Using 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

Often it is difficult to reuse a software component outside of its original context. Object-oriented frameworks can provide the context in which the component is meant to be reused and thus allow for a significant amount of reuse. An object-oriented framework is the reusable design of a system or subsystem implemented through a collection of concrete and abstract classes and their collaborations [Beck and Johnson, 1994]. The concrete classes provide the reusable components, while the design provides the context in which they are used. A framework is more than a collection of reusable components. It provides a generic solution to a set of similar problems within an application domain. The framework itself is incomplete and provides places called hooks at which users can add their own components specific to a particular application.

Developing an application from a framework differs from developing an application on its own. The framework already supplies the architecture of the application, and users fill in the parts left incomplete by the framework. A framework typically includes the main control loop and calls application extensions to perform specific tasks. Unlike the reuse of pure function libraries, framework users give up control of the design. In return, users are able to develop applications much more quickly, and a single framework can form the basis for a whole family of related applications. Since frameworks can be complex and difficult to learn, documenting all aspects of the framework is important to aid in the user's understanding of the framework and to make the framework easier to use.

In this chapter we discuss techniques and concepts related to using frameworks. Many users will use a framework as it was meant to be used, but others will want to use the framework in new, non-standard ways. Still others will want to evolve the framework to incorporate new capabilities. Although the focus of the paper is mainly on using the framework as it was meant to be used, but we discuss issues related to many other aspects of use.

The next section introduces some terms and concepts important to framework use. Section 3 discusses some of the advantages and disadvantages of using a framework. Section 4 describes how frameworks affect application development. Section 5 discusses the difficulties with using multiple frameworks and section 6 describes some issues in evolving frameworks. Section 7 presents a number of different documentation methods and how they aid in the use of frameworks. Finally some conclusions and a brief discussion of some open issues are presented in the last section.

2 Terms and Concepts

Not everyone will use a framework in the same way. In this initial section we discuss the ways in which a framework can be used, and the different types of users of frameworks.

2.1 Ways to Use a Framework

There are a number of different ways in which to use a framework. Each of them require a different amount of knowledge about the framework and a different level of skill in using it. Taligent [1995] defines three main ways in which frameworks can be used.

- **As is**: the framework is used without modifying or adding to it in any way. The framework is treated as a black box, or maybe as a collection of optional components that are plugged together to form an application.
- **Complete**: the framework user adds to the framework by filling in parts left open by the framework developers. Completing the framework is necessary if it does not come with a full set of library components.
- **Customize**: the framework user replaces part of the framework with custom code. Modifying the framework in such a way requires a detailed knowledge of how the framework operates.

2.2 Users of Frameworks

In addition to the different ways to use a framework, different people will use a framework with different goals [Bulter and Denomme, 1997].

- **Regular user**: many users will use a framework in the way that it was meant to be used. They will use it as is, or they will complete the framework as the framework designer intended. A regular user needs to know only enough about the framework to enable them to use it effectively and typically do not require an in-depth knowledge of the framework.

- **Advanced user**: some users will want to use the framework in unexpected ways, ways that the framework developers never anticipated or planned for. They will use the framework in the same way as regular users but will also customize the framework or try to add completely new and unanticipated functionality to it. Needless to say, the advanced user needs a deeper understanding of the framework.
- **Framework developer**: a framework can evolve by adding functionality or fixing errors, specialized frameworks can be derived by adding specialized classes, or the framework can be generalized to accommodate a wider domain. The framework developers performing these activities need to know all of the details of the design and implementation of the framework and must keep in mind how changes will affect applications already developed from the framework.
- **Developer of another framework**: some users simply want to learn how the framework achieves its flexibility, and need to know about the design and the decisions behind it.

Of the four types of users, the first is probably the most common. A framework is designed for a particular type of application and will be most successful when it is used to build that type of application. As an example consider a framework for building graphical user interfaces. Most users simply want to build a user interface for their application, and will use the framework as intended. A few users will push the interface paradigm to develop custom interface styles. Even fewer will be interested in evolving the framework.

3 Advantages and Disadvantages of Using Frameworks

Frameworks provide tremendous leverage for developers of new applications. For example, a framework represents a flexible design that can be easily and quickly extended to develop applications. However, frameworks are not appropriate for every application, and here we give some advantages and disadvantages of using a framework.

Advantages

- **Reusing expertise**: the single biggest advantage of using a framework is that it captures the expertise of developers within that domain. The framework developers are generally experts within the domain and have already analyzed the domain to provide a quality, flexible design. That expertise can be transferred to the application developers simply by using the framework.
- **Decreased development time**: the problem domain does not have to be analyzed again, and the framework often provides a number of components that can be used directly in an application. Users

familiar with the framework can develop new applications from a framework in much less time than without the framework. However, there is the disadvantage of learning the framework as discussed below.

- **Enhanced quality**: the framework should have a well thought out, quality design. Applications developed from the framework will inherit much of the quality design, although poorly developed applications based on high quality frameworks are still possible.
- **Reduced maintenance cost**: if a family of similar products are developed from a single framework, then maintainers will only have to learn one standard design and will be able to maintain the whole product line more easily.

Disadvantages

- **Framework mismatch**: committing to a particular framework can be inconvenient if the requirements of the application are incompatible with the design of the framework. It can be disastrous if the incompatibilities are found late in the application development cycle. Unfortunately, knowledge of what the framework can and cannot do primarily comes from experience using the framework, although clear documentation helps to alleviate this problem. Prototype projects can help familiarize users with what a framework can be used for without jeopardizing an important project.
- **Learning curve**: using a framework requires some amount of learning, just as with any relatively complex tool or technique. A complex framework can require a great deal of time to learn and may not be appropriate if very few applications will be developed from it. The cost of the initial period of learning is lessened if several applications are developed from a single framework.
- **Lack of design control**: the framework already has a design specified and implemented and any applications developed from it have to conform to that design. Application developers give up most of their control over the design, but this loss is more than offset by the advantages of using a framework.

4 Building Applications from Frameworks

Much like framework design, there are no well-defined methodologies for developing applications from frameworks. The problem is made worse since individual frameworks will often require different processes for using them. Rather than defining a methodology, we provide some general techniques for using object-oriented frameworks.

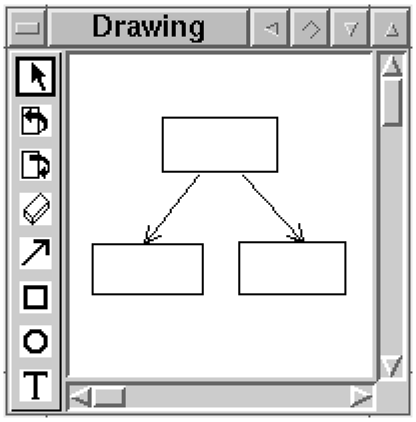

Figure 1: HotDraw.

4.1 Example Framework

In this section we will use the HotDraw framework [Johnson, 1992] which is intended for developing structured graphical editors written in Smalltalk. A view of a HotDraw application is shown in Figure 1. The major elements of HotDraw are the Tools arranged in a toolbar which are used to manipulate Figures within a Drawing. The tools provided in HotDraw include the selection tool for moving and manipulating figures, the deletion tool for removing figures, and many tools for drawing different types of basic figures such as circles or rectangles.

As an example of an application built from a framework, we used HotDraw to construct a graphical display application for the Responsive Corporate Information Model (RCIM) [Tse, 1996] manager prototype. RCIM is an aid to corporate strategic planning. It manages several alternative and/or interrelated plans consisting of the goals, objectives and action plans of the business. The plans can be represented as diagrams, and the display application draws and lays out those diagrams automatically to give a graphical depiction of the plans. An example of a plan is shown in Figure 2.

4.2 Analysis

As with any type of software development, one of the first stages involved in developing applications from frameworks is analyzing the requirements of the application. If the users are already familiar with the framework, then the requirements can be cast immediately in a form that is compatible with the framework.

If the users do not already have a framework, then they will have to find one that supports the application requirements. Frameworks already exist in many areas such as client-server communications [Brown, et al., 1995], operating systems [Campbell et al., 1993], graphical editors [Vlissides and Linton, 1990] and graphical interfaces [Weinand et al., 1988]. A framework can be chosen by viewing the documentation, or through

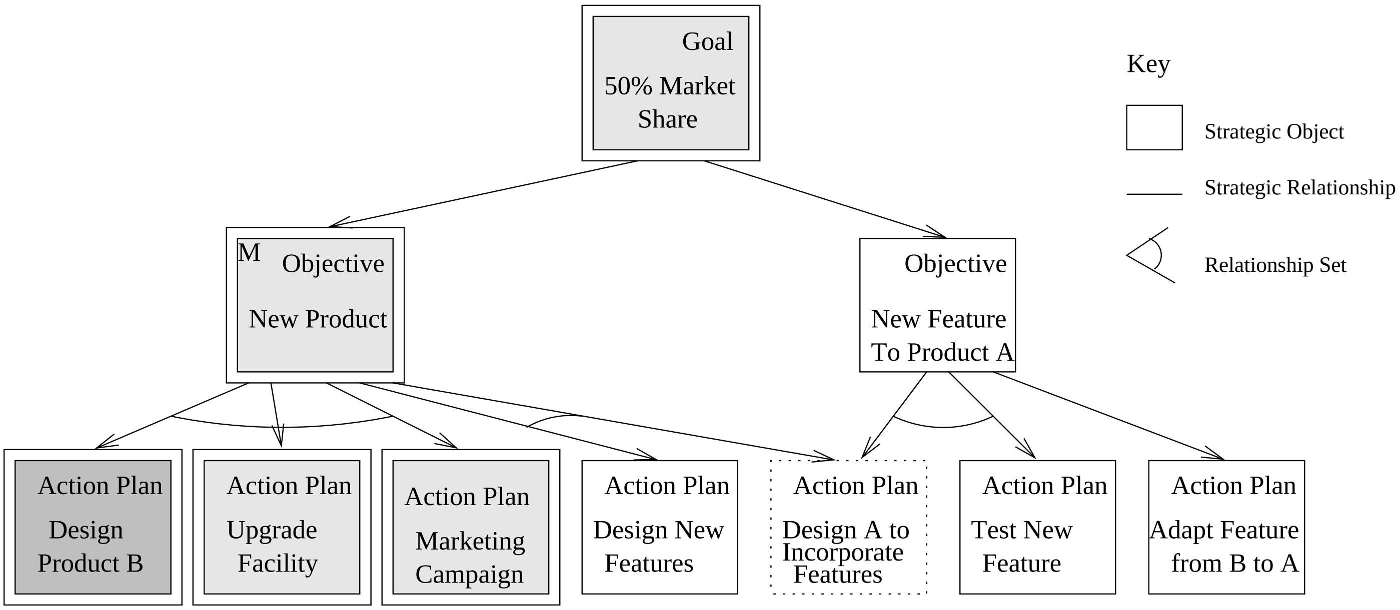

Figure 2: Example of a RCIM Strategic Diagram.

other user's experiences. In particular, the documentation covering the purpose and intended use of the framework should indicate if the framework is compatible with the requirements. The documentation should also list any limitations of the framework. By looking at the hot spots of the framework (see the chapter on designing frameworks), users can check to see if the framework has the desired flexibility. By looking more closely at the hooks of the framework, users can try to map the functional requirements met by the hooks with the requirements of the application. Non-functional requirements have to be matched to the capabilities given in the description of the purpose of the framework. Additionally, the framework determines the types of requirements that can be met. If the chosen framework supports a restrictive set of system functionalities, then any system developed from the framework may also be restrictive unless significant changes are made to the framework itself – a dangerous practice to follow.

In the RCIM graphical display application we have several general requirements. Two views are provided: the Strategic Object View which displays all of the Strategic Objects and Strategic Relationships connected below a given root object; and the Constraint Object View which displays a single Strategic Object and all Constraint Objects and Constraint Relationships connected to it. All strategic and constraint objects are represented using graphical icons that are combinations of text and polygons. Objects go through a number of different states depending on the current state of the plan and each state has a different look, some of which can be set to blink to attract attention when inconstancies in the plan arise. All relationships are represented using lines. Multiple views can be open at the same time and each view should be kept consistent with one another. Additionally, each view has a distinct layout associated with it which is performed automatically by the program.

Since HotDraw is a framework for manipulating and displaying diagrams it proved to be a good platform upon which to build a graphical display application for the diagrams. HotDraw provides Figures to represent the objects and relationships. It provides hooks for animation to perform the blinking, and it allows multiple views to be associated with a single underlying drawing each of which is updated to maintain consistency across views. While the display application does not make use of the Tools, HotDraw's support for tools and figure manipulation allows us to extend the display application into a graphical editor with little difficulty. The only major requirement not supported is automatic layout, which might be a problem if the required layout strategy is incompatible with the framework. At this point, we can look at the hot spots to see if the framework can be extended, or into the design of the framework itself. It turns out that layout routines can be added to the Drawing class with little difficulty, but this requires a more advanced use of the framework.

4.3 Learning to Use the Framework

One of the first difficulties faced by users of any framework is learning how to use it. Frameworks are typically complex and abstract, making them difficult to learn. Learning how to use a framework given only the code and interface descriptions is a daunting task and makes framework use unattractive. A framework should be easier to use as the basis for an application than building a new application without the framework, so some means of lowering the learning curve is needed. Some approaches are:

- **Framework developers as users**: when the framework developers are also the ones developing applications and maintaining the framework, they are already experts on the framework and require little, if any, time to learn it. They can also pass on their expertise to new developers on an informal basis when it is needed. As long as the framework developers do not leave and take their expertise with them, this approach can be effective for in-house frameworks. However, for frameworks that will be distributed to other users, other methods are needed.

- **Tutorial sessions**: framework developers can hold tutorials in which they show potential users what the framework can be used for and how it can be used. Often the developers go through examples which help to make the abstract details of the framework more concrete and understandable. Sessions can also be held as the framework is being developed in order to gradually expose users to new concepts in the framework [Sparks et al., 1996].

- **Tool support**: a good tool can make a framework much easier to use. With it, regular users generally do not have to learn all the details about the framework since the tool will dictate how and where adaptions can take place. The tool will perform the tedious tasks of integrating components into the framework, leaving users free to focus on design. A simple example of this is the ToolBuilder tool which comes with HotDraw [Johnson, 1992]. It allows users to build new tools simply by filling in the appropriate parameters and then automatically integrates the new tool into an application. More complex tools such as the one provided for OSEFA [Schmid, 1996] help users to develop complete applications by allowing them to select from existing components or sometimes to add their own components. However, the tool also constrains how the framework can be used, so it is not as valuable to advanced users that want to use the framework in new ways. Existing tools also tend to be tied to individual frameworks and cannot be used with other frameworks, or be used to integrate more than one framework together.

- **Documentation**: good documentation can aid users in learning the framework. It can be used to not only capture the design details and design rationale of the framework, but also to help users to learn the framework. Each type of user can be accommodated by different types of documentation. Documentation is discussed in detail later in Section 7.

4.4 Design and Implementation

Learning how to use the framework is closely related to using the framework during application design and implementation. However some knowledge of how to use the framework is necessary before application system design can start.

Since the framework should already define an abstract design of the application and provide much of the implementation, users have much less work to do at this stage. Frameworks are used as a whole rather than piecemeal. When building an application from a framework, the framework core will form the basis for the application. Regular users will not have to do much additional design as they will follow what is already present in the framework. A calling framework defines the main execution loop. It forms the core of the application and it calls application extensions. The user assumes an assemblers role of providing pieces required by the framework in order to complete the application rather than building an application that calls functions within the framework. Advanced users will likely extend or modify the framework, but the additional design has to be constrained to fit the existing design of the framework.

If hooks are defined for the framework, then all users can follow them to adapt the framework. Hooks can provide support for choosing components to customize the framework, for completing the framework by filling in parameters or patterns of behavior, or by allowing new behavior to be added, with some customization. If hooks are not provided, then the user has to discover how the framework can be adapted through other documentation or communication with the framework developers or other users.

There are three general ways in which frameworks can be adapted to an application without modifying the framework itself. The framework may come with library components that can be used directly within the

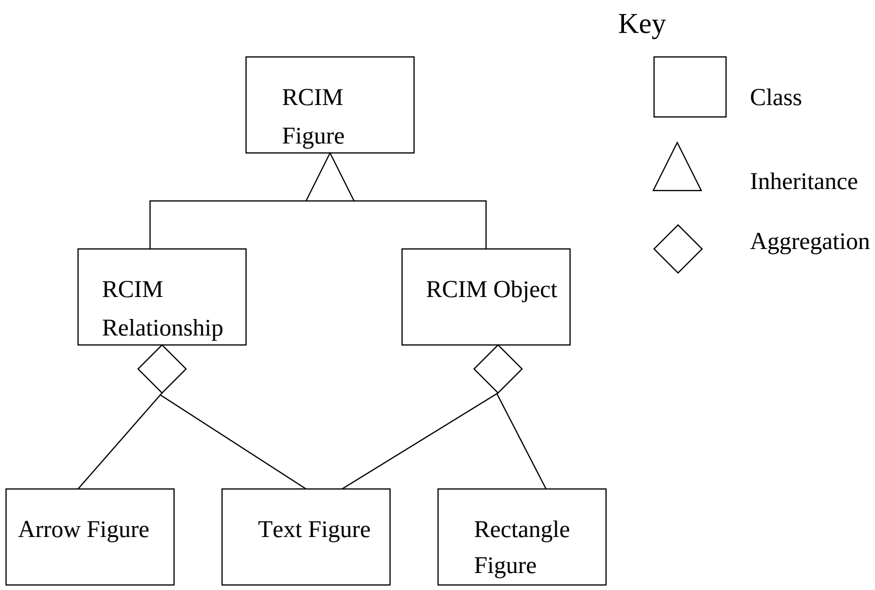

Figure 3: Composing Figures in HotDraw.

application without modification. The components have the advantage of already being developed for the framework and so should integrate smoothly. All that the user must do is select the components and connect them to the framework. Tools can be provided for the framework to make the connection and configuration automatic. The RCIM graphical display application makes use of several existing Figures within HotDraw to compose other Figures as shown in Figure 3. The RCIMObject figure is composed of existing RectangleFigures and TextFigures.

When existing components are not available, new ones must be developed and fitted to the framework. How the new components are developed depends on whether the hook into the framework is done through composition or inheritance. With composition, the user is generally provided with an interface that the framework requires and develops a component that conforms to that interface. With inheritance, an abstract class is provided which the user "subclasses" and fills in with methods or parameters. The graphical display application requires strategic objects in certain states to blink, and so uses the animation hook provided by HotDraw within the Drawing class. Drawing does not provide the required animation, so as shown in Figure 4, a new class RCIMDrawing is derived from Drawing. The step method of RCIMDrawing is then filled in to perform the desired animation.

When the framework does not already contain the desired functionality, then it has to be added. However, the addition of new functionality is specific and local to the application, and does not become part of the framework itself. If the new functionality is applicable to many different applications then the framework must evolve in order to include the functionality. Extending the framework with new functionality is a more

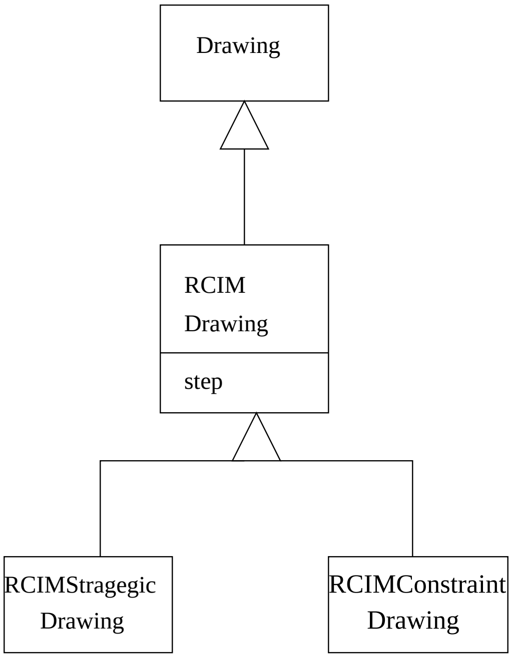

Figure 4: RCIMDrawing Class.

advanced use of the framework and requires a greater understanding of the design of the framework. While configuring and completing the framework allows regular users to treat much of the framework as a black box, extending the framework requires that the advanced user understand both the structure and behavior of the framework. New functionality has to fit into the existing structure and behavior by preserving the state of the framework. In the RCIM display application, the framework can be extended to provide automatic layout capabilities. The layout occurs at the level of the Drawing so we investigate the Drawing class and its interactions to see how to integrate the layout capabilities. Layout needs to occur when figures are added or deleted, and the layout routines are inserted into the framework's flow of control at those points after the framework has completed its normal processing. The RCIMDrawing class is modified to call the layout algorithms at those points. Layout is provided as a separate class, since it is important to be able to provide different layout algorithms for different types of drawings, and connected to the RCIMDrawing as shown in Figure 5. The Layout class also directly interacts with the Figures within the drawing in order to reposition them.

Since the layout algorithm is a general extension to the framework it might be added to the framework itself for other applications to use. New hooks can also be added to the addition and deletion methods of the Drawing class to indicate that the class can be extended easily at those points.

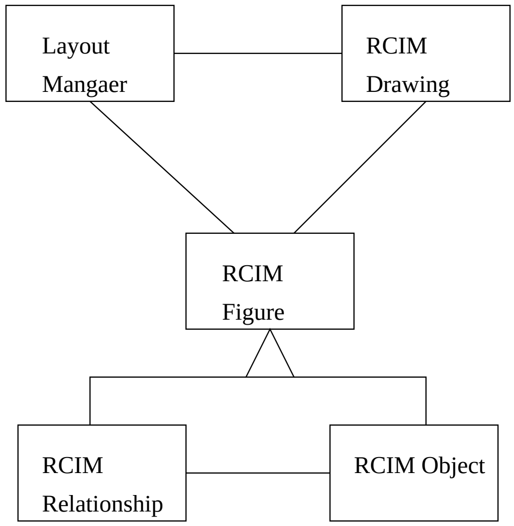

Figure 5: Inclusion of Layout in the RCIM Display Application.

4.5 Testing

Applications developed from frameworks can be tested just as any other application. The framework itself may come with tests pre-defined to determine if the application extensions are integrated properly with the framework. However, additional tests for the application specific functionality will always be needed. If errors are discovered, they can occur in three different places.

- In the interface to the framework: the framework may require a specific interface or that certain conditions be met by application extensions. Pre-conditions and post-conditions on all framework interfaces can help to highlight these errors when they occur and make them easier to track down.
- In the application: some errors will be within the application extensions. These errors tend to occur within the new functionality added to the framework.
- In the framework: the framework itself may have errors within it. Framework errors are probably the most difficult to find, since users will be unfamiliar with the fine details of the framework. When they do occur, developers have one of three difficult choices: fix the error (which may not be possible if the source code is not available), devise a work around, or stop using the framework.

5 Using Multiple Frameworks

So far we have focused on developing an application from a single framework. However, a single framework will not always cover every aspect of the application to be developed. Since small frameworks are generally preferred over large ones, sometimes more than one framework will be used in a single application. The same general techniques for using single frameworks apply, but two new problems arise: framework gap and framework overlap [Sparks et al., 1996].

Framework gap occurs when there are holes between the functionality provided by two different frameworks. One framework may expect data in a particular format that the other framework does not provide. Alternatively, one framework may expect some amount of processing to be done that the other framework does not do. In order to fit the frameworks together, application extensions have to be made to one or both to fill in the gaps. The extensions will do data conversion or extra processing so that the frameworks interface correctly.

Framework overlap is a more serious problem. In framework overlap, two frameworks provide the same functionality, or expect control over the same resources. An example of framework overlap occurs when two frameworks contain the main event loop for the application. To use the frameworks together, the overlapping functionality of one of the frameworks has to be removed or turned off, which may not always be easy or even feasible.

6 Evolving Frameworks

Just like any other software, frameworks evolve. However, little work has been done on framework evolution. We define evolution as any changes to the framework itself as opposed to additions or extensions made to the framework for specific applications. Frameworks evolve to fix errors, to add new functionality, or to restructure for more flexibility or ease of use. For example, the Tool class in the HotDraw framework has evolved to become more structured and easier to use by switching from requiring inheritance to a parameterized class. Frameworks also evolve as a normal part of the development process as more experience is gained in using them.

A problem faced by all reusable software is the potential side effects of changing the framework code for applications already built from earlier versions of the framework. In order to take advantage of the new features or fixes within the framework, the application has to be fitted to the new framework. If the application is not changed, then multiple versions of the framework have to be maintained, which undermines the ease of maintenance gained by basing a product line on a common framework. Ideally, the evolved framework can simply be inserted into the application and no changes will have to be made. The interfaces of the framework should be fixed and rarely changed [Taligent, 1995]. The hooks, which can be viewed as a special type of interface, should also remain relatively constant in order to make the upgrade easier. Obviously, this applies to frameworks which are used as is or completed. Applications which customize the framework by replacing parts of it will require much more work to fit into the new framework. Two means of evolving frameworks are refactoring and using design patterns.

6.1 Refactoring

Refactorings [Opdyke et al., 1990] are ways of restructuring a framework. They are behavior preserving transformations for evolving a framework, producing semantically equivalent references and operations. Refactorings, for example, cover aspects of creating new abstract superclasses, creating aggregate classes, changing class names, changing variable references and method calls, changing inheritance hierarchies, moving methods between classes and grouping classes together. Complex refactorings, such as moving a method between classes, are made up of smaller refactoring steps, such as updating all references to the method. Tool support is provided for the refactorings, and it aids in ensuring that all required changes are made. Using a refactoring guarantees that the behavior of the framework is preserved, although the interfaces and class structure may change. They are used for consolidating and simplifying the interfaces and functionality of the framework to make it easier to both use and refine.

6.2 Using Design Patterns

Design patterns can also be used to help evolve frameworks as a means of increasing flexibility. Johnson claims that as frameworks mature, they evolve from using inheritance to using composition [Johnson and Foote, 1988]. Many design patterns focus on composing classes together, and providing flexible and extensible solutions, so they are valuable for determining possible directions for framework evolution.

An example of using patterns to evolve a framework comes from the telecommunications domain [Hueni et al., 1995]. The initial framework is a white box framework that relies heavily on inheritance. Experience gained from using the framework helped them to identify areas where the framework could be made more flexible. Once they identified the type of flexibility required, they were able to apply appropriate design patterns to restructure the framework. Unlike refactorings, evolving a framework in this way does not guarantee that the behavior of the framework is preserved, but it does allow other types of changes to be made, such as extensions to the functionality of the framework.

7 Documentation

Framework documentation can aid in all aspects of framework use. Since the framework developers will not always be available, documentation becomes the means through which users can learn about the framework and refer to for information about the framework. A framework will typically have few developers and many users, so it is important that the documentation be made both understandable and up to date. Johnson [1992] proposes that three types of documentation are needed for frameworks.

- *The purpose of the framework.* A description of the domain that the framework is in, the requirements it is meant to fulfill, and any limitations it has.
- *The use of the framework.* A description of the way the framework builder intended the framework to be used.
- *The design of the framework.* A description of the structure and the behavior of the framework.

Framework documentation should not just consist of class diagrams describing the detailed design of the framework. While advanced users will require that information, learning all of that information will require too much effort for regular users, and so make the framework impractical for general use. Documentation describing what the framework is for and how it can be used will aid regular users in both learning and using the framework. The additional documentation shows users those parts of the framework they need to know about and those which they can treat as a black box in order to make efficient use of their time. Examples are also valuable for showing what the framework can be used for. Each of these types of documentation are discussed in the following subsections.

7.1 Purpose

There are no standard methods for documenting the purpose of a framework. Generally a high level overview of the framework is needed which serves as an introduction to the framework. The introduction can be used to help quickly decide if the framework is appropriate for the intended application. As a rough guideline, the purpose should include:

- Domain: the domain that the framework covers should be defined as clearly as possible. It is often difficult to pin down the exact domain that the framework covers, so giving examples of applications that can and have been built with the framework is a useful technique [Johnson, 1992].
- Requirements: the functional and non-functional requirements that the framework fulfills.
- Limitations: any limitations or assumptions about a framework's environment should also be stated, if possible. Putting the limitations up front prevent users from finding out halfway through a project that the framework they have chosen cannot support their requirements.

7.2 Intended Use

Beyond the general purpose of the framework, regular users are interested in how to use the framework. Documentation describing the intended uses of the framework helps users to learn the framework, and develop applications. Documentation of intended use should identify the problems that the framework solves or that users will face when using the framework, how to think about the problem in order to understand how to solve it, and the actual solution [Beck, 1994]. It captures the framework developer's experience and knowledge of how the framework can be used. These are valuable insights that otherwise would have to be gained by users through trial and error since this knowledge is not captured explicitly in the design and implementation of the framework. Several methods have been proposed for describing the intended use. In the following subsections, we introduce three documentation methods: motifs/patterns, cookbooks and hooks.

7.2.1 Motifs/Patterns

Johnson [1992] proposes that the experience and knowledge of framework developers about how the framework can be used should be captured by a series of patterns. Lajoie and Keller [1995] call these patterns motifs in order to avoid confusion with design patterns (discussed later in this section).

Each motif has the following parts:

- Name: the name identifies the motif and gives an indication of what problem the motif is intended to solve.
- Description of the problem: a brief description acquaints the user with the purpose of the motif. From the description, a potential framework user should be able to decide whether or not the motif matches the problem they want to solve.
- Description of the solution: the longest and most detailed section of the motif describes various potential ways of solving the problem. The description tends to be general and high level enough to accommodate a range of solutions. Often brief examples help the user to understand the solution.
- Summary of the solution: finally, the motif ends with a brief summary of how the problem is solved. It also points to other motifs which solve related problems.

The series of motifs starts with a general overview of the framework. The overview describes the purpose of the framework, what problems it is appropriate for and also gives a brief process for using the framework. For example, the following simple outline is given for using HotDraw: *"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."* [Johnson, 1992]

The motifs provide a high level narrative of how to solve various problems that application developers may encounter when trying to use the framework. The descriptions are informal and easy to read, but often leave out detailed information that a user may require. They are valuable for understanding the key concepts behind the framework. Each motif also links to other motifs, starting with the initial motif that describes the framework. The collection of motifs form a directed graph that moves from very general to more specific information about the framework and provides a way in which users can learn about the framework gradually without being overwhelmed by the details right away.

Lajoie and Keller [1995] combine the idea of design patterns with Johnson's patterns to provide a more complete description of a framework. In their strategy, motifs point to design patterns, contracts [Helm et al., 1990] and micro-architectures to help provide the developer with an understanding of the architecture of the framework in the context of the problems it was meant to solve.

Beck [1994] proposes a slightly different style of pattern. Each pattern starts out with a brief problem description. The next part describes the context of the pattern. The context is the set of constraints that affect the solution to the problem. Often these constraints conflict with one another and may seem to rule out a solution altogether. The description of the context is meant to give an understanding of exactly what the problem is and why the solution is structured in a certain way within the framework. The solution then describes what to do to solve the problem.

7.2.2 Cookbooks

Cookbooks for describing how to use frameworks has been around almost as long as frameworks themselves. At least two styles of cookbooks have been used. Pope and Krasner [1988] use a tutorial style in describing the basics of the Model-View-Controller framework in Smalltalk. The first parts describe the framework in general and subsequent parts describe specific parts of the framework in more detail. Finally, several examples using the framework are provided.

The second, and more common, type of cookbook is related to patterns. A cookbook of recipes [ParcPlace, 1996] is provided, with each pattern describing a problem and a series of steps to follow to solve the problem. Each recipe consists of the following parts:

- Strategy: the first section introduces the recipe, describes the problem it is meant to solve and may include some of the limitations of the solution. It provides all of the information needed to understand the context of the solution.

- Basic steps: the actual solution is captured as a sequence of steps to carry out. These steps include using development tools as well as actual code. Often a simple example is included to help demonstrate the solution.

- Variants: solutions that are essentially the same as the one given in the basic steps, but differing in some small way, are provided. The variants help to capture some of the alternative or advanced uses of the framework.

- See also: the final section directs the user to other descriptions that are related to the current description.

The recipes are provided as a catalogue and indexed by their name. Unlike motifs there is no reading order specified, so they form more of a reference than a learning tool. Like motifs, descriptions are informal so they are easy to read and usually easy to understand. The steps provided are more focused than the general notes given in a motif and therefore they provide more support to novice users.

7.2.3 Hook Descriptions

Since cookbooks and motifs are loosely structured narrative descriptions, they sometimes face the problem of being imprecise or lacking information needed by the user. Hook descriptions [Froehlich et al., 1997] provide a semiformal template for describing the intended use at a detailed level. The template helps to prompt the developer for all the required information and the semi-formal language makes the descriptions more precise.

The sections of the template detail different aspects of the hook, such as the components that take part in the hook (participants) or the steps that should be followed to use the hook (changes). The sections serve as a guide to the people writing the hooks by showing the aspects that should be considered about the hook, such as how using it affects the rest of the framework (constraints). The format helps to organize the information and make the description more precise and uniform. This aids in the analysis of hooks and the provision of tool support for them. Each hook description consists of the following parts:

- Name: a unique name, within the context of the framework, given to each hook.
- Requirement: the problem the hook is intended to help solve.
- Type: an ordered pair consisting of the method of adaption used and the amount of support provided for the problem within the framework.
- Area: the area(s) and parts of the framework that are affected by the hook.
- Uses: the other hooks required to use this hook. The use of a single hook may not be enough to completely fulfill a requirement that has several aspects to it, so this section states the other hooks that are needed to help fulfill the requirement.
- Participants: the components that participate in the hook. These are both existing and new components.
- Changes: the main section of the hook which outlines the changes to the interfaces, associations, control flow and synchronization amongst the components given in the participants section. While the changes focus on the design level, so as not to overly-constrain the developers, sometimes changes to

the code itself must be included. All changes, including those involving the use of other hooks, are intended to be made in the order they are given within this section.

- Constraints: limits imposed by using the hook, such as configuration constraints.
- Comments: any additional description needed.

Not all sections will be applicable to all hooks, in which case the entry not required is simply left out. For example, a hook that does not use any others will have no Uses declaration. An example of a hook is shown below.

Name: Create a New Composite Figure
Requirement: A new figure is needed which is composed of other figures, such as a text label within a rectangle.
Type: Enabling, Pattern
Area: Figures
Uses: SetFigures, SetConstraints
Participants: CompositeFigure, NewComposite
Changes:

```
new subclass NewComposite of CompositeFigure
property NewComposite.figures
NewComposite.figures = SetFigures
property NewComposite.figure_constraints
NewComposite.constraints = SetConstraints [NewComposite.figures]
NewComposite.constraints overrides CompositeFigure.constraints
    returns set of NewComposite.figure_constraints
```

The Create a New Composite Figure hook describes how to create new figures which are composed of other more basic figures within HotDraw. This hook was used to create the RCIM Figures within the RCIM graphical display application. The hook uses functionality provided by the framework by filling in parameters so its type is an enabling pattern. In this case the parameters are the figures contained in the composite figure and the constraints between the figures. NewComposite is defined as a subclass of the existing class CompositeFigure. NewComposite adds the figures and figure_constraints properties (instance variables in this case) and uses them to hold the component figures and constraints. The additional hooks SetFigures and SetConstraints are used to fill in the figures and constraints for the new class. For example, RCIM Object defines two component classes: Text Figure and Rectangle Figure. It also defines graphical constraints between the component figures so that the position of the text label remains constant with respect to the location of the rectangle when the location of the entire composite figure is changed. Finally, the constraints method is provided to return the set of constraints defined for a class so that they can be evaluated when needed.

Hooks are characterized along two axes: the *method of adaption* used and the *level of support* provided. The method of adaption is used to fulfill the requirement and describes the basic mechanism used to extend or adapt the framework. The level of support indicates how the change is supported within the framework, such as using provided components or requiring developers to produce their own components. For example, removing a feature, such as the use of tools in HotDraw, may be well-supported and simply require turning off a switch in the framework, or may be less well-supported and require the modification of one or more methods.

A hook's method of adaption quickly gives an application developer an idea of what the hook does. Its support type indicates how difficult it may be to use. The types can also serve as a basis for locating a desired hook. For example, if a developer needs to extend the framework in a particular area, they could retrieve all of the hooks that add a feature to that part of the framework. Each type indicates the issues that both the hook writer and the application developer must consider. For example, removing a feature such as the use of tools in HotDraw will often have an impact on other features of the framework, such as the ability for the user to manipulate the figures in a diagram.

Method of Adaption

There are several ways that a developer can adapt a framework and each hook uses at least one of these methods.

Enabling a feature that exists within the framework but is not part of the default implementation is one common means of adapting a framework. Hooks of this type often involve using pre-built concrete components that come with the framework which may be further parameterized. The hook needs to detail how to enable the feature, such as which components to select for inclusion in the application, which parameters to fill in, or how to configure a set of components. The constraints imposed by using the feature, such as excluding the use of another feature, are also contained in the hook.

Disabling a feature may be required if the default implementation of the framework has some unwanted properties. This is different than simply not choosing to enable a feature. Disabling a feature may be done through configuration parameters, or by actual modification of framework code. The hook description shows how to do the removal and it also shows the effects of the removal. For example, removing the use of tools from HotDraw significantly affects the capabilities of the rest of the framework.

Replacing or overriding an existing feature is related to disabling a feature, with the addition that new or predefined components are provided in place of the old. If the replacement requires the application developer to provide new classes or components then it is important to describe the interface and behavioral obligations that any replacement must fulfill. The replacement may also be a pre-defined component that the developer simply puts in place of the original component.

Augmenting a feature involves intercepting the existing flow of control, performing some needed actions, and returning control back to the framework. Unlike replacing behavior, augmenting simply adds to the behavior without redefining it. The framework builder can point to places in the control flow where a change to fulfill a particular requirement might be made, perhaps but not always by providing stub methods that can be overridden by developers. The hook describes any state that needs to be maintained, where to intercept the flow of control and where to return it.

Adding a new feature or service to the framework is another common adaptation and probably the most difficult to support. Unlike enabling a feature, where the developer is using existing services, possibly in new ways, adding a feature involves adding something that the framework wasn't capable of before. These additions are often done by extending existing classes with new services or adding new classes, and adding new paths of control with the new services. The hook shows what new classes or operations are needed, and indicates where to integrate them into the framework and how they interact with old classes and services. The framework builder may also provide constraints that must be met by the new class or service and which may limit the interfaces that the new class can use to interact with the framework.

Level of Support

Another important aspect of hooks is the level of support provided for the adaption within the framework. There are three main levels of support types for hooks.

The *option* level provides the most support, and is generally the easiest for the application developer to use. A number of pre-built components are provided within the framework and the developer simply chooses one without requiring extensive knowledge about the framework. This is the black-box approach to frameworks [Johnson, 1988]. Most often, components are chosen to enable features within the framework or to replace default components. If the solutions are alternatives, then the hook is a single option hook. If several alternatives can be used at once, then the hooks is a multi-option hook. For example, several existing tools can be incorporated into an application based on HotDraw using a multi-option hook.

At the *pattern* level, the developer supplies parameters to components and/or follows a well-supported pattern of behavior. Unlike option hooks, there are no complete pre-defined components to choose from, but support is generally provided for the feature through parameters to components. The simplest patterns occur when the developer needs to provide parameters to a single class within the framework. The parameters themselves may be as simple as base variables, or as complex as methods or component classes. Some common tasks may require the collaboration of multiple classes, and may also have application specific details. For these, a collaboration pattern is provided which the developer follows to realize the task. Both pattern and option hooks are well-suited for normal users of the framework because they do not require a complete understanding of the design of the framework.

At the *open-ended* level hooks are provided to fulfill requirements without being well-supported within the framework. Open-ended hooks involve adding new properties to classes, new components to the framework, new interactions among components or sometimes the modification of existing code. These modification often, but not always, are for more advanced users that have a greater knowledge of the design of the framework so that they are aware of potential problems the modification may cause. Since they are open-ended, the developer has to be more careful about the effects changes will have on the framework.

Hooks and Motifs

The level of detail and semi-formal notation makes the hooks more suitable for partial automation than motifs or cookbooks. Well-defined hooks at the option or pattern level might be fed into a tool which automatically applies them when selected and their participants are filled in by the user. Hooks can provide the basis for a flexible tool that can be applied to more than one framework. The tool can also be open-ended, since hook descriptions can be modified as the framework evolves, or new hooks can be added as they are documented over time.

However, this level of detail provided by hooks is not as useful for novice users who are just learning the framework. Hooks require a certain level of knowledge about the framework, and so may be confusing at first. Also, the hook descriptions do not present the forces involved in a solution, or trade-offs involved in alternate solutions.

Motifs and hook descriptions can be combined to provide a more complete description of the intended use of the framework. Their strengths complement each other nicely. A motif provides a general, relatively high-level description of a problem and its constraints, perhaps giving an overview of a particular hot spot within the framework. The motif can then refer to a number of more detailed and precise hook descriptions which are used to actually adapt the framework to an application.

7.3 Design

Documentation of the design is most useful to advanced users, maintainers and framework developers, since they need to know the details of the framework. Regular users may also need to understand some of the design of a framework. While normal methods such as Booch's method [1992], the Object Modeling Technique [Rumbaugh et al., 1992] or their unified effort, UML [Booch et al., 1997], can be used to document the design of a framework, they are often not sufficient for a full description. The problem with traditional notations is that the collaborative relationships between the core classes of the framework are not made explicit [Gangopadhyay et al., 1995]. In any object-oriented software, understanding the collaborations is important, but in frameworks it is crucial since frameworks are meant to be reused. The collaborations describe how the classes interact and often comprise much of the complexity of the framework. In order to better represent these collaborations, several specialized methods for documenting the design of an object-oriented framework have been developed. The following methods are briefly described in this section: multiple views, exemplars, design patterns, metapatterns, reuse contracts and behavioral contracts.

7.3.1 Multiple Views

Campell and Islam [1992] document the design of frameworks by providing several different views of the design. Each view captures a different aspect of the design and together they provide a view of both the structure and behavior of the framework. The views are:

- Class hierarchies: inheritance hierarchies explicitly show the root abstract classes and all derived concrete library classes.
- Interface protocols: the interface protocols are the methods that each class in the framework makes public. The protocols list the methods along with their return values and parameter types.
- Synchronization: path expressions specify the order in which methods of a class must be invoked. For example, the initialization method of a Figure must be invoked before it can be moved or resized. Path expressions can also define methods which should not be invoked together.
- Control flow: message passing in the run-time behavior of the framework is represented using control flow diagrams. Each diagram presents the classes and methods invoked in a particular operation and the sequence in which messages occur.
- Entity relationship model: the relationships between objects in the framework and the cardinality of those relationships are shown using E-R diagrams. Optional and mandatory links can also be expressed.
- Configuration constraints: a framework may have components that are not meant to be used together but are included in the framework library for the purposes of completeness. Components are labeled with a type attribute and a Venn diagram is used to show which types can or cannot be used together in a single application.

There are many views in this approach and integrating all of the information can be a problem, so this method is not as popular as some of the others. However, none of the other methods describe configuration constraints which can be important.

7.3.2 Exemplars

An exemplar consists of at least one instance of a concrete class for every abstract class in the framework [Gangopadhyay et al., 1995]. Collaborations between the objects are explicitly and formally represented, so the collaborations can not only be viewed, but the entire exemplar can be executed as well. A visual tool allows framework users to interactively explore the relationships between the instances and so learn the interconnections between the main classes of the framework. The exemplar is provided by the framework developers with the framework and application developers can use the exemplar by following several steps.

1. Exploration: the visual tool is used to explore the framework. The tool shows the static relationships between classes. It also allows the user to observe message passing between the objects to provide an understanding of the run-time behavior of the framework. The user has the advantage of actually being able to see the framework execute in a limited way to gain a deeper understanding of how the framework works.

2. Selection: after the framework user has gained an understanding of the framework through exploring the exemplar, the user can select objects from the exemplar that need to be replaced or modified to fit the requirements of the application being built.

3. Finding alternatives: the tool allows the user to explore the inheritance hierarchy for a selected object. It displays the abstract class and the framework library classes derived from the class that correspond to the object and the user can select the one that is needed for the application. By restricting the search to the inheritance hierarchy of the selected object, the tool cuts down on the number of classes the user has to search through.

4. Adaptation: if no appropriate replacement class exists within the framework, then a new one has to be created. The replacement can, for example, be a subclass of the existing abstract class corresponding to the object selected, or it can be a composition of more than one existing class.

An object of the replacement class can then be inserted into the exemplar model, and can be executed to see if it has the desired behavior. Framework users repeat this process for every object they wish to replace and can, in part, prototype the application by selectively replacing objects. The authors call this process *Design by Framework Completion*. By substituting classes directly, application developers are able to use the framework as is, but there is no direct support for extending the framework in new ways. However, the understanding of the framework gained by exploring the exemplar helps developers to use the framework in new ways.

7.3.3 Design Patterns

A design pattern [Gamma et al., 1995] is a solution to a common software design problem in a given context. Design patterns are valuable because they capture the expertise of software developers, expertise that can be used in education, in design, and in the documentation of object-oriented systems. Many frameworks are designed using design patterns and the patterns used serve as a form of documentation for the framework.

Design patterns consist of four general parts, although the exact format often varies between pattern authors.

- Pattern Name: the name gives a brief image of the pattern. The names is important because it becomes a part of a developer's vocabulary. The name provides a means of describing a potentially complex set of classes and their collaborations in a single word or two. In effect they can become an important and standard means of communication between and among framework developers and framework users.

- Problem: the problem is a description of both the problem that the pattern is meant to solve and the context in which the pattern is applicable. The context is often described as a set of conditions that must apply before the pattern can be used.

- Solution: the main part of the pattern describes the static structure of the pattern and the collaborations between the classes it contains. The solution is a template rather than a specific implementation. Since the problem descriptions are general, the template approach typically allows for a range of variations to fit the specific problems to be solved in an application or framework.

- Consequences: every pattern has some effect on the overall design of a framework or application, and the consequences section describes those effects. Knowing the limitations of the design patterns that a framework uses helps users decide if the framework is appropriate and can prevent users from trying to use the framework in inappropriate ways.

Design patterns aid in the understanding of frameworks in several ways. First, they provide a description of the design. The description includes both the classes and the collaborations between the classes which is crucial to understanding the framework. The patterns can also make a complex design much easier to understand since they raise the level of granularity at which the design is viewed. Instead of viewing a framework as a collection of low level classes, the framework can be viewed as a collection of interwoven patterns, each of which contains several classes that interact in standard ways. The patterns provide a shared vocabulary between framework developers and framework users that can help users to more quickly gain an understanding of the overall structure and behavior of the framework.

Second, each design pattern specifically states the problem it solves and the context in which it is used, so it can help to convey the intent of the design [Beck and Johnson, 1994]. The record of why certain design decisions were made can help the user to understand how the framework can be used, and how it should not be used (what would break the framework). The design patterns can also help maintainers during evolution of the framework to understand the assumptions behind the design and to change the design if the assumptions no longer hold. Finally, a kind of tutorial can be provided using design patterns to help users to learn the framework. An initial problem the framework solves is stated and the design pattern used is described. Subsequent problems and pattern descriptions build on earlier descriptions to slowly build up a view of the architecture and behavior of the framework.

Finally, design patterns can be used to explore or visualize the design of an existing framework [Lange and Nakamura, 1995]. By using a visual tool, users can view the framework classes within a design pattern and even watch the execution of the framework through the patterns. However, research into visualization of design patterns is still in an early stage, and users must find and identify the patterns within the framework themselves, limiting the usefulness of the approach as a learning tool.

7.3.4 Metapatterns

Metapatterns [Pree, 1995] use meta abstractions to describe ways of designing flexible connections between classes. They are called metapatterns because a design pattern can usually be described using a combination of metapatterns. However the metapatterns do not capture the specific problem and context information inherent in design patterns.

Each metapattern identifies a relationship between a template method which is defined in the framework, and a hook method that is left open for the application developer to complete. The class that contains the template method is called the template class and the class that contains the hook method is the hook class. Hook methods and hook classes are not the same as the hooks described earlier. Hooks are a means of adapting a framework, whereas hook methods are something to be adapted. A simple example of a metapattern used in HotDraw occurs between the DrawingController and the Drawing. In order to animate a drawing (such as having a Figure blinking in the RCIM graphical display application) the DrawingController calls the hook method step in Drawing. It is an example of the 1:1 connection pattern since each DrawingController is associated with a single Drawing.

Metapatterns can help document frameworks in the same way as patterns, and can be an aid to advanced users, and framework evolvers. Since not every part of the framework will have a corresponding design pattern to describe it, metapatterns can be used to help document the other areas.

7.3.5 Reuse Contracts

Reuse contracts [Steyaert et al., 1996] form a specialization interface between a class and subclasses developed from it. They consist of a set of descriptions of abstract and concrete method that are crucial to inheritors while hiding implementation specific details. Specialization clauses can be attached to methods. They consist of a set of methods that must be invoked by the method being specialized, documenting all *self* method invocations of a method. The methods within the specialization clause are called hook methods and have the same purpose as Pree's hook methods (methods to be filled in by application developers).

Reuse operators formally define how abstract classes are used by defining relationships between the reuse contract of an abstract class and the reuse contract of its subclass. Three reuse operators are defined:

- Refinement: overriding method descriptions to refine their functionality. New hook methods are added to the specialization clause of the method being overridden while keeping all of the existing hook methods. The specialization clause is extended and the behavior of the method is refined. The inverse of refinement is called coarsening. Coarsening is used to remove hook methods from a specialization clause.
- Extension: adding new method descriptions to a reuse contract. The new methods contain functionality specific to a framework library, or application class. If the new methods are all concrete, then the extension is concrete, otherwise it is abstract. The opposite of extension is cancellation in which unwanted method descriptions are removed from the reuse contract.
- Concretisation: overriding some abstract methods within the contract with concrete ones. This is often done by application developers when producing an application. The opposite of concretisation is abstraction in which concrete methods are defined as abstract. Abstraction is useful for forming abstractions when developing frameworks.

Reuse contracts can help to specify what methods need to be specialized in an abstract class, and define operators for creating new classes from an abstract class. In that way, they define how the class can be used. However, they primarily document how a subclass relates to its parent class, defining and documenting the changes between the classes. Reuse contracts are useful when frameworks evolve. The effects of changes to a parent class can be propagated down to all of its child classes through the reuse contracts. Using the relationships between reuse contracts defined by the operators, the contracts can indicate how much work is needed to update child classes, including those in previously built applications, when a framework changes.

7.3.6 Behavioral Contracts

Behavioral contracts [Helm et al., 1990], are used to help design a framework and are also valuable for learning the framework and ensuring that the framework is correctly used. The contract defines a number of participants, corresponding to objects, and defines how they interact. They form templates which can be similar to design patterns. Each contract consists of the following parts.

- Contractual obligations: the obligations define what each participant must support. The obligations consists of both type obligations (variables and interfaces) and causal obligations. The causal obligations consist of sequences of actions that must be performed and conditions that must be met.

- Invariants: the contract also specifies any conditions that must always be kept true by the contract, and how to satisfy the invariant when it becomes false.
- Instantiation: preconditions form the final part of the contract which must be satisfied by the participants before the contract can be established.

Behavioral contracts can aid both regular and advanced users in understanding how objects in the framework collaborate. A general contract can often be used in a new context, much like a design pattern, and so new framework developers will be interested in them as well. An example of a contract is given in the chapter on designing object-oriented frameworks.

7.4 Examples

Examples provide another means of learning the framework and complement all of the other types of documentation. The examples can be complete applications developed from the framework, or smaller examples to demonstrate how a particular hot spot can be used, or how given design pattern works. The examples are valuable because they make an abstract framework concrete and easier to understand. They provide a specific problem and show how the framework can be used to solve that problem. Examples are also valuable to framework evolvers for regression testing of the framework

An example application might also be modified into a new application if it is similar enough to the desired application. However, examples cannot cover every possible use of the framework and so the other forms of purpose, intent and design documentation are all still necessary.

8 Conclusions

Frameworks provide a means of capturing the knowledge of domain experts and packaging this expertise in a form that can be used by many different users. Regular users who are not experts in the domain can build quality applications based on the framework simply by configuring pre-defined components within the framework, or by completing parts of the framework left open.

Individual applications can be developed more quickly and easily using the framework since it provides much of the implementation of the application already. Entire product lines can be quickly developed from a single framework and can be more easily maintained due to the common design and code base.

Documentation is important for supporting all types of framework use. Documentation should describe the purpose of the framework in order to aid users in deciding whether or not to use the framework. It should describe how to use the framework to aid regular users to quickly develop applications from the framework. It should also describe the design of the framework to aid advanced users, maintainers and developers of new frameworks. Examples can be used to both explain ways in which the framework can be used, and to test the framework when it evolves.

Tool support is one of the most promising areas that needs to be investigated. Many uses of the framework can be described as standard patterns and tools can make a framework much easier to use by incorporating those patterns and performing all of the tedious work associated with them. Tools can make the use of a framework less error-prone. However, tools also have to allow for non-standard extensions to be made to a framework, which is much more difficult to support.

Issues of integrating multiple frameworks remain open. As frameworks become more popular, applications will be developed using more than one framework and problems of integrating those frameworks will arise.

Techniques for framework evolution also need to be devised. Little work has been done concerning how frameworks evolve and the effects of evolution on applications. Anyone who has used a graphical user interface framework and had to change their application when a new version of the framework broke the application can appreciate the need to devise methods of evolving frameworks that require little or no changes to the application extension interfaces.

References

[1] Beck, K. 1994. Patterns and Software Development. *Dr. Dobb's Journal.* 19(2):18-22.

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

[3] Booch, G. 1994. *Object-Oriented Analysis and Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, California.

[4] Booch, G., Jacobson, I. and Rumbaugh, J. 1996. *The Unified Modeling Language for Object-Oriented Development.* Rational Software Corporation (http://www.rational.com/uml/).

[5] Brown, K., Kues, L. and Lam, M. 1995. HM3270: An Evolving Framework for Client-Server Communications. In *Proceedings of the 14th Annual TOOLS (Technology of Object-Oriented Languages and Systems) Conference.* 463-472.

[6] Bulter, G. and Denomme, P. 1997. Documenting Frameworks. *Proceedings of the Eighth Annual Workshop on Institutionalizing Software Reuse (WISR).*

[7] Campbell, R.H. and Islam, N. 1992. A Technique for Documenting the Framework of an Object-Oriented System. In J*Proceedings of the 2nd International Workshop on Object-Orientation in Operating Systems.*

[8] R. H. Campbell, R.H., Islam, N., Raila, D. and Madany, P. 1993. Designing and Implementing Choices: An Object-Oriented System in C++. *Communications of the ACM.* 36(9):117-126.

[9] 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.*

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

[11] 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.

[12] Helm, R., Holland, I.M. and Gangopadhyay, D. 1990. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. *Proceedings of ECOOP/OOPSLA 90.* 169-180.

[13] Hueni, H., Johnson, R. and Engel, R. 1995. A Framework for Network Protocol Software. In *Proceedings of OOPSLA'95.*

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

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

[16] Krasner, G.E. and Pope, S.T. 1988. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming.* 1(3):26-49.

[17] Lajoie, R. and Keller, R.K. 1995. Design and Reuse in Object Oriented Frameworks: Patterns, Contracts, and Motifs in Concert. In *Object-Oriented Technology for Database and Software Systems*, V.S. Alagar and R. Missaoui (eds), World Scientific Publishing, Singapore. 295-312.

[18] Lange, D.B. and Nakamura, Y. 1995. Interactive Visualization of Design Patterns Can Help in Framework Understanding. In *Proceedings of OOPSLA'95.* 342-357.

[19] Opdyke, W. and Johnson, R. 1990. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. *Proc. of Symposium on Object-Oriented Programming Emphasizing Practical Applications.*

[20] *VisualWorks Cookbook.* 1995. Release 2.5, ParcPlace-Digitalk Inc., Sunnyvale, CA.

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

[22] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and W. Lorenson. 1991. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., Englewood Cliffs, NJ.

[23] Schmid, H.A. 1995. *Creating the Architecture of a Manufacturing Framework by Design Patterns.* In Proceedings of OOPSLA'95 Austin, TX, 1995, 370-384.

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

[25] Steyaert, P., Lucas, C., Mens, K. and D'Hondt, T. 1996. *Reuse Contracts: Managing the Evolution of Reusable Assets.* Proceedings of OOPSLA'96. 268-285.

[26] Tse. L. 1996. *A Responsive Corporate Information Model.* Ph.D. Thesis, University of Alberta, Edmonton, AB, Canada.

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

[28] 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.

[29] A. Weinand, A., Gamma E. and Marty, R. 1988. ET++ - An Object-Oriented Application Framework in C++. In *Proceedings of OOPSLA'88.* 46-57.