Creating Tools in HotDraw by Composition

Ralph Johnson
University of Illinois, Urbana-Champaign
215 PUBLICATIONS **38,066** CITATIONS

Creating Tools in HotDraw by Composition

John Brant **Ralph E. Johnson**
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Ave.
Urbana IL 61801, USA.
e-mail: {brant,johnson}@cs.uiuc.edu

Abstract

As frameworks evolve, it is common for reuse by inheritance to be replaced by reuse by composition. This has happened recently to HotDraw, a structured drawing editor framework for Smalltalk. Class Tool, which once was frequently subclassed, has been broken into several smaller components, and new tools can now be defined entirely by composing other objects. Although creating a new tool might require creating a new component, the amount of code necessary to create a new tool has decreased significantly, and it is easier to learn how to create a new tool. This is not only important for users of HotDraw, but illustrates a design technique that can be used in other programs.

1 Introduction

As frameworks evolve, it is common for reuse by inheritance to be replaced by reuse by composition. One reason that inheritance is used first might be that Smalltalk offers better support for inheritance than it does for composition. Another reason might be that it is easier to think about incrementally modifying an object than about breaking it into orthogonal parts. Whatever the reason, the first thought of designers seems to be to reuse behavior by inheritance, and designs based on composition evolve from ones that are based on inheritance.

For example, the original Model-View-Controller reused scrolling behavior by inheritance. A controller would reuse scrolling by being a subclass of ScrollController. In ObjectWorks\Smalltalk 4.1, a view is scrolled by wrapping it in a ScrollWrapper. This makes it easy to have both scrolling and non scrolling forms of a view, which was hard to achieve in the original version of MVC.

This paper describes how part of HotDraw has been redesigned to make it more reusable. The redesigned part is class Tool, which represents a user interface mode that can be selected from a palette. Originally the only components of Tool were an icon and cursor. We gave it two additional components. The result is that each tool can be described by a table, and we have a simple direct-manipulation "Tool Builder" application for defining new tools. Moreover, it is simpler and requires less code to define a new tool in the current version of HotDraw than in the original version.

2 HotDraw

HotDraw is a framework for structured drawing editors [Johnson 92]. It can be used to build editors for specialized two-dimensional drawings such as schematic diagrams, blueprints, music, or program designs. The elements of these drawings can have constraints between them, they can react to commands by the user, and they can be animated. The editors can be a complete application, or they can be a small part of a larger system.

HotDraw was originally designed by Kent Beck and Ward Cunningham when they were at Tektronix. Patrick McClaughry reimplemented HotDraw to work with ObjectWorks\Smalltalk R 4.0, and we have since modified it to work with R 4.1 and VisualWorks.

The basic HotDraw framework is made up of Drawing, Figure, Tool, Handle, and DrawingEditor. A drawing is a complete picture, while a Figure is a component of a Drawing. A Tool represents a mode of the drawing editor. Selecting a figure (with the SelectionTool, for example) causes the figure to present a set of handles that can be manipulated with a tool to change the figure. Finally, the DrawingEditor specifies the drawing being edited and the set of tools that are available to edit it. It acts as the "main program" of HotDraw, though it usually does little once the application starts. Figure 1 shows a drawing containing a RectangleFigure which is selected, an ArrowFigure, a TextFigure, a BezierFigure, and a SplineFigure. The palette of tools is on the left. The selected RectangleFigure has eight handles. The DrawingEditor is the model of the window.

Drawing and Figure are similar to CompositePart and VisualPart of ObjectWorks\Smalltalk R4.1. The only difference between Drawing and CompositePart is that drawings cache the underlying image to improve performance and they provide a **step** method to support animation. The only difference between Figure and VisualPart is that figures provide a **handles** method to access their handles, they are responsible for knowing their location in the drawing, and they notify dependents when any of their attributes change. DrawingEditor is a typical model, responsible only for providing a Drawing (i.e., acting like a ValueHolder of a Drawing) and for providing a palette of tools.

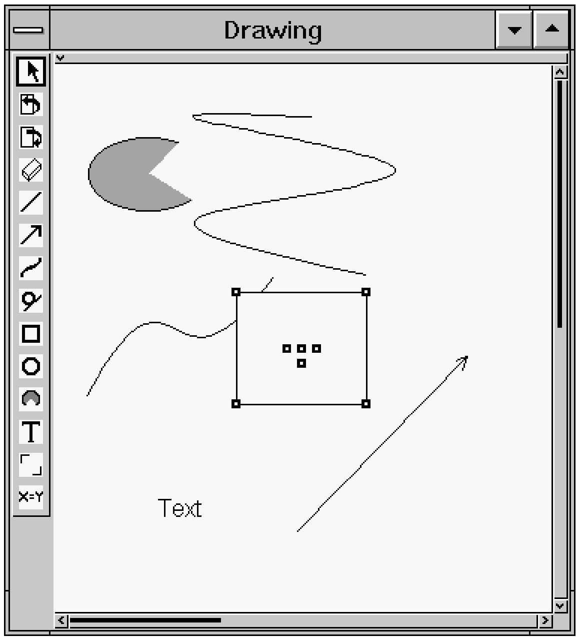


Figure 1: Default drawing editor

Tool and Handle are the classes in HotDraw that are most different from anything provided in ObjectWorks\Smalltalk. Tools are responsible for modifying the drawing based on a user gesture, such as a mouse drag or a keyboard press. A tool converts user actions into operations on a drawing, and so is similar to a controller. In fact, the DrawingController class delegates many of its operations to the current tool, and so tools could be thought of as an extension of the controller. However, a tool also has an icon that it displays in the tool palette and it has a cursor that it uses to let the user know which tool is selected. Thus, it is better to think about it as a user interface mode.

Handles represent attributes of the selected figures that the user can edit. Manipulating a handle changes the corresponding attribute in the selected figure. For example, manipulating a corner handle changes the size of a figure, and there can be handles that change the interior color of a figure or the border width and color.

A handle is a figure, so it is displayed just like any other figure. The SelectionTool is normally the only tool that is aware of handles. It requests handles from figures that it selects, and it manipulates handles.

3 Tools

After figures, new tools are the parts that developers using HotDraw need to create most often. Tools are responsible for interpreting a user action, such as a mouse drag or a keyboard press, by performing some action on the drawing such as moving a figure, creating a new figure, etc. Most new tools are made because the drawing needs to be changed in a new way. The next most common reason for new tools is to create a more refined version of some other existing tool. Still other times that new tools are created result from the need to combine actions from a few different existing tools into one new tool.

All tools implement both the press and processKeyboard methods. These methods are invoked by the DrawingController when the user presses the red (left) mouse button or a key on the keyboard, respectively. Within these methods the tool is responsible for modifying the drawing, and after all user manipulation actions are complete, the tool will then return control back to the drawing controller.

All tools in the original version of HotDraw handled user input in one of two ways. A tool could perform the operation itself, or it could delegate the responsibility to some other object such as a handle or a figure class.

One example of the tool handling the responsibility itself is when the selection tool selects a group of figures by drawing a bounding box around them. When the selection tool, receives control, it first checks to see what figure was pressed by the user. If the user didn't press any figure then the selection tool performs the multi-selection action. This is done in the marqueeFromUser method shown in Figure 2. The control loop moves the bounding box to the current mouse position. When the mouse button is released, the loop will terminate, and then all figures in the bounding area will be selected.

The other way that a tool handles user actions is to delegate them to some other objects such as a handle or a figure class to create a figure. An example of this occurs when the user uses a handle to modify an attribute of a figure. When the tool detects that the mouse is over a handle when the button is pressed, it sends the handle the invoke: message, and the handle keeps control until the button is released. An example invoke: method is given in the code segment in Figure 3. The invoke: method contains a control loop that is very similar to the one given above, in that it will read the current mouse position and perform some action based on this new position. It also breaks out of the loop when the user quits manipulating the handle (i.e., when the user releases the red mouse button). After this is done, the handle will perform any finalizing action that is needed and return control back to the selection tool.

4 Problems

Although this is an obvious way to implement user interactions, it has many problems. One problem is that code is often duplicated. These examples are typical because the most of the duplicate code is reading the sensor for either keyboard or mouse input. For example, in the sample applications that came with the original version of HotDraw, there were 11 instances of a control loop that read the mouse input. Clearly, this is not optimal code reuse. This duplicate code is more difficult to maintain, since

```
marqueeFromUser
  | gc startPoint stopPoint rectangle |
  gc := self view graphicsContext.
  gc lineWidth: 1.
  gc paint: MarqueeImage.
  stopPoint := startPoint := self sensor cursorPoint.
  rectangle := startPoint corner: stopPoint.
  [self sensor redButtonPressed]
    whileTrue:
      [self drawing damageRegion: rectangle.
      self view repairDamage.
      stopPoint := self sensor cursorPoint.
      rectangle := Rectangle vertex: startPoint vertex: stopPoint.
      gc displayRectangularBorder: rectangle at: Point zero].
  self drawing damageRegion: rectangle.
  self view repairDamage.
  self view selections: (self drawing figuresIn: rectangle)
```

Figure 2: Original Code from SelectionTool

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

Figure 3: Original Code from Handle

if the sensors change protocol, then many different code fragments must be updated. This problem of duplicate code can also be seen to a lesser extent with the code that performs the changes to the drawing.

Another problem is that the code that performs changes to the drawing is distributed over many classes, tools, handles, and figure classes. This makes HotDraw more difficult to learn and debug since it may be hard to tell which code is being executed. In addition to this code being distributed over many classes, in most cases it is also entwined with the code that reads the sensors. This also makes it harder to debug.

Other problems are more related to how tools are created. Tools in the original version of HotDraw are monolithic, encompassing the displaying of their cursors and icons as well as manipulating the drawing. As a result, it is difficult to make a new tool by composing properties of two different, existing tools. For example, suppose that you wanted a tool that created a new figure when the user clicked on the background, but allowed the user to manipulate an existing figure in the same manner as a selection tool. In order to make such a tool, you would have to subclass either a figure creation tool or the selection tool, and then copy methods from the other into the newly created class. For this example, you would probably subclass the selection tool and then copy three other methods from the creation tool into the new class. You would still need to modify the press method in the new tool so that it will act properly when pressed on the background. In addition to duplicating code, this is more work than should be required to make a tool out of existing tools, since it requires you to read code and decide which methods should be copied or changed.

5 A Solution

One way to eliminate the problems listed above without losing any functionality is to create two new types of objects and to modify the roles of tools, handles, and figure creation.

Readers and commands are new objects that help eliminate the duplicate code that appeared in many of the tools, handles, and figure creation methods. These are similar to the manipulators and commands of Unidraw, which inspired our solution [Vlissides 90]. Readers read the sensor values, and the commands perform the actual changes to the drawing. Both of these objects work together to carry out the user manipulation. Readers get the current value of the sensors and pass it onto the commands, which perform an action based on the new sensor value. This method reduces the 11 instances of a control loop that checked the mouse sensor to one. If the sensor protocol for checking the mouse changes, we will only have to change our code in a few places. Moreover, this method makes debugging new commands easier since all code for changing the drawing is located only in these command objects, and the command that is being executed is easily found from the tool being used.

Readers must implement three methods, initialize:command:, readSensor:, and effect, while commands implement methods that correspond to a particular reader. An example command for a DragReader is listed in Figure 4. This command performs the same multi- selection actions as was seen earlier. Readers rarely need to be subclassed, and in fact in the new system, there are only three readers, two for the mouse and one for the keyboard.

Splitting commands and readers from the tools, handles, and figure creation methods makes it easy to create tools. Tools contain their icon and cursors as before, but now they also contain a table that maps a given figure and a reader to a command. With this scheme when the user performs some operation, the current tool creates a reader for that operation. After creating the reader, the tool finds the figure that the user is manipulating, and looks up the command in the table that matches the reader and figure. After the command is found, the tool will clone it and initialize the reader with the command. The reader will then ini-

```
initialize: aPoint
  originalStartPoint := aPoint.
  lastPoint := aPoint

moveTo: currentPoint
  | gc rectangle |
  lastPoint := currentPoint.
  rectangle := Rectangle vertex: originalStartPoint vertex: lastPoint.
  gc := self graphicsContext.
  gc paint: MarqueeImage.
  gc displayRectangularBorder: rectangle at: Point zero.
  self drawing damageRegion: rectangle

effect
  self drawing selections: (self drawing figuresIn:
      (Rectangle vertex: originalStartPoint vertex: lastPoint))
```

Figure 4: New Code for MultiSelectionCommand

	DragReader	ClickReader
default	Multi-selectionCommand	
ConstraintHandle	Constraint command	
CommandHandle	Figure command	
Figure	Selection and Constraint command	Selection command
RectangleFigure		
... all other subclasses of Figure		

Table 1: Command table for the SelectionTool

tialize the command with the current mouse point. After the initialization phase is finished, the tool begins looping over the reader. The reader will update the command with the current mouse position, and return true when the user manipulation action is not finished. When the reader returns false, the tool will tell the reader that the user action is finished and the command to produce its effect and clean up.

Command lookup is performed by the tool trying to find a match in the table for the given figure. An example of the SelectionTool's table is shown in Table 1. If no match if found the tool climbs up the figure's inheritance hierarchy until a match is found. Once a match is found then the tool will find the closest reader for the figure that is in the table, and return its command. From the SelectionTool's table below, we can see that given a ClickReader and a RectangleFigure, the command that is selected is the selection command, since Figure is the first class in the RectangleFigure's hierarchy which has an entry in the table. On the other hand, if the reader is a ClickReader and the figure is a ConstraintHandle, then the command will be a constraint command. Since ConstraintHandle is in the table, we take the command for the closest reader in the table which is the constraint command corresponding to the DragReader. If the tool was not able to find a match for the figure after climbing the inheritance hierarchy, then the default command for the tool is chosen. For example, if the user clicked on the background of the drawing, then the SelectionTool will try to find an entry in the table for a ClickReader and a figure which is nil. Since the figure, nil, and its superclasses are not in the table, the SelectionTool will pick the multi- selection command.

By using tables to represent the tool and the possible actions that it can take, we can now create tools by just filling out a table, and also supplying it with two cursors and the icon for the tool. Furthermore, there is rarely any need to make a subclass of tool, since most information that describes a tool is given in this table. In fact the only subclass of tool in the entire HotDraw system and all of its applications is the TextTool. This tool is able to edit and create text in the drawing. A subclass was needed since the tool needed to remember the text figure being edited from one manipulation action to another. Even special figures like handles don't need any special care in processing. All we need to do is place them in our table with their respective commands.

Since defining a new tool is just filling out a table, we created the Tool Builder to facilitate this entry. An example is shown in Figure 5. There are two main parts to the tool builder. The first is the browser located on the left, which is used for create new tools and define their actions. The other part is the icon and cursor editor. This is a normal HotDraw application that is used to edit either the icon or cursor when one of the icon, cursor, or manipulating cursor buttons is selected. There are five parts to the browser. The top left pane lists the tools that are currently in the system. The middle pane of the browser lists the kinds of figures that the tool will perform actions on, and the top right pane contains the readers for the selected figure and tool. In the middle there are a set of buttons which let the user switch from editing cursors or icons to editing the command tables for the mouse and keyboard part of the tools. The bottom pane shows the command that will be invoked when this tool is invoked on the specified figure. The example given in Figure 5 shows the command that is performed when the SelectionTool is pressed on some figure besides a handle, and when the user is performing a drag action with the mouse. This command is a composite command called a GroupCommand which when executed will select (SelectionCommand) and allow the user to move the figure (ConstraintCommand).

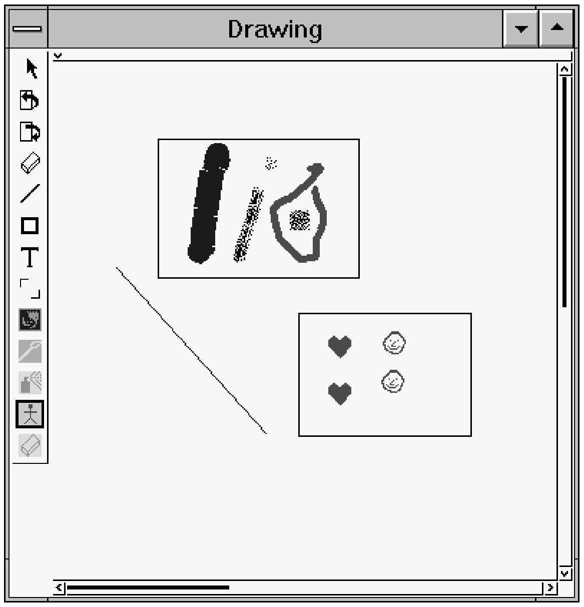


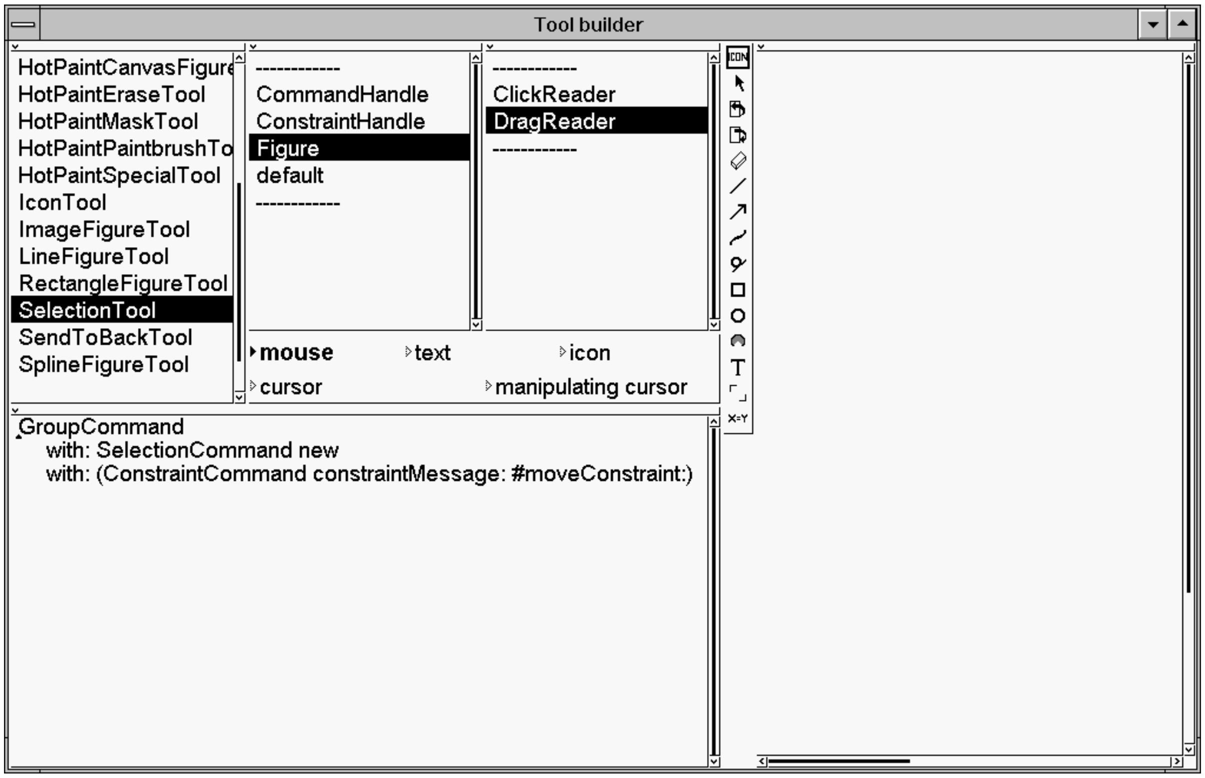
Figure 6: HotPaint drawing

6 Example

One of the example applications for HotDraw is the HotPaintEditor. This application is similar to painting programs like the Paintbrush program that comes with Microsoft Windows. An example screen is shown in Figure 6. The original implementation of this editor contained four new tool classes in addition to a new figure called CanvasFigure and a new drawing editor class.

All of the new tool classes dealt in some way with changing the CanvasFigure. They performed actions such as drawing lines, spray painting, erasing, and drawing special decals such as hearts or happy faces on the CanvasFigure. All of these tools needed to redefine the `press` method in order to perform their special action. An example of the line drawing tool's `press` method is given in Figure 7. As can be seen from this method, it contains the usual control loop that reads the current mouse location and then draws a line in the CanvasFigure. Furthermore, you can see that this tool checks that the type of the figure that is being manipulated is actually a CanvasFigure.

The PaintbrushTool class was replaced by a



```
GroupCommand
    with: SelectionCommand new
    with: (ConstraintCommand constraintMessage: #moveConstraint:)
```

Figure 5: Tool builder

```
press
  "Paint stuff on canvas object."
  | figure aPoint lastPoint |
  lastPoint := nil.
  [self sensor redButtonPressed]
    whileTrue:
      [aPoint := self sensor cursorPoint.
      figure := self figureAtPoint: aPoint.
      figure class = CanvasFigure
        ifTrue:
          [self brushWidth: figure width ** 2.
          lastPoint isNil
            ifTrue: [figure paintAt: aPoint brushWidth: brushWidth]
            ifFalse: [figure paintFrom: lastPoint to: aPoint brushWidth: brushWidth].
          lastPoint := aPoint]
        ifFalse: [lastPoint := nil]]
```

Figure 7: Original Code for PaintbrushTool

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

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

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

Figure 8: New Code for PaintbrushCommand

PaintbrushCommand class. This class replaces the press method of PaintbrushTool. The temporary variables figure and lastPoint become instance variables. This resulted in a considerable savings since much of the code was needed for these two common tasks. The resulting code for the new command is listed in Figure 8.

After converting these four tools to the new version of HotDraw, additional savings were noticed. Instead of there being a separate command for each tool, two tools were able to consolidate their commands into one. Additionally, the erase command was able to become a subclass of the line drawing command. The separation of tools into three parts resulted in smaller, more reusable objects being created.

7 Conclusion

Inheritance is not the only reuse technique offered by object-oriented programming. Often the most reusable designs are those where the primary reuse technique is instance composition.

Conventional programmers who are trying to write reusable software often make their programs table driven. This is a good technique for object-oriented programs too. The difference is that the elements stored in the table can be other complex objects like figure classes or readers, and it is easy to use the built-in inheritance relationships to make the tables smaller, as we did with the figure classes. Thus it is easier and more efficient to make table-driven object-oriented programs. A design reaches a new level of reuse when it becomes so composable that new kinds of objects can be designed entirely by filling in tables. Then you can provide a tool so that people can customize and extend your system without programming. Even when new code must be written, such as new commands, a design style leading to small, composable objects will tend to make them simple and easy to write.

HotDraw is available by anonymous ftp from st.cs.uiuc.edu.

8 Acknowledgments

Many thanks to Gordon Davis, Brian Foote, Paul Keefer, Dan LaLiberte, Don Roberts, and Bill Walker for providing many helpful comments.

References

[Johnson 92] Ralph E. Johnson. Documenting frameworks using patterns. In *Proceedings of Conference on Object-Oriented Programming, Systems, and Applications*, pages 63–76. ACM, 1992.

[Vlissides 90] John M. Vlissides. *Generalized Graphical Object Editing.* PhD thesis, Stanford University, June 1990.