University of Alberta

# A Tool for Enacting Hooks on an O-O Framework

by

**Luyuan Liu** ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta

Spring 2002

0-612-69731-2

**University of Alberta**

**Library Release Form**

**Name of Author**: Luyuan Liu

**Title of Thesis**: A Tool for Enacting Hooks on an O-O Framework

**Degree**: Master of Science

**Year this Degree Granted**: 2002

Luyuan Liu

Luyuan Liu

85, 4936 Dalton Dr. NW
Calgary, Alberta
Canada, T3A 2E4

Date: Oct. 18, 2001

**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 **A Tool for Enacting Hooks on an O-O Framework** submitted by Luyuan Liu in partial fulfillment of the requirements for the degree of **Master of Science**.

[illegible]

Dr. Paul G Sorenson (Supervisor)

[illegible]

Dr. Jim Hoover (Internal Examiner)

[illegible]

Dr. Marek Reformat (External Examiner)

**Date**: Oct. 18, 2001

# Abstract

O-O frameworks are difficult to understand and use. *Hooks* [16] focus on how the framework is intended to be used, and provide guidance in how and where to perform the changes that will fulfill some requirements within the application being developed. This thesis presents a proof of concept for the usefulness of the *hooks* model approach. We developed a tool called **HookMaster** based on Rational Rose 98 and Visual Basic. **HookMaster** can understand *hook* descriptions and interactively guide the user through the enactment of *hooks*. The output of the tool includes *class diagrams* and *collaboration diagrams* which can be used as the base of further application development.

# Acknowledgements

First of all, I am deeply indebted to my parents, Shun Liu and Xiulian Qi. Where I am today is in no small part due to their love and support. Also, I would like to thank my wife, Guohui Jia, for her love and support.

Second, I would like to express my sincere thanks to Dr. Paul G Sorenson, my supervisor for his encouragement and assistance throughout the course of this research. His many suggestions have been invaluable.

Then, I would like to thank all the members of the SERL team at the University of Alberta.

Finally, to my brother and his wife, Lubo Liu and Xiaorong Jiang, for their support.

# Contents

**1 Introduction** **1**

1.1 Motivation . . . . . . . . . . 1

1.2 Rationale . . . . . . . . . . 3

1.3 Hook Model . . . . . . . . . . 5

1.4 Need for a Visual Tool . . . . . . . . . . 7

1.5 Chapter Map . . . . . . . . . . 9

**2 Background and Literature Review** **11**

2.1 Concepts and Properties of Frameworks . . . . . . . . . . 12

2.1.1 Users and Developers of Frameworks . . . . . . . . . . 12

2.1.2 Framework Concepts . . . . . . . . . . 12

2.1.3 Framework Categorization . . . . . . . . . . 14

2.1.4 Hot Spots and Hooks . . . . . . . . . . 15

2.1.5 Desirable Properties . . . . . . . . . . 16

2.2 The Design of Object-Oriented Frameworks . . . . . . . . . . 18

2.2.1 Domain Analysis . . . . . . . . . . 19

2.2.2 Design and Implementation . . . . . . . . . . 20

2.2.3 Composition and Inheritance . . . . . . . . . . 22

2.3 The Use of Object-Oriented Frameworks . . . . . . . . . . 23

2.3.1 Ways to Use a Framework . . . . . . . . . . 23

2.3.2 Learning to Use the Framework . . . . . . . . . . 24

2.4 Documentation . . . . . . . . . . 25

2.4.1 Hook Descriptions . . . . . . . . . . 25

2.4.2 Design . . . . . . . . . . . . . . . . . . . . . . . . . 30
2.5 Tool Support Object-Oriented Development . . . . . . . . . . 30
2.6 UML and Rational Rose 98 . . . . . . . . . . . . . . . . . . . 31
2.6.1 UML views . . . . . . . . . . . . . . . . . . . . . . . . 32
2.6.2 Rational Rose 98 . . . . . . . . . . . . . . . . . . . . . 34

**3 Requirements for Hooks Tool** **37**

3.1 Requirements for Application Developers . . . . . . . . . . . . 39
3.1.1 Hook Views . . . . . . . . . . . . . . . . . . . . . . . . 41
3.1.2 View Consistency . . . . . . . . . . . . . . . . . . . . . 44
3.1.3 Hook Interference . . . . . . . . . . . . . . . . . . . . . 44
3.1.4 Hook Books, Examples and Use Cases . . . . . . . . . 46
3.2 Maintainers and Developers . . . . . . . . . . . . . . . . . . . 48

**4 Architecture for Realizing HookMaster** **49**

4.1 Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.2 Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.2.1 Logical System Architecture . . . . . . . . . . . . . . . 50
4.2.2 Subsystem Descriptions . . . . . . . . . . . . . . . . . . 51
4.2.3 Class Diagram . . . . . . . . . . . . . . . . . . . . . . . 55
4.3 Use Cases . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
4.3.1 Use Cases Diagram . . . . . . . . . . . . . . . . . . . . 55
4.3.2 Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.3.3 Open Model . . . . . . . . . . . . . . . . . . . . . . . . 57
4.3.4 Replay . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
4.3.5 Enact . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
4.3.6 Save . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
4.3.7 SaveAsNewVersion . . . . . . . . . . . . . . . . . . . . 64
4.3.8 Exit . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
4.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

**5 Prototype Implementation of HookMaster** **69**

5.1 The Scope of Implementation . . . . . . . . . . . . . . . . . . 69

5.1.1 Technological Consideration for Prototype . . . . . . . 70

5.2 Implementation of the Prototype . . . . . . . . . . . . . . . . 71

5.2.1 Rational Rose 98 . . . . . . . . . . . . . . . . . . . . . . . . 72

5.2.2 User Interface . . . . . . . . . . . . . . . . . . . . . . . . . . 76

5.2.3 Parser . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

5.2.4 Hook Table . . . . . . . . . . . . . . . . . . . . . . . . . . 81

5.2.5 Monitor-Controller . . . . . . . . . . . . . . . . . . . . . 82

**6 Validation and Improvement of Prototype** **85**

6.1 Using HookMaster with CSF . . . . . . . . . . . . . . . . . . 86

6.1.1 The CSF Motifs . . . . . . . . . . . . . . . . . . . . . . . 86

6.1.2 Validation for Hook Enactment . . . . . . . . . . . . . 86

6.1.3 Limitations and Possible Improvement . . . . . . . . . 96

**7 Conclusions and Future Work** **98**

**Bibliography** **101**

**Appendix** **105**

**A Grammar for Hook Descriptions** **105**

**B Class Diagrams** **108**

**C A simple parser of the new element statement for creating the new subclass from the *hook* description grammar** **118**

# List of Figures

2.1 Application Developed from a Framework . . . . . . . . . . . 13

3.1 The CSF Main View . . . . . . . . . . . . . . . . . . . . . . . 40

3.2 CSF: Handle Message Hook View . . . . . . . . . . . . . . . . 42

3.3 Mutual Exclusion Subsets . . . . . . . . . . . . . . . . . . . . 45

4.1 Component Diagram of Architecture . . . . . . . . . . . . . . 51

4.2 Class Diagram for UI and Parser Subscriptions . . . . . . . . . 56

4.3 Use Cases Diagram . . . . . . . . . . . . . . . . . . . . . . . 57

4.4 System Setup . . . . . . . . . . . . . . . . . . . . . . . . . . 58

4.5 Open Model . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

4.6 Replay . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

4.7 Enact . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

4.8 Save . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

4.9 Save As A New Version . . . . . . . . . . . . . . . . . . . . . 66

4.10 Exit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

5.1 The Rose Extensibility Interface . . . . . . . . . . . . . . . . . 70

5.2 The Diagram Class Hierarchy . . . . . . . . . . . . . . . . . . 75

5.3 The UML Elements . . . . . . . . . . . . . . . . . . . . . . . 75

5.4 The class diagram for the user interface . . . . . . . . . . . . 76

5.5 The Implementation for Rose Interface . . . . . . . . . . . . . 77

5.6 Rational Rose Scenario . . . . . . . . . . . . . . . . . . . . . 78

5.7 The parsing diagram . . . . . . . . . . . . . . . . . . . . . . 80

5.8 The class diagram of the Hook Table . . . . . . . . . . . . . . 82

5.9 The class diagram of the Monitor . . . . . . . . . . . . . . . . . 83

6.1 The New CommAwareObject Hook View . . . . . . . . . . . . 88

6.2 The Handle Message Hook View . . . . . . . . . . . . . . . . . . 89

6.3 The New Outbox Hook View . . . . . . . . . . . . . . . . . . . . 93

6.4 The New Inbox Hook View . . . . . . . . . . . . . . . . . . . . . 94

6.5 The Send Message Hook View . . . . . . . . . . . . . . . . . . . 96

7.1 Entity-Relation Diagram for Framework Information . . . . . 100

B.1 The MainUI class . . . . . . . . . . . . . . . . . . . . . . . . . . 108

B.2 The FileManager class . . . . . . . . . . . . . . . . . . . . . . . 109

B.3 The NewClass class . . . . . . . . . . . . . . . . . . . . . . . . . 109

B.4 The NewProperty class . . . . . . . . . . . . . . . . . . . . . . . 109

B.5 The NewOperation class . . . . . . . . . . . . . . . . . . . . . . 110

B.6 The Caller class . . . . . . . . . . . . . . . . . . . . . . . . . . . 110

B.7 The Monitor class . . . . . . . . . . . . . . . . . . . . . . . . . . 110

B.8 The Controller class . . . . . . . . . . . . . . . . . . . . . . . . . 111

B.9 The cParser class . . . . . . . . . . . . . . . . . . . . . . . . . . 112

B.10 The cHookTable class . . . . . . . . . . . . . . . . . . . . . . . 113

B.11 The cHook class . . . . . . . . . . . . . . . . . . . . . . . . . . 114

B.12 The cClass class . . . . . . . . . . . . . . . . . . . . . . . . . . 115

B.13 The cOperation class . . . . . . . . . . . . . . . . . . . . . . . . 115

B.14 The cProperty class . . . . . . . . . . . . . . . . . . . . . . . . 115

B.15 The cParameter class . . . . . . . . . . . . . . . . . . . . . . . . 116

B.16 The cStatement class . . . . . . . . . . . . . . . . . . . . . . . . 116

B.17 The cStack class . . . . . . . . . . . . . . . . . . . . . . . . . . 117

# Chapter 1

# Introduction

## 1.1 Motivation

Software reuse is a potential means of improving software productivity and quality. Unfortunately, experience and research show that small pieces of software often cannot be reused outside of their original context. In contrast, Object-Oriented frameworks support large-scale software reuse, and provide the context in which that reuse takes place. Applications are built from frameworks by extending or customizing parts of the framework, while retaining the original design. An object-oriented application framework is the reusable design for a family of applications implemented through a set of abstract classes and their collaborations [1]. The design provides a means of organizing related components without placing the classes or components in a generic library. Each component is reused in the context for which it was created and does not have to be modified. Frameworks solve larger-grained problems than individual function-specific components, thus making the effort of finding and reusing them much more cost effective than for small components. If a family of similar applications are developed from a single framework, then maintainers will only have to learn one standard design and will be able to maintain the whole product line more easily.

Although much of the research in this thesis applies to frameworks in general, the analysis is on Object-Oriented frameworks. When we use the term *frameworks*, we mean Object-Oriented frameworks unless we explicitly state otherwise.

Frameworks currently exist for graphical editors (HotDraw [20], UniDraw [2]), user interfaces (ET++) [3], manufacturing systems (OSEAFA [25]), client-server communications [4], operating systems (Choices [10]), and network protocol software [19].

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. Due to the potentially large size and complexity of frameworks, the ability to quickly understand and apply a framework is a critical issue.

A *hook* is introduced as a means of easing the understanding and use of a framework [16]. *Hooks* focus on how the framework is intended to be used, and provide strong guidance in how and where to perform the changes that will fulfill some requirements within the application being developed. They provide an alternative, supplementary view to the design of the framework. A *hook* may involve something as simple as inheriting from an existing class in the framework, or as complex as adapting the interaction of a large number of classes within the framework. The framework builder, who is the most knowledgeable about the framework, defines the *hooks* and through them passes on his/her knowledge to the application developer. In this way, the framework builder can tell the application developer what needs to be completed or extended in the framework, or what choices need to be considered about parts of the framework in order to develop an application using the framework. The *hooks* help to ensure that changes or extensions integrate smoothly into the design and implementation of the framework by imposing some constraints concerning how the framework can be adapted.

In this thesis, we help to develop a proof of concept for the usefulness of the *hooks* model approach to the framework use, and demonstrate that application developed using *hooks* for an O-O framework can be well supported with a visual tool.

## 1.2 Rationale

A framework can be quite complicated and difficult to understand. Properly documenting the framework is important in order to ease its understanding and use. The ultimate goal of design for reuse is reusable software, which requires the design of general, extensible software component [22]. Application design in an O-O framework should be oriented towards design-level reuse, in order to make best use of the framework. This poses particular challenges because their descriptions at different levels of abstraction are required. Furthermore, such documentation should address the needs of developers with varying levels of experience with the framework [20].

Much work on framework documentation has focused on the design and architecture of the framework. Three approaches that have been used are: providing several views of the design, using design patterns, and using exemplars.

Campbell and Islam [10] propose a six part approach to documenting the framework design. 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. All of these diagrams concentrate on the design of the framework and not on the purpose of the framework or how to use it.

*Design patterns* [20] are a popular means of describing frameworks. Beck and Johnson used them to help show how the architecture of the HotDraw framework is derived [1]. The OSEAFA framework was also described using design patterns [25]. Work was done to allow the interactive visualization of design patterns within a framework [23]. Design patterns can help to show the decisions that were made regarding the design of the framework. 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 application developer can see that a design pattern was used and immediately understand some of the advantages and limitations of the design. Although design patterns are an excellent means of documenting the design, they are general and do not explicitly describe the purpose of a framework, nor how it should be used.

*Exemplars* [18] 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 their interactions. The interactions between classes can be explored through the exemplar using a special tool to provide a deeper understanding of the framework. Class hierarchies can also be browsed by the tool to make choosing the desired classes easier. While this approach allows for the understanding of the design of a framework, it is not as well suited for providing either the purpose of the framework or how the framework is meant to be used.

Less work has focused on the purpose and intended use of frameworks. Two such approaches are cookbooks and patterns. The *cookbook* in [21] consists of a general description of the purpose of the Smalltalk Model-ViewController (MVC) framework, 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. A different type of *cookbook* found in [24] focuses 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 set of steps to follow along with some examples for solving the problem. However, the steps are narrative descriptions and are not well-structured or uniform.

Johnson's *patterns* [20] 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, gives general narrative advice and examples about ways to solve the problem, summarizes the solution, and refers 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.

In [22] the idea of design patterns is combined with Johnson's *patterns*, which they rename *motifs*, to provide a more complete description of a framework [22]. In their strategy, *motifs* point to design patterns, contracts 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 are no more structured than *patterns*.

*Hooks* focus on the intended use of the framework much like cookbooks or motifs but do not focus on the design like design patterns or exemplars. *Hooks* provide an alternative view to design documentation. Rather than presenting the design or even the reasons why a particular design was chosen, hooks show how and where a design can be changed. They present knowledge about the usage of the framework.

## 1.3 Hook Model

*Hooks* are more structured and uniform than *motifs* or *cookbooks* and help the framework builder to be more precise about how the framework is intended to be used [16]. Further, the characterization for *hooks* introduced in this thesis shows many of the ways in which changes can be made to a framework and how these changes can be supported.

*Hooks* provide solutions to focused problems. For more complex problems. groups of *hooks* can be provided with each focusing upon a smaller problem within the larger. more complex problem. Each hook details the changes to the design that are required. the constraints that must be followed and any effects upon the framework that the hook imposes, such as configuration constraints. Only the information needed to solve the problem is provided within the hook. Developers are then able to quickly understand and use the *hook*.

*Hooks* are meant to be used for developing applications from a framework. not developing new frameworks from old ones. Selecting options. filling in parameters or extending the framework for a particular application are all *hooks*. However. modifying or refactoring an existing framework is beyond the scope of the *hook* model and are not considered to be valid *hooks*.

Each valid hook is written 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 application developers by showing all aspects that should be considered, such as how using a *hook* affects the rest of the framework (constraints). The format helps to organize the information, prompts for the required information and makes the description more precise and uniform. All these aspects aid in the analysis of *hooks* and the provision of tool support for them. Each *hook* description consists of the following parts:

- **Name**: a unique name, within the context of the framework, given to each hook.
- **Requirement**: the problem the hook is intended to help solve.
- **Type**: an ordered pair consisting of the method of adaption used and the amount of support provided for the problem within the framework.

- **Area**: the parts of the framework that are affected by the hook.
- **Uses**: the other hooks required to use this hook. The use of a single hook may not be enough to completely fulfill a requirement that has several aspects to it, so this section states the other hooks that are needed to help fulfill the requirement.
- **Participants**: the components that participate in the hook. These are both existing and new components.
- **Preconditions**: limits imposed before enacting the changes specified in the hook.
- **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.
- **Postconditions**: limits imposed after applying the changes specified in the hook, such as configuration constraints.
- **Comments**: any additional description needed.

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

## 1.4 Need for a Visual Tool

Since *hooks* are important points of access to the framework, their evaluation should be among the high priority quality activities.

To study how people learn and use frameworks, and evaluate the *hook* notation, an experiment was introduced in the CMPUT 401 course which required the use of the CSF framework[26] in projects. The students were given the framework and many of its *hooks*. The experiment showed that the students found the notation quite difficult to understand and use. They rated the *hooks* as less useful than either the example code given or the introduction tutorial lessons that were provided. Nevertheless, *hooks* are targeted at the right problem (understanding interactions between the framework and application) but need to be tied more closely to examples and the barriers to their use need to be lowered by clearer and better explained notation that is enhanced with tool supports.

In the thesis, we develop a tool to assist the application developer in interacting with a framework using *hooks*. The tool should provide two primary ways to develop applications. First, developers use the tool to develop applications from the framework without changing the framework. Second, framework maintainers will use the tool to evolve or modify the framework and/or the *hooks*. The tool developed in the thesis primarily focuses on the first aspect.

As mentioned earlier, *cookbooks* can be represented as tutorials that describe the basis of the framework and examples of its use [21]. Similarly, framework *patterns* [20] document common problems and solutions within examples through a general narrative. The information contained in *cookbooks* and framework *patterns* is valuable but can not be easily interpreted by an automated tool.

A tool for the exploration and use of framework through *exemplars* is described in [18]. A concrete instance is provided for all abstract classes in the framework. These can then be executed through the tool to learn the behavior of the framework. A tool also exists for examining or discovering the *design patterns* [17] used in a framework [23]. These tools allow exploration of the design of the framework. In [23] class and object-oriented views of the structure and behavior of frameworks are provided, with information accurate enough to enable developers to reuse and maintainers to support undocumented parts of frameworks. But these tools do not explicitly describe how the framework can be used.

Graphical user interface builders represent one of the most successful applications of framework technology. GUI builders are built on the top of specific toolkits and provide an interactive editor for GUI development. They allow the developer to rapidly prototype the presentation layer and experiment with alternative designs. GUI builders allow users to visually position components on a screen and to adjust a list of parameters provided with the component. Typically, the user can also define or fill in methods that can respond to events within the system. However, GUI builders just focus on a single visual framework.

Using the same basic ideas that exist in GUI builders, the *hook* tool is based on the notation of *hooks*, which describes how the framework is intended to be used and shows where changes can be made. The requirements for the tool are discussed in Chapter 3. The tool creates *hook views* to represent *hook* notation by extending UML language to include *hooks* [6]. Also, the change and constraint specifications of *hooks* are defined by a context-free grammar (see Appendix A). The *hook* tool aids users by semi-automatically enacting the changes within *hooks*. The tool handles propagation of changes between *hook views*. In addition, the tool is flexible enough to provide support for many different frameworks, which is not currently done in existing tools.

## 1.5 Chapter Map

In the thesis, we outline the key requirements and design of the tool. Chapter 2 describes an overview of the design and use of frameworks, followed by a more detailed discussion of the hook model. Chapter 3 describes the requirements for the tool. Chapter 4 discusses the design rationale, and describes the architecture and use cases of the tool. Chapter 5 focuses on issues related to the prototype implementation of the tool. Then, we will demonstrate an example through using this tool, and describe the experiment in Chapter 6. Finally, contributions and future directions are given in Chapter 7.

# Chapter 2

# Background and Literature Review

An object-oriented framework is the reusable design of a system or subsystem implemented through a collection of concrete and abstract classes and their collaborations [1]. The concrete classes provide the reusable components, while the design provides the context in which they are used. A framework is more than a collection of reusable components. However, it must provide a generic solution to a set of similar problems within an application domain. The framework itself is incomplete with respect to a particular application but provides places at which users can add their own components specific to a particular application. The following sections will discuss the design of use of object-oriented frameworks.

## 2.1 Concepts and Properties of Frameworks

We begin our discussion of framework design, by examining some concepts and terms associated with frameworks, starting with the roles involved in framework technology and a more in depth look at the parts of a framework.

### 2.1.1 Users and Developers of Frameworks

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

- Framework Designer (or Framework Developer), develops the original framework and is typically an expert in software architecture and the application domain.
- Framework User (also called Application Developer or Framework Client), uses the framework to develop application.
- Framework Maintainer, refines and redevelops the framework to fit evolving requirements.

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

### 2.1.2 Framework Concepts

Several different parts can be identified within an application developed from a framework as shown graphically in Figure 2.1. Applications are developed from frameworks by filling in missing pieces and customizing the framework in the appropriate areas.

The parts of a framework are:

- Framework Core: The core of the framework generally consists of abstract classes that define the generic structure and behavior of the framework. It forms the basis for the application developed from the framework. However. the framework can also contain concrete classes that are meant to be used in all applications built from the framework.
- Framework Library: Extensions to the framework core organized in the form of a component library contain concrete components that can be used with little or no modification in the development of applications developed from the framework.
- Application Extensions: Application specific extensions made to the framework (also called an ensemble) are additional software created by the framework user in order to produce an application.
- Application: The application consists of the framework core. the used framework library extensions. and any application specific extensions needed.
- Unused Library classes: Typically. not all of the classes within a framework library are needed in an application that can be developed from the framework. Those not needed are called the unused library classes.

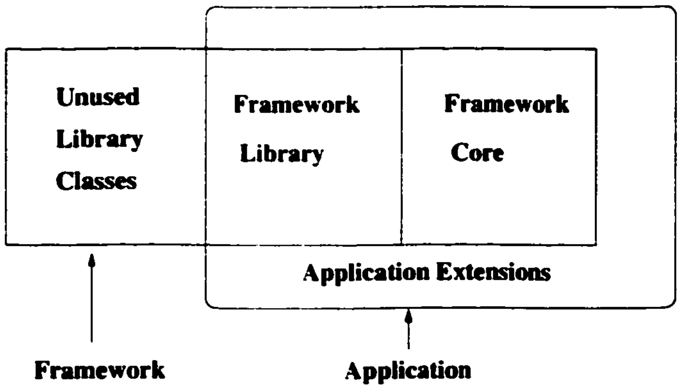


Figure 2.1: Application Developed from a Framework

### 2.1.3 Framework Categorization

Several different means of classifying frameworks have been proposed. Here we present three relatively orthogonal views of frameworks. A framework can be categorized by its scope, its primary mechanism for adaptation and the mechanism by which it is used.

#### Scope

The scope of the framework describes how broad of a domain the framework is applicable too. Adair defines three framework scopes [25].

- Application frameworks contain horizontal functionality that can be applied across domains. They incorporate expertise common to a wide variety of problems. These frameworks are usable in more than one domain. Graphical user interface frameworks are a typical example of an application framework and are included in most development packages.
- Domain frameworks contain vertical functionality for a particular domain. They capture expertise that is useful for a particular problem domain. Examples exist in the domains of operating systems [10], manufacturing systems [25], client-server communications [4].
- 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.

#### Customization

The means of customizing is another way in which frameworks can be categorized. The reference [11] defines two types of frameworks, white box and black box.

- White box frameworks, also called architecture driven frameworks, 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 framework knowledge to use.

- Black box frameworks, also called data-driven frameworks, 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 [11] argues that frameworks tend to mature black box frameworks.

We are interested in supporting both forms of customization.

#### Interaction

In [12], frameworks are differentiated based on how they interact with the application extensions, rather than their scope or how they are customized.

- Called frameworks correspond to code libraries (such as the Booch's libraries [13]). Applications use the framework by calling functions or methods within the library.

- Calling frameworks incorporate the control loop within the framework itself. Applications provide the customized methods or components which are called by the framework. Here we will be primarily focusing on calling frameworks.

### 2.1.4 Hot Spots and Hooks

In the *hook* model[16], application extensions are connected to a framework through *hooks*. *Hooks* are ways of extending or adapting a framework at particular points in the framework to provide application specific functionality. They are the means by which frameworks provide the flexibility to build many different applications within a domain. We will discuss *hooks* details in Section 2.4.

*Hot spots* [15] also called *hinges*, are the general areas of variability within a framework where placing hooks is beneficial. A hot spot may have many hooks within it. The area of Tools within HotDraw is a hot spot because different application will use different tools. A Data Flow Diagram application will have tools for creating and manipulating the DFD that are standard for iconic interfaces, whereas a PERT chart application, because of its strict temporal ordering constraints, will likely have different creation and manipulation tools. There are a number of hooks within the Tool hot spot which define the various ways in which new tool can be defined.

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

### 2.1.5 Desirable Properties

Frameworks are meant to be reused to develop applications, and so reusability is very important. Software reusability means that ideas and code are developed once, and then used to solve several software problems, thus enhancing productivity, reliability and quality. A good framework has several properties such as ease of use, extensibility, flexibility, and completeness which can help to make it more reusable.

## Ease of Use

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

## Extensibility

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

## Flexibility

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

#### Completeness

Although frameworks typically have some degree of incompleteness, because they can not cover all possible variations within a domain, relative completeness is a desirable property. Default implementations can be provided for the abstractions within the framework and thereby avoiding reimplemention for every application. In addition, application developers can run the framework to gain a better understanding of how it works. The framework library can provide the implementations of common operations, which the developer can choose, making the framework easier to use as well as more complete.

#### Consistency

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

## 2.2 The Design of Object-Oriented Frameworks

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

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

Object-oriented technology is a natural fit for frameworks. Just as a subclass is a specialization of a parent class, an application can be thought of as a specialization of a more general framework. One of the ways to use a framework is to specialize the generic classes that are provided in the framework into application specific concrete classes.

### 2.2.1 Domain Analysis

As with any type of software development, the first stage in framework design is the analysis of the problem domain.

One of the key decisions that needs to be made when building a framework is domain coverage. Does the framework apply to a large domain, a narrow part of a domain, or even several domains? There are benefits and drawbacks to frameworks that cover a large domain or a large part of a domain versus small frameworks which cover a narrow part of a domain. A framework with broader coverage will be reusable in more situations, and thus be flexible, but may be unwieldly and difficult to maintain because of its size. Building a widely applicable framework is significant undertaking, requiring a lot of resources are experience in developing several specific applications from that domain.

After the domain of the framework is determined, analyzing the domain of the framework helps to determine the key abstractions that form the core of the framework. Examining existing applications within the domain of the framework is a useful means of identifying the abstractions. In order to gain domain expertise, a framework designer may also want to build an application within the domain if the designer is not already an expert in the domain.

Another analysis approach involves the development of scenarios for the operation of the framework and reviewing them with potential users of the framework [12]. Scenarios help to define the requirements of the framework without committing developers to any design decisions [7]. The scenarios can be abstracted into use cases [7] to help identify the key abstractions and interaction patterns the framework needs to provide.

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

## 2.2.2 Design and Implementation

The design determines the structures for the abstractions, frozen spots and hot spots. The design and implementation of the framework are often intertwined. Abstractions can be difficult to design properly the first time and parts of a framework may have to be redesigned and reimplemented as the abstractions become better understood.

However, some general guidelines have been identified. The reference [14] suggests:

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

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

Some additional general design advice proposed is to:

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

At this stage, the specific hooks for each hot spot must also be designed and specified. The hooks show specific ways in which the framework can be adapted to an application, and so are an important part of the framework. Hooks can be described in a formal manner using a context-free grammar(see Appendix A).

Often, trade-offs must be considered when identifying the *hot spots* and designing the *hooks* in general. Frameworks can not be arbitrarily flexible in all directions. Some of the required flexibility can be determined by examining existing applications. Often the framework designer has to rely on experience and intuition to make these trade-offs. Subsequent testing may require changes in the structure of the hot spots. Further trade-offs occur between flexibility and ease of use. The most flexible framework has very little actually defined and so requires a great deal of work on the part of the framework user.

After testing, the abstractions of the framework will often need to be extended or refined. Building a framework is a highly iterative process, so many cycles through these steps will be performed before the final framework is produced.

### 2.2.3 Composition and Inheritance

Inheritance and composition are the two main ways for providing hooks into the framework. Composition is often recommanded over inheritance as it tends to be easier for the framework user to use but each has strengths and weaknesses. The type of customization used in each case depends upon the requirements of the framework [13].

Composition involves the use of callbacks or parameterized types. The class of the framework to be adapted will have a parameter to be filled in by the application developer which provides some required functionality. Since the customization is done by filling in parameters, the framework user does not need an in-depth knowledge of how the particular component operates.

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

Composition is generally used when the interfaces and uses of the framework are fairly well defined, whereas inheritance provides flexibility in cases where the full range of functionality can not be anticipated. Composition forces conformance to a specific interface and functionality which can not be easily added to or changed. It has been proposed that frameworks start out as white box frameworks that rely on inheritance. As the domain becomes better understood and more concrete support classes are developed, the framework evolves to use more composition and becomes a black box framework.

## 2.3 The Use of Object-Oriented Frameworks

In this section we discuss techniques and concepts related to using frameworks. Many users will use a framework as it was meant to be used, but others will want to use the framework in new, non-standard ways. Still others will want to evolve the framework to incorporate new capabilities.

### 2.3.1 Ways to Use a Framework

There are a number of different ways in which to use a framework. Each of them require a different amount of knowledge about the framework and a different level of skill in using it. In [14], three main ways in which frameworks can be used are defined:

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

Most of the work involving *hooks* focuses on the second type of framework. It is very rare that you can use the framework As-Is. Customizing frameworks is a tricky and risky task that makes *hooks* identification difficult to achieve.

### 2.3.2 Learning to Use the Framework

One of the first stages involved in developing applications from frameworks is analyzing the requirements of the application. A framework can be chosen by viewing the documentation, or through other user's experiences. If the users are already familiar with the framework, the requirements can be cast immediately in a form that is compatible with the framework.

One of the first difficulties faced by users of any framework is learning how to use it. Learning how to use a framework given only the code and interface descriptions is a daunting task that makes framework use unattractive. A framework should be easier to extend as the basis for application development than building a new application without the framework. Some means of lowering the learning curve are needed. Some approaches are:

- Framework developers as users: when the framework developers are also the ones developing applications and maintaining the framework, they are already experts on the framework and require little, if any, time to learn it.

- 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.

- Tool support: a good tool can make a framework much easier to use. With it, regular users generally do not have to learn all the details about the framework since the tool will dictate how and where adaptions can take place. Unfortunately, very few such tools exist and the development of such a tool is a primary focus of this thesis.

- Documentation: good documentation can aid users in learning the framework. It can be used to not only capture the design details and design rationale of the framework, but also to help users to learn the framework.

## 2.4 Documentation

Framework documentation can aid in all aspects of framework use. Since the framework developers will not always be available, documentation becomes the means through which users can learn about the framework and refer to information concerning the framework. A framework will typically have few developers and many users, so it is important that the documentation be made understandable and up to date.

Beyond a description of the general purpose of the framework, regular users are interested in how to use the framework. Documentation describing the intended uses of the framework helps users to learn the framework, and develop applications. Documentation of intended use should identify the problems that the framework solves or that users face when using the framework, how to think about the problem in order to understand how to solve it, and the actual solution. The documentation should capture the framework developer's experience and knowledge of how the framework can be used. Several methods have been proposed for describing the intended use: *motifs/patterns* [23], *cookbooks* [21] and *hooks* [16]. These were outlined in Chapter 1. We focus on *hooks* in this chapter.

### 2.4.1 Hook Descriptions

The approach described in this section is primary due to the work of Garry Froehlich. More details of this work can be found in [16].

*Hook* descriptions provide a semiformal template for describing the intended use at a detailed level. The template helps to prompt the developer for all the required information and the semi-formal language makes the descriptions more precise.

The sections of the template detail different aspects of the *hook*, such as the components that take part in the *hook*(participants) or the steps that should be followed to use the *hook*(changes). The sections serve as a guide to the people writing the hooks by showing the aspects that should be considered about the *hook*, such as how using it affects the rest of the framework(constraints). The format helps to organize the information and make the description more precise and uniform. This aids in the analysis of hooks and the provision of tool support. An example of a *hook* taken from CSF[26] is shown below:

**Name**: New FileManager
**Requirement**: An application needs a mechanism to write data to a file.
**Type**: Enabling, Pattern
**Area**: Data, Persistence
**Uses**: Register FileManager
**Participants**: FileManager(provided). NewFM
**PreConditions**: None
**Changes**:

    new subclass NewFM of FileManager
    NewFM.write(String classname. Criteria key, Data dataObject) overrides
    FileManager.write(String classname. Criteria key, Data dataObject)
    NewFM.read(String classname, Criteria key) overrides
    FileManager.read(String classname. Criteria key) return Data
    Register FileManager[FM = NewFM]

**PostConditions**: None
**Comments**: None

The New FileManager *hook* describes how to create new file manager which are responsible for read/write data within CSF. This *hook* uses functionality provided by the framework by filling in parameters so its type is an enabling pattern. NewFM is defined as a subclass of the existing class FileManager. In this class, the operations of read and write override the corresponding operations of superclass FileManger, and send it class name of the Data object and a key under which to wrtie/read it along with the actual Data object. Each FileManager can handle any number of classes, and must be registered. Registering involves which classes a particular FileManager is responsible for. In order to register it, the Register FileManager hook is invoked by the New FileManager hook. Finally, the postcondidtion method is provided to return constraints defined so that they can be evaluated when needed. In this case, the postcondition is none.

*Hooks* are characterized along two axes: the method of adaption used and the level of support provided. The method of adaption describes the basic mechanism used to extend or adapt the framework. The level of support indicates how the change is supported within the framework, such as using provided components or requiring developers to produce their own components.

## Method of Adaption

There are several ways that a developer can adapt a framework and each *hook* uses at least one of these methods. These include enabling, disabling, replacing and augmenting.

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

Disabling a feature may be required if the default implementation of the framework has some unwanted properties. This is different than simply not choosing to enable a feature. Disabling a feature may be done through configuration parameters, or by actual modification of framework code. The *hook* description shows how to do the removal and it also shows the effects of the removal.

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

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

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

## Level of Support

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

The option level provides the most support, and is generally the easiest for the application developer to use. A number of pre-built components are provided within the framework and the developer simply chooses one without requiring extensive knowledge about the framework. This is the black-box approach to frameworks [11]. Most often, components are chosen to enable features within the framework or to replace default components. If the solutions are alternatives, the *hook* is a single option *hook*. If several alternatives can be used at once, the *hook* is a multi-option *hook*.

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

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

### 2.4.2 Design

Documentation of the design is most useful to advanced users. maintainers and framework developers. since they need to know the details of the framework. Regular users may also need to understand some of the design of a framework. There are some normal approaches for documenting the design of the framework. such as Booch's method[5] and the Object Modeling Technique[5]. however. they are often not sufficient for a full description. The problem with traditional notations is that the collaborative relationships between the core classes of the framework are not made explicit. UML [7] is the unified effort of Booch's method and OMT. It provides a set of standard diagram notations to present the design of the framework. in particular, it proposes dynamic design to help understanding behaviors of the framework.

## 2.5 Tool Support Object-Oriented Development

In order to aid application developers, support should be provided for exploring the documentation of the design and use of the framework. A good tool can make a framework much easier to use. 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[20]. It allows users to build new tools simply by filling in the appropriate parameters and then automatically integrates the new tool into an application. More complex tools such as the one provided for OSEFA [25] 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. Users must learn multiple tools and do integration between frameworks by hand. but a tool based on *hooks* can be flexible enough to support many different frameworks.

The notation 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. The hook tool can aid users by extending the UML language to include hooks. Also, the structured description of those changes within hooks can be enacted interactively with the user of the tool. We will discuss the requirements of the tool in next chapter.

## 2.6 UML and Rational Rose 98

The Unified Modeling Language (UML) is a general-purpose visual modeling language that is designed to specify. visualize. construct and document the artifacts of a software system [5][6]. It fuses the concepts of Booch[5], OMT[5], and OOSE[7]. The result is a single. common, and widely used modeling language. It captures decisions and understanding about systems that must be constructed. It is used to understand, design, browser, configure, maintain, and control information about such systems. It is intended for use with all development methods, lifecycle stages. application domains, and media. The modeling language is intended to unify past experience about modeling techniques and to incorporate current software best practices into a standard approach. UML includes semantic concepts, notation and guidelines and has static, dynamic, environmental. and organizational parts. It is intended to be supported by interactive visual modeling tools that have code generators and report writers. The UML specification does not define a standard process but is intended to be useful with an iterative development process. It is intended to be used with most existing object-oriented development processes.

The UML captures information about the static structure and dynamic behavior of a system. A system is modeled as a collection of discrete objects that interact to perform work that benefits an outside user. The static structure defines the kinds of objects important to a system and its implementation, as well as the relationships among the objects. The dynamic behavior defines the history of objects over time and the communications among objects to accomplish goals. Modeling a system from several separate but related viewpoints permits it to be understood for different purposes.

## 2.6.1 UML views

UML divides the various concepts and constructs into several views [7]. A view is simply a subset of UML modeling constructs that represents one aspect of a system. One or two kinds of diagrams provide a visual notation for the concepts in each view.

### Static View

The static view models concepts in the application domain, as well as internal concepts invented as part of the implementation of an application. This view is static because it does not describe the time-dependent behavior of the system, which is described in other views. The main constituents of the static view are classes and their relationships: association, generalization, and various kinds of dependency, such as realization and usage. A class is the description of a concept from the application domain or the application solution. Classes are the center around which the class view is organized; other elements are owned by or attached to classes. The static view is displayed in *class diagrams*, so called because their main focus is the description of classes. Classes can be described at various levels of precision and concreteness.

### Use Case View

The use case view models the functionality of the system as perceived by outside users. called *actors*. A use case is a coherent unit of functionality expressed as a transaction among actors and the system. The purpose of the use case view is to list the actors and use cases and show which actors participate in each use case. Use cases can also be described at various levels of detail. They can be factored and described in terms of other. simpler use cases. A use case is implemented as a collaboration in the interaction view.

### Interaction View

The interaction view describes sequences of message exchanges among roles that implement the behavior of a system. A classifier *role* is the distinguished from other objects of the same class. This view provides a holistic view of behavior in a system, that is, it shows the flow of control across many objects. The interaction view is displayed in two diagrams focused on different aspects: *sequence diagrams* and *collaboration diagrams.*

A *sequence diagram* shows a set of messages arranged in time sequence. Each classifier *role* is shown as a lifeline. A *sequence diagram* can show a scenario, that is , an individual history of a transaction. One use of *sequence diagram* is to show the behavior sequence of a use case.

A *collaboration diagram* models the objects and links that show their interaction. The objects and links are meaningful only in the context provided by the interaction. A classifier *role* describes an object and an association *role* describes a link within a collaboration. A *collaboration diagram* shows the *roles* in the interaction as a geometric arrangement. One use of a *collaboration diagram* is to show the implementation of an operation. The collaboration shows the parameters and local variables of the operation, as well as more permanent associations.

Both *sequence diagrams* and *collaboration diagrams* show interactions, but they emphasize different aspects. A *sequence diagrams* shows time sequence as a geometric dimension, but the relationships among *roles* are implicit. A *collaboration diagram* shows the relationships among *roles* geometrically and relates messages to the relationships, but time sequence is less clear because it is described by the sequence numbers.

**Extensibility Constructs**

UML provides three main extensibility constructs: constraints, stereotypes, and tagged values [6]. A constraint is a textual statement of a semantic relationship expressed in some formal language or in natural language. A stereotype is a new kind of model element devised by the modeler and based on an existing kind of model element. A tagged value is a named piece of information attached to any model element. These constructs permit many kinds of extensions to UML without requiring changes to the basic UML metamodel itself. They may be used to create tailored versions of the UML for an application area.

### 2.6.2 Rational Rose 98

Rational Rose 98 is a graphical software modeling tool which supports UML modeling [8]. It provides complete component development capabilities through integration with integrated development environments such as Visual Basic, Java, and C++. Rose 98 adds features to enable and simplify the assembling of components into complex applications or larger components. With Rose 98, developers can see all aspects of the component model. Rational Rose 98 has several features as follows:

### Multi-Language Development and Other Add-Ins

Rose can host multiple, independent, active add-in in a single session, allowing developers to build components in mixed languages, such as Java, Visual Basic, C++, and Ada. Also developers can manage add-ins through Rose's Add-In manager. This feature allows developers to quickly and accurately customize their Rose environment depending on the development needs.

### Round-trip Engineering

Rational Rose 98 supports true round-trip engineering, which is the capability of forward engineering Rose model components into source, allowing the generated source to be modified, and then reverse engineering the modified code back into the Rose model. It allows developers to move easily from analysis to design to implementation and back to analysis again, and thus supports all phases of a project's lifecycle. Rose's support for round-trip engineering ensures that the iterative development cycle is controlled and productive.

### Stereotypes

Rational Rose 98 provides a way for developers to extend the graphical notation of UML to support special cases as notations in their model. Rose 98 supports stereotypes, thereby allowing developers to create custom icons that represent functions and components not currently supported in UML notation or special cases required to fully define a set of applications. Using stereotypes, it is possible to accomplish all these, yet maintain UML guidelines and checking.

### Rose Extensibility Interface(REI)

REI enables Rose 98 to better integrate with tools and programs from third-party vendors and provides extensive capabilities that allow developers to easily access and extend Rose's capabilities. It provides three mechanisms to exchange information with other tools: Custom RoseScripts that are Visual Basic compatible. OLE and ODBC drivers for database access. The REI enables direct access to Rose model information, enabling other tools to read. write, and modify Rose models. extend Rose capabilities. and customize icons using stereotypes. OLE is a core capability of the REI. The OLE integration simplifies the information interchange between Rose and other OLE-enabled applications. By exposing the entire Rose model in Rose 98 via OLE. Rose can easily share information with other OLE-enabled applications.

# Chapter 3

# Requirements for Hooks Tool

An O-O framework is the reusable design of a system or subsystem implemented through a collection of concrete and abstract classes and their collaborations. The concrete classes provide the reusable components, while the design provides the context in which they are used. A framework is more than a collection of reusable components. It provides a generic solution to a set of similar problems within an application domain. The framework itself is incomplete and provides places at which users can add their own components specific to a particular application.

Because frameworks are reusable designs, not just code, they are more abstract than most software, and consequently documenting them can be more difficult. Frameworks are designed by experts in a particular domain and then used by non-experts. The principal audience of framework documentation is someone who wants to use the framework to solve a given problem, not someone building a software catheral. Consequently, the documentation for a framework must meet several requirements. It must describe [20]

- the purpose of the framework

- how to use the framework

- the detailed design of the framework.

A set of *hooks* can satisfy the first two purposes and can assist in the third. *Hooks* focus on how the framework is intended to be used, and provide an alternative and supplementary view to the design of the framework. *Hooks* are provided to guide the application developer in how to use a framework. Each hook captures the relevant knowledge of some potential use of a framework in a form that provides guidance to application developers. By describing the hooks, implicit knowledge about how to use the framework is made explicit and open to study, refinement or reuse.

A good tool that supports the expression and enactment of *hooks* 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 assist in directing how and where adaptions can take place. The tool performs the tedious tasks of integrating components into the framework, leaving users to focus on the specific requirements and design issues for an application.

A main goal of this thesis is to develop the prototype of a visual tool for developing applications from frameworks using *hooks*. 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 core. Second, framework maintainers will use the tool to evolve or modify the framework itself.

In order to aid application developers, support for exploring the documentation of the design and use of the framework should be provided, along with support for actually adding application specific elements (e.g. classes, methods, variables). *Hooks* provide the basis for documenting the changes the framework builder intended to be made to the framework. The structured description of those changes within *hooks* can be enacted interactively by the user of the tool. Aid to framework maintainers is provided by allowing extensions to the framework and the hooks themselves to be made and incorporated back into the framework. This tool must be flexible enough to support many different frameworks, thereby enabling framework builders to adapt the tool to their framework instead of going through the expense of developing a custom tool for each framework. The flexibility to add new components, add new hooks, modify hooks or modify parts of the framework is required by the framework maintainer.

## 3.1 Requirements for Application Developers

When application developers start development using a framework, the tool first must create an instance of the framework as a basis for their application development. The tool incorporates the assumption that the original design or implementation of the framework should not be modified. This assumption preserves the benefits of maintaining a common code base between a family of applications; otherwise, the application is no longer an extension of the framework, but an evolution of its code base.

The tool distinguishes between two types of classes:

- *Framework classes* are classes provided with the framework and these should only be modified in prescribed manner.
- *Application classes* are classes added to the framework by application developers in order to implement specific functionality. They can be modified freely.

As the application is developed, it is important to distinguish between these two types of classes using graphical properties such as color or shading. Figure 3.1 shows a simplified view of a client-server framework (CSF [26]) in UML. The ability to see the overall design of the framework is one of the main requirements for the tool. UML provides a standard diagramming notation for both the structure and behavior of the framework through class and interaction diagrams. What must be added is explicit support of the notion of *hooks* that works in conjunction with the UML.

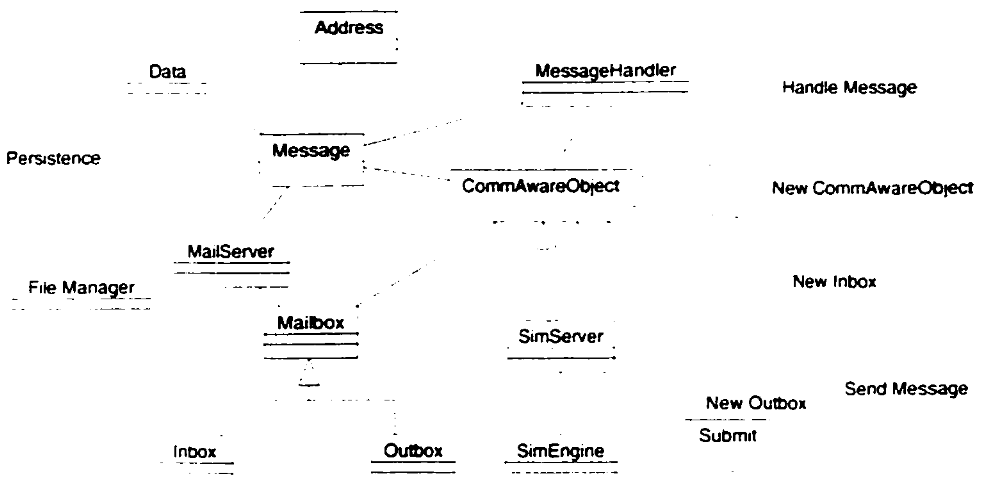


Figure 3.1: The CSF Main View

The overall view of the application, as shown in Figure 3.1, is called the main view. It contains all of the classes of the framework, along with any additional classes that are added by application developers. For example, Data is a framework class while SimServer is an application class. *Hooks* such as Handle Message are shown as ovals which connect to the primary class participant for the *hook*. Whenever a user tries to modify or delete one of the framework classes such as CommAwareObject, a warning is given with an explanation of the potential danger of changing the framework. There are cases when the framework must be modified to complete the development of an application because of some limitation of the framework or because the application developers are scavenging parts of the framework. Modifying application classes such as SimServer do not cause any warnings.

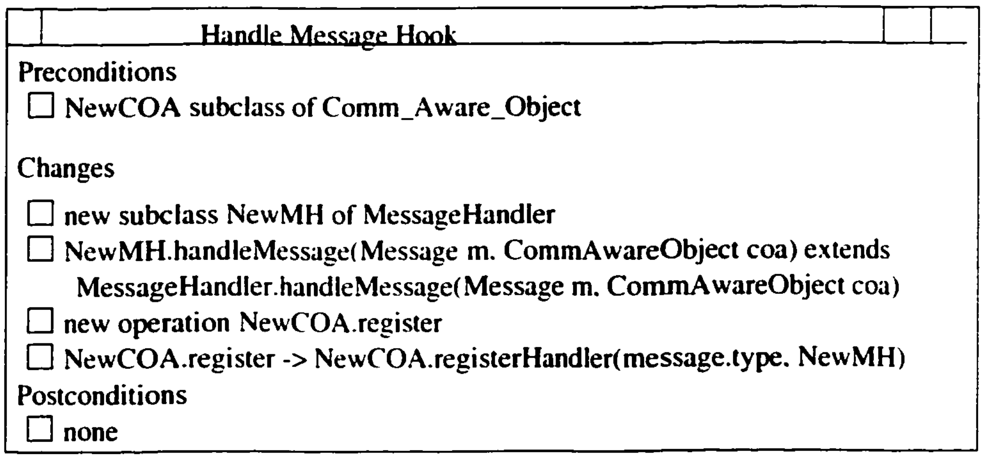
Option *hooks* are a special case, since they include optional components to the application, that are typically provided as part of a framework library. Attempting to modify the components themselves produces a warning, but modifying the choice of components does not.

Users of the tool can develop applications in two main ways. In the first, application developers use the editing tool to add or modify application classes as desired. The second, preferred, method is choosing and enacting hooks within *hook views*.

## 3.1.1 Hook Views

When a user clicks on one of the *hooks* within the main view, a *hook view* opens. A *hook view* contains a subset of the overall view of the framework that applies to the given *hook*. Both class and collaboration diagrams are contained within the view. The view forms the context in which the *hook* applies, in particular, the participants of the *hook* and related classes. Since determining the complete context of a *hook* is difficult, *hook views* are defined by the framework builders rather than generated automatically and dynamically. Two or more *hooks* may share the same context and thus the same view. As an example, the *hook view* shown in Figure 3.2 provides the context of the Handle Message *Hook*. It contains not only the diagrams but also an additional window, the enacting window, which lists the pre and post conditions of the *hook* that was requested by the user along with the changes that can be made using the *hook*.

Within a *hook view*, application developers can enact the changes for the *hook*, semi-automatically with the tool's help. The allowable conditions and change statements are defined by a grammar that is interpreted by the tool. The tool interactively guides the user through the conditions and changes contained in the *hook*. It performs as much as it can automatically and requests user input as required any required. At the first stage in the process of enacting the *hook*, any preconditions of the *hook* must be satisfied. Next each of the changes within the *hook* are enacted, or ignored if the user deems they are not needed. Finally, all the postconditions of the *hooks* should be satisfied. Checking a single precondition or postcondition, or enacting a single change statement is called a *step*.



Handle Message Hook

Preconditions

☐ NewCOA subclass of Comm_Aware_Object

Changes

☐ new subclass NewMH of MessageHandler

☐ NewMH.handleMessage(Message m, CommAwareObject coa) extends MessageHandler.handleMessage(Message m, CommAwareObject coa)

☐ new operation NewCOA.register

☐ NewCOA.register -> NewCOA.registerHandler(message.type, NewMH)

Postconditions

☐ none

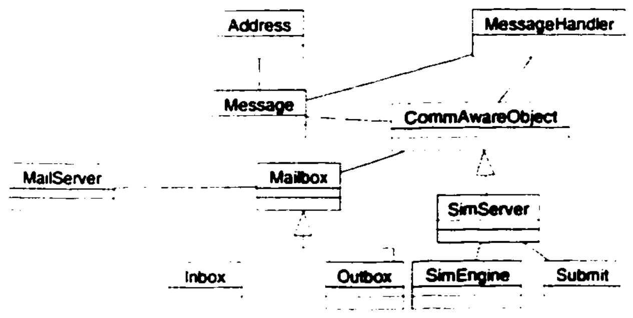


Figure 3.2: CSF: Handle Message Hook View

The application developer has the following actions that can be performed within a *hook view*:

- *Start enactment.*
- *Do a step.* (Check one precondition, one postcondition or perform one of the changes within the *hook*.)
- *Undo a step.*
- *Undo all steps and start over.*
- *Suspend enactment.* (This action is required in order to perform some other tasks or leave the work for a later time.)
- *Invoke another hook.* (*Hooks* which use other *hooks* place the current *hook view* in a suspended state and open up a new *hook view*.)
- *Resume with playback.* (If modifications to the application have been made within the context of the *hook view*, then preconditions must be checked again to ensure that none of the modifications have violated preconditions. After the preconditions are checked, changes that were previously made by the developer can be 'played back' meaning they are automatically invoked by the tool.)
- *Resume without playback.* (If no modifications have been made since a *hook* was last enacted, then the application developer can resume stepping through the *hook* from where he last left off.)
- *Commit with all conditions met.* (Changes made within the *hook view* are propagated to the main framework view.)
- *Commit without all conditions met.* (Changes should only be propagated when all the postconditions of the *hook* are satisfied.)
- *Throw away changes.* (Any changes within the *hook view* are simply discarded and the view is closed.)

### 3.1.2 View Consistency

When changes are committed, consistency must be maintained between the *hook views* and the main framework view. Consistency is maintained by propagating the changes from the *hook view* to the main view. All of the changes are logged by the tool and can be played back in both the main view and the *hook view*. The framework classes that participate within the *hook* serve as the anchors that exist in both the framework and the *hook view*. In general, propagating an arbitrary set of changes from one view to another is a very difficult problem. Due to the nature of the changes that can be made within a *hook*, the problem becomes much simpler. Only application classes are modified, so two views will always have a set of framework classes in common. Furthermore, the changes involve some modification, but these are mostly additions of classes, methods or properties that can be easily propagated to the main view. However, in some cases, there will be interference between *hooks* that must be detected.

### 3.1.3 Hook Interference

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 an active or suspended *hook view*. Interference can occur when:

- A participant of a *hook* is deleted.
- A participant of a *hook* is modified.
- A namespace conflicts arises, such as two classes being given the same name.

In the general case, whenever arbitrary changes are made to a class in any view, all *hook views* currently open with that class as a participant must have their preconditions checked and potentially undergo a *resume with playback*

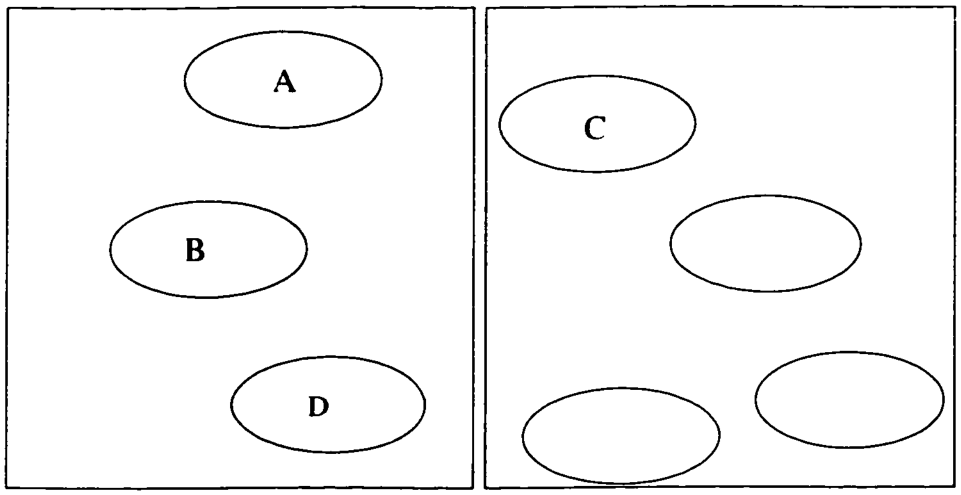


Figure 3.3: Mutual Exclusion Subsets

*operation.*

If the changes only involve the enactment of *hooks*, then the tool can help to prevent interference by defining sets of mutually exclusive *hooks*. ***Hooks*** with the same application class participants have the potential to interfere with one another. Such ***hooks*** are said to belong to the same participant set. Two *hooks* within the same participant set cannot be enacted at the same time. For example, suppose ***hooks*** A, B and D have some common participant application classes. Suppose C does not share any common participant classes with A, B or D. Then, as shown in Figure 3.3, ***hooks*** A, B and D cannot be enacted at the same time, but ***hooks*** A and C can. In this manner, the set of *hooks* for a framework can be grouped into mutual exclusion subsets as shown in Figure 3.3. Since ***hooks*** do not allow the deletion of participants, only ***hooks*** which modify the potential participants of ***hook*** A need to be considered for belonging to a common participant set for ***hook*** A.

Currently, the framework builder defines the sets of mutual exclusion in ***HookMaster***. The default is that all ***hooks*** are considered to be in the same set and therefore ***hooks*** can only be enacted serially. Namespace conflicts can occur for example when two new subclasses of a single class have the same name. This potential conflict can be checked prior to committing the changes to the *hook view*.

*Hooks* are automatically considered to interfere with themselves, but this is not always the case. A *hook* may not conflict with itself if it only adds things, unless the *hook* is meant to be used only once. Two separate views (two different windows) of the same *hook* may not have the same application classes as participants, and therefore shouldn't be in conflict. However, one view update may be overwritten by the second view update. The general problem of merging view updates is very complicated. Currently, Rational Rose 98 insures that such updates are serialized.

Two *hooks* can also invoke incompatible options within option *hooks*. The options should be specified in pre or post conditions to ensure that this doesn't happen.

The tool does not consider implicit conflicts such as those created by a method or class which calls a participant during normal framework operations. A distinction can be made between interference during the enactment of *hooks* and *ad hoc* changes to the code done outside the *hook* enactment process. Such *ad hoc* changes can cause some subtle conflict within the application and should be avoided.

### 3.1.4 Hook Books, Examples and Use Cases

*Design diagrams* and *hook views* should not be the only means of presenting the framework. Other information such as use cases, examples and class descriptions give additional information to the user to aid in application development. Use cases provide a means of describing typical scenarios, often involving sets of actions associated with framework classes. They are valuable to people first learning to use the framework, or learning to use the framework in a different way.

Example applications are equally valuable to show how the framework can be used. Users typically grasp concrete examples more quickly than abstract descriptions. The examples illustrate both of the framework in general (sample applications) and of individual *hooks* and use cases.

Descriptions of classes and methods 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 *hook book* is a listing of all *hooks* within the framework. The book lists the name and the requirement sections of the *hook*. The *hook book* can be used to browse the list of *hooks*, or it can be searched for particular keywords or matching requirements. Additionally, the *hook book* can be used to monitor which *hooks* have been used, how many times they have been used, and which *hook views* are currently open among all of the multiple users of the tool on a given framework.

All of these things, the use cases, the examples, the class and method descriptions, and the *hook book*, are linked together to form a web of information about the framework which can be easily browsed. Use cases link to a series of *hooks* that are used within the use case. They and *hooks* also point to examples of the use of the framework, and conversely, examples point to the *hooks* that have been used in constructing the examples. *Hooks* also point to the descriptions of the methods and classes that participate in them.

Finally, a log of all of the changes made through the *hooks* is kept which can then be reviewed as parts of a technical review or when errors are detected. The log is also an invaluable tool when changes to the framework itself are made (ie. a new version is released). If *hooks* have changed in the framework, the log will show which parts of the application have to be modified to work with the new version of the framework.

## 3.2 Maintainers and Developers

One of the key advantages of the tool is its ability to be used for any framework. Once the initial development of a framework is completed, the framework builders import their design and codes into the tool and then define the *hooks* for the framework. However, the tool as currently designed and prototyped is not appropriate for the initial development phases of a framework. Development typically happens using a tight spiral model, rapid development approach. Constantly updating the framework model for the tool would require too much unnecessary overhead. Once the code base has become stable, it can be imported into the tool to aid in the use and future evolution of the framework.

During evolution, changes are typically made to the main view and then the *hook views* are updated accordingly. Maintaining consistency is usually not a major concern, since modifications should be made by few maintainers. Evolution of the framework involves two main areas: modifications to the *hooks* and modifications to the design of the framework itself.

*Hooks* can also be modified by framework maintainers, either to correct errors, to improve the *hook* or to evolve the underlying framework has evolved in some way. The main difference between modifying an existing *hook* and adding a new *hook* is that the *hook* description and view already exists, otherwise the same process is followed.

The framework itself will evolve over time to add new functionality or to recast existing functionality in a new way. Changes to the existing framework can be accomplished within the tool (after turning off the warnings). Any changes to the participants of a hook will require changes to the *hook* and will likely require changes to existing applications as well.

# Chapter 4

# Architecture for Realizing HookMaster

Existing tools, such as graphical user interface builders [27], only support one framework and are not easily customized. *Hooks*, which describe the intended use of a framework, can form the basis of a general, flexible tool that supports application development for any O-O frameworks.

## 4.1 Analysis

A general tool will enable framework builders to specialize 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. Users of the tool can develop applications in two main ways. Firstly, application developers use its editing capabilities to add or modify application classes as desired. The second, preferred, method is enacting ***hooks*** within ***hook views*** because the ***hook*** description is a formalized description for application development as prescribed by the framework developers.

A *hook* is defined by a grammar made up of several sections. The tool provides a parser to interprete *hook* statements. The tool processes the activities contained in the *hook* description, performing as many actions as it can automatically and requesting information when the user interaction is required. Aid to framework maintainers is provided by allowing extensions to the framework and the *hooks* themselves to be made and incorporated back into the framework. The tool allows maintainers to add new framework components, add new *hooks*, modify *hooks* or modify parts of the framework. The tool incorporates the assumption that the framework view and *hook views* are based on UML as support by the Rational Rose tool [5][8]. UML provides a standard diagramming notation for both the structure and behavior of the framework through class and interaction diagrams [7], and Rational Rose is one of the few tools at present that implements UML notation. A key feature of the *hook* tool is its explicit support on extended notation for *hooks* in UML. Also, the tool includes support for interpreting each statement of the change section within *hooks*. The user can semi-automatically enact with *hooks*.

## 4.2 Design

The following section describes the architecture of the tool. Each subsystem is described in detail, and can be mapped to an actual design description in the Subsystem Description section.

### 4.2.1 Logical System Architecture

The tool is partitioned based on common attributes into an object-oriented architecture with the following five components as shown in the component diagram in Figure 4.1. The **Monitor** is a system module, and according to the definition of a module in UML, it can be represented as a component. **Rose** is treated as a package that has separate specification and realization parts.

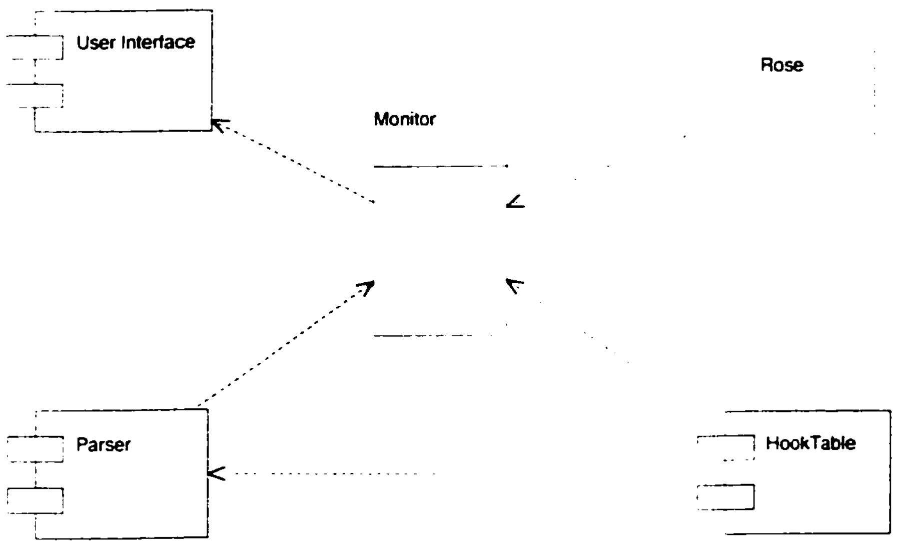


Figure 4.1: Component Diagram of Architecture

### 4.2.2 Subsystem Descriptions

#### User Interface

Two of the tool requirements are to support the application developer in selecting a *hook* and then enacting that *hook*.

*Hooks* are currently text-based, yet it is relatively difficult to understand them independent of examples and other framework documentation [16]. *Hooks* are targeted at the problem of understanding interactions between the framework and applications. A more graphical definition representation for *hooks* has been developed for incorporation into the tool, thereby allowing the user to locate the variation points corresponding to the *hooks* easily and quickly. Consequently, an intuitive easy to use interface is a key requirements of the tool.

In order to adequately represent the *hook views*, the user interface has two parts. Firstly, a main window displays the *hook*, and provides a menu that the user can easily use to select the functions supported by the tool. The *hook* description is shown when the user selects a *hook* from a list of the current *hook* names. The user can enact the *hook* by simply clicking the enactment button. The preconditions, change statements, and postconditions of the *hook* are semi-automatically executed step by step. For new element change statements, the corresponding dialogs are popped up, and the user inputs the application name within the dialog. All new applications are shown on the *hook view*. The other types of statements can be automatically examined through the user interface. Each statement is clearly highlighted after it has been enacted.

The second part of the user interface contains windows for **Rational Rose 98** to represent the *hook view*. What part of the application representation is presented automatically to the user in the **Rational Rose 98** windows depends on the nature of the interactions that take place during enactment.

The user interface works closely with the **monitor**. It is the main program that initializes the system when the tool starts. The user interface also handles the input from users, and receives updated state information from the **Monitor**. The **User Interface** component includes the following classes: **MainUI, FileManager, NewClass, NewProperty, NewOperation, Caller** and the **Rational Rose 98 API**. Their inter-relationship are shown in the class diagram in Figure 4.2.

### Monitor-Controller

The **Monitor-Controller** controls the exchange of state information and operations among the **User Interface**, the **Parser**, and **Rational Rose 98**. Specifically, it monitors the user interface state and notifies the corresponding subsystems when the state changes, as shown Figure 4.2.

A critical part of the design rationale for the **Monitor-Controller** is the provision of strong support for reuse and evolution of the system by implementing a general broadcast mechanism for system events. Other subsystems in the tool can register an interest in an event by associating a function with an event. When the event is announced, the tool itself invokes all of the functions that are registered for the event. New subsystems may be added to the existing system by registering their interest in events in the **Monitor-Controller**. For example, in the future, when the Java development environment is integrated with the tool, this can be accomplished by simply registering the development environment interface to receive the events of the system.

## Parser

A *hook* description is written in a specific format made up of several sections. The sections detail different aspects of the hook. The change section is the section of the *hook* that prescribes the changes to the interfaces, associations, control flow and synchronization among the components listed in the participants section. When the user enacts the *hook*, all change statements that can be handled are executed. All statements that require user assistance are enacted in cooperation with the user. All allowable condition and change statements are defined by a context-free grammar, that helps to make the description precise and uniform, and aids in the analysis of *hooks*.

The **Parser** works closely with **Monitor-Controller** and **Hook Table**. The **Monitor-Controller** takes as input the text description of *hooks*, and sends it to the **Parser**. The **Parser** interpretes the changes statement of the *hook* according to the change statement grammar, and stores the identifier and other key information in the **Hook Table**. The **cHookParser** class belongs to the **Parser** component as shown in Figure 4.2.

## Hook Table

As a *hook* is enacted, a log is kept of all the activities associated with the change statements. This log can later be reviewed when errors are detected, or for *hook* enactment playback. The log is also invaluable in determining the effects of changes to the framework itself (ie. a new version is released).

The **Hook Table** stores all identifiers, other key information from the **Parser**, and any inquiry information from the user. All informations in the **Hook Table** is persistent and is therefore retained when the application developer quits the system.

The **Hook Table** contains **cHookTable, cHook, cClass, cProperty, cOperation, cParameter and cStatement** classes, as shown Figure 4.2.

**Rational Rose 98**

A key requirement of the tool is that it works in conjunction with **Rational Rose 98**. Using the UML notation, we can create and refine *hook views* within an overall design model representing the framework domain. A overall model in **Rational Rose 98** can contain many different kinds of elements, such as classes, objects, use cases, and their inter-relationships. A model contains diagrams and specifications, that provide a means of visualizing and manipulating the model elements and their properties. We can control which elements, relationships, and property icons appear on each diagram, using facilities provided by the application window of **Rational Rose 98**. Within its application window, each diagram is displayed in a diagram window, and each specification is displayed in a specification window.

When the code base of an object-oriented framework is finished, **Rational Rose 98** can reverse the source code into UML diagrams. From this representation it is possible to create the framework view and *hook views*. In order to capture the design of the framework to which the *hooks* can be attached, each view contains two kinds of diagrams. The *class diagram* is used to represent the static relationship between classes of the framework, and the *collaboration diagram* is used to present the dynamic behaviors. Each *hook view* is a subset of the framework view, which forms the context for the *hook* description.

**Rational Rose 98** only works closely with the **Monitor-Controller** subsystem. It is invoked by the **Monitor-Controller** when the developer starts the tool. The **Monitor-Controller** accesses to the *hook* documentation and the participants from **Rational Rose 98** when the developer chooses a *hook view* from the user interface. Also, when the developer enacts a *hook*, the **Monitor-Controller** notifies, modifies or creates the elements within both the framework and the *hook view* in **Rational Rose 98**.

### 4.2.3 Class Diagram

The static relationships among classes of the *hook* tool are represented as Figure 4.2. The detail description of each class will be discussed as part of the implementation description in the next chapter.

## 4.3 Use Cases

When the initial development of a framework has been completed, the framework builders only have to import their design into the tool and then define the *hooks* for it. That is, once the code base has become stable, it can be imported into the Rational Rose 98 to create the framework view and *hooks views* which are nested in the logical view of Rational Rose 98.

The following use cases describe high-level interactions between the tool user and the tool. Interactions are mainly defined in terms of actions users take through the user interface of the tool.

### 4.3.1 Use Cases Diagram

The use cases diagram (Figure 4.3) represents external behavior of the entire system as visible to framework users.

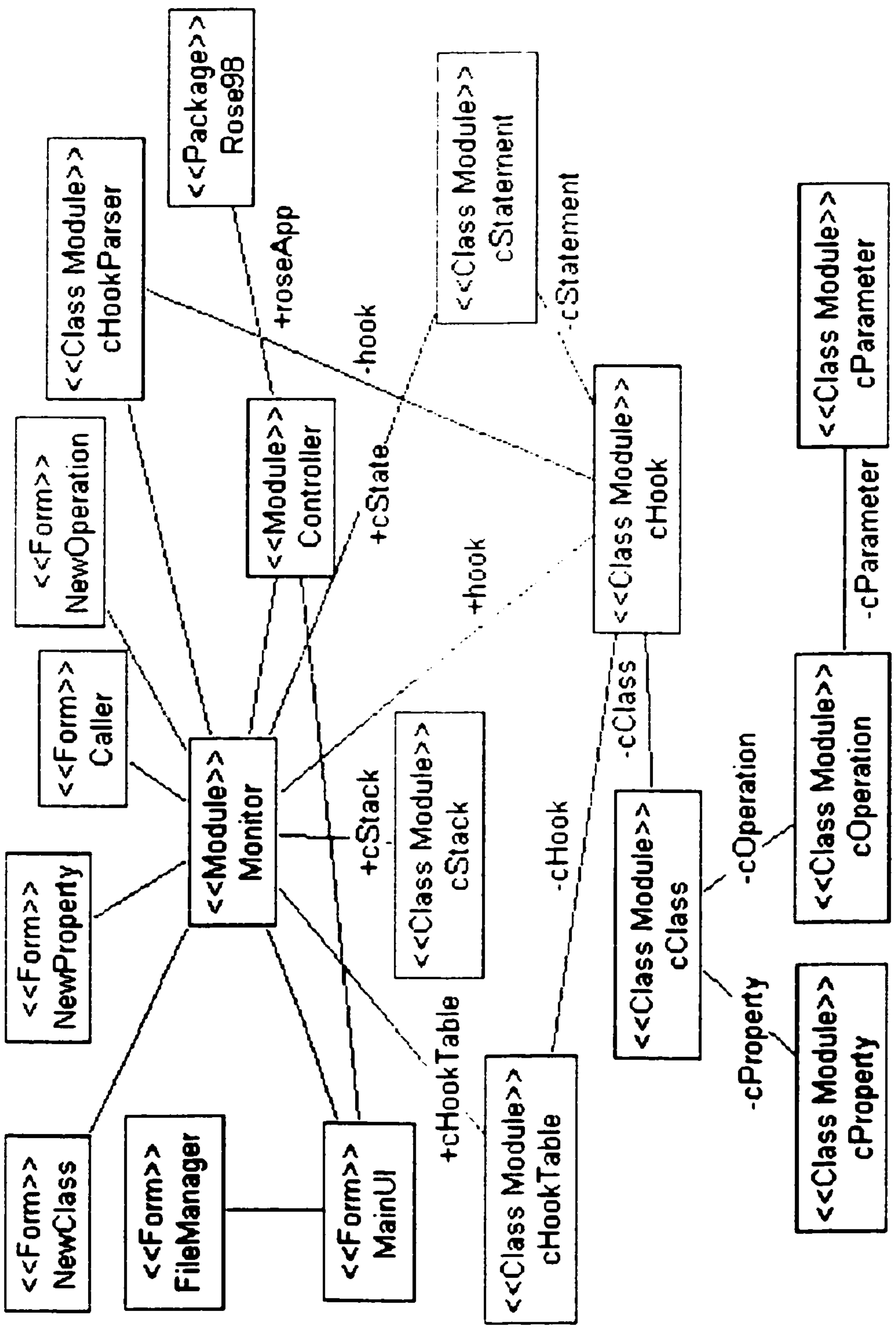


Figure 4.2: Class Diagram for UI and Parser Subscriptions

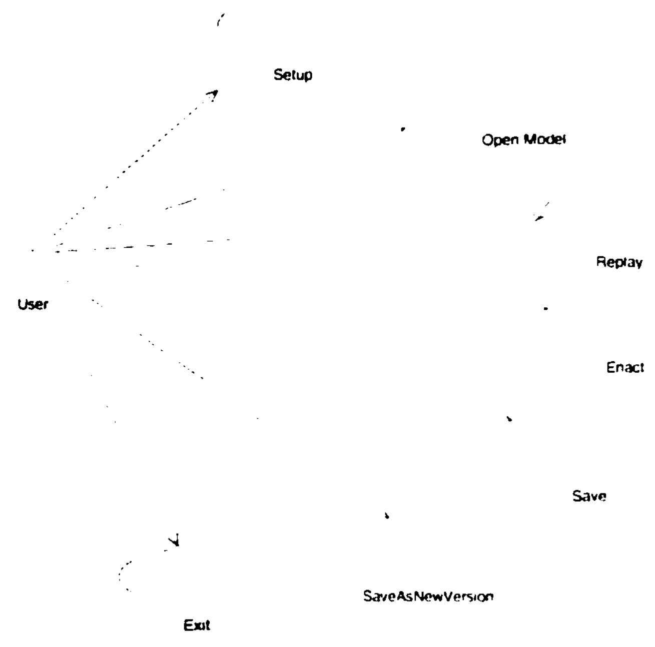


Figure 4.3: Use Cases Diagram

### 4.3.2 Setup

- The user starts up the tool to work on a new application for the first time.
- The user interface displays the main window and the application interface of Rational Rose 98.

Figure 4.4 shows the scenario.

### 4.3.3 Open Model

The tool initially assumes that a model file exists. If it does not, the framework builder has to reverse the source codes of the framework into UML diagrams within Rational Rose 98, and create the framework and *hook views*, then, save these as a model file of Rational Rose 98.

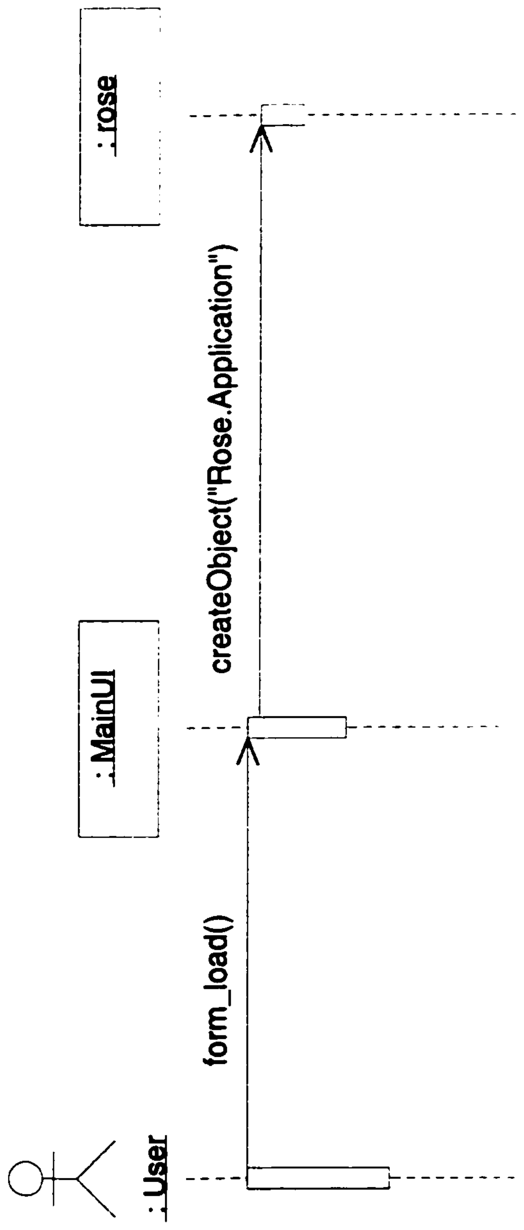


Figure 4.4: System Setup

- The user opens a model file by selecting the open menu from the main window. A dialog box is presented that contains the list of directories and files. The user selects one directory and one model file to open.

- The application interface of Rational Rose 98 opens the model file, and displays the framework view and the *hook views* in the browser window.

- The main window receives from Rational Rose 98 the names of the framework and all the *hooks* within the model file, and displays them in a list on the main window.

- The user can view the framework view and the *hook view* in two ways: 1. selecting one name from the list of the main window, in which case the view will be displayed on the diagram window of Rational Rose 98 interface, or 2. clicking one icon that represents the view within the browser window of Rational Rose 98 d:.ectly.

Figure 4.5 shows scenario for Open Model use case.

### 4.3.4 Replay

In this use case, changes that were previously made can be played back without modification. A scenario diagram for the replay activities is given in Figure 4.6 (The read_from_file function uses two different data types as paramter. It gets the string of the file name at the first time, then, use the file index handler which is an integer.).

- The user selects the replay item from the main window. A dialog displays a version tree including all versions that were previously made. The model file of this version is opened when the user chooses one.

- A list of the main window displays the *hooks* that have been completed. The application information can be played back to the user within the

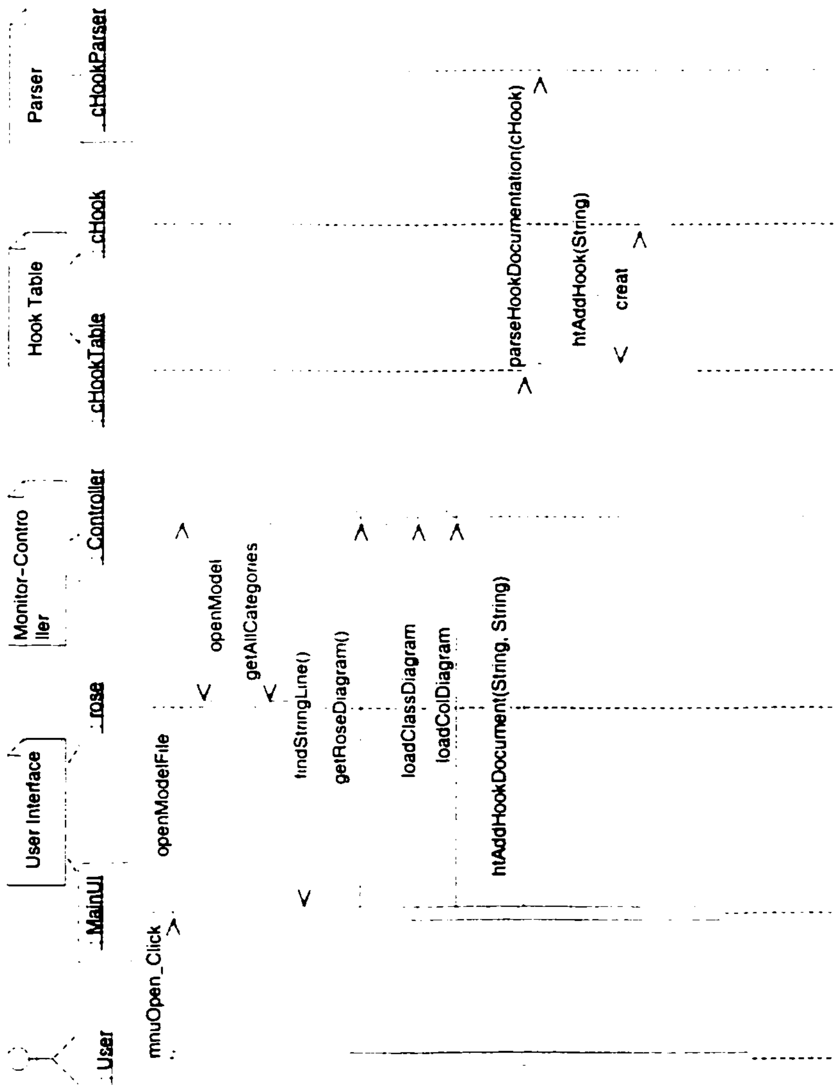


Figure 4.5: Open Model

main window and the application interface of Rational Rose 98 by choosing a *hook* name within the list, or clicking one icon from the browser window within Rational Rose 98.

### 4.3.5 Enact

The user can enact with the *hook*. The scenario is shown as Figure 4.7.

- The user chooses a *hook* name from the *hook* list on the main window. The text description will be displayed on a text field of the main window. Meanwhile, the corresponding *hook view* is shown on the diagram window of Rational Rose 98, which includes a *class diagram* and a *collaboration diagram.*
- The user starts enact the *hook* by selecting the enactment element from the main window (Note: checking a single precondition or postcondition, or enacting a single change statement is called a step.).
- All preconditions are checked first. If they are satisfied, each of the changes within the *hook* can be made. When the input is required in a step, a dialog box for this statement is popped up to ask for user input. The change the user requests is reflected in graphical UML notation on the *hook view* within the diagram window of Rational Rose 98. The next step is enacted.
- If the user undoes steps, the changes within the *hook view* are discarded.
- When a *hook* uses another *hook*, the current *hook view* is placed in a suspended state, and the new *hook view* is opened up.

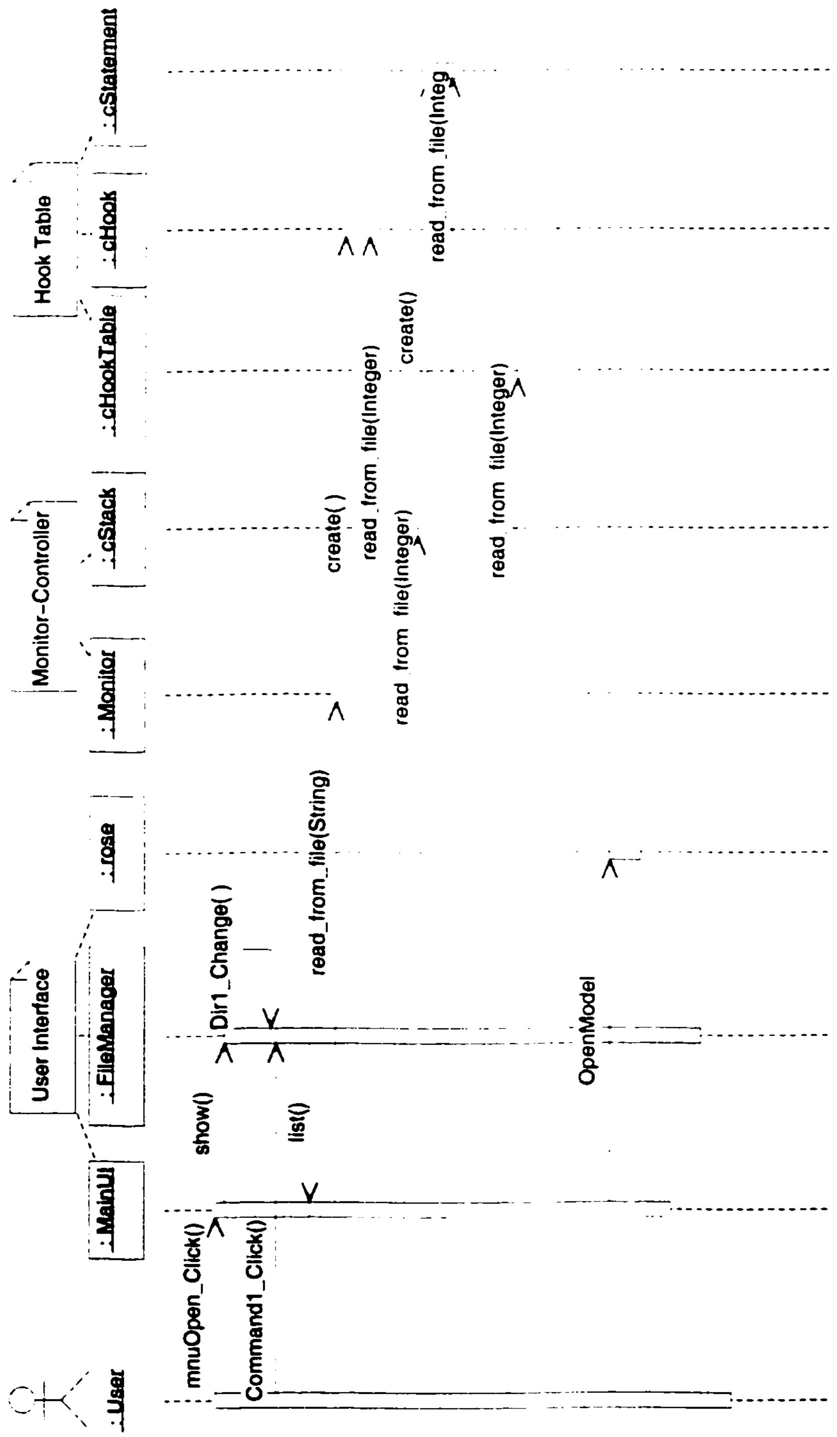


Figure 4.6: Replay

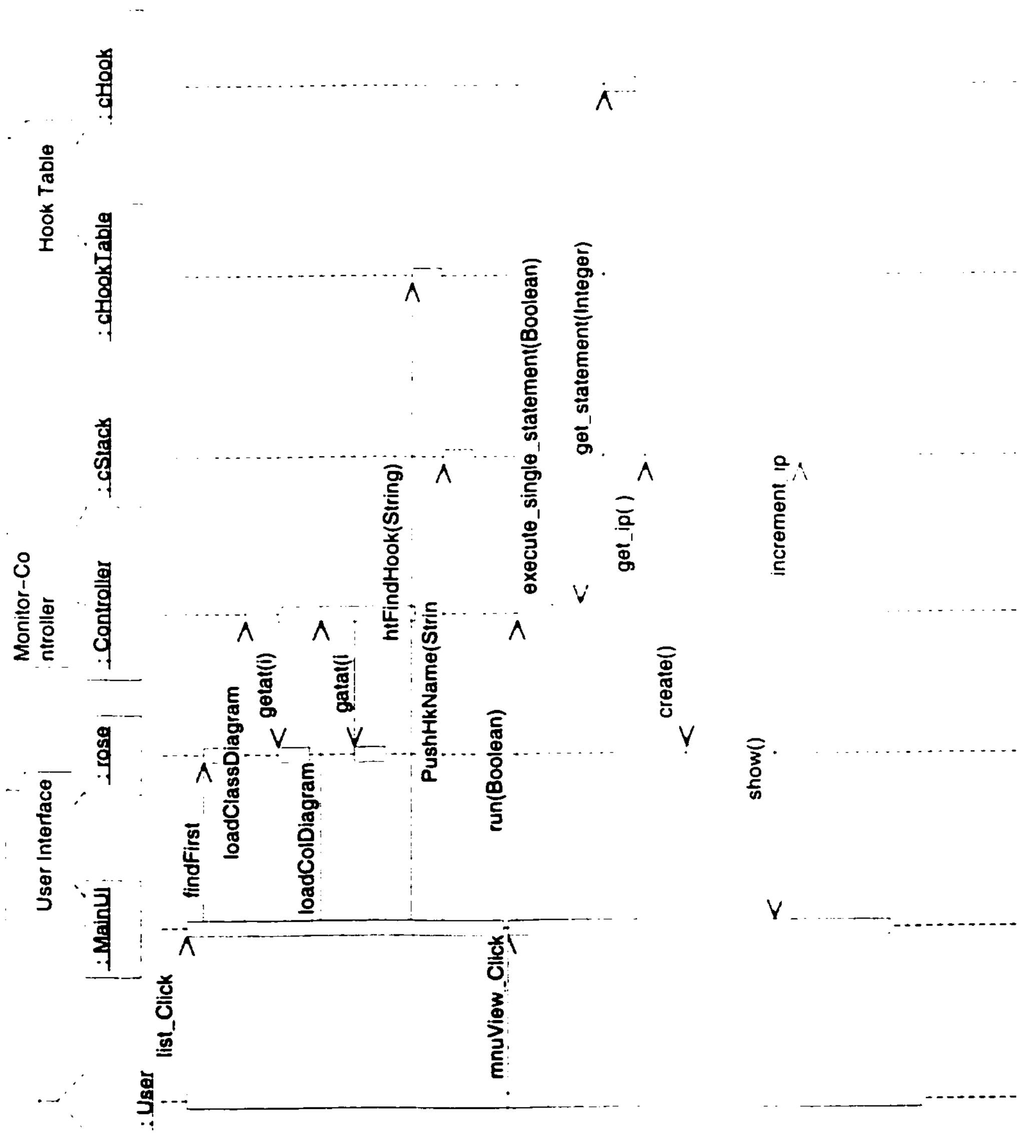


Figure 4.7: Enact

### 4.3.6 Save

All user activities can be saved in the current model file. The scenario is given in Figure 4.8 (The write_to_file function uses two different data types as paramter. It gets the string of the file name at the first time. then. use the file index handler which is an integer.).

- The user selects the save element from the main window.
- The changes that were done by the user can be saved into the current model file as graphical UML notations within Rational Rose 98. Also. the tool automatically creates a text file to log all of the changes.

### 4.3.7 SaveAsNewVersion

In this use case, the changes that were completed can be saved as a new version. The scenario is shown as Figure 4.9 (The write_to_file function uses two different data types as paramter. It gets the string of the file name at the first time. then, use the file index handler which is an integer.).

- The user selects the saveas element for the main window.
- The changes that were done by the user can be saved as a new version within the version tree.

### 4.3.8 Exit

The user quits the tool. The scenario is shown as Figure 4.10.

- The user selects the exit element for the main window. The main window is closed.
- The application interface of Rational Rose 98 is notified to exit.
- The tool is shut down.

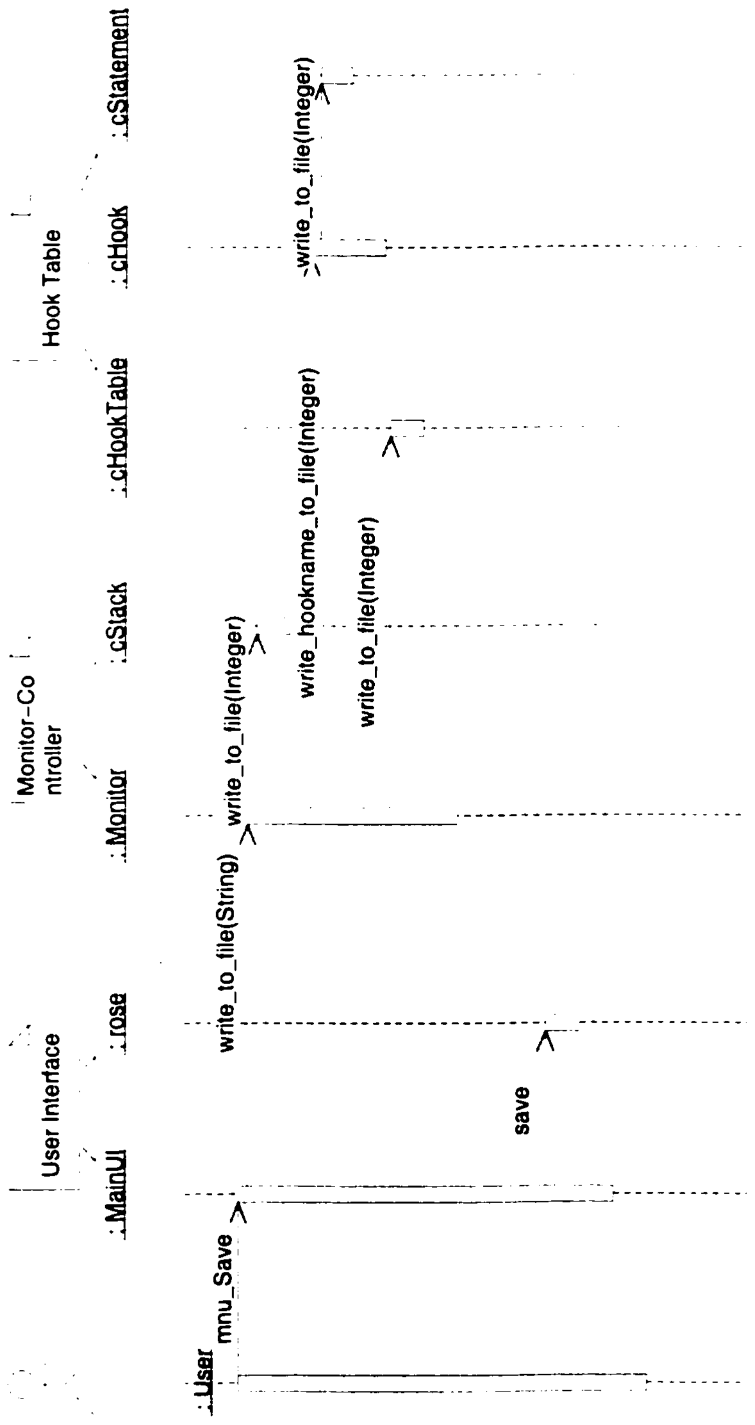


Figure 4.8: Save

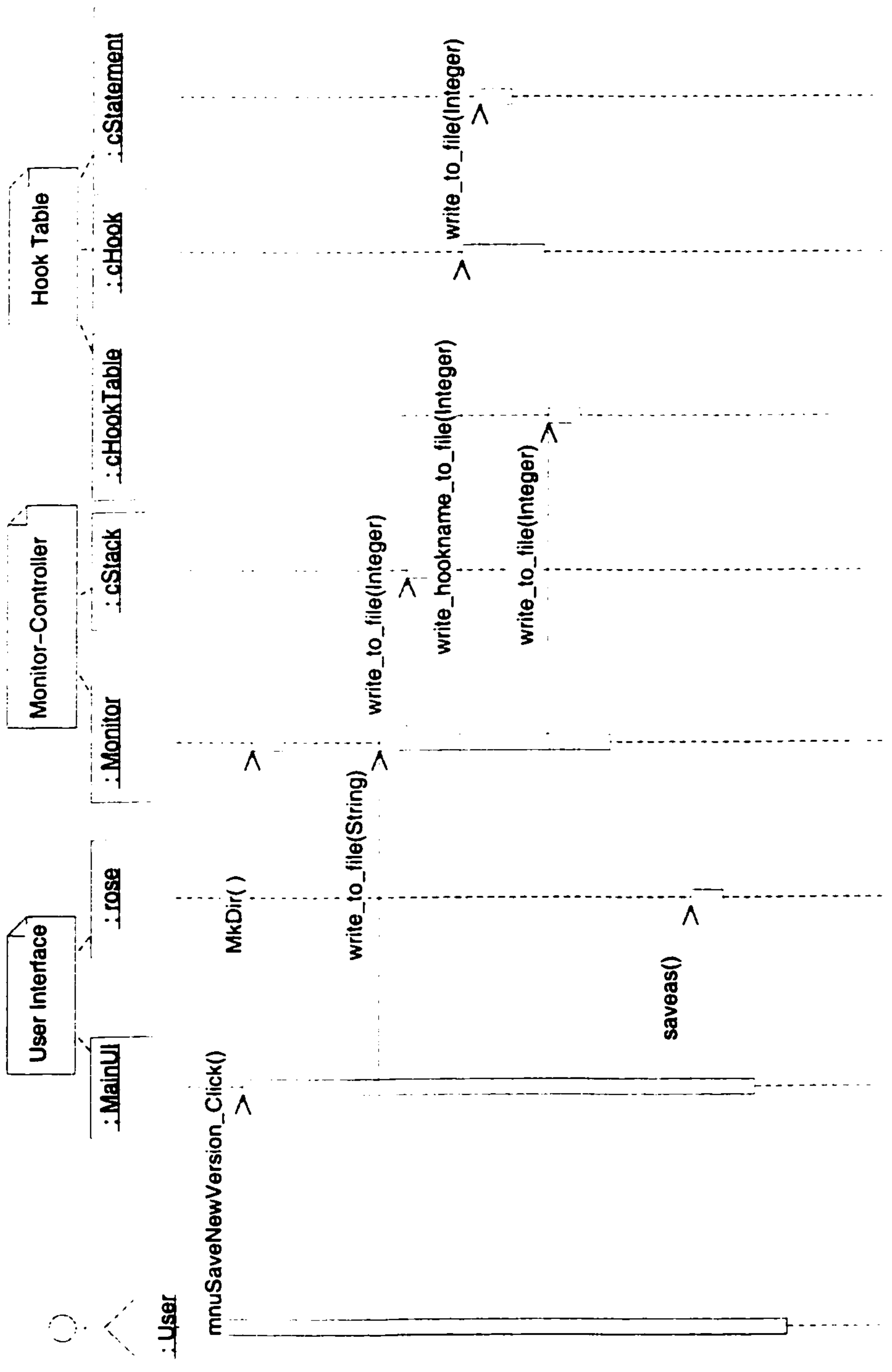


Figure 4.9: Save As A New Version

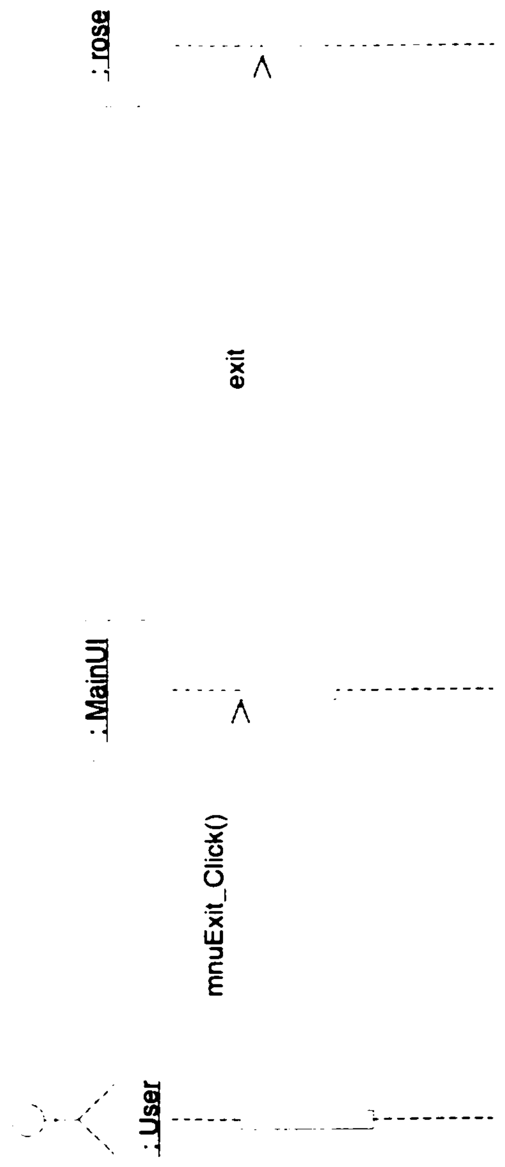


Figure 4.10: Exit

## 4.4 Summary

In this chapter we have presented several aspects related to the design of the **HookMaster** tool according to the tool functionalities. First, we have described the architecture of the tool. We partitioned the system into five components. The dependency relationship between two components represents a change in which one component may affect or supply information needed by the other component. We have added implicit invocation to the system instead of tightly coupling the user interface and Rational Rose 98 through direct procedure calls. One important benefit is that it provides strong support for reuse. We have designed a module **Monitor-Controller** to monitor and control events between the user interface, Rational Rose 98, the *hook* parser and enactment engine. We register a component (module) to receive announced events by associating one of its procedures with each event of interest. In this manner, the system can be easily integrated with other tools.

Secondly, we introduced the use cases that define the coherent behaviors. All use cases describe external behavior of the system as it appears to users.

# Chapter 5

# Prototype Implementation of HookMaster

## 5.1 The Scope of Implementation

To demonstrate that applications using *hooks* can be well-supported with a visual tool on UML, we implemented a prototype called **HookMaster** for the hook tool. The focus of the work is on representing knowledge about the use of object-oriented frameworks to application developers. The prototype followed the design of the framework as identified in the previous chapter. The *hook* model serves as a basis for building **HookMaster**. This chapter also describes the process that application developer uses to develop applications from a framework, and how reuse information can be extracted from the architecture of a framework. Through the tool, we validate that the *hooks* can be graphically presented in a way that allows the framework user to easily build applications.

The prototype handles all types of change statements except these involving the explicit removal, fill-in and modification of code. Our prototype successfully provides support for the selection of *hooks* and the enactment and replay of individual *hooks*. It also provides some support for the evolution of frameworks, although additional work needs to be done in this area in the future. The prototype demonstrates how *hooks* can be connected to UML design notations.

### 5.1.1 Technological Consideration for Prototype

The prototype implementation is based on the assumption that the *hook views* are nested in Rational Rose 98. Rational Rose 98 supports a relatively complete implementation of UML, and offers the Rose Extensibility Interface (REI) to enable the integration of Rational Rose 98 with the other tools and programs. REI provides extensive capabilities that allow users to easily access and extend Rose's capabilities [8]. Figure 5.1 shows the core Rose components, the REI, and the relationships between them.

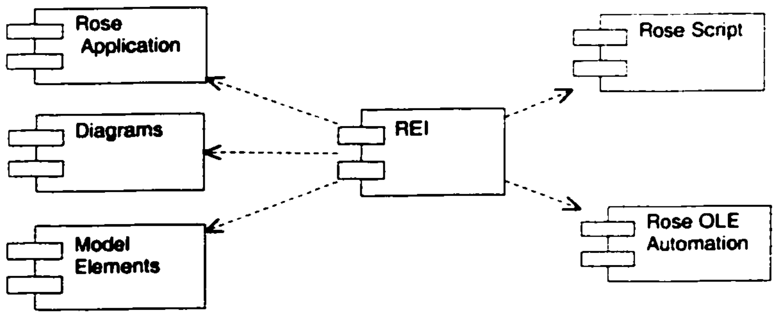


Figure 5.1: The Rose Extensibility Interface

Communication with the REI is through Rose Scripts that are Visual Basic compatible or through Rose OLE Automation. In both cases, the REI calls used in the prototype development are defined in the Rose Extensibility Interface Reference[8]. A main design consideration was the clear separation of the user interface from Rational Rose 98. To accomplish this, the Rose OLE Automation was deployed to share information with Visual Basic which is also an OLE-enabled application.

By loading a type library for Rose OLE Automation, we were able to use Rose class names to access the REI from the Visual Basic environment. That is, when working in Visual Basic, instead of using the Visual Basic object type **Object**, we used the name of the actual Rose class. Also, we were able to check the syntax of the properties and methods at compile time (early binding) instead of when the code is executed (late binding).

For Rose OLE Automation, the Rose Application object must be created in order to control the Rose application. There exists a global Rose Application object for Rose OLE Automation. To accomplish this, an instance of a Rose Application object is initialized when using Rose OLE Automation by calling **CreateObject** from within the application in Visual Basic. This call returns the object which implements Rose API's application object. We use the Rose Application object to get the current model by using a property called **CurrentModel**. This property is also used subsequently to open, control, save, or close a Rose model.

Visual Basic is a good visual development tool for rapidly implementing the prototype, and provides support for OLE. Visual Basic was used as an automation controller to call REI, and extract the relevant information that we need from a complex framework model.

## 5.2 Implementation of the Prototype

Following the previous the design of the prototype in Chapter 4, this section describes the implementation of the prototype.

### 5.2.1 Rational Rose 98

To comprehend the quality of the design of an object-oriented framework, a distilled presentation of design information and rationale is better than the mere presentation of source code. If the right information is extracted from the source code and presented in a more abstract, yet understandable form, source code access should not be necessary to understand the overall framework design. Generally, the framework unifies data structure and behavioral features into a single generic view for a family of applications. Data and behavior are closely related. The UML provides the static view using *class diagrams* and the interaction view using *collaboration diagrams* to capture information about the dynamic behavior of a system.

In order to improve the understandability and usability of the framework, the source code of the framework is reversed into UML diagrams, and *hook views* are then created using Rational Rose 98. *Class diagrams* are generated to represent the object structure of the framework, and *collaboration diagrams* are created to represent the dynamic behaviors between objects. The framework diagram is treated as the main view. To create the *hook views* on the framework view, we deployed UML's stereotyping feature.

UML provides three extension mechanisms to allow modelers to make some common extensions for modeling without having to modify the underlying modeling language. The extensions can be defined, stored as part of a model, and passed to other tools. The extensibility mechanisms are *constraints, tagged values, and stereotypes.*

A *constraint* is a semantic restriction represented as a text expression. *Constraints* can express restrictions and relationships that can not be expressed using UML notation. They are particularly useful for stating global conditions or conditions that affect a large number of model elements. For example, the constraint **xor** between two associations that share a common class means that a single object of the shared class may belong to only one of the associations at one time.

A *tagged value* is a pair of strings, a tag string and a value string that stores a piece of information about a element. The tag is a name of some property the modeler wants to record, and the value is the value of that property for the given element. For example, the tag might be **author**, and the value might be the name of the person responsible for the element. *Tagged values* can be used to store arbitrary information about elements. They are particularly useful for storing project management information, such as the creation date of an element, its development status. *Tagged values* can also be used to store information about stereotyped model elements as discussed below.

The most powerful mechanism is the *stereotype* which is a special model element defined in the model itself. The information content and form of a *stereotype* are the same as those of an existing base model element, but its meaning and usage is different. A tool can store and manipulate the new element the same way it does an existing element. The general notation for the use of a *stereotype* is to use the symbol for the base element but to place a keyword string above the name of the element. For example, the keyword string **communication** can be placed above or in front of the name of the model element that is used to support the communication aspect of a framework. We use the *stereotype* to tailor UML to create a graphical representation for a *hook*.

The main view of the framework is treated as a package, we define each *hook* as a category which inherits from the package, and each category is stereotyped by the *hook* name. Each *hook view* is a subset of the main framework view, which contains the participants within the *hook*, including a *class diagram*, a *collaboration diagram*, and the external documentation of the *hook* description.

The two basic elements of UML *class diagrams* are classes, packages and their relationships (e.g. generalization, aggregation, association, and dependency) [29]. Packages allow collections of classes and class hierarchies to be formed as a means of abstraction. The framework is considered as a package, and each *hook* is a category which is a subset of the package. In UML *class* *diagrams*, a class definition has three parts: the class name, attributes and operations. Connectors depict relationships, or links, between components and may be constrained to a subset of classes or packages.

A *collaboration diagram* models the objects and links involved in the implementation of an interaction. It can be constructed by taking the union of all the collaborations needed to describe all the operations of the object. Usually, a *collaboration diagram* contains objects of the classes, their links and messages. Each message has a sequence number, and is shown as labeled arrows attached to links.

Rational Rose 98 allows users to use a set of library functionalities to access each of *elements* automatically. Figure 5.2 is a *class diagram* showing the class hierachy of UML *elements* in Rational Rose 98. An *element* is the abstract base class for most constituents in the UML. It acts as an anchor to which a number of mechanisms may be attached. An *element* is specialized to *model elements* and *view elements*. A *model element* is an abstraction drawn from the system being modeled, such as a class, a message, and so on as shown in Figure 5.3. A *view element* is a projection of a single *model element* or a collection of *model elements*. The *view elements* are graphical symbols used to build the models (diagrams). The *view elements* can also be specialized to diagrams. All *elements* have names. A *package* is a grouping mechanism that can own or refer to *elements* (or to other *packages*) as shown in Figure 5.2. In the prototype, we treated the framework view as a *package*, and all *hook views* as categories stereotyped as subsets of *package*. The *elements* within a *package* can be of various kinds, such as model elements, view elements, models.

Most *model elements* have corresponding *view elements* that define their representation. The *class diagram* in Figure 5.3 shows how the *model elements* are specialized to the modeling concepts used in the UML by Rational Rose 98.

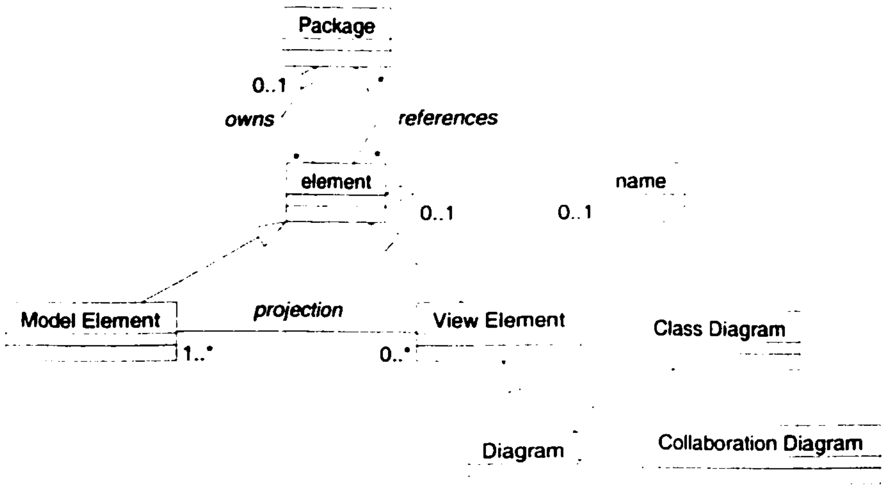


Figure 5.2: The Diagram Class Hierarchy

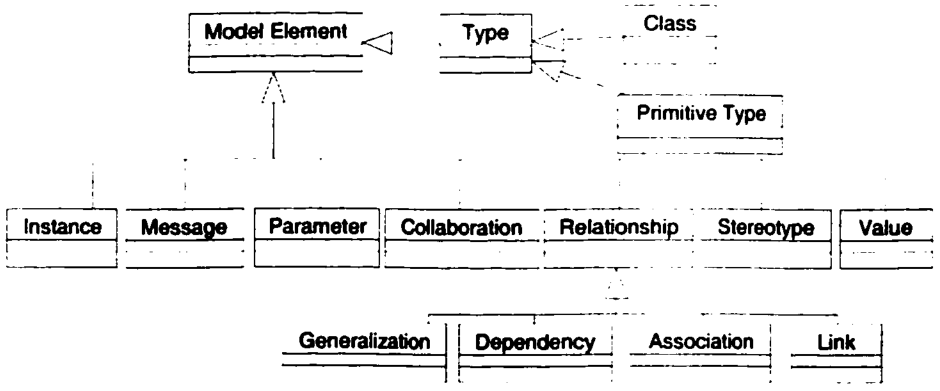


Figure 5.3: The UML Elements

## 5.2.2 User Interface

The user interface of **HookMaster** contains a main window, a Rational Rose API, and a set of dialogs. The main window and dialogs are implemented using Visual Basic 5.0, and interact with the Rose interface by OLE automation as shown in Figure 5.4.

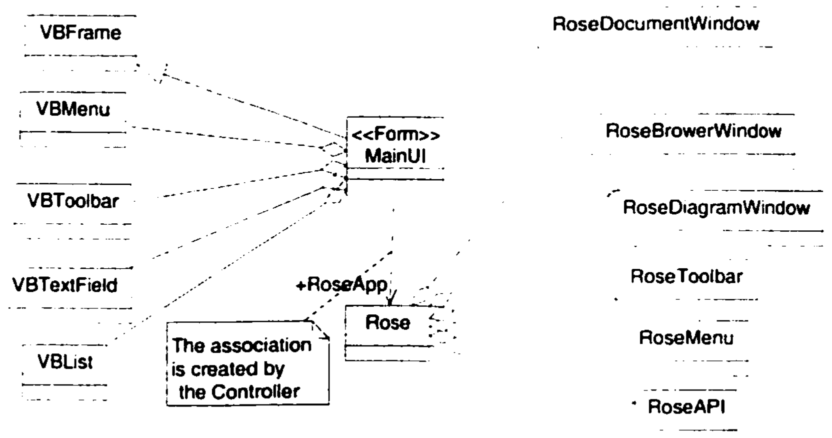


Figure 5.4: The class diagram for the user interface

The MainUI class is the class by which the tool is entered. When the user starts, this class sets up the system, and invokes Rational Rose 98 through the Controller class (see Figure B.8 in Appendix B). To use Rose as an OLE Automation server, we initialize an instance of a Rose Application object. We do this by calling CreateObject from within the MainUI class (see Appendix B Figure B.1). The following Visual Basic code shows how to instantiate the Rose application object:

```
Set roseApp = CreateObject("Rose.Application")
roseApp.Visible = True
```

The visibility of the interface of Rational Rose 98 is controlled by the Visible property within the MainUI class. After creating the Rose Application object, we may get all the application elements in the Rational Rose 98 specification by accessing the Rose library using the REI. Figure 5.5 shows the interface classes used to interact with Rationl Rose 98. After the user opens a model from the

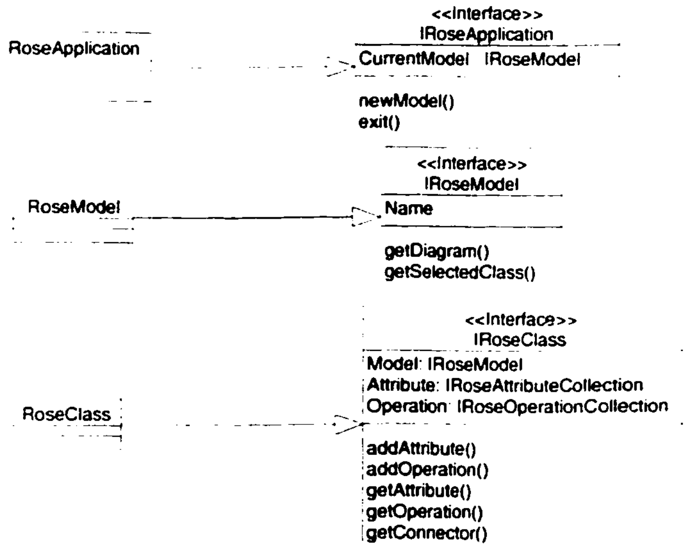


Figure 5.5: The Implementation for Rose Interface

user interface, the roseApp which has a property called CurrentModel will set up the model opened as the current model. Subsequently, we can retrieve all the elements from within the model as depicted in the scenario diagram Figure 5.6.

In the **HookMaster** prototype, one of the first activities involves opening the model which contains the *main* and *hook views* (Figure B.2 in Appendix B shows the FileManager class which supports the user's selection of the directory and the version of the model files). All the *hook* names in this model can be retrieved using **GetAllCategories** function of the REI in the MainUI class, and displayed on the List1.

When the user selects one *hook* name from the List of the main window, the corresponding *hook view* can be opened in Rational Rose 98, and it contains a

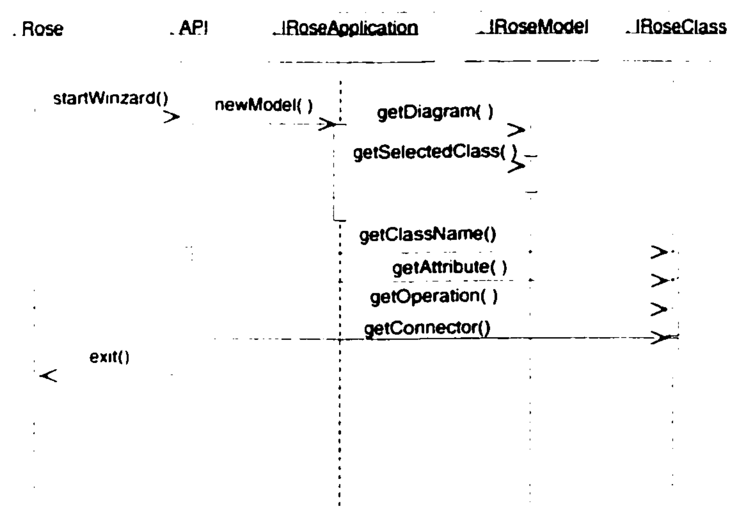


Figure 5.6: Rational Rose Scenario

*class diagram* and a *collaboration diagram*. Within the *hook view*, all elements of the class diagram and collaboration diagram can be accessed by using the functions of the REI. According to the discussion in last subsection, we extract components, connectors, objects, messages and links. Also, the REI provides the property of **documentation** which can be used to access the text description of the *hook* within the *hook view*. A text field in the MainUI class displays the *hook* description.

The prototype has the capability to replay the previous application development activities. A version tree is established to manage the various version of the model. When the user finishes enacting the *hook*, all the information concerning the enactment is saved in the current model version. Otherwise, the prototype allows the user to save the model as a new version by creating a subdirectory in the current directory. The version name consists of the model name and the system time, that is, the name of the model file plus the current system time (year+month+day+hour+minute). Each version also has a log file called **display** text file which stores the entire state of Rose model and the breakpoints.

The FileManager class managers the versions of the model file as shown in Figure B.2 in Appendix B. After choosing one version, all the *hooks* completed so far in an enactment of that version are displayed in the List2 of the main window. Those that remain to be completed are shown in the List1.

Most importantly, the user can enact with *hook views* through the user interface. Several different types of dialogs are popped up based on the kind of statement in the change section of the *hook*. The results of an enacted step are shown in the Rose interface automatically. In Appendix B, Figure B.3 shows the class implemented for creating a new class. Figure B.4 and Figure B.5 show the classes for adding new attributes and operations. The Caller class, shown in Figure B.6 allows the user to select an existing class of the framework.

### 5.2.3 Parser

The prototype contains a parser that analyzes all the *hook* descriptions, which are defined by a context-free grammar [32]. The Appendix A lists the *hook* description grammar. The parser has been implemented by the parser class as shown in Figure B.9 in Appendix B.

Lexical analysis is the first stage in processing the *hook* description. The input to the lexical phase is a sequence of characters. The analyzer groups them into lexemes, which are word-like elements, such as keywords, identifiers, and punctuation. These elements are indivisible units of the *hook* description grammar. The purpose of lexical analysis is to identify the lexemes in the input string and replace them with tokens. The tokens identify what the lexeme was in sufficient detail for parsing. The output of the lexical phase is a sequence of tokens. This sequence is an abstraction of the source. For example, the *hook* change statement:

**New subclass NewCOA of CommAwareObject**

has five meaningful entities: three keywords **New, subclass** and **of**, and two identifiers **NewCOA** and **CommAwareObject**. The tokens for keywords will always be the same. The tokens for identifiers are pointers to a particular string, which is the identifier name recorded in the hook table.

The next stage is the syntactic analysis in which the parser processes the stream of tokens and determines whether the syntactic structure of the input string matches the *hook* description grammar. Specifically, the input is the sequence of tokens arising from the lexical analysis and the parser is to organize these into a correct parse tree that is based on the *hook* description grammar. A parse tree is a hierarchical representation of a statement, starting at the root and working down toward the leaves. The general data flow of the parsing phase is illustrated by the Figure 5.7.

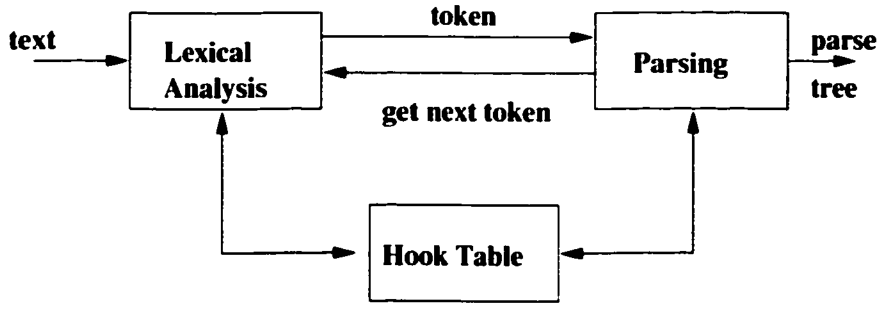


Figure 5.7: The parsing diagram

In the prototype, we deployed a common method for writing parsers called *recursive descent*. In this method, the parser proceeds by a straightforward inductive analysis of the *hook* description grammar. The basic idea is that a procedure is defined for each nonterminal of the grammar. The tokens are scanned from the left to right, and each time a nonterminal is encountered, the corresponding procedure is called, possibly recursively.

The *hook* description grammar is suitable for recursive descent parsing because the language is small, and parser efficency is of little concern. The grammar starts with the root non-terminal(⟨statement⟩) and works down toward the terminals, i.e., the leaves of the grammar parse tree. The expected terminals in the grammar are matched by the parser against the actual token types returned by the lexical analyzer.

The parser is built directly from the productions in the grammar. Non-terminals become function calls, terminals are matched directly against tokens by the token type returned by the scanner. To simplify coding, the returned token type is kept in a global variable. The lexical analyzer is called after a match of a terminal token, to keep the global token type always pointing to the next token (the look ahead token). The very first look ahead tokens is read by an initial call to lexicalAnalyzer() before any of the grammar's parsing functions are called.

Appendix C shows a simple parser of the new elements statements for creating the new subclass from the *hook* description grammar.

During parsing, we add information to the **Hook Table** to record the roles of the various identifiers for use in future phases (ie. identifier names, types and scope information).

### 5.2.4 Hook Table

The parser deals with tokens in different ways depending on their type. All token types represent terminal symbols defined in the grammar. The terminal symbol represents a class of syntactically equivalent tokens, and a nonterminal symbol stands for a class of syntactical alternatives for that phrase of the grammar. The Hook Table is used to store the symbols and the attributes associated with them, and in this sense it assumes the standard role of a symbol table in most compilers. There are two major interactions between the Hook Table and the rest of the **HookMaster** tool. First, symbols and related information must be added to the table by the parser. Secondly, information added to the table may need to be retrieved at some later time by the system.

Its complete class diagram is given in Figure 5.8. The Hook Table is made up of entries consisting of a name and its corresponding type. Entries in the table are built up in several stages. In lexical analysis, when a *hook* name is discovered, it is put into the cHookTable class according to its type. The cHookTable class will create an object of the cHook class. For an identifier, its name is discovered in lexical analysis although its type will not be determined until syntactic analysis.

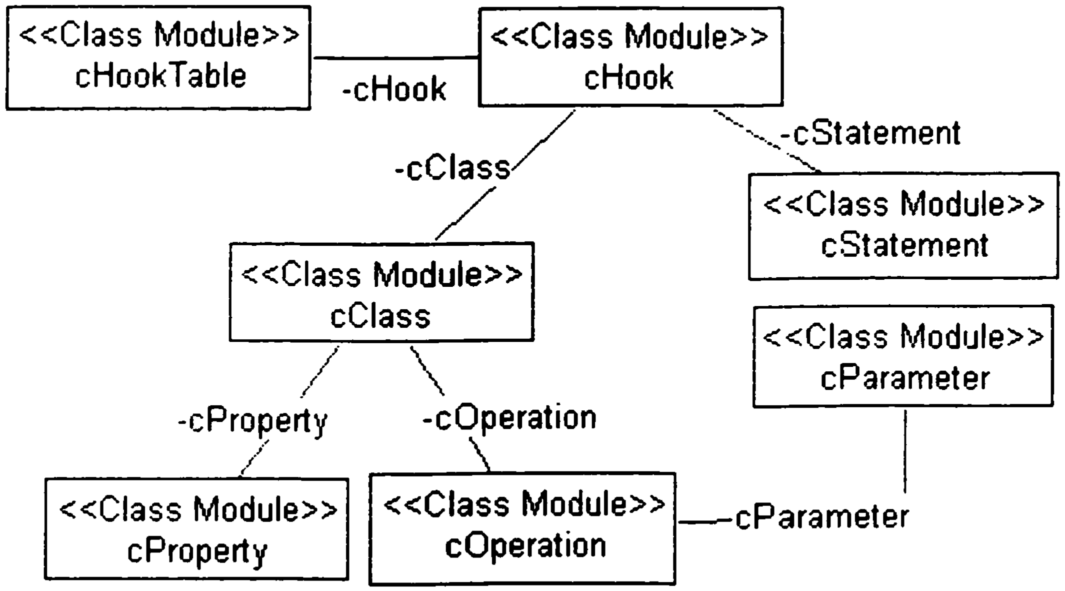


Figure 5.8: The class diagram of the Hook Table

### 5.2.5 Monitor-Controller

The Monitor-Controller modules were designed to support reuse and system evolution. New modules may be added to the existing tool by registering their interest in events. Similarly, one module may be replaced by another without affecting the interfaces of modules that implicitly depend on it. Figure 5.9 shows the class diagram for the Monitor-Controller.

The tool is configured as a collection of components (see Figure 4.2) written in Visual Basic 5.0 to interact with Rational Rose 98. To achieve a clear separation between these components, event broadcast is handled by a sepa-

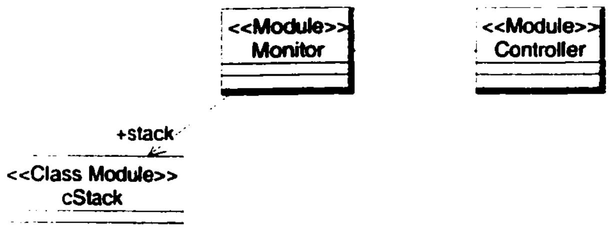


Figure 5.9: The class diagram of the Monitor

rate dispatcher process in the Monitor (see Figure B.7 in Appendix B) that communicates between components through function calls.

All interactions with Rational Rose 98 are grouped into the Controller module (see Figure B.8 in Appendix B), which interacts with the Monitor module as events are posted. The Monitor invokes various classes according to the different types of events. Any new module can be added by defining an object in the Monitor module, and it will be globally visible. Although this approach creates a significant amount of implicit coupling, it provides tremendous flexibility and allows the prototype to evolve quite easily. When some functionalities are not needed, we can simply cancel the related registration of the module corresponding to the functionalities.

When the user opens a framework along with *hooks* within the tool, the Monitor notifies the Controller module to invoke Rational Rose 98, and retrieve the *hook* descriptions from the Rational Rose 98 definition of the framework. Then, it invokes the Parser to parse all statements of the *hook* description, and to store the information into the Hook Table. When the user starts enacting a *hook*, the Monitor module executes the action according to the statement type by accessing the information from the Hook Table, and, as necessary, routing the information from the Hook Table to the Controller. The Controller module invoke the functionalities of Rational Rose 98. When a statement has been completed, the Monitor notifies the main window to highlight this step.

# Chapter 6

# Validation and Improvement of Prototype

We are in the fortunate position of being able to validate the prototype of the tool using the existing CSF(Client-Server framework) which has been developed by Garry Froehlich of the Software Engineering Research Lab of the University of Alberta [26]. The CSF is a communications framework that provides the basic infrastructure for developing small client-server or peer-to-peer programs and is suitable for student projects. CSF is implemented as a Java program that handles all of the communication over a TCP/IP connection and has some persistent storage capabilities as shown in Figure 3.1. The CSF was given to the students of CMPUT 401 to use, and to evaluate the *hook* notation in the spring and fall terms of 1999. The student project groups have successfully developed several applications using the framework and its *hooks*.

## 6.1 Using HookMaster with CSF

### 6.1.1 The CSF Motifs

The CSF allows any two objects that inherit from CommAwareObject to communicate with each other regardless of where they are on the network. These objects communicate by sending Message objects to each other. This means that any object within a client program can communicate with any object within a server program or even another client program. A Message object contains the information about its source and destination objects along with a Data object that can contain an arbitrary number of attributes and methods. The method of sending messages is very general, but has one restriction: the class of the Data object being sent must defined in both the client and the server programs.

The basic paradigm for the communications framework is that of an email service. CommAwareObjects create Messages which are placed in Outboxes. The Outbox sends the message to a MailServer which routes the message to the appropriate machine. The routing is done according to the address of the message(corresponding to an email address). A MailServer at the destination machine receives the message and further routes it to the appropriate Inbox. The Inbox notifies the receiver, which is also a CommAwareObject, that a message has arrived and the receiver can take any action necessary based on the message [26].

### 6.1.2 Validation for Hook Enactment

When a user selects the New CommAwareObject *hook* shown below, the communication *hook view* is displayed (see Figure 6.1). The changes window of the main window has been removed for simplicity. In this example, we are using the client server framework to create a simple internet-based simulation, such as a farming simulator. Users of the simulator enter a turn's worth of information using a client program or web-browser and then submit it to the server. The server then invokes the simulation engine and returns the results of the simulation to the user. Here we focus on the creating part of the server side. In order to create a server class which can communicate with any clients using CSF, the New CommAwareObject *hook* is used.

```
Name: New CommAwareObject

Requirement: An object needs to communicate across the network.

Type: Enabling Pattern

Area: Communication

Participants: NewCOA, Comm_Aware_Object (provided), Message

Uses: Handle Message

Preconditions: none

Changes:

// First create a new subclass of Comm_Aware_Object.

new subclass NewCOA of Comm_Aware_Object

repeat as necessary

        fill in Message.type // a string name of the message

// (corresponding to the type) that the COA

// should respond to.

        // invoke the Handle Message hook.

        Handle Message[NewCOA = NewCOA, Message = Message]
```

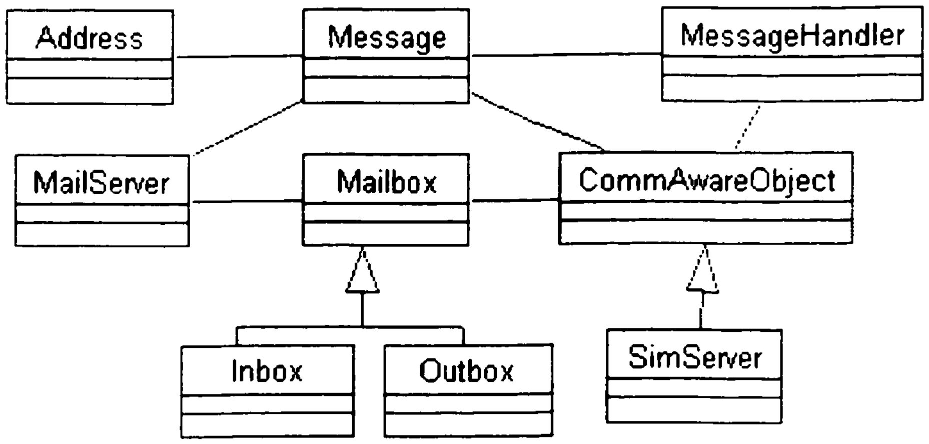


Figure 6.1: The New CommAwareObject Hook View

Along with the view, the text field of the main window is displayed. It contains the preconditions and changes sections of the *hook*. When the user is ready to start enacting the *hook*, he or she selects the enactment element within the main window. **HookMaster** checks the preconditions at the beginning of the enactment. Our example *hook* is an enabling *hook*, and it has no specific preconditions. The first step of the change statements is then enacted. This involves the creation of a new application subclass of the framework class CommAwareObject. NewCOA is a variable representing the new application class. The CreateClass dialog box is popped up to allow the user to input a subclass name in the role of NewCOA. SimServer is the subclass name used in the example shown in Figure 6.1. The next step is a repeat loop which simply designates that all of the steps inside of the repeat loop body should be repeated until the user is finished. The 'fill in' step brings up a dialog box that requests a string representing the message type. In our simulator example, we want to create a message for submitting the information for a turn to the servers, so the message type is given the name SubmitTurn (not shown) (Note: CommAwareObjects communicate through messages represented by the Message class, each of which has a unique type.). The second step within the loop invokes the Handle Message *hook*. To enact this *hook*, the current

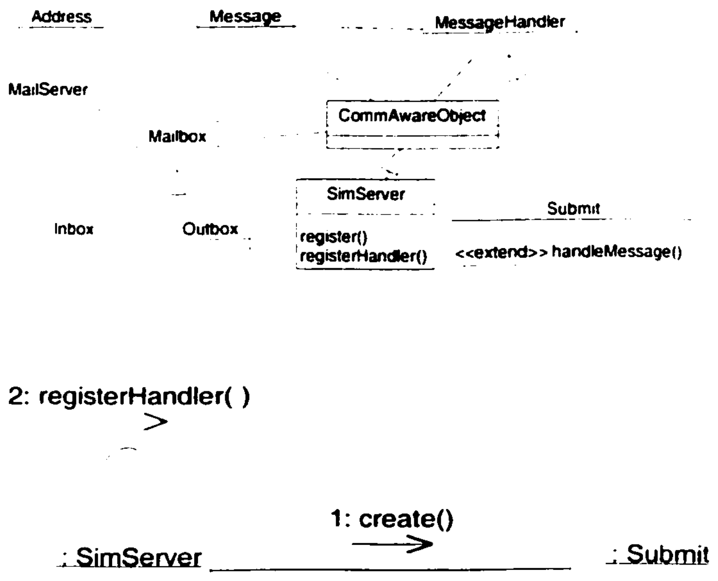


Figure 6.2: The Handle Message Hook View

*hook view* is suspended and a view for the Handle Message *hook* is opened.

```
Name: Handle Message

Requirement: When an object receives a message,

    it needs to respond to it in some way.

Type: Enabling Pattern

Area: Communication

Participants: Message, NewCOA, MessageHandler, NewMH

Uses: none

Preconditions:

NewCOA subclass of Comm_Aware_Object

Changes:
```

```
// First, create a new MessageHandler subclass.

new subclass NewMH of MessageHandler

// Specialize the handleMessage method and fill in the

// appropriate code.

// 'extends' infers that handleMessage must call its superclass

// ie. code 'super.handleMessage(m, coa);'

NewMH.handleMessage(Message m, CommAwareObject coa) extends

    MessageHandler.handleMessage(Message m, CommAwareObject coa)

// The next line is for 'hooking' up the handler.

// Registering can occur within the initialization method of the

// object.

// (the name 'register' is just a placeholder).

new operation NewCOA.register

NewCOA.register -> NewCOA.registerHandler(Message.type, NewMH)

Postconditions: none
```

The Handle Message *hook* has the same context as the previous *hook*, although a new window is opened on the screen. Before the *hook* can be enacted. the participants are mapped to the parameters given with the *hook* invocation from the New CommAwareObject *hook*. As shown in the *hook* call, the NewCOA participant(SimServer) is mapped to the NewCOA participant in Handle Message. The Message participant is similarly mapped to Message in Handle Message. The first step in the enactment of the *hook* is to check the precondition. In this case, there is only one, and it checks to see if NewCOA is a subclass of CommAwareObject. This indeed is the case, so enaction can proceed to the first change step.

The first step creates a new subclass NewMH of MessageHandler. NewMH is another application participant. In this case we are creating a handler for the SubmitTurn message, so NewMH corresponds to Submit in Figure 6.2. Next the handleMessage method on NewMH is filled in by the user. A NewOperation dialog box then asks the user for a new operation, or a link to an existing method, within NewCOA (SimServer) from which registration of the handler can occur. The last step in the changes section then invokes the callback method registerHandler within NewCOA(see the collaboration diagram of Figure 6.2).

The changes are then committed to the parent view, which is the New CommAwareObject *hook view*. Once the changes are committed, the New CommAwareObject *hook* enactment resumes and the repeat loop is invoked again, unless the user decides it is finished. To assist in this, a message box is popped up to ask the user if they wish to continue. For our example, we have no more messages to add, so the loop is stopped by clicking the "No" button of the message box and the changes are committed to the main view and the New CommAwareObject view.

After creating the sending and receiving objects, we send a message using the CSF. First of all, we set up the communication, that is, the sending object within the client must know the Address of the receiving object's Inbox, or list of addresses if broadcasting to more than one receiver. In order to create an Outbox, we use the *hook* New Outbox whose view is shown in Figure 6.3. After opening the view, **HookMaster** checks the precondition. It succeeds because the SimClient has been created. According to the first step of the changes section, the init operation called initialize is added into the SimClient class. Then, the tool can create the two invocations (NewCOA.init invokes addr and Outbox.Outbox(self, Address)) in the collaboration diagram for the next two steps as shown in Figure 6.3. Finally, the postcondition is checked.

```
Name: New Outbox

Requirement: An object needs to be able to send messages

     across the network

Type: Enabling Pattern

Area: Communication

Participants: NewCOA, Outbox (provided)

Uses: none

Preconditions:

NewCOA subclass of Comm_Aware_Object

Changes:

new operation NewCOA.init

// the Address and the Outbox should be created within

//the initialization method of NewCOA.

NewCOA.init -> addr = new Address(IP, port, name)

NewCOA.init -> Outbox.Outbox(self, Address)

Postconditions: NewCOA.Outbox
```

For clarity, the sending object is assumed to be in a client and the receiving object is assumed to be in a server. A CommAwareObject can have multiple Inboxes (for example, having one Inbox for low priority messages and one for high priority messages). Each Inbox has its own address which consists of an IP address of the machine it is on, the port number of the MailServer on that machine, a name. The receiving object creates an Inbox and assigns it an Address. The Inbox registers itself with the server's MailServer and waits for incoming Messages. To create an Inbox, we use the New Inbox *hook* whose

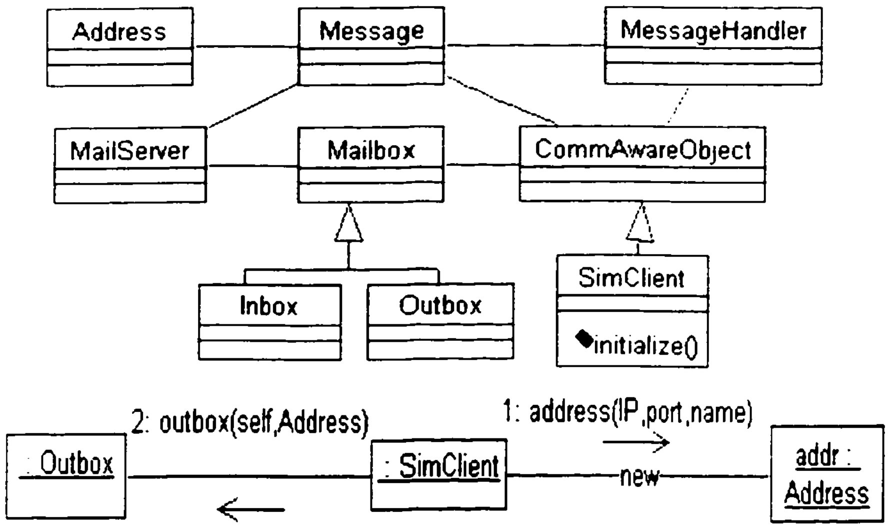


Figure 6.3: The New Outbox Hook View

view is shown in Figure 6.4.

```
Name: New Inbox

Requrement: An object needs to be able to receive messages

    across the network

Type: Enabling Pattern

Area: Communication

Participants: NewCOA, Inbox (provided)

Uses: none

Preconditions:

NewCOA subclass of Comm_Aware_Object

Changes:

new operation NewCOA.init
```

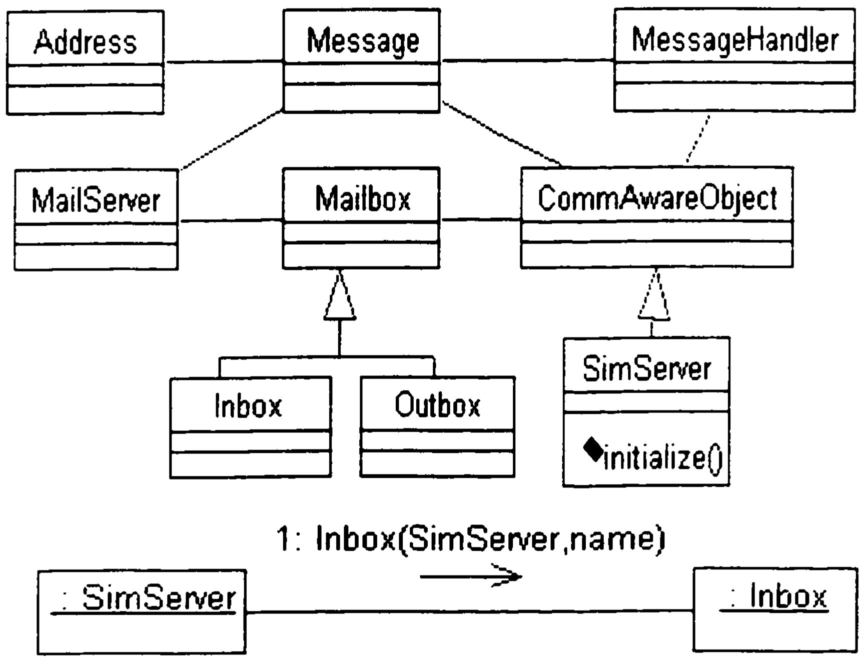


Figure 6.4: The New Inbox Hook View

```
fill in name: String

NewCOA.init -> Inbox.Inbox(NewCOA, name)

Postconditions: NewCOA.Inbox
```

Messages are normally sent asynchronously. That is, the sender does not block while waiting for a reply. However, in some cases, the sender may require a reply before continuing, and the message must be sent synchronously. A message consists of the return address, the to address, the message type, and a Data object that contains the information of the message. In order to send messages, the sending object creates a NewCOA(SimClient) and gives it a Message with the receiving object's address. SimClient grabs a temporary Address for itself and registers itself with client's MailServer. SimClient contacts the client's MailServer and asks it to send the message. Instead of immediately returning control to the sending object, it blocks as it waits for a return message through it own Address. The server's MailServer receives a Message through a socket. The MailServer checks the last part of the Address to which the Message is to be sent and then searches for an Inbox that matches the Address. If an Inbox is found, the Message is placed in the Inbox. The Inbox notifies the receiving object that it received a Message. The receiving object checks the type of the Message and then invokes the appropriate handler for that Message (see the Handle Message *hook* to see how to set up and use the handlers). Then, the receiving message sends a message back as in Sending a Message. SimClient unlocks and passes the returned Message back to the sending object. If the return Message is not received in a certain amount of time, SimClient sends a timeout exception back to the sending object instead.

In order to easily send messages using CSF, the CSF builder provides the Send Message *hook* to detail how to send message as follows. The Figure 6.5 shows the evolution after the changes section is completed.

```
Name: Send Message

Requirement: An object needs to send a message to a single

     object on a remote machine.

Type: Enabling Pattern

Area: Communication

Participants: NewCOA, Inbox, Outbox, CommAwareObject

Uses: none

Preconditions:

NewCOA subclass of Comm_Aware_Object

Outbox NewCOA.out exists

Inbox NewCOA.in exists
```

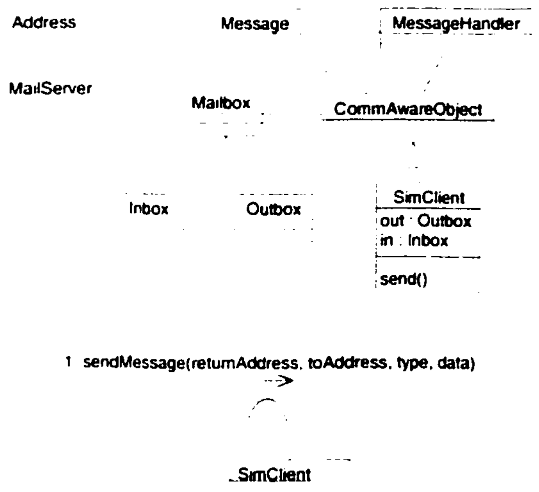


Figure 6.5: The Send Message Hook View

```
Changes:

new operation NewCOA.send

fill in toAddress: Address, type: String, data: Data

fill in returnAddress = NewCOA.in.getAddress()

NewCOA.send -> NewCOA.sendMessage(returnAddress, toAddress,

type, data)

Postconditions: none
```

### 6.1.3 Limitations and Possible Improvement

The **HookMaster** prototype has some limitations at the current stage.

Currently, all of the *hook views* must be nested under the logical view of Rational Rose 98, since Rational Rose 98 does not allow the developer to create their own views independently. So, when the user opens a model file from **HookMaster**, not only are the *hook views* loaded, but also all other kinds of views of Rational Rose 98 show up on the browser window, such as the component view, use case view and so on. Ideally, the interface containing the model only shows the user the *hook views* of the model. Meanwhile, the *class diagram* and the *collaboration diagram* within a *hook view* can be opened up at the same time when the user selects the *hook* from the main window. However, because of a technical limitation of Rational Rose 98, only one of either the *class diagram* or the *collaboration diagram* of the *hook view* can be opened on the interface at one time. If the *class diagram* is opened, the corresponding *collaboration diagram* can be opened manually by the user clicking the *collaboration diagram* icon in the browser window. This, however, is a cumbersome approach.

A second major limitation of the tool prototype is the poor performance provided by the animation feature when the user starts replaying. The reasons are because we have to store each step of enacting the *hook* as a new model file. Usually, the size of the model file is quite big; when the tool plays each step back, it has to load the corresponding model file. Ideally, the prototype should have the functionality that play back each step of the changes section step by step. To overcome this limitation, we have tried simulating the "find" functionality of Rational Rose 98, which is basically the functionality of searching by keywords. Initially, we expect to attach a distinct tag to the data generated by each step and retrieve the data by the tag whenever needed. Unfortunately, the functionality is not provided by REI. Currently, we use colors to differentiate the data of the final applications when replaying.

Finally, the tool prototype can not reflect all types of change statements in Rational Rose 98. Specifically, for the add/remove codes statement from within a framework class, the tool is not able to properly represent the code body added or removed because the UML class notation only presents the three aspects: class name, attributes names and operation names, but not the code body.

# Chapter 7

# Conclusions and Future Work

By focusing on the intended use of a framework, rather than the design of the framework, *hooks* are able to provide a more structured and uniform specification than textual narrative for affecting the changes needed to develop an application. Early experience [16] in using *hooks* strongly suggests that tool support for their enactment is important to the overall effectiveness in application software development.

In the thesis, we have demonstrated that a *hook* tool can be constructed to aid in the use of object-oriented frameworks using the same basic ideas that exist in graphical user interface builders. We have connected *hooks* to the design of the framework by creating *hook views* in extended UML notations. We have added explicit support for the notation of *hooks* to UML. This *hook* tool begins with the main view of the framework, which is evolved into applications through the tool-supported enactment of *hooks*. The overall view contains all of the classes of the framework, along with any additional classes that are added by application developers. The tool supports the assumption that the original design or implementation of the framework should not be modified. This assumption preserves the benefits of maintaining a common code base among a family of applications.

A *hook view* contains a subset of the overall view of the framework: both *class* and *collaboration diagrams* are contained within the *hook view*. The view forms the context for the *hook* which includes the participants of the *hook* and related classes. *Hook views* are created by the framework builders rather than generated dynamically since determining the complete context of a *hook* is difficult and requires the knowledge of the framework builder. *Hooks* that share the same context are defined with the same view.

Within a *hook view*, application developers can enact the changes defined by the *hook*, semi-automatically with the tool's help. A key aspect of the enactment is a parser we have developed that supports the interpretation of the statements within the change section of a *hook*. The tool interactively processes the activities contained in the *hook*, performing as much as it can automatically and requesting any information from the developer whenever necessary.

To accomplish this requires some work on behalf of the framework builders. Once the framework code base is stable, they must reverse the source code into UML diagrams, and create the main and *hook views*. Both *hooks* and the **HookMaster** tool are designed to allow maintainers to easily add new or modify existing *hooks* in the framework.

In the future we would like to increase support for application developers and framework maintainers. Specifically, there are two aspects involved; one is support for generating Java code, the other involves augmenting the tool with more information about the framework.

With respect to the first aspect, we are planning to integrate **HookMaster** with another tool called **JavaMaster** using OLE so that it can manipulate code directly. When the user enacts the *hook* with the tool, it will interact with **JavaMaster** to generate new Java code according to the changes prescribed in the change statements of the *hook* descriptions.

Secondly, design diagrams and *hook views* should not be the only means of presenting the framework. Other information such as *examples* and *use cases* give additional information to users to aid in application development. They are valuable when first learning to use the framework, or learning to use it in a different way. Users typically grasp concrete *examples* more quickly than abstract descriptions. *Examples* are applications that illustrate the use of the framework through *hooks*. *Use cases* describe typical scenarios, which link to the *hooks* that implement the scenario. Normally, an example can contain multiple *use cases*.

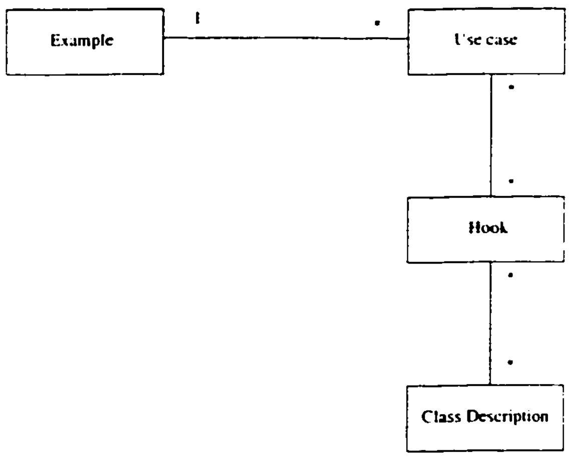


Figure 7.1: Entity-Relation Diagram for Framework Information

Our goal is to have a system that maintains a fully linked repository of framework information as shown in Figure 7.1 so that users can easily find the relevant information from any point in the repository. For example, given a certain *hook* the system can find all the *examples* and the associated *use cases* for the *hook*.

# Bibliography

[1] K. Beck and R. Johnson. *Patterns Generate architectures* The Proceedings of ECOOP'94. Bologna. Italy. 1994, 139-149.

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

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

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

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

[6] H. Eriksson and M. Penker. *UML ToolKit* John Wiley and Sons, Inc., 1998, 217-254.

[7] G. Booch, I. Jacobson and J. Rumbaugh. *The Unified Modeling Language Reference Manual* Addison-Wesley, 1999, 1-62, 85-92, 101-112.

[8] *Rational Rose 98 Extensibility Reference Manual* Rational Co., 1998.

[9] *Borland Delphi for Windows.* Borland International, Inc., Scotts Valley, CA, 1995, 168-189.

[10] 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.

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

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

[13] G. Booch. *Object-Oriented Analysis and Design with Applications.* The Benjamin/Cummings Publishing Co., CA, 1994, 344-368.

[14] Taligent. *The Power of Frameworks.* Addison-Wesley Publishing Co., MA, 1995, 155-180.

[15] W. Pree. *Deisn Patterns for Object-Oriented Software Development.* Addison-Wesley Publishing Co., MA, 1995, 233-261.

[16] G. Froehlich, H.J. Hoover, L. Liu and P. Sorenson. Hooking into Object-Oriented Application Frameworks. In *Proceedings of the 1997 International Conference on Software Engineering* (Boston, Mass, 1997), 491-501.

[17] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995, 290-326.

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

[19] H. Hueni. R. Johnson and R. Engel. *A Framework for Network Protocol Software.*. In Proceedings of OOPSLA'95. Austin. TX, 1995. 81-96.

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

[21] 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-September 1988), 26-49.

[22] 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'Avancement des Sciences. Montreal. Canada. 1994, 44-61.

[23] 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.

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

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

[26] G. Froehlich. *Using the Communications Framework* The Software Engineering Research Lab, Dept. of Computing Science, U of Alberta, 1998, 1-18.

[27] Ben Shneiderman. *Designing the User Interface* Third Editon, Addison-Wesley, 1998, 416-466.

[28] M. Shaw, D. Garlan. *Software Architecture* Alan Apt, 1996, 97-212.

[29] A. Egyed, P. B. Kruchten. *Rose/Architect: a tool to visualize architecture* IEEE, March 1999, 58-72.

[30] T. Pittman, J. Peters. *Art of Compiler Design, The: Theory and Practice* Prentice Hall, May 1997, Chapter 4.

[31] J. Rekers. *A parser generator for finitely ambiguous context-free grammars* In Proceedings of Computing Science in the Netherlands, CSN'87, Amsterdam, 1987, 113-1240.

[32] J. Rekers and W. Koorn. *Substring parsing arbitrary context-free grammars* SIGPLAN Notices, 26(5), 1991, 59-66.

# Appendix A

# Grammar for Hook Descriptions

$\langle$hook$\rangle$ ::= $\langle$name$\rangle$

$\langle$requirement$\rangle$

$\langle$type$\rangle$

$\langle$area$\rangle$

[ $\langle$uses$\rangle$ ]

$\langle$participants$\rangle$

$\langle$changes$\rangle$

[$\langle$preconditions$\rangle$]

[$\langle$postconditions$\rangle$]

[$\langle$comments$\rangle$]

$\langle$name$\rangle$ ::= Name: $\langle$string$\rangle$

$\langle$requirement$\rangle$ ::= Requirement: $\langle$string$\rangle$

$\langle$type$\rangle$ ::= Type: $\langle$method$\rangle$, $\langle$level$\rangle$

$\langle$method$\rangle$ ::= enabling | adding | replacing | augmenting | disabling

$\langle$level$\rangle$ ::= $\langle$option$\rangle$ | $\langle$pattern$\rangle$ | open

$\langle$option$\rangle$ ::= single option | multi-option

$\langle$pattern$\rangle$ ::= parameter pattern | collaboration pattern | pattern

⟨area⟩ ::= Area: ⟨string⟩ [, ... ⟨string⟩]

⟨participants⟩ ::= Participants: ⟨identifier⟩ [⟨type⟩] [⟨style⟩] [, ...⟨identifier⟩ [⟨type⟩] [⟨style⟩] ]

⟨type⟩ ::= set of ⟨identifier⟩ [, .., ⟨identifier⟩] |sequence of ⟨identifier⟩ [, ... ⟨identifier⟩]

⟨style⟩ ::= new | exists

⟨uses⟩ ::= Uses: ⟨hook name⟩ [, .., ⟨hook name⟩]

⟨changes⟩ ::= ⟨statement⟩ [, .., ⟨statement⟩]⟨statement⟩ ::= ⟨loop statement⟩ | ⟨hook statement⟩ |⟨new element statement⟩ | ⟨method statement⟩ |⟨modify statement⟩ | ⟨participant statement⟩ |⟨option statement⟩ | ⟨behavior statement⟩

⟨loop statement⟩ ::= ⟨loop id⟩ ⟨statement⟩ [.. ⟨statement⟩]

⟨loop id⟩ ::= repeat [as necessary] | forall ⟨var⟩ in ⟨set⟩

⟨hook statement⟩ ::= ⟨identifier⟩ = ⟨hook name⟩ "[" ⟨identifier⟩ "]" |⟨hook name⟩ "[" ⟨identifier⟩ = ⟨rhs⟩ [, .., ⟨identifier⟩ = ⟨rhs⟩] "]"

⟨new element statement⟩ ::= [new] subclass ⟨identifier⟩ of ⟨identifier⟩ |[new] property ⟨qualified identifier⟩ ⟨whereclause⟩ |[new] operation ⟨qualified identifier⟩

⟨whereclause⟩ ::= read of ⟨identifier⟩ maps from [set of] ⟨qualified identifier⟩ |write of ⟨identifier⟩ maps into [set of] ⟨qualified identifier⟩

⟨method statement⟩ ::= ⟨qualified identifier⟩ ⟨method operation⟩⟨qualified identifier⟩ [⟨return expression⟩] |⟨qualified identifier⟩ ⟨return expression⟩

⟨method operation⟩ ::= copies | specializes | overrides | extends

⟨return expression⟩ ::= returns ⟨string⟩ |returns [set of | sequence of] ⟨rhs⟩

⟨modify statement⟩ ::= remove code '⟨string⟩' [, .., '⟨string⟩'] |replace '⟨string⟩' with '⟨string⟩'

⟨participant statement⟩ ::= fill in ⟨identifier⟩ [, .., ⟨identifier⟩] |⟨identifier⟩ add ⟨set⟩

⟨option statement⟩ ::= choose ⟨identifier⟩ from ⟨set⟩

⟨behavior statement⟩ ::=synchronization ( ⟨qualified identifier⟩, ⟨qualified identifier⟩ [,.., ⟨qualified identifier⟩ ) [in ⟨qualified identifier⟩][provided]| [control flow] ⟨qualified expression⟩ -⟩ [⟨read/write⟩]⟨qualified expression⟩ [provided]

⟨read/write⟩ ::= read | write | read and write

⟨rhs⟩ ::= ⟨identifier⟩ |⟨set⟩ |instance of ⟨var⟩ [[of | with] ⟨attribute⟩ ⟨var⟩] |⟨loop id⟩ ⟨rhs⟩

⟨set⟩ ::= ⟨var⟩ | ( ⟨identifier⟩. ⟨identifier⟩ [, .., ⟨identifier⟩] )

⟨var⟩ ::= ⟨identifier⟩ | ⟨qualified identifier⟩

⟨attribute⟩ ::= ⟨identifier⟩

⟨qualified identifier⟩ ::= ⟨identifier⟩.⟨identifier⟩

⟨preconditions⟩ ::= Constraints: ⟨string⟩ [, .., ⟨string⟩]

⟨postconditions⟩ ::= Constraints: ⟨string⟩ [, .., ⟨string⟩]

⟨comments⟩ ::= Comments: ⟨string⟩

# Appendix B

# Class Diagrams

This section lists all classes of the tool prototype.

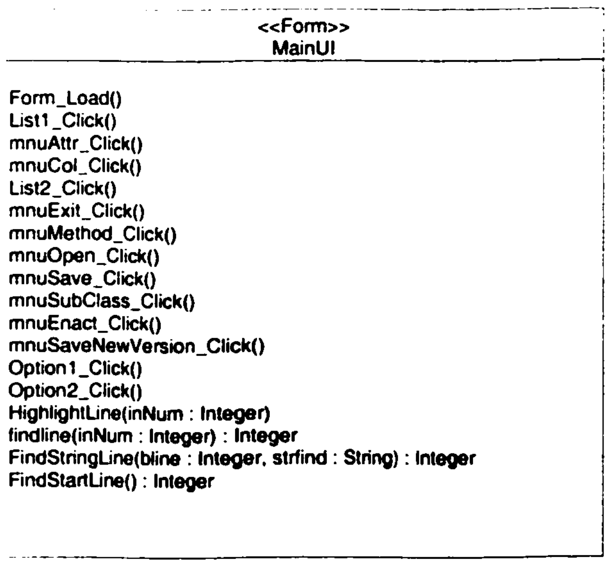


Figure B.1: The MainUI class

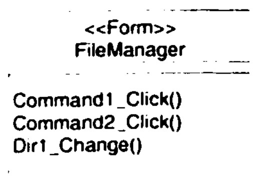


Figure B.2: The FileManager class

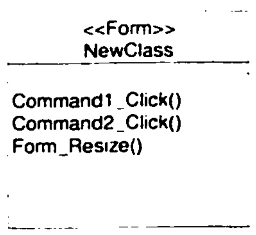


Figure B.3: The NewClass class

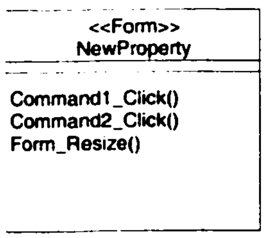


Figure B.4: The NewProperty class

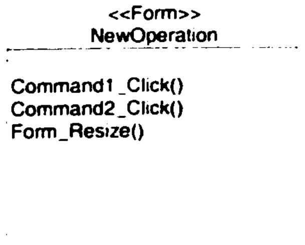


Figure B.5: The NewOperation class

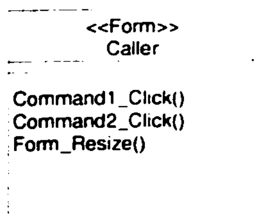


Figure B.6: The Caller class

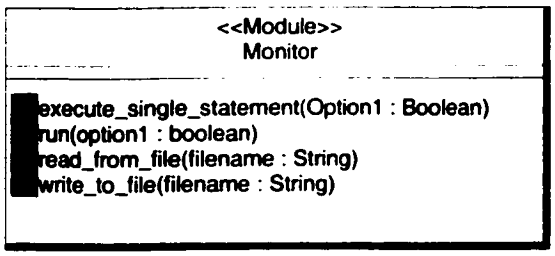
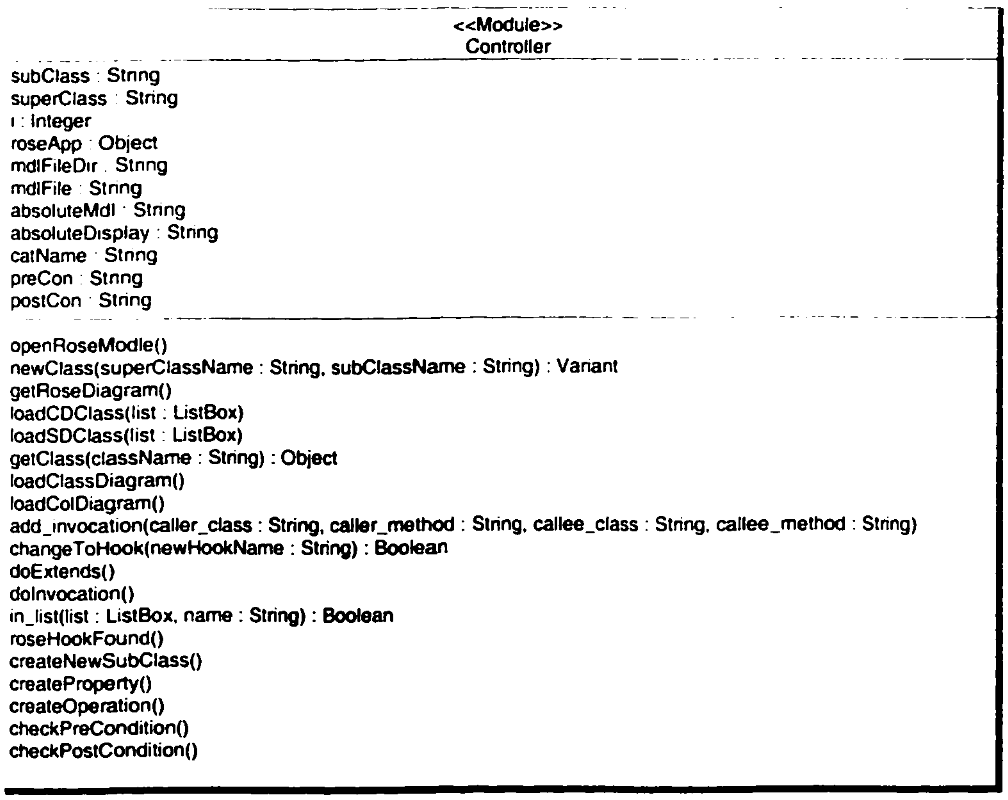

<<Module>>
Monitor
execute_single_statement(Option1 : Boolean)
run(option1 : boolean)
read_from_file(filename : String)
write_to_file(filename : String)

Figure B.7: The Monitor class



```
<<Module>>
Controller

subClass : String
superClass : String
i : Integer
roseApp : Object
mdlFileDir : String
mdlFile : String
absoluteMdl : String
absoluteDisplay : String
catName : String
preCon : String
postCon : String

openRoseModle()
newClass(superClassName : String, subClassName : String) : Variant
getRoseDiagram()
loadCDClass(list : ListBox)
loadSDClass(list : ListBox)
getClass(className : String) : Object
loadClassDiagram()
loadColDiagram()
add_invocation(caller_class : String, caller_method : String, callee_class : String, callee_method : String)
changeToHook(newHookName : String) : Boolean
doExtends()
doInvocation()
in_list(list : ListBox, name : String) : Boolean
roseHookFound()
createNewSubClass()
createProperty()
createOperation()
checkPreCondition()
checkPostCondition()
```

Figure B.8: The Controller class

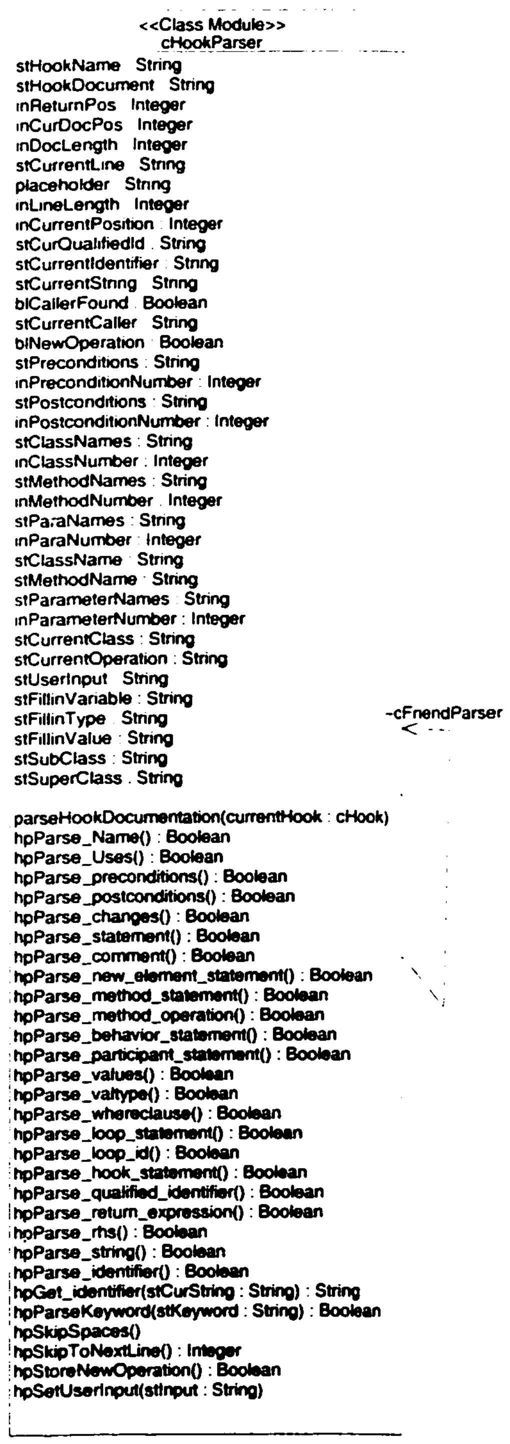
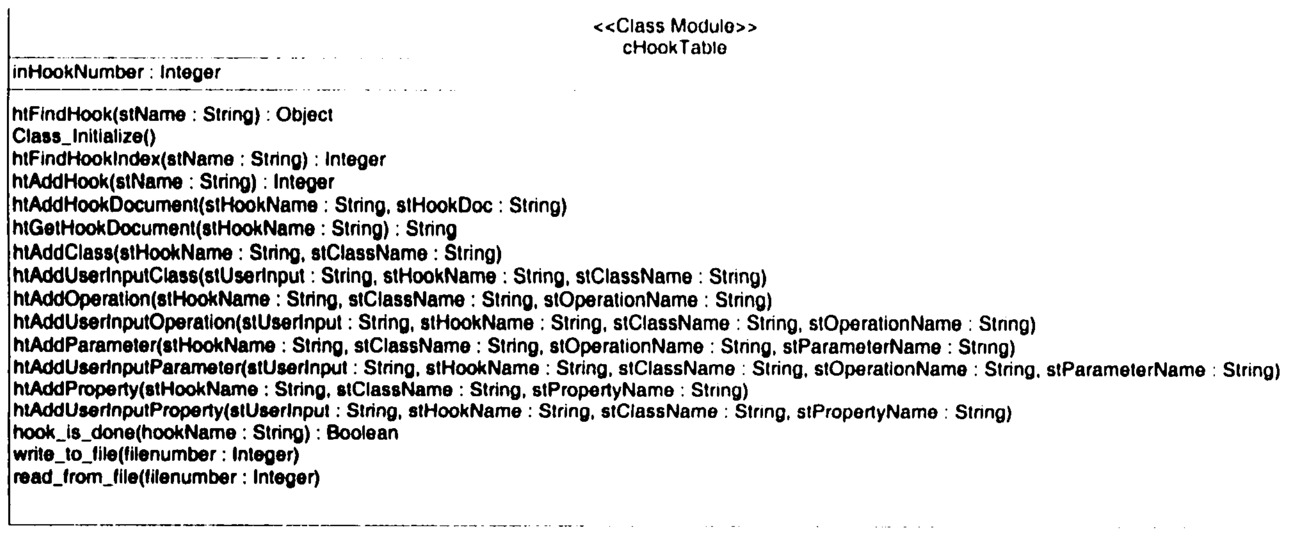
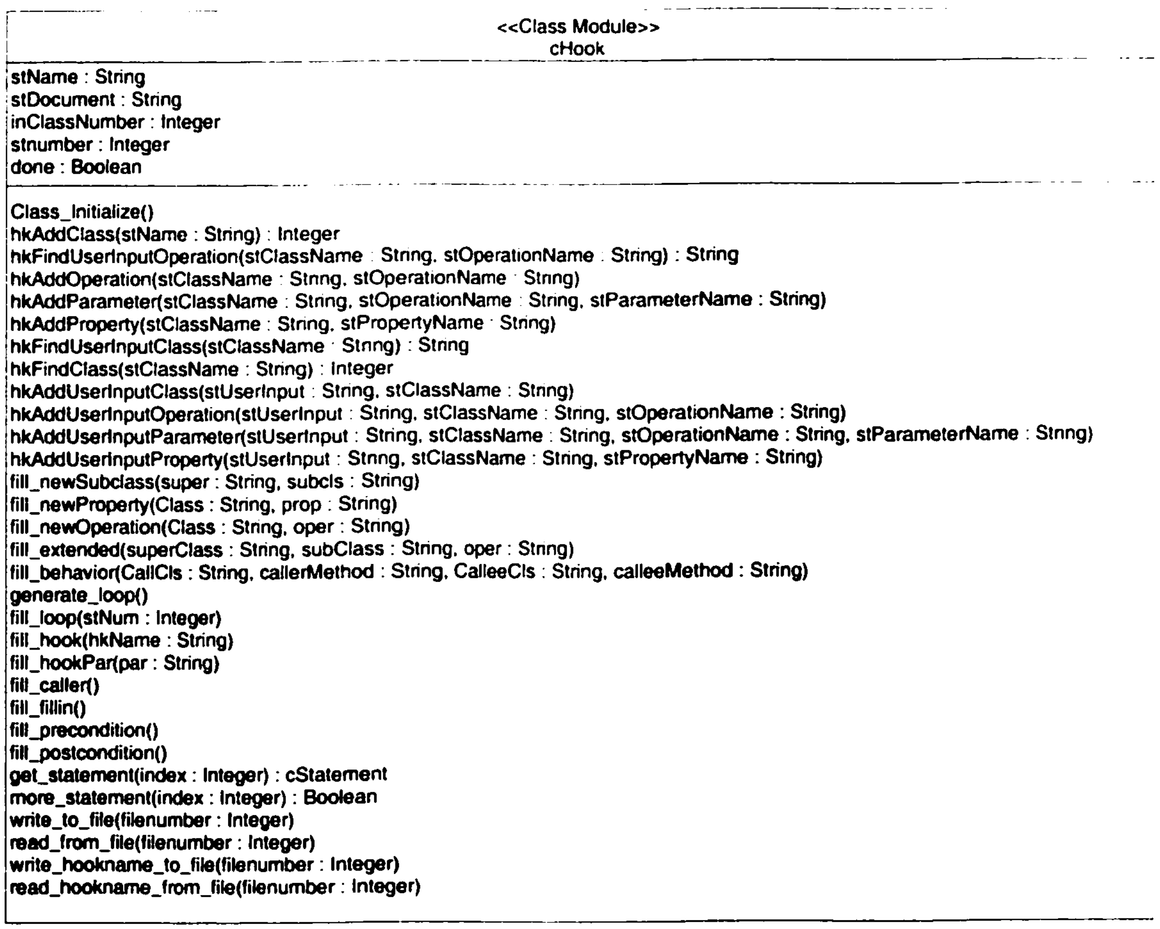

```
<<Class Module>>
cHookParser
stHookName  String
stHookDocument  String
inReturnPos  Integer
inCurDocPos  Integer
inDocLength  Integer
stCurrentLine  String
placeholder  String
inLineLength  Integer
inCurrentPosition : Integer
stCurQualifiedId . String
stCurrentIdentifier : String
stCurrentString  String
blCallerFound . Boolean
stCurrentCaller  String
blNewOperation  Boolean
stPreconditions : String
inPreconditionNumber : Integer
stPostconditions : String
inPostconditionNumber : Integer
stClassNames : String
inClassNumber : Integer
stMethodNames : String
inMethodNumber  Integer
stParaNames : String
inParaNumber  Integer
stClassName  String
stMethodName  String
stParameterNames  String
inParameterNumber : Integer
stCurrentClass : String
stCurrentOperation : String
stUserInput  String
stFillinVariable : String
stFillinType  String
stFillinValue  String
stSubClass : String
stSuperClass . String

parseHookDocumentation(currentHook : cHook)
hpParse_Name() : Boolean
hpParse_Uses() : Boolean
hpParse_preconditions() : Boolean
hpParse_postconditions() : Boolean
hpParse_changes() : Boolean
hpParse_statement() : Boolean
hpParse_comment() : Boolean
hpParse_new_element_statement() : Boolean
hpParse_method_statement() : Boolean
hpParse_method_operation() : Boolean
hpParse_behavior_statement() : Boolean
hpParse_participant_statement() : Boolean
hpParse_values() : Boolean
hpParse_valtype() : Boolean
hpParse_whereclause() : Boolean
hpParse_loop_statement() : Boolean
hpParse_loop_id() : Boolean
hpParse_hook_statement() : Boolean
hpParse_qualified_identifier() : Boolean
hpParse_return_expression() : Boolean
hpParse_rhs() : Boolean
hpParse_string() : Boolean
hpParse_identifier() : Boolean
hpGet_identifier(stCurString : String) : String
hpParseKeyword(stKeyword : String) : Boolean
hpSkipSpaces()
hpSkipToNextLine() : Integer
hpStoreNewOperation() : Boolean
hpSetUserInput(stInput : String)
```

Figure B.9: The cParser class



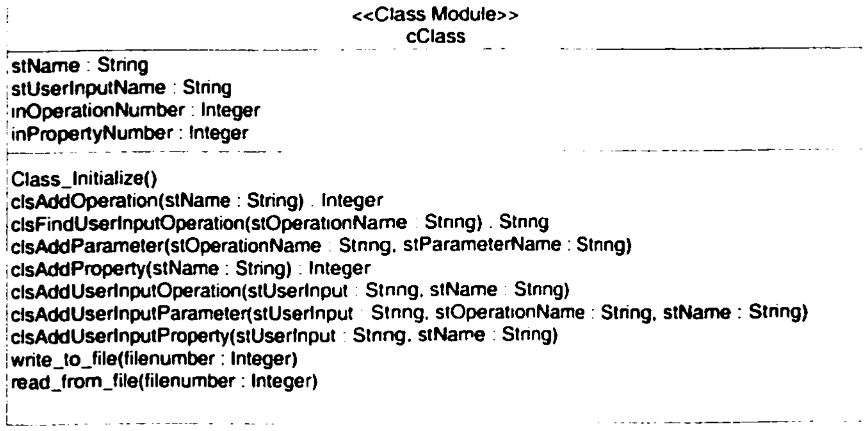
```
<<Class Module>>
cHookTable
inHookNumber : Integer
htFindHook(stName : String) : Object
Class_Initialize()
htFindHookIndex(stName : String) : Integer
htAddHook(stName : String) : Integer
htAddHookDocument(stHookName : String, stHookDoc : String)
htGetHookDocument(stHookName : String) : String
htAddClass(stHookName : String, stClassName : String)
htAddUserInputClass(stUserInput : String, stHookName : String, stClassName : String)
htAddOperation(stHookName : String, stClassName : String, stOperationName : String)
htAddUserInputOperation(stUserInput : String, stHookName : String, stClassName : String, stOperationName : String)
htAddParameter(stHookName : String, stClassName : String, stOperationName : String, stParameterName : String)
htAddUserInputParameter(stUserInput : String, stHookName : String, stClassName : String, stOperationName : String, stParameterName : String)
htAddProperty(stHookName : String, stClassName : String, stPropertyName : String)
htAddUserInputProperty(stUserInput : String, stHookName : String, stClassName : String, stPropertyName : String)
hook_is_done(hookName : String) : Boolean
write_to_file(filenumber : Integer)
read_from_file(filenumber : Integer)
```

Figure B.10: The cHookTable class



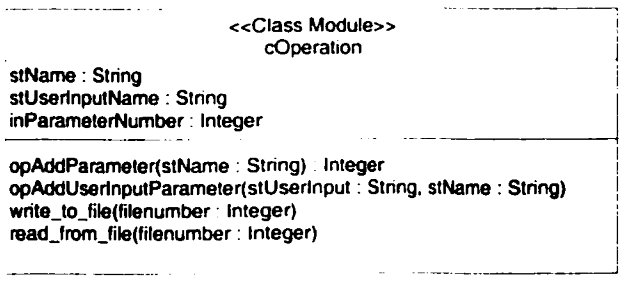
```
<<Class Module>>
cHook
stName : String
stDocument : String
inClassNumber : Integer
stnumber : Integer
done : Boolean

Class_Initialize()
hkAddClass(stName : String) : Integer
hkFindUserInputOperation(stClassName : String, stOperationName : String) : String
hkAddOperation(stClassName : String, stOperationName : String)
hkAddParameter(stClassName : String, stOperationName : String, stParameterName : String)
hkAddProperty(stClassName : String, stPropertyName : String)
hkFindUserInputClass(stClassName : String) : String
hkFindClass(stClassName : String) : Integer
hkAddUserInputClass(stUserInput : String, stClassName : String)
hkAddUserInputOperation(stUserInput : String, stClassName : String, stOperationName : String)
hkAddUserInputParameter(stUserInput : String, stClassName : String, stOperationName : String, stParameterName : String)
hkAddUserInputProperty(stUserInput : String, stClassName : String, stPropertyName : String)
fill_newSubclass(super : String, subcls : String)
fill_newProperty(Class : String, prop : String)
fill_newOperation(Class : String, oper : String)
fill_extended(superClass : String, subClass : String, oper : String)
fill_behavior(CallCls : String, callerMethod : String, CalleeCls : String, calleeMethod : String)
generate_loop()
fill_loop(stNum : Integer)
fill_hook(hkName : String)
fill_hookPar(par : String)
fill_caller()
fill_fillin()
fill_precondition()
fill_postcondition()
get_statement(index : Integer) : cStatement
more_statement(index : Integer) : Boolean
write_to_file(filenumber : Integer)
read_from_file(filenumber : Integer)
write_hookname_to_file(filenumber : Integer)
read_hookname_from_file(filenumber : Integer)
```

Figure B.11: The cHook class



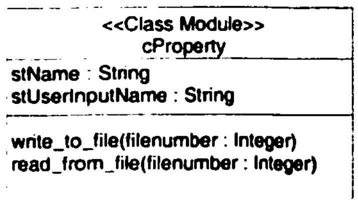
<<Class Module>>
cClass
stName : String
stUserInputName : String
inOperationNumber : Integer
inPropertyNumber : Integer
Class_Initialize()
clsAddOperation(stName : String) : Integer
clsFindUserInputOperation(stOperationName : String) : String
clsAddParameter(stOperationName : String, stParameterName : String)
clsAddProperty(stName : String) : Integer
clsAddUserInputOperation(stUserInput : String, stName : String)
clsAddUserInputParameter(stUserInput : String, stOperationName : String, stName : String)
clsAddUserInputProperty(stUserInput : String, stName : String)
write_to_file(filenumber : Integer)
read_from_file(filenumber : Integer)

Figure B.12: The cClass class



<<Class Module>>
cOperation
stName : String
stUserInputName : String
inParameterNumber : Integer
opAddParameter(stName : String) : Integer
opAddUserInputParameter(stUserInput : String, stName : String)
write_to_file(filenumber : Integer)
read_from_file(filenumber : Integer)

Figure B.13: The cOperation class



<<Class Module>>
cProperty
stName : String
stUserInputName : String
write_to_file(filenumber : Integer)
read_from_file(filenumber : Integer)

Figure B.14: The cProperty class

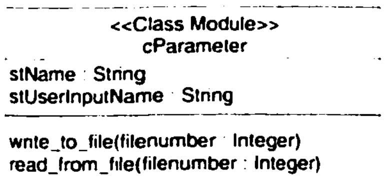

<<Class Module>>
cParameter
stName : String
stUserInputName : String
write_to_file(filenumber : Integer)
read_from_file(filenumber : Integer)

Figure B.15: The cParameter class

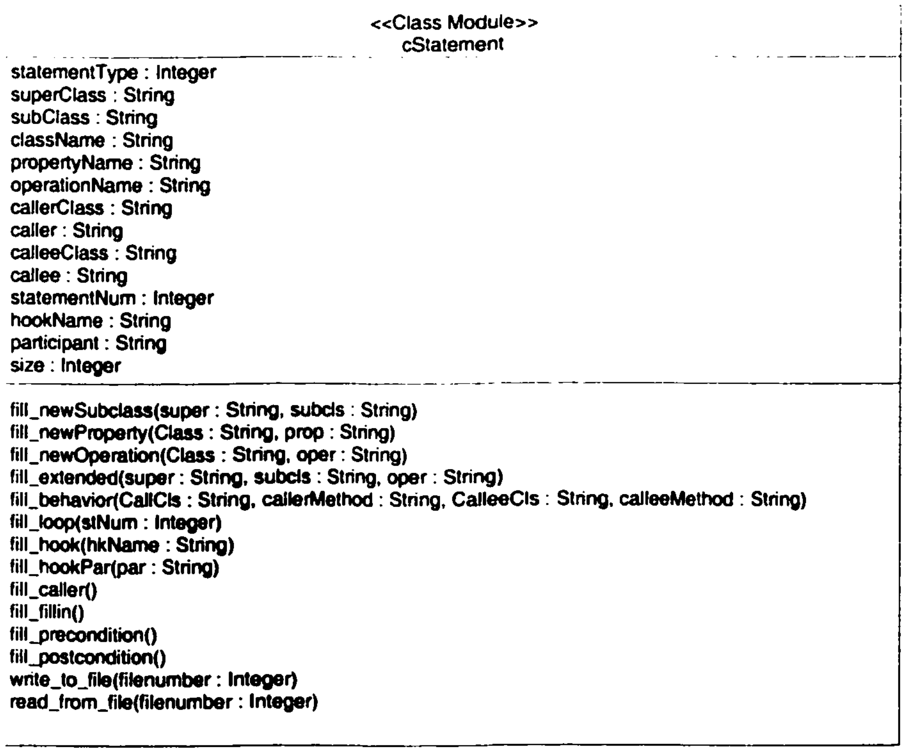
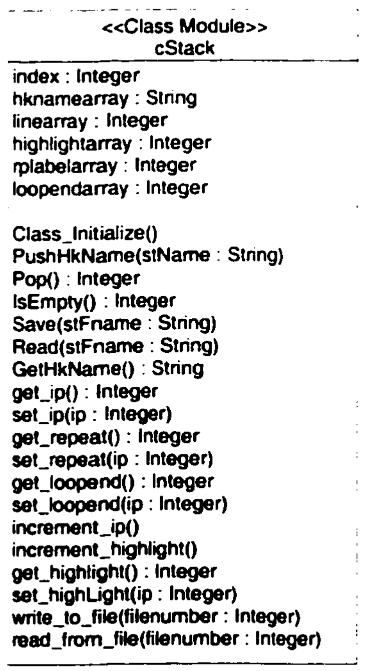

<<Class Module>>
cStatement
statementType : Integer
superClass : String
subClass : String
className : String
propertyName : String
operationName : String
callerClass : String
caller : String
calleeClass : String
callee : String
statementNum : Integer
hookName : String
participant : String
size : Integer
fill_newSubclass(super : String, subcls : String)
fill_newProperty(Class : String, prop : String)
fill_newOperation(Class : String, oper : String)
fill_extended(super : String, subcls : String, oper : String)
fill_behavior(CallCls : String, callerMethod : String, CalleeCls : String, calleeMethod : String)
fill_loop(stNum : Integer)
fill_hook(hkName : String)
fill_hookPar(par : String)
fill_caller()
fill_fillin()
fill_precondition()
fill_postcondition()
write_to_file(filenumber : Integer)
read_from_file(filenumber : Integer)

Figure B.16: The cStatement class



```
<<Class Module>>
cStack
index : Integer
hknamearray : String
linearray : Integer
highlightarray : Integer
rplabelarray : Integer
loopendarray : Integer

Class_Initialize()
PushHkName(stName : String)
Pop() : Integer
IsEmpty() : Integer
Save(stFname : String)
Read(stFname : String)
GetHkName() : String
get_ip() : Integer
set_ip(ip : Integer)
get_repeat() : Integer
set_repeat(ip : Integer)
get_loopend() : Integer
set_loopend(ip : Integer)
increment_ip()
increment_highlight()
get_highlight() : Integer
set_highLight(ip : Integer)
write_to_file(filenumber : Integer)
read_from_file(filenumber : Integer)
```

Figure B.17: The cStack class

# Appendix C

# A simple parser of the new element statement for creating the new subclass from the *hook* description grammar

```
hpParse-changes:
        hpParse-changes = False
        if hpParse-statement = True then
          hpParse-changes = True
          * to get "⟨statement⟩"
    hpParse-Statement:
        hpParse-statement = False
        if new-element-statement = True then
          hpParse-statement = True
        * to get "⟨new element statement⟩"
      Else
        * to parse the other statements
new-element-statement:
      new-element-statement = False
      if the "subclass" keyword found then
        if hpParse-identifier = True then
          * to get "⟨the subclass identifier⟩"
          if the "of" found then
            if hpParse-identifier = True then
            new-element-statement = True
            * to get "⟨the superclass identifier⟩"
hpParse-identifier:
      hpParse-identifier = False
      * Skip the character of non-digit and non-letter
      getIdentifierString()
```