Documenting Frameworks using Patterns

Ralph E. Johnson
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Ave.
Urbana IL 61801

Abstract: *The documentation for a framework must meet several requirements. These requirements can all be met by structuring the documentation as a set of patterns, sometimes called a "pattern language". Patterns can describe the purpose of a framework, can let application programmers use a framework without having to understand in detail how it works, and can teach many of the design details embodied in the framework. This paper shows how to use patterns to document a framework, and includes a set of patterns for HotDraw as an example.*

Christopher Alexander, an architect, developed the idea of a "pattern language" to enable people to design their own homes and communities [Alexander et. al.]. A pattern language is a set of patterns, each of which describes how to solve a particular kind of problem. His pattern language starts at a very large scale, explaining how the world should be broken into nations and nations into smaller regions, and goes on to explain how to arrange roads, parking, shopping, places to work, homes, and places of worship. The patterns focus on finer and finer levels of detail, passing though a discussion of how to arrange rooms in a house, and finally describing the type of material to use for walls, how to decorate rooms, and how to provide lighting.

Alexander's pattern language is supposed to be a document that non-architects can use to design their own communities and homes. No specialized training is needed to use it. It focuses on common design problems that non-architects will encounter, like how to build bedrooms and row houses, rather than uncommon ones, like how to build concert halls and cathedrals.

To be presented at OOPSLA'92.

Although Alexander uses the term "pattern language" to describe his document, it is not a formal language like a context-free language, for example. A pattern language is a structured essay, not a mathematical object. Therefore, we will replace that term with the term "patterns".

A framework is a reusable design of a program or a part of a program expressed as a set of classes [Deutsch][Johnson and Foote]. Like all software, it is a mixture of the concrete and the abstract. Since frameworks are reusable designs, not just code, they are more abstract than most software, which makes documenting them difficult. Frameworks are designed by experts in a particular domain and then used by non-experts. The principal audience of framework documentation is someone who wants to use the framework to solve typical problems, not someone building a software cathedral. Patterns seem to be well suited for this audience.

This paper shows one way to document frameworks with patterns. It is essentially an experiment to see how well patterns work to describe a framework. The result is a set of patterns that are included in the appendix. The main purpose of a set of patterns is to show how to use a framework, not to show how it works, but patterns can also describe a great deal of the theory of its design.

1. A Format for Patterns

One of the important features of Alexander's pattern language is its structure; each pattern is written in a particular format and patterns are arranged so that each pattern leads into the next. The format that Alexander uses is unlikely to be suitable for software. For example, since he is building physical objects, he requires pictures both for stating the problem that a pattern solves and for stating the solution. Thus, the patterns in this paper have a different format that the one he used.

Each pattern describes a problem that occurs over and over again in the problem domain of the framework, and then describes how to solve that problem. Each pattern has the same format. The format used in this paper is to first give a description of the problem in italics. This is followed by a detailed discussion of the different ways to solve the problem, with examples and pointers to other parts of the framework. The pattern ends with a summary of the solution, again in italics, followed by pointers to other patterns that are needed to fill it out and embellish it.

The appendix to this paper describes a set of ten patterns for HotDraw, a drawing framework. For example, pattern 4 is

> **Pattern 4: Complex Figures**
>
> *Some figures have a visual presentation with internal structure. For example, they may have attributes that are displayed by other figures. It should be possible to compose them from simpler figures.*
>
> Complicated figures like PERTEvent can be thought of as being composed of simpler figures. For example, a PERTEvent has a RectangleFigure and several TextFigures for the title, the duration, and the ending date. Complex figures like PERTEvent are subclasses of CompositeFigure. A CompositeFigure is a figure with other figures as components, and it displays itself by displaying its components. It has a bounding box that is independent of the bounding box of its components, and it will display its components only if they are inside of its bounding box. The selection tool and text tool will operation on its components when the left shift key is pressed. Custom tools can operate directly on the components, if you want.
>
> …
>
> *Complex figures should be a subclass of CompositeFigure, and figures that display one of its aspects should be a component of it.*
>
> To enforce constraints between the components of a complex figure, see **Constraints (5)**.

This pattern assumes that the reader has read other patterns. In this case, pattern 4 is preceded by pattern 2, which defines figures and bounding boxes. It also depends on pattern 1 to introduce the PERT chart example.

Patterns are problem-oriented, not solution-oriented. Each pattern describes how to solve a small part of the larger design problem. Sometimes the solution is to make a new subclass (*e.g.* pattern 2), sometimes it is to parameterize an object of an existing class (*e.g.* patterns 10), and sometimes it requires connecting several objects together (*e.g.* pattern 3).

2. Framework documentation

Documentation for a framework has three purposes, and patterns can help fulfill each of them. Documentation must describe

- the purpose of the framework
- how to use the framework
- the detailed design of the framework.

Patterns are best suited for teaching how to use a framework, but we will see that with care, a set of patterns can meet all three of these purposes for framework documentation.

2.1. Describing the purpose of a framework

A framework is a reusable design for solutions to problems in some particular problem domain. The purpose of the framework, i.e. the problem domain for which it is designed, must be the first thing that the documentation describes. If the framework turns out to be inappropriate then the reader does not have to continue reading. The first section should be small, so that every potential reader will have time to read it. It is not unreason-able to expect that an experienced designer will have read the first section of the documentation for every framework owned by his or her organization and so know the purpose of each one. This will reduce the problem of finding the right software to reuse.

Each pattern describes the problems it is meant to solve. The first pattern for a framework describes its application domain. It does this by giving examples, as do other ways of documenting frameworks. In addition, the first pattern introduces the rest of the patterns in the language, and it will usually tell which patterns should be read next. Thus, it acts both as a catalog entry for the framework and as a road map.

2.2. Describing how to use a framework

Next, the documentation for a framework should show how to use it to build applications. Most users of a framework want to know as little as possible about the framework. This means that they are not interested in a description of the design of the framework, but want a kind of cookbook, giving detailed instructions for using the framework. This is similar to the minimalist instruction of Caroll[Caroll][Rosson et. al.], which tries to show how to solve particular problems and make a system useful to a user as soon as possible.

Most documentation for frameworks describes how the framework works first, and then describes how to use it. However, part of the folklore of frameworks is that nobody understands a framework until they have used it, so using it is more important than reading about the theory behind it. In fact, I believe that theory should follow practice, that a discussion of the theory behind a framework is only understandable once the framework itself is understood, and that the best way to get a general understanding of a framework is to use it. Thus, we need to explain how to use a framework without explaining how it works.

This is not a new idea; the documentation for MacApp has long contained a cookbook [Apple] and the first documentation for Model-View-Controller (MVC) was called a "Cookbook for Model-View-Controller" [Krasner and Pope]. Patterns are more like the MacApp cookbook than the MVC cookbook. The MCV cookbook is a collection of examples, but is still a tutorial designed to be read as a unit. The MacApp cookbook is a collection of "recipes" with some cross references, where each recipe solves a particular problem like how to drag an object. It does not make use of structure as much as the set of patterns for HotDraw. None of these documents is as deterministic as a typical cookbook, which gives algorithms for making dishes, not just hints or rules of thumb.

2.3. Describing the design of a framework

The third purpose of documentation for a framework is to describe its design. This not only includes the different classes in the framework, but also the way that instances of these classes collabo-rate. Although programmers can usually interconnect objects without completely understanding how they work, and can even make subclasses following a cookbook, a framework is most useful to someone who understands it in detail.

The major weakness of cookbooks is that they describe the typical way that the framework will be used. A good framework will be used in ways that its designers never conceived. Thus, a cookbook is insufficient to describe every use of the frame-work. This is probably why the MVC cookbook also includes an informal description of the design of the framework, as well as instructions on how to use it[Krasner and Pope]. On the other hand, someone's first use of a framework usually fits the stereotype of the framework's designer, and using a framework helps provide the intuition needed to read a more formal specification.

In contrast to formal specifications like contracts [Helm et. al.], patterns are an informal technique aimed primarily at describing how to use a framework, not describing its algorithms, patterns of collaboration, or shared invariants. However, patterns provide many opportunities for describing a design, and it is possible to include much of the design of a framework in patterns for it. The section between the statement of the problem to be solved and the summary of the solution elaborates upon the solution and explains why it works. This section usually describes part of the design of a framework. If the patterns are designed carefully then they can also describe the design of a framework.

Where should design information be placed in a set of patterns? In general, detailed design information should be hidden as long as possible from the reader, because most readers are not interested in seeing it. On the other hand, it is necessary to make sure that the patterns contain the information somewhere and that the information is not duplicated. The technique that I used was to treat the set of patterns as a directed graph, using the references from one pattern to another as the edges, and to place design information as far from the first pattern as was feasible. Some design information, such as the fact that a HotDraw application is made up of Figures, Drawings, Handles, and Tools, must be described in the very first pattern, but most of it can be moved to subsequent patterns without using forward references.

3. The Role of Examples

Examples play a key role in the documentation of frameworks. The documentation for MVC [Krasner and Pope], MacApp [Apple], and Unidraw [Vlissides] all include many large examples. Examples make frameworks more concrete, make it easier to understand the flow of control, and help the reader to determine whether he or she understands the rest of the documentation.

Examples are also important in documentation based on patterns. Both the problem that a pattern is supposed to solve and its solution are often described in terms of examples. This is most obvious when describing the initial pattern, the one that describes the purpose of the framework. It is usually hard to specify a problem domain precisely, but a small set of examples usually makes the general area clear. These examples are not intended to show how to use the framework to build applications, nor to explain the design of the framework, but rather to show what the framework is good for. Thus, the examples for a user interface framework will be finished user interfaces, while the examples for a compiler framework will be finished compilers. The examples will point out user-level features that the framework provides, but they will not explain how these features are provided or how the application designer uses them.

Examples can also be used to show features of the design. The patterns for HotDraw use examples to motivate the need for complex figures (pattern 4) and for constraints (pattern 5). These patterns start with example applications of HotDraw and finish with a discussion of the code needed to implement these applications.

Examples can be used to test drive the patterns. The examples should exercise all the patterns, so someone can make sure that they learn all the patterns by trying out all the examples. The examples that have been used to describe the purpose of HotDraw (i.e. the examples in the first pattern) are sufficient to test drive all the patterns.

Not only does the documentation for HotDraw use examples heavily, but HotDraw comes with several applications that show concrete examples of applying the framework to a particular problem. Studying working examples is a time-honored way of learning a framework, and patterns are a way of reinforcing, encouraging, and system-atizing this method, not a replacement for it.

It is interesting to note that examples also play a key role in design (*c.f.* use cases [Jacobson] or walk-throughs [Wirfs-Brock *et. al.*]). This is just another example of the general rule that concrete examples are easier to understand than abstractions.

4. Patterns for HotDraw

Most of this paper is an appendix that contains an example of patterns for a framework. The framework is HotDraw, which is a framework for semantic graphic editors that was originally developed at Tektronix by Kent Beck and Ward Cunningham. It has been reimplemented several times. The latest version, and the one that the patterns describe, was written by Patrick McClaughry for ParcPlace ObjectWorks for Smalltalk-80 release 4.0. This implementation was simpler than previous ones because the design of the user interface framework for Smalltalk-80 release 4.0 was influenced by HotDraw.

HotDraw is not unique as a graphics editor framework. Unidraw is a similar framework for the C++/X windows environment that is built upon the InterViews user interface framework [Vlissides]. DRAW_Master is a graphics editor framework for the C++/OS 2 and Windows environment that is built upon the GUI_Master application framework[Veltman and Riksen]. HotDraw is simpler and less powerful than the others, but this simplicity makes it a better example for a paper like this.

The first pattern describes the purpose of HotDraw. The patterns form a directed graph, with the first pattern as the entry point. Other patterns are arranged so that those closest to the first pattern are the ones that are most often used. For example, the second pattern describes how to make subclasses of Figure, which is something that almost every user of HotDraw will need to know. The third pattern describes (among other things) how to install a handle on a figure, but the sixth pattern describes how to make new kinds of handles, which is usually unnecessary.

The patterns assume that the reader knows Smalltalk. Some of the patterns contain fragments of Smalltalk code. They also assume that the reader is using Smalltalk, so they often refer to examples in HotDraw and imply that the reader can quickly look them up using the browser. This suggests that patterns would work well with an on-line documentation system, perhaps one that is based on hypertext.

5. Reflections

The set of patterns got its start at the workshop on "Towards an Architecture Handbook" organized by Bruce Anderson at OOPSLA'91. Kent Beck and I shared an interest in both HotDraw and Alexander's pattern language, so we took the opportunity to write some patterns for HotDraw. Later that fall I gave the patterns to some graduate students learning HotDraw, but the patterns were not helpful. At the time each pattern was only a paragraph, and both the examples and the theory of design of HotDraw were separated from the patterns.

I studied Alexander's pattern language again, and realized that each pattern relied on examples and that he provided explanations for his patterns in terms of underlying theories of the properties of building materials or social interaction. When I integrated the examples with the patterns I found that the examples did not illustrate all the patterns, so I created another one or two examples. I put enough of the theory of design in the first pattern to define the main vocabulary and put the rest as far down in the patterns as possible, but far enough forward to both avoid duplication and to make sure the information was present before it was needed.

The next group of people who read the patterns knew Smalltalk well, but most did not know HotDraw. Their feedback provided a few changes, but after a few iterations they were satisfied. The patterns are the only documentation for a version of HotDraw that has been distributed since early 1992, and users say they are satisfied with it. Since there are no complaints, it is hard to improve the patterns, which shows the weakness of this kind of informal testing. It would probably be worthwhile to try out the patterns in a controlled setting where it would be possible to watch how people use the patterns and what aspects of HotDraw are hard to learn.

Writing the patterns was not hard once I figured out the format. Picking good examples is never easy, but the patterns showed the features that they needed to demonstrate. Probably the hardest part of writing a set of patterns for a framework is knowing the framework well enough to know how likely a feature is to be customized.

The structure of the patterns seemed to help design them. Thinking about patterns as a directed graph made it easier to decide where to place examples and design information, and made it easier to analyze the patterns for completeness.

Designing patterns for HotDraw took a lot of work, but it also helped clarify the design of the framework, so it was a useful exercise in itself. Designing patterns requires analyzing how people are expected to use the framework, and this points out ways that using it is awkward.

Although the HotDraw patterns are easy to follow on paper, patterns might be even more effective as hypertext. A hypertext system could make it easier to follow links to patterns, to show more examples, to hide technical design explanations from casual readers, and to separate the various parts of a pattern. Several people who read earlier versions of this paper commented that patterns are similar to hypertext documentation systems that they have seen, though I have little experience with hypertext systems. The HotDraw patterns were an imitation of Alexander's pattern language, not hypertext or the MacApp cookbook. It is interesting that the final result is so similar to them.

Conclusion

Patterns are a good way to describe frameworks because first-time users of a framework will usually not want to know exactly how it works, but will only be interested in solving a particular problem. As long as the patterns are powerful enough to describe most initial uses of the framework, it will meet the needs of the users.

A set of patterns for a framework will be sufficient documentation for many of the framework's users. Only those who want to go outside the usual bounds of the framework or those with an emotional need to know how the framework works in detail will need to read traditional design documentation. The internal structure of a set of patterns minimizes the amount that has to be read to solve a problem, and also provides places to put design information.

This paper is just a first step in creating a method for documenting frameworks. In the future, patterns should be tested against other ways of documenting a framework to see which best helps users build applications. Patterns should be used for larger frameworks to see whether they scale well, though Alexander's work suggests that they will. Although just a first step, the patterns for HotDraw in the appendix show that patterns are an attractive possibility for solving the problems of documenting frameworks.

Acknowledgements Kent Beck and Ward Cunningham not only invented HotDraw, they also introduced me to Alexander's pattern languages. This paper also owes a great deal to those who read it and provided it with much-needed criticism, including Bruce Anderson, Kent Beck, Bob Hinkle, Brian Foote, Pat McClaughry, Carl McConnell, Bill Opdyke, Don Roberts, Michael Seif, Dan Walkowski, and the reviewers.

Bibliography

[Alexander et. al.] Christopher Alexander, Sara Ishikawa and Murray Silverstein, with Max Jacobson, Ingrid Fiksdahl-King and Shlomo Angel. *A Pattern Language.* Oxford University Press, New York, 1977.

[Apple] *MacApp Programmer's Guide.* Apple Computer, 1986.

[Carroll] John M. Carroll. *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill.* MIT Press, Cambridge, Mass., 1990.

[Deutsch] L. Peter Deutsch, "Design Reuse and Frameworks in the Smalltalk-80 Programming System", pages 55-71, *Software Reusability, Vol II*, ed. Ted J. Biggerstaff and Alan J. Perlis, ACM Press, 1989.

[Jacobson] Ivar Jacobson, "Object Oriented Development in an Industrial Environment", Proceedings of OOPSLA '87, pages 183-191, October, 1987.

[Johnson and Foote] Ralph E. Johnson and Brian Foote, "Designing Reusable Classes" *Journal of Object-Oriented Programming,* 1(2):22-25, 1988.

[Krasner and Pope] 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.

[Helm et. al.] Richard Helm and Ian M. Holland and Dipayan Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", Proceedings of OOPSLA '90, pages 169-180, October 1990.

[ParcPlace] User's Guide to Objectworks\Smalltalk Release 4. ParcPlace Systems, 1990.

[Rosson et. al.] Mary Beth Rosson, John M. Carroll, and Rachel K.E. Bellamy, "Smalltalk Scaffolding: A Case Study of Minimalist Instruction", In *Proceedings of CHI'90,* pages 423-429, May 1990.

[Veltman and Riksen] B.W.J. Veltman and A.J.O.M. Riksen, "DRAW_Master, a new branch of the GUI_Master class tree", *Journal of Software Research,* special issue, 14-23, Vleermuis Software Research, December, 1991

[Vlissides] John M. Vlissides, *Generalized Graphical Object Editing,* PhD Thesis, Stanford University, 1990. Also technical report CSL-TR-90-427.

[Wirfs-Brock et. al.] Rebecca J. Wirfs-Brock, Brian Wilkerson, and Lauren Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood CLiffs, NJ, 1990.

Appendix Patterns for HotDraw

Pattern 1: Semantic Graphic Editors

HotDraw is a framework for structured drawing editors. 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.

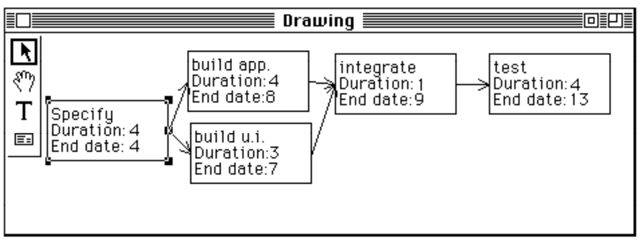

Figure 1 - PERT chart editor

Many programs need to edit two-dimensional drawings such as schematic diagrams or blueprints. Sometimes the elements of these drawing can be treated independently, but often the elements have constraints between them. For example, Figure 1 shows a PERT chart consisting of rectangles representing events and arrows between the rectangles representing dependencies between events. Events display duration and ending dates, and have starting dates that are not displayed. The starting date of an event is the maximum of the ending dates of all the events that precede it, *i.e.* that have a connection to it, and the ending date of an event is the sum of its starting date and its duration. Changing the duration of one event will cause its ending date to change. This may cause the starting dates of other events to change, and so the ending dates, as well.

Direct manipulation techniques are usually the best way to edit a drawing. A user will manipulate an element of a drawing by using the mouse to select the element and then operating upon it. For example, when an event in the PERT chart is selected (e.g. the left-most event-rectangle in Figure 1), it presents a set of handles. Dragging the corner handles will change the size of the event's rectangle, and selecting the handle on its right side will create an arrow that can be dropped on another event to indicate that the first event must precede the second. Moving an event or changing its size will cause the arrows connected to that event to move, too.

There are many ways to operate upon an element. One way is to move a handle that controls an attribute of the element. For example, events could have handles that control durations, and the duration would then be changed by moving the handle. A second way is to pop up a menu of operations on the element and select one. Events could have a “change duration” menu item, which would prompt for a new duration. A third way is to choose a specialized tool from the palette to change the element's attribute. For example, the text tool can edit text, so a good way to change the duration and title of an event is to edit them with the text tool. In the case of the PERT chart editor, the text tool is the best way to change the duration, but handles are the best way to change the size of the display of an event, and menus are the best way to cut and paste events. The designer of a drawing editor needs a wide range of options, but must pick the most appropriate ones to keep from confusing the user.

The palette on the left of the editor in Figure 1 offers four tools: a selection tool (which is currently active), a scrolling tool, a text tool, and a tool for creating new events. Selecting a tool with the mouse makes it active.

HotDraw is a framework for semantic drawing editors (i.e. editors for drawings whose elements have constraints on their behavior) that was used to build the PERT chart editor of Figure 1. The most important class in HotDraw is Figure, which is the class of drawing elements. Figures are responsible for rendering, hit-testing, and notifying dependents when their appearance changes. Other important classes are Drawing (which represents the entire drawing), Tool, Handle, and DrawingEditor.

Drawing editors created with HotDraw can be part of a larger application. For example, Figure 2 shows a network editor. The top pane in the editor is a HotDraw drawing of a network. The left two panes on the bottom select two of the nodes, while the bottom right pane lets the user display and change a weight on the edge between the two nodes. The network is drawn so that nodes are close together if they have a lot of communication between them, i.e. if their edge has a large weight. Adding and deleting nodes and edges is done using the three lower panes, but the user can rearrange (within limits) the nodes in the drawing directly. Note that the network editor has no palette. All the user can do with the drawing directly is drag nodes.

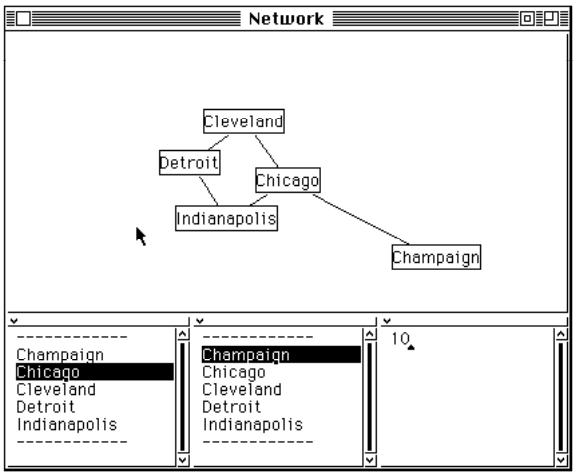

Figure 2 - A Network Editor

The network editor is also an example of how drawings can be animated. It treats the graph layout problem as a variation of the n-body problem, and solves it iteratively. Each node has a repulsive force on every other node, but edges act like springs to keep nodes together. The drawing is animated by showing intermediate steps in the n-body solution. This lets the user rearrange nodes if the network is in a suboptimal, but stable, configuration.

DrawingEditor new open will create the drawing editor shown in Figure 3, which is the default version of HotDraw. This is what you will get if you don't redefine the tools that the drawing editor has in its palette.

Figure 3 shows the default drawing editor. Most of the tools in the palette on the left create different kinds of figures. The exceptions are the first two, which are the selection tool and the scrolling tool, the third and fourth, which rearrange the order of figures by bringing a figure to the front or to the back, and the fifth, which deletes figures. It is probably a good strategy to play with this editor for a while to learn the user interface. The menu on selected figures provides commands to cut and paste them, but all other changes to figures are performed by manipulating their handles.

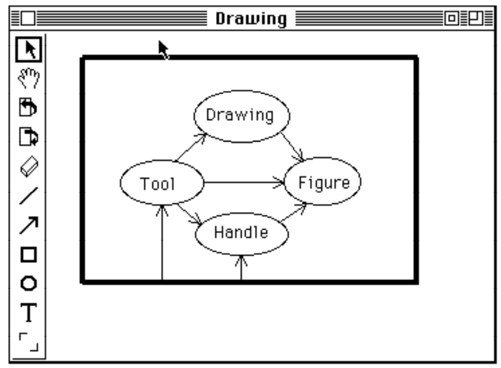

Figure 3: Default drawing editor

To design a drawing editor using HotDraw, first list the atomic elements of the diagram. Each one will be a subclass of Figure. Decide whether the drawing needs to be animated. List the tools that will be in the palette and make a subclass of DrawingEditor with a ***tools*** *method that returns an array of the tools making up the palette.*

To create a subclass of Figure, see **Defining drawing elements (2)**.
To animate a drawing, see **Animating drawings (9)**.
If the drawing editor is going to be part of a larger tool, see **Embedding a drawing in another tool (7).**
To put a tool in the palette, see **Tools(8)**.

Pattern 2: Defining Drawing Elements

There are an infinite variety of primitive figures that can be included in a drawing. Thus, there needs to be a way to make new figures for each application.

Each kind of drawing element is a subclass of Figure. Note that there are already subclasses of Figure for the simple geometric objects, such as EllipseFigure, RectangleFigure, and LineFigure. Class CompositeFigure is the superclass of complex figures liked PERTEvents, which can contain other figures as components. Sometimes these classes are suitable to be used directly. For example, the lines connecting events in the PERT chart are just instances of LineFigure. Often it is necessary to make a subclass of Figure or one of its existing subclasses.

Most figures will be either CompositeFigures or subclasses of existing geometric figures like RectangleFigure or LineFigure, so they will inherit the visual presentation of their superclass. A subclass of Figure that is not a subclass of any other subclass must define its own visual presentation. Figure is a subclass of VisualComponent, so it must define **displayOn:**. It must also define **origin**, **extent**, and **translateBy:**. These are the only methods necessary to render and move a figure, and are the minimum methods necessary to define a new subclass of Figure.

The main distinction between Figure and other subclasses of VisualComponent is that a figure keeps track of other objects that depend on it. This has no effect on methods that do not change the state of a figure, such as **origin**, **extent**, and **displayOn:**. However, methods that change some attribute of a figure must notify the objects that depend on it. This is done by sending the **willChange** message to itself before changing the attribute, and sending the **changed** message to itself afterwards.

For example, EllipseFigure has an instance variable "ellipse" that contains a rectangle representing its bounding box. It implements **origin** and **extent** by returning the origin and extent of ellipse. It implements **translateBy:** as

```
translateBy: aPoint
    self willChange.
    ellipse := ellipse translateBy: aPoint
    self changed
```

In HotDraw, the dependents of a figure are almost always constraints (to create a constraint, see **Constraints(5)**).

Most figures have other attributes. For example, geometric figures like RectangleFigure and EllipseFigures have an interior color, a border color, and a border width. PERTEvents have duration, ending dates, and titles. Changing any of these attributes also requires notifying the dependents of the figure. An easy way to ensure that a figure gets the **willChange** and **changed** messages whenever it changes one of its attributes is to change the attribute in only one place, and to have that method send the messages to itself.

Each drawing element in a HotDraw application is a subclass of Figure, and must implement displayOn:, origin, extent, and translateBy:. In addition, a subclass of Figure can implement any method needed by the application.

To write **displayOn:**, see [ParcPlace 90][1].
To let the user change the size or the color of a figure, see **Changing drawing element attributes (3)**.
To see how to implement complex figures like PERTEvents, see **Complex Figures (4)**.
To enforce constraints between different figures, see **Constraints (5)**.

Pattern 3: Changing drawing element attributes

There are at least three ways to edit a figure's attribute; with a handle, with the figure's pop-up menu, or with a special tool. Each technique is appropriate in different cases.

The size of a figure and other numeric attributes are best edited with handles. Textual attributes like names, or numeric attributes that must be precise like dates, are best edited by displaying the text as part of the figure and letting the user edit it with the TextTool. Use of the TextTool implies that the figure is a CompositeFigure (see **Complex Figures (4)**).

A figure's **handles** method returns a collection of handles on the figure. The **handles** method for Figure returns resizing handles on the four corners, so it is common for a **handles** method to call it (with `super handles`) and add more handles. SelectionTrackHandle has class methods that create customized handles. For example, **colorOf:** will change the inside color of a figure, **borderColorOf:** will change the color of the border, and so on. For good examples of a **handles** method, see LineFigure or RectangleFigure.
A figure's menu is returned by its **menu** method. By default, an operation in the menu is sent to the

[1]Ideally this reference would be to another pattern, but the set of patterns for a complete system will be very large. Providing a pointer to existing documentation is an easy way to reduce the number of patterns.

DrawingView. Operations whose selectors are in a collection returned by the **menuBindings** method of a figure are instead sent to the figure. The default menu defined in Figure implements cut and paste. See Figure for a simple example and TextFigure for a more complex example with hierarchical menus.

List the attributes of the drawing element that you want to edit. For each attribute, decide whether to edit it with a handle, an operation from the menu, or a tool, and update the ***handles*** *method, the* ***menu*** *method, or the list of tools in the drawing editor.*

To make new kinds of handles, see **Handles (6)**. To make new kinds of tools, see **Tools(8)**.

Pattern 4: Complex Figures

Some figures have a visual presentation with internal structure. For example, they may have attributes that are displayed by other figures. It should be possible to compose them from simpler figures.

Complicated figures like PERTEvent can be thought of as being composed of simpler figures. For example, a PERTEvent is a RectangleFigure with several TextFigures for the title, the duration, and the ending date. Complex figures like PERTEvent are subclasses of CompositeFigure. A CompositeFigure is a figure with other figures as components, and it displays itself by displaying its components. It has a bounding box that is independent of the bounding box of its components, and it will display its components only if they are inside of its bounding box. The selection tool and text tool will operation on its components when the left shift key is pressed. Custom tools can operate directly on the components, if you want.

In general, a figure should be a subclass of CompositeFigure whenever one of its attributes will be edited directly by a tool. The most common example is that an attribute is a string, and must be edited with the text tool. Instead of storing the text attribute in an instance variable, store it in a TextFigure. Do this by first ensuring that the attribute is read and written only by a pair of accessing methods. Instead of a string-valued instance variable, make a TextFigure-valued instance variable, and make the string's accessing methods read and write it from the TextFigure.

This can be generalized for any kind of attribute that is represented by another figure. The attribute should be stored in the component figure, changes to the attribute result in changes to the figure, and changes to the figure result in changes to the attribute. If changes to one component might effect others then constraints should be used. (See **Constraints (5)**).

The **initialize** method of the complex figure must create the figure representing the attribute and add it to the complex figure. It may also need to create constraints. PERTEvent is a good example.

Complex figures should be a subclass of CompositeFigure, and figures that display one of its aspects should be a component of it.

To enforce constraints between the components of a complex figure, see **Constraints (5)**.

Pattern 5: Constraints

Often an attribute of one figure is a function of the attributes of other figures. For example, in the PERT chart editor of Figure 1, the starting date of an event is the maximum of the ending dates of all the events that precede it, i.e. that have a connection to it. As another example, the endpoints of the lines connecting events depend on the locations of the events. Handles also depend on the figure to which they are attached.

Smalltalk has a standard way of keeping track of dependences between objects. Each object has a collection of dependents (usually empty), and the **addDependent:** and **removeDependent:** messages update this collection. Sending the **changed:** message to an object will cause the **update:** message to be sent to all its dependents. This same mechanism is used to maintain constraints in HotDraw.

HotDraw extends the standard Smalltalk class library by making constraints be objects. Traditionally each object that can depend on another defines its own update method. The problem with this is that many update methods are similar, so there is a lot of duplicated code. Making constraints be objects means one figure does not directly depend on another. Instead, the dependents of an object are always constraints, and updating a constraint will cause it to modify other figures in response.

Many of the HotDraw figures have the constraints that they need built automatically. For example, a line that connects two figures is just a LineFigure, except that there are two constraints that are created along with it. The first constraint depends on the starting figure and updates the starting point of the LineFigure, and the second depends on the ending figure and updates the ending point of the LineFigure. These constraints do nothing more than keep track of the two figures and the messages to be sent to get a position from one figure and to change the position of the other. These constraints are created by the LineFigure class **startLocation:stopLocation:** method, which creates a line connecting two figures.

Other times you will need to create constraints for a figure. There are two standard classes of constraints, PositionConstraint and MultiheadedConstraint. A PositionConstraint is typically used to maintain a constraint between the position of two figures. For example, it is used to connect a line to other figures. A MultiheadedConstraint depends on a set of figures.

Both kinds of constraints are customizable. A PositionConstraint refers to a position on a figure with a Locator, and also knows a message that it sends to the dependent object when the figure that it depends on changes. A Locator refers to a position on a figure by keeping track of the figure and knowing the message to send to the figure to learn its position. The “bounding box accessing” protocol of class Figure defines methods to compute many positions, such as **center**, **left**, **bottomRight**, etc. A MultiheadedConstraint is created with a block that it evaluates whenever any one of the figures that it depends on changes.

It is not hard to create new kinds of constraints. A constraint must define the **update:** method, but otherwise its design is entirely up to you. Fortunately, simple uses of HotDraw will not require using constraints explicitly.

Each constraint in a drawing should be represented by a constraint object.

To add constraints between figures automatically when you add lines between them, see **Adding Lines (10)**.

Pattern 6: Making new kinds of handles

Handles vary in behavior. Sometimes handles change the size of a figure, sometimes they change its color, sometimes they create new lines. In general, one could imagine pressing on a handle performing any kind of operation. Moreover, handles can be attached to any part of a figure, and they move when the figure moves. This means that handles must be parameterized in some way.

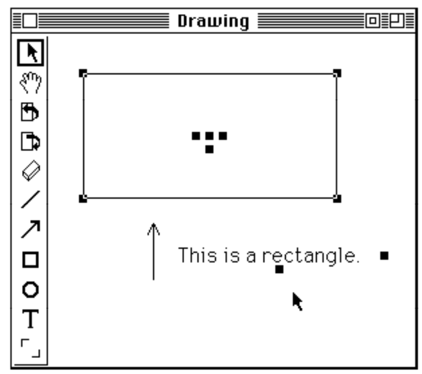

Figure 4 - Handles on Two Figures

Figure 4 shows some of the different kinds of handles in HotDraw. It has three figures in it, and the rectangle figure and the text figure have been selected and so have handles. The interior handles in the rectangle change its inside color and the color and the width of its border. Its corner handles change its shape. The right-most handle on the text figure controls the width of the text field and the bottom handle controls the font size of the text. All the handles shown, except the one in the center of rectangle, are instances of SelectionTrackHandle. Like most of the Handle subclasses, SelectionTrackHandle is parameterized and so a new subclass rarely needs to be written.

SelectionTrackHandle changes an aspect of each selected figure, including its own. It is a subclass of TrackHandle, which only changes an aspect of its own figure. Both kinds of handle are parameterized by a message that the handle sends whenever it is dragged, with an argument that is the distance that the handle has been moved. Nearly any numeric attribute can be edited by a TrackHandle or a SelectionTrackHandle. The only precondition is that there must be a message to set the numeric attribute.

The expression **SelectionTrackHandle widthOf: aFigure** will create a handle that can change the width of a RectangleFigure or an EllipseFigure. It does this by sending the **borderWidthBy:** message to the figure, as can be seen in the definition of the **widthOf:** method, which specifies that the handle will be -10@0 off-center of the figure, and will only "sense color" (which means to use just the y-value of the distance that the handle has been dragged as an argument to the **borderWidthBy:** message).

```
widthOf: aFigure
    ^self
        on: aFigure
        at: #offCenter:
        with: -10@0
        sense: #senseColor:
        change: #borderWidthBy:
```

The handle in the center of the rectangle of Figure 4 is a ConnectionHandle, which is a handle that creates a line from its figure to another figure. A ConnectionHandle can be parameterized with a block taking the source and destination figures as arguments; that block is evaluated when the figure is created. See **Adding lines (10)**.

If you do need to make a new subclass of Handle, you only need to define one method, **invoke:**. The argument of **invoke:** is a DrawingView, since it needs to access the controller (and hence the mouse), the drawing, and the display. Since **invoke:** gets sent whenever the user presses the mouse button on a handle, it is possible to make handles with any kind of behavior.

For example, Figure 5 shows the diagramming inspector, a HotDraw application that displays the interconnections between objects. Each box is an instance of ObjectFigure, and represents an object. When an ObjectFigure is selected, it presents handles for each instance variable that is non-nil. Clicking on one of these handles creates a new ObjectFigure representing the value of the instance variable, unless that object already has an ObjectFigure, in which case a line is drawn to it.

ObjectFigure requires a new kind of handle, one that is activated by clicking, not dragging. ReferenceHandle is a subclass of Handle that implements these new handles. It redefines the **invoke:** method to first find an ObjectFigure for the object, creating a new one if necessary. It then creates a line to the ObjectFigure and adds the line to the drawing. It takes several methods to implement all of this.

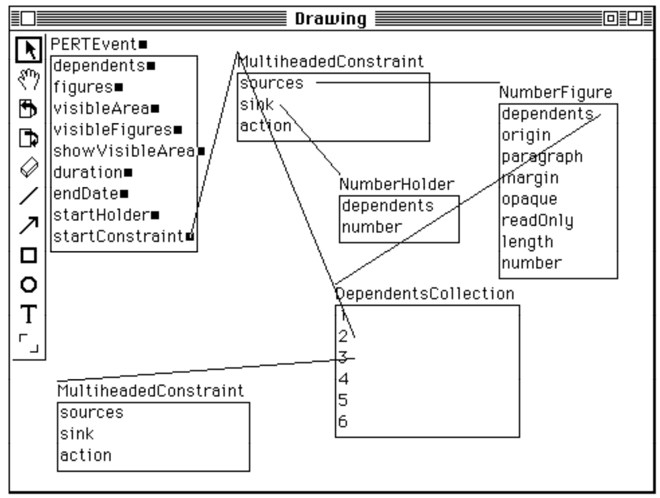

Figure 5 - The Diagramming Inspector

A handle that changes only an aspect of its figure when it is dragged is an instance of TrackHandle. A handle that changes an aspect of all the selected figures when it is dragged is an instance of SelectionTrackHandle. A handle that creates a line from its figure to another figure is a ConnectionHandle. Handles that perform other functions will need to be new subclasses of Handle.

To create a ConnectionHandle, see **Adding lines (10)**.

Pattern 7: Embedding a drawing in another program

Drawings are often part of a complex user interface that includes text panes, buttons, lists, and so on. HotDraw is built on top of the Smalltalk-80 Model/View/Controller framework, and so should be able to be a small part of a complex application.

In addition to Drawing and its components, a HotDraw application will have a DrawingView, a

DrawingController, and a DrawingEditor. The DrawingView and DrawingController are designed to fit into the Model/View/Controller framework and are rarely customized, so little needs to be said about them. The DrawingEditor is the model, and is responsible for keeping track of the drawing, the set of tools, the current tool, the menu of operations on the drawing (in distinction to the menus of operations on figures, for which each figure is responsible), and many of the operations on drawings. For example, the standard DrawingEditor class has methods for reading and writing drawings to files.

For simple applications of HotDraw, a subclass of DrawingEditor only needs to redefine the tools (See **Tools(8)**). For example, that is all that is needed for the PERTChart. The Diagramming Inspector of Figure 5 does not even have its own subclass of DrawingEditor, but just uses the default palette. However, complex user interfaces like that of the network editor of Figure 2 require more complex subclasses. The editor must not only support DrawingEditor protocol, it must also support the other panes of the user interface.

If an application only displays one drawing at a time then the model should be a subclass of DrawingEditor, but if there is more than one drawing then it is better to make the model be a completely new class. DrawingView is a pluggable view (see [ParcPlace]) so it can be parameterized with the messages to get the tools, menu, and drawing from the editor. To see how to parameterize a DrawingView, look in the instance creation protocol of the class methods of DrawingView.

The DrawingEditor is responsible for creating the user interface. This usually is nothing more than creating a top-level window and embedding the palette and the DrawingView within it. However, a complex application like the network editor of Figure 2 requires a method for creating all the panes and connecting them together. A good example is the **open** method of NetworkEditor. For more on how to embed a DrawingView in a larger application, see Chapter 17 of [ParcPlace].

*Make a drawing part of a complex application by making a DrawingView be a subview of the application's view and giving the DrawingView a model that responds to DrawingEditor protocol, i.e. that implements the **currentTool**, **menu**, **drawing**, and **drawing**: methods.*

Pattern 8: Tools

Tools represent the modes of the user interface to a drawing. Selecting a tool from the palette lets the user manipulate figures, create new figures, or perform operations upon a figure or the entire drawing. An important part of designing an editor using HotDraw is to design the set of tools that will be on the palette.

The **tools** method in DrawingEditor returns an ordered collection of tools that makes up the palette. These tools usually include the selection tool, creation tools for each drawing element that the user will create with a creation tool, a tool to scroll the drawing, and tools to move figures from front to back and from back to front. There are standard Tool subclasses for these tools, but it is also possible to define new subclasses of Tool.

Several of the subclasses of Tool are parameterized, so new subclasses of Tool are rare. CreationTool is parameterized by the class of the figure to create, the icon to display in the palette, and the cursor. Most subclasses of Figure define a class method **creationTool** that returns an initialized CreationTool that can be installed in the palette. EllipseFigure and RectangleFigure both contain good examples.

Two more parameterized classes of tools are DrawingActionTool, which performs an action on the drawing, and FigureActionTool, which performs an action on a figure. A DrawingActionTool is parameterized with two blocks, one that is evaluated when the tool is selected and the other when it is deselected. On the other hand, a FigureActionTool is parameterized with a single block that is evaluated when the mouse is clicked on a figure. DrawingActionTool has class methods **loadTool** and **saveTool**, which return tools to read and write the current drawing to a file, respectively. FigureActionTool has class methods that return tools to bring a figure to the front of the drawing, that send it to the back, and the delete it. To make other tools that perform single actions, use these methods as models.

*The **tools** method of the DrawingEditor defines the tools that will be on the palette by returning an* *ordered collection of tools, which are instances of subclasses of Tool.*

Pattern 9: Animating drawings

Constraints, handles and tools let a drawing react to a user, but cannot give a drawing a life of its own. Animation requires a controlling object to direct all the figures in a drawing.

Animation is provided in HotDraw by making a subclass of Drawing that defines the **step** method. This is the main reason that Drawing is subclassed. A drawing is repeatedly sent the **step** message whenever HotDraw is running. The purpose of the **step** method is to move each of the drawing's figures. For example, MovingDrawing simulates the n-body problem by giving each figure a velocity and assuming that figures exert forces on each other. Its **step** method is

```
step
     animating ifFalse: [^super 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 calculateForceFrom:
                       self figures] .
 "Last, move each figure."
 self figures do: [:fig1 | fig1 step]
```

Typically there is some way to turn animation off, in this case by setting the "animating" variable of the moving drawing to false. The easiest way to set this variable is by a tool. The tool should be an instance of DrawingActionTool that is parameterized by a block that sends the **startAnimation** message to the drawing when the tool is selected and a block that sends **stopAnimation** when the tool is deselected. (See **Tools (8)**).

*Animate a drawing by making a subclass for it that defines the **step** method to perform the next step in the animation.*

Pattern 10: Adding lines

Lines are often used to connect figures. Sometimes these connections have no semantics, i.e. they are a by-product of other actions and deleting or moving them does not affect other figures. However, sometimes adding a line will result in constraints being added or other figures created.

The best way to create a connecting line from one figure to another is to add a ConnectionHandle at the point from which the line starts. By default, this line has no semantics. However, a ConnectionHandle can be parameterized with a block that is evaluated when the line is added. For example, adding a line from one PERTEvent to another means that the destination PERTEvent depends on the source. The handle to create these lines is created (in the **handles** method of PERTEvent) by

```
handle := ConnectionHandle
                    on: self
                    at: #rightCenter
                    class: LineFigure.
handle
  action:
     [:source :dest |
       dest startConstraint
              addSource: source endFigure].
```

The action block depends on the fact that PERTEvents have startConstraints (to compute the starting date) that are MultiheadedConstraints (see **Constraints(5)**), and so understand the **addSource:** message.

The user should create lines that connect figures by pressing a ConnectionHandle on one of the figures. ConnectionHandles can be parameterized so that connecting two figures with a line can perform some other action on them as well, such as adding a constraint or testing whether a connection is allowed.

HotDraw is available by anonymous ftp from st.cs.uiuc.edu in /pub/st80_r4/hotdraw.