University of Alberta

Hooks: An Aid to the Use of Object-Oriented Frameworks

by

Garry John James Froehlich ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of *Doctor of Philosophy*

Department of *Computing Science*

Edmonton, Alberta
Fall 2002

0-612-81191-3

University of Alberta

Library Release Form

Name of Author: *Garry John James Froehlich*

Title of Thesis: *Hooks: an Aid to the Use of Object-Oriented Frameworks*

Degree: *Doctor of Philosophy*

Year this Degree Granted: *2002*

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

[illegible]

112 E 26th Ave
Vancouver, B.C.
Canada, V5V 2G9

Date Sept 23 2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled Hooks: an Aid to the Use of Object-Oriented Frameworks submitted by *Garry John James Froehlich* in partial fulfillment of the requirements for the degree of *Doctor of Philosophy*.

[illegible]

Dr. Paul Sorenson

[illegible]

Dr. Jim Hoover

[illegible]

Dr. Eleni Stroulia

[illegible]

Dr. Marek Reformat

[illegible]

Dr. Jan Bosch

Date August 27, 2002

Abstract

Software reuse is a promising field that advocates reusing existing software instead of continually rebuilding everything. Application frameworks are one type of reusable software. They provide a standard template for a particular type of program, such as a graphical editor or database. The template provides all of the basic features and is then modified into a finished program, much in the same way that a new car can be designed by modifying a standard template.

Frameworks; however, are often large and complex pieces of software. In the literature, work has already been done to show what the pieces of the framework are, how they fit together and what each of them does. However, wading through all of the design and implementation information is an arduous process and in the end does not describe one of the most important aspects of a framework: how to use it. I am proposing a notion called hooks which correspond to all of the places in a framework that can be modified or added to in some way: the buttons, knobs and dials of a framework. Hooks let framework users quickly grasp the information they need to actually build applications with a particular framework. I am also categorizing hooks based on the type of change (such as adding parts to a framework or customizing existing parts) and how well the change is supported in the framework. This type of categorization has not been done in the past, and no one has studied the properties and effects of these different types of change.

The hooks model itself has been applied to several frameworks including the commercial Size Engineering Application Framework. In addition, an in-house framework has been constructed for client-server computing and used as the basis for a study in the senior year software engineering course to learn how people can best approach the use of frameworks and to test the validity of hooks.

The knowledge gained from this work helps users of the framework by providing the information and guidance they need to build finished programs from the framework. It also helps the people who develop frameworks by providing them with the knowledge they need to design frameworks that are flexible and easy to use.

Acknowledgment

The author would like to thank his supervisors, Dr. Paul Sorenson, Dr. H. James Hoover and former co-supervisor (before she moved on to another position) Dr. Ling Liu for their guidance, inspiration and especially their patience and perseverance in seeing this thesis to completion.

The author would also like to thank his family who made it possible to pursue graduate studies. Finally, the author would like to thank Christine, his wonderful wife. Yes, everyone, it is finally done.

Table of Contents

List of Tables

List of Figures

Chapter 1 – Introduction

The notion of software reuse is simple: instead of writing the same software over and over again, write it once in such a way that it can be used again and again. Its potential benefits are very appealing allowing developers to reduce the amount of time needed to bring new products to market, to ensure the quality of new software by using quality pieces of reusable software, and to incorporate the sometimes hard won expertise of developers into reusable pieces of software so that new developers can benefit from that expertise.

As a concept, the reuse of software has also been around for a long time. Krueger gives many examples of types of reusable software [Krueger 92]. High level languages were an early and very successful attempt to reuse programming notions such as loops and arrays. Their usage is now so common, that this is no longer considered to be a form of software reuse. Another early practice was code scavenging in which programmers often reused code by cutting from one application and pasting into another and then modifying the code to fit the new application. However, much like grabbing pieces from several different models of cars from a scrap yard and then welding them together, the scavenged pieces of software often don't fit together very well.

Two basic criteria are critical to software reuse:

1. Software to be reused must be built to be reused. It must be generic and abstract enough to be used in several different applications.
2. A piece of generic reusable software won't fit any specific problem exactly. It must be customizable.

Many new techniques have been proposed to provide clean pieces of reusable software that are designed to be reused. At the code level, components are formed as packaged software behind well-defined interfaces. Components are an attempt to divide software into manageable and reusable pieces, and the interfaces allow the components to be assembled relatively cleanly. Component models, such as Java Beans (http://java.sun.com/products/javabeans/) are a common part of software development environments. In addition, there are research models such as 3C [Edwards 90] to describe components. Components provide for the second criteria above (customization) through modifiable properties.

At the design level, design patterns [Gamma et. al. 95] capture the experience of others. Each pattern describes a solution to a common problem, and captures the expertise of other developers. The general solutions encapsulated in each pattern can be customized or adapted to a specific problem.

Object-oriented frameworks, the focus of this thesis, combine both the code and design levels to provide a more powerful form of reuse. An object-oriented application framework tends to be much larger and more complex than individual software components or design patterns. In fact, it is often composed, at least in part, of many different components and patterns. Beck and Johnson define a framework as the reusable design of a system or subsystem implemented through a set of abstract classes and their collaborations [Beck and Johnson 94]. Frameworks provide several advantages over other types of reuse:

- Frameworks contain both the components and the architecture (context) in which they are to be used. Conflicts generated by choosing incompatible components are avoided.
- Frameworks solve larger-grained problems than individual components or design patterns. This makes the effort of finding and reusing them much more cost effective than for small components.

Frameworks currently exist in areas such as graphical editors (HotDraw [Johnson 92], UniDraw [Vlissides 90]), user interfaces (ET++ [Weinand et al. 88]), manufacturing systems (OSEAFA [Schmid 95]), client-server communications [Brown et al. 95], and operating systems (Choices [Campbell et al. 92]).

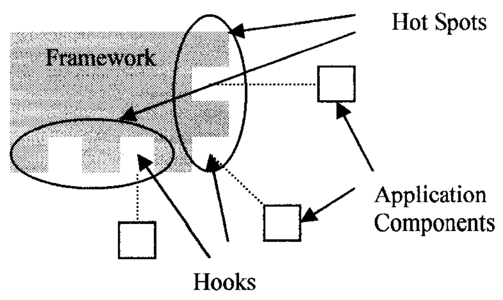

Figure 1.1: Framework and Hooks

Frameworks are designed to be generic and abstract, satisfying the first criterion given earlier in this section. They incorporate the notion of variability in order to allow developers to customize a framework to solve a specific problem. Variability can be defined at several levels as shown in Figure 1.1. The levels important to this thesis are:

- Hot spots [Pree 95] are simply the areas within the domain of the framework that are open and meant to be customized in some way. An example of a hot spot in a common user interface framework is the menu system (the menu bar at the top of most window-based applications).
- A hot spot can contain several hooks [Froehlich et al. 97]. A hook, a concept introduced in this thesis, is a set of variation points that are used together to perform some task on the framework (such as adding a new menu item in a graphical user interface framework).
- A hook description presents the steps that must be performed in order to utilize a hook. In the past, describing how to use the framework has been done in prose or point form and such descriptions can be vague or incomplete. A hook description provides a structured template that describes in a pre-defined grammar the changes necessary to adapt the framework to fulfill a particular requirement.

- Variation points are individual places within the framework where choices are meant to be made about the design and implementation of an application built from the framework. A variation point can be something as small as the name of a menu item in the graphical user interface framework, or as large as choosing a complete communication protocol.

Describing how to make use of the variation points within hot spots through hook descriptions is a key to framework use. *My primary thesis statement is: including knowledge about how a framework is intended to be used, through hook descriptions, makes frameworks easier to understand and use.* Hooks are meant to apply to frameworks in general, but the research to support the thesis statement has been done on single, sometimes called monolithic, frameworks (as opposed to collections of interacting frameworks). We believe that the work applies to frameworks in general, but cannot make more general claims. The work primarily applies to the use of frameworks, but as part of the research, data has been collected about the impact of the use of hooks on the design and evolution of frameworks as well.

1.1 Using Frameworks

Due to their potentially large size and complexity, frameworks have a steep learning curve. The ability to quickly understand and apply a framework is a critical issue. A framework, like any other type of reusable software, should take less time to understand and use than it would take to build an equivalent application without the framework. In our studies on frameworks we discovered that developers first need to understand the overall model of the framework and then understand the details of how and where applications interact with the framework. They also need to ensure that their application requirements match the functionality provided by the framework.

It's very easy to get lost when faced with the challenge of understanding and making enhancements to a large code base such as a framework. The code itself doesn't easily give up the secrets of what it does and how it should be used. In most cases, the framework contains little information about how the builders of the framework intended the software to be used. Without any sort of guidance, users have to dig through the code and examples to find out on their own where and how to add their application specific components. Hooks are targeted squarely at the interactions between frameworks and applications. We found that, in the process of completing this thesis work, hooks proved useful for describing to the developers where and how their application should interact with the framework and thus allowing them to make fewer and less critical mistakes when using the framework. They also were useful for reviewing whether or not the developers used the framework correctly, and for constructing examples of use.

In addition, hooks provide the basis for building tools to support not just a single framework, but frameworks in general. Since the hook notation has a defined grammar, it can be parsed by a machine and used in a tool to aid development using frameworks.

1.2 Designing Frameworks

Beyond their advantages for framework use, hooks affect framework design as well. It is generally acknowledged that the hot spots of a framework have to be considered and planned for at all stages of the development of a framework from collecting the requirements to implementation and testing [Pree 1995]. Since developers are familiar with designing and constructing applications, frameworks tend to be designed like applications. There has been some experience building software libraries and components, but frameworks require not only the ability to produce flexible and reusable components, but also flexibile and reusable connections between components. Additionally, modern programming languages and design notations lack the mechanisms for describing properties such as flexibility. For example, you can't describe how something is extensible in (standard) UML [Rumbaugh et. al. 99], and you can't describe how a component should be used with just the Java language.

Hooks provide a notation for stating in a structured template all of the properties and conditions in support of a particular extension to, or modification of, the framework. The usability and extensibility is captured by the hook. The hooks describe the ways in which the framework can 'bend' and the ways in which it cannot.

The description of the hooks provides other benefits. They provide a means of inspecting the intended usage of a framework so that errors of excess complexity can be caught and fixed. Further, they can provide the basis for generating test cases for applications built from the framework.

1.4 Thesis Outline

The remainder of this thesis describes the details of the hooks model and its application to object-oriented frameworks. The focus of the work is on representing knowledge about the reuse of object-oriented frameworks to build applications. The model was evaluated in several different ways:

- By applying the model to an existing standard research framework called HotDraw, and testing it on others such as MFC and Swing.
- By applying the model to the construction of a new framework (developed by the author of this thesis), the Client-Server Framework (CSF).
- By presenting a framework and its hooks to several teams of developers for use.
- By applying the model to existing frameworks in conjunction with the developers for the purpose of documentation and evolution using the

Size-Engineering Application Framework (SEAF) and the Sandwich framework.

The remainder of this document is laid out as follows:

- Chapter 2 provides more background on frameworks and research work related to framework documentation and use.
- Chapter 3 presents the means of describing hooks and a discussion of the types of hooks.
- Chapter 4 discusses the design issues related to hooks and frameworks from our experience using several frameworks, including HotDraw, CSF, Sandwich and SEAF.
- Chapter 5 describes a case study in which CSF was given to several groups of developers to help evaluate the usefulness of hooks.
- Chapter 6 outlines work on a tool to support the use of hooks.
- Chapter 7 gives the conclusions, contributions and future work of the thesis.

Chapter 2 – Background

2.1 An Example: HotDraw

To motivate the use of frameworks, we started with a simple problem. We wanted to build a program to dynamically display Responsive Corporate Information Model diagrams [Tse 96] to be called the RCIM display application. We wanted to build the program as quickly as possible, which meant looking for some reusable components and/or frameworks.

RCIM is an aid to corporate strategic planning. Strategic planning here is the identification of long term goals of a corporation and the fine-grained steps that are needed to reach those goals. It manages several alternative and/or interrelated plans consisting of objectives of the business. The plans can be represented as diagrams with simple arcs from a goal to its subgoals and markers on the arcs to indicate the importance of a subgoal and any alternative subgoals. These diagrams were intended to be integrated with the existing forms-based editor for RCIM that is written in Smalltalk.

In the RCIM graphical display application we have several general requirements. Two views are provided, one for strategic goals and one for constraints on goals, consisting of objects (the actual goals or constraints) and the relationships between them. 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 that is performed automatically by the program.

To build the RCIM graphical display, we looked at a framework called HotDraw. HotDraw [Johnson 92] is a simple framework from the University of Illinois at Urbana-Champaign for developing structured graphical editors written in Smalltalk. A view of a HotDraw application is shown in Figure 2.1. The **major** elements of HotDraw on which we focus are the tools (shown in the toolbar on the left) which are used to manipulate figures (the polygons within the window on the right) within a drawing (represented by the entire drawing window on the right). The tools provided in HotDraw include the selection tool for moving and manipulating figures, the deletion tool (shown as an eraser) for removing figures, and many tools for drawing different types of basic figures such as circles or rectangles.

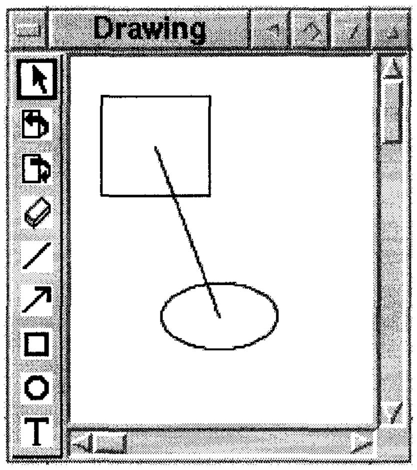

Figure 2.1: Basic View of the HotDraw Framework

However, Figure 2.1 does nothing to show that HotDraw is any different from a typical drawing program. Being a framework, it is *incomplete*. It is meant to be extended. It is meant to be used to construct diagram editors of all sorts. The diagram editors could be for anything from data-flow diagrams to organizational charts.

An object-oriented framework can be defined as the reusable design and implementation (in an OO language) or a system or subsytem [Johnson and Foote 88]. The key aspect is that HotDraw is not simply a collection of unrelated library components, but a well-designed system.

The framework (as with all frameworks) has what are called hot spots and frozen spots [Pree 95]. Hot spots, also called hinges [Fayad and Cline 96], are the general areas of variability within a framework where the framework can be extended. 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.

In contrast, frozen spots within the framework capture the commonalties across applications. They are fully implemented within the framework and typically have no extensions associated with them. In HotDraw, the drawing controller is an example of a frozen spot. The Tools used may vary, but the drawing controller remains a constant underlying mechanism for interaction.

The functionality provided by HotDraw matched well with the requirements to build a graphical display application for the RCIM diagrams. Additionally, work was done to find and document the hooks within the HotDraw framework. These hooks were then used in the application. HotDraw provides Figures to represent the objects and relationships. It provides a mechanism 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 our 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.

We used the capabilities for displaying figures and animation while adding support for limited automated layout of diagrams. 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 completed application simply consisted of a few classes built on top of HotDraw, while reusing the majority of the framework's functionality.

2.2 Ways to Use a Framework

Our approach incorporated a number of the different ways in which to use a framework. Each of them requires a different amount of knowledge about the framework and a different level of skill in using it. Taligent [Taligent 95] 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. This is how we used HotDraw to build the RCIM display application.

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.

2.3 Types of Frameworks (where are the frameworks?)

While there usually isn't a shelf at the local software store where you can pick up the latest framework, frameworks are already a prevalent part of software development. The first commercial frameworks appeared in the 1980s with the appearance of MacApp [Schmucker 86] and the Smalltalk environment [Goldberg and Robson 89] for application development. Recently there has been a boom in product-line architectures. Companies such as Cummings, Hewlet-Packard, Celsius Tech Systems and others have built upon the notions of large-scale reusable software to quickly put out new products [Clements and Northrop 01]. A product-line architecture is essentially a base architecture that covers a domain (diesel engine control systems, printers, or ship controls systems). The architecture is typically implemented through a number of components and frameworks [Bosch 00][Jazayeri et al. 00]. The idea derives from work done by Parnas [Parnas 77] on program families.

Frameworks are also almost ubiquitous in support of development platforms. Microsoft, Borland and IBM all ship frameworks in support of their compilers. The Java language has a number of frameworks to support rapid application development.

Of course, there are also a number of frameworks in the research community, such as HotDraw and ones for speech recognition [Srinivasan and Vergo 98], networks [Hueni et al. 95], and higher-order functions [Laufer 95].

What are the important properties of frameworks? 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.

2.3.1 Scope

The scope of the framework describes how broad an area the framework is applicable too. Adair [Adair 95] 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 such as MacApp or MFC or even the San Francisco project [Carey et al. 00] 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. 92], manufacturing systems [Schmid 95], client-server communications [Brown et al., 1995] and financial engineering [Eggenschwiler and Gamma 92]. HotDraw is another such example.

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. The Client-Server Framework is an example of a support framework. The line between Application and Support frameworks is a fuzzy one though, and the two have been combined to be called foundation frameworks [Hoover et al. 00].

2.3.2 Customization

The means of customizing is another way in which frameworks can be categorized. Johnson and Foote [Johnson and Foote 88] define two types of frameworks, white box and black box. While this is an important dichotomy, a framework will often (almost always) 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 95] 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 95], 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.

2.3.3 Interaction

Andersen Consulting [Sparks et al. 96] differentiates frameworks based on how they interact with the application extensions, rather than their scope or how they are customized. However, much like black-box vs. white-box, frameworks tend not to fall into one category or the other, but combine both styles.

Called frameworks correspond to code libraries (such as the Eiffel libraries [Meyer 94] or Booch's libraries [Booch 94]). Applications use the framework by calling functions or methods within the library.

Calling frameworks, the category into which most traditional frameworks belong, 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 thesis we will be primarily focusing on calling frameworks, although much of the material applies to called frameworks as well.

2.4 Benefits of Frameworks

Using frameworks as a basis for building applications has several advantages centering around the qualities of reusing a large scale design and implementation.

2.4.1 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 92] 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.

2.4.2 Maintenance

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

2.4.3 Quality

The framework not only provides a reusable design, but a tested design proven to work. It therefore forms a quality base for developing new applications.

However, there is one major concern with the use of frameworks: the learning curve.

2.5 The Learning Curve

A small study has shown that small frameworks improve overall development time [Chen and Chen 94]. Other authors have indicated that the use of frameworks at the enterprise level improves development [Hamu and Fayad 98]. As indicated previously, some companies are having good success with product line architectures in some areas. However, developers have to learn the framework first and frameworks can be quite complicated and difficult to understand. Taligent has developed a system of frameworks called CommonPoint [Cotter and Potel 95] which provides support for tasks common to modern window-based applications, such as 2D and 3D graphics, compound documents, multi-user collaboration, and printing. CommonPoint consists of over one hundred small and interconnected frameworks with over 600,000 lines of code in total. During beta testing it was reported that developers required an average of four to five months to learn enough about Taligent's CommonPoint application framework system to use it effectively [Myers 95].

Learning how to use a framework given only the code and interface descriptions is a daunting task and makes framework use unattractive. Essentially it becomes a matter of program comprehension. Program comprehension is the reconstruction of the logical structure and goals that were used in writing a program to understand what the program does and how it does it. [Boehm-Davis 88][Brooks 83]. It makes sense to clearly document just what the framework can do.

Further, the effort of using a framework is highly dependent on how well that framework supports the desired functionality of the application. In extreme cases, if the framework does not support the functionality desired, using the framework can be more costly than not using the framework [Bosch 99]. A framework should be easier to use as the basis for an application than building a new application without the framework. A means of lowering the learning curve is needed. Some of the major approaches are detailed in the following subsections below.

2.5.1 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, or to reduce the loss of framework expertise because of the departure of developers, other methods are needed.

2.5.2 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. 96].

2.5.3 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 adaptations 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 92]. It allows users to build new tools simply by filling in the appropriate parameters and then automatically integrating the new tool into an application. More complex tools such as the one provided for OSEFA [Schmid 96] 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.

More general framework tools are also appearing. FRED [Hakala et al. 01b] is a framework editor that relies on common patterns in the framework to guide the user with fine grained development steps. The framework must be annotated with specialization patterns [Hakala et al. 01a] which describe the steps needed to implement some desired functionality. The patterns indicate when methods or classes need to be created or inherited from existing framework classes, and describe code snippets that implement desired functionalities.

Similarly, HiFi [Campo et al. 97][Ortigosa and Campo 99] uses intelligent agents to help provide tool support for framework development. The agent incorporated into the tool sifts through a collection of rules and determines a sequence of activities that should be followed based upon what functionality the user wants to achieve. The functionality is pre-defined within the tool and associated with the rules. Some of the tasks are automated, while many more are simply directives to the user to implement a method or class. These tools tend to focus on one framework, and further work needs to be done on addressing the issues involved in integrating multiple frameworks, such as framework gap (where two frameworks need an intermediary to communicate) and especially overlap (where two frameworks attempt to control the same resource or perform the same function) [Mattsson and Bosch 97].

UML has been extended to incorporate some support for frameworks in the UML-F version [Fontoura et al. 00]. It attaches name-value pairs to existing UML constructs to identify the hot spots within a framework. These tags can be used to distinguish between framework and application classes, as well as defining what user defined type of variation point exists. However, there is currently little support beyond labeling.

While not specifically targeted at frameworks, the Programmer's Apprentice [Rich and Waters 90] uses similar concepts. The tool is meant to be an intelligent assistant for the requirements analysis, design and implementation phases of development. It uses a clichés, which are common programming constructs such as a hash algorithm, to bring programming to a higher level of discourse. Much like the design patterns, specialization patterns and rules above, clichés define elements that need to be completed by the user. The apprentice also incorporates constraints to ensure that the clichés used are consistent with one another.

2.5.4 Documentation

Properly documenting a framework is important in order to ease its understanding and use. By reducing the time needed to learn the framework, applications can be developed more quickly, or more time can be spent on other issues, such as quality. The documentation specifically proposed for frameworks focuses on easing the understanding of frameworks, by showing the interactions between classes in the framework or by showing how the framework is intended to be used. Johnson [Johnson 92] 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.

While the design of the framework can be documented in existing diagramming techniques and the purpose can be documented with traditional narrative descriptions, hooks are intended to focus on the use of the framework which is not captured in the other two descriptions.

2.6 Framework Documentation

Below, we describe several of the proposed types of documentation that have been specifically designed to represent frameworks, or have been applied to frameworks.

2.6.1 Multiple Views

Campbell and Islam [Campbell and Islam 92] propose a six part approach to documenting the design. Their approach is meant to support the description of the abstract properties of interacting components through specification of the relationships between the various components of a framework. They recommend the use of class hierarchies, interface protocols, entity-relationship diagrams, control flow diagrams, synchronization path statements and configuration diagrams to describe the structure and behavior of the framework. Together, the diagrams show several aspects of the design. Class hierarchies show all of the abstract classes and their concrete subclasses. Interface protocols list, for each class, all of the public methods, their argument types and return types. Synchronization path statements list the order in which the methods of a class should be invoked. Control flow diagrams display the run-time behavior of the system. Entity-relationship diagrams show the quantitative relationships between instances of classes in the system. Finally, configuration diagrams show which concrete classes can be used together in the same application and which conflict with one another. In addition, Campbell and Islam define a means through which control flow, synchronization and entity-relationship diagrams can be extended to guarantee that any application uses the structure and behavior defined. The different views combine to define the relationships between classes in the framework, but they do not provide any advice for using the framework.

2.6.2 Design Patterns

Design patterns are a very popular means of describing frameworks. A design pattern names, abstracts and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design [Gamma et al. 95]. Design patterns are proven solutions to common design problems that can enhance the flexibility and reusability of a framework. Beck and Johnson have used design patterns to help show how the architecture of the HotDraw framework is derived [Beck and Johnson 94]. The derivation states a sequence of problems and the design patterns used to solve them to build up the design of the framework. Through this derivation, a developer can gain some understanding of why certain decisions were made about the design of the framework which Beck and Johnson claim as crucial for customizing the framework to a particular problem. The OSEFA framework has also been described using design patterns [Schmid 95].

Using commonly known design patterns can also help developers understand the framework by serving as a common vocabulary between the framework builder and the application developer. The design patterns serve as a means of abstraction, allowing a developer to understand groups of classes and their interactions. The application developer can see that a design pattern was used and immediately understand some of the advantages and limitations of the design.

Further work has been done to allow the interactive visualization of design patterns within a framework [Lange and Nakamura 95]. The visualization is done through a tool called Program Explorer which can provide views of the structure and behavior, both at the object and class level, of a running C++ program. Being able to visualize the design patterns in a program links the abstract understanding of the architecture provided by design patterns to the concrete details of the actual framework. Unfortunately, Program Explorer does not automatically recognize design patterns in the code; the developers must recognize the patterns on their own through exploration of the code. As Bosch has pointed out, traceability is often lost when using design patterns as the pattern can be implemented in many different ways [Bosch 98] and thus can be hard to recognize.

Further work has been done on recognizing patterns and hot spots within applications and frameworks. The SPOOL environment [Schauer et al. 99] looks for characteristic incomplete methods that exist within a framework and must be filled in within an application (called template methods). Another tool uses pattern matching techniques to attempt to automatically detect whole design patterns [Campo et al. 97]. It searches for the characteristic class structures inherent in such design patterns as the Composite [Gamma et al. 95].

In some cases, code can be generated directly from the design patterns. [Budinsky et al. 96] have taken simple design patterns with characteristic structures (again using the Composite pattern) and present users with a dialog that allows them to fill in any needed parts of the pattern and choose from a number of means of implementing the pattern. [Florijn et al.] breaks patterns down into their component parts (called pattern fragments) such as class or method roles along with the constraints between roles. A pattern can then be invoked and all of the fragments are inserted into an application.

Pattern languages are becoming a popular means of guiding the generation of software and have been identified as somewhat useful in documenting frameworks as well [Brugali and Menga 99]. A pattern language is the set of patterns for a specific domain along with any structuring principles and can be considered a high-level language [Alexander 77]. Pattern languages exist for many different domains, including business resource management [Brega et al. 99] information management [Aarsten et al. 00] and for evolving [Roberts and Johnson 96b] and designing frameworks themselves [van Gurp and Bosch 01].

Design patterns can in a limited way, show how to use parts of the framework. Many design patterns are extensible, and so define how new classes can be used in the pattern. However, the description of design patterns is of necessity very general in order to apply to a wide range of situations. When used in a framework, a design pattern must be made concrete by filling in all of the details specific to the problem within the framework. The general design pattern description will not contain those specific details which can affect how the framework should be used. While they are an excellent means of helping to build and document the overall design of a framework, design patterns are not well-suited to documenting how a framework should be used.

2.6.3 Metapatterns

Metapatterns [Pree 95] 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 collection of variation points (and a hook description discusses how to use the hook). Hook methods in essence, are a single variation point.

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.

2.6.4 Exemplars

Exemplars [Gangopadhyay and Mitra 95] provide a different means of understanding frameworks. An exemplar consists of a concrete implementation provided for all of the abstract classes in the framework, and of their interactions. In justification of this approach, the authors state that frameworks often consist of a small core of abstract classes that define the major interactions within the framework, and then a larger number of more concrete classes can be derived from the abstract classes but still follow those interactions. They claim that learning a framework involves understanding the responsibilities of the abstract classes, the interaction between the classes, and the virtual methods defined by the abstract classes which are left to the subclasses to implement.

The framework builder provides the exemplar, which is separate from the actual framework. The interactions between classes can be explored through the exemplar using a special tool which shows the structure of the classes and message passing between the classes and allows the developer to follow the sequence visually. Using the tool, developers can gain an understanding of the core architecture of the framework and how its parts interact. The developer then identifies the classes to be used or modified for the application. Class hierarchies can then be browsed by the tool to find a class which fits the application or one that can be modified. While this approach allows for the understanding of the design of a framework, it is not as well suited for providing the purpose of the framework, or how the framework is meant to be used.

2.6.5 Examples

While exemplars show the core abstract classes of a framework, many frameworks are simply documented with examples showing various aspects of use, which may or may not include these core classes. Examples are often complete programs that use the framework or small fragments of code that show a particular use of the framework. A study has indicated that examples are a better means of learning a framework than exploring the class hierarchy of the framework [Shull et al. 00]. However, they admit that while examples are tremendously useful, when a task deviates significantly from the examples, an example based learning approach fails.

2.6.6 Cookbooks and Motifs

The cookbook in [Krasner and Pope 88] consists of a general description of the purpose of the Smalltalk Model-View-Controller (MVC) framework. MVC is used as the basis for many user-interfaces and an implementation is provided in Smalltalk. The cookbook describes the major components of the framework and their roles, and follows with a number of examples to illustrate how the components can be used. It is presented as a tutorial to learn the framework. The cookbook shows the purpose of the framework and does provide general advice about how to use it. However, the entire cookbook is intended to be read as a unit and there is little structure to the information. There is no indexing of problems and their solutions, so finding a particular solution requires scanning through the entire text.

A different type of cookbook found in [ParcPlace 95] does provide an indexed set of solutions, each focusing on specific issues such as how to create an active view in MVC. Each entry in this cookbook defines a problem to solve and then gives a prescriptive set of steps to follow for solving the problem. The approach is flexible enough to be applied to anything from using a visual interface builder to showing how to add elements to lists. However, the steps are narrative descriptions and are not well-structured or uniform; anything can be written as a step and in any level of detail, from a line of code to a description of some aspect of a particular tool. There is no specific language or grammar in which to write the cookbook entries, which can lead to imprecise or ambiguous instructions. Finally, the cookbook entry sections do not include a place to specify the effects that using that entry may have on other parts of a program, such as requiring other changes.

Patterns, introduced both by Johnson [Johnson 92] and Beck [Beck 94], fall roughly into the same category as a cookbook, documenting the purpose and use of a framework as well as a little of the design. Each pattern describes a problem that application developers will face when using the framework. Beck's patterns then describe the conflicting constraints involved in the problem and how they can be resolved. Johnson's patterns give general narrative advice and examples about ways to solve the problem. Both then summarize the solution, and potentially refer to related patterns. The collection of patterns makes up a directed graph indicating the order in which to read them, starting with the main pattern which describes the purpose of the framework. Much like the use of design patterns to document the architecture of a framework, using a linked set of patterns to document the use of a framework allows a developer to gradually build up an understanding of the framework without being overwhelmed with details. However, like the cookbook approach, the problem discussions and solutions are narratives and not well-structured. The level of detail for the problems and solutions can vary widely between patterns. Much like cookbooks, they can also be imprecise.

Lajoie and Keller [Lajoie and Keller 94] combine the idea of design patterns with Johnson's patterns, which they rename motifs, to provide a more complete description of a framework. In their strategy, motifs point to design patterns, contracts [Helm 90] 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. Motifs give advice and examples on how to design solutions to problems using the framework and help to show the purpose of the framework, but they provide no more structure than patterns.

2.6.7 APIs and Interface Description Languages

In a different approach to that of cookbooks or patterns, Taligent describes two interfaces through which developers can use a framework [Taligent 95]. The client API (Application Programming Interface) defines how to use the framework as it was meant to be used with no modifications to the fundamental behavior of the framework. The customization API provides for the modification of the fundamental behavior. Authors at Taligent claim that thinking about these two APIs helps to improve the design of the framework since together they define how application specific code interacts with the framework.

Two additional APIs are defined which are used by the client and customization APIs. The calling API defines all the functions provided to applications by the framework. The subclassing API defines the functions that the framework calls and which the developer can override through inheritance to provide application specific functionality.

Unfortunately, the concept of the client and customization APIs has not been developed further. The calling and subclassing APIs cannot completely describe the client and customization APIs. Using a framework involves more than calling or overriding functions. It can also involve choosing and configuring components within the framework, adding new components and even modifying existing components.

2.6.8 Reuse Contracts

Reuse contracts [Steyaert et al. 96] 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.

2.6.9 Behavioral Contracts

Behavioral contracts [Helm et al. 90], 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 consist 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.

2.6.10 Interface Modeling Languages

Languages such as Darwin [Magee et al. 94] or those used in Java Beans are intended to define interfaces between modules or components. While not directly associated with frameworks, the techniques can be applied to frameworks. Darwin in particular defines what services a component supplies and those that it requires, making dependencies on the environment explicit. It also incorporates a mechanism for defining composite components that can expose the interfaces of components inside of it. These languages however, do not target the intended use of the components.

2.7 Summary

Frameworks are a valuable type of software reuse. Their use is beginning to see major gains in time to market. However, one of the key difficulties with frameworks is learning how to use them. There are many different ways to convey how to use frameworks, including tutorials and tools. Documentation is one of the key and most widely used means of describing how to use a framework. However, many techniques, such as design patterns, were originally intended to document other aspects then the intended use of a framework and document use only partially. Others rely on narrative descriptions which are difficult to produce automated tools for, or generate test cases from.

Hooks, described in the next chapter, are intended to provide a template in which framework developers and document how they intend their frameworks to be used. They are categorized into different types to help developers understand the properties of the changes they are specifying for using a framework. Additionally, they are intended to form a basis for tool support.

Chapter 3 – Hooks

3.1 The Concept of Hooks

The need for hooks has arisen due to the complexity of frameworks and they way in which they are used. Unlike typical software applications, frameworks are built once, but reused many times by many different developers. On top of that, large frameworks such as Taligent's framework or MFC can take 5 months or more to learn before developers can effectively use the framework. Other complex, framework-like programs, such as SAP can take years to configure properly. Most work on frameworks focuses on their construction. While construction is important, as Kent Beck [Beck 94] noted that it makes sense to invest effort where the payoff will be greatest – that is in helping developers effectively use the framework. Our studies on framework use have confirmed this need. We've found that understanding the interactions between the framework and an application is critical to successfully using the framework. Unfortunately information about the interactions simply isn't captured adequately in traditional project documentation.

Currently, the usual means of learning about the interactions involves talking to the framework builders or someone with experience using the framework. Unfortunately, those people may not always be available, as is usually the case with commercial frameworks. Users then resort to experimentation and support groups. An example is the numerous user groups, news groups and message boards surrounding popular frameworks such as MFC.

Some means of describing the interactions is needed, and so we've developed hooks to fill that need. Hooks are the places in a framework than can be extended by application developers to quickly implement features. For example, when creating a drop down menu using a graphical user interface framework, the application developer does not have to specify the underlying graphical look of the menus, how they interact with windows or user input devices (typically a mouse or keyboard). Instead, the developer simply decides on the menu structure and writes the action routines that occur when a user selects a menu item. Hooks are what truly separate frameworks from applications.

Research has indicated that users, especially new users, are task-based in their learning and use [Erdem et al. 98]. They will find out what task they need to perform and then look for examples and other ways of making the framework perform that task. Hooks build on this notion along with that of design patterns, and even cook books. They identify a task and then show how that task can be accomplished using the framework. However, unlike framework patterns and cook books, hook descriptions are low level, focused and have a greater amount of precision. A hook will show exactly how to build a drop down menu, but won't discuss the benefits of alternatives such as pop-up menus or whether a menu is the best solution to a problem.

Hooks can be thought of as scaffolding used while building a program. They are a construction-time mechanism that describes how an application can interface to a framework, as in Figure 3.1, but don't appear in the final program itself.

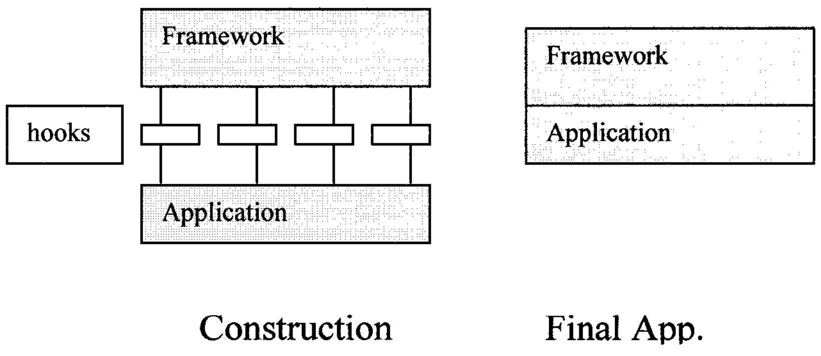

Figure 3.1 – Hooks as Scaffolding

However, that doesn't mean the information regarding the construction of the application should be thrown out once it is complete, if applications are ever truly complete.

The hook information can be useful during evolution, both of the application and the framework.

3.2 Goals

What properties should a description of the interactions between a framework and application have?

3.2.1 Define the intended interactions between a framework and application

The intended use of a framework is how the developers of the framework envisioned that it would be used. It encompasses all of the extensions that the framework was designed to easily accommodate, and so hooks should describe these. Often the interactions are described with an API, but this only captures calls from the application to the framework. Frameworks also follow the 'don't call us, we'll call you' principle. That is, the framework often calls parts of the application, and these types of interactions must be captured by the hooks as well.

3.2.2 Allow for better support

There is a huge potential for supporting the construction of applications from frameworks with additional tools and techniques. Many frameworks already come with simple support tools. Hooks should be structured in such a way as to complement the development of these tools. First, interactive tools can be used to help guide inexperienced users build (simple) applications from the framework. As we've discovered with our studies with framework users, providing some sort of a high-level process for describing how to get started with the framework and guiding them towards the hooks that fit their needs is just as important as describing the hooks themselves.

Second, tools can be used to automate all of the routine work involved in making an extension. If an extension involves creating a new subclass and adding a certain number of methods and variables, then a tool can easily build all of this infrastructure for the developer.

Third, once the interactions are well-defined, much better support can be provided for testing those interactions. Work can be done on providing a suite of test cases with a framework which can be used to test any application built with the framework.

Finally, knowledge of the interactions helps in evolution. One of the biggest problems in framework use occurs whenever the latest and greatest version of a framework is released. Application developers are then forced to discover any incompatibilities and fix their applications to conform to the new framework, or simply ignore any benefits of the new framework. By clearly defining the interactions within a hook, it should be possible to state exactly what changes are needed to conform to the new framework version, and possibly avoid the need for changes altogether.

3.2.3 Support characterisations of the interactions

Finally, we hope to capture certain characteristics of the interaction itself. Here we look at:

- Flexibility/restrictiveness: how much does freedom or support does the framework provide in supporting a task?
- Testability: how easy should it be to test an application extension developed using the hook?
- Automatability: as described above, how can the hook be automated?
- Evolution: how does evolution of the framework affect the hook?

3.3 Hooks in Detail

In order to achieve the above goals, hooks are described in a specific format made up of sections. The sections 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 (post-conditions). The format helps to organize the information, helps to prompt for the required information and makes the description more precise and uniform, which aids in the analysis of hooks and the provision of tool support for them.

3.3.1 Hook Descriptions

Each hook description consists of the following parts:

Name	A unique name, within the context of the framework, given to each hook.
Task	The problem the hook is intended to help solve.
Type	The amount of support provided for the problem within the framework. Types are described in detail later in this chapter.
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	These are both existing and new components that participate in the hook. Typically they are the classes within the framework and the application that are directly used in the hook or are required to understand the

context of the hook. Defining the participants is left to the developer and is for the reference of the framework user.

Parameters

The parameters consist of all of the variables within the hook that the user must complete in some way. The parameters can be anything from simple integers to methods to entire classes. A parameter may already hold a value when the hook is initiated; as is the case when one hook requires the use of another hook (see uses section).

Pre-Conditions

Constraints on the parameters (or the context) that must be true before the hook can be used. Note that these preconditions apply to the enactment of the hook itself. Any preconditions and invariants on the classes and methods should be defined within the specification for the framework and, of course, should also be followed. Hooks are not intended to replace good class design, but complement it. Preconditions can consist of:

- Elements that must exist within the application, such as subclasses of framework classes. For example, in order to send a message in the CSF, a structure called an outbox must have been previously defined so that the message can be placed in it.
- Constraints on parameters. These are construction-time constraints. For example, in the CSF when a connection-based router is selected on the server, the client must also use a connection-based router. Run-time conditions; however, should be a part of the framework itself, and not the hooks.
- Assertions about the state of the application. For example, in HotDraw a hook exists to remove the drawing tools (so the framework is used strictly for display). That hook changes the state of the application, so that hooks which rely on having the drawing tools can no longer be used.
- Notes to the framework user which cannot be represented

	through one of the above means.
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. All changes, including those involving the use of other hooks, are intended to be made in the order they are given within this section. The changes are written in a specific scripting language (defined in Appendix A) that can be parsed by a machine and enacted interactively with the user. Chapter 6 discusses the tool.
Post-Conditions	Any constraints on the parameters that must be true after the hook has been used (see preconditions).
Comments	Any additional description needed.
Related Hooks	Pointers to other hooks the user may want to use in conjunction with the current hook.

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.

As an example we'll use a hook taken from a small framework developed as a part of this research called the Client-Server Framework (CSF). The full description of the framework can be found at www.cs.ualberta.ca/~garry/framework. Briefly, CSF defines the mechanisms and protocols for communication between two programs (such as a client and server, or peers) across a network, so that the developer can use them transparently without having to deal with sockets or network errors. The communications framework provides for synchronous and asynchronous communication and connection-based or connectionless protocols, and is suitable for small scale student projects. It is a java program that handles all of the communication over a TCP/IP connection and has some persistent storage capabilities. The framework uses a model similar to email for communication. However, it is not an email program, does not use any email protocols and is not restricted to sending text messages.

Information sent between programs using the CSF is encapsulated within a subclass of the Data class. Here is the hook description:

Name	New Data
Task	Information to be packaged and sent across the network is encapsulated as Data.
Type	Open (Base Class)
Uses	None
Participants	Data (framework), NewData (app);

Parameters	NewData: name;
	myVar: set of variable;
	NewData.readData(DataInput in);
	NewData.writeData(DataOutput out);

Pre-Conditions	Data exists;
Changes	• New subclass NewData of Data; • New operation NewData.NewData(); • Repeat as necessary • { • new property NewData.myVar; • }; • NewData.readData(DataInput in) extends Data.readData(DataInput in) throws SendException; • NewData.writeData(DataOutput out) extends Data.writeData(DataOutput out) throws SendException;
Post-Conditions	Subclass NewData of Data;
Related Hooks	Send Data: to send the information in the new data class across the network.

As the task states, all information sent across the network through the framework is a subclass of Data. The type is open, which will be discussed later. The hook does not use other hooks. The participants section states that two classes are involved in the hook. The first is the Data class from the framework. The second is the NewData class, from the application. Note that because NewData is an application class, the actual name of the class must be supplied by the user. The parameters are all of the values that the user of the hook must supply in order to complete the hook.

In order to use this hook, the user would follow the changes section and perform the following steps.

- Supply the name of the application class, for example UserData, and then create a new subclass within the application. *New subclass NewData of Data;*
- Create a new constructor for the UserData class with no arguments, which is required by the framework. *New operation NewData.NewData();*
- Create any instance variables required by the class, for example user_id and password. *new property NewData.myVar;*
- Finally, extend the read and write methods to read and write the variables to a stream.

3.3.2 Why this particular format?

People have commented that the hook description is not very understandable when it is first read. The reason is that the hook description itself is written in the grammar mentioned above and detailed in Appendix A. The grammar describes a relatively simple scripting language that a tool can parse, and the semantics of the grammar are defined such that the tool can interactively work with the application developer to generate code or diagrams automatically. Hook descriptions can also be accompanied by textual descriptions, examples and diagrams to help users understand their purpose and how they work. It is the format, along with the type model, that forms a basis for the possibilities of tool support discussed in Chapter 6.

3.4 Types of Hooks

In the example above, New Data is described as an open hook. What does that mean? Hooks are divided into four basic types called levels of support: option, template, open and evolutionary. Levels of support are taken from the types of parameters within the hook and are defined by the completeness of the mechanisms that the framework provides to use the functionality within the hook.

3.4.1 Option

With option hooks the developer has to choose from a set of pre-built components without worrying about their internal workings. This is the black-box approach to frameworks described by Ralf Johnson [Johnson 1992]. Option hooks only contain option parameters.

Option parameters are complete. No user input is required other than selecting one option or another. They can be thought of as choosing an answer in a multiple choice example. The types of changes that can occur in option hooks are those that require the user to select from a set of options or connect pre-defined components together.

3.4.2 Template

If the framework has a pre-defined method of fulfilling a task, but the details of the method are application dependent and so have to be filled-in by the application developer, then the hook is a template hook. Support is provided for the task through parameterization. For example, the developer may need to follow a specific template when creating a new subclass, or follow a particular flow of control to accomplish a particular task.

Template hooks contain either option or template parameters. Template parameters are not complete like option parameters but have a given type defined within the framework or within the hook. They can be thought of as fill-in-the-blank exam questions. In the New Data Hook, the class name NewData is a template parameter. Template parameters are considered to be internal parameters. They can be said to be 'typed' by the framework. They are fully defined by the framework, or the hooks for the framework, and the user cannot introduce elements outside of that definition. In the case of NewData, the user cannot introduce a class name already in use by the framework, or produce a class that is not a subclass of the Data class within the framework. Template parameters can be simple strings or numeric values, or complex interpreted scripts. An example is a spreadsheet such as Microsoft Excel. The format of the formulas within each cell is defined by the program and not some external programming language that Excel does not understand.

3.4.3 Open

Open hooks provide guidelines and conditions for the developer when adding new elements to the framework. Open hooks occur when the developer needs to override or fill in methods or create custom code that cannot be scripted in a template. Creating the action methods for a drop down menu is an example of an open hook, since what the program does within the action methods is completely open to the developer. Developers are advised to restrict themselves to accessing only the participants within the framework, following the Law of Demeter [Lieberherr et al. 88], but tasks unforeseen by the framework builder may require otherwise.

Open hooks can contain option, template or open parameters. Open parameters can be thought of as short answer questions on an exam. In the New Data Hook, the two methods readData and writeData are open parameters. Both must be coded by the framework user based on the requirements of their particular application. Open parameters are considered to be external to the framework (as opposed to internal option and template parameters). That is, the framework has no control over the elements introduced through an open parameter. Changes at the open level usually involve specializing, overriding or introducing new methods to classes or introducing a completely new component that will be called by the framework. This is the case in graphical user interface builders where developers can introduce their own window creation routines.

3.4.4 Evolutionary

An evolutionary hook makes changes to the framework itself, either by changing parts of the code or breaking invariants defined on the framework. Evolutionary hooks can contain any of the 3 types of parameters defined, but they may also change fixed elements of the framework. There are no restrictions on evolutionary hooks and thus they are difficult to support. There are no evolutionary parameters. In fact, an evolutionary hook changes parameters in ways they were not originally intended to be changed, including modifying the framework itself. An evolutionary hook typically involves modify and replace change commands. An evolutionary hook in a GUI framework might change the mechanism by which menus are generated and displayed, overriding part of the framework.

3.5 Characterising the levels of support

The levels of support have an impact on verification, testing and automation of the hooks and the levels of support can be characterized in four ways.

3.5.1 Restrictions

First, restricting the scope of change restricts the testing needed for application extensions. It should also be noted that each level of support is an extension of the level inside of it as shown in Figure 3.2. An open hook can contain templates and option hook choices, but differs from option and pattern hooks in that it imposes fewer restrictions.

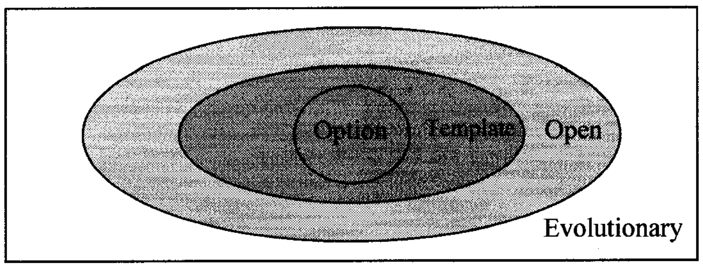

Figure 3.2: Hook Type Hierarchy

- Option – Complete. Developers are limited to a strict set of choices and cannot deviate from them. These choices should maintain the integrity of the framework by not violating any invariants or altering the architecture of the framework.
- Template – Defined format. Developers supply a set of parameters to the template which must fall within certain bounds defined by the framework.
- Open – Constrained, but no internally defined format. Developers may add custom code to the application, but not alter the framework itself or violate pre- and post conditions placed on the hook.
- Evolutionary – Unconstrained. developers may alter the framework and violate invariants if necessary.

3.5.2 Automation

The level of support also affects the ability to automate the hook (provide tool support for it).

- Option – once the user has chosen an option or set of options, the installation of those options can be completely automated.
- Template – again, once the user has supplied the parameters to the template,

the execution of the template can be automated.

- Open – less automation can be performed here. A tool can generate skeletal code and diagrams that the user must fill in.
- Evolutionary – much like open, some skeletal code can be generated, but no more than that.

3.5.3 Testing and Verification

- Option – since the options are already present within the framework, the options should have already been tested by the framework builders before the framework was released, including interactions between options.
- Template – test cases can be generated to cover the range of parameters that the developers can supply and then adapted to test the applications. The templates should not be able to violate invariants on the framework, so no verification is needed.
- Open – automated testing breaks down here, and verification is needed, perhaps using model checking, to ensure that the developers do not violate conditions placed upon the framework or try to circumvent or violate the architecture of the framework.
- Evolutionary – since there are no restrictions imposed upon the framework, nothing can be done for testing and verification. It must be handled on a case by case basis.

3.5.4 Evolution

The following discussion refers to the effects on applications due to framework changes.

- Option – since all options are contained within the framework, modifying the options will not require any changes to the application as long as the options maintain their functionality.

- Template – changes to the framework will be transparent to the application developer if they do not change the templates. If they do, then migration plans can be easily created and even automated.
- Open – so long as the application developers stayed within the restrictions of the hook, then unless the hook itself changes, no changes should be needed to the application. However, there may still be unforeseen interactions with the code they write.
- Evolutionary – changes to the framework will likely conflict with changes made by the application developers.

3.6 Summary

Hook descriptions document how the framework developer intended the framework to be used. They provide a template to describe what framework users should do and where they should go within the framework to accomplish a specific task. New users of the framework will approach the use of the framework looking to accomplish specific tasks, and so hooks directly address their needs. Expert users of the framework may also want to be reminded of all of the specific details of what needs to be done to accomplish a task, and hook descriptions contain that as well. The descriptions are written in a small scripting language that is intended to be used with a tool in order to automate the use of hooks and the framework and to guide framework users through a process of using the framework. Finally, work has been done on categorizing the types of changes that can be done with a framework and their impact. An option hook may be easy and quick to use, but will be much more limited in what it allows the user to accomplish than an open hook. On the other hand, option hooks are easy to provide test cases for and can be automated easily.

The next chapter provides many more examples of hooks and describes the various patterns of hook that exist at each of the levels of support.

Chapter 4 – Hook Design

One of the tenets of extreme programming [Beck 95] is that developers working on a single application should not worry about designing for extensibility or generality because they cannot predict what sort of extensibility will be needed for such applications. However, this is exactly what is needed when designing framework. The implication is that designing reusable software is a different process than designing applications, one in which the types of extensibility needed should be known and planned for before implementation even takes place.

Design techniques for developing frameworks are starting to appear as in [van Gurp et al. 01] and [Demeye et al. 97]. Other approaches advocate developing families of frameworks by generalizing from specific to abstract problems [Schmid 97], or developing product lines [Weiss and Lai 99] [Pasetti and Pree 00]. Pree [Pree 99], for example, advocates a development process in which the hot spots are identified early in analysis and their design is considered throughout the rest of development. Hook design fits well into this process. They are the interface between a framework and application extensions made to the framework and designing and documenting them up front is a worthwhile activity. It is more difficult to go back and document hooks after the fact, or to try to rediscover them within a framework to try to figure out how the builders of the framework intended the framework be used.

Unfortunately there is little information on how to design the hooks. What little there is often doesn't discuss the tradeoffs of one approach over another. This chapter discusses some aspects of designing hooks and introduces hook patterns – commonly used methods of designing the hot spots – and identifies some of the strengths and weaknesses of each pattern.

4.1 Applying Hooks

The discussion on hook design is taken from experience with applying hooks in three basic ways. First, existing frameworks, such as Hotdraw (discussed in Chapter 2) and Java Swing were looked at and the major hooks documented to see if the technique was flexible enough to cover the task people perform with the framework. Second, a new framework was built, incorporating hooks throughout development and then used in a study detailed in Chapter 5. Finally, the hooks concept was applied to two other frameworks working with the developers of those frameworks. The two frameworks, Sandwich [Liew 99] and EAF [Hoover et al. 00] are discussed briefly below.

Sandwich is a person web-assistant framework developed by Wendy Liew. Sandwich is intended to serve as a client-side proxy to a web browser and to utilize what she calls agents to enhance the web browsing experience. Agents can, for example, keep a history of web sites visited, or capture partially completed forms so that the user of the agent can go back later and complete the form without losing any of their previous work.

The Size Engineering Application Framework (SEAF) project is focused in part on the re-engineering and re-development of the Size Master-Plus engineering software package for sizing and selecting pressure relief valves. The project involves the collaboration of Teledyne Fluid Systems - Farris Engineering in Edmonton and researchers in the Software Engineering Research Laboratory at the University of Alberta. To support current and future product development, two frameworks have been built. The first is the user interface and persistence framework which is responsible for coordinating the interaction between an application's user interface, the underlying database of information, and actual calculations embedded in the application. The second is the engineering framework which provides a worksheet model that guides the engineer through a complex calculation, reminding them of important steps, ensuring that key decisions and sub-calculations are made in the proper order, and recording the process for future review. It is the second framework that hooks have been applied to.

4.1.1 Finding Hooks

Typically, framework design begins with choosing a particular domain, such as graphical user interfaces or communications, and using domain engineering techniques, whether structured or informal. These techniques can range from a full-fledged domain analysis using FODA [Cohen et al. 92] or similar technologies to compile the result of experience in designing many similar applications in the past.

Commonalities, as the name implies, are things that are static or fixed across all applications within a domain. An example might be the concept of a window in most graphical user interfaces. Variabilities are just the opposite. They frequently change between applications. The amount of variability is tied to the flexibility of the hot spot; the more variable, the more flexible. In fact, the amount of variability corresponds to the types of support presented in Chapter 3. An example from the CSF is the MailServer set of classes that connect clients and servers together. The only type of variation allowed is the selection from a limited set of options (using an option hook) to select the type of connection desired (connected or connectionless). On the other end of the spectrum, Data subclasses can be extended through an open hook and new functionality added. Once the variabilities have been identified, then the hooks can be designed.

4.1.2 What should be hooked?

It was mentioned previously that hooks occur within the hot spots, or variation points, of a framework. Any part of a service or process that occurs within the variable part of a framework is a potential hook. Any services within the variable part might be extended or modified. Any processes including the hot spots might also be extended or modified. For example, when sending a message in the CSF (see Figure 4.1), the message is sent from a CommAwareObject via an Outbox through the MailServer to another MailServer to an Inbox and finally to the destination CommAwareObject. Of these, the CommAwareObject and the Data being sent are meant to be extended (through subclasses) by the framework user, while the MailServer, Inbox and Outbox are not.

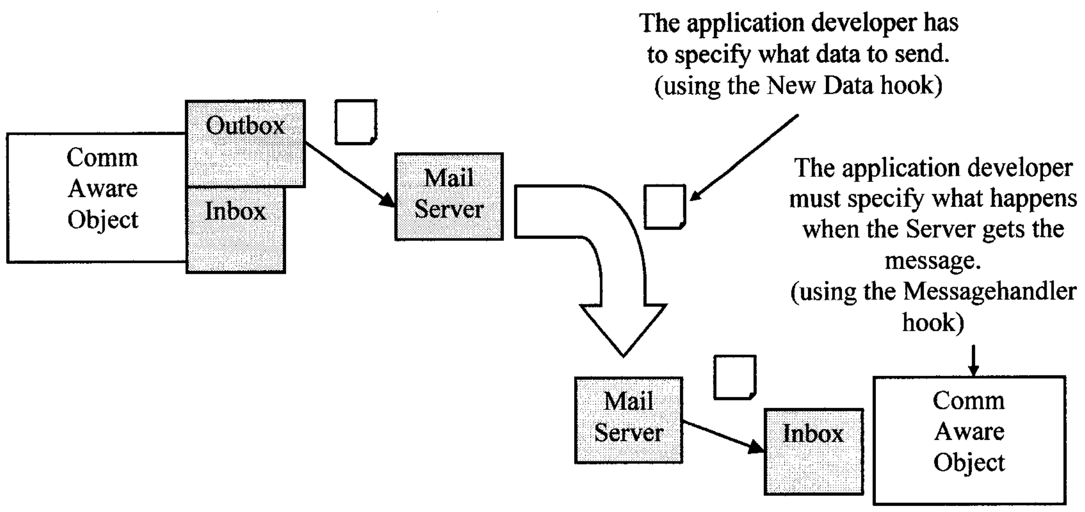
The application developer has to specify what data to send. (using the New Data hook)
Comm Aware Object
Outbox
Inbox
Mail Server
The application developer must specify what happens when the Server gets the message. (using the Messagehandler hook)
Mail Server
Inbox
Comm Aware Object

Figure 4.1: Sending a Message using the CSF

More specifically, there are four key concepts to keep in mind when designing hooks at hot spots:

- **Access to Key Events:** Key events in the framework, or key steps in standard framework processes should be accessible to application developers to allow an application to respond to them. For example, applications will differ in the way they handle error conditions and should be allowed the opportunity to process them. Or in the CSF, a key event is the arrival of a message, which must be handled within CommAwareObject.

- **Access to Services:** Obviously access should be provided to the services of the framework. For example, it does no good to provide a communications framework without the ability to send messages (to communicate in some way).

- **Extension of Services:** Services provided by the framework should be extendible by the application. The extension is typically tied closely to having access to key events. When those events occur, the framework should allow the application to perform some additional processing or to change the workflow of the service altogether. In the CSF, the communication protocol does not include encryption, but can be extended to include it by defining a plug-in to the MailServer.

- **Replacement of Actors and Services:** Key actors and services should be replaceable by application modules. A typical example is allowing users to write their own sorting routines as they know the characteristics of their data better than the framework designer ever would. Allowing for replacement is a factor in managing evolution in the framework. Without the capability, application developers will be forces to modify the framework in non-standard ways and thus be incompatible with future versions of the framework.

However, while there are many potential hooks, not all will necessarily be included for a particular framework.

4.1.3 What should not be hooked

When developing frameworks it might seem like it is desirable to make everything open and extensible by the application developer, and it is. However, there are other forces acting on framework development. First, there are the related issues of scalability and maintainability. Our work thus far has been on somewhat smaller frameworks, and we cannot make claims about scalability. However, describing all of the hooks for a large framework will likely be a time consuming task, and if the recommendations of the previous section are followed to the letter, it may become a monumental task. Further, all of those documented hooks have to then be maintained and updated whenever the framework itself is updated. As with any sort of documentation, documenting hooks should follow the goal of achieving the most impact. *The most commonly used hooks (that the framework builders anticipate) should be described.* Other hooks may or may not be described based on the amount of resources available (time and money) and how often it is anticipated that they would be used.

Second, making actors and services extensible has the consequence of greatly reducing the size and number of frozen spots within the framework. Why is this a problem? It gives the framework developers much less room for change in the future. Everything that is extensible becomes part of the framework/application interface and should be kept fixed if at all possible. In the CSF, the MailServer is kept frozen to outside developers, even though it is a key actor and should be replaceable by the criteria above. This is so that it can be modified and optimized by the framework developer. Keeping it frozen made it possible to modify it even while other groups were developing applications on top of the framework and to substitute in new versions of the framework without breaking any of their code.

4.1.4 What Makes a Good Hook?

There are currently no metrics for measuring the value of a hook, but there are several properties that should be adhered to.

Exposes variablities but not commonalities

This seems like an obvious statement, but it is easy to put commonalities within a hook. If the hook contains change statements that have no parameters, then the hook likely needs to be reworked.

Simplicity

Simplicity refers both to the size of the hook and the concepts contained with the hook. Complex hooks are obviously more difficult to understand. They also tend to reflect unnecessary complexity within the underlying framework.

Robustness

Hooks should be immune to changes within the frozen spots of a framework and resistant to changes within the code of the hot spots. The hook descriptions form an interface, and like a good interface, should not overly depend on the underlying framework implementation. It is achieved by isolating variable parts of the framework from frozen ones. For example, in the Prothos framework (a framework for developing business applications through HTML) objects can be made persistent simply by inheriting from a base persistence class. How the persistence is achieved is left to the internal implementation of the framework and can be modified without adversely affecting applications built on the framework.

Modularity

From experience in documenting hooks in Sandwich the goal of modularity of hooks was identified (called atomicity in [Liew 99]). Modularity in this case means that a hook should be applicable by itself, or should clearly indicate which other hooks it uses (in the uses portion of the hook description) and should not duplicate the actions of other hooks. For example, there is a standard means of creating new agents in Sandwich, and all agents follow it. However, hooks for specialized types of agents were also written (agents which manipulate HTML forms – to automatically fill them in for example). Instead of writing multiple hooks to cover the creation of each type of agent, a base hook should be written, and other more specialized hooks should then use that hook.

4.1.5 What Type of Support should be used?

If it has been decided that some variability exists within a part of a subframework, then the framework builder must decide what type, or types, of hooks to provide. At a high level, the choice is linked to the type of variation. The builder should use:

Option – when there is a limited set of variations, or when parts of the framework can be used or not used separately.

Template – when providing a service in which the parameters to the service vary, but the service itself does not. This is the common API style. Templates can also be used when the framework needs to be in control of the interaction (as opposed to open or evolutionary hooks where the application is often in control).

Open – when more flexibility is needed, when it is impossible to predict all of the variations that are possible for a given service.

Evolutionary – when the scope of variabilities and commonalities are not known, or when developers need to go beyond the capabilities of the framework and application developers may need to substitute their own components. The need for evolutionary hooks *may* also indicate that the domain that the framework targets has changed.

4.1.6 Using More than One Type of Hook

One hook cannot be all things to all people. That is, if a hook is simple and very easy to use, then it is not likely to be very flexible. So is a framework doomed to be one or the other (i.e. simple and inflexible or sophisticated and flexible)? No. In fact the simple solution is to provide multiple hooks for a single problem or requirement.

For example, in HotDraw one of the key features is the ability to use various commands encapsulated in tools to manipulate graphical figures on the screen. There is a wide variety of tools supplied with the framework that cover many of the common activities, such as moving figures around or resizing them. If that's all that were provided, then the framework would be easy to use – simply select the tools needed and quickly produce a new application. However, it would lack any sort of reasonable flexibility. Fortunately, HotDraw also provides hooks for producing new graphical manipulation tools. There are templates for building parameterized tools, and also open hooks for producing radically different tools.

4.1.7 Extending the Hook Description Language

Another result from working with frameworks such as Sandwich, and the GUI builders that come with frameworks such as MFC, is the lack of mechanisms within hooks for dealing with tools or additional files. Hooks operate on the framework, typically the code and design representation of the framework. They are not intended to tell the user how to make use of framework tools. That is, there are no commands within hooks that tell the framework user to move the mouse or click a particular button. In fact, translating hooks to some other form of representation is the responsibility of the hooks tool (described in Chapter 6). However, this extends to the gray area of configuration files as well. In Sandwich, part of the configuration of the framework occurs in text files external to the framework. While the hook changes can represent the configuration variables, it cannot tell the user that those variables reside within the configuration file. Additional comments were required for that. Complicating matters further, while text comments are fine for a user, a tool could not possibly know automatically where the variables were supposed to be.

Custom hook commands can be added, currently through the use of the Note change. Anything after a Note can be interpreted by additions to the tool. For example, to add a configuration variable to a text file in Sandwich the statement might be:

```
        Note setconfigvariable AgentName : string in agentinit.txt
```

The tool or user can look at this directive and know to include the AgentName in an external text file rather than trying to code it into the application itself. In this way, the hook descriptions can be extended if needed to include commands specific to individual frameworks or provide direction beyond the scope of the current language.

4.1.8 Using Hooks as a Walthrough of the Intended Use

We have found that applying the hooks model to a framework has several benefits. The structured description provided by hooks can help clarify understanding and prevent incorrect use of the framework by application developers. In the review of the EAF framework, a particular misconception about the framework was cleared up, and in our study using the CSF (see Chapter 5), we found that more significant errors do occur when the hooks were unavailable.

Writing the hooks forces the framework builder to state precisely how some part of the framework should be used. By forcing a walkthough of the intended use of the framework, defining hooks can help to expose deficiencies within the framework.
Writing hooks also exposes unnecessarily complex structure within the framework and forces the framework builders to either justify the complexity or to rethink it. While the proper hooks for using the framework may exist, those hooks may be complex or difficult to use. Often a complex hook or set of hooks is a reflection of needlessly complex structure within the framework. In an earlier version of the EAF framework, deriving a new type of data element then involved inheriting from three different classes and creating a complex pattern of interactions between the three. After attempting to describe the hooks for it, the structure was streamlined to the current and easier to use version. Ease of use is one of the desirable features of frameworks and describing hooks

The remainder of this chapter goes into more detail on the type of hook to use by identifying a number of common patterns along with their strengths and weaknesses.

4.8 Hook Patterns

Each general type of support can be broken down into common patterns. These are much like, and are influenced by, the work on design patterns [Gamma et al. 95]. Each takes a common problem and identifies a typical solution to that problem, along with the benefits and drawbacks of that solution. These patterns can be used to decide exactly what type of hook is needed for a particular hot spot. The following is a catalog of some of the patterns that I've observed in building and maintaining my own framework and analyzing several other frameworks. They are by no means a complete set.

4.8.1 Option Hook Patterns

4.8.1.1 Switches

Problem

> Functionality within the framework occurs in most applications in the domain, but not all. The functionality may also be tightly related to other functionality (that is within the same component as other functionality).

Solution

> Build a switch (such as a parameter) that allows the given functionality to be turned on or off as needed. A more complex switch might choose between several pre-defined options.

Example

> Switches are commonly used in GUI editors to set the properties of a particular element of a user interface. They are typically represented as Boolean variables.

Advantages

Switches are the simplest hook for developers to use since it generally requires the setting of a single parameter.

Concerns

Switches don't provide a clean separation between different functionalities. Even if the switch is turned off, it doesn't remove the functionality from the code (it simply does not use it) so bloated or slow programs may result.

Related Patterns

Choose Components offers the same type of support, but separates functionalities into separate components that can be added or removed.

Controlled Modification needs to be used when the switch is not actually built into the framework, but must be added after the framework has been developed.

4.8.1.2 Pre-defined Components

Problem

There is a well understood and finite set (or finite subset) of variations that exist across applications.

Solution

Create a set of related components that the user can pick and choose from and encapsulate one variation within each component. Some components may require the use of other components, or preclude the use of other components.

If the variations are well understood, then it makes sense to include them within the framework itself, as the role of the framework is to reduce the load on the application developers. Components included with the framework can be tested within the framework and do not require further testing by developers. They can be evolved within the framework without adversely affecting applications (if they maintain their interfaces).

Example

In Hotdraw a pre-defined set of drawing tools exists and these can be chosen from and incorporated into an application. The tools are the components to select among and the Select Existing Tools hook prompts the user to select the tools and then incorporates them into the application.

Name	Select Existing Tools
Requirement	The application needs a particular tool or set of tools that is already provided as a part of HotDraw.
Type	Option
Uses	Incorporate Tools
Participants	Tool (fw),

Parameters	ChosenTools : set of class Tool;
Pre-Conditions	ToolsEnabled

Changes	Repeat as necessary {
	  Choose t from ExistingTools;
	  ChosenTools add t;
	}
	Incorporate Tools [ChosenTools];

Post-Conditions	If BringToFront in ChosenTools then SendToBack in ChosenTools;
Comments	The tools BringToFront and SentToBack are a pair and should both be included if their functionality is needed. Incorporate Tools is used to integrate the tools into the application.
Related Hooks	Create Tool: to make an entirely new tool Compose Tool: to put together a new tool with existing components.

Advantages

Using components maintains a clean separation between the alternatives so that they can be used or not used as needed and no residual code gets left within the application.

Pre-defined components also carry the advantage of being part of the framework and thus can be fully tested within the context of the framework. They can be changed by framework builders without breaking the applications built using them if the application conforms to the hooks.

Concerns

Developers may come up with variations that are not covered by the set of components, so an open hook should also be provided to allow them to develop their own components.

Related Hook Patterns

Base Property hooks can be used to provide the ability for application developers to produce their own components. API hooks might also be used to ensure that the developed component interacts properly with the rest of the framework.

4.8.2 Template Hook Patterns

4.8.2.1 Interface

Problem

A process, sometimes called workflow, is common to many or all applications within the domain, but the parameters to the steps within the process are variable. The time at which the process is invoked may also be left up to the application.

Solution

Define the process as a sequence of method calls to the framework. This type of hook is closely related to contracts [Helm et al 90]. The hook indicates the constraints on the collaboration (the order in which things must be invoked, constraints on data, etc.) and identifies a service in the framework that applications can use.

The simplest collaborations involve only a single method call to the framework, with the rest of the sequence is completely encapsulated within the framework.

Example

Sending a message in the CSF is a classic, simple template hook. It consists of collecting the required parameters (the To address, the From address, the type of message and the data itself) and then indicates that the *send* method should be invoked with those parameters.

Name	Send Message, Asynchronous
Task	One program needs to send information or a request to another program on the network.
Type	Template, Interface
Uses	Incorporate Tools
Participants	NewCAO (app), Outbox (framework), Inbox (framework), CommAwareObject (framework);

Parameters	NewCAO : name;
	MessageType : string;
	toAddress, data : name;

Pre-Conditions	Subclass NewCAO of CommAwareObject;
	Operation NewCAO.send();
	Instance OutBox NewCAO.out;
	Instance Inbox NewCAO.in;

Changes	Acquire MessageType : string;
	Acquire toAddress : name;
	Acquire data : name;
	Assign NewCAO.in.getAddress() to returnAddress in NewCAO.send();
	NewCAO.send() ->
	   NewCAO.sendMessage(out,toAddress,MessageType,data)
	   handles SendException;

Related Hooks	Send Message, Synchronous : to block the current process while it waits for an answer. Handle Message: to determine what actions to perform when a message is received.

Advantages

Processes can be encapsulated within the framework, requiring less work for application developers.

Concerns

The contract between the framework and the application has to be well-defined to help ensure that the parameters passed to the framework do not violate any of the framework rules. Additionally, this type of hook does not provide any flexibility in altering the process.

Related Hook Patterns

User Exits can be used when more flexibility is needed.

4.8.2.2 Interpreter

Problem

A high-degree of flexibility is needed in a service, but the framework builder wants to restrict the amount of damage a user can do.

Solution

Build a custom scripting language that the user can program to provide a large but highly restricted degree of functionality.

Example

Many programs use this, including Microsoft Excel which allows the user to develop formulas in each of its cells. Many game editors allow devout fans to script up new scenarios, and even tools such as Rational Rose allow user defined scripts to control functionality.

Advantages

While the interpreter allows for a lot of flexibility, it also does not require the user to learn a general programming language. It also restricts the user in the types of scripts they can write.

Concerns

Some users may find the scripting language to be overly restrictive and would like a full programming language.

Related Patterns

Base Properties can be used to provide more flexibility than a script.

4.8.2.3 Parameterized Service

Problem

A service or process within the framework has a known range of variation, but it is impractical or even impossible to build a component for each point within the range of variation.

Solution

Provide components that encompass the range of variation and then allow the application developer to construct new services or processes using those components. The developers use a 'building block' approach to constructing new services.

Example

In Hotdraw tools for manipulating drawings have been broken down to the point where a tool is made up of a table of components. Each table entry consists of a *reader* for interpreting mouse commands, a *command* for performing some action and a *figure* that the action will be performed on. By constructing these tables it is possible to put together a wide variety of tools. For example a tool to draw a circle would use the mouse_click reader, a creation command to produce the circle and the figure would be the drawing itself. Out of this, a circle tool would be produced that created a circle every time the user selected the circle tool and clicked on the canvas provided by the HotDraw framework.

Name	Compose Tool
Task	A new type of tool is needed that is not already provided by the framework.
Type	Template
Uses	Incorporate Tools, Choose Figure, Choose Reader, Choose Command, DefaultEvents;
Participants	Tool (fw), NewTool (app), Figure (fw), Reader (fw), Command (fw), Drawing;

Parameters	Icon, cursor, manipulatingCursor, toolName, Figure, Reader, Command : name;
	NewTool : name;

Pre-Conditions	ToolsEnabled

Changes	Acqure icon : name;
	Acquire cursor : name;
	Acquire manipulatingCursor : name;
	Acquire toolName : name;
	Repeat as necessary {
	    Figure = Choose Figure [ChosenFigure];
	    Reader = Choose Reader [ChosenReader];
	    Command = Choose Command [ChosenCommand];
	    EventTable add (Figure, Reader, Command); }
	Drawing.init() -> Tool.DefaultTools add (toolName, eventTable, icon, cursor,
	    manipulatingCursor);
	Incorporate Tools [ChosenTools = (toolName)];

Comments	When the tool is used, it finds the Figure it was used on and a Reader for the type of input received, then executes the associated Command. If there is no entry for the Figure, then an entry for that Figure's superclass will be used instead.
Related Hooks	Select Existing Tools: to choose from the tools that come with HotDraw.

Advantages

New services can be put together quickly and easily using the building blocks approach. The application developer can also extend the service easily if needed (although such extensions go beyond the scope of this type of hook).

Concerns

Defined extensions are not as easily tested as option hooks. It also may be difficult to provide enough building blocks to satisfy the various demands of different applications, forcing developers to produce their own.

Related Hook Patterns

Choose Components can be used if there is a very limited set of choices. Base Properties can be used if more flexibility is needed.

4.8.3 Open Hook Patterns

4.8.3.1 Base Properties

Problem

A service, such as persistence, is defined by the framework, but the framework developers cannot predict or want to leave it to the application developer to define the actual parameters to the service (such as the data to make persistent) or allow them to add their own functionality.

Solution

Create a base class within the framework that encapsulates the variable part of the service. Within the hook, define the methods and variables that must exist within the application extension of the service.

For example, in the CSF, the interactions between Data and the rest of the framework are predefined, but the actual values and methods must be implemented by the developers. In addition, they can add new methods to their subclasses as needed.

Example

In the CSF, all information to be sent across a network derives from the Data class. The Data class defines the base properties – in this case the required methods that the framework will call. (See New Data Hook in Chapter 3). In Prothos, persistence is defined as a basic property and new business classes can be built from the persistence class, while defining the attributes of the business class and any functionality it needs.

Advantages

Allows inter-class dependencies and contracts to be defined within the framework, and extended to applications. For example, you can define a window subclass with extra functions, but it will still have all of the base properties and methods needed for the operating system to talk to it.

It is easy for the framework builder to extend or modify the base properties independent of any application simply by evolving the base class that the application classes extend. So if the underlying persistence model in Prothos (a framework for developing standard forms-based business applications over the world wide web) requires extra variables to be inserted into all persistent classes, it can be done at the base class level without affecting the applications (unless the applications are required to supply the values for those variables). The base properties are part of the frozen spots of the framework and thus should not affect the hook itself.

Concerns

What base property hooks gain in flexibility, they lose in ease of use. This type of hook requires a good understanding of the framework in order to not break any of the invariants on the framework.

4.8.3.2 User Exit

Other names and variants

Template and hook

Callback

Problem

A process must be performed across all applications within the domain, but one or more individual steps (rather than just parameters – the step itself) within the process vary.

Solution

Encode the common sequence within a method in the framework, called the template method. The template method should call an abstract method within the framework, called the hook method, which is left for the application developer to implement. Pree details variations on writing the template and hook methods.

An obvious example of the user exit idea is the implementation of a menu item in many GUI frameworks. When a menu item is selected, the framework will invoke a method (action routine – the user exit) that the developer supplies.

Template methods imply that the framework is in control of the process. The framework calls the application and then waits for it to return control so that the framework can continue processing. Sometimes there are also pre and post conditions imposed upon the hook method to help ensure that the application does not break the operation of the framework

Callbacks are another common means of implementing the template and hook. With callbacks, a method or function is registered to handle an event. However, they are conceptually the same. When the framework encounters the event, it invokes the callback method then waits for control to return.

Example

When a message arrives in the CSF (the process), a MessageHandler is invoked. The application developer is then free to perform whatever actions are necessary in order to respond to the message. The hook method, or user exit, in this case is the HandleMessage method of the MessageHandler.

Name	Handle Message
Task	An object has received a message and now needs to take action in response to the message.
Type	Open, User Exit
Participants	NewCAO (app), MessageHandler (framework), NewMH (app), MessageType (app);
Parameters	NewMH, myMH, NewCAO.init : name;

Pre-Conditions	NewCAO subclass of CommAwareObject;
	NewCAO.init() exists;

Changes	New subclass NewMH of MessageHandler;
	NewMH.handleMessage(Message m, CommAwareObject coa) extends MessageHandler.(Message m, CommAwareObject coa);
	Create Object myMH as MessageHandler(NewCAO) in NewCAO.init();
	NewCAO.init() -> NewCAO.registerHandler(MessageType, NewMH);

Advantages

Like all open hooks, the main advantage of a user exit is the flexibility it provides. The framework simply hands control to the application to do whatever it needs to do. If callbacks are used instead, then the hookup can occur dynamically.

Concerns

The framework relies on the application to maintain a set of invariants, which may be defined within a contract. Also, the application does not get to specify when the user exit should be called, that is controlled by the framework.

Related Hook Patterns

Interfaces can be used when less flexibility is needed, or when the application should be in control of when a process is invoked.

4.8.3.3 Intercept

Problem

A service is common across most applications, but in some cases it may be desirable to change part of that service for a particular application without replacing the whole service.

Solution

Build a wrapper around the service, so that when a request is made to the service, the wrapper intercepts the request and either passes it on to the original service or reroutes it to the altered functionality.

Technically, there are typically three ways of doing this in object-oriented languages. First, inheritance provides a built-in means of performing intercepts. Simply create a subclass of the original class and override the method in question.

Second, an adapter (using the adapter pattern [Gamma et. al. 1995]) can be built around a class or set of classes. The adapter substitutes itself for all instances of the original service, and generally just passes most method calls on to the original service.

Finally, the original service can be cloned and the changes made directly to the cloned class.

Example

In the following example from Hotdraw, again we are focusing on the Tool class. In this case, the application may require that a Tool should be notified and take action whenever the user of the application unselects a figure. Normally notification is only sent when the user selects a figure.

Name	Tool Shifts Focus
Task	A new type of tool is needed that performs some operation when the selected figure has possibly changed.
Type	Open (Intercept)
Area	Tools
Uses	Incorporate Tools, Provide Tool Initialization;
Participants	Tool (fw), NewTool (ap);

Parameters	NewTool : name;
	toolName : string;

Preconditions	ToolsEnabled

Changes	Subclass NewTool of Tool;
	NewTool.mouseCommandFrom(figure) specializes
	Tool.mouseCommandFrom(figure);
	New operation NewTool.shiftFocus();
	Serialize (NewTool.shiftFocus(),
	Tool.mouseCommandFrom(figure)) in
	NewTool.mouseCommandFrom(figure);
	Acquire toolName : name;
	Provide Tool Initialization [NewTool, toolName];
	Incorporate Tools [ChosenTools = (NewTool, toolName)];

Related Hooks	Select Existing Tools: to use tools supplied with the framework. Compose Tool: to produce new types of tools out of existing components.

Advantages

When using an intercept, there is no need to replace an entire service. Intercepts allow for default services to be included within a framework, but also allow application developers to extend them.

Concerns

By intercepting and modifying a service, the application has to be careful not to break any conditions required by the framework. When using such a hook it is easy to break Liskov's Principle of Substitutability [Liskov 88]. In this case it means that the intercepted service must be useable within the framework everywhere that the original service was used. If that is not the case, then errors may occur.

Related Patterns

Replacement can be used if the entire service needs to be changed.

4.8.3.4 Replacement

Problem

A service is common across applications, but developers may want to develop their own service in order to take advantage of special optimizations available to the specific data set of their application, to remove (or impose) limitations on the service, or simply to provide an enriched set of functionality.

Solution

Simply allow the service to be replaced by a user developed service. Any conditions that the service must conform to should be stated.

Example

In the MFC framework, users can change the default window creation routines. Other frameworks allow memory allocation routines to be replaced.

Advantages

By allowing replacement, the framework developer makes the framework much more flexible and improves its ability to evolve. Eventually application developers will want to go beyond the services provided by the framework and replacement is often the only way to do it.

Concerns

Replacing a framework service requires that the framework be tested again in the context of the replacement. Framework invariants, preconditions and post conditions will also need to be reconfirmed. Replacement also requires a lot of effort and knowledge about the framework.

Related Hook Patterns

If the user wants to keep some aspect of the service without completely rewriting it, then an intercept should be provided instead.

4.8.4 Evolutionary Hook Patterns

4.8.4.1 Controlled Modification

Problem

The framework has a certain limitation in one of its services or processes. A workaround has been found, but that workaround is not included within the framework itself.

Solution

Incorporate the steps involved in the workaround into a hook. The workaround itself should be a piece of pre-defined design or code and not require a lot of extra effort on the part of application developers.

Example

In Hotdraw, if an application does not require graphical tools (for example when you simply want a graphical display), then code within the DrawingEditor class needs to be modified. This can be done by producing a subclass of DrawingEditor and changing and removing a few lines of code.

Name	Disable All Tools
Task	The application does not require the use of tools for any purpose.
Type	Option
Uses	None
Participants	DrawingEditor (fw),
	NewDrawingEditor (app);
Parameters	NewDrawingEditor : name;
Pre-Conditions	ToolsEnabled
Changes	Subclass NewDrawingEditor of DrawingEditor;
	NewDrawingEditor.open(n,withLabel) copies

	DrawingEditor.open(n,withLabel);
	Remove code 'aToolPalleteView := …;' in NewDrawingEditor.open(n,withLabel);
	Remove code 'container add: aToolPaletteView …;' in NewDrawingEditor.open(n,withLabel);
	Replace 'leftOffset: 4 + …;' with 'leftOffset: 0;' in NewDrawingEditor.open(n,withLabel) ;
Post-Conditions	ToolsEnabled = false
Comments	Using this hook disables all mouse based tools within HotDraw.

Advantages

A service within the framework may be still useful with certain minor modifications in which case it doesn't make sense to spend the effort to replace the entire service. A controlled modification reduces the effort needed and the modification can be thoroughly tested once and then used many times.

Concerns

Controlled modifications are essentially a work around or a 'hack' and therefore do not often follow good programming practices. Work arounds can compromise the architecture of the framework, and make the application incompatible with future versions of the framework. Additionally, controlled modifications break one of the tenets of good hook design – they place commonalities within hooks that really belong within the framework itself. The need for controlled modifications is a strong indication that the framework itself needs to evolve.

Related Hook Patterns

A Switch or a Parameterized Service can be built into the framework to remove the need for the modification.

4.8.4.2 Cloning

Problem

The set of variations for a service are not well known, or the application developer may need to change the service in unforeseen ways.

Solution

Implement a sample class which covers all of the basic functionality and copy it directly. Then let the user customize it in any way they see fit.

Example

Forms in the Prothos framework are created by cloning an existing general form class and then making any needed changes.

Advantages

Cloning is one of the most free-form of the type of hook patterns. Developers have access to all of the code for a particular class and are free to tweak, add or change the code as they need too. It offers a great deal of flexibility.

Concerns

Clones of example classes tend to be very brittle with regards to changes made to the framework. Often developers will then end up with a series of similar, but related applications based on different versions of the framework and updating all of them to the same code base may take a significant amount of rework.

Related Hook Patterns

Base Properties can be used instead to help alleviate the brittleness of cloning.

Choose Components can be used if there are a known set of commonly used components.

4.9 Conclusions

Several different frameworks have been examined, from the role of a new user, of working with the developers and of developing a new framework. This experience has been applied to describing how to design hooks when building frameworks. Hook design is an important part of framework design. The interface between the framework and the application is the part of the framework that will be most scrutinized and used by application developers so making it clear, easy to use and flexible is paramount. Framework developers need to identify the variation points within their framework and can then developer hook descriptions for the framework. Some of the things that should have hooks provided for them are access to key services and the ability to extend them. However, providing hooks also makes it more difficult to later modify the framework itself, so a balance needs to be struck between providing variation points and leaving other places frozen.

A number of commonly used patterns of hooks have also been identified. These are broken down by the type of change they produce (option, template, open or evolutionary). These patterns, much like design patterns, are intended to be a resource for developers and to help capture some of the experience built into existing frameworks.

Chapter 5 – Using Frameworks and Hooks

Framework development typically requires substantially greater investment of time and effort than a single application, so success of the framework can be more critical from a development cost perspective. Frameworks can fail for many reasons, such as being targeted at the wrong problem or being poorly marketed, but, according to Booch [Booch 95] the biggest reason that frameworks fail to be adopted is that they are difficult to understand. In commercial frameworks, and in many proprietary frameworks, an object-oriented framework is developed by one group of software engineers and used by a different group of developers. Often, it is used by not one, but many groups of developers, so it makes sense to invest the effort to make the framework easy to understand and use.

The question is, how can frameworks be made easy to understand and use? Adhering to principles such as completeness (covering as much of a domain as possible) and flexibility (allowing a great degree of change to the framework) are meant to increase the usability of frameworks. But these principles and others need to be built on a foundation of knowledge of how frameworks are actually used. In particular, this knowledge is needed to properly develop and recommend tools and documentation for aiding in the use and comprehension of frameworks.

In this chapter, an exploratory study is discussed that was conducted over a two-year period from 1998 to 2000 to help gain some of that knowledge. The study had multiple goals, but two are relevant to this work:

- Does the use of hooks improve the understandability and usability of a framework?
- How do people with little or no knowledge of a framework approach development using that framework? How can this development effort be supported?

5.1 Set up of the Study

In order to study how frameworks are used, approximately 90 students over three terms of a senior year software engineering course were divided up into teams of five or six students each. Note that the study was repeated three times. Each term involved new students – students did not continue across terms. All of the students were in their senior year, and the majority of them had also worked outside of the university on a sixteen-month industrial internship. Prior to the start of the study, two pilot projects were held in one term of the class to determine the suitability of the framework.

Each team had three months to design and implement a small client-server application of their choosing with the requirement that a small framework developed in-house for client server or peer-to-peer communication be used as part of the project (called CSF – Client-Server Framework).

The framework comes with several types of documentation in order to facilitate its use:

- Use cases give an overview of the use of the framework and points to individual hooks where developers have to provide their own classes or methods.
- Design documentation provides a high-level overview of the major classes of the framework and their relationships to one another. This includes both class diagrams and collaboration diagrams along with textual descriptions.
- Hook descriptions show how and where the framework can be enhanced in order to meet application specific requirements.
- Examples show some specific uses of the framework and provide running code that the developers can experiment with.
- Interface documentation and code show the methods, classes and actual code

of the framework. The source code of the framework was made available.

Additionally, the framework developer was available throughout the course of the study to answer questions that arose.

The teams were all given the same framework as outlined above. In order to assess the value of hooks, the hooks for the framework were divided into two sets of roughly equal complexity. One set was made available to the development teams, while the other was not.

Because the development teams were composed of students, some guidance was provided in the development process. In addition, several deliverables were required at predefined times in their projects.

Near the beginning of the course, the developer of the CSF gave two overview lecturers (totally approximately 2 ½ hours) on the design and use of the framework. A simple example using the CSF was also provided to give a concrete instance of the use of the framework.

A technical review was held in the third week of the course with the purpose of allowing the development teams to study the CSF and then ask questions of the CSF developer. Each of the students reviewed the framework documentation on their own and prepared a list of questions to ask. Each team had their own 30 minute session to ask their questions. Discussion was kept to a minimum by providing quick answers, or deferring the answers until later. The purpose of the review was to collect questions or identify difficulties in understanding the framework and to address them immediately if possible, or later if needed. From this experience a set of 'frequently asked questions' and answers were produced.

Each team had to produce an analysis document that gave an overview of what they were going to develop and an early indication of how they were using the framework in their products. Each team also produced a detailed design document. A design review was held, this time with the purpose of finding defects in the design documents. Specifically, we were looking for the correct use of the framework in their products. This review followed a process similar to the previous review. However, in this case, the teaching team took on the role of the reviewers and students became the authors.

After the final product was delivered, each team filled out a short survey to document their subjective experience of using the framework.

In addition, we monitored the progress of the teams through the following in three ways. First, meetings were held each week with each team to gauge their progress and address any concerns they might have. Second, each student was required to keep daily time logs of their activities during the course. Finally, the students' questions for the CSF developer that came in over the term, i.e., after the first review, were answered using FAQ and recorded for later study.

The following sections detail some of the results we found.

5.2 Evolution of the Documentation

As part of the post project survey, the participants were asked to rate the value of different parts of the documentation in terms of both relevance and clarity. Hooks were included as one of the parts. Results are given on a scale of 1 to 5 where 5 is of high value.

Table 5.1: Developer Ratings of the Documentation

Term	Use Cases	Design	Examples	Hooks	Code	Process
1	2.8	1.8	3.6	1.9	4.0	N/A
2	2.5	2.0	3.5	2.7	4.1	3.4
3	2.8	1.8	4.0	3.8	4.0	3.8

As can be seen from the first line of the table, hooks were not highly rated at all in the first term of the study. In fact, the overall documentation was not rated highly (with the exception of code and examples) in the first term.

We found the general pattern of use to be roughly uniform across the developers. Members started out investigating the high-level framework documentation (use cases) to gain an overall, architectural understanding of the framework, and to participate in the framework review. They then moved to exploring examples and to the code to learn more details of the interaction between applications and the framework. As the hook documentation evolved over the course of the study, the developers relied more upon it and the specific hook examples as opposed to the more general examples.

We then re-evaluted the documentation to determine what had gone wrong. The language the hooks were written in turned out to be difficult to understand, so in the second term diagrams were included to help show how the hooks related to the framework, and finally in the third term more detailed explanations of the hooks were added. The relatively high rating of the use cases (compared with hooks and design) initially also caught our attention. The use cases provided pieces of a high level process of how to use parts of the framework. It was decided to then provide a complete high-level process for using the framework that would also indicate which hooks to use at each step of the process.

Finally, the list of questions in the FAQ was developed over the first two terms. The FAQ provided another means of indexing the hooks and the knowledge about the framework. Each question in the FAQ referred to hooks related to the question if appropriate. In the third term, no new questions were added to the FAQ as the documentation became complete.

5.3.1 Developer Interaction Required

Entries in the FAQ came from both the initial framework review in the first two terms and from correspondence with the developer. Correspondence was done primarily through email, and dropped significantly across the terms. However, this result is slightly mitigated by the carry-over of knowledge between terms. For example, in the third term of the study we know that one group enlisted the help of someone who had used the framework in a previous term (employed a ready-made framework expert as opposed to developing their own). However, as part of the survey, we queried the developers about receiving outside help and found that only a small minority did. The table below shows the amount of correspondence required with the developer across the three terms.

Table 5.2: Amount of Correspondance with the Developer

Term	Correspondance (emails)
1	112
2	49
3	7

The numbers in the second and third terms dropped dramatically as we learned what was required within the documentation. It shows that the documentation was sufficient to enable the use of the framework.

5.3.2 Suggested Framework Documentation

From the study we determined that there are several facets to understanding frameworks.

- Architectural understanding – what is the general architecture of the framework, and what are the important structures?
- Process understanding – what general steps are required to build a framework

compliant application?

- Interface understanding – where are the variation points in the framework and how can they be used?
- Code understanding – when something breaks in the application or the framework, or more details are needed than are provided in the documentation, then developers will go to the code to try to understand it.

Hooks alone are at too low a level and do not cover enough aspects of the framework to provide sufficient documentation on there own. A wide range of documentation is required. Johnson [Johnson 1992] has proposed the types of documentation needed: purpose (the domain of the framework), intent (process and interface understanding) and design (architectural and code understanding). The case study supported this conjecture.

The types of documentation that can be recommended from the study are:

- An overview of the framework including what types of applications it supports and any limitations it has. The overview can be a simple text document.
- Architectural documents. The architecture can be based on any standard technique or language familiar to the developers (such as UML).
- An overall process of use discussed below.
- Hooks and Examples of use. Examples are always useful as they give concrete uses of the framework. Hooks describe the variabilities and relationships of the framework (as in you must create an observer class and a related subject class in the observer pattern). Where examples may not cover all of the important details, hooks will. Another way of looking at the relationship is that there are several examples for any given hook. In fact, hooks and examples should be included together. There are two types of examples needed to support framework use: examples that show the general structure of an application using the framework and specific examples of

using each hook.

- The code itself

5.3.3 Defining an Overall Process

It became clear during the first phase (term 1 of 3) of the study that new developers needed guidance when first approaching the framework. We defined a high level process consisting of a series of steps that guides the new framework user through the use of the framework. Figure 5 shows the guidance process presented for the Client-Server Framework (CSF). Each of the steps was written in natural language and discusses a key concept of the framework, much like framework patterns [Johnson92]. The steps link to the hooks themselves, which in turn link to the framework. These steps provide three benefits:

Mapping. The high level process guides the user in deciding how to map their tasks onto the tasks supported by the framework. For example, the first two steps, Discover Data and Identify Persistence, guide the framework user in deciding what they need to communicate across a network, and which information needs to be persistent (stored in files or in a database). They are then pointed to specific hooks in the framework, specifically New Data and Read/Write Data, which handle transmitting and storing information in the CSF.

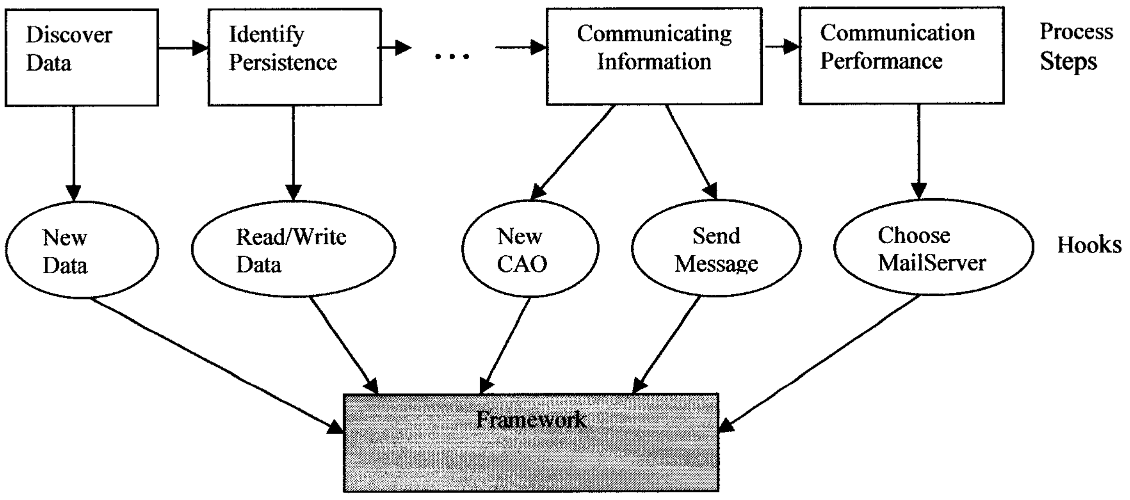

Figure 5.1: Overall Process for CSF

Order. The steps of the process explain the order in which things should be done to successfully use the framework, which isn't clear from a simple list of supported tasks. Even in the case of the CSF, we found that when new framework users were presented with the unordered set of hooks and the dependencies between them (e.g. the Send Message hook depends on the New Data hook), they needed some coaching in where to get started and where to proceed from there. The ordering shown in Figure 5 is somewhat simplistic, but appropriate for the CSF framework. However, in practice for larger and more complex frameworks, choices made in some steps may affect other steps, so more complex graphs would be required.

Details. Similarly to work done by Lajoie and Keller [Lajoie and Keller 94] in which high level descriptions called motifs are mapped to the design patterns of the framework, the process steps themselves are based on the hooks and point to specific hooks in order to make the changes. The hooks map into the framework itself, so the transition from requirements, to architecture to implementation can be made smoothly. For example, in the Communication Performance step, the framework user decides on the specific communication needs, such as performance, of the product. The Communication Performance step directs them to, in this case, the Choose MailServer hook, in which an 'engine' best matching the requirements is chosen. The hook itself points to the places in the framework that it affects.

Ideally, the steps will identify single tasks (or sets of similar tasks) which can then be mapped directly to a hook. The refinement performed potentially offers an additional benefit. Tasks mapped to the framework during the process steps allow for requirements tracability. That is, each requirement identified is mapped to a single hook, which then points to a specific part of the framework that fulfills that requirement. Additionally, support for this type of process can be incorporated into a tool, not just for a single framework as in the case of GUI builders, but for frameworks in general. We have constructed the prototype of such a tool called HookMaster for supporting the enaction of hooks on frameworks described in Chapter 6.

5.4 Mistakes People Make

To help determine the usefulness of the documentation, we collected the total number of errors made in using the framework (e.g. using the framework interface). To measure them, we looked at both the errors present in the final product and the errors made during development (based on analysis and design documents provided by the teams).

As can be seen in the table, errors made decreased across terms from an average of over 2 significant errors to under 1 significant error. The types of errors we considered are detailed in the next section.

Table 5.3: Average Framework Errors

Term	Fewest Errors	Most Errors	Average Errors
1	1	5	2.67
2	1	3	1.83
3	1	2	1.25

What also can be gained from the table is the disparity in the number of errors from the better groups to the worst groups closed. In term one there was a wide range, where experience plays a much larger role in understanding the framework, to term three where the framework was well enough understood through better documentation to make many fewer errors.

By learning what types of mistakes people make we are hoping to be able to determine how to detect and then help correct the different types of errors. Also, if the errors are known, then the documentation can be geared toward providing the information needed to avoid those errors in the first place.

There were five general classes of errors as described below.

5.4.1 Hook Errors

The hooks outline the ways in which applications can extend or interact with the framework. In some cases, people make mistakes when following them (or don't follow them). Usually, the mistakes are simple, such as not implementing a method required by a hook or setting a parameter to an incorrect value. Generally a small amount of debugging or inspection can find the problem, and by referring to the hook, it can be fixed easily.

For example, the New Data hook requires the developer to implement three methods - a constructor, one for writing its data to a stream (so the framework can send it over the network) and one for reading it back. When the read or write isn't implemented, data appears to be lost when the application is executed.

When sending a message using the synchronous communication mechanism (although this hook was not supplied to the developers), the developer must create a framework object to handle the synchronous send, create the message to send and set the timeout value (how long the sender should wait for a reply). When the timeout isn't set or is set too low, then only timeouts and never replies are received and it can look like the framework is not operating correctly.

5.4.2 Modifying or Extending Frozen Spots

Frozen spots are the commonalities in frameworks - that is, the places that are meant to be the same for every application within the domain. They are much less flexible than the hot spots (the variation points). It's generally easy to see when developers are subclassing frozen classes. Such modifications make the application very brittle to changes in the framework, and require a lot of extra work to 'force fit' the framework to do what you want (which is opposed to the way frameworks are meant to be used).

Example: Persistence Manager is the generic interface for storing and retrieving data objects in the CSF. It isolates the rest of the program from the the actual details of how information is permanently stored. FileManagers handle the actual interactions with disk and/or databases or whatever storage mechanism is used. Users of the framework are meant to provide a FileManager, and not change the Persistence Manager in any way. Changing the Persistence Manager to talk directly to a file system is an incorrect use and requires that the FileManager concept be removed. The application then becomes incompatible with new versions of the framework. It also breaks the abstraction so that only one type of storage can be used.

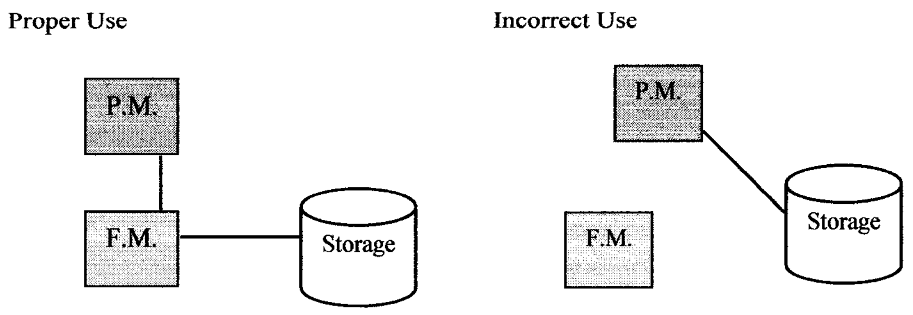

Figure 5.2: Incorrect use of the Persistence Manager in the CSF

5.4.3 Breaking Run-Time Constraints

Some properties have to be maintained when using a framework. These are framework specific, and tend to cause some of the more subtle errors in application code.

A simple condition for the CSF is 'the communication engine (MailServer) must always be enabled for the lifetime of the program.' Shutting it down invokes the clean up of the communication aspects of the framework. Often, users will attempt to shut down the engine and still send or receive messages and then spend a lot of time trying to fix the resulting problems.

5.4.4 Duplicating Services

Every framework provides a certain number of services. If developers aren't aware of them, or dislike them, they may attempt to duplicate the services in their own code. When the framework evolves, they are unable to take advantage of new features of the service, and of course, duplicating the service tends to require a great deal of work.

An obvious example is writing duplicate communication code in the CSF. Some teams did decide to produce parallel communication channels which reduces the benefits of using the framework.

Duplication of the built in dispatch mechanism was also a problem. When a message arrives an object has to decide how it wants to handle it. Basically it registers handlers for each type of message (the default are called messagehandlers). However, some developers dispatch **all** messages to one messagehandler and then write their own dispatch routine within that message handler, which is unnecessary and simply makes their code less efficient.

5.4.5 Using the Wrong Service

Sometimes developers decide to use a service that doesn't meet their needs when a better one is available. That means that the (often unstated) requirements of the application do not match the service provided by the framework. Unfortunately, a lot of work goes into attempting to work around the perceived problems with the service.

For example, messages can be sent in two ways: asynchronously, in which control returns back to the application and no response is required, and synchronously, in which the framework waits for a response (or a timeout) before returning control to the application. Attempting to force fit a synchronous scheme over top of the asynchronous scheme is difficult and error-prone.

Another example involves the mailservers. Users can choose a connection or a connectionless protocol. Unfortunately, users have chosen the connectionless protocol and then tried in vain to automatically detect when a client disconnects from a server, which can be done easily with a connection based protocol.

5.5 Usefulness of Hooks

We used two measures to discern the value of the hooks specifically. These included both objective (errors per service) and subjective measures (perceived difficulty of each service.

5.5.1 Perceived difficulty of each service

The CSF can be divided up into a number of services, but four are of interest for the study.

Hooks were provided for:

- Asynchronous communication – sending messages across the network without blocking the sender.
- Mail server – choosing and using the correct 'engine' for message passing based on the type of application being produced.

Hooks were not provided for:

- Synchronous communication – sending messages across the network and then waiting for an immediate reply within a given time limit.
- Persistence – saving and restoring objects to and from persistent storage.

All of the services were described in the overall documentation and were a part of the overall process in order to minimize the tendency for developers to use only the parts of the framework they are aware of.

We then examined the surveys provided to the students and the actual difficulties encountered. Over the three terms, the services were ranked in the following order of difficulty to understand and use:

1 Synchronous communication (most difficult)
2 Persistence

3 Mail Servers

4 Asynchronous communication (least difficult)

Clearly the parts of the framework without hooks were perceived as being more difficult to use than the parts with hooks. However, how did this translate to actual errors.

5.5.2 Errors per service

Finally, to help judge the effectiveness of hooks and examples, we divided the errors made by each group into the different services that they affected.

Table 5.4: Average Errors by Framework Service

Term	Synchronous	Persistence	Sychronous + Persistence	Asynchronous	Mail Server	Asynchronous + Mail Server
1	0.83	0.5	1.33	0.67	0.67	1.34
2	0.83	0.33	1.16	0.5	0.17	0.67
3	0.5	0.25	0.75	0.25	0.25	0.5

In term one, the errors were roughly evenly spread over the different services, and it made little difference whether hooks were available for the service or not. The numbers correspond to the difficulty the framework users had in understanding the hooks and the framework in the first term. However, over the three terms, the number of errors decreased more quickly in the services with hooks than those without hooks. The errors made with the Mail Server components decreased significantly, as did those with the asynchronous message passing service.

The number of errors in the services without hooks also decreased due to improvements in the overall documentation. However, they did not decrease as much as those with hooks. The best comparison can be made between the synchronous and asynchronous services. They have similar set up and similar usage within the framework, but the use synchronous message passing produced more errors and did not improve as much as the use of asynchronous message passing.

While errors in the services with hooks were less than in the services without hooks, the difference was not as great as expected. However, it was found that the types of errors were different in the unhooked sections than the hooked sections as shown in Table 5.5. Many of the errors in the sections for which hooks were provided tended to be errors made in enacting the hooks themselves. That is, the errors were simple mistakes, such as skipping a step in the hook, which could be easily corrected. Errors in the sections for which hook descriptions were not provided tended to be more significant, such as using the wrong service, or breaking constraints on the framework.

Table 5.5: Average Types of Errors per Term with Hooks / without Hooks

Term	Hook Error	Modifying Frozen Spots	Duplicating Services	Breaking Constraints	Using the Wrong Service
1	0.67 / 0.5	0 / 0.17	0.33 / 0.33	0.17 / 0	0.17 / 0.33
2	0.5 / 0.67	0 / 0	0.17 / 0	0 / 0.17	0 / 0.33
3	0.25 / 0.25	0 / 0	0.25 / 0	0 / 0.25	0 / 0.25

The significant errors that occurred in sections with hooks in terms 2 and 3, were unexpectedly, duplicating framework services. In most cases, this was a conscious decision to circumvent the framework's capabilities in favor of custom functionality.

5.6 Summary

We conducted a study of the use of a small framework over three terms of a senior year software engineering course in order to learn how new users approach development using a framework and to evaluate the effectiveness of hooks. Over the course of the study, we refined our documentation for the framework and learned that an overall process, design description, examples, code and hooks are important to the successful use of a framework.

We found that hooks help in the use of the framework in three ways. First, they reduce the perceived difficulty of using a part of the framework as there is visibly documentation explaining how to perform tasks with the framework. Second, they reduce the number of errors that users will make with a part of the framework, although that reduction in total errors was not as great as expected. Finally, they reduce the severity of the errors that users make with the framework. More serious errors require more effort in using the framework and thus decrease the benefit of using the framework. This case study has indicated that providing hooks for a framework does make a framework easier to use.

Chapter 6 – Tool Support

6.1 Introduction

The hooks model was originally designed to have some form of tool support and our studies with framework usage reinforced the need for that tool support. We found that it is easy to make simple errors that tool support would help eliminate. In fact, much of the work involved in using a hook can be automated.

There is already a strong case for tool support for frameworks. Graphical user interface builders represent one of the most successful applications of framework technology. GUI builders allow developers to quickly piece together a window-based user interface from predefined components such as buttons, windows and menus. The interactions of the components are well-defined within the framework but the components are given user-defined functionality. However, frameworks exist in many different areas beyond user interfaces, such as manufacturing [Schmid 95], communications [Hueni 95], operating systems [Campbell et al. 92] and engineering [Hoover et al. 00]. Tools for these kinds of frameworks can help, and in some cases are critical in making their use more successful.

6.1.1 Who would use supporting tools?

A tool for developing applications from frameworks needs to be both an aid to the user and flexible enough to be used in different ways. There are two primary ways in which we envision such a tool being used. First, application developers use the tool to quickly develop applications from the framework without changing the framework. Second, framework maintainers will use the tool to evolve or modify the framework itself. In this Chapter we focus on the application developers.

6.1.2 What value would a tool provide?

1. Automation: a tool can perform many of the mundane tasks associated with using hooks, such as producing the correct design and code to connect the user application to the framework.
2. Tracking and review: each enactment of a hook can be recorded for later review and playback, for bug tracking, evolution, or simply learning how an example was constructed.
3. Learning: a tool can provide an overall process to guide novice users through development using the framework. It can provide examples of use and show users how the hooks are applied. Finally, it can provide a starting point in using the framework for the first time.
4. Generation of tests
5. Analysis of code and design

This chapter will focus on the first three issues, automation, review and learning. The remaining two are interesting additions, but left for future work.

6.1.3 Why a new tool?

Existing tools for commercial frameworks, such as GUI builders, focus exclusively on a single framework (e.g. MFC). A number of research tools, such as FRED, now allow for general framework use, but do not support the integration of multiple frameworks. In the case of multiple frameworks, users must learn multiple tools and do integration between frameworks by hand. A tool based on hooks can be flexible enough to support many different frameworks. However, general tools need to be flexible enough to allow customizations of the tool itself, specific to the framework(s) being used. That is, a tool should be a framework itself that serves as the basis for creating development tools for using frameworks. A hooks tool will also enable framework builders to adapt the tool to their framework instead of going through the expense of developing a custom tool for each framework, or not providing tool support at all.

6.2 Overview of a Hooks Tool

A hooks tool is primarily meant to aid in the use of hooks. Such a tool should interact with other tools as part of the overall design and implementation stages of development. As shown in Figure 6.1, the hooks tool, called HookMaster [Liu 01], takes as input a database of hook descriptions. The hook descriptions catalog all of the hooks for the frameworks to be used.

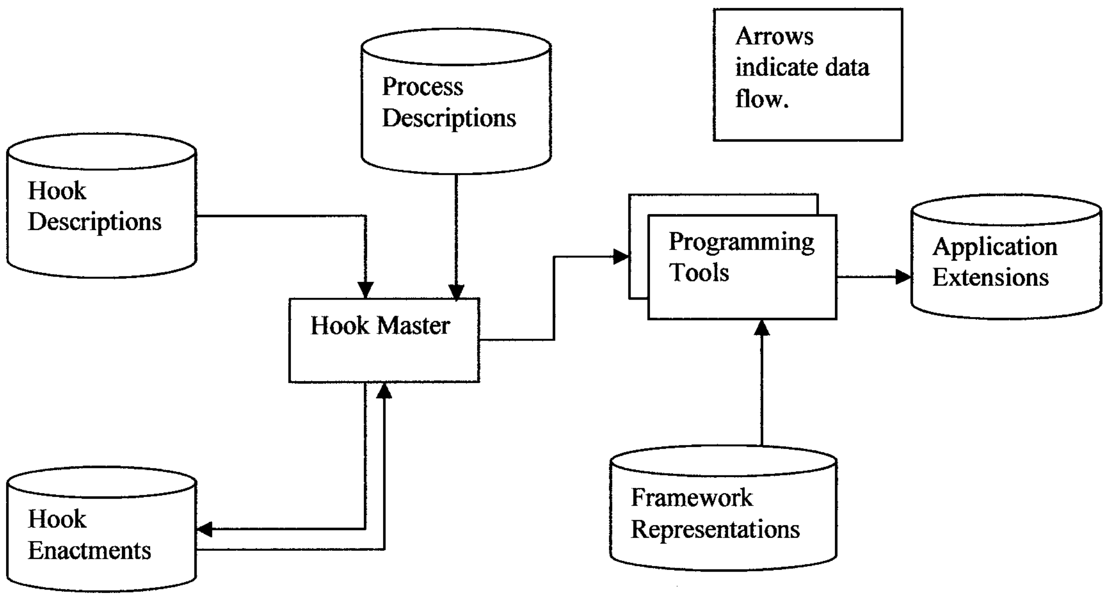

Figure 6.1: Overview of Hooks Tool

The purpose of HookMaster is to aid in hook enactments. Hook enactment is the process of actually using a hook, that is, interacting with the user and making the extensions specified within the hook. The hook enactments database contains a record of all of the hooks that have been used, along with how and where they were used. Specifically it records the values for all parameters within a hook. So if an application developer selects and enacts the New Data hook (from the CSF), the enactment record would contain the name of the class created, any data fields entered and the code for the required methods. The hook enactments database contains the hook enactments for a single application. HookMaster will also take in process descriptions (outlined in Chapter 5) which guide the application developer through the actual use of the frameworks. There may be many different process descriptions for a framework or combination of frameworks.

When enacting a hook, HookMaster will also interact with other tools. Target tools include UML design tools such as Rational Rose, and implementation tools such as the Java Master editor developed as part of the Framescan project. These tools, unlike hook master, deal directly with some representation of the framework – typically UML diagrams or source code. A protocol has been developed to handle this interaction between Hook Master and external tools (see Appendix B for the protocol and Java examples). Basically the protocol allows Hook Master to send commands to these tools in order to actually put in place the changes specified within the hook. During the above enactment of the New Data hook, Hook Master would interact with, for example, both Rational Rose and Java Master to add the new class to the UML diagram and to actually create the source code for it. All of the changes are stored as application extensions. The HookMaster tool itself should be extensible by allowing it to connect to new tools, allowing new processes and hooks to be defined, allowing custom hook directives (via the Note change) and allowing new types of interactions with the user (user interfaces).

6.3 Capabilities of the Tool

6.3.1 High Level Process Support

Novice users are one of the main targets of a tool. However, simply showing them a list of hooks for a framework and then expecting them to be able to dive right in and use them is unreasonable. Novice users need some form of high level support to help them get started using the framework.

The tool can provide this support by incorporating one or more high level processes associated with the framework. The process is defined initially by the framework developer, but framework users can add their own processes, or modify existing ones as they gain experience. Once the novice has an initial set of application requirements, the high level process would guide the user through the selection and enactment of the appropriate hooks to meet those requirements. As outlined in chapter 6, such a process was provided (in html form only) as part of the study involving the CSF. The process not only helped the user decide what hooks they need to use, but also helped them to apply the hooks in the correct order.

For example, an application developer is producing a simple chat application that will connect multiple users on multiple computers. The first step in the process given in Chapter 5 for the CSF framework is to discover data – identify the information that the program needs to pass across the network. In a sample application, to provide a basic level of authentication, the application developer might pass an authentication record consisting of a user name and password from the client to the server. The discover data process step points to the New Data hook. The application developer following the process then goes to the New Data hook and enacts it.

6.3.2 Translating Hooks to Design and Code

Hook enactment involves having the user choose a hook, or presenting the user with one, and having them fill in the details required by the hook. Since the parameters are already defined for a hook, the user need simply provide values for them. To continue the above example (chat application), the application developer was directed to the New Data hook. The parameters for the New Data hook are:

- NewData: a new subclass of the Data class
- myVar: the instance variables for the new subclass
- NewData.readData(): a method to read the instance variables from a stream
- NewData.writeData(): a method to write the instance variables to a stream

In this example, the application developer supplies the following values for the parameters:

- NewData = Auth_Rec
- myVar = (id : string, password : string)

The application developer is also presented with method headings for the two methods and must fill in the appropriate code for the readData and writeData methods. HookMaster then translates the changes given by the user into commands that are sent to the tools connected to HookMaster. For example, the completed hook enactment can be sent to Rational Rose and Java Master to realize the enactment in UML and Java code form. The tool generates the following commands:

- CreateClass(chat,Data,Auth_Rec) – to create the new subclass
- CreateProperty(chat,Auth_Rec,id,string) – to create the instance variables
- CreateProperty(chat,Auth_Rec,password,string)
- CreateMethod(chat,Auth_Rec,Auth_Rec,(),void) – to create a constructor needed by the framework. This is a change specified in the hook and can be performed automatically once the class name is known.
- OverrideMethod(chat,Data,Auth_Rec,readData,(in : DataInput),void) – to create the method to read data from the stream
- AddCode(chat,Auth_Rec,readData,(in: DataInput),*code for readData method*)
- OverrideMethod(chat,Data,Auth_Rec,readData,(out : DataOutput),void)
- AddCode(chat,Auth_Rec,readData,(in: DataInput),*code for readData method*)

In the case of Rational Rose, these are not sent directly to Rose, but to a wrapper that can translate the commands into something Rose will understand. In Rose the following class would be added to the framework diagram connected to the Data class as shown in Figure 6.2.

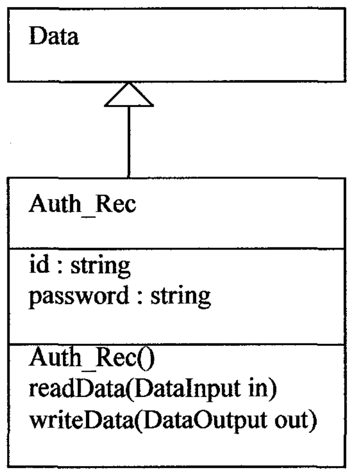

Figure 6.2: UML Representation of the New Data Hook Enaction

The commands are also sent to Java Master which produces the following Java code:

```
public class Auth_Rec extends Data
{
        String id;
        String password;

        public Auth_Rec() {}

        public void readData(DataInput in) throws IOException
        {
                id = in.readUTF();
                password = in.readUTF();
        }


        public void writeData(DataOutput out)
        {
                out.writeUTF(id);
                out.writeUTF(password);
        }
}
```

Finally, an enactment record is also created containing the hook description, the values of the parameters that the application developer specified and a record of which tools the commands were sent to successfully. This information can later be used to review or replay the enactment process.

Hook enactment need not be tied to any one interface style and in fact should not. The simplest way to do it involves the wizard approach used by many companies. In this case, the user would be led through all of the steps of the hook in sequence and have to fill in the appropriate values at each step. An alternative approach could involve the generation of template code (as is done in some tools), or using a property list of all of the variable parts of the hook, allowing the application developer to quickly fill in the desired details. A batch process is also viable, where a number of enactments are stored and then processed by a tool all at once. This is done in the EAF framework where an engineering worksheet is produced from a large number of formulas with each formula being represented by a class. The formulas can be written into a text file and converted to classes all at once without forcing the application developer to do them one at a time.

An additional extension to the hook tool involves directives. A hook may contain a directive for steps outside the scope of the hook language. Such steps might include entering information into an external text file as was done in the Sandwich framework. The application developer must manually ensure that some action is performed. The HookMaster tool is flexible enough to allow extensions that interpret and help automate these directives. In the case of Sandwich, the custom directive is recognized by an addition to the parser and enacted by an addition to the commands recognized by the enactment engine. However, no additions to the protocol are needed.

6.3.3 Reviewing and Replaying

Since hook enactments are recorded and stored, they can also be replayed or reviewed at any time. The replay process simply involves showing how the steps within a hook were actually performed. The steps themselves can be replayed one at a time. In the case of the New Data hook, a replay would show the NewData class added, the variables added and the methods required.

Replaying carries several benefits:

- Hooks enactments can be inspected as part of a formal or informal review process to check for errors or omissions.
- Examples that show the use of the framework can be constructed. The replay process allows application developers to see how the examples were built and the corresponding hooks for each part of an example.
- Hook enactments can be reapplied. That is, a hook enactment can be called up, have minor changes made to it and then be applied to the application as either a new enactment or a modification of the old enactment.

Further, the tracking of hook enactments allows the possibility of rollback to a previous version of an application simply by undoing the enacted steps. However, the consequences of a rollback can be costly, especially in an environment of multiple developers (as is typical) and the full effects and ramifications still need to be studied.

6.3.4 Hook Interference

Modifications to an application made during reviews or during revisions can lead to the problem of invalidating the correctness of previously made hook enactments. The interference occurs when two hooks enact changes which conflict with each other, or when general changes are made to an application that may conflict with a hook enactment. Interference can lead to subtle logic errors in an application. A tool can help to manage interference by maintaining the dependencies between an application and the hooks.

There are two basic types of dependencies:

1. Hooks can require certain structures within a framework and extensions.
2. Hooks can require that certain assertions be true about the framework and extensions.

Both of these are captured within the preconditions of a hook. An example of the first type of dependency is the preconditions of the New MessageHandler hook (in Chapter 4) which state that a CommAwareObject subclass exist and that the subclass have an initialization method.

An example of the second type can be found in Hotdraw. Hooks exist to create new graphical drawing tools, and a hook also exists which prevents all drawing tools from being used in the application. Obviously the creation hooks are invalidated by the prevention hook.

Such dependencies have two consequences. First, if the preconditions of a hook are not met within the current application extensions, then a hook cannot be enacted. Second, and possibly more importantly, if a change is made to the framework or extensions, then those changes may invalidate previous hook enactments. A tool can check both of these by doing a simple brute force check on all hooks and hook enactments to see if their preconditions are still met after a change. A better means of checking for interference involves keeping track of dependencies. Since dependencies exist between hooks and the framework or assertions on the framework, if a modification changes part of the framework, then the hook enactments that depend upon that particular part of the framework can be rechecked. The dependency list can be stored along with the hook enactments.

A distinction can be made between interference during the enactment of hooks and logical errors within the application. Changes might cause some subtle conflict within the application, but this is a logical error in relation to the framework or even the application extensions and not a direct result of the hooks. A logical error invalidates the framework, but does not invalidate the hook. Detection of logical errors is a much larger and more complex problem outside the scope of hooks.

6.4 Integrating hooks with other documentation

The HookMaster tool could also be extended to help guide the new user through all of the information about a framework. Other information such as design diagrams, examples and class descriptions give additional information to the user to aid in application development. Users typically grasp concrete examples more quickly than abstract descriptions. The examples should be both of the use of the frameworks in general (sample applications) and of individual hooks. As described above, examples can be tied to hooks and the replay mechanism of the tool can be used to take new users through an example step by step so that they learn how it was constructed. Descriptions of methods and classes are also necessary. These help application developers to understand the purpose of a class or method as these descriptions are not contained within the hook descriptions.

The high level process, the examples, the class and method descriptions, and the hooks are linked together to form a web of information about the framework which can be easily browsed. Processes link to a series of hooks that are used within the use case. Use cases and hooks also point to examples of the use of the framework, and conversely, examples point to the hooks that have been used within them. The web of information should make navigating and learning a new framework easier.

6.5 Summary

Using the same basic ideas that exist in graphical user interface builders, a tool or tools can be constructed to aid in the use and evolution of object-oriented frameworks. The notion of hooks helps to form the basis of the tool by describing how the framework is intended to be used and showing where changes can be made. A hook tool can aid users by extending the UML language to include hooks and by semi-automatically enacting the changes within hooks. The tool handles propagation of changes between views and helps to prevent inconsistencies. Additional support comes from extensive use of use cases, examples, and class description. To support evolution, the tool is flexible enough to allow hooks to be added or modified along with the framework itself. Finally, the tool is flexible enough to provide support for many different frameworks, or more than one framework at a time.

Chapter 7 – Conclusions and Future Work

7.1 Contributions

This thesis research has made several contributions in the area of O-O frameworks; in particular, in how the use of frameworks can be effectively documented. To provide important background research in the area, a study has been performed over several terms of a senior year software engineering course. A moderately complex framework, the CSF, was built using the notion of hooks and used as the basis of over a dozen applications. Through that experience, we've identified how developers approach and use frameworks, what parts of a framework need to be documented (and how to document them), and what sorts of mistakes developers make when using frameworks.

The concept of hooks themselves has been evaluated by working with developers of the EAF and Sandwich frameworks to document the intended use of their frameworks. Existing, external frameworks were also examined to determine the types of hooks that exist and to ensure that the hooks model covered the types of change discovered.

7.1.1 Documenting Framework Usage

Current practices of writing natural language descriptions and cookbook style 'recipes' to describe how to use a framework can be imprecise and miss important information. Even examples, while very valuable in showing specific uses, can exclude important points that the user needs to know. Further, the above styles cannot be easily automated to provide support for new users of a framework. The hook notation identified helps enforce precision in documentation and can be parsed by a tool. We have also identified some of the relevant properties of each type of variation point. These properties include ease of use, flexibility and testability.

The hooks model was evaluated through the aforementioned study and we found that providing hooks does somewhat reduce the number of errors that new developers make when using a framework. Providing hooks also lowers the perceived difficulty of using the framework and more importantly reduces the severity of the mistakes made.

7.1.2 Application of the Hooks Model to Documenting and Building Frameworks

The hooks model has been tested on several frameworks. Testing the model has involved not only developing hook descriptions for existing frameworks, but working with framework developers. I have also developed a new framework, CSF, incorporating the concept of hooks. In particular, the hooks model was applied in detail to four different frameworks:

1. Hook descriptions were developed for the HotDraw framework, written in Smalltalk, and an application was developed from that framework based on the hooks described.
2. Hook descriptions were also developed for the Engineering Application Framework from Avrasoft in conjunction with the developer of the framework. The framework is written in Object Pascal, and showed that hooks can be easily applied to different languages. The exercise of applying the hooks model also showed how describing the intended use of a framework can affect and simplify the design of a framework.
3. Hooks were applied to the Sandwich framework (a person web assistant) in conjunction with the developer [Liew 99]. Sandwich is written in the Java language. This experience corroborated our experience with the EAF in that applying hooks can help to simplify the design of a framework. It also uncovered how modularity is important to hook design.

4. I developed a new framework for this research, called the Client-Server Framework (CSF) in Java. Development of this framework allowed me to incorporate hooks into the design from the beginning. This framework was then used as the basis for the study into framework use reported in Chapter 6.

7.1.3 Identification of Common Patterns and Aids to Design

For each type of hook, patterns, much like design patterns, have been identified. The patterns follow the context-problem-solution template used by [Gamma et al. 95]. These patterns primarily focus on ease of use vs. flexibility and can be used by framework designers to build the desired level of flexibility into a framework.

We also found that the act of developing the hook documentation provided a valuable amount of feedback on the design of the framework. In particular, when hooks became complex to describe (that is, required a large number of change statements along with additional comments to the developer that could not be captured with change statements), the underlying design could be simplified to promote better understanding with users of the framework. The review process could also identify inconsistencies or missing functionality in the framework.

7.1.4 Types of Documentation Needed to Support a Framework

Hooks are focused on the specific changes needed to fulfill a particular requirement in an application. However, other documentation is also required. Examples are invaluable for showing concrete uses of a framework. Design documentation and code is also required to foster architectural understanding and to allow developers to trace problems that they encounter when using the framework.

Our study into framework use also identified the need for a high-level process to guide first time users through application development using the framework. An index of common questions and where to find the answers also helped greatly. We found that after two rounds of review, enough questions were answered that the developers were able to confidently use the framework without requiring support from the framework developer.

7.1.5 Types of Mistakes Made with using Frameworks

Through the study, we were able to identify how new users approach development using frameworks and the types of problems they have. The types of mistakes have been characterized. Users may use one service when another would have suited their needs better, may attempt to duplicate services, may break constraints on the framework, may attempt to evolve the framework unnecessarily, or may make simple mistakes when implementing hooks.

7.1.6 Enabling General Tool Support for Framework Use

As the notation for hook descriptions was designed to be parsible and enactable, tool support can easily be provided for selecting and using hooks. A proof-of-concept tool called HookMaster has been developed that automatically parses hook descriptions and interactively works with application developers to quickly produce consistent application extensions to the framework, as well as track and review the extensions made.

7.2 Future Work

As with all research, a number of interesting questions have been discovered during the course of this thesis.

7.2.1 Formalization

The current hook description is structured and precise, but can benefit from a firm grounding in formal logics. The most pressing need is a full description of the types of pre and post conditions that are appropriate for hooks.

Formal descriptions of these conditions could also help in determining whether or not hooks interfere with one another. That is, if the changes made by one hook invalidate or conflict with the changes (or potential changes) made by another hook.

7.2.2 Metrics

It is possible to characterize several of the properties of a hook via the type system; however, a more rigorous approach to measuring those and other characteristics can be developed because of the base that the hooks provide. Some of the properties of interest are complexity (to judge whether or not a hook will be difficult to implement or the underlying design is simply too complex), ease of use and flexibility.

7.2.3 Tool development and refinement

A production-quality tool for enacting hooks can be developed, as evidenced by the prototypes we have produced. Once such a tool is available, it should be used in further studies and refined. Some of the key issues with tool development still to be addressed are:

- User interface: what is the best type of interface for various types of users from novice to expert? While a wizard style may be appropriate for novices, it will likely be too obtrusive for experts. The best type of interface for experts has yet to be determined.
- Integration with other tools: thus far, we have experienced some difficulty interfacing with tools like Rational Rose. Integration is always a difficult problem, and we've attempted to limit the amount of interaction needed and to carefully define a protocol for information passing between a hook tool and other tools.
- Tracking user changes: one of the biggest issues facing the tool is dealing with changes made outside of hook use. Should developers be allowed to access (and modify) the code directly, or should it all be through hooks. Once users are allowed to change the code, then it becomes difficult or impossible for the hooks tool to accurately track hook enactments and changes. It might seem that we cannot limit access to the code; however, it should be possible by limiting the level of hooks used.

7.2.4 Automated test case generation

When users enact hooks, either manually or automatically, it is possible to automatically generate regression tests for that hook. Hooks capture something that many other techniques do not, that is the intent of the user. For example, a hook for the observer pattern would not only capture the fact that there are supposed to be subjects and observers, but also which objects are supposed to be observing which subjects. Tests could be generated to ensure that the intent of the user is actually reflected in the code.

7.2.5 Controlled experiments

The study performed was valuable in that it allowed us to observe how novices approach framework development and to observe what mistakes they made or what techniques worked particularly well. However, it was difficult to rigorously control the study and some of the results are subjective. More experiments should be performed to confirm our results using different frameworks and a more controlled study set up.

7.2.6 Investigating the evolution of frameworks

Finally, while some preliminary work has been done with the evolution of frameworks, much remains unknown. Frameworks have two axes of change: variation in space and variation in time. Variation in space concerns the domain of the framework. The space represents the different applications that can be built using the framework. Variation in time is more traditionally thought of as evolution. How the two types of variation are related is one an important question currently facing the frameworks community. This question has implications for hooks as well. Hooks deal with variation in space, but if the two types of variation are closely aligned or are equivalent, then hooks can be written to specifically evolve frameworks and applications built on frameworks. That is, hooks potentially can be built into a framework to allow applications to smoothly evolve over time.

Ralf Johnson [Johnson 1992] has stated that frameworks evolve from white-box (inheritance driven and open ended) to black-box (component driven). Our characterization of variation points tends to support this claim by demonstrating that variation points can evolve from white-box to black-box. Open hooks can be made into template hooks by adding a type system to the hook parameters. Template hooks can be made into option hooks by enumerating and including all choices within the framework.

We also benefited from experience with the evolution of our frameworks. Both the CSF and EAF/Prothos have been used extensively, and have had to evolve to accommodate new features, or fix problems. With the CSF, we've found when using the hooks as a strict interface to the framework, that we can modify the frozen spots of the framework without forcing users to redevelop or modify their applications. However, these results are preliminary and need more study in order to clearly determine under what conditions framework changes leave existing applications unaffected.

Certainly, hooks show promise as a means to determine whether or not an application needs to be changed when a framework evolves. There are a number of ways in which the framework and/or the hooks can evolve.

- *Changes to the underlying framework.* These occur when the common requirements change, or new ones arise, or to make the framework more robust. Such changes can include refactoring [Opdyke and Johnson 90] the framework classes without affecting the hooks. The variable requirements themselves don't change so the hooks don't change either; the only changes are in the underlying implementation.

- *Changing hooks to make them more flexible.* Sometimes the variable requirements supported by the framework are not flexible enough to accommodate the requirements of the products within the domain. Generally, one or more hooks need to be evolved along with parts of the underlying framework in order to support the new flexibility. As shown in figure 4, to make a hook more flexible, it must be made more open as well.

- *Changing hooks to provide more support for the product developer.* As a framework matures, the exact variations between products within a domain can be catalogued and support for the different variations built directly into the framework, making it more of a black-box framework [ref Johnson]. Building in variations makes the framework easier to use as it contains more support for the product developer.

- *Adding new hooks*. New hooks can be added in two ways: As users gain more experience with a framework they may discover ways to support a requirement that was not documented by the original framework developers. These experiences can be documented in a hook description, usually an open hook. New functionality can be added to the framework itself, and this functionality accessed through hooks.

More work needs to be done to determine when hooks are affected by evolution to the underlying framework. A more formal model of frameworks will help here and also help to determine just how the application should evolve when changes to the hooks are made.

7.3 Summary

This work has made several contributions to the study of the design and use of frameworks. The concept of hooks has been introduced to document how a framework is intended to be used. This concept has been evaluated by applying it to existing frameworks, by working with other framework developers and by developing new frameworks. A set of guidelines for developing hooks and a collection of patterns from existing frameworks has been gathered to help developers choose the types of changes they wish to allow in their frameworks. A study was undertaking to help understand how frameworks are used and to evaluate the effectiveness of hooks. From that study, a recommended set of documentation has been produced as well as an outline of the types of errors that users make. Hooks were also shown to somewhat effective in reducing the numbers of errors new users make using frameworks, but also to reduce the severity of the errors they make.

A tool for supporting hooks is under development, and its overall purpose and goals have been outlined. The tool helps to automate parts of the hooks, to guide users through the process of developing applications using the hooks and can also be used as a learning or reviewing tool through the replay mechanisms.

The work has also opened a number of new questions and issues that require future study. Further experiments are needed to further refine our knowledge about frameworks. More work needs to be done with automated test case generation. Evolution of frameworks is also a particularly interesting and relevant area of exploration. Finally, the hook descriptions have begun to be adopted (for example see the SalesPoint framework at http://ist.unibw-muenchen.de/Lectures/SalesPoint/javadoc/.index.html), but more outside evaluation is also needed.

References

[Aarsten et al. 00] A. Aarsten, D. Brugali and G. Menga. A CIM Framework and Pattern Language. In Domain Specific Application Frameworks: Frameworks Experience by Industry. M.E. Fayad, R. Johnson (eds.). John Wiley & Sons. 2000. 21-42.

[Adair 95] D. Adair, Building Object-Oriented Frameworks, AIXpert, February 1995.

[Alexander 77] C. Alexander. A Pattern Language: Towns, Buildings, Construction. Oxford University Press, New York. 1977.

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

[Beck 94] K. Beck. Patterns and Software Development. Dr. Dobb's Journal, Feb. 1994. 18-22.

[Birrer and Eggenschwiler 93] A. Birrer and T Eggenschwiler. Frameworks in the Financial engineering Domain: An Experience Report. In Proceedings of ECOOP'93, 1993.

[Boehm-Davis et al. 88] D.A. Boehm-Davis. Software Comprehension. In the Handbook of Human-Computer Interaction, M. Helander (ed.). Elsevier Science Publishers B.V. North Holland, 1988. 107-121.

[Booch 94] G. Booch. Object-Oriented Analysis and Design with Applications. The Benjamin/Cummings Publishing Company, Inc., Redmond City, CA. 1994.

[Booch 95] G. Booch. Object Solutions: Managing the Object-Oriented Project. Addison-Wesley, Reading, MA. 1995.

[Bosch 98] J. Bosch. Design Patterns as Language Constructs. Journal of Object-Oriented Programming, Vol. 11, No. 2, May 1998, 18-32.

[Bosch 99] J. Bosch. Framework Problems and Experiences. In Building Application Frameworks: Object-Oriented Foundations of Framework Design. M. Fayad, R. Johnson and D.C. Schmidt (eds.). John Wiley & Sons, 1999. 55-82.

[Bosch 00] Bosch J.: Design & Use of Software Architectures – Adopting and Evolving a Product-Line Approach. Addison-Wesley, 2000.

[Braga et al. 99] R. Braga, F. Germano and P. Masiero. A Pattern Language for Business Resource Management. Proceedings of the 6th Pattern Languages of Programs Conference (PLoP'99), Monticello-IL, USA. 1-34.

[Brooks 83] R. Brooks. Towards a theory of the comprehension of computer programs. Intl. Journal of Man-Machine Studies, 18. 1983. 543-554.

[Brown et al. 95] K. Brown, L. Kues and M. Lam. HM3270: An Evolving Framework for Client-Server Communications. In Proceedings of the 14th Annual TOOLS (Technology of Object-Oriented Languages and Systems) Conference, Santa Barbara, CA. 1995. 463-472.

[Brugali and Menga 99] D. Brugali, and G. Menga. Frameworks and Pattern Languages: an Intriguing Relationship. ACM Computing Surveys, March 1999.

[Brugali et al. 00] D. Brugali, G. Menga and A. Aarsten. A Case Study for Flexible Manufacturing Systems. In Domain-Specific Application Frameworks: Frameworks Experience by Industry, M.E. Fayad, R. Johnson (eds.). John Wiley & Sons. 2000. 85-99.

[Budinsky et al. 96] F. Budinsky, M. Finnie M, J. Vlissides J and P. Yu. Automatic Code Generation from Design Patterns. IBM Systems Journal, Vol. 35, No. 2, 1996, 151-171.

[Campbell and Islam 92] R.H. Campbell and N. Islam. A Technique for Documenting the Framework of an Object-Oriented System. In Proceedings of the 2nd International Workshop on Object-Orientation in Operating Systems. Paris, France. 1992.

[Campbell et al. 92] R.H. Campbell, N. Islam, D Raila and P Madany. Designing and Implementing Choices: An Object-Oriented System in C++. Communications of the ACM, 36(9), Sept. 1993. 117-126.

[Campo et al. 97] M. Campo, C. Marcos and A. Ortigosa. Framework Comprehension and Design Patterns: a Reverse Engineering Approach. In Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, Madrid, España, June 1997.

[Campo and Price 99] M. Campo and T. Price. Luthier- Building Framework-Visualization Tools. In: Implementing Object-Oriented Application Frameworks: Frameworks at Work, Mohamed Fayad, Ralph Johnson (Eds.), Wiley, USA, September1999.

[Carey et al. 00] J. Carey, B. Carlson and T. Grasner. SanFrancisco Design Patterns: Blueprints for Business Software, Addison-Wesley. 2000.

[Cohen et al. 92] S. Cohen, J. Stanley Jr., A. Peterson and R. Krut Jr. Application of Feature Oriented Domain Analysis to the Army Movement Control Domain. Technical Report CMU/SEI-91-TR-028. Software Engineering Institute. 1992.

[Cotter and Potel 95] S. Cotter and M. Potel. Inside Taligent Technology. Addison-Wesley. Reading, MA. 1995.

[Chen and Chen 94] D.J. Chen and D.T. Chen. An experimental study of using reusable software design frameworks to achieve software reuse. Journal of Object-Oriented Programming, May 1994.

[Clements and Northrop 01] P. Clements and L.M. Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley. 2001.

[Demeye et al. 97] S. Demeyer S, T. Meijler, O. Nierstrasz and P. Steyaert. Design Guidelines for Tailorable Frameworks. Communications of the ACM, Vol. 40, No. 10, October 1997, 60-64.

[Deutsch 89] L.P Deutsch. Design Reuse and Frameworks in the Smalltalk-80 system. In T. J. Biggerstaff and A. J. Perlis, editors, Software Reusability, Vol II, ACM Press, 1989

[Durham and Johnson 96] A. Durham and R. Johnson. A Framework for Run-time Systems and its Visual Programming Language. In Proceedings of OOPSLA '96, Object-Oriented Programming Systems, Languages, and Applications. San Jose, CA. October 1996.

[Edwards 90] S.H. Edwards. The 3C model of Reusable Software Components. In Proceedings of the Third Annual Workshop: Methods and Tools for Reuse. June 1990.

[Eggenshwiler and Gamma 92] T. Eggenshwiler and E. Gamma. ET++ SwapsManager: Using Object Technology in the financial Engineering Domain. In Proceedings of OOPSLA92, 1992. 166-177.

[Erdem et al. 98] A. Erdem, W.L. Johnson and S. Marsella. Task Oriented Software Understanding. In Proceedings of the 13th Automated Software Engineering Conference. 1998.

[Fayad and Cline 96] M. Fayad and M.P. Cline. Aspects of Software Adaptability. In Communications of the ACM 39(10). October 1996. 58-59.

[Florijn et al. 97] G. Florijn G, M. Meijers and P. van Winsen. Tool Support for Object-Oriented Patterns. In Proceedings of the 11 th European Conference on Object-Oriented Programming (ECOOP'97), LNCS 1241, 1997, 472-496.

[Fontoura et al. 00] M. Fontoura, W. Pree, and B. Rumpe. UML-F: A Modeling Language for Object-Oriented Frameworks, 14th European Conference on Object Oriented Programming (ECOOP 2000), Lecture Notes in Computer Science 1850, Springer, Cannes, France. 2000. 63-82.

[Froehlich et. al. 97] G. Froehlich, H.J. Hoover, L. Liu and P.G. Sorenson. Hooking into Object-Oriented Application Frameworks. In Proceedings of the 19th Intl Conference on Software Engineering, Boston, May 1997. 491-401.

[Froehlich et. al. 99a] G. Froehlich, H.J. Hoover, L. Liu and P.G. Sorenson. Designing Object-Oriented Frameworks. In CRC Handbook of Object Technology. CRC Press, 1999. Chapter 26.

[Froehlich et. al. 99b] G. Froehlich, H.J. Hoover, L. Liu and P.G. Sorenson. Using Object-Oriented Frameworks. In CRC Handbook of Object Technology. CRC Press, 1999. Chapter 26.

[Froehlich et. al. 99c] G Froehlich, H.J. Hoover, L. Liew and P.G. Sorenson. Application Framework Issues when Evolving Businiess Applications for Electronic Commerce. In Proceedings of the 32nd Hawaii International Conference on Systems Sciences (HICSS 32) January 1999, Maui, Hawaii.

[Froehlich et. al. 99d] G. Froehlich, H.J. Hoover, L. Liu and P.G. Sorenson. Reusing Hooks. In Object-Oriented Application Frameworks. M. Fayad, D.C. Schmidt and R. Johnson (eds.) John Wiley, 1999. 219-235.

[Froehlich et. al. 00] G. Froehlich, H.J. Hoover and P.G. Sorenson Choosing an Object-Oriented Domain Framework. In ACM Computing Surveys 2000.

[Gamma et al. 95] E. Gamma, R. Helm, R Johnson and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA. 1995.

[Gangopadhyay and Mitra 95] D. Gangopadhyay and S. Mitra. Understanding Frameworks by Exploration of Exemplars. In Proceedings of the 7th Intl. Workshop on Computer Aided Software Engineering. Toronto, Canada. 1995. 90-99.

[Goldberg and Robson 89] A. Goldberg and D. Robson. Smalltalk-80: The Language. Addison-Wesley. 1989.

[Hakala et al. 01a] M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa, and J. Viljamaa. Annotating Reusable Software Architectures with Specialization Patterns In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), Amsterdam, August 2001, 171-180.

[Hakala et al. 01b] M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa, and J. Viljamaa. Generating application development environments for Java frameworks. In: Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering (GCSE'01), Erfurt, Germany, September 2001, Springer, LNCS2186, 163-176.

[Hamu and Fayad 98] D. Hamu, M. Fayad. Achieve Bottom-Line Improvements with Enterprise Frameworks. Communications of the ACM, Vol. 41, No. 8, August 1998.

[Helm et al. 90] R. Helm, I.M. Holland and D. Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In Proceedings of OOPSLA'90. Ottawa, Canada. 1990. 169-180.

[Hoover et al. 00] H.J. Hoover, T. Olekshy, G. Froehlich and P.G. Sorenson. Developing Engineered Product Support Applications. In Proceedings of the 1st Software Product Line Conference, Denver CO, August 2000. 451-476.

[Hueni et al. 95] H. Hueni, R. Johnson and R. Engel. A Framework for Network Protocol Software. In Proceedings of OOPSLA'95. Austin TX. 1995.

[Jacobson et al. 99] Jacobson I., Rumbaugh J., Booch G.: The Unified Software De-velopment Process. Addison-Wesley, 1999.

[Jazayeri et al. 00] Jazayeri M., Ran A., van der Linden F.: Software Architecture for Product Families. Addison-Wesley, 2000.

[Johnson and Foote 88] R. Johnson and B. Foote. Designing Reusable Classes. Journal of Object-Oriented Programming, 2(1), June/July 1988. 22-35.

[Johnson 92] R. Johnson. Documenting Frameworks Using Patterns. In Proceedings of OOPSLA'92. Vancouver, Canada. 1992. 63-76.

[Keefer 94] P. Keefer. The Accounts framework. Master Thesis, University of Illinois, 1994.

[Koskimes and Mossenback 95] K. Koskimes and H. Mössenback. Designing a Framework by Stepwise Generalization. In Proceedings of the 5th European Software Engineering Conference, Barcelona. Lecture Notes in Computer Science 989, Springer-Verlag, 1995. 479-497

[Krasner and Pope 88] G.E. Krasner and S.T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming. 1(3), August/Sept. 1988. 26-49.

[Krueger 92] C.W. Krueger. Software Reuse. ACM Computing Surveys, 24(2), June 1992. 131-183.

[Lajoie and Keller 94] R. Lajoie and R.K. Keller. Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts and Motifs in Concert. In Proceedings of the 62nd Congress of the Association Canadienne Francaise pour l'Advancement des Sciences. Montreal Canada. 1994.

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

[Läufer 95] K. Läufer. A Framework for Higher-Order Functions in C++. In Proceedings of the Conference on Object-Oriented Technologies (COOTS), Monterey, CA, June 1995.

[Lieberherr et. al. 88] K.J. Lieberherr, I. Holland and A.J. Riel. Object-Oriented Programming: An Objective Sense of Style. In Proceedings of OOPSLA'88. San Diego, CA. Sept. 1988. 323-334.

[Liew 99] W. Liew. Sandwich: A Personal Web Assistant. MSc. Thesis. University of Alberta, Edmonton, AB, Canada. 1999.

[Liskov 88] B. Liskov. Data Abstraction and Hierarchy. SIGPLAN Notices, 23, 5. May 1988.

[Liu 01] L. Liu. A Tool for Supporting Hooks. MSc. Thesis. University of Alberta, Edmonton, AB, Canada. 2001.

[Magee et al. 94] J. Magee, N. Dulay and K. Kramer. Regis: A Constructive Development Environment for Distributed Programs. In IEEE/IOP/BCS Distributed Systems Engineering, 1(5) September 1994. 304-312.

[Mattsson 96] M. Mattsson, Object-Oriented Frameworks – A Survey of Methodological Issues. Licentiate thesis, LU-CS-TR, 96-167, Department of Computer Science, Lund University, 1996.

[Mattsson and Bosch 97] M. Mattsson and J. Bosch. Framework Composition: Problems, Causes and Solutions, Michael Mattsson. In Proceedings of TOOLS USA '97.

[Meyer 94] B. Meyer. Reusable Software: The Base Object-Oriented Component Libraries. Prentice Hall. 1994.

[Moser and Nierstrasz 96] S. Moser and O. Nierstrasz. The Effect of Object-Oriented Frameworks on Developer Productivity. IEEE Computer. Septmeber 1996. 45-51

[Myers 95] W. Myers. Taligent's CommonPoint: The Promise of Objects. IEEE Computer. March 1995. 78-83.

[Opdyke and Johnson 90] W.F Opdyke and R. Johnson. Refactoring : An Aid in Designing Object-Oriented Application Frameworks. In Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications, 1990.

[Ortigosa and Campo 99] A. Ortigosa, M. Campo. SmartBooks – A Step Beyond Active-Cookbooks to Aid in Framework Instantiation. In: Technology of Object-Oriented Languages and Systems 25. IEEE Press, June 1999, 131-140.

[Ortigosa and Campo 00] A. Ortigosa and M. Campo. Towards Agent-Oriented Assistance for Framework Instantiation. In Proceedings of OOPSLA 2000. October 2000.

[ParcPlace 95] VisualWorks Cookbook. Release 2.5. ParcPlace-Digitalk Inc. Sunnyvale, CA. 1995.

[Parnas 76] D.L. Parnas. On the Design and Development of Program Families. IEEE Transactions on Software Engineering 2(1). March 1976. 1-9.

[Pasetti and Pree 00] Pasetti A., Pree W.: Two Novel Concepts for Systematic Product Line Development. In: Donohoe P. (eds.): Software Product Lines: Experience and Research Directions (First Software Product Lines Conference, Denver, Colorado), Kluwer Academic Publishers, 2000.

[Pree 95] W. Pree. (1995). Design Patterns for Object-Oriented Software Development. Addison-Wesley, Reading-MA. 1995.

[Pree 99] W. Pree. Hot-spot-Driven Development. In Building Application Frameworks: Object-Oriented Foundations of Framework Design. M. Fayad, R. Johnson and D.C. Schmidt (eds.). John Wiley & Sons. 1999. 379-393.

[Pree and Koskimies 99] W. Pree and K. Koskimies. Framelets - Small is Beautiful. In: Building Application Frameworks: Object-Oriented Foundations of Framework Design. M.E. Fayad, D.C. Schmidt and R.E. Johnson, ed.), John Wiley & Sons 1999, 411-414.

[Rich and Waters 90] C. Rich and R.C. Waters. The Programmer's Apprentice. ACM Press and Addison Wesley. 1990.

[Roberts and Johnson 96a] D. Roberts and R. Johnson. Evolve Frameworks into Domain-Specific Languages. In Procedings of the 3rd International Conference on Pattern Languages for Programmin, Monticelli, IL, USA, September 1996.

[Roberts and Johnson 96b] D. Roberts and R. Johnson. Evolving Frameworks: A Pattern Language for Developing Frameworks. In Proceedings of Pattern Languages of Programs, Allerton Park, Illonois. Sept. 1996.

[Rumbaugh et al. 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorenson. Object-Oriented Modeling and Design. Prentice-Hall Inc. Engelwood Cliffs, NJ. 1991.

[Rumbaugh et al. 99] Rumbaugh J., Jacobson I., Booch G.: The Unified Modeling Language Reference Manual. Addison Wesley, 1999.

[Schauer et al. 99] Schauer R., Robitaille S., Martel F., Keller R.: Hot Spot Recovery in Object-Oriented Software with Inheritance and Composition Template Methods. In: Proceedings of the IEEE International Conference on Software Maintenance 1999 (ICSM '99), Keble College, Oxford, England. IEEE Computer Society Press, 1999, 220-229.

[Shaw and Garlan 96] Shaw M., Garlan D.: Software Architecture - Perspectives on an Emerging Discipline. Upper Saddle River, NJ, Prentice Hall, 1996.

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

[Schmid 97] H.A. Schmid. Systematic Framework Design by Generalization. Communications of the ACM, 40(10). 48-51.

[Schmucker 86] K.J. Schmucker. Object-Oriented Programming for the Macintosh. Hayden Book Company, 1986.

[Shull et al. 00] F. Shull, F. Lanubile and V. Basili. Investigating Reading Techniques for Object-Oriented Framework Learning. IEEE Transactions on Software Engineering, 26(11). November 2000. 1101-1118.

[Sparks et al. 96] S. Sparks, K. Benner and C. Faris. Managing Object-Oriented Framework Reuse, IEEE Computer, Septmeber 1996. 52-62.

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

[Srinivasan and Vergo 98] S. Srinivasan and J. Vergo. Object Oriented Reuse: Experience in developing a framework for speech recognition applications. In Proceedings of the 20th International Conference on Software Engineering. Kyoto, Japan. 1998.

[Taligent 95] Taligent Press. The Power of Frameworks for Windows and OS/2 Developers. Addison-Wesley, Reading MA. 1995.

[Tse 96] L. Tse. A Responsive Corporate Information Model. Ph.D. Thesis, University of Alberta, Edmonton, AB, Canada. July 1996.

[van Gurp et al. 01] J. van Gurp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In Proceedings of WICSA 2001. August 2001.

[van Gurp and Bosch 01] J. van Gurp and J. Bosch. Design, Implementation and Evolution of Object-Oriented Frameworks, Software: Practice and Experience. 33(3), March 2001. 277-300.

[Vlissides and Linton 90] M. Vlissides and M.A. Linton. Unidraw: A Framework for Building Domain-Specific Graphical Editors. ACM Transactions on Information Systems. 8(3), July 1990. 237-268.

[Weinand et al. 88] A. Wienand, E. Gamma and R. Marty. ET++ - An Object-Oriented Application Framework in C++. In Proceedings of OOPSLA'88. San Diego, CA. 1998. 46-57.

[Weiss and Lai 99] D. Weiss and C.T.R Lai. Software Product Line Engineering: A Family-Based Software Development Process. Addison-Wesley, 1999.

[Wirfs-Brok and Johnson 90] R.J. Wirfs-Brock and R. Johnson. Surveying current research in object-oriented design. Communications of the ACM. September 1990.

Appendix A – Grammar for Hook Descriptions

The grammar for hooks follows the hook template described in Chapter 3.

```
<hook> ::=      <name>
                <task>
                <type>
                [<uses>]
                <participants>
                <parameters>
                [<preconditions>]
                <changes>
                [<postconditions>]
                [<commentsblock>]
                [<relatedhooks>]
```

The name of the hook.

```
<name> ::=      Name: <string>
```

The task the hook is meant to aid in performing.

```
<task> ::=      Task: <string>
```

The type of the hook is one of the four basic types, plus an optional string describing the pattern used (see Chapter 4 for patterns).

```
<type> ::= Type: <level>[, <string>]
<level> ::= option | template | open | evolutionary
```

The participants (classes and components) that are used directly within the hook or are of interest to anyone using the hook. Participants can be single entities or defined as a set or sequence of elements. The style indicates whether or not the participant listed is a part of the framework, or is intended to be created as a part of the application.

```
<participants> ::= Particpants: <identifier> [<participant type>] [<style>] [, ..,
        <identifier> [<type>] [<style>] ]
<participant type> ::= (set of <identifier> [, .., <identifier>]) |
        (sequence of <identifier> [, .., <identifier>])
<style> ::= (framework) | (app)
```

The parameters are the actual values that users must fill in when using the hook. Each parameter has a name and a type. Types are intended mainly for the hook tool, so that it can help to enforce that the proper values are provided. The type 'name' refers to any valid class, variable or method name.

```
<parameters> ::= Parameters: <identifier> <parm_type> [, ..,
        <identifier> <parm_type>]
<parm_type> :: = name | variable | string | integer | float | set of <identifier>
                | sequence of <identifier>
```

A list of hooks that the current hook makes use of in the changes section.

```
<uses> ::= Uses: <hook name> [, .., <hook name>]
```

All of the prescribed changes needed to perform the given task. Changes consists of a set of *statements separated by semicolons.*

```
        <changes> ::= <statement> [comment] '\nl' [... <statement> [comment] '\nl']
        <statement> ::=
                <loop statement> |
                <hook statement> |
                <new element statement> |
                <method statement> |
                <modify statement> |
                <parameter statement> |
                <option statement> |
                <behavior statement> |
                <directive> |
                <comment> |
                {<statement>}

        <comment> ::= // <string>
```

Loops can be defined as a repeat loop or a for loop. Repeat loops in this case have no defined end. It is up to the user of the hook to specify when they are done with the loop. For loops perform the loop body once for each element of the set specified in its declaration.

```
        <loop statement> ::= <loop id> <statement>
        <loop id> ::= repeat [as necessary] | forall <var> in <set>
```

One hook uses another hook much in the same way as typical procedures. The hook to be used is said to be the called hook. Parameters can be passed from the current hook to the called hook. To do so, the parameter list is encased in brackets and a parameter in the current hook is mapped into a parameter in the called hook. If no mapping is given, then the parameters in the list are mapped in order to the parameters of the called hook.

A return value can also be specified. The first identifier is the name of a participant within the current hook, and it is given the value of the last identifier within the brackets. For example in "Figure = Choose Figure [ChosenFigure];" the parameter Figure is given the value of ChosenFigure after the hook 'Choose Figure' has been enacted.

```
        <hook statement> ::=    [<identifier> =] <hook name> "[" <identifier> [= <identifier>]
                                [, ..,<identifier> [= <identifier>]]"]";
```

The new element statement is used to create a class, variable (instance variables are called properties and method variables are called localvars), or methods (called an operation).
A new subclass first gives the name of the class and then the name of the class it is inheriting from.
A new property gives the a qualified identifer that gives both the name of the property and the class that it belongs to. A property can be mapped directly to another variable in another class through the where clause.
A new operation gives the name of the method (and class it is in) and possibly an expression defining its return value.

```
<new element statement> ::= [new] subclass <identifier> of <identifier>; |
        [new] property <qualified identifier> <whereclause>; |
        [new] operation <qualified identifier> [<return expression>];
        [new] localvar <identifier> : <identifier> in <qualified identifier>
<whereclause> ::=
        read of <identifier> maps from [set of] <qualified identifier> |
        write of <identifier> maps into [set of] <qualified identifier>
```

Method statements are used to specialize or modify existing methods. Three operations are identified. Copies takes the entire method as is and duplicates it. Extends creates a new method with the implicit declaration that it will call its parent method. Overrides simply declares that a method will be used in place of its parent method. Limited return values can be specified, as can a list of exceptional conditions that can end the method's normal operation (similar to the Java 'throws' statement).

```
<method statement> ::=   <qualified identifier> <method operation>  <qualified identifier>
                         [<return expression>] [<exception expression>];
<method operation> ::= copies | extends | overrides
<return expression> ::= returns <string> |
        returns [set of | sequence of] <identifier>
<exception expression> ::= throws <identifier> [, .. <identifier>]
```

The modify statement is used to directly manipulate code in some way, either by adding a new line of code, removing a line of code or replacing a line of code. The qualified identifier indicates where the code is within the framework/application.

```
<modify statement> ::= remove code '<string>' [, ..., '<string>'] from <qualified identifier>; |
        replace '<string>' with '<string>' in <qualified identifier>; |
        add code '<string>' in <qualified identifier>
```

The parameter statement group is used to get values from the user or to manipulate parameters within the framework. It can be used to add elements to a set, to assign a value to a parameter or to ask the user to speicify a value (using the Acquire command).

```
<parameter statement> ::=
        <identifier> add <set>; |
        Assign <identifier> : <string>; |
        Acquire <identifier> : <parm_type> |
```

When the user needs to choose from a collection of options, the option statement is used. It allows the user to choose one element from a set. To allow for more than one selection, use a loop in conjunction with this statement.

```
<option statement> ::= choose <identifier> from <set>;
```

There are two types of behavior statements. The first, serialize, specifies a list of methods and identifies the order in which they must be invoked. The second indicates that one method should invoke another method. The invocation statement can also indicate that a particular exceptional case must be handled. The provided keyword indicates that the method call is already a part of the framework and the user need not take any additional action.

```
<behavior statement> ::=
        serialize ( <qualified identifier>, <qualified identifier> [,
                .., <qualified identifier> ) [in <qualified identifier>] [provided];|
        <qualified identifier> -> [<read/write>] <qualified identifier> [handles <identifier>]
[provided];
<read/write> ::= read | write | read and write
```

A directive is a note to the user to perform some action. It can be done manually, or the HookMaster tool can be extended to recognize the new command and take a more automated action.

```
<directive> :: =   Note <string>; | * <string>;
```

An identifier is simply a collection of characters conforming to the naming convention used by the language the framework is written in (not shown here). A string is any set of characters.
<set> ::= <identifier> | (<identifier>, <identifier> [, .., <identifier>])
<var> ::= <identifier> | <qualified identifier>
<attribute> ::= <identifier>

A qualified identifier is used to determine where within a hook a particular entity is, be it a method, class or some other thing. It gives the complete path to that entity. An example would be Data.readData defining the method readData within the class Data. Qualified identifiers can also indicate the module or file name a class is in, and can vary based on the language a framework is implemented in.
<qualified identifier> ::= <identifier>.<identifier>[...<identifier>][(<identifier>[, .., <identifier>)]

Conditions can require that a certain entity exists within the framework or the application, that a given class is a subclass of another specified class, that assertions on the framework are met, or they can be a direct note to the user to check some condition.

```
<preconditions> ::= Preconditions: <condition>; [.. <condition>;]
<condition> ::=   <qualified identifier> exists |
                  <qualified identifier> subclass of <qualified identifier> |
                  [not] <assertion> |
                  <string>
<postconditions> ::= Postconditions: <string> [; ..; <string>]
```

<commentsblock> ::= Comments: <string>

<related hooks> ::= Related Hooks: <string>

Appendix B – Protocol for the Hooks Tool

This appendix contains the commands that HookMaster currently translates hook enactments to. The commands are then given to outside tools (or wrappers for those tools) and translated to diagrams and code. The examples and steps given are for JavaMaster, the tool to produce Java code from hook enactments.

Operations

1. Create a class

- verify that the class does not already exist
- create a new class file (ie. Server.java)
- create the new class declaration within the file (with the appropriate superclass)
- in Java query the user (ie. give them a list of choices) as to which Java packages should be imported for use by the class. The framework package should be imported by default.

Protocol:

CreateClass(package, superclass, class)

Example:

```
import csf.*

public class Server extends CommAwareObject{
}
```

This class will be created in file Server.java

2. Create a method

- need to find out if it is public or private (default to public)
- find the class file
- find the class declaration within the file
- ensure that the method does not already exist
- add the method, including the appropriate return type if given
- open the file up in an editor right away, centered on the new method to allow users to embellish the method.

Protocol:

CreateMethod(package, class, method_name, parameters, return_type)

Example:

```
public class MyCOA extends CommAwareObject{
        public boolean connectToServer(){
        }
}
```

3. Create a property

- find the class file
- find the class declaration within the file
- add the name of the instance variable at the end of the variable declarations
- allow the user to move the declaration anywhere they want within the list of declarations
- generate get and set methods for that variable

Protocol:

CreateProperty(package, class, property_name, property_type)

Example:

```
public class Server extends CommAwareObject{
        Inbox in;

        public void setInbox(Inbox in1){
                in = in1;
        }

        public Inbox getInbox(){
                return in;
        }
}
```

4. Override a method

- find the superclass file
- find the class declaration within the superclass file and ensure that the method exists and that the parameter list is the same as passed to the 'Java Master' tool through the protocol (look for any inconsistencies between what is in the file and what was expected by the hook).
- Find the class file

- Find the class declaration within the file
- Ensure that the overriden method does not already exist
- generate the overriden method

Protocol:

OverrideMethod(package, superclass, class, method_name, parameters, return_type)

Example:

```
public class MyData extends Data
{
        public void readData( DataInput in ) throws IOException{
        }
}
```

5. Specialize a method

- find the superclass file
- As in overriding a method, find the class declaration within the superclass file and ensure that the method to specialize exists and that it matches the parameters passed to the tool through the protocol.
- Find the class file
- Find the class declaration within the file
- Ensure that the method does not already exist
- generate the method, including a call to the superclass method.

Protocol:

SpecializeMethod(package, superclass, class, method_name, parameters, return_type)

Example:

```
public class MH1{
        CommAwareObject owner;

        public void handleMessage(Message m, CommAwareObject coa){
                super.handleMessage(m,coa);
        }
}
```

6. Modify code

- find the class file to modify
- find the class declaration **within the** file
- find the method within the **class**
- search for the code within the method body to remove and replace it with the replacement code or with an empty string if there is no replacement code.

Protocol:

ModifyCode(package, class, method_name, method_parameters, original_code, replacement_code)

7. Method call

- find the class file
- find the class declaration within the file
- find the method body
- insert the method call at the end of the method
- bring up an editor that allows the user to move the method call where needed

Protocol:

MethodCall(package, class, method_name, method_parameters, called_method_name, called_method_parameters, called_method_return_type)

Example:

```
public Server extends CommAwareObject{
        public Server(){
                // Create and register the message handler object
which
                // is invoked when the "test message" message is
received.
                registerHandler("test message", new MH2());
        }
}
```

8. Synchronize method calls

- find the class file
- find the class declaration within the file

- find the method body
- insert the one method call followed by the other (or ensure that they happen in that order).

Note: Synchronize will be left out of the initial version of the tool.

9. Adding code

- find the class file
- find the class declaration
- find the method body
- insert the code at the end.
- Bring up an editor that allows the user to move the code wherever they want within the method.

Protocol:

AddCode(package, class, method_name, method_parameters, code)

Example:

```
public class Server {
        public void startServer(){
        try{
                AppletServerMailServer ms = new
AppletServerMailServer("square-crk.cs.ualberta.ca",8199);
                }
                catch (Exception e){
                        System.out.println("Server cannot be
started.");
                }
        }
}
```

10. AddLocalVar

- find the class file
- find the class declaration
- find the method body
- check the parameters of the method to ensure the right one has been found
- insert the variable at the end of the method

Protocol:

AddLocalVar(package, class, method_name, parameters, varName, varType)

Example:

```
AddLocalVar(someClass, main, "String[]", IP, String)

Result:

class Class {
        static public void main (String[] argv) {
                ...

                String IP;
        }
}
```

11. Directives

Directives are notes directly to the developer using the hook and are not meant to be automated. The developer simply indicates whether they were done or not. There is no special handling for directives built into the current version of JavaMaster.

Description of Protocol Parameters

All of these parameters should be consistent across the entire protocol.

- *Package*: the Java package (equivalent to the directory) that a class file is in.
- *Superclass*: the parent class of the class being modified.
- *Class*: the class being modified.
- *Method_name*: the name of the method within Class that is being modified.
- *Parameters*: the list parameters of the method being altered, including any exceptions it throws (?).
- *Return_type*: if the method being modified is a function, the type or class of variable that it returns.
- *Called_method_name*, *called_method_parameters* and *called_method_return_type* are identical to *method_name*, *parameters* and *return_type* except that they are used to produce a method call.
- *Property_name*: the name of an instance variable being added to a class.
- *Property_type*: the type of the instance variable being added to a class.
- *Code*: code being added to a class
- *Original_code*: a list of lines of code to be modified or removed
- *Replacment_code*: a list of lines of code that will replace the original_code
- *objectName:* the name of the instance to be created.
- *className:* the name of the class of which an instance is being created.

- *c_dParameters:* the varible names to be used in construction/destruction of the instance.
- *varName*: the name of the varible to be added.
- *varType:* the type of the varialbe to be added.
- *Initialize:* the value (literal or variable) that the new variable will be assigned.