Designing Object-Oriented Frameworks

Garry Froehlich, H. James Hoover, Ling Liu, Paul Sorenson
Department of Computing Science
University of Alberta
Edmonton, AB. T6G 2H1
{garry,hoover,lingliu,sorenson}@cs.ualberta.ca

1 Introduction

Most software reuse has focused on code reuse, such as reusing parts of existing applications, reusing library functions or reusing pre-built components. With the recent interest in design patterns [Gamma et al., 1995] and object-oriented frameworks, the focus is shifting away from just reusing code to reusing existing designs as well. *Design patterns* provide a reusable piece of a design which solves a recurring design problem in software construction. An *object-oriented framework*, which is the focus of this chapter, is the reusable design and implementation of a system or subsystem [Beck and Johnson, 1994]. It is typically implemented as a set of abstract classes which define the core functionality of the framework along with concrete classes for specific applications included for completeness. Users of the framework complete or extend the framework by adding custom application specific components or functions to produce and application.

Designing a framework differs from designing a single application in at least two respects. First, the level of abstraction is different. Frameworks are meant to provide a generic solution for a set of similar or related problems or an entire domain, while applications provide a concrete solution for a particular problem.

Second, frameworks are by their nature incomplete. Whereas an application design has all of the components it needs to execute and perform its task, a framework design will have places within it that need to be instantiated by adding concrete solutions to a specific application problem. A framework does not cover all of the functionality required by a particular domain, but instead abstracts the common functionality required by many applications, incorporating it into the common design, and leaving the variable functionality to be filled in by the framework user.

Due to these differences, framework design focuses on providing flexible abstractions that cover the functionality required by applications within a domain and making the framework easy to use. These abstractions in turn provide ways in which application developers can customize the framework. Object-oriented technology is a natural fit for frameworks. Just as a subclass is a specialization of a parent class, an application can be thought of as a specialization of a more general framework. One of the ways to use a framework is to specialize the generic classes that are provided in the framework into application specific concrete classes.

There is yet to be an established standard for designing and developing frameworks. The purpose of this chapter is not to propose a standard, but instead to identify some of the key issues and techniques that affect framework design. Hopefully, as framework development becomes more common and better understood, standard approaches will emerge. We assume that the reader is already familiar with object-oriented design, so we will focus on the factors that differ between regular application vs. framework development. Several properties, such as ease of use and flexibility, have been identified in the frameworks literature as aiding the reuse of the framework. We discuss several techniques, such as the benefits of using inheritance, composition, and the use of hooks which help a framework attain these properties. Some key terms are defined in section 2 and the benefits of using frameworks are described in section 3. Sections 4, 5 and 6 forms the core of the chapter, describing the issues to consider when designing frameworks and methodologies for doing the actual design. Section 7 briefly discusses framework deployment issues. Section 8 summarizes the chapter and lists some of the open issues in framework design.

2 Concepts and Properties of Frameworks

Before discussing framework design, we define some concepts and terms associated with frameworks, starting with the roles involved in framework technology and a more in depth look at the parts of a framework.

2.1 Users and Developers of Frameworks

Three different roles can be associated with the development and use of frameworks:

- *Framework Designers* or framework developers, develop the original framework.
- *Framework Users*, sometimes called application developers or framework clients, use (reuse) the framework to develop applications.
- *Framework Maintainers* refine and redevelop the framework to fit new requirements.

The different roles are not necessarily filled by different people. Often the framework designer is also one of the framework users and framework maintainers.

2.2 Framework Concepts

Several different parts can be identified within an application developed from a framework as shown graphically in Figure 1. Applications are developed from frameworks by filling in missing pieces and customizing the framework in the appropriate areas. Application development is discussed in the chapter on using object-oriented frameworks.

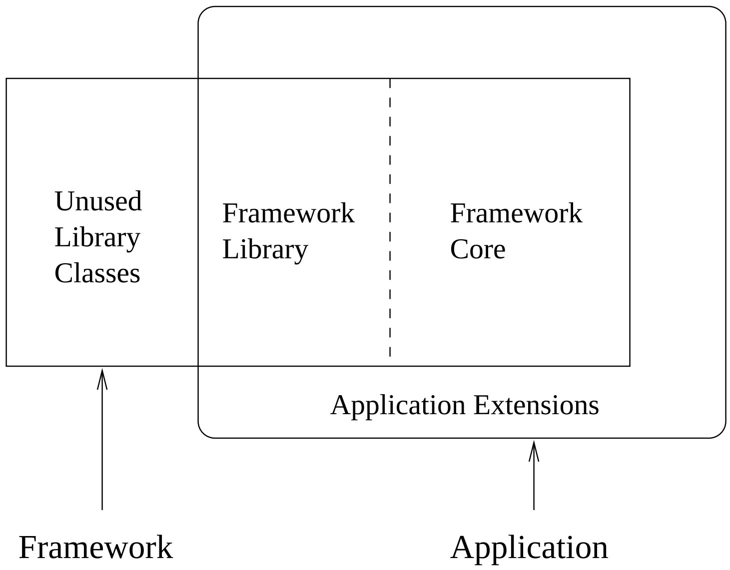

Figure 1: Application Developed from a Framework.

The parts of a framework are:

- Framework Core: The core of the framework, generally consisting of abstract classes, that define the generic structure and behavior of the framework, and forms the basis for the application developed from the framework. However, the framework core can also contain concrete classes that are meant to be used as is in all applications built from the framework.
- Framework Library: Extensions to the framework core consisting of concrete components that can be used with little or no modification by applications developed from the framework.
- Application Extensions: Application specific extensions made to the framework, also called an *ensemble* [Cotter and Potel, 1995].
- Application: In terms of the framework, the application consists of the framework core, the used framework library extensions, and any application specific extensions needed.
- Unused Library classes: Typically, not all of the classes within a framework will be needed in an application that can be developed from the framework.

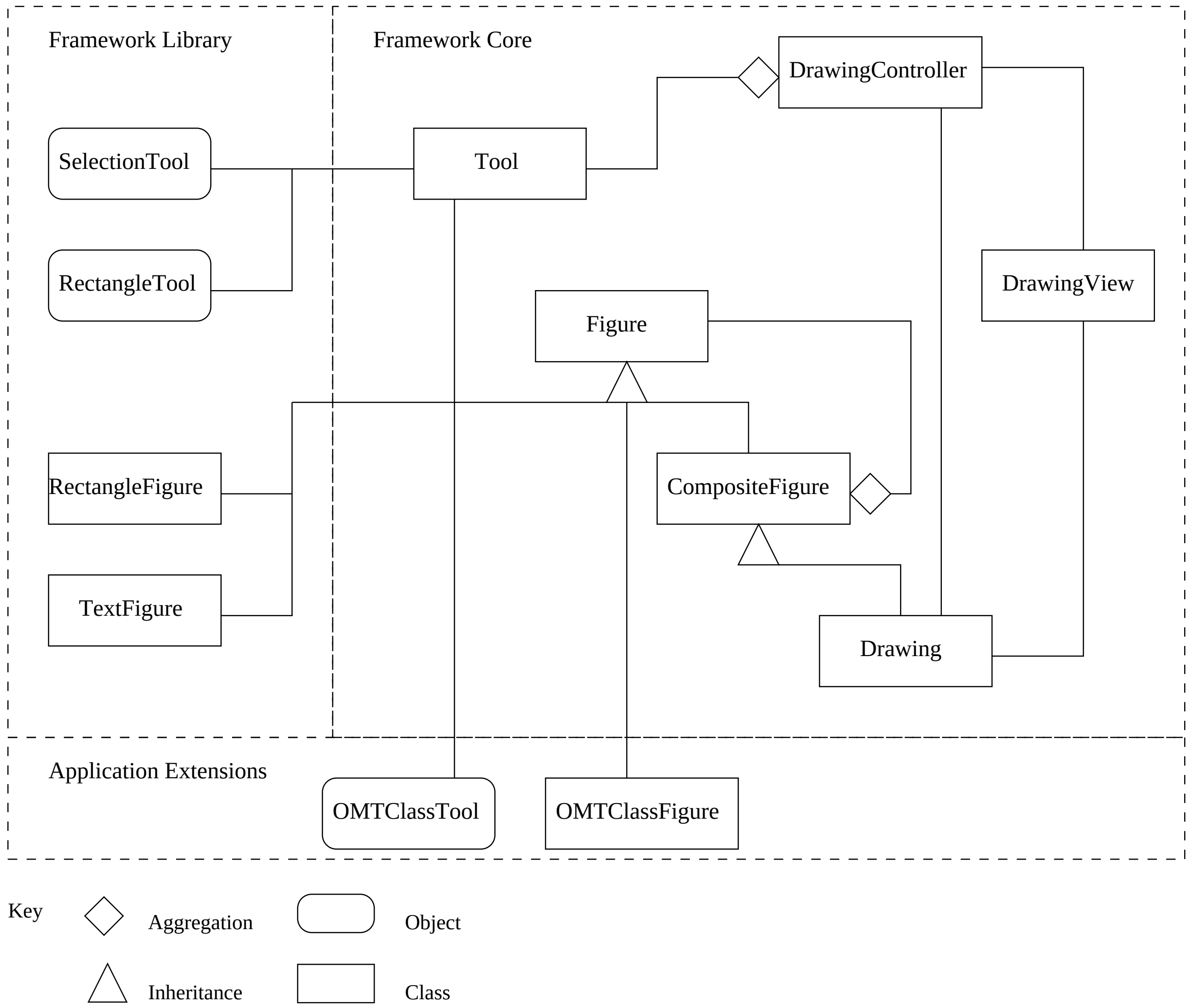

Figure 2: Simplified View of the HotDraw Framework.

A simplified example of the HotDraw framework [Johnson, 1992] is shown in Figure 2. HotDraw is a framework for building graphical editors, such as class diagram editors, written in Smalltalk.

The core of the HotDraw framework consists of classes that define the interactions and interfaces of the *key abstractions* of the framework. Some of the key abstractions in HotDraw are Figures (along with CompositeFigures and Drawings) which represent items within a diagram such as lines and boxes, Tools which plug into the DrawingController to handle user actions such as moving Figures around on the screen or creating new Figures, and DrawingViews which handle the display for the application.

Some of the pre-built components supplied with HotDraw in its framework library consist of SelectionTool for selecting and moving Figures and RectangleTool for creating RectangleFigures within a Drawing. These are tools that may be useful in any application, but do not necessarily need to be included in every application built from the framework. Common Figures are also provides, such as RectangleFigure for representing rectangles and TextFigure for text labels.

HotDraw applications often require new application specific Tools and Figures to be derived. As an example, two application specific classes are shown in Figure 2. The complete application allows Object Modeling Technique [Rumbaugh et al., 1988] class diagrams to be drawn and edited, but only two classes are shown here. OMTClassTool derives from the core Tool class and allows OMTClassFigures to be created and edited.

2.3 Hooks and Hot Spots

Application extensions are connected to a framework through *hooks*. Hooks are the places in a framework that can be adapted or extended in some way to provide application specific functionality [Froehlich et al., 1997]. They are the means by which frameworks provide the flexibility to build many different applications within a domain. For example, HotDraw has hooks for producing new types of Figures and Tools as well as more complex activities such as disabling the use of Tools to make a display-only application instead of an editor.

Hot spots [Pree, 1995], also called hinges [Cline, 1996], are the general areas of variability within a framework where placing hooks is beneficial. A hot spot may have many hooks within it. The area of Tools within HotDraw is a hot spot because different applications will use different tools. A Data Flow Diagram application will have tools for creating and manipulating the DFD that are standard for iconic interfaces, whereas a PERT chart application, because of its strict temporal ordering constraints, will likely have different creation and manipulation tools. There are a number of hooks within the Tool hot spot which define the various ways in which new tools can be defined.

In contrast, *frozen spots* [Pree, 1995] within the framework capture the commonalties across applications. They are fully implemented within the framework and typically have no hooks associated with them. In HotDraw, DrawingController is an example of a frozen class. The Tools used may vary, but the DrawingController remains a constant underlying mechanism for interation.

2.4 Framework Categorization

Several different means of classifying frameworks have been proposed. Here we present three relatively orthogonal views of frameworks. A framework can be categorized by its scope, its primary mechanism for adaptation and the mechanism by which it is used. The scope defines the area the framework is applicable to, whether a single domain or across domains. The adaptation mechanism describes whether the framework relies primarily upon composition or inheritance for reuse. Finally, the means of usage describes how the framework interacts with the application extensions; by either calling the application extensions, or having the extensions call the framework.

2.4.1 Scope

The scope of the framework describes how broad an area the framework is applicable too. Adair [1995] defines three framework scopes.

- *Application frameworks* contain horizontal functionality that can be applied across domains. They incorporate expertise common to a wide variety of problems. These frameworks are usable in more than one domain. Graphical user interface frameworks are a typical example of an application framework and are included in most development packages.
- *Domain frameworks* contain vertical functionality for a particular domain. They capture expertise that is useful for a particular problem domain. Examples exist in the domains of operating systems [Campbell et al., 1993], manufacturing systems [Schmid, 1995], client-server communications [Brown et al., 1995] and financial engineering [Eggenschwiler and Gamma, 1992].
- *Support frameworks* provide basic system-level functionality upon which other frameworks or applications can be built. A support framework might provide services for file access or basic drawing primitives.

2.4.2 Customization

The means of customizing is another way in which frameworks can be categorized. Johnson and Foote [1988] define two types of frameworks, white box and black box. While this is an important dichotomy, a framework will often have elements of both black box and white box frameworks rather than be clearly one or the other.

- *White box frameworks*, also called *architecture driven frameworks* [Adair, 1995] rely upon inheritance for extending or customizing the framework. New functionality is added by creating a subclass of a class that already exists within the framework. White box frameworks typically require a more in-depth knowledge to use.
- *Black box frameworks*, also called *data-driven frameworks* [Adair, 1995], use composition and existing components rather than inheritance for customization of the framework. Configuring a framework by selecting components tends to be much simpler than inheriting from existing classes and so black box frameworks tend to be easier to use. Johnson argues that frameworks tend to mature towards black box frameworks.

2.4.3 Interaction

Andersen Consulting [Sparks et al., 1996] differentiates frameworks based on how they interact with the application extensions, rather than their scope or how they are customized.

- *Called frameworks* correspond to code libraries (such as the Eiffel libraries [Meyer, 1994] or Booch's libraries [Booch, 1994]). Applications use the framework by calling functions or methods within the library.

- *Calling frameworks* incorporate the control loop within the framework itself. Applications provide the customized methods or components which are called by the framework ("don't call us, we'll call you"). In this chapter we will be primarily focusing on calling frameworks, although much of the material applies to called frameworks as well.

2.5 Desirable Properties

Frameworks are meant to be reused to develop applications, and so reusability is very important. Software reusability means that ideas and code are developed once, and then used to solve many software problems, thus enhancing productivity, reliability and quality. With frameworks, reusability applies not only to the code, but also the design. A good framework has several properties such as ease of use, extensibility, flexibility, and completeness [Adair, 1995] which can help to make it more reusable.

2.5.1 Ease of Use

Ease of use refers to an application developers ability to use the framework. The framework should be both easy to understand and facilitate the development of applications, and therefore ease of use is one of the most important properties a framework can have. Frameworks are meant to be reused, but even the most elegantly designed framework will not be used if it is hard to understand [Booch, 1994]. In order to improve the user's understanding of the framework, the interaction (both the interface and the paths of control) between the application extensions and the framework should be simple and consistent. That is, the hooks should be simple, small and easy to understand and use. Additionally, the framework should be well-documented with descriptions of the hooks, sample applications and examples that the application developer can use.

2.5.2 Extensibility

If new components or properties can be added to a framework easily, then it is extensible. Even if a framework is easy to use, it must also be extensible to be truly useful. A simple parameterized linked list component may be completely closed and easy to use, but its reusability is enhanced if it can be easily extended to include new operations.

2.5.3 Flexibility

Flexibility is the ability to use the framework in more than one context. In general, this applies to the domain coverage of the framework. Frameworks that can be used in multiple domains, such as graphical user interface frameworks, are especially flexible. If a framework is applicable to a wide domain, or across domains, then it will be reused more often by more developers. However, flexibility must be balanced with ease of use. In general, a framework with many abstract hooks will be flexible, but will also be either difficult to understand, require too much work on the part of the application developer to use, or both.

2.5.4 Completeness

Even though frameworks are incomplete, since they cannot cover all possible variations within a domain, relative completeness is a desirable property. Default implementations can be provided for the abstractions within the framework so they do not have to be re-implemented within every application, and application developers can run the framework to gain a better understanding of how it works. The framework library can provide the implementations of common operations, which the developer can choose, making the framework easier to use as well as more complete.

2.5.5 Consistency

Consistency among interface conventions, or class structures is also desirable. Names should be used consistently within the framework. Ultimately, consistency should speed the developers understanding of the framework and help to reduce errors in its use.

3 Benefits and Concerns of Building a Framework

The design of any type of framework requires the consideration of many issues. The first, and possibly the most important decision is whether or not a framework is needed. Although frameworks have benefits such as design reuse, their are also drawbacks, such as the increased cost of building a good object-oriented framework as compared to a single application.

Benefits

1. Reuse. Quite simply, the main benefit of frameworks is the ability to reuse not only the implementation of a system, but the design as well. The framework helps to reduce the cognitive distance [Krueger, 1992] between the problem to be solved and the solution embodied in a software system. Once a framework has been developed, the problem domain has already been analyzed and a design has been produced which can be reused to quickly develop new applications.

2. Maintenance. Since all applications developed from a framework have a common design and code base, maintenance of all the applications is made easier.

3. Quality. The framework not only provides a reusable design, a tested design proven to work and therefore forming a quality base for developing new applications.

Concerns

1. High Cost. Building a framework is often significantly more costly than developing a single application. A great deal of time is spent defining and refining the abstractions that form the core of the framework.

2. Shifting Domains. The domain in which the framework is to be built must be relatively stable and well-understood. If the domain changes, then, unless the framework can be easily modified to fit the new domain, much of the effort put into development will be lost.

3. Evolution. Any changes to the framework will affect the applications developed with the framework. If the architecture or any interfaces change then upgrading applications to use the new framework may be costly. If the applications are not upgraded, then the advantage of having a common code base is lost.

Typically, the best candidates for frameworks are applications that are developed repeatedly, with minor variations [Taligent, 1995]. Domains that change rapidly, or are new enough not to have a base of existing applications are generally not good candidates for frameworks.

4 Design Process

Frameworks should be developed from 'scratch'. It is unlikely that an application can be transformed into a framework in a straightforward manner. Frameworks, just like most reusable software, have to be designed to be reusable from the very beginning.

As Booch [1996] suggests, object-oriented development in general and framework development in particular requires an iterative or cyclic approach in which the framework is defined, tested and refined a number of times. Additionally, small teams or even individual developers are recommended for framework development so that each member of the development team has a good overall understanding of the framework.

Standard software development methodologies are not sufficient for developing object-oriented frameworks [Pree, 1995]. For example, traditional methods do not take into account the need to design the hooks of a framework which provide for the flexibility and extensibility of the framework. They tend to focus on the functional requirements. Hooks are also requirements of a framework, but they are quasi-functional. They do not perform functions within the system, but instead allow the framework to be customized to support a wide range of functionality. Hooks should be considered throughout the process of requirements analysis through to testing [Cline, 1996].

While there is no agreed upon standard for designing frameworks, some techniques have been proposed [Sparks et al., 1996] [Taligent, 1995] [Johnson, 1993] [Pree, 1995]. The proposed approaches are still immature and provide guidelines rather than a fully defined methodology. Each of the approaches can be characterized by several general steps: analysis, design and implementation, testing, refinement.

The steps are the traditional stages of software development, but each is tailored to the design of frameworks. Typically, the framework is not built during a single pass, but through multiple iterations of the steps.

4.1 Analysis

As with any type of software development, the first stage is the analysis of the problem domain. In the case of frameworks, this requires a domain expert. The expert identifies the size of the domain that the framework covers, the abstractions that will be incorporated within the framework, and how variations between applications within the domain will be dealt with.

One of the key decisions that needs to be made when building a framework is deciding on how large of a domain it will cover. Does the framework apply to a large domain, a narrow part of a domain, or even apply to several domains? There are benefits and drawbacks to frameworks that cover a large domain or a large part of a domain versus small frameworks which cover a narrow part of a domain. A wide framework will be reusable in more situations, and thus be flexible, but may be unwieldy and difficult to maintain. Building a widely applicable framework is a significant undertaking, requiring a lot of resources.

Narrow frameworks may be insufficient for an application, and developers will have to add significant amounts of additional functionality to produce an application. Narrow frameworks are also easily affected by changes in the domain. While a wider framework might be able to evolve, a narrow framework may no longer be applicable. On the other hand, narrow frameworks do tend to be smaller and less complex and therefore easier to maintain. They are also potentially easier to use in other contexts because they aren't all encompassing of a particular domain. Finally, using a large framework often requires using all of the functionality within it, regardless if it is needed or not. Having several smaller frameworks instead of one large one allows framework users to only take the functionality they need. In general, the benefits of building small frameworks outweighs the drawbacks, and so small frameworks are typically recommended.

After the domain of the framework has been determined, analyzing the domain of the framework helps to determine the primary or key abstractions that will form the core of the framework. For example, graphical drawing framework, such as HotDraw, will contain abstractions for representing the drawing, the figures the drawing contains, and the graphical tools used to manipulate the figures.

Examining existing applications within a the domain of the framework is a useful means of identifying the abstractions [Johnson, 1993]. In order to gain domain expertise, a framework designer may also want to build an application within the domain if the designer is not already an expert in the domain [Taligent, 1995].

Developing scenarios for the operation of the framework and reviewing them with potential users of the framework is another recommended analysis approach [Sparks et al., 1996]. Scenarios help to define the requirements of the framework without committing developers prematurely to any design decisions. The scenarios can be abstracted into use cases [Jacobson, 1992] to help identify the primary abstractions and interaction patterns the framework needs to provide.

The hot spots, the places of variation within the framework, also need to be identified. Again, examining existing applications will help identify which aspects change from application to application and which remain constant.

4.2 Design and Implementation

The design determines the structures for the abstractions, frozen spots and hot spots. The design and implementation of the framework are often intertwined. Abstractions can be difficult to design properly the first time and parts of a framework may have to be redesigned and reimplemented as the abstractions become better understood [Pree, 1995]. Parts of the framework may undergo redesign even while other parts are being implemented.

In order to refine the abstractions, reviews of the design are recommended [Sparks et al., 1996]. Reviews examine not only the functionality the design provides, but also the hooks and means of client interaction provided by the framework.

Specific techniques that aid in the design of frameworks will be discussed later in the chapter. However some general guidelines have been identified through experience by framework developers working on the Taligent frameworks and ET++. In order to develop easy to use and flexible frameworks, Taligent [1995] suggests:

- reduce the number of classes and methods users have to override
- simplify the interaction between the framework and the application extensions
- isolate platform dependent code
- do as much as possible within the framework
- factor code so that users can override limiting assumptions

- provide notification hooks so that users can react to important state changes within the framework

Some additional general design advice proposed by Birrer and Eggenschwiler [1993] is to:

- consolidate similar functionality into a single abstraction,
- break down larger abstractions into smaller ones with greater flexibility,
- implement each key variation of an abstraction as a class (and include it in the framework library), and
- use composition rather than inheritance.

At this stage, the specific hooks for each hot spot must also be designed and specified. The hooks show specific ways in which the framework can be adapted to an application, and so are an important part of the framework. Hooks can be described in an informal manner or a semiformal manner using templates [Froehlich et al., 1997].

Often, trade-offs must be considered when designing the hooks and structuring the hot spots in general. Frameworks cannot be arbitrarily flexible in all directions (i.e.. they only bend in certain ways and bending them in other ways will break them). Some of the required flexibility can be determined by examining existing applications. Often the framework designer has to rely on experience and intuition to make these trade-offs. Subsequent testing may require changes in the structure of the hot spots. Further trade-offs occur between flexibility and ease of use. The most flexible framework would have very little actually defined and so require a great deal of work on the part of the framework user. The framework should incorporate as much functionality as it can and all the interfaces should be clearly defined and understandable, sometimes at the expense of flexibility.

4.3 Testing

There are two types of testing that a framework can undergo. First, a framework should be tested in isolation; that is, without any application extensions. Testing the framework by itself helps to identify defects within the framework, and in so doing isolates framework defects from errors that might be caused by the application extensions, or in the interface between the framework and the application extensions. Andersen Consulting has followed an approach of requiring framework designers to produce a test plan that provides a minimum of 50% block coverage of the framework with a goal of 70% coverage [Sparks et al., 1996]. It is important to catch defects in a framework since any defects will be passed on to all applications developed from the framework. Defects in the framework force users to either fix the defects themselves, or find a work around, both of which reduce the benefits of using the framework.

Second, the true test of a framework really only occurs when it is used to develop applications. Designers never truly know if a framework can be reused successfully until it actually has been. Using the framework serves as a means of testing the hooks of the framework, the points where interactions between application extensions and the framework occur. Using the framework also helps to expose areas where the framework is incomplete and helps to show areas where the framework needs to be more flexible or easier to use. The applications produced from this kind of testing can be kept as examples of how to use the framework and are also valuable for regression testing when parts of the framework change.

4.4 Refinement

After testing, the abstractions of the framework will often need to be extended or refined. Building a framework is a highly iterative process, so many cycles through these steps will be performed before the final framework is produced. A rule of thumb among framework developers is that three separate applications must be developed from a framework before it is ready for deployment and distribution. After the core has been successfully developed and implemented, the framework library can be further developed to make the framework more complete.

5 General Framework Development Techniques

5.1 Abstract and Concrete Classes

A good framework often has a core of abstract classes which embody the basic architecture and interactions among the classes of the framework. Not all of the core classes need to be or should be abstract, but the ones involved in hot spots generally are. For example, HotDraw has classes for Figure and Tool which must be customized to use. Framework designers derive new classes from abstract classes by filling in the methods deliberately left unimplemented in the abstract classes or by adding functionality. The abstract classes should be flexible and extensible. These classes capture the properties of key abstractions, but just as importantly, they capture the interactions between elements of the framework as well. The Figure class interacts with many classes to provide its functionality, such as Tool, Drawing and DrawingView, and this interaction is defined at the level of the Figure class. Any classes derived from it will follow that interaction.

A framework will generally have a small number of these core classes [Gangopadhyay and Mitra, 1995], but will also have a number of concrete classes which form the framework library. These concrete classes inherit from the abstract classes but provide specific and complete functionality that may be reused directly without modification in an application developed from the framework. HotDraw provides a number of Figure classes within the library that can be easily reused. However, the library can also contain parameterized object instances, such as the Tool instances provided with HotDraw. Library components may also be composed of a number of core classes rather than derived from one exclusively. Providing these concrete classes makes the framework both more complete and easier to use. If concrete classes are not needed in a particular application, then they can simply be excluded from the application.

5.2 Hot Spots and Frozen Spots

As mentioned earlier, frameworks contain hot spots which are meant to encompass the variability between applications within the domain of the framework. Hot spots provide the flexibility and extensibility of the framework and their design is critical to the success of the framework.

Two questions to consider about a hot spot are: [Pree, 1995]

- what is the desired degree of flexibility, remembering that flexibility has to be balanced with ease of use?
- must the behavior be changeable at run-time, in which case composition is preferred over inheritance?

Pree's hot spot approach [Pree, 1995] defines several metapatterns which aid in hot spot design. Each metapattern defines a set of relationships between template methods and hook methods within an abstract base class. Template methods define the flow of control for the framework and either perform actions directly or defer functionality to hook methods. Hook methods are left unimplemented within the framework and are specialized by the application developer to fit the needs of the application. Design patterns also help in structuring hot spots as described in a later section.

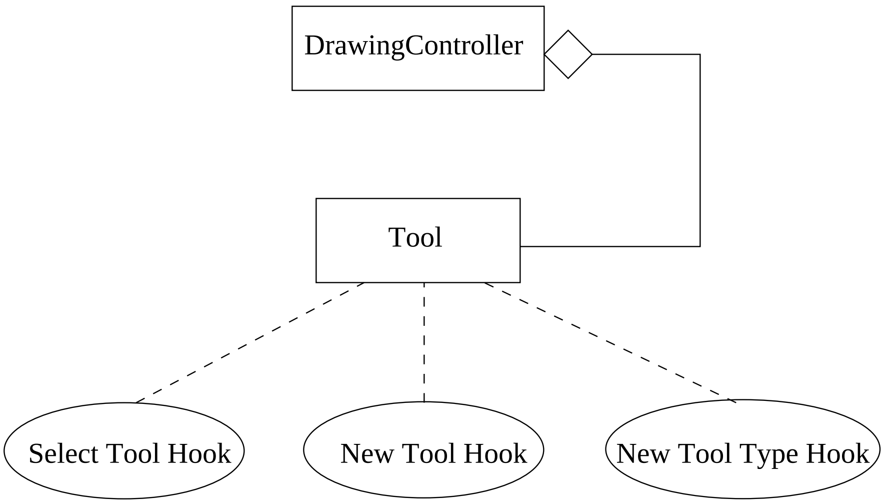

Figure 3: Hooks for the Tools Hot spot.

Each hot spot will likely have several hooks associated with it. The hooks describe how specific changes can be made to the framework in order to fulfill some requirement of the application [Froehlich, 1997]. Figure 3 shows the Tool hot spot and three hooks within it. There are three main ways in which a framework can be modified using hooks and each of these are illustrated in the three hooks depicted in Figure 3.

- Several pre-built components can be provided within the framework library which can be simply used as is in applications, such as the Select Tool Hook. If there are a limited number of options available, then the components provided may be all that is needed.
- Parameterized classes or patterns can be provided (such as the use of template and hook methods) which are filled in by framework users. The Tool class in HotDraw is an example of a parameterized class and the New Tool Hook describes how the parameters must be filled in.
- When it is difficult to anticipate how a hot spot will be used, or if a lot of flexibility is needed, then new subclasses are often derived from existing framework classes and new functionality is added, corresponding to the New Tool Type Hook.

5.3 Composition and Inheritance

Inheritance and composition are the two main ways for providing hooks into the framework. Composition is often recommended over inheritance as it tends to be easier for the framework user to use (data driven as opposed to architecture driven) but each has strengths and weaknesses. The type of customization used in each case depends upon the requirements of the framework.

5.3.1 Composition

Composition typically involves the use of callbacks or parameterized types. The class of the framework to be adapted will have a parameter to be filled in by the application developer which provides some required functionality. Since the customization is done by filling in parameters, the framework user does not need an in-depth knowledge of how the particular component operates. In HotDraw, the Tool class is parameterized so that new Tools can be quickly and easily produced. The main parameter for customizing the Tool class is the command table shown here for the example OMTClassFigure tool.

Reader	Figure	Command
DragReader	OMTClassFigure	MoveCommand
ClickReader	OMTClassFigure	ExpandCommand

The OMTClassTool shown here has two entries in its command table which determines what actions it takes depending upon the context in which the tool was used. The Reader determines the type of use, such as DragReader or ClickReader. The Figure is the type of Figure selected by the tool and the Command is the action to take on the Figure using the Reader. So, if the OMTClassTool is dragged on the OMTClassFigure then it will invoke the MoveCommand which moves the Figure. Tools can be easily produced by combining different Commands, Readers and Figures. The separation of Readers and the Commands from the Tool class also makes the individual Readers and Commands more reusable. Many Tools will have actions for dragging and clicking the mouse and they can all use the same Reader classes.

Composition also allows the component used to be changed dynamically. The DrawingController class in HotDraw uses composition for this purpose. The class has a slot for the Tool that is currently being used, but there are many different Tools in a typical HotDraw application and the user changes Tools frequently. The tool that is currently in the DrawingController class may have to be changed at run-time, so Tool was made into a separate class with each tool having the same interface so that it can be made the current tool whenever the user selects it.

When composition is combined with a large number of existing concrete classes within the framework library, adapting the framework becomes a simple matter of choosing the right components and connecting them together. Composition tends to be the easier of the two types of adaptation to use. A developer should understand the interface to the component, the functionality the component provides and little else about the inner workings of the framework. More generally, the developer simply needs to understand which existing concrete component performs the desired function.

5.3.2 Inheritance

Inheritance involves specializing methods from an abstract class, or adding functionality to an abstract class. Inheriting from a class requires a considerable understanding of the abstract class and its interactions with other classes, thus it can be more error prone and more difficult to use than composition. The advantage of inheritance is extensibility. An application developer can easily add completely new functionality to a subclass of an existing class, which is not as easily accommodated with composition.

New Figures in HotDraw are derived by inheritance. New Figures often require added functionality or instance variables that cannot be anticipated by the framework developers and inheritance provides the means by which the new methods and variables can be added. For example, the OMTClassFigure in Figure 4 is derived from CompositeFigure and adds two new methods, addvar and addmethod. These methods are used to add new variable and method text labels to the class box. It also contains TextPopupFigures derived from TextFigures for the text labels. The TextPopUpFigures contain an additional attribute desc which holds a description or code fragment and a method popupdesc for displaying the description. Since none of this functionality was contained in HotDraw before, and it is impossible to anticipate the full range of additional functionality that might be needed in a Figure, it must be added by inheritance.

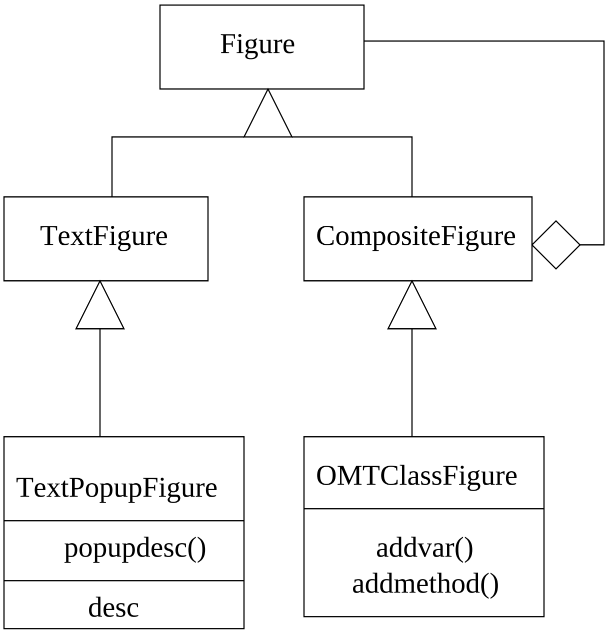

Figure 4: Inheritance Diagram for OMTClassFigure.

Often the abstract class will provide default methods that need to be overridden in a child class. Hook methods are an example of this. The method is called by another template method, but the specific functionality is class specific and can't be pre-determined. HotDraw provides a step method for Drawings that is used for animation, but is left empty by default since no one method can account for the complete variety of possible ways to animate a Drawing.

5.3.3 Composition vs. Inheritance

Composition is generally used when interfaces and uses of the framework are fairly well defined, whereas inheritance provides flexibility in cases where the full range of functionality cannot be anticipated.

Composition forces conformance to a specific interface and functionality which cannot be easily added to or changed. All Tools use Readers to determine the location of the mouse pointer and what action is being taken, and invoke Commands to perform actions on Figures. In practice this works fairly well and covers the range of actions needed to be performed by Tools. When something outside of this paradigm is required, however, the Tool class itself must be subclassed to provide the extra functionality.

It has been proposed that frameworks start out as white box frameworks that rely upon inheritance. As the domain becomes better understood and more concrete support classes are developed, the framework evolves to use more composition and becomes a black box framework [Johnson and Foote, 1988]. The step method for animating Drawings was mentioned above. While it is currently blank, and must be overridden through inheritance, it may be possible to provide a number of standard ways to animate Drawings. Instead of providing a new Drawing class for each standard type of animation, it would be better to move the animation routines into its own class using the Strategy design pattern [Gamma et al., 1996] which can be linked into the Drawing class as needed.

Beyond components that developers have to connect, a framework might gain the ability to decide how to connect components itself. This could possibly be done dynamically or by using a development tool, freeing the developer to simply choose the components needed without doing any programming.

5.4 Framework Families

Much in the same way that an individual class is a specialization of a more abstract class, one framework may be a specialization of a more abstract framework. There are two reasons why such a family of frameworks might be built.

First, a specialized framework can provide more support for special cases within the domain. Abstract frameworks provide the basic services and interactions, while the specialized frameworks extend that to provide more specialized support. The abstract framework provides flexibility while the specialized frameworks provide more completeness and ease of use.

Second, framework families are a means of dealing with complexity. Instead of including all possible options within a single, complex framework, related options can be bundled into specialized frameworks.

An alternate approach for producing framework families is to develop an initial problem solution and then to generalize it [Koskimies et al., 1995]. For example, a framework that is developed to solve a graphing problem might be generalized into a more abstract framework. The generalized framework is then abstracted again into a more abstract problem. This process continues until the most general problem solution is found. The generalization approach quite naturally produces a framework family with the most abstract and flexible framework at the top of the 'family tree' and the more specific framework at the bottom. Unfortunately, it is not always clear what a more general solution to the problem would be, or what the most general solution is.

6 Specific Development Techniques

6.1 Domain Analysis

Domain analysis techniques, such as FODA [Kang et al., 1990], focus on finding the requirements for an entire domain rather than just for a single application. Domain analysis [Arago, 1991] identifies the concepts and connections between concepts within a domain without determining specific designs. The analysis can help to identify the variant and invariant parts of the domain, and the primary abstractions needed for a framework. In the Feature Oriented Domain Analysis technique, the generic parts of the domain are abstracted from existing applications by removing all of the factors that make each application within the domain different. The abstraction is done to the point where the product produced from the domain analysis covers all of the applications. Differences are factored out by generalizing existing parts of applications, or aggregating the variant aspects into more generic constructs. The resulting concepts are then the primary invariants of the framework. Variations within the domain are captured in FODA by refining the common abstractions through specialization and decomposition. Parameters are defined for each refinement which capture the variant aspects. The refinements and parameters can then form the basis for the hooks of the framework.

6.2 Software Architecture

All software systems, including frameworks, must have a solid foundation at the level of *software architecture* [Perry et al., 1992]. Software architecture studies the higher level issues involved in software design, such as the organization of the system, physical distribution of subsystems, performance and composition of design elements [Shaw and Garlan, 1996]. Architecture consists of components, the pieces of software such as UNIX filters or object-oriented classes, and the connectors that allow them to communicate with each other, such as UNIX pipes, or object-oriented methods.

6.2.1 Architectural Styles

An architectural style [Shaw and Garlan, 1996] defines the types of components and connectors that can exist within a particular style, and the rules of how they can be composed. These rules can aid in application development from a framework by helping to describe the types of applications that can or cannot be built, and general rules for their structure.

For example, HotDraw uses the object-oriented architectural style. The components within HotDraw are classes and the connectors are the messages that are passed between classes. Object-oriented architectures allow flexible structures in which objects can collaborate with other objects in arbitrary ways to provide functionality, but each object must know about the objects it collaborates with beforehand, unlike filters in the pipe and filter architecture. However, HotDraw is a framework for graphical drawing applications rather than a command line filter and is well-suited to the object-oriented style.

6.2.2 Domain Specific Software Architectures

Domain Specific Software Architectures [Tracz, 1994] are closely related to object-oriented frameworks. By focusing on a specific domain, a DSSA can provide more specialized components and connectors than architectural styles and even provide components that can be used within applications just as is done with frameworks. A DSSA consists of three things:

- a software architecture with reference requirements and domain model
- infrastructure to support it
- process to instantiate/refine it

The reference requirements define the requirements that an application has within the domain. The domain model is a lexicon of terms within the domain, such as objects, relationships or even actions that occur within the domain. The infrastructure can refer to tool support provided to build an application within the domain, as well as pre-built components than can be used within applications.

A DSSA can be provided as a framework (which satisfies the architecture and some of the infrastructure requirements), but can also include such things as fourth generation languages or application generators. However, the requirements for a DSSA apply equally well to object-oriented frameworks, and suggest that frameworks should be part of a larger package that includes requirements, tool support and a process for using the framework.

6.3 Design Patterns

A design pattern captures some of the expertise of object-oriented software developers by describing the solution to a common design problem that has worked in the past in several different applications. The description of the solution will detail the benefits and drawbacks of the pattern and perhaps name alternatives. Design patterns have been associated with object-oriented frameworks from their inception even though they are not limited to use in frameworks. They are appropriate for designing parts of frameworks and particularly designing hot spots within frameworks because many design patterns enhance flexibility or extensibility.

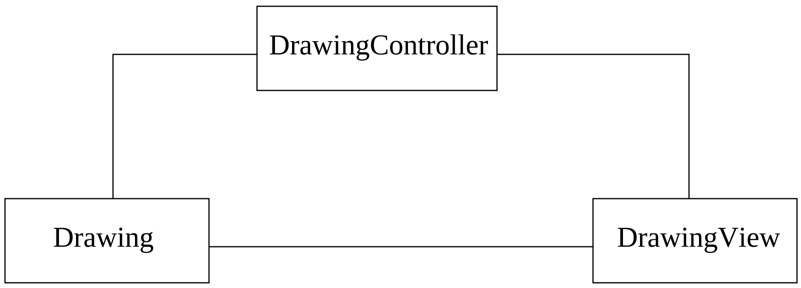

Figure 5: The MVC Pattern in HotDraw.

Model-View-Controller

Context: Interactive applications with a flexible human-computer interface.

Problem: User interfaces are especially prone to change requests, and when the functionality of an application changes, the user interface must change to reflect that. Additionally, different users require different interfaces, such as keyboard vs. menu and button. If the user interface is tightly tied to the functionality of the application, then it becomes difficult to change and maintain. The following forces influence the solution:

- Different windows display the same information and each must reflect changes to the information.
- Changes to the user interfaces should be easy, including supporting different look and feel standards.

Solution: Divide the application into three areas: processing, output, and input. The three components that incorporate these areas are called the model, the view and the controller.

- Model: encapsulates the data and functionality of the application.
- View: displays the data within the model. Multiple views can be attached to a single model to display the data in different ways, or to display different parts of the data.
- Controller: receives input from the user and invokes methods within the model to fulfill user requests. Each view can have its own controller and controllers can be changed to provide different means of input.

Figure 6: The Model-View-Controller Architectural Pattern

Three levels of design patterns have been identified [Buschmann et al., 1996]. Architectural patterns describe a general high level architecture that can form the basis of a framework or application, and are closely related to architectural styles. However, many patterns use a particular style and are not styles themselves. The patterns identify specific components and connectors that are used in the architecture. For example, the HotDraw framework uses the object-oriented architectural style, and the Model-View-Controller architectural pattern [Buschmann et al., 1996] described briefly in Figure 6. As shown in Figure 5 the Drawing forms the basis of the model, DrawingView corresponds to view and DrawingController corresponds to the controller. Using the pattern gives the framework flexibility by separating the display (DrawingView) from the user input mechanisms (DrawingController) and the internal representation of the drawing (Drawing). New views or types of views can be easily added, new user input mechanisms can be implemented or even removed so the application becomes a simple display.

At the next level down, the patterns are specifically called design patterns. These patterns don't form the basis for an entire framework, but help to structure specific parts of a framework. They are especially

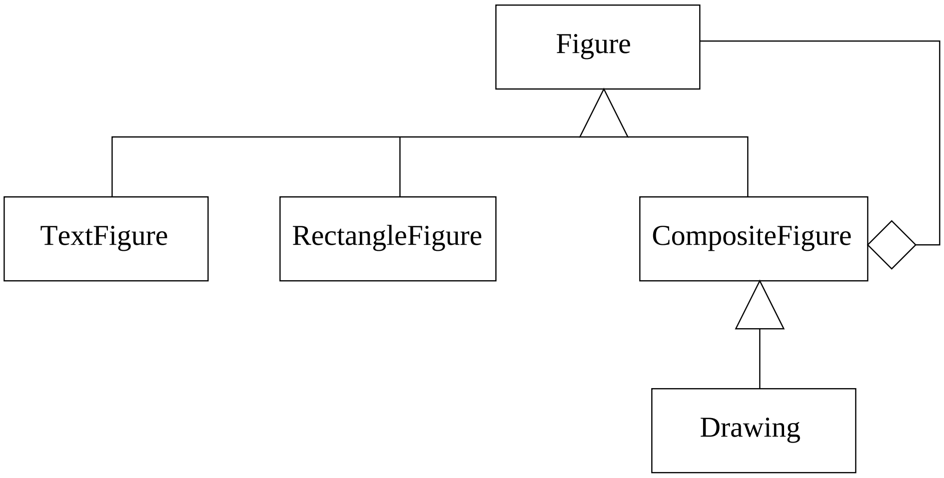

Figure 7: The Composite Pattern in HotDraw.

useful in adding flexibility to hot spots in the framework. The Composite Pattern [Gamma et al., 1996] is used in HotDraw as shown in Figure 7 to help structure the Figures hot spot. Basically, CompositeFigure is made a subclass of Figure with the same interface as Figure, but also may contain any number of Figures. Drawing is a specific type of CompositeFigure. CompositeFigures will respond to the same methods that are invoked on Figures, so that they can be treated interchangeably with ordinary Figures. This allows flexible and arbitrary collections of new Figures and nested CompositeFigures developed for an application rather than forcing a flat model in which Drawings can only contain simple Figures.

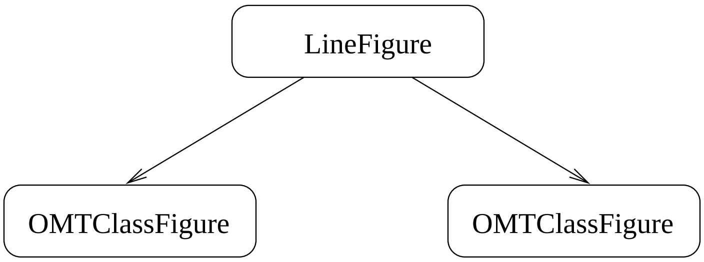

Figure 8: The Observer Pattern in HotDraw.

HotDraw also incorporates the Observer Pattern [Gamma et al., 1996] to provide a means in which Figures can be linked to each other, also within the Figures hot spot. Figure 8 shows a sample HotDraw application in which two OMTClassFigures are connected by a LineFigure. When one of the OMTClassFigures is removed, the LineFigure should be removed as well by the application. In order to provide this feature, the Observer Pattern is used. LineFigure is made a dependent of each OMTClassFigure it is connected to and when one OMTClassFigure is deleted, the framework automatically sends the class the appropriate deletion message.

The OMTClassFigure then notifies each of its dependents so that they can take the appropriate action, deleting itself in this case. Notifications can also be sent to dependents whenever a given Figure changes, giving application developers an easy to use mechanism through which one Figure can reflect changes in another.

The lowest level patterns are called *idioms*. Idioms tend to be implementation language specific and focus on specific low level issues. For example, an idiom might describe how exception handling can be provided in a specific language.

6.4 Open Implementation

Open implementation [Maeda et al., 1997] is another technique for supporting flexibility. Software modules with open implementation can adapt their internal implementations to suit the needs of different clients. The modules themselves support several different implementation strategies and provide a strategy control interface to allow client modules to help choose or tune the implementation strategy that best suits the client. For example, a file system might support several different caching strategies and allow the caching strategy to be tuned based on the usage profile of the application. The application provides this information to the module with open implementation, perhaps indicating that it will perform a sequential scan of a file, or random access. Typically, the strategy selection information is provided during the initialization of the module.

This notion can be applied in particular to the framework library. Allowing clients to help select the implementation strategy can help applications to tune a library class or module for efficiency, and make it more suitable for a wider variety of applications.

Four styles of open implementation interfaces have been identified [Kiczales et al., 1997]

- Style A: No control interface is provided which is the same as providing a black box component or module. The client has no means of tuning the implementation so this approach is best used when only one implementation of a component is appropriate.

- Style B: The client provides information about how it will use the component, typically through setting parameter values when initializing the component. The component itself is responsible for choosing the best implementation strategy based on the information provided. The approach is easy for clients to use, but does not give them much control over the component.

- Style C: The client specifically specifies the implementation strategy to use from a set of strategies. The component has no control over the selection, and this approach is the same as providing several interchangeable components. It is best used when the component cannot select the appropriate strategy through declarative information, as in style B.

- Style D: The client actually provides the implementation to use. The component simply provides the interface, but no implementation. In frameworks, abstract classes are used in a similar way, defining the interface, but leaving the implementation of methods to the application developer. It is best used when the set of possible implementations cannot be predetermined.

6.5 Contracts

One of the difficulties in any type of framework is ensuring that the framework receives or produces the correct information. Integrating classes is an issue in any application development, and using a framework is in part an integration issue. Therefore, it is critical that the interfaces to the framework, both the methods of the framework that are called by an application and the methods of an application that the framework calls, be clearly defined.

6.5.1 Simple Contracts

Simple contracts as proposed by Meyer [1988] are a means of specifying the preconditions required by a method or operation and the postconditions that a method ensures. Specifying and enforcing conditions can help to cut down on the level of defensive programming needed within a framework, or any piece of reusable software. For example, a reusable component within the framework library does not have to handle every possible exception or error condition in the input if the preconditions for the component's methods are clearly identified and enforced.

As Meyer argues, clearly specifying the conditions helps application developers since they do not have to guess what the framework does, and can handle errors that might best be handled by the application. Forcing the framework to handle every possible exception can reduce performance and make the framework less attractive for use.

Further, some means of enforcing the contracts may help reduce the number of errors application developers make when using the framework. In Eiffel, contracts are built into the language. In other languages, Hollingsworth [1992] proposes that separate classes be built to represent contracts. These classes will ensure that the preconditions and postconditions are met for the framework to operate correctly. These contract classes could be included with the release of a framework and used during the development of an application and removed to improve efficiency when the application is released.

6.5.2 Behavioral Contracts

More complex behavioral specifications have been proposed. [Helm et al., 1992] Also called contracts, they cover not only the interactions between two classes, or the conditions placed on one method, but on the total interaction between multiple classes, often involving multiple methods to perform some task. The

```
contract ExecuteTool
    Tool supports[
        s: Sensor
        execute() ↦ Reader → initialize(s,Command); Reader → manipulate(s); Reader → Effect ]
    Reader supports[
        c : Command
        initialize(sensor,c) ↦ Δc {command = c}; c → initialize(sensor.mousepoint)
        manipulate(sensor) ↦ c → execute(sensor.mousepoint)
        effect ↦ c → effect ]
    Command supports[
        initialize(loc)
        execute(loc)
        effect ]
end contract
```

Figure 9: The ExecuteTool Behavioral Contract

specification is flexible in that one contract can specialize another, and contracts can be composed of other contracts. When actually using the contract, a conformance declaration is produced which states how the classes involved fulfill the contract.

Specifying behavioral contracts within a framework helps to capture design information that might be needed by application developers. Figure 9 shows the ExecuteTool contract used in HotDraw which details how Readers, Commands and Tools interact in order to perform the task. When the Tool executes, it first initializes the Reader with the mouse sensor and the Command to be executed. Afterwards, it sends the manipulate command to the Reader which in turn actually executes the command on the Figure found at the current location of the mouse. When the user is finished using the tool (i.e. lets go of the mouse button during a drag operation) the Tool sends the effect message to the Reader which performs any clean up operations needed, including sending the effect message to the Command. When application developers add a new Reader, Command or Tool class to an application developed from HotDraw, they need to conform to this contract. The contract makes the information explicit and so the application developer does not need to synthesize this information from the source code, which may not even be possible if the source code of the framework is not available.

7 Framework Deployment

Once an object-oriented framework has been implemented, refined and tested to the point where it is stable, it is ready to be deployed by application developers. There are a number of questions to be considered when deploying a framework. First, what sort of aids for learning the framework are provided? Andersen Consulting has used three different approaches [Sparks et al., 1996].

- Roll out sessions: sessions are held after a framework has reached a certain phase in its development in order to introduce users to the current set of features. This approach has the advantage of gradually introducing the users to the framework as it is developed.
- Example Applications: examples help to show specific ways in which the framework can be used. Examples can also potentially be modified and used in a new application being developed.
- Reference Documentation: documentation that describes the purpose of the framework, how to use it and how it is designed can and should be provided. Documentation techniques will be covered in the next chapter on using object-oriented frameworks.

Additionally, a framework developer might be involved with application development as well. When the framework designer is also involved in using the framework, application developers can use the designer's expertise with the framework. The framework designer may also gain insights into how the framework can be modified or improved.

Second, will the framework be distributed as source code or a binary executable? Binary executables do not allow developers to modify the framework but instead require them to rely on the interfaces provided. This may make future upgrades of the framework easier to incorporate into existing applications if the interfaces don't change, and will protect the framework code against tampering. However, developers cannot fix any errors they encounter in the framework and are forced to devise workarounds. Additionally, developers cannot examine the binary implementation to help to understand the framework as they can with source code.

Third, how will changes to the framework be handled? When bugs are fixed or new features are added to the framework, how will existing applications be affected? Ideally, only the internal implementations of framework components will change, leaving interfaces and hooks the same, but this is not always possible. This issue will be examined more fully in the chapter on using object-oriented frameworks.

8 Summary and Open Issues

Object-oriented frameworks provide the means for reusing both the design and implementation of a system or subsystem. As such, they provide tremendous leverage for the application developer. The design of similar applications do not have to be repeatedly constructed out of existing or new code components, but instead an existing design can be customized quickly to provide the desired functionality. Due to the common design base, several applications developed from the same framework can also be maintained more easily.

However, frameworks are more difficult to design and develop than single applications. Frameworks must be general enough to capture the commonalties of all the applications that might be built within the domain of the framework and they must also be flexible enough to allow for the variations that exist between those applications. All of this generality and flexibility has to be made easy to use for application developers if the framework is to be a success.

There are no mature framework development methodologies. There are, however, guidelines and promising techniques for aiding in the development of frameworks and several of these have been presented. Domain analysis can be used to identify the commonalties and variations (frozen and hot spots) in the domain. Inheritance and composition can be used to provide different sorts of ease of use and flexibility. Design patterns and open implementation techniques help to structure the framework and to provide the right types of flexibility.

Providing such a methodology is one of the key open issues in the area of object-oriented frameworks. A framework development process needs to focus on the identification of hot spots and frozen spots. How are the right abstractions for the framework determined and how should they be structured? It also needs to focus on building flexibility and ease of use into the framework, which in essence is designing the hot spots of a framework.

Some techniques are available for developing flexibility in a framework, but there is very little help for deciding on which tradeoffs to make. Which part of the framework should be made more flexible and how will it affect the other parts? Does flexibility need to be sacrificed for more ease of use? While there are guidelines for making frameworks easier to reuse, there is no criteria quantifying just how easy they should be to reuse.

Object-oriented frameworks are a key means of moving software development away from the traditional means such as continuously redeveloping commonly used code or trying to reuse and fit together unrelated and often mismatched components. Frameworks provide the components and the context, together with an overall structure and design to ensure component compatibility and provide a powerful form of software system reuse.

References

[1] Adair, D. 1995. Building Object-Oriented Frameworks. *AIXpert.* Feb. 1995.

[2] G. Arango, G. and R. Prieto-Diaz, R. 1991. *Domain Analysis Concepts and Research Directions,* IEEE Computer Society.

[3] Beck, K. and Johnson, R. 1994. Patterns Generate Architectures. *Proceedings of ECOOP 94.* 139-149.

[4] Birrer, A. and Eggenschwiler, T. 1995. Frameworks in the Financial Engineering Domain: An Experience Report. *Proceedings of ECOOP 93.* 21-35.

[5] Booch, G. 1994. Designing an Application Framework. *Dr. Dobb's Journal.* 19(2):24-32.

[6] Brown, K., Kues, L. and Lam, M. 1995. HM3270: An Evolving Framework for Client-Server Communications. *Proceedings of the 14th Annual TOOLS Conference.* 463-472.

[7] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. 1996. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons Ltd, Chicester England.

[8] Campbell, R.H., Islam, N., Raila, D., and Madany, P. Designing and Implementing Choices: An Object-Oriented System in C++. *Communications of the ACM.* 36(9):117-126.

[9] Cline, M.P. 1996. Pros and Cons of Adapting and Applying Patterns in the Real World. *Communications of the ACM.* 39(10):47-49.

[10] Cotter, S. and Potel, M. 1995. *Inside Taligent Technology,* Addison-Wesley Publishing Company, Reading, MA.

[11] Eggenschwiler, T. and Gamma, E. 1992. ET++ SwapsManager: Using Object Technology in the Financial Engineering Domain. *Proceedings of OOPSLA 92.* 166-177.

[12] Froehlich, G., Hoover, H.J., Liu, L. and Sorenson, P. 1997. Hooking into Object-Oriented Application Frameworks. *Proceedings of the 1997 International Conference on Software Engineering.*

[13] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA.

[14] Gangopadhyay, D. and Mitra, S. 1995. Understanding Frameworks by Exploration of Exemplars. In *Proceedings of 7th International Workshop on Computer Aided Software Engineering (CASE-95).* 90-99.

[15] Helm, R., Holland, I.M. and Gangopadhyay, D. 1992. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. *Proceedings of ECOOP/OOPSLA 90*. 169-180.

[16] Hollingsworth, J. 1992. *Software Component Design-for-Reuse: A Language Independent Discipline Applied to Ada*, Ph.D. thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, Ohio.

[17] Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G. 1992. *Object-Oriented Software Engineering–A Use Case Driven Approach.* Addison-Wesley, Wokingham, England.

[18] Johnson, R. 1992. Documenting Frameworks Using Patterns. *Proceedings of OOPSLA 92*. 63-76.

[19] Johnson, R. 1993. How to Design Frameworks. *Tutorial notes from OOPSLA 1993.*

[20] Johnson, R. and Foote, B. 1988. Designing Reusable Classes. *Journal of Object-Oriented Programming.* 2(1):22-35.

[21] Kang, K., Cohen, S., Hess, J., Novak, W. and Peterson, A. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. (CMU/SEI-90-TR-21, ADA 235785). Pittsburgh, Pa.: *Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa..*

[22] Kiczales, G., Lamping, J., Lopes, C.V., Maeda, C., Mendhekar, A., and Murphy, G. 1997. Open Implementation Design Guidelines. *Proceedings of the 19th International Conference on Software Engineering.* Boston, Mass. 481-490.

[23] Koskimies, K. and Mossenback, H. 1995. Designing a Framework by Stepwise Generalization. *Proceedings of the 5th European Software Engineering Conference. Lecture Notes in Computer Science 989.* Springer-Verlag. 479-497.

[24] Krueger, C.W. 1992. Software Reuse. *ACM Computing Surveys.* vol. 24(6): 131-183.

[25] Maeda, C., Lee, A., Murphy, G., and Kiczales, G. 1997. Open Implementation Analysis and Design. *Proceedings of the 1997 Symposium on Software Reusability in ACM Software Engineering Notes* 22(3):44-53.

[26] Meyer, B. 1988. *Object-Oriented Software Construction.* Prentice Hall, Englewood Cliffs, NJ.

[27] Meyer, B. 1994. *Reusable Software The Base Object-Oriented Component Libraries.* Prentice Hall, Englewood Cliffs, NJ.

[28] O'Connor, J., Mansour, C., Turner-Harris, J. and Campbell, G. 1994. Reuse in Command-and-Control Systems. *IEEE Software* 11(5):70-79.

[29] Perry, D.E. and Wolf, A.L. 1992. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes.* 17(4):40-52.

[30] Pree, W. 1995. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley Publishing Company, Reading, MA.

[31] Rumbaugh, J., Blaha, M., Premerlani, W., Frederick, E. and Lorenson, W. 1991. *Object-Oriented Modeling and Design,* Prentice Hall, Englewood Cliffs, NJ.

[32] Schmid, H.A. 1995. Creating the Architecture of a Manufacturing Framework by Design Patterns. Proceedings of OOPSLA 95. 370-384.

[33] Shaw, M. and Garlan, D. 1996. *Software Architecture - Perspectives on an Emerging Discipline,* Prentice Hall, Englewood Cliffs, NJ.

[34] Sparks, S., Benner, K. and Faris, C. 1996. Managing Object-Oriented Framework Reuse. *IEEE Computer.* 29(9): 52-62.

[35] Taligent. 1995. *The Power of Frameworks.* Addison-Wesley Publishing Company, Reading, MA.

[36] Tracz, W. 1994. Domain-Specific Software Architecture (DSSA) Frequently Asked Questions (FAQ). *ACM Software Engineering Notes.* 19(2):52-56.

[37] Vlissides, M. and Linton, M.A. 1990. Unidraw: A Framework for Building Domain-Specific Graphical Editors. *ACM Transactions on Information Systems.* 8(3):237-268.