompositeFigure uses its collection of figures to perform the required tasks of figures, such as displaying itself. A CompositeFigure displays itself by iterating over its components, displaying each of them. A PERTEvent is represented by a single composite figure that has a collection of text figures. These text figures represent the name of the event, the duration, and the ending date. Since a composite figure is manipulated as a whole, moving one element of the composite figure moves all of its figures. Also, the creation and deletion code can be encapsulated into the CompositeFigure (or subclass) instead of the object that creates it.

## 3.3 Metafigures

Often a figure represents a non-HotDraw object. This is the case in the DrawingInspector application, where any object in the Smalltalk system is represented by a figure, and this figure must know what object it is representing. Other times new attributes need to be added to figures. ObjectWorld adds an animation attribute to every figure. This attribute is able to move the figure as directed by the user. Another place where new attributes are needed is for lines that connect two figures. For example, many applications have figures that contain references to other figures they are connected to. Deleting a line should also delete the reference, but a line without additional attributes does not know what it is connected to. If a line contains an attribute that has the figures that it is connected to then it can notify those figures to delete the references when it is deleted.

In all of these cases, figures need more information than the normal display information. One way to add this information is to create a subclass of the figure and add the extra information by adding methods or instance variables to the subclass. Although this technique works as intended and is sometimes useful, it has some undesirable properties. One of these properties is that we must create a new subclass for every new representation that we want. In ObjectWorld this results in creating a new subclass of every figure already in HotDraw and then adding the animation attributes to each of them. If you want different animation attributes for figures, you must create a subclass of each figure for each animation attribute. This results in many new classes being created. Not only are these classes difficult to maintain because of the number of classes, they are also difficult to maintain since much of the code between figures with the same animation attributes is duplicated. Still another undesirable property of this technique is that it is difficult to support new figures. For example, if you have a new figure and want to add it to the figures available in ObjectWorld, you must create a subclass and add the animation attributes to that subclass. This requires you to have a good understanding of how the animation attributes work.

In order to limit some of this subclassing of figures, metafigures were introduced. Metafigures are instance variables of figures that can hold various different things from additional attributes for the figure to the object it is representing. Metafigures can also hold the menu for their figures. The metafigure can redefine the menu of its figure, without needing to subclass the figure. All the metafigure must do is define the `menu` method that returns an alternate menu for the figure. In the ObjectWorld program each figure in the drawing has a metafigure that holds its animation attribute, and these attributes are used to animate their figure. There is only one class for each animation attribute that works for all types of figures, even new unknown figures that are added later. As a result, ObjectWorld is able to use the standard HotDraw figures without modification. Furthermore, since these attributes are components of figures, they can be changed at run time whereas, with inheritance, the properties could not easily be changed.

# Chapter 4

# Drawing

The Drawing is the center of HotDraw. Most actions in HotDraw result in some message being sent to the Drawing. It is responsible for keeping track of the figures for selection and for animation. The Drawing can add or remove figures, select a figure by displaying its handles, and animate figures by explicitly moving them. As a result, the Drawing is one of the most important classes in HotDraw.

## 4.1 Layered composite figure

A Drawing is like a composite figure except that it contains the selection and can animate its figures. If the Drawing were a subclass of composite figure then the Drawing could reuse most of the behavior of composite figure. This subclass would contain the selection information and the code to add, remove, and display figures would be inherited from composite figure.

Although the composite figure representation for the Drawing is more than adequate for most uses of HotDraw, occasionally one needs something somewhat more complicated. For example, if a Drawing stores all figures in one composite then it is difficult to make a set of figures invisible. Several application use different techniques to achieve this. The original version of the Hypercard clone used separate collections to store each card. Whenever a card was to be displayed, it would first remove all figures for the old card from the drawing and then add the new card's figures to the drawing, thus achieving invisible figures by removing those figures that are not to be displayed. In another application, certain types of lines needed to be visible or invisible depending on what the user wanted to see. In this application a visible

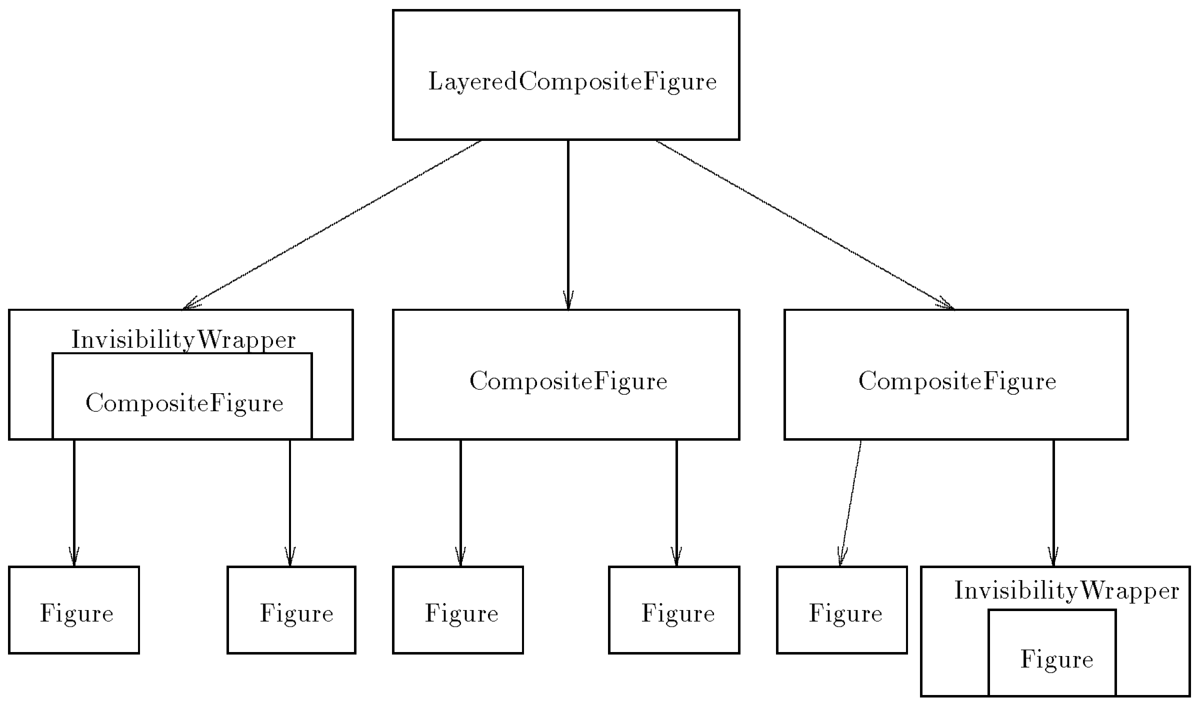


**Figure 4.1**: Structure of a layered composite figure

attribute was added to all figures to determine if they were supposed to be displayed or not. In any case, making figures invisible seems to be a necessity for some programs.

One way to implement invisibility is to make a special InvisibilityWrapper that is placed around a figure to be made invisible. It then intercepts any message from the drawing that performs display operations, such as `displayOn:`, so that the figure is not displayed. All other messages are forwarded onto the original figure. This approach does not need extra space in every figure for an invisibility attribute as the approach with the lines above, and it also does not have the overhead of adding and removing a figure when it becomes visible or invisible as the Hypercard clone has.

To make sets of figures invisible, we use a layered approach where the drawing contains several layers of figures. Each layer is then made invisible by wrapping it in the Invisibility-Wrapper. Each layer contains a composite figure that is used to store the set of figures in that layer. The Drawing is then a layered composite figure instead of just a composite figure. Figure 4.1 shows the overall structure of a layered composite figure. In this figure, the layered figure contains several composite figures, each of them contains several normal HotDraw figures.

Both the first composite figure and the last figure in the third composite figure are not being displayed since they are wrapped in InvisibilityWrappers.

Using a layered figure, the Drawing can add figures to a certain layer of the drawing. You can use the normal add: and addAll: methods to add figures to a default layer or can use the add:in: and addAll:in: methods to add figures to a particular layer. Once figures have been added to the layers, layers can be made invisible by the message hideLayer: or visible by showLayer:, or figures can be made invisible by the message hide: and visible by show:. These methods wrap or unwrap figures in the InvisibilityWrapper as describe above. Figures that are hidden, not only are not displayed, they are never selected by either the figureAt: or figuresIn: methods which are normally used to select the figures which were clicked on.

## 4.2 Handle layer

Handles are special graphic objects that are displayed when a figure is selected. They can perform all of the required operations of figures, and as a result, they are implemented as a subclass of figure. Using this implementation handles can be added and removed from the drawing like normal figures.

Although handles can be added and removed as normal figures, there are a couple of problems with that implementation. One problem is that handles should be displayed on top of all other figures in the drawing. If we add handles as normal figures then we must take precautions that normal figures are added behind the handles. Another, more important problem is that since operations on handles are performed so frequently, having the handles stored with normal figures can lead to very poor performance.

Instead, handles are stored in a separate layer of the drawing. This layer is stored in front of all other layers so we do not need to perform any checking when new figures are added. Furthermore, since handles are the only figures stored in this layer, we can make special optimizations. For example, we can remove all the handles from the drawing by simply replacing the handle layer with a new empty layer, or we can make all handles invisible by making just the handle layer invisible. If handles were just added to the drawing, both of these cases would have needed to iterate though all of the figures, but with this implementation there is no need to iterate though the figures.

```
step
    "First, calculate the new velocities of each figure by calculating the gravitational
    force that each has on the others."

    self figures do: [:fig1 | fig1 force: (self figures inject: 0 @ 0 into: [:sum :fig2 |
        sum + (fig2 forceOn: fig1)])].
    self figures do: [:fig1 | "Last, move each figure."
        fig1 step].
    self solveConstraints
```

**Figure 4.2**: MovingDrawing's step method

## 4.3 Animation

One of the things that makes HotDraw different from many drawing editors is that it can animate drawings. For example, the NetworkEditor program uses animation to iteratively solve the n-body problem. It lets you specify nodes and weights between two nodes. The Drawing iteratively computes the figure's position. Each iteration computes a repulsion force on each node and also an attraction force that is calculated from the weights between nodes. Once these forces are computed, the drawing moves the figures by the force specified for that figure. The drawing also allows you to move the nodes so that if they have reached a stable, but unoptimal positioning, you can move them to achieve a better positioning.

To have an animated drawing, make a subclass of Drawing that redefines the step method. The step message is sent by the controller when no events are being processed. An example of the step method for the MovingDrawing program is shown in Figure 4.2. The MovingDrawing is a simple application in which every figure that is attracted to the others, coming together in some central location. The step method first calculates the forces that are acting on each figure, and then moves each figure by the force acting on it.

# Chapter 5

# Tools

Tools are responsible for handling much of the user input. All keyboard and left mouse button input are handled by the current tool using its processKeyboard and press methods. These methods translate the user input into actions in the drawing such as creating a new figure or moving figures.

## 5.1 Single class approach

One way to implement tools is to write different processKeyboard and press methods for each new tool. A new tool would be created by subclassing an existing tool and then implementing the new behavior. As we will see, this way leads to duplicate code and a system that is made of monolithic components that are hard to understand or reuse. A better way is to divide the tool into a set of components. First, we will look closely at the result of making tools be a single object.

Figure 5.1 is an example of a tool created using the single class approach. It shows the press method for a scrolling tool. The scrolling tool moves all of the figures in the drawing when the user performs a click and drag operation. There are three sections to this example. The first part performs some initialization by getting the current mouse position and changing the cursor. The next section loops reading the current mouse position and moves the figures to their new position. When the user releases the mouse button, the final section is executed. For this tool, it just sets the cursor back to its previous display. This is a typical press method in

```
press
    "Make my drawing follow the cursorPoint."

    | oldPoint newPoint |
    oldPoint := self sensor cursorPoint.
    self class closedHandCursor beCursor.
    [self sensor redButtonPressed]
        whileTrue:
            [newPoint := self sensor cursorPoint.
            newPoint ~= oldPoint
                ifTrue:
                    [self view scrollBy: newPoint - oldPoint.
                    oldPoint := newPoint]].
    self cursor beCursor
```

**Figure 5.1**: ScrollingTool's press method

```
invoke: aView
    | aCollection anObject sensor newPoint oldPoint |
    aCollection := aView selections.
    sensor := aView controller sensor.
    oldPoint := sensor cursorPoint.
    aView hideHandlesWhile: [[sensor redButtonPressed]
            whileTrue:
                [newPoint := sensor cursorPoint.
                anObject := self sense: newPoint - oldPoint.
                anObject notNil ifTrue: [aCollection do: [:each |
                    self change: each by: anObject]].
                oldPoint := newPoint.
                aView repairDamage]]
```

**Figure 5.2**: SelectionTrackHandle's invoke: method

that it performs some initialization, it then loops reading the mouse position and updating the drawing until the mouse button is released, and then it performs some finalization action.

Although this implementation works fine for tools in which all figures were manipulated using the same techniques, it does not work well for tools that manipulate figures differently. For example, handles are manipulated by the selection tool. If the handle manipulation action is performed by the tool then every type of handle would need a corresponding action in the tool's press method. These different actions would then be chosen by a "switch" statement. Instead, if these different actions are moved into the handles then the press method can be simplified by using polymorphism of the handles. Using this technique, all handles implement an invoke:

```
createNotifying: aView
    | startPoint stopPoint aSensor aLineFigure gc |
    aSensor := aView controller sensor.
    startPoint := aSensor cursorPoint.
    stopPoint := aSensor cursorPoint.
    gc := aView graphicsContext.
    gc displayLineFrom: startPoint to: stopPoint.
    [aSensor redButtonPressed]
        whileTrue:
            [aView drawing damageRegion: ((Rectangle vertex: startPoint vertex: stopPoint)
                    expandedBy: 2 @ 2).
            aView repairDamage.
            stopPoint := aSensor cursorPoint.
            gc displayLineFrom: startPoint to: stopPoint].
    startPoint = stopPoint
        ifFalse:
            [aLineFigure := self start: startPoint stop: stopPoint.
            ^aLineFigure].
    ^nil
```

**Figure 5.3:** LineFigure's createNotifying: method

method that performs the action specific to the handle. Figure 5.2 shows the invoke: method for a track handle. As with the other example shown above, it also performs an initialization step, loops reading the sensor values, and then performs a finalization step that in this case does nothing.

Figure creation tools also needed special actions in the press method. If the press method created the figure directly then there would need to be a creation tool for each type of figure. Furthermore, each creation tool would need to know some of the representation of the type of figure it creates. For example, a rectangle creation tool would need to know that rectangle figures are defined by two points, and it would also need to display a rectangle during creation. Instead, if each figure class implements a createNotifying: method then there can be one parameterized creation tool that creates every type of figure. This creation tool would take the class of the figure to create, and when used would then send the createNotifying: message to that class. An example createNotifying: method is shown in Figure 5.3. This figure shows the createNotifying: method for the line figure. Once again we can notice the same three phases of the code, the initialization, the looping, and the finalization.

Although this implementation is simple, it has many problems. One problem is duplicate code. As seen in the three examples shown above, the code that reads the sensor is present in all examples, and in fact, some form of a sensor reading loop occurs 11 times in a previous version of HotDraw with its sample applications. This makes it hard to maintain the code. For example, if the sensor protocol changes in the next release of Smalltalk then one would need to find and change all of the sensor code that is distributed across many classes, tools, handles, and figure classes. Furthermore, in order to write a new tool, you must know about the sensor protocol, adding more complexity to writing new actions.

Another problem with this implementation is that although distributing the code over multiple classes makes the system more reusable, it also makes it more difficult to learn and debug. It is hard to tell which piece of code was being executed for a given input action. For example, suppose you move a rectangle around the screen using the selection tool. Instead of this action being performed by the selection tool, it is performed by an invisible track handle that is created specially for that purpose. Similar designs were also used in some figure creation methods which used invisible handles to specify the figure size. These designs were created to make more reusable actions, but as a result made learning and debugging the system more difficult since often you would need to look at many classes to find the code that was being executed.

Still another problem with this system for tools is that new tools cannot be created by combining actions from existing tools. For example, suppose you want a new tool that creates a new rectangle figure when clicked on the background of the drawing, but manipulates the figures like the selection tool when clicked on an existing figure. You would subclass one of the tools and then copy the code from the other tool into the new tool. For this example, you would probably subclass the selection tool and copy the creation code from the creation tool.

## 5.2 Multi-class tools

Another approach for building tools is to split its responsibilities into several classes. We split it into these three classes: Reader, Command, and Tool. These classes are similar to manipulators, commands, and tools in Unidraw [6] from which the HotDraw solution was derived. A Reader is an object that reads the sensor values for an event and translates them into some form understood by Commands. Commands then act upon those translated values by changing the

| *SelectionTool's command table* | **DragReader** | **ClickReader** |
|---|---|---|
| **Drawing** | MultiSelectionCommand | |
| **CommandHandle** | ConstraintCommand | |
| **ConstraintHandle** | FigureCommand | |
| **Figure** | Selection & Constraint Command | Selection Command |
| ...all other subclass of Figure | | |

**Figure 5.4**: Mouse command lookup table for the SelectionTool

drawing. Finally, the Tool contains an icon, cursors and a table that specifies how Commands relate to Readers and figures.

Under this model, the Tool receives processKeyboard and press messages as before, but upon receiving them, the Tool creates a Reader for the user action and also checks to see what figure the user is manipulating. Once both the figure is known and the Reader is created, the Tool looks up and creates the appropriate Command from the Tool's table. After the Command is created, the tool initializes the Reader with the message initialize:command: which then initializes the Command. Once the Reader and Command are initialized, the Tool loops sending the readSensor: message to the Reader. The Reader then updates the command with the sensor information by sending the moveTo: message, giving it a chance to change the drawing. If the Reader is done reading input then the readSensor: message returns false. At that point, both the Reader and Command are sent the effect message, which gives the Reader and Command a chance to perform any finalization action that is needed.

Command lookup is performed based on the Reader and the figure being manipulated. Each tool contains a different table that describes what command to create. Figure 5.4 shows the command lookup table for mouse commands using the selection tool. The figure class is looked up in the table first. If no match is found then the search continues up the figure's inheritance hierarchy until a match is found. After the row in the table is found, the reader's class is found, resulting in a command. This command is then cloned and used to manipulate the drawing.

The command protocol is dependent on the type of reader. A command for a ClickReader must implement both initialize: and effect methods, and a command for a DragReader also implements a moveTo: method in addition to the initialize: and effect methods. These methods are already implemented by the MouseCommand which by default does nothing to change the

```
initialize: aPoint
    lastPoint := aPoint.
    figure paintAt: lastPoint brushWidth: figure width ** 2

moveTo: aPoint
    figure
        paintFrom: lastPoint
        to: aPoint
        brushWidth: figure width ** 2.
    lastPoint := aPoint

effect
    figure paintAt: lastPoint brushWidth: figure width ** 2
```

**Figure 5.5**: HotPaint's PaintbrushCommand for a DragReader

drawing. Figure 5.5 shows the PaintbrushCommand for a DragReader. The initialize: method takes the initial mouse point where the button was pressed and draws a point on the figure at that location. The moveTo: method draws a line from the last known mouse location to the current location (given as an argument to the moveTo: method). Finally, the effect method simply draws a point at the last known mouse location.

A command for a TextReader must implement the initialize, keyboardEvent:, and effect methods. The initialize and effect methods initialize and finalize actions for the command. These are similar to the initialize: and effect methods of MouseCommands. The keyboardEvent: method takes a keyboard event and then performs some action. The TextCommand is the default command for a TextReader. Like the MouseCommand, this command does not change the drawing.

Commands are also constructed from combining existing commands. The CompositeCommand is used to combine a set of mouse commands, and the CompositeTextCommand is used for text commands. The selection tool uses a CompositeCommand that selects the figure (SelectionCommand) and moves the figure (ConstraintCommand). While it is possible to combine any commands, you should only combine those commands that work together. For example, you would not want a CompositeCommand that would delete the figure (DrawingCommand) and move the figure since the move operation is not valid for deleted figures.

The main advantage of this design is that it results in more reusable code. The 11 instances of a mouse reading control loop are reduced to just one instance located in the DragReader. Furthermore, commands can be reused across different tools. For example, if you want a tool

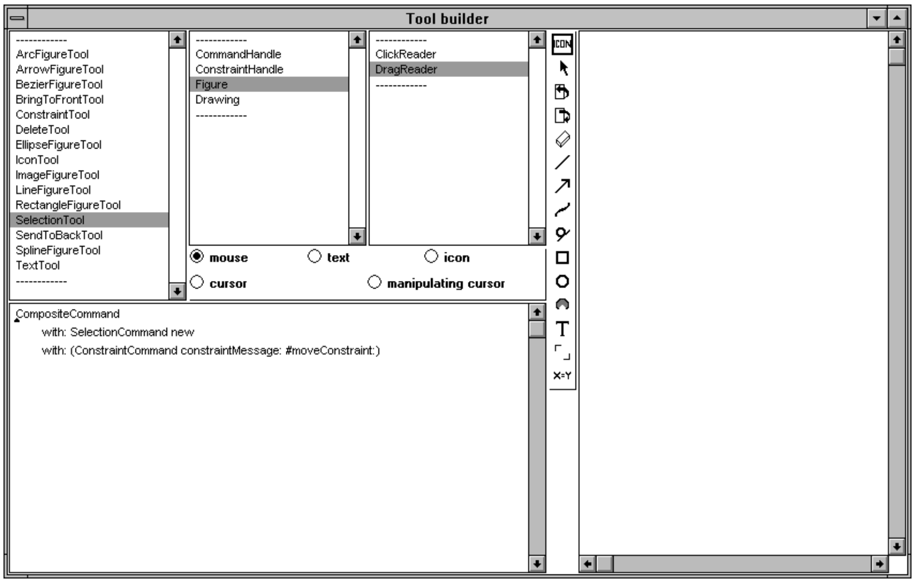


**Figure 5.6**: ToolBuilder editing the SelectionTool

like in the example above that creates a rectangle figure when clicked on the background and works as a selection tool when clicked on a figure, then you just need to create a table like the selection tool's table except that the slot for the Drawing (background) would contain a command to create the rectangle figure. Moreover, it is possible to make a program that constructs a tool's table, and thus, it is possible to define a new tool without any programming.

The ToolBuilder is a browser like tool that can be used to specify new tools or edit old ones. Figure 5.6 shows a ToolBuilder that is being used to edit the SelectionTool for the Figure class and a DragReader. The right part of the window is a HotDraw application that you can use to visually edit cursors and icons for the tools. The left half of the window has five parts. The top left pane lists the tools in the system. You can add or remove tools from this pane. The top middle pane lists the sets of figure classes that the selected tool acts on, and to its right is the set of readers for each figure class. The middle of the browser contains five buttons that you can use to switch between editing the two cursors, the icon, or the two different input modes (keyboard or mouse). The bottom pane contains the code to create a command for the specified tool, figure, and reader. This code must evaluate to a command that matches the selected reader. Some applications may require that no action is performed when you manipulate a figure. For these applications you can use a MouseCommand (or TextCommand). The browser lets you specify new tools by creating an icon, cursor, a manipulating cursor (the cursor that is displayed while the user is manipulating the drawing), the keyboard input lookup table, and the mouse input lookup table. Almost all tools can be specified in this way, with the only exception being the TextTool.

Another advantage of this design is that special actions for special figures can easily be specified by filling out the table. Handles are no longer a special case for tools. They are controlled by placing a special entry in the table for handles as can be seen in the SelectionTool's table in Figure 5.4. The HotPaintEditor is another example that needs special actions for special figures. For example, the HotPaintPaintbrushTool only paints designs on CanvasFigures. A previous version added special lines of code to the tool's `press` method that checked the type of figure being manipulated. This extra code added nothing to the overall action that was being performed, it only made the actions more difficult to understand. But the current version checks the figure's class before the command is created so that the command does not need to perform any checking itself. This makes the code for the command cleaner since it can focus directly on the code that changes the figure and not worry about the extra code that checks the types of figures.

One final advantage for this implementation is that debugging code is made simpler. The programmer can find out quickly what piece of code is not functioning properly by examining the command table. Since commands are the only objects that are directly responsible for performing changes to the drawing, there is no need to look in the other classes such as tools or handles to see how an action is performed.

# Chapter 6

# Constraints

Constraints are an essential part of drawing editors. They have been used in many editors including Sketchpad developed by Ivan Sutherland in 1963 [5]. Although they are not as obvious as other parts of the drawing such as figures or tools, constraints are just as important. For example, when you connect two figures with a line, you constrain both end points of the line to stay connected to the figures. Without constraints you would need to manually move the line when you moved one of the figures, but a constraint makes the line follow the other figures. Constraints can also be used for non-geometric information. The PERTEvent figure uses constraints for its ending date. The constraint computes the maximum starting date of its inputs and adds it to the PERTEvent's duration. Tools that move figures also use constraints. The SelectionTool places a constraint between the mouse position and a figure so that the figure moves with the mouse. Since constraints are used in many places, they are an important part of HotDraw.

## 6.1 Distributed constraints

There are two basic ways to implement constraints. This section describes distributed constraints like those in the older version of HotDraw, and the next section describes a more centralized constraint system that is used in the current version of HotDraw.

The distributed constraints of the old HotDraw were implemented using standard Smalltalk dependency mechanisms. Under these mechanisms an object A registers itself as a dependent of another object B by sending the `addDependent:` message to B. After A is registered as

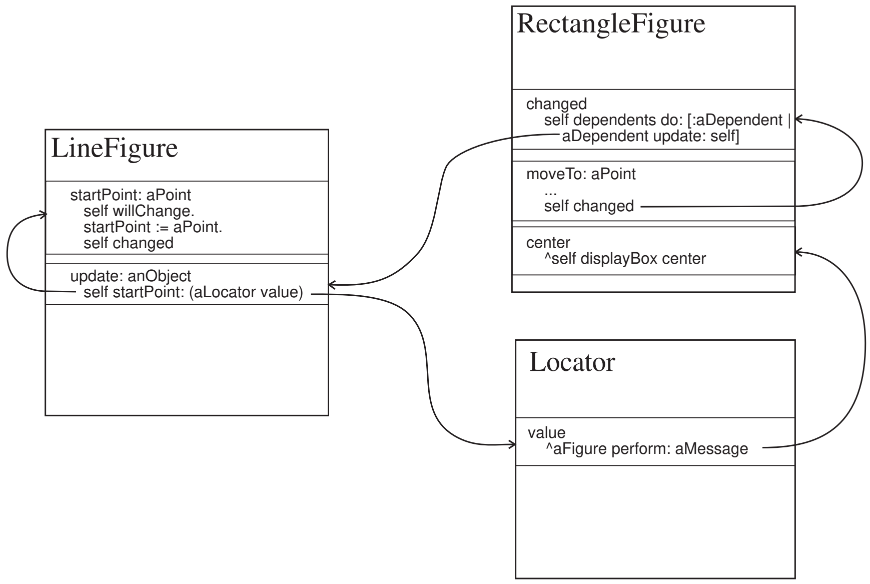


**Figure 6.1**: Constraint between a line and a rectangle

a dependent of B, whenever B changes, it sends an update:with:from: message to all of its dependents including A. This dependency system makes it easy to implement distributed local propagation constraints. These constraints are simplified somewhat in HotDraw. If a HotDraw figure changes, instead of sending the update:with:from: message to its dependents, it sends the changed message to itself, and the changed method then notifies all of the dependents with the update: message.

Constraints in the first version of HotDraw were implemented by having the constrained figure register itself as a dependent of the constraining figure. The constrained figure would implement the constraint inside its update: method. The constrained figures sometimes relied on special objects called Locators so that the update: method could be more general. Locators were parameterized with a message to send to the constraining figure to get a point. For example, if you want the start point of a line figure to be constrained to the center of a rectangle figure, you would use a Locator that returns the center of the rectangle figure. This example is shown in Figure 6.1. When the rectangle changes its position, it sends the changed message to itself. This results in the update: message being sent to the line figure. The line figure calculates its new start position by asking the Locator for its value, and then sets its start position to that value.

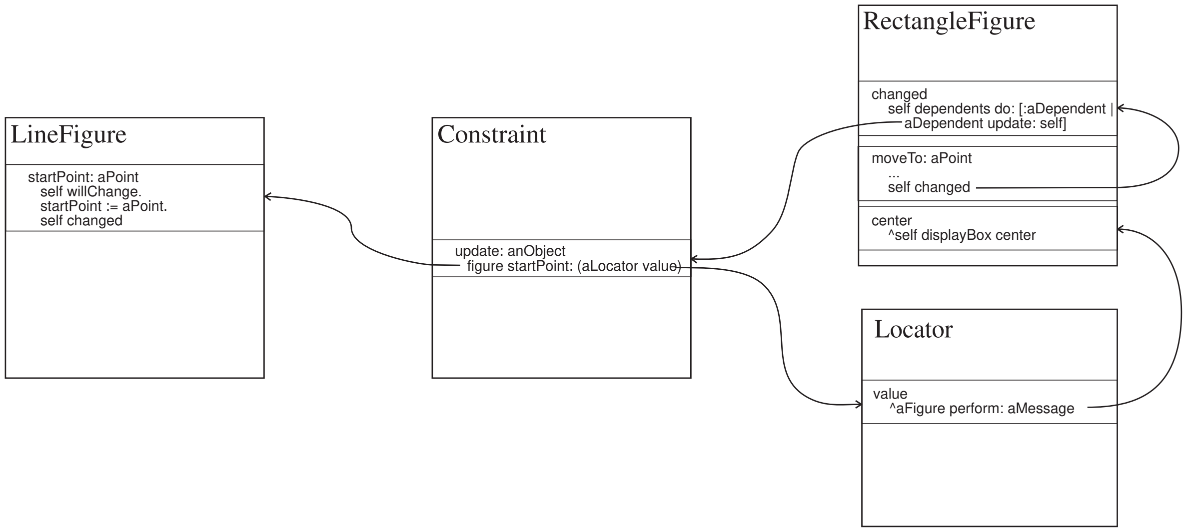


**Figure 6.2**: Constraint between a line and a rectangle using a Constraint object

Although these constraints can constrain figures to certain points of other figures, they have one significant problem. This problem is that for every figure that needs to be constrained, there needs to be a subclass of an existing figure to implement the constraint in the `update:` method. As a result, Constraint objects were introduced. Constraint objects pulled out the constraint part in the figure's `update:` method into the `update:` method of the Constraint. If a constraint is needed between two figures, a Constraint object is created and registered as a dependent of the constraining figure. Upon receiving an `update:` message from the constraining figure, the Constraint evaluates the constraint and changes the constrained figure to satisfy the constraint. Like before, Locators are used to make the Constraints more general. Figure 6.2 shows the constraint example seen before, only using Constraint objects. In this figure the code from the `update:` method of the line is now in the Constraint object. This method makes constraints more general since they can be used across different types of figures without needing to subclass a figure to make a constraint.

Even though this decentralized approach for constraints is easy to implement, it has some problems. One problem is that it is hard to implement two-way constraints or a circular set of constraints. For example, if we have a constraint on figure B that sets some value of figure A and a constraint on figure A that sets some value of figure B, we would need to place checks in the constraint code so that it does not get into infinite cycle of `changed/update:` messages

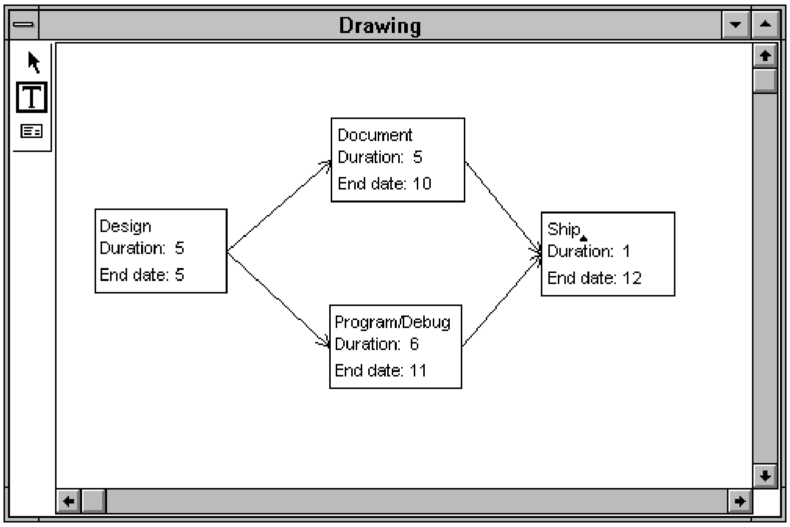


**Figure 6.3**: Local propagation constraints

where A updates B which updates A, and so on. The basic constraints in the earlier version of HotDraw did not have this check for infinite updating, and as a result there were special immobile line figures that were used to connect figures. If you were allowed to move these lines without two-way constraints then the line would become disconnected from the other figures, since you could not constrain the other figure to the line while constraining the line to the other figure.

Another problem with these constraints is that they were difficult to create interactively. The ability to create interactive constraints is needed so that users can create new figures by composing existing figures and adding constraints. Users should not always need to write code to create a new figure. In order to specify constraints interactively, there needs to be some idea of what constraints can be created, and for these constraints there has to be a method that can set the value computed by the constraint and other methods that the constraint can use to read values to help compute the constraint. The original HotDraw had only a few such methods, and these were mostly for the bounding box of the figure.

Other problems arise as a result of the constraint system being distributed. One problem is that it is difficult when copying figures to copy only the connected constraints. If you copy a set of figures, the constraints between the figures should also be copied, but you should not copy constraints that refer to figures that are not copied. The distributed approach to constraints made it difficult to tell which constraints were attached to figures that were being copied. As a result, constraints were not copied in the original version of HotDraw.

Still another problem with these constraints is that solving constraints can be inefficient. For example, the PERTChart shown in Figure 6.3 has four PERTEvents that are connected. Each one of these connection represents a constraint. If you change the "Design" PERTEvent's duration, the drawing should update the other three PERTEvent's ending dates. This update is performed by solving constraints. These local propagation constraints have the "Ship" PERTEvent updated twice, once after the "Document" PERTEvent is updated, and again after the "Program/Debug" PERTEvent is updated. The best case would only need to update "Ship" once, only after both the "Document" and "Program/Debug" PERTEvents have been updated.

Other inefficiencies can arise due to the granularity of the change propagation. For example, if you are editing the PERTChart shown in Figure 6.3, moving the "Design" event results in the sending of the update: message to all of its dependents, and two of its dependents are the constraints for the two connected PERTEvents, "Document" and "Program/Debug". Since these constraints get the update: message they reevaluate themselves, but this is unnecessary since the ending date of the "Design" event did not change. Since the granularity of the change propagation is at the figure level, some constraints are evaluated unnecessarily.

## 6.2 Centralized system

A centralized approach for constraints helps solve the problems listed in the previous section. A centralized constraint system knows about all the constraints and is able to satisfy them all at once. HotDraw uses the ColbaltBlue system created by Bjorn Freeman-Benson [1]. It has been modified somewhat in order for it to work with HotDraw. Each Drawing has its own constraint system, located in the ConstraintDrawing. This constraint system is responsible for adding and removing constraints as well as solving them. These operations are performed by the methods addConstraint:, addAllConstraints:, removeConstraint:, removeAllConstraints:, and solveConstraints.

One of the problems with the old constraint system was the large granularity of constraints. Constraints were always between figures, not between attributes of figures. In the new constraint system, constraints are between attributes of figures. A figure has a value holder for each attribute that is involved in constraints, and stores a constrained attribute in a value holder instead of directly in an instance variable. Figures create new value holders by sending the variableWith: message to themselves. These value holders belong to the constraint system, so figures always change the value of a value holder by asking the constraint system. This lets the constraint system change other value holders to ensure that all constraints are maintained. A figure requests that the constraint system changes a value by sending the value: message to the value holder holding the value. Figures can read their values from the value holders by sending the value message. For example, instead of containing two points for the origin and corner and three numbers for the colors (border and fill) and the border width, an ellipse figure would contain two point value holders and three regular value holders, and statements such as "origin := aPoint" would become "origin value: aPoint".

Since figures never explicitly set their values (this is done by the constraint system), they do not know when their values change and, as such, cannot initiate the damage protocol through the methods willChange and changed. Instead, the constraint system tells each figure when it is about to change and after it has performed all changes, effectively initiating the damage routines. This simplifies figures somewhat since they no longer are required to send the willChange/changed messages when performing updates to their value holders. This eliminates many "nuisance bugs" that occur as a result of forgetting to properly damage the drawing before or after a change. Another advantage of this system is that since the constraint system owns the value holders, it controls when the values are set and can check that a request to set a value is valid before the change is performed.

In order for the constraint system to contain only the current figures' value holders, the drawing needs to add (remove) the value holders for a figure to the constraint system when it adds (removes) a figure to itself. Thus, each figure must define methods to return the value holders. The main method that every figure that adds value holders must redefine is variablesDo:. This iterates over all of the value holders that a figure contains. Similarly, methods must be defined to return the constraints that a figure contains. Once again, there is one standard method that must be redefined; constraintsDo:.

Although variablesDo: and constraintsDo: are the only methods that a figure needs to redefine for the drawing to add and remove it, new types of figures should also redefine at:, at:put:,

| Methods | Comment |
|---|---|
| equalityBetween: aVar1 and: aVar2 | aVar1 = aVar2 |
| equalityBetween: aVar1 and: aVar2 plusOffset: aConstant | aVar1 = aVar2 + aConstant |
| plus: aVar1 and: aVar2 equal: aVar3 | aVar1 + aVar2 = aVar3 |
| origin: origin corner: corner xPercent: xPercent yPercent: yPercent offset: offset equal: aPoint | Layout aPoint in the rectangle |

**Figure 6.4**: Commonly used HotDrawConstraints

```
rightSideOrigin: origin corner: corner to: aPoint
    ^HotDrawConstraint new constraints: (Array
        with: (ColbaltBlueNonuniqueConstraint greaterEqualBetween: aPoint y and: origin y)
        with: (ColbaltBlueNonuniqueConstraint greaterEqualBetween: corner y and: aPoint y)
        with: (SkyBlueConstraint equalityBetween: aPoint x and: corner x))
```

**Figure 6.5**: Constraint to connect a point to the right side of a figure

variableNamesDo:, and constraintNamesDo:. Redefining these methods allows the figure to be copied and also make it easier to define new constraints interactively. The variableNamesDo: and constraintNamesDo: iterate over the names of all the value holders/constraints, and the at: and at:put: are used to explicitly get or set the value holder or constraint for a particular name.

Once a figure's attributes have been converted into value holders, constraints can be placed either between figures or inside of a figure. The standard way to create a constraint is to first get the class of the constraint by sending the constraintClass method to a figure (or command), and then send the message to create the constraint. For example, to create a constraint for a rectangle figure that specifies that its origin is 30 pixels left and 40 pixels higher than the corner, the rectangle figure would execute "self constraintClass equalityBetween: corner and: origin plusOffset: 30 @ 40". Figure 6.4 shows some of the commonly used constraint creation methods. The other creation methods can be found in the *instance creation* protocol of HotDrawConstraint.

Occasionally you may need to create a new type of constraint. There are two basic ways to create a new type of constraint. The easiest way is to create a new HotDrawConstraint by combining a set of the more primitive SkyBlueConstraints or ColbaltBlueNonuniqueConstraints. For example, if you want a constraint that connects a point to the right side of a rectangle, you would create a HotDrawConstraint as shown in Figure 6.5. This code first creates a ColbaltBlueNonuniqueConstraint that constrains the point to be below the top of the rectangle. Next, it creates a similar constraint to make the point be above the bottom of the rectangle. Finally, it creates a SkyBlueConstraint to constrain the point's x value to be equal to the rectangle's

```
equalityBetween: variable1 and: variable2
    | constraint method1 method2 |
    constraint := SkyBlueConstraint new.
    constraint name: (variable1 printString), ' = ', (variable2 printString).
    constraint strength: SkyBlueStrength required.
    constraint variables: (Array with: variable1 with: variable2).
    constraint errorfunction: [:vars | ((vars at: 1) contents - (vars at: 2) contents) abs].

    "variable2 := variable1"
    method1 := SkyBlueMethod new.
    method1 inputs: (Array with: variable1).
    method1 outputs: (Array with: variable2).
    method1 external: false.
    method1 block: [:in :out | (out at: 1) contents: (in at: 1) contents].

    "variable1 := variable2"
    method2 := SkyBlueMethod new.
    method2 inputs: (Array with: variable2).
    method2 outputs: (Array with: variable1).
    method2 external: false.
    method2 block: [:in :out | (out at: 1) contents: (in at: 1) contents].

    constraint methods: (Array with: method1 with: method2).
    ^constraint
```

**Figure 6.6**: SkyBlue equality constraint

right side. To find the existing kinds of primitive constraints, look in the instance creation methods of SkyBlueConstraint and ColbaltBlueNonuniqueConstraint.

If the basic primitive constraints do not exist, you will first need to create a SkyBlueConstraint or a ColbaltBlueNonuniqueConstraint and then create the HotDrawConstraint. The ColbaltBlueNonuniqueConstraint is just a special type of a SkyBlueConstraint that is used for inequality constraints such as a "greater than" constraint. These constraints are created by specifying the name (used for debugging), strength, variables, error function, and methods. Figure 6.6 shows a creation method for an equality constraint. First, a constraint object is created, given a name, and assigned a strength. There are four different strengths that are used to give priority to a constraint. For example, if you have a constraint with *required* strength and another with *weak* strength, you are specifying that the *required* constraint is given priority over the *weak* constraint if only one can be satisfied. The next part of the code specifies the collection of variables that the constraint uses and the error function. The error function is a block that takes the collection of variables as an argument and returns a positive number that is an indication of how close the constraint is to being satisfied. It returns zero if it is already satisfied. The last part of the code creates the methods for the constraint. There should be a method for every possible way to solve the constraint. For the equality constraint (x = y), there are two solutions (x := y and y := x). A method is created by specifying the input and output variables, an external flag, and a block to solve the constraint. The block has two arguments. The first is the input variables, which are read only, and the second is the output variables. When it is executed, it sets the values of the output variables so that the constraint is satisfied. The external flag is set to true if it is possible that the method might execute to some different value even if the values of its inputs has not changed (e.g., the method makes a reference to some external object). For most methods this flag is false.

# Chapter 7

# Changes to HotDraw

This chapter discusses many of the changes that have been made to HotDraw since its last release. It lists the changes necessary to convert old HotDraw programs into new ones. Additionally, it can also be used as a guide to the types of transformations made as frameworks evolve.

## 7.1 Figures

**New Figures** Several new figures have been added to HotDraw. Two of these, spline and bezier, are new figures that can be used in ordinary HotDraw applications. These figures should have no affect on existing HotDraw code.

Other figures were added to help implement drawings. These figures are the ContainerFigure, LayeredContainerFigure, and InvisibilityWrapper. The ContainerFigure and LayeredContainerFigure are composite figures that the Drawing uses to hold its figures, and the InvisibilityWrapper is used by the Drawing to make individual figures or layers invisible. Sending the message hide:, hideLayer:, show:, or showLayer: to the Drawing causes it to add or remove an InvisibilityWrapper on a specified figure or layer in the drawing. Section 7.3 describes the effects of these new figures on the Drawing.

**Constraints** As a result of having a centralized constraint system, many changes need to be made to figures so that they would support the constraints. These include changing the variables to value holders and adding the support methods for the new value holders. To

```
variablesDo: aBlock
    aBlock value: origin.
    aBlock value: corner.
    aBlock value: width.
    aBlock value: color.
    aBlock value: fillColor.
    aBlock value: transparent.
    super variablesDo: aBlock

variableNamesDo: aBlock
    #(#origin #corner #width #color #fillColor #transparent) do: aBlock.
    super variableNamesDo: aBlock

at: aVariableName
    aVariableName = #origin ifTrue: [^origin].
    aVariableName = #corner ifTrue: [^corner].
    aVariableName = #width ifTrue: [^width].
    aVariableName = #color ifTrue: [^color].
    aVariableName = #fillColor ifTrue: [^fillColor].
    aVariableName = #transparent ifTrue: [^transparent].
    ^super at: aVariableName

at: aVariableName put: aVariable
    aVariableName = #origin ifTrue: [^origin := aVariable].
    aVariableName = #corner ifTrue: [^corner := aVariable].
    aVariableName = #width ifTrue: [^width := aVariable].
    aVariableName = #color ifTrue: [^color := aVariable].
    aVariableName = #fillColor ifTrue: [^fillColor := aVariable].
    aVariableName = #transparent ifTrue: [^transparent := aVariable].
    super at: aVariableName put: aVariable
```

**Figure 7.1**: EllipseFigure's value holder methods

change figures that contain the values directly in variables to ones that contain them in value holders, you need to perform three actions.

First, change the initialization method so that instead of assigning a value directly to an instance variable, you create a value holder with the value and assign it to the instance variable. To create a value holder, the figure must send the variableWith: message to itself. Effectively, initialization statements of the form "origin := aPoint" are transformed into "origin := self variableWith: aPoint". New constraints can also be created by sending the constraintClass message, and then sending the appropriate creation message to this class, such as equality:between:.

```
copy: aDictionary
    | aCopy |
    ^aDictionary at: self
        ifAbsent:
            [aCopy := super copy: aDictionary.
            aCopy setFigures: (figures collect: [:each | each copy: aDictionary]).
            aCopy]

postCopy: aDictionary
    super postCopy: aDictionary.
    self resetVisibleFigures.
    figures
        do:
            [:each |
            each postCopy: aDictionary.
            each container: self]
```

**Figure 7.2**: CompositeFigure's copying methods

Second, change all uses of the instance variables into "instVar value" (i.e., send the value message to the value holder to get the value). Third, change assignment statements of the form "instVar := aValue" to "instVar value: aValue" except for those statements that were already changed in the initialization method for the figure.

After all variable references have been converted into value holders, you need to make them available to other objects such as the Drawing. You can do so by redefining the four methods variablesDo: variableNamesDo:, at:, and at:put:. Figure 7.1 shows an example of each of these for the EllipseFigure. The variablesDo: evaluates a block for each value holder it contains, effectively iterating over its value holders. The variableNamesDo: is similar except that it gives a unique name to each of the value holders so that the at: and at:put: can read and write the value holders. These last three methods are needed so that figures can have a generic copy method.

All of the constraints that a figure contains should also be available to the drawing. The approach for constraints is similar to the one for value holders. If a figure contains a constraint then it must also redefine constraintsDo: and constraintNamesDo:. These are similar to variablesDo: and variableNamesDo:. Also you must also add in the appropriate code in the at: and at:put: methods as for the variables above.

**Copying** Figures can now perform deep copies. This is done by adding copy: and postCopy: methods that work in much the same ways as the standard copy and postCopy methods. These methods take a dictionary that can be used to aid a deep copy. This allows the constraints and value holders for each figure to also be copied. In order to make a general copy method in class Figure, the copy methods use the methods defined under the Constraint's section above. Subclasses of Figure that need special copying, such as CompositeFigure with its collection of other figures, need to redefine copy: and/or postCopy: so that correct copies are made. Figure 7.2 shows the CompositeFigure's copy: and postCopy: methods. These methods are redefined so that the components figures of the composite figure are also sent the copy: and postCopy: messages.

Figures are copied by sending the copyFigures: message to the drawing. This method returns an object that can be pasted into the drawing by using either paste: to paste the figures at their original location or paste:at: to paste the figures at a specified point.

**Colors** In order to easily make constraints on color information, colors have been changed from ColorValues to indices in a color table that is located in a class variable of class Figure. Figures can locate the index of a color value by sending the message colorIndex:, and can look up the value at a color index by sending the message colorAtIndex:. This method returns a color value that can be used on graphics contexts. These changes can have a small affect on subclasses of the standard figure classes. These classes should send these messages instead of directly accessing the color information.

**Accessing Methods** Some of the accessing methods for figures that were originally used only by handles have been removed. These methods are no longer used by handles, instead handles use constraints to perform these tasks. Other methods that help to create constraints have been defined, such as topLeft:, topRight:, and center:offset:. To support these methods, figures need to implement the two methods originVariable and cornerVariable that return the value holders for the origin and corner. Line figures do not have value holders that represents their origin or corner and so these methods do not work for line figures. These changes could have some impact on existing code that uses the handle accessing methods, but these methods can be modified to work in the new system.

**handles**

```
^(OrderedCollection with: (ConstraintHandle twoDHandleFor: self on: (points at: 1)))
    add: (ConstraintHandle twoDHandleFor: self on: (points at: 2));
    add: (ConstraintHandle twoDHandleFor: self on: (points at: 3));
    add: (ConstraintHandle twoDHandleFor: self on: (points at: 4));
    add: (CommandHandle connectionFor: self
            at: #center:
            class: LineFigure);
    add: (ConstraintHandle verticalHandleFor: self
            on: color
            at: #center:offset:
            offset: 10 @ 0
            sensitivity: 5);
    add: (ConstraintHandle verticalHandleFor: self
            on: fillColor
            at: #center:offset:
            offset: −10 @ 0
            sensitivity: 5);
    add: (ConstraintHandle verticalHandleFor: self
            on: width
            at: #center:offset:
            offset: 0 @ 10
            sensitivity: 5); yourself
```

**Figure 7.3**: RectangleFigure's handles method

**Handles** As a result of the new constraint system, Handles are created differently. They are no longer parameterized with a Locator for their position, instead they have a constraint placed between their location and the figure that owns them. Figure 7.3 shows the new handles method for a RectangleFigure. Handles in the original version whose location were specified by specific points such as topLeft are now created using the on: variety of handles, and the others that were specified from two or more points are now created using the on:at: or on:at:offset: variety.

There are only two main handle classes, a ConstraintHandle and a CommandHandle. The ConstraintHandle creates a constraint between the mouse and some value holder of the figure, such as a point. Manipulating the handle changes the value. The CommandHandle executes commands when it is pressed. Most TrackHandles in the original version can be updated to be ConstraintHandles since they change some value holder. Those Track-

```
press
    | figure lastPoint aPoint |
    aPoint := self sensor cursorPoint.
    figure := self drawing figureAt: aPoint.
    figure class = CanvasFigure ifFalse: [^self].
    lastPoint := aPoint.
    figure paintAt: lastPoint
        brushWidth: figure width ** 2.
    [self sensor redButtonPressed]
        whileTrue: [
            aPoint := self sensor cursorPoint.
            figure paintFrom: lastPoint
                to: aPoint
                brushWidth: figure width ** 2.
            lastPoint := aPoint].
    figure paintAt: lastPoint
        brushWidth: figure width ** 2
```

```
initialize: aPoint
    lastPoint := aPoint.
    figure paintAt: lastPoint
        brushWidth: figure width ** 2

moveTo: aPoint
    figure paintFrom: lastPoint
        to: aPoint
        brushWidth: figure width ** 2.
    lastPoint := aPoint

effect
    figure paintAt: lastPoint
        brushWidth: figure width ** 2
```

**Figure 7.4**: HotPaint's PaintbrushTool

Handles that cannot be converted into ConstraintHandles can instead be converted into CommandHandles as described in section 7.2.

## 7.2 Tools and Handles

**Commands** Tools have been decomposed into readers and commands. Although you will probably never make a new reader, you will probably need to create new commands which can then be plugged into a tool using the ToolBuilder. Converting an action from an existing tool is simple. First you identify the initialization, the control loop, and the finalization actions in the original tool. Tools that act only on click events have only one section so the other two are blank. Once you have identified these areas, you can convert these actions into commands. Most commands only require a transfer of these parts into the initialize:, moveTo:, and effect methods of the new command (or the initialize, keyboardEvent:, and effect for text commands). Figure 7.4 shows an example of a tool in HotPaint that was converted into commands. On the left is the original version of the tools, and the right contains the command for that action. The code for the sensor has been removed from the command. Furthermore, the code that performed the check of the figure's class has been removed since it is automatically done by the tool when the command is selected.

**Handles** Handles in the original version of HotDraw contained the routines to change the drawing, but in the new version this is performed by commands. To convert an action from a handle to a command, follow the guidelines listed above for tools. If the handle was originally a subclass of a TrackHandle then convert its invokeStep: method to the moveTo: method in the new command. Once the command for the handle has been created, you can create the CommandHandle by giving it an instance of the new command.

**Figure Creation Methods** Figure creation methods have also been affected by the new tools. Instead of class methods that created figures by reading the sensor, like the createNotifying:, figure creation is now done by commands. Most likely, you will be able to use the FigureCreationCommand to create your new figure, but in case you cannot, you should be able to create a new command by identifying the three parts of the creation method and converting these into a command, as was done above for tools.

## 7.3 Drawing and the DrawingView

**Handle and Figure accessing** The original HotDraw had the current selection in the DrawingView, and also contained the methods that accessed figures such as addFigure: and remove: that were just forwarded on to the Drawing. The new version has moved these methods from the DrawingView into the Drawing. The DrawingView contains a Drawing that contains both the selection (handles) and a ConstraintDrawing. The ConstraintDrawing contains the figures in a layered composite figure and also contains the constraint system. Old code that uses the accessing methods in the DrawingView should be changed to access the Drawing instead.

**Layered drawings** Both the Drawing that contains the selection and the ConstraintDrawing are layered. Layers let you make a set of figures invisible quickly by wrapping a layer in an InvisibilityWrapper. Both of these drawings are subclasses of a new figure called a

LayeredContainerFigure. The Drawing has two layers, one for the handles, and one for its ConstraintDrawing. On the other hand, the ConstraintDrawing can have many different layers, but many applications only use one layer. Each layer can contain a set of related figures.

These changes should only affect those subclasses of Drawing that directly accessed the figure variable of the Drawing, since this variable is now inherited from the ContainerFigure and only contains a collection of composite figures. Subclasses of Drawing should send the **figures** message, instead, to get the figures in the drawing. The **figures** method returns a collection of figures like that contained by the original figures variable.

**Damage** Damage is now handled by the DrawingView instead of the Drawing. This allows drawings to be contained in many different DrawingViews, whereas before, a Drawing could only be contained in one DrawingView since it would not be redrawn in all view properly. This should have little or no affect on most applications since the previous damage routines were general enough to work with most applications without being modified.

# Chapter 8

# Conclusions

HotDraw was originally designed by Kent Beck and Ward Cunningham of Tektronix. It was written for Tektronix Smalltalk but was ported to early version of ParcPlace Smalltalk. It was used at the University of Illinois both under Tektronix and ParcPlace Smalltalk. Release 4.0 from ParcPlace completely changed the graphics model of Smalltalk and significantly changed the implementation of Model-View-Controller. Pat McClaughry reimplemented HotDraw for Release 4.0, but kept the design. The new graphics model was much closer to the design of HotDraw than the old model, so this was not hard.

Over the years various changes were made to HotDraw, mostly by Ralph Johnson. None of the changes were significant; the biggest were adding metafigures and constraints as objects (but without a centralized constraint system). All of the versions had drawing, figures, tools, and handles as classes that application programmers would subclass. The latest redesign makes HotDraw much more composable, and takes it another step closer to allowing people to customize HotDraw without programming.

## 8.1 Design changes

HotDraw underwent several different transformations. One transformation was changing the inheritance hierarchy. This was done to both drawings and handles, which became subclasses of figure. These changes primarily affected the implementation of the framework. Some of the operations that were implemented by drawings and handles could now be inherited. Another effect was that some conditional tests for command lookup in tools could be simplified. Since these changes did not affect the interfaces, they had only minor effects on programs that used the framework.

Another type of transformation was converting classes whose primary reuse technique was inheritance to use composition. This transformation was performed on tools. Unlike the previous transformation, this one affected most programs that used the framework, as well as the framework itself. Every tool needed to be split into its components to be usable. The resulting system was more reusable since tools could easily be created by plugging together readers and command, and also it was easier to create new actions since less knowledge was required to write a command. One problem with this change was that it resulted in a more complicated design, but much of this complexity is hidden from users.

Adding features is another common design change that was used on HotDraw. This change helped standardize different implementations of the same features. For example, the new HotDraw has a standard way to make figures invisible. You do not need to decide which implementation you want to use to make figures invisible, since that is already addressed by the framework. Furthermore, different programs use the same implementation so it is easier to understand them. One drawback of adding features is that having many features can complicate the design and make learning the framework more difficult.

A final type of design change that occurred to HotDraw was moving operations from one object to another. These changes occurred mainly to the DrawingView and Drawing. Many of the operations that the DrawingView performed such as managing the selection were moved to the Drawing, and damaging was moved from the Drawing to the DrawingView. There are two reasons why these transformations were performed. One is that some of these operations were more closely tied to the object they were moved to than the other. For example, damaging is more closely tied to views, since they are responsible for redisplaying the damaged area. The other reason why these transformations were performed was to limit the number of interfaces that an object must know about. Tools in the original version were responsible for changing the drawing and also updating the view. To limit the number of interfaces that these tools needed to know about, some of the actions on drawings were put in the DrawingView. But in the new version, tools have been split into commands. Tools still update the view, but commands only interact with the drawing so the actions on drawings that were in the DrawingView are now in the Drawing.

## 8.2 Future work

Although HotDraw has gone through several design iterations, it is not "done." There are still several areas in the design that can be improved. One area is tools and controllers. Tools are essentially controllers that accept only left mouse button and keyboard input. If tools handled all input then they would be a controller, and they could replace the DrawingController. Middle and right mouse button input for the tool could either be handled in the same fashion as left mouse button input by using readers and commands, or since all applications perform these operations the same, it could be handled in standard methods.

Another area that can be improved is handles and menus for figures. Currently, new handles and menus can only be specified by subclassing and overriding the handles and menu methods. This makes it difficult to interactively create new figures that have special handles or menu since we must create subclass to override these methods. Instead, we would like to build a new type of figure simply by composing its graphical representation with its handles and menu. There are several approaches that can be used to solve this problem. One approach is to use wrappers. Instead of creating a new figure class, you would wrap the figure in an object that defines the menu or handles for the figure. Another approach would be to store the menu and handle information in some object and retrieve it when it is needed. This approach would be similar to the command table that tools use to map manipulation actions of figures to commands. Two possible locations for storing this information are either in the SelectionCommand or in some new application description. If an application description is used, it could also describe the tools and drawing class that the drawing editor should create without needing to create a new drawing editor class. Still another approach would be to store the handles and menu in an instance variable of the figure.

A final way that HotDraw might be improved is in the way it separates the visual representation of figures in a drawing from the application objects being represented. HotDraw uses Figures for the visual part and Metafigures for the nonvisual part. However, metafigures aren't used a lot. In general, metafigures are an appendix to HotDraw, since the drawing stores figures directly and metafigures indirectly, and the editor manipulates figures directly and metafigures indirectly. Unidraw [6] also separates the visual representation (the VisualComponent) from the nonvisual part (the Component), but makes Components be the important part. VisualComponents are dependents of Components, so changing a Component causes the VisualComponent to change, too. In general, metafigures have worked well, but it seems that half the applictions of HotDraw do not use them at all, and it is possible that Unidraw's design might be better.

# Bibliography


[1] Bjorn N. Freeman-Benson. Converting an existing user interface to use constraints. In *User Interface Software and Technology*, 1993.

[2] Ralph E. Johnson. Documenting frameworks using patterns. In *Proceedings of Conference on Object-Oriented Programming, Systems, and Applications*, pages 63–76. ACM, 1992.

[3] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[4] ParcPlace Systems. *User's Guide to Objectworks\Smalltalk Release 4.1*, 1992.

[5] Ivan Edward Sutherland. *Sketchpad, A Man-Machine Graphical Communication System.* PhD thesis, Massachusetts Institute of Technology, January 1963.

[6] John M. Vlissides. *Generalized Graphical Object Editing.* PhD thesis, Stanford University, June 1990.