INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Automation of Object Behavior Validation

By

James Hyjek

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

Master of Computer Science

**Ottawa-Carleton Institute for Computer Science
School of Computer Science**

**Carleton University
Ottawa, Ontario
18/11/97**

**© copyright
1998, James Hyjek**

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-26979-5

The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis

Automation of Object Behavior Validation

submitted by James Hyjek
in partial fulfillment of the requirements for
the degree of Master of Computer Science

Thesis Supervisor

Department

Carleton University

29/01/98

Abstract

Using **Computer Aided Software Engineering (CASE)** tools, designers can formally and electronically model their systems during the design phase. These design models can describe both the architecture and the behavior of the system being implemented. For projects using object-oriented methods and languages, objects form the building blocks of these models. The design model can not only capture class definitions and relationships, but also the intra-object and the inter-object behavior.

This thesis shows that information can be extracted from these design models and used to verify that the implemented system conforms to the design specification. CASE tools allow this process to be automated. This thesis focuses on validating the implemented application's internal behavior. Specifically, inter-object behavior which can be modeled using **Interaction Diagrams**. These diagrams show the flow of messages between collaborating objects. This thesis demonstrates that an application's inter-object behavior can be automatically validated against its interaction diagrams.

Acknowledgments

I would like to take this opportunity to thank those who helped make this work possible.

Professor John Pugh, whose guidance and suggestions helped bring this thesis to completion. My wife Anna, who put up with those long months of constant work. Jeff Alguire, who provided feedback and suggestions. My managers at Nortel/BNR who allowed me to schedule time-off at a moment's notice. All my friends and colleagues, who provided support and suggestions throughout this period.

Trademarks

- **VisualWorks** and **Smalltalk-80** are trademarks of ParcPlace-Digitalk.
- **Objectory** is a trademark of Rational Software Corporation.
- **X Windows** is a trademark of The Massachusetts Institute of Technology.
- **X-Runner** is a trademark of Mercury Interactive.
- **Java** is a trademark of Sun Microsystems Inc.
- Other products may be trademarks of their respective companies.

Table of Contents

1. Introduction	1
1.1 Black Box vs. White Box Testing.....	5
2. Related Work	9
2.1 Object Explorer.....	9
2.2 VisualWorks' Profiler.....	10
2.3 Bob Hinkle's Breakpoint Articles.....	11
2.4 Jeffrey Alguire's Thesis	11
2.5 Rational Rose	12
2.6 Together/Professional	12
2.7 Probe-ADvance Connect.....	13
2.8 The Smalltalk Test Mentor	13
3. Technology Overview	14
3.1 VisualWorks Smalltalk.....	14
3.2 Objectory.....	17
4. Automating the Validation Process.....	20
4.1 Capturing Behavior in the Design Model	20
4.2 Verifying the Behavior of the Implemented Application.....	27
4.2.1 Message Events	31
4.2.2 The Execution Profiler	33
5. Capturing Information in Objectory	35
5.1 Objectory's Interaction Diagrams	39
5.2 Events Flow Generator.....	42
6. Events Flow File Format.....	47
7. Brief Review of Smalltalk	56
7.1 Compiled Methods.....	56
7.2 Executing Bytecode Instructions.....	57
7.3 The MethodContext	58
7.4 Method Dictionary	63
8. How the Events Flow Processor works.....	65
8.1 Processing the Events Flow File	66
8.2 Method Enter Trap	69
8.3 Inserting Monitor Points into Methods	72
8.3.1 Primitive Methods.....	75
8.4 Recompiling the Monitored Method.....	77

9.	The Application Monitoring Session	83
9.1	Monitoring a Running Application	83
9.2	Objects Encountered During the Monitoring Session.....	91
9.3	Recording Method Arguments.....	96
9.4	Capturing the Method's Return Value	97
9.4.1	TrapMethodContext	100
9.4.2	Processing the Return Value	104
10.	End of the Monitoring Session	106
11.	Sample Monitoring Session	108
12.	Performance and Scalability	111
12.1	Setting up the Environment	111
12.2	Monitoring the Running Application.....	113
13.	Conclusions and Future Enhancements.....	116
13.1	Conclusions.....	116
13.2	Future Enhancements.....	118
14.	Alternative Designs	121
14.1	Subclassing from Debugger	121
14.2	Wrapping the Monitored Method	122

Figures

Figure 1 Automating the iterative loop in an object oriented development process	8
Figure 2 Visualworks Smalltalk class browser	15
Figure 3 Example of Objectory design model diagrams	18
Figure 4 Interaction diagram example.....	22
Figure 5 State-transition diagram for the aThermostat object.....	25
Figure 6 Interaction and state-transition diagrams with the aThermostat object	26
Figure 7 Execution monitoring system overview.....	29
Figure 8 Creating design objects in Objectory	35
Figure 9 Creating a method in Objectory	38
Figure 10 Interaction diagram in Objectory	40
Figure 11 Depth first traversal of an interaction diagram	44
Figure 12 Unique object identifiers.....	53
Figure 13 Events Flow File generated from Objectory	55
Figure 14 At any time the system can have several method contexts	60
Figure 15 MethodContext's instance variables.....	62
Figure 16 Looking up a method in the receiver's class	64
Figure 17 MethodEnterTrap inserting a monitor point in the source code	75
Figure 18 Monitor points in methods that call primitives.....	76
Figure 19 Recompilation and replacement of a monitored method	79
Figure 20 Replacing a method with a monitored method in a MethodDictionary	82
Figure 21 MethodEnterEvent and EncounteredObjectsDictionary	90
Figure 22 objectID from Events Flow File to EncounteredObjectsDictionary	95
Figure 23 Getting the method's arguments from the given MethodContext	97
Figure 24 Inserting a TrapMethodContext to capture method_3's return value.....	99
Figure 25 returnTrap source code and bytecode	101
Figure 26 TrapMethodContext and MethodExitTrap	103
Figure 27 Sample Objectory interaction diagram.....	108
Figure 28 Sample Events Flow File	109
Figure 29 Thermostat's setTemperature: method with monitor point	109
Figure 30 Log file for a successful monitoring session.....	110
Figure 31 Log file with an event that did not occur	110
Figure 32 Memory overhead (bytes) / # of Monitor Points.....	112
Figure 33 Processing overhead (msec) / # of Monitor Points	114
Figure 34 Wrapper method.....	123

Typographic Conventions (Code Syntax)

<u>Example</u>	<u>Description</u>
Keyword (specified in bold)	Always appears as is in statements.
<i><Variable></i>	Indicates a variable for which you must substitute a value.
	OR. Must choose one of several choices.
(a b)	Choose a OR b, () do not appear in code.
[]	Anything appearing between [and] is optional.
*	repeat 0 or more times whatever this follows (e.g.: Z * can produce 0 or more Z concatenated).

Typographic Conventions (Document Text)

<u>Description</u>	<u>Example</u>
Smalltalk Class names will be shown in bold.	MethodEnterTrap
Smalltalk method names (selectors) will be shown in bold.	methodEnter:callBackValue:
Smalltalk variable names will be shown in bold.	senderObject
Terms specific to the execution monitoring system may appear in italics, especially when first introduced or when it is important to distinguish them.	<i>method enter event</i>
Terms specific to this research area may also appear in italics, especially when first introduced or when it is important to distinguish them	<i>Black box</i> testing

1. Introduction

This thesis tackles the problem of verifying if an application's implementation deviates from its intended design. Automating the process of knowing where this deviation occurs facilitates the job of re-synchronizing the application and the design. This helps ensure that the application is implemented according to design and the design documentation is current. This chapter describes the problem domain; later chapters expand upon the solution. To demonstrate the validity of this research a *proof of concept* prototype was built. The prototype is also described in this thesis.

Most software projects use a development process that incorporates: *requirements analysis, design, implementation, and testing*. At requirements analysis the requirements specification is captured. During design, models are built which describe how the application will be realized. Implementation involves the actual coding. Testing verifies that the application was implemented correctly.

The system models built during design act as blueprints for the application's implementation. However, on most projects the implementation will deviate from the design. This is true regardless of whether the development process follows an iterative object-oriented model or a classical waterfall model. The deviation may be intentional or accidental on the part of the developer. Having incorrect design documentation can result in incorrect future design and coding decisions. Designers and developers may make decisions based on incorrect documentation.

In an iterative development model, the designer/developer will iterate between the design and implementation, incrementally improving and enhancing the application¹. It is not unusual for the developer to make enhancements to the implementation, which do not appear in the design models². In this case, any enhancement/deviations must be propagated back to the design models, so they are in sync. In practice, developers find updating the design model too much overhead. They will often just keep changing the implementation until they get what they want; then they will try to reverse engineer their work to update the design. By this point in time, there may be so many changes that it is difficult to remember them all. We should also not overlook the possibility that the developer accidentally deviated from the design.

An iterative development environment would benefit from automating the process of comparing design models to the implementation. Reducing the amount of work on the developer's part provides him/her with more incentive to keep the design documentation current. If the design models could be automatically updated to match the implementation, this would facilitate closure of the design/implementation iterative cycle.

Let us now look at how our research benefits the waterfall model. In theory, the waterfall model assumes that once a development phase is done, it is not revisited later. Therefore, once design is done, we go onto implementation, and then testing. In practice, there can be some iteration back to a previous phase. However, depending on the formality of the process, revisiting a previous phase can be a big deal (e.g.: manager's approval).³

In the waterfall model, the designer and implementer may not be the same person. On large projects, a handful of senior architects is involved at the design phase, and an army of junior programmers is responsible for implementation. During the design phase, the system's architecture and behavior are defined. For the purpose of conceptual integrity across the whole system, the granularity of these designs often gets down to the definition of *essential* classes and behavior. These designs also specify critical application logic that shows how essential objects are to interact with each other to carry out some important task. For example, you could have a very detailed description of how the objects in a nuclear reactor system would carry out the shutdown procedures.

In the waterfall model, having the implementation deviate from the design is much more serious than in an iterative model. Because the designer and implementer may not be the same person, some of these deviations can be due to misinterpretation of the design by the programmers⁴. These deviations can go unnoticed, even during testing. Validating that the implementation conforms with the design now becomes very important.

Whether a project follows an iterative or waterfall development process, the task of verifying consistency between the design model and the implementation is important. However, this validation carries an overhead. On projects that span years, this would be a repetitive on-going process. Therefore, this validation process must be automated if it is to gain acceptance in the software development industry.

The first step in automating our validation process starts with capturing the design information. This is where CASE tools come in. They allow the designer to build and capture the design models. Once electronically captured, this information can be automatically extracted for the purposes of verification. It is obvious that the more information you can include in the design model, the more extensive the automated validation can be. This thesis investigates whether it is possible to capture sufficient information in the design model to automate the process of verifying that the implementation and design are consistent with each other. Verifying whether the implemented application's structure and behavior conforms to what was described in the design model.

Two types of information can be captured in the design model: the static architecture of the application and its dynamic behavior. The static architecture describes how the application is put together. It can contain things like class definitions, relationships, etc. The dynamic behavior describes how the application behaves internally at runtime. This can contain descriptions of the inter-object and intra-object behavior. Inter-object behavior refers to the interaction between objects. In an object-oriented application, this would be the message passing between objects. Intra-object behavior refers to how the object behaves internally when it receives a message. In an object-oriented application, this would be contained in the methods. (Inter/intra-object behavior is discussed further in section 4.1.)

The problem of how to automate the validation of all the information in the design model can fuel several theses. This thesis focuses on validating the behavior of objects in a running application. Design models can describe both *intra-object* behavior and *inter-object* behavior. This thesis will show that it is possible for the design model to capture enough information on inter-object behavior to automate the process of verifying that the objects in the running application interact according to the design model.

1.1 Black Box vs. White Box Testing

Testing is an important part of the development process. Whether it is developer testing, integration testing, or regression testing; it presents an opportunity to validate the running application against the design. Integration testing, in some cases, is the first time that many of the components interact with each other. It is usually difficult for developers to do interoperability testing of their components early on in the application's development. This is because it requires other components that may not have been developed yet.

Today, most software projects only perform what is called *black box* testing. This is where the application is treated as a black box. Testers input information into the application and compare the outputs against expected results⁵. This testing only verifies that the application conforms to the specifications produced during the *requirements analysis* phase. It does not verify that the application was implemented according to the design specifications.

This thesis proposes that the software development process should put as much emphasis on verifying that the application conforms to the design model, as it does on verifying that it conforms to the requirements specification. After all, an application is not just a black box, but rather a complex system of inter-operating objects.

Developers usually don't have tools to assist them in validating design compliance. Most projects rely on *code reviews* to do the job. These manual reviews verify that the source code is complying with the static definitions in the design model (e.g.: class/variable/method names). Code reviews are not good at predicting the inter-object behavior of a running application. These code inspections tend to only find the obvious bugs. Subtle errors in logic spanning multiple modules are difficult to spot. These behavioral errors are not easy to spot by looking at the code, but show up a runtime. By using information in the design model, we can automate the validation of this inter-object behavior.

For software projects that have multiple releases and last years, regression testing is critical. Ensuring that behavioral changes are not accidentally introduced over time into the application is difficult. The migration of key personnel from the team can cause design knowledge to be lost. The behavior of a particular subsystem can change over the years because new people fixing bugs don't realize the subtle interdependencies of components. Years of patch releases introduce variations into obscure areas of the system.

Regression testing on these systems should be automated⁶. It should also validate that the application's implementation is still consistent with the design.

Software companies today do recognize the necessity of regression testing. They have also come to realize that it can be time consuming and resource intensive; especially if done manually. This has created a market for commercial tools that attempt to automate the testing process. Most of these tools are only good for black box testing. There is a lack of tools, which help to verify whether an application conforms to the design. There are a few tools that compare the static class definitions in the source code against the design model. Unfortunately, there is still a great need for tools which verify that the behavior of objects in an application conform to the design. This is especially true for languages like Smalltalk, whose customer base is not as large as languages like C/C++. Vendors have less incentive to fill in the development environment gaps for Smalltalk.

Commercial testing tools, like X-Runner by Mercury Interactive, implement this black box testing by recording the user's and application's interactions with the windowing system (e.g.: X Windows). Once the user's keystrokes and mouse movements have been recorded, these tools can automatically run the application and play them back. The application's new output is compared to a previous run's and differences are noted. Unfortunately, these tools can only monitor interactions with the windowing system. They cannot monitor the behavior of objects inside the application (white box testing⁷). The application's business logic cannot be validated.

This section talked about black box and white box testing. In an iterative object-oriented development process, finding discrepancies between the implemented application and the design model does not mean the implementation is wrong. It could mean that the design model must be updated. Automating this kind of white box testing would make it easier to provide closure on the design/implement/test iterative loop.

Figure 1 Automating the iterative loop in an object oriented development process

2. Related Work

At the time this thesis was started, there is no product that provides the complete set of capabilities discussed in this thesis for Smalltalk. Smalltalk's market share is smaller than more popular languages such as C/C++. Therefore, it does not attract the same numbers of tool vendors. Most of the work being done in this research area is by academics or individual developers. At the onset of this thesis, a search was done to find any related products. Of the handful that was found, only one dealt with recording message sends between objects at runtime. None talked about extracting information from CASE tools to validate inter-object behavior. The next few sections will review the related works that were found.

2.1 Object Explorer

Object Explorer is a profiling tool developed by Kent Beck⁸. It figures out the relationships between objects in an application; then graphically displays them. It works like a global inspector, allowing the developer to see the objects and their instance variable. It also includes a "contract" feature, which show how one object is being used by another.

The Object Explorer is a great debugging tool, but it does not validate inter-object behavior against design information found in CASE tools. No details concerning the tool's implementation were obtained. Therefore, it was not of much use for this thesis work.

2.2 VisualWorks' Profiler

VisualWorks Smalltalk has an add-on package called *VisualWorks Advanced Tools*. This package comes with two profilers: a time profiler and a memory allocation profiler.

The time profiler shows how much time is being spent in different portions of the code. It monitors the running application at regular time intervals. The way the time profiler works, is it periodically stops the running application to see which method is currently executing. It keeps a tally of which methods it has encountered and how often. This is how it estimates how much time was spent in each method. The user can vary the sampling intervals. Reducing the sampling intervals improves the accuracy, but will slow down the application. The time profiler enforces its sampling interval by monitoring the system clock.

The memory allocation profiler works by putting a wrapper method around the primitive methods that allocate space for objects. When the memory allocation profiler is started, it replaces the system memory allocation methods with its own. This allows the profiler to know exactly how much space is being allocated for the application's objects. After the profiling session, the original methods are restored.

The sampling technique used by the time profiler would not be useful for the purposes of this thesis. We could not guarantee that the sampling rate would be short enough to capture all the message events we were looking for.

The method wrapper technique used by the memory allocation profiler was useful for this thesis. The idea did serve as the basis for one of the alternative designs for our Smalltalk **Execution Monitor** (see section 14.2). Unfortunately, the installation of Smalltalk used for the *proof of concept* prototype did not have the *VisualWorks Advanced Tools* installed. Only the documentation was obtained. So, the source code for the *method wrappers* could not be examined.

2.3 Bob Hinkle's Breakpoint Articles

The Smalltalk Report featured two articles by Bob Hinkle on breakpoints⁸ and parameterized compilers⁹. His articles describe how to insert breakpoints without effecting the original source code. He does this by creating customized Smalltalk **Compiler** and **Parser** classes.

Bob Hinkle's papers were very useful in understanding how the Smalltalk **Compiler** works. His technique of inserting breakpoints provided ideas for our *proof of concept* prototype. However, his implementation was for a Smalltalk environment without Envy. The environment used in our *proof of concept* prototype has Envy. Envy is a source code control system (see section 3.1). Because it effects the way source is handled, much of Bob Hinkle's work could not be used.

2.4 Jeffrey Alguire's Thesis

This master's thesis¹⁰ talks about analyzing the message sends in Smalltalk/V applications. The information is gathered for the purpose of understanding the

interactions and collaborations among objects. These interactions and collaborations are displayed to the user as interaction diagrams. This system would be used by a developer to understand and debug his/her application.

His thesis does not address using CASE tools or extracting information from the design models. It also deals with a different implementation of Smalltalk than is used in this thesis. The Smalltalk monitoring mechanism in his thesis was built by subclassing the Smalltalk/V debugger. This approach carries the overhead of having the debugger running. It experienced problems monitoring the *Graphical User Interface* classes. There was also some difficulty capturing return values. Section 14.1 examines this approach and goes into more details as to why it was rejected for our **Execution Monitor**.

2.5 Rational Rose

Rational Software Corporation puts out a product called Rational Rose. It will automatically generate source code from the design models. It will also parse the source code and generate structural design models. It does not monitor the behavior of the implemented application and generate the corresponding behavioral models.¹¹

2.6 Together/Professional

This software product is put out by Object International Software. It will automatically generate source code from the design models. It will also parse the source code and generate structural design models. It does not monitor the behavior of the implemented application and generate the corresponding behavioral models.¹²

2.7 Probe-ADvance Connect

This software product is put out by IC&C GmbH. It provides most of the capabilities described in this thesis. It will automatically generate source code from the design models. It will also parse the source code and generate structural design models. It is capable of monitoring the message sends in a running application and capturing this information. It works with Smalltalk and provides its own design modeling capabilities rather than integrating with Objectory. The product's messaging monitoring capability has only been out since mid-1997, and was not available at the start of this thesis.¹³

2.8 The Smalltalk Test Mentor

This software product is put out by SilverMark. It provides automated testing capabilities. The user can record and play back user interface interactions with the application being tested. The user can also write scripts to automatically execute tests. It keeps track of which application code was executed/tested (code coverage statistics). It does not monitor/capture message events between objects and generate the corresponding behavioral models.

This chapter looked at some of the previous work in areas related to this thesis research. Most of the works described above provided ideas, but none came close to providing the complete picture. Chapter 14 looks at some of these ideas more closely and describes why they were rejected for this thesis.

3. Technology Overview

This chapter introduces the two key technologies used in this research. Since they are referred to often in this thesis, now is a good time to describe them. The intent is to give some background on them and recommend further reading. These technologies deal with the areas of object-oriented programming, analysis & design methods, and development processes. The reader is expected to have some general knowledge in these areas.

The two technologies used in this research are VisualWorks Smalltalk and Objectory. It should be noted that the theoretical aspects of this thesis are not restricted to either of these technologies. It is applicable to other object orient languages (e.g.: C++, Java); although, Smalltalk is one of the easiest to work with. Nor is it restricted to Objectory. It is also applicable to any OOA/D method and tool that allows the user to model the interactions between objects.

3.1 VisualWorks Smalltalk

Visualworks is an interactive development environment for Smalltalk-80. Smalltalk-80 is a dialect of Smalltalk, which is an object-oriented programming language. The Visualworks framework provides a *Graphical User Interface* builder, along with a rich library of foundation classes. It is one of the primary development environments for Smalltalk. Figure 2 shows an example of a Visualworks Smalltalk class browser.

Figure 2 Visualworks Smalltalk class browser

Smalltalk has been around for a while and there are a few good books on it. One of the earliest and still the best is Adele Goldberg's **Smalltalk-80, the language and its implementation**¹⁴. Though knowledge of Smalltalk is not necessary to understand this thesis, it will help.

Why was Smalltalk chosen for this research thesis? Firstly, the growing popularity of object-oriented languages has made them an important research area. Smalltalk is a pure object-oriented language, thus making it an excellent candidate. Secondly, much of the Smalltalk development environment is implemented in Smalltalk, and a good deal of the

source code is available. This research required the ability to modify some of the internals of the development environment.

Note to the reader: wherever this thesis mentions Smalltalk, it is referring to the Visualworks Smalltalk product with Envy installed.

Another product that is closely associated with the Visualworks environment is called Envy. Envy is a source code control system. It allows a team of developers to collaborate on the same project. Envy provides the standard checkin/checkout access control, ownership, etc.

When installed, Envy is very tightly integrated with Visualworks. The creation, deletion, and modification of all classes and methods go through Envy. For example, before a developer can modify code, (s)he must have permissions to do so in Envy. You can control which users have permissions on which classes.

Envy keeps a history of changes, by maintaining previous editions of classes and methods. For example, a developer creates a new edition of a method and makes modifications to that new edition. Even when the new edition is released, Envy still maintains all previous editions. This allows the developer to back-out his/her changes if there is a problem.

3.2 Objectory

Objectory is an object-oriented development process and method. There is also a tool, by the same name, which supports both the process and the method.

Its origins started in 1967, when Dr. Ivar Jacobson proposed a new set of modeling concepts for developing large telecommunication switching systems within Ericsson. The main concepts were geared toward a real-time system communicating with its environment via signals.¹⁵

The modeling concepts continued to evolve over time as more research was done in this area. In 1992 Jacobson published a book called **Object-Oriented Software Engineering - A Use Case Driven Approach**¹⁶, which discusses the principles of Objectory.

Today, Objectory encompasses both a comprehensive system development process and a method. Objectory supports the tasks of development, from requirements specification, through analysis and design, to a tested, complete system.¹⁷

At each of the phases in the Objectory development process, the designer develops models that describe the system to be developed. This modeling is applied according to the Objectory method, which describes in detail the different steps in the development of a system.¹⁸

Objectory is based on a use-case driven approach. A use-case is a sequence of transactions performed against a system, which produce a measurable result for the user. They are often referred to as scenarios or tasks performed by the user against the system.

One way to describe the flow of events in a use-case is with interaction diagrams.¹⁹ These diagrams are useful at the design stage for showing how objects collaborate to perform the use-case. Interaction diagrams and their importance to my research will be discussed in section 4.1.

There is a commercial tool called Objectory by Objectory AB, which supports the process and method. The Objectory *support environment* (Objectory SE), is an integrated set of tools. It facilitates use-case driven modeling throughout the development phases. For the purposes of this thesis, we are interested in the modeling tools for the design phase.

Figure 3 Example of Objectory design model diagrams

Figure 3 shows an example of two types of modeling tools found in Objectory. On the left is an *object association diagram* showing the composition and relationship of design objects. On the right is an *interaction diagram* showing the flow of message events between objects.

Objectory was chosen for this research thesis because it is object-oriented. Since the development environment was Smalltalk, an Object-Oriented Analysis / Design (OOA/D) method and tool was a good fit. If both the method and language are object-oriented there are fewer semantic gaps going from design to implementation. Objectory is one of the leading OOA/D methods, and a pioneer in the use of use-cases. It is an excellent choice.

This chapter introduced the main tools and environments used in this research. The reader should have a basic understanding of the underlying technologies before reading this thesis. If necessary, the two textbooks suggested above will provide a good background.

4. Automating the Validation Process

As mentioned in the *Introduction* chapter above, this thesis is interested in verifying that the implemented application behaves according to the design model. It is also interested in automating this verification process. To do this, two areas need to be addressed:

1. We need a way of capturing the desired application's behavior (business logic) as part of the design model.
2. We need to be able to monitor the objects in a running application to verify that they are behaving as specified in the design model.

4.1 Capturing Behavior in the Design Model

Many CASE tools, like Objectory, support the building of design models. CASE tools allow the designer to define the classes, objects, their variables and methods. The designer can also specify object relationships and behavior. When taken altogether, this information can provide an extensive description of what the implemented application should be like. Verifying that the implementation and the design model agree with each other is a time consuming process. It would be useful if there was a tool for automating the process.

There are tools out on the market that will reverse engineer the source code to get the class definitions. These definitions can be compared against the ones in the design model to identify differences. Although a good code review will achieve the same results, this is

a time consuming task that is easy to automate. This thesis goes beyond the simple verification of the class definitions, which only reflect the static nature of an application. This thesis looks at extending the automatic verification process to include an application's dynamic behavior. This is a much more difficult nut to crack.

Let's look at what the Objectory tool provides for describing the object behavior in the design model:

1. Interaction Diagrams.
2. State-Transition Diagrams.

Objectory is a use-case driven method, and interaction diagrams are employed to refine the description of these use-cases. These diagrams show how objects collaborate to perform the use-case. They describe the flow of events in terms of messages between objects.²⁰

The diagrams themselves consist of vertical columns that are used to represent objects (see Figure 4). At the top of each vertical column is a rectangular box with the name of the object that the column represents. The thick black vertical column on the far left is the environment boundary. It always appears in interaction diagrams, and represents everything outside the scope of the objects in the diagram.

Messages sent between objects are shown as horizontal arrows with the message name.

When an object (column) receives a message it executes a method. These method executions are shown as rectangular boxes on the columns. While within the method, messages can be sent to other objects; these are represented as arrows leaving the method rectangles. The time axis for an interaction diagram progresses chronologically from top to bottom. If a message arrow appears above another in the diagram, then it is sent earlier. Note that the time axis is not linear for message events. The distance between two message events in the diagram is not related to the real time between these events.

Figure 4 Interaction diagram example

The interaction diagram in Figure 4 shows how three objects cooperate to perform a specific task: “raising the temperature in a heating system”. All three objects are instance

objects. **aThermostat** is an instance of the **Thermostat** class. **aTemperatureSensor** is an instance of the **TemperatureSensor** class. **aFurnace** is an instance of the **Furnace** class. The object **aThermostat** represents a thermostat that is used to set the temperature in our heating system. The **aTemperatureSensor** object represents a temperature sensor that measures the room temperature. The **aFurnace** object represents a furnace that generates heat.

The interaction diagram in Figure 4 describes the following scenario. From the environment boundary, on the left, a **setTemperature:** message is sent to the **aThermostat** object. The thermostat is being asked to set the temperature to some given value. First, the **aThermostat** object checks the current room temperature by sending a **temperature** message to the **aTemperatureSensor** object. The **aTemperatureSensor** object will return the current temperature. In this scenario, it is assumed that current temperature is less than the new temperature setting. Therefore, the **aThermostat** object raises the room temperature, by sending a **startHeating** message to the **aFurnace** object.

Interaction diagrams can be as comprehensive or as vague as the designer wishes. In the above diagram, there are message events that are probably not shown. It is up to the designer to decide what are the important collaborative events to be documented.

The previous chapters talked about inter-object behavior. This is the interaction that takes place between objects as they cooperate to perform a task. In an object-oriented system,

this interaction takes the form of message passing between objects. Interaction diagrams are good for capturing this inter-object behavior.

The state-transition diagram is the second mechanism that Objectory offers the designer to capture behavior. This diagram describes the internal behavior of objects in terms of state transitions. It shows the actions taken by an object in specific states, when specific messages are received.²¹ The designer uses the state-transition diagram to describe an object in terms of a finite state machine. Objectory not only allows you to model states and messages, but also tasks, decisions, loops, etc.²²

If interaction diagrams model inter-object behavior, then state-transitions diagrams model intra-object behavior. (Intra-object behavior refers to internal behavior).

Figure 5 shows a state-transition diagram for the **aThermostat** object, introduced in Figure 4. The same “raising the temperature in a heating system” scenario is also used. In this diagram, events flow chronologically from top to bottom. The object **aThermostat** starts in the **Idle** state and receives the message **setTemperature:**. As a result, **aThermostat** sends the message **temperature** to the **aTemperatureSensor** object to get the current room temperature. If the current temperature is not lower, then it goes into the **Idle** state. If the current room temperature is lower than the new setting, the message **startHeating** is sent to the **aFurnace** object; then **aThermostat** goes into the **Idle** state.

Figure 5 State-transition diagram for the aThermostat object

In Figure 5, the ovals with the labels **Idle** represent states. In this state-transition diagram the object **aThermostat** started in the **Idle** state and ended in the **Idle** state. Just below the first **Idle** state, the symbol next to the **setTemperature: x** message represents a message input to **aThermostat**. The symbol just below that with the **temperature** message represents a message send from **aThermostat**. Below that, we find a diamond symbol that represents a decision. The question is **temperature < x?**, with separate **Yes** and **No** paths.

Both Figure 4 and Figure 5 describe different aspects of object **aThermostat**'s behavior. Interaction diagrams show inter-object behavior; state-transition diagrams show intra-object behavior. By using both, the designer provides a good picture of the objects' behavior during the application's execution. A side by side comparison of Figure 4 and Figure 5 illustrates the point more clearly (see Figure 6). Both show the behavior of **aThermostat** when receiving the **setTemperature:** message.

Figure 6 Interaction and state-transition diagrams with the **aThermostat** object

This section has looked at the two types of behavioral diagrams available in Objectory. Though there is lots of different information that can be found in the design model, this thesis will concentrate on inter-object behavioral information.

Why focus on inter-object behavior? Unlike most other parts of the design model, inter-object behavior cannot be verified by simply examining the source code. This is especially true for dynamically typed languages such as Smalltalk, where data types are resolved at run time. Since inter-object behavior can only be confirmed at run time, it is still a source of many bugs. Most run time testing takes the form of black box testing, which is poor at spotting subtle errors.

This section was intended to tie together interaction diagrams and inter-object behavior. Since this thesis deals with validating inter-object behavior, it will focus on interaction diagrams. Chapter 5 will look at extracting this information from Objectory for use during the automated verification phase.

4.2 Verifying the Behavior of the Implemented Application

The previous section talked about modeling an application's inter-object behavior using interaction diagrams. In this section, we are interested in verifying that the implemented application conforms to this modeled behavior. If there is any discrepancy, then the designer must modify either the application or the design model. It is important that the design model and the application agree with each other.

Inter-object behavior in an application is hard to verify. We can not just look at the source code and deduce the objects' interactions. This is especially true for our target language: Smalltalk. Smalltalk is dynamically typed, so some data type information is only available at run time. Variable references are not resolved until run time. This means you can not be certain of the object it will be bound to, nor the object's class. This information is necessary to determine inter-object behavior.

To properly verify the inter-object behavior of a Smalltalk application, it must be monitored at run time. You must be able to look inside and see which messages are being sent to which objects, and in what sequence. This is referred to as *White Box* testing.

For our purposes, it is necessary to both record the inter-object behavior and compare it to the documented behavior in the interaction diagrams. This thesis looks at how to do just that.

As part of this research, a *proof of concept* prototype was built to demonstrate the feasibility of this thesis. The prototype was an **Execution Monitoring** system for Smalltalk. There are two parts to this system; the first part extracts the information from Objectory's interaction diagrams. The second part monitors a running Smalltalk application. It uses the information from Objectory to verify that the application's objects behave in accordance with the interaction diagrams.

Both parts of the **Execution Monitoring** system are implemented as separate mechanisms and can be used on their own or together. Chapter 5 goes into details on the mechanism for extracting information from Objectory. Chapters 8, 9 and 10 go into detail on the mechanism for monitoring a running Smalltalk application.

Figure 7 Execution monitoring system overview

Figure 7 shows an overview of the whole **Execution Monitoring** system. The shaded components are the ones built for our *proof of concept* prototype. On the left side of the

figure is the Objectory tool. Designers use Objectory to create interaction diagrams. The mechanism built to extract the information from these diagrams is called an **Events Flow Generator**, shown just below the interaction diagrams.

The **Events Flow Generator** produces a file called the **Events Flow File** (shown in Figure 7). The **Events Flow File** contains a description of all the *message events* in the interaction diagrams. It specifies what order the messages occur in, which objects are involved and their classes. The reason it is called an **Events Flow File** is that it specifies the flow of message events occurring in an application.

Once the information has been extracted from Objectory, it can be used to validate the inter-object behavior of a Smalltalk application. Since we will be monitoring this application's execution, it is shown in Figure 7 as the **Monitored Application**.

On the right-hand side of Figure 7 is the Smalltalk environment. Inside there are two components that are part of this prototype: the **Events Flow Processor** and the **Execution Monitor**. The **Events Flow Processor** is an application written in Smalltalk. It reads in the **Events Flow File** and prepares the environment and target application for monitoring.

For each *message event* in the **Events Flow File**, the **Events Flow Processor** inserts a *monitor point* in the **Monitored Application**. This is similar to a breakpoint that is used by most debuggers. During the execution of the **Monitored Application** if a *message event* occurs for which we have set a *monitor point*, then the **Execution Monitor** is

invoked. The **Execution Monitor** verifies that the actual *message events* occurring agree with those specified in the **Events Flow File**.

After the monitoring session, the **Execution Monitor** produces a **Log File**, containing all the valid *message events* encountered and a list of any discrepancies. If there are no discrepancies then the inter-object behavior of the application conforms to the interaction diagrams in the design model. Discrepancies take the form of message events which did not occur, occurred out of sequence, did not have the correct arguments, etc.

The format of the **Log File** produced by the **Execution Monitor** is the same as the **Events Flow File** extracted from Objectory. This was done on purpose so that the **Log File** could be fed back into the Smalltalk monitoring environment, as an **Events Flow File**. This introduces another use for this Smalltalk monitoring environment - as an execution profiler. This option will be discussed later. (Chapter 11 shows an **Events Flow File** and corresponding **Log Files** in a sample monitoring session).

4.2.1 Message Events

Message events play an important role in this research. This section explains what message events are, and how they are related to both interaction diagrams and Smalltalk messages.

“The purpose of the Interaction Diagrams is to describe flows of events in terms of stimuli transmissions between objects.”²³. In Smalltalk, these stimuli transmissions are

messages. Our **Execution Monitoring** system refers to them as *message events* or *method events*.

Since the **Events Flow Processor** has to set *monitor points* in the application, these *message events* have to map into something specific in Smalltalk. The decision was made to treat the *message events* in the **Events Flow File** as *method enter events* in Smalltalk.

Sending a message to an object causes the method corresponding to that message to be invoked. This invocation is seen as the method being entered; hence, the term *method enter event*. By the same logic, the flow of execution also exits the method when the method returns; this is seen as a *method exit event*. Method enter and exit events are covered in detail later on.

The **Events Flow Generator** only extracts *method enter events* from Objectory. This is due to Objectory's lack of formal support for message returns in interaction diagrams (see section 4.1). However, the Smalltalk **Execution Monitor** captures and records both types of events.

Message events contain much more information than just who the sending and receiving objects are. For *method enter events* the arguments are recorded. For *method exit events* the return value is recorded.

Each message event in the **Events Flow File** must contain sufficient information to uniquely identify it. During the monitoring session, the **Execution Monitor** must be able

to correctly map Smalltalk message events to **Events Flow** message events. Chapter 6 explains in detail the format of message events and will show that they are unique. The intent of this section was to introduce and explain them to the reader.

4.2.2 The Execution Profiler

This section looks at an alternate use for our execution monitoring system. If we ignore the **Events Flow Generator** in Objectory, that still leaves us with a Smalltalk execution monitoring system. This system can be used on its own as an execution profiler.

The **Events Flow File**, which feeds the Smalltalk execution monitoring system, is a plain ASCII file and can be created by hand. The user can populate it with the message events of interest. This allows specific methods to be monitored.

The format of a message event in the **Events Flow File** requires: the sender and receiver objects identifiers, the sender and receiver classes, the sender and receiver methods, the receiver's arguments and return value data types (see chapter 6). This is way too much detail for the user to enter by hand. Fortunately, it is possible to specify wildcards in message events. The wildcards can be applied to sender objects and sender methods. Therefore, you could ask the profiler to record all invocations of method `zot:` of class `Foo`. Since the **Events Flow Processor** has to set *monitor points* in the receiver method, wildcards are not permitted for receiver method and receiver class. However, you can have a wildcard indicating that all instance of the receiver class are to be monitored.

A typical profiling example would be to create an **Events Flow File** that would tell the **Execution Monitor** to record all **zot:** messages sent to any instance of the class **foo**. The **Log File** resulting from this profiling session will capture detailed information on which objects invoked the monitored method and in what order.

As mentioned earlier, the **Log File** produced by a profiling session has the same format as an **Events Flow File**. This means that it can be used as input into an execution monitoring session. Since its message events are very detailed and in a specific order, it can be used to check if the behavior of the application changes over time. This is a useful tool for performing automated regression testing.

Because the execution monitor inserts *monitor points* to methods, it is not practical to use it for extensive execution profiling. The overhead would be too great if a user wanted to profile every method in every class. The execution monitor works best when the user has specific methods to be monitored.

This section discussed how the execution monitoring system could function as an execution profiler.

5. Capturing Information in Objectory

Previous chapters talked about what kinds of design information we are interested in capturing. This chapter looks at how to capture this information in Objectory. The reader will obtain an understanding of the CASE tool and its limitations. This will provide insight into some of the design decisions made in building the *proof of concept* prototype. We will look at how the execution monitoring system extracts information from Objectory.

Before the design information can be extracted from Objectory SE, it must first be entered. A designer will typically start by defining the objects of interest. Objectory provides tools for defining classes, class instances, class/instance variables, class/instance methods, relationships, etc. Figure 8 shows two of these tools.

Figure 8 Creating design objects in Objectory

A design object can represent a class, an instance or both. Since we are targeting Smalltalk, it is better to create separate design objects to represent the class and the class instance. The reason being that the **Events Flow Generator** must distinguish between classes and instances. In Smalltalk, a class is a separate object from its instances. They both have their own methods, variables and receive their own messages. Therefore, they should be represented as separate design objects in Objectory. Otherwise, our **Events Flow Generator** could not distinguish whether a message was being sent to the class or an instance.

Objectory assigns an internal unique identifier to each design object. As the **Events Flow Generator** extracts information from interaction diagrams, it uses this identifier to differentiate between objects.

Objectory provides no way of explicitly labeling a design object as a class or an instance. For our prototype, the decision was made to use the design object's name to do this. If the design object is a class, its name has the word "class" appended to it. For example, the design object representing the **Thermostat** class is called "Thermostat class". The design object representing an instance of the **Thermostat** class is called "Thermostat".

The author is aware that the above naming convention does not conform to the standard naming practice. Typed variables typically take on the form: a<Noun> or an<Noun>²⁴. By these rules, the **Thermostat** instance should be called "aThermostat". However, using "Thermostat" and "Thermostat class" makes parsing easier in our Smalltalk **Execution**

Monitor. It would require a little effort to modify the **Execution Monitor** to accommodate the standard convention, but is not necessary for the purposes of this prototype.

The **Events Flow Generator** also requires other information on the methods to be monitored. Specifically, it needs the method's signature. The signature describes the classes of the input arguments, and the class of the return value. For example,

Thermostat has an instance method **setTemperature:** which takes a single argument.

The argument is a **SmallInteger** that is the new temperature to set the thermostat to. This method returns the receiver object (self); in this case, self is an instance of **Thermostat**.

Figure 9 shows an example of defining a method and its signature in Objectory.

For each method defined, Objectory provides a free format text field for the signature. For the purposes of our prototype a format was chosen which would capture the necessary information and be easily parsed by the **Execution Monitor**:

<Return Value's Class> (<Argument's Class>, ...)

where *<Return Value's Class>* is the class of the object returned by the method. The arguments list is comma separated and enclosed in parentheses. There can be zero or more arguments. If there are zero arguments then empty parentheses appear. *<Argument's Class>* is the class of the argument.

Create Operation

Name:

Signature

☒ Instance ☐ Class

Description:

Accept
Cancel
Help

Figure 9 Creating a method in Objectory

Figure 9 shows how the **setTemperature:** instance method is created for **Thermostat**.

The name field is used to specify the method's Smalltalk selector name. Note that the signature field follows the format described above. It tells us that the **setTemperature:** method takes a single argument, an instance of the **SmallInteger** class. It also tells us that the method returns an instance of the **Thermostat** class, in this case, it will be 'self'.

Though this thesis focuses on extracting information from interaction diagrams, Figure 8 shows some of the other design information captured within the CASE tool. This information includes such things as class definition (e.g.: method and variable names), and object relationships (e.g.: containment, inheritance, reference). It would be possible

to extract all this information and then verify that the implemented application conforms. Most of this information deals with the static nature of the application and can be verified by examining the source code. This thesis instead deals with the harder problem, of verifying the dynamic behavior of an application.

5.1 Objectory's Interaction Diagrams

The previous section showed how design objects are created. This section will talk about using interaction diagrams to specify the inter-object behavior.

Figure 10 shows an Objectory interaction diagram, which is part of the “Heating System” application introduced in section 4.1. It describes the flow of *message events* required to “raise the temperature on a thermostat”. As stated earlier, the designer usually does not describe every message event that occurs, only the essential events.

Figure 10 Interaction diagram in Objectory

From left to right in Figure 10 we start with the message **setTemperature:** coming from the environment boundary and being sent to an instance of **Thermostat**. The **Thermostat** object first checks the room temperature by sending a **temperature** message to an instance of **TemperatureSensor**. If the room temperature is below the new setting, a **startHeating** message is sent to an instance of the **Furnace**.

Each column in an interaction diagram is mapped to an Objectory design object. In this example, each column is mapped a different design object (**Thermostat**, **TemperatureSensor** and **Furnace**). However, it is possible for multiple columns to be

mapped to the same design object. This can happen if the design object represents an instance of a class. For example, we only created a single design object called **Furnace** to represent all instances of the **Furnace** class. If our heating system had two furnaces, then our interaction diagram would have two columns mapped to the same **Furnace** design object. In this case, each column would represent a separate instance. So how does the **Events Flow Generator** distinguish between the two instances? Fortunately, each column is assigned a unique identifier in Objectory. (Remember, the columns really are supposed to represent object instances in interaction diagrams). The **Events Flow Generator** uses these column identifiers as object instance identifiers. Chapter 6 covers this in more detail.

The message events in Figure 10, like **setTemperature:**, are mapped to design object methods. This allows the **Events Flow Generator** to query Objectory for their signature. Figure 9 shows an example of a design object method and its signature.

When it comes to message events in interaction diagrams there is one serious omission in the Objectory tool; there is no direct support for depicting messages sent to *self*. The diagram editor should provide a specific message stimulus arrow with this semantic. The Objectory manual does suggest a work-around using the comment field for the message stimuli. However, it would be impractical for the **Events Flow Generator** to rely on this in a production environment.

Another feature that is missing from the interaction diagrams is the ability to specify that message events are executed concurrently rather than sequentially. For the purposes of this thesis, sequential execution is assumed and methods block until they return.

Overall, Objectory's interaction diagrams do capture sufficient information for the purposes of our *proof of concept* prototype. However, for our execution monitoring system to be successful in the real world, the above mentioned omissions would have to be corrected. Note that these omissions are a problem with Objectory, and not the execution monitoring system. The execution monitoring system can easily handle message to *self* and concurrent message event flows.

5.2 Events Flow Generator

Once all the necessary information has been captured in Objectory's design model, it must be extracted in a form the execution monitoring system can use. This process of extracting the information and producing an **Events Flow File** must be automated. Fortunately, the Objectory environment comes with a mechanism to help do this. This mechanism is part of their code generator facility.

Objectory's code generators works by extracting information from the design model and producing source code. For example, the design objects, introduced at the beginning of chapter 5, describe classes. They define the class name, methods, attributes (variables), etc. Objectory's code generator is capable of reading these design objects and generating files containing the corresponding Smalltalk class definitions.

Objectory already comes with code generators for Smalltalk and C++; but users can also build their own code generators. Objectory provides a **Code Generator Language** and compiler for this purpose. The **Events Flow Generator** is written in this **Code Generator Language**. Once the **Events Flow Generator** is installed in Objectory, it can be invoked from Objectory's system browser, just like any of the built-in code generators.

The **Code Generator Language** is a simple script-like language. It allows the user to query the design model for information, and produce formatted output files. It is suited more for querying design objects, rather than interaction diagrams. Despite its limitations and poor documentation, it was adequate for the purposes of our *proof of concept* prototype.

When the **Events Flow Generator** is run against the design model, it starts by extracting a list of all the interaction diagrams. It then processes the diagrams one at a time. For each diagram, its environment boundary is first located. This is because all message event flows start from the environment boundary. The **Events Flow Generator** queries the environment boundary to get a list of all the message stimuli leaving it. In Figure 11, the two message stimuli leaving the environment boundary are numbered 1 and 4. These message stimuli are traversed to find their destination columns.

Each message stimulus causes a method execution; this is depicted as a rectangular box on the destination column (see Figure 11). The **Events Flow Generator** queries the method execution to determine which method it is mapped to and to get a list of all the

message stimuli it sends out. In Figure 11 the first method execution is mapped to the method `setTemperature:`, and it sends out the message stimuli 2 and 3. These stimuli are in turn traversed and processed the same way those sent out from the environment boundary were.

Figure 11 Depth first traversal of an interaction diagram

To understand the order, in which the **Events Flow Generator** traverses the interaction diagram, think of the diagram as a hierarchical tree. Starting from the environment boundary, a depth first traversal is performed on the hierarchical tree. Figure 11 numbers the message stimuli in the order in which they are traversed. This happens to be the same order in which the message events will occur at run time.

Let us now look at what information the **Events Flow Generator** extracts as it traverses the interaction diagram. It is looking for *method enter events* (see section 4.2.1). Each message stimulus begins a *method enter event*. The following information is extracted:

- **Sender Class:** This is the class of the object sending the message stimulus. The class name is gotten by querying the column sending the stimulus to determine which design object it is mapped to. The design object's name will be the class name. In Figure 11 the first column is mapped to the design object named **Thermostat**. The name indicates that it is an instance of the **Thermostat** class. If column one were mapped to the **Thermostat** class itself, then the design object's name would be **Thermostat class** (naming convention explained in chapter 5).
- **Sender Id:** Objectory assigns a unique id to each column. This is used as the sender's object ID.
- **Sender Method:** This is the method sending the message stimulus. On the interaction diagram message stimuli originate from the *method executions* (depicted as rectangular boxes on the columns). These *method executions* can be queried to find out which method they are mapped to. In Figure 11 the first *method execution* on the **Thermostat** column is mapped to the **setTemperature:** method. Note, that **setTemperature:** is the sender method for the message stimuli **temperature** and **startHeating**.

- **Receiver Class:** This is the class of the object receiving the message stimulus. Like the sender class, it can be gotten by querying the corresponding column and the design object.
- **Receiver Id:** This is similar to the Sender Id, except it is the unique identifier of the column receiving the message stimulus.
- **Receiver Method:** Similar to the sender method; this is the method which is invoked as a result of the receiver object receiving the message stimulus. In the case of the receiver method, we extract both its name and its signature.

Messages originating from the environment boundary are not assigned a specific sender class, id, or method. They are treated as if they could come from any class. This is explained in the next chapter.

This section looked at what information the **Events Flow Generator** extracts from Objectory. The next chapter looks at this information in more detail.

6. Events Flow File Format

This chapter describes the contents and format of an **Events Flow File**. This is a plain ASCII text file. Its purpose is to capture the flow of *message events* that can occur during the running of a Smalltalk application. An **Events Flow File** can be created in one of three ways:

- Running the **Events Flow Generator** within Objectory captures the *message events* found in the interaction diagrams.
- A text editor can be used to create one by hand. It is expected that a user will only go through the trouble of producing an **Events Flow File** by hand, if they are interested in using the **Execution Monitor** as an execution profiler (see section 4.2.2).
- After an application monitoring session, the **Execution Monitor** produces a **Log File** of all the *message events* that it encountered. This log is in the same format as an **Events Flow File**. Therefore, it can be used as input into a future monitoring session, to determine if anything has changed.

The format of the **Events Flow File** allows a sequence to be assigned to the *message events*. Files generated by both the **Events Flow Generator** and the **Execution Monitor** always specify the sequence in which *message events* must occur.

Files created by hand usually don't have a sequence specified. In this case the user wants the **Execution Monitor** to just monitor the behavior of certain objects and produce an execution **Log File**.

The contents and format of the **Events Flow File** are described line by line in the table below.

Line #	Syntax	Description
1	<Title>\$	A text string that identifies the events flow in this file. Line ends with \$.
2	SOURCE (Objectory Recording Other)\$	<p>Indicates how this file was created. It could have been:</p> <ul style="list-style-type: none"> - Generated from Objectory. - Recorded by the Execution Monitor. - Constructed by other means (e.g.: by hand). <p>() do not appear. Line ends with \$.</p>

3	EVENTS_SEQUENCED (Y N)\$	Do the events in this file appear in chronological sequenced. If Y then they do. If N then there is no order to the events. () do not appear. Line ends with \$.
4 - n	<Message Event>	The remaining lines in the file describe the message events that make up the events flow. (See below for details)

Line 4 and each line after it in the **Events Flow File** specifies a *message event* by describing its *method event signature*. This signature contains enough information about the *message event* for the **Execution Monitor** to recognize it when it occurs during the monitoring session. It describes the objects sending and receiving the message, the message arguments and return value. The syntax of a *method event signature* is:

<Sender Class>\$<Sender Id>\$<Sender Method>\$<Receiver Class>\$<Receiver Id>\$<Receiver Method>\$<Receiver Method Signature>\$

- The character '\$' is a field separator and appears at the end of the line.

- *<Sender Class>*: If the sender object must belong to a specific Smalltalk class then that class's name is given in this field. If the sender object is an instance of a class, then that class's name appears in this field (e.g.: **Thermostat**). If the sender object is the class itself then the class's name followed by the word "class" appears in this field (e.g.: **Thermostat class**).

If the sender can belong to any class then the string "***ANY***" appears in this field.

- *<Sender Id>*: If the sender object must be a specific instance of the *<Sender Class>* or the class itself, then this field will hold that object's unique identifier. This identifier is a string of numerics with optional vertical bars (e.g.: 23 or 3600315|241547|12|15). Object identifiers extracted from Objectory contain these vertical bars. (The object identifier is explained later in this chapter).

If the sender object can be any instance of the *<Sender Class>*, then this field contains the string "***ANY***".

- *<Sender Method>*: If the method sending the message to the receiver must be a specific method then this field will hold its Smalltalk method selector name.

If the sender method can be any method, then this field contains the string "***ANY***".

- *<Receiver Class>*: This field contains the name of the class to which the receiver object belongs. This is a mandatory field; a specific class name must be given. If

the receiver object is an instance of a class, then that class's name appears in this field. If the receiver object is the class itself then the class's name followed by the word "class" appears in this field.

- *<Receiver Id>*: If the receiver object must be a specific instance of the *<Receiver Class>* or the class itself, then this field will hold that object's unique identifier. This identifier is a string of numerics with optional vertical bars (e.g.: 23 or 3600315|241547|12|15). If the receiver object can be any instance of the *<Receiver Class>*, then this field contains the string "**ANY**".
- *<Receiver Method>*: This field contains the Smalltalk selector name of the method being invoked by the sender. It is the method corresponding to the message being sent. This is a mandatory field; a specific method selector name must be given. (Note this field is used in conjunction with the next field *<Receiver Method Signature>*).
- *<Receiver Method Signature>*: This field is used in conjunction with the previous field *<Receiver Method>*. This is a mandatory field; a specific signature must be given. It describes the receiver method's arguments and return value using the following format:
<Return Value's Class> (*<Argument's Class>*, ...)
 where *<Return Value's Class>* is the class of the object returned by this method. The arguments list is comma separated and enclosed in parentheses. There can be

zero or more arguments. If there are zero arguments then empty parentheses appear. *<Argument's Class>* is the class of the argument. The *<Return Value's Class>* is separated from the arguments list by a single space character.

In the descriptions of the *<Receiver Class>*, *<Receiver Method>* and *<Receiver Method Signature>*, it was stated that these fields are mandatory. They must have specific values, rather than the string “*ANY*”. The reason being that the **Events Flow Processor** reads in the **Events Flow File** and inserts monitor points in the specified Smalltalk methods. It therefore has to know the receiver's class and method. This is explained in detail in chapter 8).

Both the *<Sender Id>* and *<Receiver Id>* fields take object identifiers. Object identifiers are used to differentiate between objects in the **Events Flow File**. This is useful when there are multiple instances of the same class.

Figure 12 Unique object identifiers

Figure 12 depicts the use of unique object identifiers to distinguish between the various sender and receiver objects. The picture on the left shows four objects. The objects with identifiers 1 and 4 are sender objects. The objects with identifiers 2 and 3 are receiver objects. The message events occurring in the left picture can be captured in the **Events Flow File** with the following entries:

```
System$1$*ANY*$Thermostat$2$setTemperature:$Thermostat (SmallInteger)$
```

```
System$4$*ANY*$Display$3$update:$Display (String)$
```

The “*ANY*” indicates we don’t care what the sender method is.

The right-hand picture in Figure 12 is similar to the left-hand picture. The difference is that the sender object for both messages is the same object. The message events occurring in the right-hand picture can be captured in the **Events Flow File** with the following

entries:

```
System$1$*ANY*$Thermostat$2$setTemperature:$Thermostat (SmallInteger)$
```

```
System$1$*ANY*$Display$3$update:$Display (String)$
```

By having the unique identifier ‘1’ appear as the sender for both events, we are telling the **Execution Monitor** that the sender Smalltalk object in the second event must be the exact same Smalltalk object which appeared in the first event.

The significance of object identifiers will be reviewed again when we look at the **Execution Monitor** (See section 9.2).

Figure 13 is an example **Events Flow File**. The first line is the title that describes this events flow. The second line indicates that this file was crated by hand. Line 3 says that the events must occur in the sequence they are given in the file. Lines 4 to 7 are message events. The “*ANY*” entries in the 4th line indicate that the sender object can be any object. In line 4 the **new** message is sent to the **Thermostat class**. This will return an instance of **Thermostat**. In the 5th line the **setTemperature:** message is being sent to an instance of the **Thermostat**. It takes a single argument that is a **SmallInteger** and returns the **Thermostat** object (self). In lines 6 and 7 the **Thermostat** object is sending a message to an instance of **TemperatureSensor** and an instance of **Furnace**, respectively.

```

Raise Temperature$
SOURCE Other$
EVENTS_SEQUENCED Y$
*ANY*$*ANY*$*ANY*$Thermostat class$1$new$Thermostat ()$
*ANY*$*ANY*$*ANY*$Thermostat$2$setTemperature:$Thermostat (SmallInteger)$
Thermostat$2$setTemperature:$TemperatureSensor$3$temperature$SmallInteger ()$
Thermostat$2$setTemperature:$Furnace$4$startHeating$Furnace ()$

```

Figure 13 Events Flow File generated from Objectory

Because these message events are sequenced, they must occur during the monitoring session in the order they are given in the **Events Flow File**.

Now that we know how **Events Flow Files** are created and what they look like, the following chapters will deal with using them in the Smalltalk execution monitoring environment.

7. Brief Review of Smalltalk

This chapter reviews the parts of Smalltalk, which are essential to understanding how the **Execution Monitoring** system works. It will review the interpreter, compiled methods, method dictionaries, method contexts, and evaluation stacks. This chapter is not meant to be comprehensive, but rather provides the reader with sufficient understanding of Smalltalk for the scope of this thesis. However, the reader is expected to have a cursory knowledge of Smalltalk. Familiarity with basic concepts like classes, messages, virtual machines, etc. is essential. A good reference for this is *Smalltalk-80, The Language and its Implementation*²⁵ by Adele Goldberg.

7.1 Compiled Methods

When a developer writes the source code for a method, Smalltalk captures and stores this code in an instance of the **String** class. The Smalltalk compiler translates these strings into bytecodes that can be understood by the interpreter. It is these bytecodes which are executed when a method is invoked.

The Smalltalk compiler takes a source code string and produces an instance of **CompiledMethod**²⁶. Each method has a corresponding instance of **CompiledMethod**. The **CompiledMethod** contains the method's bytecodes and has a reference to the source code string.

In Smalltalk, methods belong to classes. Each class has a **MethodDictionary** that stores the **CompiledMethods** for that class. When a method is invoked at run time, its **CompiledMethod** is looked up in the corresponding class's **MethodDictionary**.

7.2 Executing Bytecode Instructions

The bytecodes found in **CompiledMethods** are executed by the interpreter. The interpreter processes the bytecode instructions, in the following manner²⁷:

1. Fetch the bytecode pointed to by the **instruction pointer**.
2. Increment the instruction pointer.
3. Execute the bytecode.

The above 3-step cycle is performed repeatedly. There are variations to the steps that involve JUMP instructions and branches.

To perform the above cycle the interpreter needs to maintain **state** information about its execution environment at all times. The interpreter's state holds:

- The **CompiledMethod** whose bytecodes are being executed.
- An instruction pointer to the next bytecode to be executed in the **CompiledMethod**.

- The receiver object which invoked this method.
- The arguments passed into the method.
- Any temporary variables in the method.
- A stack.

7.3 The MethodContext

The current state information held by the interpreter depends heavily on the **CompiledMethod** it is executing. If the method being executed sends a message to another object, this will invoke another method and force the interpreter to start executing another **CompiledMethod**. As a result, its state information will change. The interpreter must retain its old state information, if it is to resume executing the old method after the new method returns. Fortunately, the Interpreter supports the notion of *contexts* to help it do this.

The interpreter executes a method's code within a context. The *context* ties together information on the compiled method and the interpreter's state. There exists a class within Smalltalk to do this, called **Context**. There is a specific subclass called **MethodContext** that is used to process methods.

When the interpreter comes across a message send, in the current method it is executing, it saves some state information in the current **MethodContext** before switching over to

the new method. A new **MethodContext** is created to process the newly invoked **CompiledMethod**. The old **MethodContext** is said to be "suspended" and the new current context now becomes the *active context* in the system²⁸. Since method invocation can be nested (e.g.: *method_1* calls *method_2* which calls *method_3*), the system may hold several suspended contexts and one active context. It is actually the responsibility of the **MethodContext** objects to keep track of each other.

Every context knows the previous context it suspended. For example, the active context for the current method being processed has a reference to the suspended context of the method that invoked the current method. Figure 14 depicts the scenario where *method_1* of *Object_1* sends message *method_2* to *Object_2*, *method_2* sends message *method_3* to *Object_3*. *MethodContext_1*, which is being used to process *method_1*, was suspended when *method_2* was invoked. *MethodContext_2* in turn has been suspended by *MethodContext_3*. *MethodContext_3* is the active context; it will exist until *method_3* has finished executing. When *MethodContext_3* terminates, control is passed back to its *sender*, which is the **MethodContext** it suspended. In this scenario, control is passed back to *MethodContext_2*, and the interpreter's state variables are restored to what they were at the point *MethodContext_2* was suspended. When *method_2* is finished executing, control is passed back to its sender, *MethodContext_1*.

Figure 14 At any time the system can have several method contexts

Below are the instance variables in the **MethodContext** class, which are of interested to our **Execution Monitor**. How the **Execution Monitor** uses them is discussed in later chapters.

sender: The method context that invoked this method context. Figure 14 shows, **sender**

is the method context which was suspended by this method context.

(**MethodContext** is a subclass of **Context**, and in reality **sender** could be an instance of any **Context**'s subclasses.)

receiver: The object that received the message, which this context is processing.

stack: An **Array** holding the evaluation stack for this context. The top of the stack holds the arguments and temporary variables for the current method. The rest is used as a working stack, to push/pop values as the method's bytecodes are executed.

method: The **CompiledMethod** being executed in this context. (**CompiledMethod** is a subclass of **CompiledCode**, and in reality **method** could point to an instance of any of **CompiledCode**'s subclasses.)

pc: program counter (it is also referred to as **instruction pointer**.) It points to the next bytecode instruction to be executed within this context. When an active context is suspended, **pc** points the instruction at which to resume execution when the suspended context becomes active again.

Figure 15 depicts the **MethodContext** and the objects it references. The

CompiledMethod is shown as referencing an array of bytecodes. These are the compiled bytecodes for that method. The instruction pointer (**pc**), in the **MethodContext**, points to the next instruction to execute. All this information needs to be retained by a suspended context, so that execution may resume once it becomes active again.

Figure 15 MethodContext's instance variables

In Smalltalk, there is a pseudo variable called **thisContext**. This variable can be accessed from within a method, just as any other pseudo variables like **super** and **self**. **thisContext** references the current context object. Our **Execution Monitor** makes use of this variable to determine what the current active **MethodContext** is for a monitored method.

Chapter 9 discusses how **MethodContexts** are used by our execution monitoring system to access a monitored method's arguments and get its return value.

7.4 Method Dictionary

All **CompiledMethods** that belong to a class are stored in that class's method dictionary.

Each class has an instance of **MethodDictionary** whose keys are the method selectors and values are the **CompiledMethods**. A method selector is a symbol consisting of the '#' character followed by the method's name. For example a method called 'printString' would be represented by the selector '#printString'.

When the interpreter comes across a *message send* bytecode, it finds the **MethodDictionary** in the *receiver* object's class. Then it uses the message's selector to lookup the corresponding **CompiledMethod**. If there is no key matching the method selector in the class's **MethodDictionary**, then the interpreter searches the receiver's super class's **MethodDictionary**. If no match is found, the super class's super class is searched, etc. This recursive search up the class hierarchy is repeated until the **CompiledMethod** is found. If it is never found then the receiver has obviously been sent a message it does not understand; in which case this is an error. (See Figure 16)

Figure 16 Looking up a method in the receiver's class

Our execution monitoring system deals intimately with **CompiledMethods** and **MethodDictionaries**. It has to do this to avoid leaving any traces of the monitoring session in the Envy source code control system. Our execution monitoring system compiles its own methods and inserts them into the **MethodDictionaries**. All this is covered in later chapters.

8. How the Events Flow Processor works

This chapter and the following chapters talk about how our *proof of concept* prototype was implemented. This chapter starts by describing the **Events Flow Processor**.

The **Events Flow Processor** is part of the Smalltalk *application monitoring system* (see Figure 7). It reads in and processes the **Events Flow File** produced by the Objectory **Events flow Generator**. The **Events Flow File** specifies which messages and objects should be encountered during the monitoring session. It is the job of the **Events Flow Processor** to take this information and set up the environment for the **Execution Monitor** to do its job during the monitoring session.

For each *method enter event*, the **Events Flow Processor** creates and inserts a *method enter trap* into the application's code. These traps will be triggered later during the actual monitoring session. The **Events Flow Processor** also saves the descriptions, given in the **Events Flow File**, of the expected sender/receiver objects and methods. All this information is captured and passed along to the **Execution Monitor**.

In our prototype, a class called **EventsFlowProcessor** was created. It encapsulates the functions of the **Events Flow Processor**. The following sections describe, in more detail, the processing of the **Events Flow File** and the setting up of the environment for a monitoring session. They also go into details about the implementation of our execution monitoring system.

8.1 Processing the Events Flow File

This section discusses how the **EventsFlowProcessor** processes *method enter events* in the **Events Flow File**. The **EventsFlowProcessor** reads in the file and processes it line by line. Each line specifies a *method enter event* which should occur during the execution of the monitored application. The *method enter event* describes the sender object, the receiver object, and the message sent (see chapter 6.)

The receiver method, described in the *method enter event*, corresponds to an actual Smalltalk method in the receiver class. The **EventsFlowProcessor** will locate the corresponding Smalltalk method and insert a *method enter trap* into it. This *method enter trap* will be triggered when the receiver method is entered during the monitoring session. A class called **MethodEnterTrap** was created for this purpose (see section 8.2). An instance of this class is created by the **EventsFlowProcessor** for each *method enter event* to handle the insertion of the *method enter trap*.

Each *method enter event* describes the sender and receiver objects. These are the actual Smalltalk objects which will be participating in the *method enter event*. The **EventsFlowProcessor** captures these descriptions in **TrapObjects**. **TrapObject** class was created for holding information on sender and receiver objects. A separate instance of **TrapObject** is created for each sender and receiver. The **TrapObjects** themselves are stored in **MethodEnterTraps**, along with all the other information concerning the *method enter event*. The **Execution Monitor** uses the descriptions in these **TrapObjects**

to validate that the Smalltalk objects encountered during the monitoring session are the ones we are expecting.

Below is a description of the instance variables found in the **TrapObject**.

- **anyClass:** This is a boolean variable. If true, then the encountered object can be an instance of any class. If false, then the encountered object must be an instance of the class specified by the variable **classOfObject**, shown below.
- **classOfObject:** If the instance variable **anyClass**, shown above, is false then the object encountered during the monitoring session must be an instance of the class specified by this variable.
- **anyObject:** This is a boolean variable. If true, then the object encountered during the monitoring session can be any object (restricted only by the **classOfObject** variable). If false, then the object is given a unique identifier specified by the instance variable **objectID**, shown below.
- **objectID:** If the instance variable **anyObject**, shown above, is false then this variable holds a unique object ID. An object ID is a string of numerics with optional vertical bars (e.g.: 23 or 3600315|241547|12|15). (Chapter 6 talks about object IDs.)

All of the above instance variables combined identify the characteristics of the Smalltalk *object* which the **Execution Monitor** is expecting to encounter during the running of the monitored application. The variables are interpreted as follows:

- If **anyClass** is true and **anyObject** is true then the encountered *object* can be any instance of any class.
- If **anyClass** is true and **anyObject** is false then the encountered *object* can be of any class but must be a specific Smalltalk object. The encountered *object* is assigned a unique identifier given by the variable **objectID**. All encountered *objects* in all message events having the same **objectID** must reference the same Smalltalk object during the application monitoring session.
- If **anyClass** is false and **anyObject** is true then the encountered *object* can be any instance of the class specified by the variable **classOfObject**.
- If **anyClass** is false and **anyObject** is false then the encountered *object* must be an instance of the class specified by the variable **classOfObject**. The encountered object is also assigned a unique identifier given by the variable **objectID**. All encountered *objects* in all message events having the same **objectID** must reference the same Smalltalk object during the application monitoring session.

8.2 Method Enter Trap

For each *method enter event* in the **Events Flow File** a **MethodEnterTrap** is created.

The purpose of this trap is to see to it that the **Execution Monitor** is notified when a monitored method is entered during the running of the application monitoring session. It does this by inserting a *monitor point* into the monitored method. The *monitor point* is similar to a breakpoint; when it is triggered, it notifies the **Execution Monitor**. *Monitor points* are discussed in section 8.3. Aside from inserting monitor points, the **MethodEnterTrap** holds information on the *method enter event*. This section looks at what information is stored in **MethodEnterTraps**.

All the information contained in the *method enter event* in the **Events Flow File**, is stored in the **MethodEnterTrap**. It contains descriptions of the sender and receiver objects, the sender method, the receiver method selector and signature (see chapter 6). All these together form the *method enter event signature*.

The descriptions of the receiver object and method are used to locate the actual Smalltalk method to insert a *monitor point* in. During the monitoring session if a monitored method is entered, the *monitor point* is triggered. The **Execution Monitor** is notified that a *method enter event* has occurred. It uses the information in the corresponding **MethodEnterTrap** to determine if this is the method event we were looking for. The actual Smalltalk sender and receiver objects and methods are compared against the expected ones described in the **MethodEnterTrap**.

As the **EventsFlowProcessor** creates **MethodEnterTraps**, they are added to an indexed list, called the *monitored path*. Their order in the list is the same as the order of their corresponding *method enter events* in the **Events Flow File**. Each **MethodEnterTrap** has its unique index within this list. When a **MethodEnterTrap** creates and inserts a *monitor point*, it provides the *monitor point* with this index. When the *monitor point* is later triggered during the monitoring session, it passes this index number to the **Execution Monitor** (see section 8.3). This allows the **Execution Monitor** to find the corresponding **MethodEnterTrap**. Obviously, the **Execution Monitor** also has access to the *monitored path* list. Figure 17 shows a *monitored path* list.

As mentioned in chapter 6, the events in the **Events Flow File** can be sequenced. In which case, events must occur in the order they appear in the file. This information is also captured in the *monitored path* list. The **Execution Monitor** accesses this information to determine if events are occurring in the proper sequence.

The instance variables in the **MethodEnterTrap** which form the *method enter event signature* are:

- **senderObject**: This variable holds an instance of **TrapObject**, which describes the sender object which will be encountered during the monitoring session.

- **anySenderMethod:** This is a boolean variable. If true, then the sender method, which sent this event's message, can be any method. If false, then the sender method's selector name is held in the variable **senderMethod**, shown below.
- **senderMethod:** If the instance variable **anySenderMethod**, shown above, is false then this variable holds the method's selector name.
- **receiverObject:** This variable holds an instance of **TrapObject**, which describes the receiver object which will be encountered during the monitoring session.
- **receiverMethod:** This variable holds the method's selector name. This is not an optional variable, we need to know which method to insert the *monitor point* in.
- **receiverMethodSignature:** This variable describes what the receiver method's arguments' classes and return value's class should be. This is a mandatory variable; a signature must be assigned.

The above instance variables are used to describe the *method enter event*. There are two remaining instance variables; these deal with the insertion and execution of the *monitor point*.

- **hasOccurred:** This variable is set to false when the **MethodEnterTrap** is created. It will later be set to true by the **Execution Monitor** if the event has occurred during the monitoring session.

- **originalCompiledMethod:** This variable stores the original **CompiledMethod** for the receiver method we are inserting a *monitor point* in. By inserting the *monitor point* we modify and replace the original method. Later, when the monitoring session is ended this variable is used to restore the original **CompiledMethod**. (See chapter 10).

The **hasOccurred** instance variable is only set to true if the actual event occurred during the monitoring session. At the end of the monitoring session, all the **MethodEnterTraps** in the *monitored path* list are checked. Those that did not occur are recorded as errors.

This section described **MethodEnterTraps**. It mentioned that there is one **MethodEnterTrap** for each *method enter event* found in the **Events Flow File**. The **MethodEnterTrap** is responsible for inserting *monitor points*. It also stores the *method enter event signature* which is used by the **Execution Monitor** to validate events during the monitoring session.

8.3 Inserting Monitor Points into Methods

An essential element of our execution monitoring system is the *monitor point*. A *monitor point* is similar to a breakpoint found in most debugging environments. It is inserted at the beginning of a Smalltalk method. During the application's monitoring session, when the *monitor point* is encountered, the application's execution is interrupted and the **ExecutionMonitor** is called to do some processing. After the **ExecutionMonitor** is finished processing, the suspended application resumes its execution. (The

ExecutionMonitor class implements most of the behavior of our **Execution Monitor** and will be covered in chapter 9). This section looks at how *monitor points* are inserted.

As mentioned in a previous section, monitor points are inserted by **MethodEnterTraps**. These **MethodEnterTraps** are only interested in detecting when a method is first entered. To ensure that the *monitor point* is triggered on method entry, it is inserted before the method's first executable statement. In our *proof of concept* prototype, the insertion algorithm will insert a *monitor point* after the opening comments and the temporary variables declarations, but before the first executable statement.

Let's look at how the *monitor point* is actually implemented. The *monitor point* is a single Smalltalk statement, which is composed of a message send to the **ExecutionMonitor** class. This message, when received by the **ExecutionMonitor**, is treated as a *method enter event* (see section 9.1).

The reason the message is sent to the **ExecutionMonitor** class is that classes are globally visible. Remember that the *monitor point* is implemented as a Smalltalk statement that is inserted and compiled into the method. The **ExecutionMonitor** class is visible from any active method context, but an instance of it wouldn't be. Besides, at the time *monitor points* are being inserted, an instance of the **ExecutionMonitor** has not been created yet. Therefore, we use the class name in the statement to make sure it compiles.

The actual Smalltalk statement is composed and inserted by the corresponding **MethodEnterTrap**. The syntax of the message is:

ExecutionMonitor methodEnter: *thisContext* **callBackValue:** *trapIndex* .

The first argument “*thisContext*” is a Smalltalk pseudo variable that contains the current **MethodContext**, within which the monitored method is executing (see section 7.3). The second argument “*trapIndex*” is an integer value. This integer, is the index of the corresponding **MethodEnterTrap** in the *monitored path* list (See section 8.2).

Figure 17 shows how the original method source code has a *monitor point* inserted into it.

The modified source code is referred to as the *Monitored Source Code* in the figure, and shows the inserted statement: **ExecutionMonitor methodEnter:** **thisContext** **callBackValue:** 2. The second parameter “2” refers to the index position of the corresponding **MethodEnterTrap** in the *monitored path* list.

Figure 17 `MethodEnterTrap` inserting a monitor point in the source code

It is possible to have more than one *method enter event* for the same monitored method.

Therefore, it is possible to have more than one *monitor point* inserted into the same method. They would all execute independently of each other and be processed independently.

8.3.1 Primitive Methods

There are a small number of methods in Smalltalk known as *primitive methods*. These are built into the virtual machine. They allow access to the underlying environment and virtual machine structures. They are used to interact with the environment upon which the virtual machine is built (e.g.: operating system, file system)²⁹. Methods such as `+` for small integers and `at:` for objects with indexed instance variables use primitive methods.

Primitive methods are treated differently by the Smalltalk environment. They are invoked differently than other methods and their source code is inaccessible³⁰. If their source code can not be modified, we can not insert a *monitor point* in them. Therefore, our execution monitoring system cannot monitor primitive methods.

Programmers can include calls to primitive methods in their own code. There are over 100 primitive methods in Smalltalk and all have the syntactic form: “<primitive #>“, where # is the integer assigned to the primitive method. When a primitive is used in a regular method, it must be the first executable statement in that method. In such methods, *monitor points* must be inserted after the primitive (see Figure 18). In our *proof of concept* prototype, the *monitor point* insertion algorithm handles methods with primitives correctly.

Figure 18 Monitor points in methods that call primitives

We can only partially monitor methods that call primitives. As shown in Figure 18 the code appearing after the <primitive #> statement is referred to as **Failure Code**. If the primitive exits successfully, then the entire method is exited, and the **Failure Code** is never executed. Therefore, our monitor point code would never be executed. However, if the primitive were to fail, then the **Failure Code** is executed. In this case, our monitor point code would be executed.

Our execution monitoring system can only monitor methods with primitives if the primitive exits unsuccessfully.

These last two sections dealt with inserting *monitor points* into methods. *Monitor points* behave similarly to breakpoints; interrupting the monitored method's execution by sending a *method enter event* message to the **Execution Monitor**. Their purpose is to notify the **Execution Monitor** when a monitored method is entered. This is why they are inserted before the first executable statement, the exception being for methods with primitives. In the later case, *monitor points* are inserted right after the primitive.

8.4 Recompiling the Monitored Method

This section deals with recompiling the monitored method after the *monitor point* code has been inserted into it. We then look at replacing the original compiled method, in the class's method dictionary, with the new one. All this is done without leaving any trace in Smalltalk's source code control system, Envy.

Producing a compiled method with the newly inserted *monitor point* is very tricky. This is because we do not want our modifications to show up in the Smalltalk source code control system. Our application monitoring system should be as unobtrusive as possible; leaving no traces of itself in the user's code.

Our *proof of concept* prototype uses the source code control system Envy (see section 3.1). Most large software projects use a source code control system. Since Envy is the most popular one for Smalltalk, it is used in this research.

If you were to just modify the code and invoke the regular Smalltalk compilation methods, the changes would be recorded in Envy. A new edition of the method would be created. We didn't want the insertion of *monitor points* to create new method editions. The solution was to bypass part of the Smalltalk compilation mechanism and modify other parts. Special compilation methods were written and some small modifications were made to an existing class.

The **MethodEnterTrap** orchestrates the whole recompilation and replacement process. It inserts the monitor points, recompiles the method and then replaces it in the class's **MethodDictionary**. Figure 19 shows part of the actual code in **MethodEnterTrap**.

The first step to recompiling a method is to find its source code. As shown in Figure 16, the **CompiledMethod** can be found in the **MethodDictionary** of its class. The method's selector is used as a key when searching the **MethodDictionary**. There already exists a

method, **findSelector:**, when sent to any class, will return an array containing both the actual class or super class which contains the method and the **CompiledMethod** itself (See lines 1 to 3 in Figure 19). Note that the **MethodEnterTrap** already knows the “receiverClass” and “selector” on line 1.

```

:
1. tmpArray := receiverClass findSelector: selector.
2. aClass := tmpArray at: 1.
3. originalCompiledMethod := tmpArray at: 2.

4. sourceCode := originalCompiledMethod sourceString.
5. modifiedSourceCode := "insert monitor point statement into sourceCode string. (not shown)."
6. newCompiledMethod := aClass envyCompile: modifiedSourceCode notifying: nil ifFail: [^nil].

7. (aClass methodDictionary) at: selector put: newCompiledMethod.
8. aClass flushVMmethodCacheEntriesFor: selector.

9. newCompiledMethod isMonitored: true.
10 newCompiledMethod monitoredSourceCode: modifiedSourceCode.

:

```

Figure 19 Recompilation and replacement of a monitored method

Line 4 shows how the original method’s source code string is obtained by sending message **sourceString** to the **CompiledMethod**. After the *monitor point* is inserted, the method is recompiled within the context of its class (see line 6). This will produce a

CompiledMethod, but will not add it to the source code library. Note the use of the **envyCompile** method on line 6, which is part of the Envy system.

The original compiled method is replaced by the new **CompiledMethod** in the class's **MethodDictionary** on line 7. There is one more step necessary to ensure that the old compiled method is not being used. We make sure the old compiled method is flushed out of the Smalltalk virtual machine method cache on line 8. This is necessary because Smalltalk will cache methods it has already executed, for performance reasons. If you don't remove that method from the cache, then it will keep executing the old version. Now, the next time that method is called, its new **CompiledMethod** will be gotten out of the **MethodDictionary**, executed and cached.

Note that none of the recompilation steps described above effects the Envy source code library. Since we are not storing modified source code in Envy, we need to store it some place else. The decision was made to store the modified source code directly in its **CompiledMethod**. This is a convenient place to store it, since the **CompiledMethod** must have access to it (see line 4). Besides, other changes relating to accessing source code were already being made to the **CompiledMethod** class. The **CompiledMethod** class was extended by adding two new instance variables:

- **isMonitored**: This is a boolean variable. If true, then this compiled method has a *monitor point* inserted in it. If false or nil, then it does not.

- **monitoredSourceCode**: If the instance variable **isMonitored**, shown above, is true then this variable holds the new source code string for this method. This source code string will contain a *monitor point*.

Lines 9 and 10 of Figure 19 show **isMonitored** is set to true, and **monitoredSourceCode** being assigned the modified source code. There also had to be a modification made to **CompiledMethod**'s instance method **sourceString**, used on line 4. **sourceString** now checks **isMonitored**. If **isMonitored** is true then it returns the contents of **monitoredSourceCode**, if false then it uses the existing logic to extract the source code from Envy.

There is one more step in our recompilation routine that is not obvious, but shown on line 3. When the original compiled method is gotten from the **MethodDictionary** it is stored in the instance variable **originalCompiledMethod**, belonging to the **MethodEnterTrap**. This variable is used after the monitoring session is over. As part of the clean up, the monitored methods are reset to their original code. To do this, we need to keep the original **CompiledMethod** around so that it can be restored in the class's **MethodDictionary**.

Figure 20, below, depicts a method that has been replaced by a monitored method in its class's **MethodDictionary**. Note that the original **CompiledMethod** is kept track of by the **MethodEnterTrap**.

Figure 20 Replacing a method with a monitored method in a MethodDictionary

9. The Application Monitoring Session

The previous chapters described how the **Events Flow Processor** read in the **Events Flow File** and prepared the Smalltalk environment for the **Execution Monitor** to do its work. This chapter talks about how the **Execution Monitor** does its job during the monitoring session. *Method exit events* will also be covered.

9.1 Monitoring a Running Application

Once the **Events Flow Processor** has inserted all the *method enter traps*, the application is ready to be run through the monitoring session. The **Execution Monitor** is passive; it is not an active process that runs at the same time as the application being monitored. The **Execution Monitor** is not a stand-alone application at all; it can not run on its own.

It is the *monitor points*, within the running application, that invoke the **Execution Monitor**. The **Execution Monitor** is only active long enough to process method enter and exit events. The **Execution Monitor** does keep some state information between invocations. This state information is primarily to keep track of method traps waiting to be sprung and method events that have occurred.

A new class called “**ExecutionMonitor**” was created to perform the main functions of our **Execution Monitor**. In our *proof of concept* prototype there is only one **Execution Monitor** in the system; therefore, there is only a single instance of this class created. This single instance is created before the monitoring session and exists for the entire session.

Although this single class instance does most of the work, the class itself still is involved during the monitoring session. Remember that the *monitor point* sends the notification message to the **ExecutionMonitor** class (see section 8.3). It is the class that passes this message on to its instance. Since there is only one class instance, it is obvious whom to pass the message to. A future enhancement could be to support multiple **Execution Monitors**, each one specialized (e.g.: monitoring messages, memory usage, etc.). In this case, we would have multiple instances of the **ExecutionMonitor**. The **ExecutionMonitor** class would then have to route messages to the appropriate **ExecutionMonitor** instance.

The reason the message is sent to the **ExecutionMonitor** class is that classes are globally visible. Remember, the *monitor point* is implemented as Smalltalk statement that is inserted and compiled into the method. The **ExecutionMonitor** class is visible from any active method context, but an instance of it wouldn't be.

When an **ExecutionMonitor** is notified of a *method enter event*, the **Execution Monitor** first validates this event's signature against a *method enter event signature* we are expecting. If this is an expected event, then it is recorded and the corresponding **methodEnterTrap** is marked as having occurred.

For each *method enter event* which occurs, the **ExecutionMonitor** captures its *method enter event signature* in a separate instance of a new class called **MethodEnterEvent**.

The *method enter event signature* consists of the sender and receiver objects, the sender

method, and the receiver method. The **MethodEnterEvent** is described later in this section.

There are three pieces of information that the **ExecutionMonitor** retains for the duration of the monitoring session. These are implemented as instance variables:

- **monitoredPath:** This is a list of all the **MethodEnterTraps** built by the **Events Flow Processor** (see section 8.2). Before starting the monitoring session, this list is passed into the **ExecutionMonitor**. During the monitoring session, the **Execution Monitor** checks the *method enter events*, it receives, against this list.
- **recordedPath:** This is a list of **MethodEnterEvents** and **MethodExitEvents** recorded by the **Execution Monitor** during the monitoring session. The *method enter* and *exit events* are recorded in the order in which they occur.
- **encounteredObjects:** This is a dictionary containing objects encountered during the monitored application's run. These objects are identified when the **ExecutionMonitor** processes a method enter or exit event. They can be sender or receiver objects. Each object is assigned unique identifier key in the dictionary.

Let's now take a closer look at how the **ExecutionMonitor** processes method events. The actual message sent from a *monitor point* to the **ExecutionMonitor** class is:

methodEnter: *aMethodContext* **callBackValue:** *trapIndex*.

See section 8.3 for an explanation of the message syntax.

The *aMethodContext* parameter is the active method context (*thisContext*) of the monitored method that triggered the *method enter event*. The **ExecutionMonitor** uses *aMethodContext* to build an instance of a **MethodEnterEvent** object. The **MethodEnterEvent** is a new class that was created to hold information on the actual *method enter events* which occur during the monitoring session. The information it holds is similar to the **MethodEnterTrap** object described in section 8.2. A new instance is created for each *method enter event*. The **MethodEnterEvent** extracts information from the *aMethodContext*. We get the sender and receiver objects, the sender and receiver methods, and we get the arguments passed into the receiver method.

The second parameter to the method “**methodEnter: callBackValue:**”, *trapIndex*, is used to locate the corresponding **MethodEnterTrap** for this *monitor point* (see section 8.2). If you recall, at the time the *monitor point* was inserted it was given the *trapIndex* of the **MethodEnterTrap**. This is an index into the *monitored path* list of **MethodEnterTraps**. It is this *trapIndex* which is passed in as the second parameter.

The **ExecutionMonitor** finds the corresponding **MethodEnterTrap** using the *trapIndex*, and compares it to the new **MethodEnterEvent**. If the *method enter event signature* in the **MethodEnterEvent** matches the *signature* in the **MethodEnterTrap**, then this is the event we were looking for. In which case, the **MethodEnterEvent** is appended to the **recordedPath** list found in the **ExecutionMonitor**. If the *method enter event signatures* don't match, then the event is ignored and nothing happens.

Just because the **ExecutionMonitor** has received a *method enter event*, doesn't mean that it is the event we are looking for. A *monitor point* will trigger a *method enter event* anytime the monitored method is entered. The **ExecutionMonitor** must then validate it against the expected **MethodEnterTrap**. In the case where the events read in from the **Events Flow File** are sequenced, then it must match the next **MethodEnterTrap** in the sequence. If there is no sequence to the expected events, then the **MethodEnterEvent** need only match the **MethodEnterTrap** at the given *trapIndex*.

. Below are the rules for comparing a **MethodEnterEvent** with a **MethodEnterTrap**:

1. If the **MethodEnterTrap** indicates that the sender object's class can be any class then go on to comparison 2. Otherwise, verify that the sender object's class in the **MethodEnterEvent** matches the one given in the **MethodEnterTrap**. If it does not match, then fail the comparison.
2. If the **MethodEnterTrap** indicates that the sender method can be any method then go on to comparison 3. Otherwise, verify that the sender method in the **MethodEnterEvent** matches the one given in the **MethodEnterTrap**. If it does not match then fail the comparison.
3. If the **MethodEnterTrap** indicates that the receiver object can be any object then go on to comparison 4. Otherwise, verify that the receiver object in the **MethodEnterEvent** matches the one given in the **MethodEnterTrap** (see section 9.2). If it does not match then fail the comparison.

4. If the **MethodEnterTrap** indicates that the sender object can be any object then go on to comparison 5. Otherwise, verify that the sender object in the **MethodEnterEvent** matches the one given in the **MethodEnterTrap** (see section 9.2). If it does not match then fail the comparison.
5. Verify that the receiver object's class in the **MethodEnterEvent** matches the one given in the **MethodEnterTrap**. If it does not match then fail the comparison. This comparison is necessary because methods can be inherited by subclasses.
6. If we have made it this far, then the comparison is successful and this **MethodEnterEvent** is the one we are looking for.

MethodEnterEvents that do not pass the above comparison are ignored, the **ExecutionMonitor** exits, and the monitored application resumes execution.

If the **MethodEnterEvent** matches the **MethodEnterTrap** it is appended to the **recordedPath** list in the **ExecutionMonitor**, and the monitored application resumes execution.

Below is a description of the instance variables in a **MethodEnterEvent**. These instance variables captures information on the actual *method enter event signature*:

- **senderObject:** This is an instance of **EncounteredObject** (see below) and holds information about the sender object. The sender object is the Smalltalk object which sent the message that triggered this *method enter event*.
- **senderMethod:** This variable holds the method selector name of the sender's method. This is the method from which the message was sent that triggered this event.
- **receiverObject:** This is an instance of **EncounteredObject** (see below) and holds information about the receiver object. The receiver object is the Smalltalk object which received the message that triggered this *method enter event*.
- **receiverMethod:** This variable holds the method selector name of the receiver's method. This is the method which received the message that triggered this event.
- **numArgs:** number of arguments the receiverMethod takes.
- **args:** An array holding the actual objects passed in as arguments to the receiver method. They appear in the same order as they do on the stack for the given method context.

The information on the actual sender and receiver objects is captured in a new class called **EncounteredObject**. For each sender and receiver object a new instance of

EncounteredObject is created. An **EncounteredObject** has the following instance variables:

- **actualObject**: This references the actual Smalltalk object itself.
- **objectID**: This variable eventually gets assigned an identifier, which will become the unique identifier for the actual object. (Note, if the corresponding object in the corresponding **MethodEnterTrap** has an identifier then that identifier is assigned to this variable).

Figure 21 shows a **MethodEnterEvent** and its **EncounteredObjects**.

Figure 21 MethodEnterEvent and EncounteredObjectsDictionary

This chapter looked at the **Execution Monitor** and how it works. We discussed the **ExecutionMonitor** class, which implements most of the logic. For each *method enter event* the **Execution Monitor** creates an instance of **MethodEnterEvent**, to hold the *method enter event signature*. If the *method enter event signature* in the **MethodEnterEvent** matches the one in the corresponding **MethodEnterTrap**, then it is recorded.

9.2 Objects Encountered During the Monitoring Session

The **Execution Monitor** keeps track of all relevant sender and receiver objects it has encountered in an *encountered objects dictionary*. A new class called **EncounteredObjectsDictionary** was created for this purpose. The **ExecutionMonitor** keeps a reference to this dictionary via its instance variable **encounteredObjects**.

The keys in the **EncounteredObjectsDictionary** are the unique object identifiers assigned to the encountered objects (see section 9.1). The corresponding values are the actual Smalltalk objects that are the senders and receivers. Figure 21 shows an **EncounteredObjectsDictionary**. There is only one encountered objects dictionary used during the entire monitoring session.

Let's look at why it is necessary to keep track of encountered objects, and assign unique identifiers to them. Chapter 6 described the **Events Flow File** format, and the *method enter event signature*. Each event identified its sender and receiver objects by giving them an identifier. In some cases the identifier was “*ANY*”; this meant it would match any

object. If there was a specific identifier, then it will be assigned to an encountered object by the **ExecutionMonitor**.

The **EncounteredObjectsDictionary** maps unique object identifiers to actual Smalltalk objects. The object identifier can be a string of numerics with optional dashes. This object identifier serves two purposes: 1) It represents the actual object outside of the Smalltalk environment. It is this object identifier that is read in from the **Events Flow File**, and it is written out to the execution **Log File**. 2) The **Execution Monitor** uses it to cross-reference the actual object against what the **MethodEnterTrap** expects. Remember that **MethodEnterTraps** are created before the monitoring session; so we have not encountered any actual Smalltalk objects yet. We need some way to represent the same object in multiple **MethodEnterTraps**.

The previous section described the comparison rules to validate that the *method enter event signature* in the **MethodEnterEvent** matches the one in the **MethodEnterTrap**. Steps 3 and 4 talked about verifying that the receiver/sender object in the **MethodEnterEvent** is the same object that is specified in the **MethodEnterTrap**. Below is an example of how such a comparison is done on the sender object:

1. The **senderObject** variable is gotten from the **MethodEnterTrap**. This **senderObject** is an instance of **TrapObject** and acts as a proxy for the actual Smalltalk object we are expecting to encounter. This **TrapObject** proxy contains a description of the actual object we are expecting. It gives the expected object's

class. It also gives its **objectID**. This **objectID** is the same as is used for the keys in the **EncounteredObjectsDictionary**.

2. The **EncounteredObjectsDictionary** is searched using the **objectID** key. If a matching value object is found, then it is compared to the actual sender object found in the **MethodEnterEvent**. Both must be the same Smalltalk object, since both will have the same **objectID**.
3. If both objects are the same then the **senderObject** in the **MethodEnterEvent** is assigned the **objectID** found in the **MethodEnterTrap**'s **senderObject** (Note, that the **MethodEnterEvent**'s **senderObject** points to an **EncounteredObject**). If both objects are not the same then this **MethodEnterEvent** is not the event we are looking for.

At the second step, if there was no entry matching the **objectID** key in the **EncounteredObjectsDictionary**, then this *method enter event* is the first time the **Execution Monitor** encounters that **objectID**. If it is the expected class, then the actual object found in the **MethodEnterEvent**'s **senderObject** is assigned the **objectID** found in **MethodEnterTrap**'s **senderObject**. This new (**objectID** key, actual Smalltalk object value) pair is added to the **EncounteredObjectsDictionary**. From now on, any future Smalltalk objects encountered, which correspond to that **objectID**, must match that actual Smalltalk object in the dictionary.

Figure 22 shows how an `objectID` is used throughout the various stages of application monitoring. In the **Events Flow File**, `objectID_1` appears as the receiver object's identifier in the first entry. The **MethodEnterTrap**, which is created from this entry, is also shown in the figure. The **MethodEnterTrap**'s `receiverObject` variable points to a **TrapObject**. This **TrapObject** represents `objectID_1` and acts as a proxy for the expected Smalltalk receiver object. Later during the monitoring session when the actual *method enter event* occurs, an instance of **MethodEnterEvent** is created. Its variable `receiverObject` points to an **EncounteredObject**. This **EncounteredObject** is assigned the identifier `objectID_1`, because it corresponds to the receiver object found in the **MethodEnterTrap**. Finally, a new (key, value) entry is placed in the **EncounteredObjectsDictionary**. This entry maps `objectID_1` to the Smalltalk object, which was the actual receiver object.

In Figure 22, the second event in the **Events Flow File** also refers to `objectID_1`. A **MethodEnterTrap** will be created for this event. There will be a corresponding **MethodEnterEvent** created, when this event occurs during the monitoring session. It is important to note that its **EncounteredObject** representing `objectID_1` must reference exactly the same Smalltalk object as in the first event. This will also be true for all other method events in which `objectID_1` participates. All their **EncounteredObjects** representing `objectID_1` must reference exactly the same Smalltalk object.

Figure 22 objectID from Events Flow File to EncounteredObjectsDictionary

This section describes how the **EncounteredObjectsDictionary** maps object identifiers, gotten from the **Events Flow File**, to actual Smalltalk objects encountered during the monitoring session. Once an object identifier has a mapping, that mapping is used to validate future *method enter events* which occur during the monitoring session. In all future events, that object identifier must resolve to that same Smalltalk object for those events to be accepted.

9.3 Recording Method Arguments

This section talks about how the **Execution Monitor** uses the *evaluation stack*, found in a **MethodContext**, to capture the message arguments. As mentioned earlier, when the

program enters a monitored method, the *monitor point* invokes the **ExecutionMonitor**.

The monitored method's active context **thisContext** is passed into the **ExecutionMonitor** (See section 8.3).

The **MethodContext** passed into the **Execution Monitor** contains the state of the interpreter at the time it entered the monitored method. The evaluation stack, at the time that context was suspended, is part of this state (see Figure 15). On this stack you can find the arguments which were passed into the monitored method.

The evaluation stack in Smalltalk is implemented as an instance of **Array**. When a method is invoked, a new **MethodContext**, with a new evaluation stack is created. If the method takes n arguments, then the 1st n elements on the stack will contain the arguments in the order they appear in the message signature.

The number of arguments that a method takes can be gotten from the method's corresponding **CompiledMethod**. The **ExecutionMonitor** can get a monitored method's **CompiledMethod** by asking the given **MethodContext** for it.

Figure 23 shows some sample code, which gets the **CompiledMethod** and then the evaluation stack from a **MethodContext**. It then proceeds to extract the method's arguments from the evaluation stack.

```

:
:
aCompiledMethod := aMethodContext method.
anEvaluationStack := aMethodContext stack.
numOfArguments := aCompiledMethod numArgs.
1 to: numOfArguments do: [:index |
    (anArgument := anEvaluationStack at: index)].

```

Figure 23 Getting the method's arguments from the given MethodContext

9.4 Capturing the Method's Return Value

This section will describe how the **Execution Monitor** captures the value returned by monitored methods. When methods exit, they always return an object to their caller. In the method's source code, you can specify which object to return using the caret symbol (^) followed by the object. Other languages like C++ use the more familiar keyword "return". In Smalltalk, if the method does not have a return statement, it will by default return the receiver object which is executing the current message (**self**)³¹. Either way, there is always a return value for the **Execution Monitor** to capture.

Up until this section we have only dealt with *method enter events*, we now introduce the *method exit event*. The *method exit event* is similar to a *method enter event*, except that it deals with exiting rather than entering a monitored method. Though the **Events Flow File** generated from Objectory only contains *method enter events*, the **Execution Monitor** has

been extended to capture a method's exit. This allows the execution monitoring system to record a much more detailed account of the application's execution. Since we set *method enter traps* to capture *method enter events*, we also set *method exit traps* to capture *method exit events*. These will be covered in the next few sections.

The problem of trapping *method exit events* was solved differently from trapping *method enter events*. Inserting *monitor points* to trigger on method exit was too complicated. A method may have several points of exit throughout its body of the code. To insert *monitor points* to trap the method exit you would have to check all its conditional paths for return statements. In some cases, the method may not have an explicit return statement, but just finish. In the latter case, **self** is returned.

Looking at Figure 14, we can spot a simple, fast and efficient mechanism to trap a method's return value. In Figure 14 when method_3 exits, its method context (MethodContext_3) relinquishes control to the *sender* method context (MethodContext_2). If we inserted a *trap MethodContext* between MethodContext_3 and MethodContext_2, we could execute our own *trap* method. This *trap* method could record the return value from method_3 and itself return with that value. Method_2 would not be able to tell that the return value it received did not come directly from method_3.

The insertion of a *trap MethodContext* is the approach taken by our **Execution Monitor**. When a monitored method is entered during the monitoring session, it triggers a call to the **ExecutionMonitor**, passing in the pseudo variable **thisContext**. One of the

tasks performed by the **ExecutionMonitor** is to create a trap **MethodContext** and insert it between the given method context and its sender as shown in Figure 24. A new class called **TrapMethodContext** was created for this purpose. The **TrapMethodContext** will be described in more detail in section 9.4.1.

Figure 24 Inserting a **TrapMethodContext** to capture **method_3**'s return value

In our example above, **method_3** is a monitored method. During the application's run **method_3** is entered and its *monitor point* calls the **ExecutionMonitor**, passing in its active context, **MethodContext_3**. The **Execution Monitor** creates a **TrapMethodContext**, and sets its sender variable equal to **MethodContext_3**'s sender variable, which in this case is **MethodContext_2**. The **Execution Monitor** then resets **MethodContext_3**'s sender variable to point to the new **TrapMethodContext**.

Now, when `method_3` exits, our **TrapMethodContext** takes control, processes the return value and passes it on to `method_2`.

9.4.1 TrapMethodContext

This section looks at how the **TrapMethodContext** is implemented and how it is used to capture the return value.

The **TrapMethodContext** is a subclass of Smalltalk's **MethodContext**. This subclass adds only one new public instance method to allow the method context's **program counter** to be set. (Section 7.3 explained that the program counter points to the next bytecode to be executed within that context).

Unlike a regular **MethodContext**, the **TrapMethodContext** is not created by the interpreter to execute a method from the beginning. A **TrapMethodContext** is created by the **Execution Monitor** to execute only the tail end of a very special method called **returnTrap**. This method was created for processing return values.

Figure 25 returnTrap source code and bytecode

Figure 25 shows **returnTrap**'s source code and the resulting compiled bytecode. The source code shows that the first statement is a *self halt*, but this statement is never executed. The reason it is included is to make it easy to produce a **MethodContext**, whose execution is suspended at line 4 of the bytecodes. When execution resumes, it will resume at line 4, as shown in Figure 25 by the instruction pointer **pc**. If we resume executing at bytecode line 4, it will pop the value off the stack and assign it to **tmpReturn** (local 0 in figure). If this is confusing, just remember what is happening as the interpreter is executing the bytecodes of a method and it encounters a message send within the method. The current active method context is suspended, and part of that context is an instruction pointer that points to the next bytecode instruction to execute. When the message returns, the suspended context resumes execution at the instruction

pointed to by the stored instruction pointer (**pc**). In Smalltalk, a return value is pushed on the stack just before a method returns; therefore, the next pop instruction will retrieve it.

When the **Execution Monitor** creates a **TrapMethodContext** the **method** variable is set to point to **returnTrap**'s **CompiledMethod** and the instruction pointer (**pc**) is set to 4 (see 26). Looking back at Figure 24, we see that when **method_3** returns, **returnTrap** resumes execution from line 4 of the bytecodes. The first thing it does is capture the return value in the temporary variable **tmpReturn**, then calls the **Execution Monitor** to process it. **returnTrap** then exits, returning with **tmpReturn**. **Method_2**, which has been waiting for **method_3** to return, gets the value **tmpReturn** from **returnTrap** and continues executing.

Figure 26 TrapMethodContext and MethodExitTrap

This section explained that the **TrapMethodContext** is a subclass of **MethodContext**. It is inserted between the monitored method's **MethodContext** and sender **MethodContext**. The **TrapMethodContext** is setup to execute a method called **returnTrap**. When the monitored method exits, the **returnTrap** method captures the return value, processes it, and passes it on to the sender method.

9.4.2 Processing the Return Value

This section talks about how the **Execution Monitor** processes the captured return value. Before we can go into the *method exit event*, we must backup to the insertion of the method exit trap.

It was mentioned earlier that the **ExecutionMonitor** creates and inserts the **TrapMethodContext**; this is not entirely accurate. The **ExecutionMonitor** creates an instance of a class called **MethodExitTrap**, which handles the actual creation and insertion of the **TrapMethodContext**. A separate instance of **MethodExitTrap** is created for each valid *method enter event*.

The **returnTrap** method, described in the last section, is an instance method of **MethodExitTrap**. This method's purpose is to notify the **Execution Monitor** that a *method exit event* has occurred. The notification message passes two things: 1) **self**; 2) the return value from the monitored method. (See line 2 of the source code in Figure 25). In this case "**self**" will be the instance of **MethodExitTrap** that inserted the **TrapMethodContext** (see 26).

The reason the **MethodExitTrap** is passed into the **Execution Monitor** is to help with the processing of the monitored method's return value. The **MethodExitTrap** knows the *expected* class the return value should have, and the corresponding *method enter event*.

At any time in the running of the monitored application, we may have several outstanding *method exit traps* waiting to be sprung. When one is sprung, we match it with the corresponding *method enter event*. By pairing up these enter and exit events we get an accurate picture of the application's execution. Unfortunately, our **Execution Monitor** does not log the exit events. This is because we wanted to keep the **Log File** similar to the

Events Flow File generated from Objectory. Chapter 14 talks about enhancing the system to log *method exit events*.

When the **Execution Monitor** receives notification of the *method exit event*, it first verifies the given *return value*. The class of the given *return value* must match the class specified in the receiver method's signature (See chapter 6). If it does not match, then both the *method exit event* and the corresponding *method enter event* are invalidated. Remember, corresponding method enter and exit events represent a single method execution. Both must match the expected signature.

This section discussed **MethodExitTraps** and how the **returnTrap** method notifies the **Execution Monitor** of a *method exit event*. For each *method exit event*, the **Execution Monitor** verifies that the return value is of the expected class.

10. End of the Monitoring Session

This chapter looks at the tail end of the monitoring session. The **Execution Monitor** is responsible for removing all method enter traps and restoring the monitored methods to their original form. The **Execution Monitor** also produces a **Log File** of *method enter events* which have occurred, and a list of those which have not.

After the monitored application has finished running, the monitoring session is over. The **Execution Monitor** is now invoked by the user to clean up the Smalltalk environment and produce a **Log File**.

Before the monitoring session started, the **ExecutionMonitor** has given a list of method enter traps that were set by the **Events Flow Processor** (see section 8.2). This list contains all the **MethodEnterTrap** objects created. As was mentioned in section 8.4, the **MethodEnterTrap** retains a reference to the original **CompiledMethod** for the corresponding monitored method.

For each **MethodEnterTrap** in the list, the **Execution Monitor** sends it a message telling it to restore the original **CompiledMethod**. The **MethodEnterTrap** replaces that method's entry in its Class's **MethodDictionary** with the original **CompiledMethod**. The same logic is used, as when the trap was installed (see section 8.4).

Now all that is left to do is produce a **Log File**, containing all the *method enter events* which occurred and which did not occurred. The **Execution Monitor** prints out all the

method enter events that are captured in its **recordedPath** list. In the last chapter we mentioned that *method exit events* were also recorded in this list; but these are ignored. The reason being that we want the **Log File** to have the same format as the **Events Flow File** generated from Objectory. Chapter 14 talks about enhancing the system to log *method exit events*.

To print a list of *method enter events* which did not occur, the **Execution Monitor** checks each **MethodEnterTrap** in its **monitoredPath** list. This is the same list gotten from the **Events Flow Processor** before the monitoring session (see section 8.2).

MethodEnterTraps which were not triggered by a valid event during the monitoring session will have their **hasOccurred** flags still set to false. A description of these **MethodEnterTraps** is appended to the **Log File**.

Once the **Log File** is produced, the user may examine it. A clean **Log File** with no errors may also be fed back into the system as an **Events Flow File**.

This chapter described how the **Execution Monitor** cleaned up after the monitoring session and produced a **Log File**. This chapter also concludes the description of our *proof of concept* prototype. The next few chapters will provide sample executions, conclusions, alternate designs, and future enhancements.

11. Sample Monitoring Session

This chapter shows how our *proof of concept* prototype handles a sample interaction diagram.

First the designer would bring up the Objectory tool and create an interaction diagram describing the inter-object behavior of some operation which the application performs.

Figure 27 show a sample diagram.

Figure 27 Sample Objectory interaction diagram

Once the diagram was completed, the designer would run the **Events Flow Generator** on it. This would produce the **Events Flow File** shown in Figure 28.

```

Raise Temperature$
SOURCE Objectory$
EVENTS_SEQUENCED Y$
*ANY*$ANY*$ANY*$Thermostat$36003151241547112115$setTemperature:$Thermostat (SmallInteger)$
Thermostat$36003151241547112115$setTemperature:$TemperatureSensor$36003151241547112116$Temperature$SmallInteger ()$
Thermostat$36003151241547112115$setTemperature:$Furnace$36003151241547112117$startHeating$Furnace ()$

```

Figure 28 Sample Events Flow File

This **Events Flow File** is read in by the **Events Flow Processor**, which prepares the Smalltalk environment for monitoring. For each *message event* a *monitor point* is inserted into the corresponding receiver method (The receiver methods are shown in bold in Figure 28). Figure 29 shows the Furnace's `setTemperature:` method with a *monitor point* inserted as the first executable statement.

```

setTemperature: desiredTemperature
"The current room temperature is read from the temperature sensor. If the current temperature is
lower than the desired temperature, the furnace is started. If it is equal to or greater than, then the
furnace is turned off."

```

```

ExecutionMonitor methodEnter: thisContext callBackValue: 1.

```

```

((temperatureSensor temperature) <= desiredTemperature)
  ifTrue: [ furnace startHeating.]
  ifFalse: [ furnace stopHeating.].

```

Figure 29 Thermostat's `setTemperature:` method with monitor point

Once the **Events Flow Processor** is finished preparing the environment, the application can be run. As it is running, *method events* are occurring. These are verified and recorded by **Execution Monitor**. At the end of the monitoring session, a **Log File** is produced.

```
Raise Temperature$
Source Recording$
EVENTS_SEQUENCED Y$
JamesTest$1$runHeatingApp$Thermostat$36003151241547112115$setTemperature:$Thermostat (SmallInteger)$
Thermostat$36003151241547112115$setTemperature:$TemperatureSensor$36003151241547112116$temperature$SmallInteger ()$
Thermostat$36003151241547112115$setTemperature:$Furnace$36003151241547112117$startHeating$Furnace ()$
```

Figure 30 Log file for a successful monitoring session

If all the events occurred as expected then we have the **Log File** shown in Figure 30. Note that it is identical to the input **Events Flow File**, except that the **Source** indicates it was a monitor session **Recording** rather than generated from **Objectory**. We also have a specific sender object in the first *message event*, rather than *"*ANY*\$*ANY*\$*ANY*"*.

```
Raise Temperature$
Source Recording$
EVENTS_SEQUENCED Y$
JamesTest$1$runHeatingApp$Thermostat$36003151241547112115$setTemperature:$Thermostat (SmallInteger)$
Thermostat$36003151241547112115$setTemperature:$TemperatureSensor$36003151241547112116$temperature$SmallInteger ()$
#Trap number 3 did not occur as expected.
```

Figure 31 Log file with an event that did not occur

Figure 31 shows what the **Log File** would look like if the third *message event* did not occur as expected. We get an error message as the last line.

12. Performance and Scalability

This chapter looks at the performance and scalability of our **Execution Monitoring** system. If our system were to be used on a project, what impact would it have on the application? How would it affect the timing? How would it effect the memory usage?

It should be noted that our *proof of concept* prototype was built to demonstrate this thesis' assertions. Very little optimization went into it. Never the less, it embodies the fundamental characteristics of a *Monitor Point* based execution monitoring system. Therefore, any performance metrics on our prototype are still informative.

Our system does not add extra overhead to the Objectory environment. We only use Objectory's existing features to build the models and generator. However, our system does add overhead to the monitored application. Therefore, this analysis will concentrate on the Smalltalk parts of our system.

12.1 *Setting up the Environment*

All the classes that make up our execution monitoring system add an extra 45.27 Kbytes to the Smalltalk environment. Before the monitored application runs, our **Events Flow Processor** inserts method enter traps (*monitor points*) into the application. Inserting method enter traps (*monitor points*) incurs on average 840 bytes per trap. The amount of memory varies because we create a copy of the monitored method (see section 8.3). For

our performance analysis, we assumed the average method source is about 7 lines of code and 2.25 lines of comment.³²

Figure 32 Memory overhead (bytes) / # of Monitor Points

Figure 32 show the results of our performance analysis. It can be seen that the memory overhead increases linearly with the number of *monitor points*. As expected, each new method monitored incurs the same overhead. In respect to memory requirements, our execution monitor scales very well.

It would be possible implement our **Execution Monitor** to consume less space per monitor point. There are several improvements that can be made. Some examples are: Storing modified source code in a file, rather than in memory. **TrapObjects** could be

reused by multiple **MethodEnterTraps**. **MethodEnterTraps** could be removed from the *monitored path* list once they were sprung.

12.2 Monitoring the Running Application

Once the setup is complete, we can run the monitored application. For each *monitor point* (method enter trap) we encounter, a method exit trap is inserted (see section 9.4).

Therefore, the overhead for a *monitor point* includes the time it takes to process both the method enter trap and its corresponding method exit trap. Our tests show that each monitor point adds approximately 0.04% to the application's run time (3 msec on a P5/133Mz with 48MB).

Figure 33 shows the results of our timing analysis. For the most part, we see the processing time increases linearly as we increase the number of *monitor points*. Although, several runs were made and averages were calculated, we still do not get a smooth graph. This is attributed window95 operating system activities (e.g.: scheduler, background processes, caching). Smalltalk also does its own caching; this will have an effect on the timing as well. These timing blips had a noticeable effect on the shorter application runs. However, with longer application runs, these blips are smoothed out. They still occur, but their overhead is less of a factor.

Figure 33 Processing overhead (msec) / # of Monitor Points

It was expected that the time graph would be approximately linear. The amount of work done to process each *monitor point* is the same. However, as we add *more monitor points* the image size increases. This would eventually have an effect on the caching and swapping. Thus increasing the overhead slightly. This would not have a significant effect until the image was very large. Adding more memory to the computer would help compensate.

Our tests show that the current timing overhead is acceptable. With 0.04% overhead per *monitor point*; even 100 *monitor points* would only slow the application down by 4%. In practice, there probably won't be hundreds of *monitor points* per run. The reason being that there should not be to be hundreds of message events in the corresponding interaction diagram. It is possible to combine interaction diagrams to produce one **Events Flow File**. However, we recommended that the monitoring runs be kept to a manageable size.

13. Conclusions and Future Enhancements

This chapter concludes the research work in this document. It then looks at possible future enhancements to this research.

13.1 Conclusions

This thesis shows that CASE tools can be used to automate the design/implement/test iterative loop in the object-oriented software development life-cycle. It also shows how this work can benefit the classical waterfall development model. Both development models can benefit from automating the tedious process of "validating that the implementation and design are synchronized".

CASE tools can capture both an application's static architecture and its dynamic behavior. Objectory provides *design object* editors and *object association* diagrams to model the static architecture (e.g.: class definitions, inter-object relationships). Dynamic design information is modeled using *interaction* diagrams (inter-object behavior) and *state-transition* diagrams (intra-object behavior). This thesis demonstrated that it is possible to extract this information from the design models, and compare it against an implemented application.

This research chose the most difficult part of the design model to validate the inter-object behavior. Unlike the static architecture, it cannot be validated by just looking at the source code. In dynamically typed languages, like Smalltalk, you need to monitor the

running application to determine the inter-object behavior. The intra-object behavior can be validated by comparing the *state-transition* information against the source code and by monitoring the runtime.

There are products already that will reverse engineer the source code and produce the static architecture models (e.g.: Rational Rose). By combining previous work in reverse engineering code and this thesis work, we can validate both the static architecture and dynamic behavior of applications. At the time of this research, there were no products that did this for Smalltalk applications. This thesis shows that it is possible to capture sufficient information in the design model to automate the validation process.

By automating the validation process, we facilitate the design/implement/test iterative loop. This makes it easier for the designer/developer to ensure that the design documentation reflects the implementation, and visa versa. This will reduce bugs and help ensure that the documentation is up to date.

In our *proof of concept* prototype, any discrepancies found during the validation are brought to the developer's attention. (S)he decides whether to update the design models or the source code to remove these discrepancies. Unfortunately, this updating is still manual. This could be automated if the CASE tool, development environment, and our **Execution Monitoring** system were all integrated. Then we would have closure on the design/implement/test iterative loop.

13.2 Future Enhancements

This section looks at some future enhancements that could be made to our **Execution Monitoring** system. These enhancements would further facilitate the design/implement cycle in an iterative development process.

The first thing would be a tighter integration with the **CASE** tool. Our system has a separate tool to extract information from Objectory and a separate tool for monitoring the Smalltalk environment. The **CASE** tools could be extended to interact directly with Smalltalk, thus making it a complete system.

The scope of the automatic validation can be extended to include all the other information which is captured in the design model. Static information like class definitions could easily be verified by examining the Smalltalk source code. Intra-object behavior can be extracted from state-transition diagrams and verified during the application monitoring session.

In an iterative development process, the developer usually does some redesigning during the implementation phase. In this case, it is the design models that must be updated to match the implementation. Our system could be enhanced to automatically update these design models. After the monitoring session the user would be presented with a list of discrepancies and asked if the design models should be automatically updated. In the case where there are no design models, our system could create them for the first time. This is a useful facility to document legacy systems.

Currently, the **Events Flow File** and the **Log File** only have *method enter events*.

However, the **Execution Monitor** does capture *method exit events* (see section 9.4.2).

The system could be easily enhanced to record both in the **Log File**. They could be used to produce a more chronologically accurate interaction diagram. This more detailed **Log File** could also be fed back into the **EventsFlowProcessor** to allow for a more detailed monitoring session. The reason *message exit events* are not recorded in our *proof of concept* prototype, is that Objectory's interaction diagrams don't support message returns.

Currently the *method enter events* in the **Events Flow File** specify the method signature.

The method signature specifies the arguments' classes and return value's class. The signature could be enhanced with object ids for the arguments and return values; just as it is done for the sender and receiver objects. Unfortunately, Objectory's interaction diagrams currently do not support this feature.

Record and playback could be added to our system. During the testing phase, the user's keystrokes and mouse movements could be recorded. The output of the application could also be recorded. Later, regression tests could be automatically run, playing back the user's interactions and comparing the application's output against the original recording.

Currently the user can only view the *recorded message events* in the Log File as plain ASCII text. It would be helpful to provide a viewer tool, which graphically displayed the Log File in the form of an Interaction Diagram. This viewer could also read in the original Events Flow File and graphically show the differences with the Log File.

As mentioned in earlier chapters, our execution monitoring system could be used to automate regression testing. A useful enhancement would be to keep track of which messages were exercised during the monitoring session. This information would be used to determine "code coverage" statistics (how many methods of a particular class have we exercised).

14. Alternative Designs :

This section examines alternative designs for the Smalltalk execution monitoring environment. It examines their strengths and weakness, and compares them to the *monitor point* based design that was described in this thesis.

14.1 Subclassing from Debugger

One of the most prominent alternative designs would have been to subclass the **Execution Monitor** from the Smalltalk debugger. Subclassing from the debugger is an approach taken in another thesis described in section 2.4. After reading the thesis and talking to the author, it was decided there were too many shortcomings.

First of all, bringing up the **Execution Monitor** would require a separate debugger application, which the monitored application would have to run under. Having this debugger application running would mean extra overhead. According to the thesis in section 2.4, this overhead dramatically slowed down applications with *Graphical User Interfaces* (GUI). Trying to process all the GUI messages would even cause the system to crash under long usage.

The debugger was used to set breakpoints and jump through the application which was being monitored. In the VisualWorks Smalltalk debugger, breakpoints are set by inserting 'self halt' statements into the code. We would then have to worry about the source code

being modified in Envy. Remember, we wanted no trace of the monitoring session to exist after it was over.

The debugger approach had a problem capturing the values returned by methods, if the method returned 'self'. This problem was solved by parsing the returning method to determine if the value returned would be self. This alone incurs a large overhead for each message, effecting performance to a noticeable degree.

14.2 Wrapping the Monitored Method

This approach came to light late in the writing of this thesis. It is definitely an alternative to the design that was finally adopted. It involves renaming the monitored method, and inserting a wrapper method in its place. The wrapper method would do all the validation and recording which the current *Monitor Point* design does. It would also call the original method that it had replaced.

In many ways, this is similar to *Monitor Point* design that was finally adopted. There are no extra processes involved, the running application does not know wrapper methods are being used, and it leaves no trace of the wrapper methods in the Envy source code library.

The advantage of the wrapper method design is that it is a little easier to implement. We don't need to insert a *trap method context* into the call stack to trap the return value. This is because the wrapper method can trap the return value when it calls the original method. The return value can be validated, recorded, then passed on up the call stack.

```

1  anyMethod
2  "This is a wrapper method replacing the method called anyMethod"
3  | correspondingEnterEvent tmpReturn |
4  correspondingEnterEvent := ExecutionMonitor methodEnter: thisContext
                           callBackValue: 2.
5  tmpReturn := self anyMethod_Original
6  ExecutionMonitor methodExitFor: correspondingEnterEvent with: tmpReturn.
7  ^tmpReturn

```

Figure 34 Wrapper method

The wrapper method technique works by creating a wrapper method for each method we want to monitor. First, the monitored method is renamed. Then a wrapper method is created and given the original name of the monitored method. Figure 34 shows a wrapper for the method called **anyMethod**. The original **anyMethod** has been renamed **anyMethod_Original**.

When the wrapper method is invoked, it first notifies the **ExecutionMonitor** of the *method enter event* (line 4). The **ExecutionMonitor** determines if it is interested in this event. If it is, then it creates a **MethodEnterEvent** and returns it. The **MethodEnterEvent** is stored in the temporary variable **correspondingEnterEvent**. If

the **ExecutionMonitor** is not interested in this *method enter event*, then it returns **nil**. In line 5 the original method is called and its return value stored in **tmpReturn**. In line 6 the **ExecutionMonitor** is send a method exit event message, to notify it that **anyMethod_Original** has returned. The *method exit event* message pass in the **correspondingEnterEvent** variable. If **correspondingEnterEvent** is **nil** then the **ExecutionMonitor** ignores the *method exit event* message. In line 7 **tmpReturn** is passed back, just as the original **anyMethod** would have done.

The disadvantage of the wrapper method technique is that it has a longer setup time than the *Monitor Point* technique. For each method event in the **Events Flow File**, the **EventsFlowProcessor** must rename the monitored method and create a wrapper method. If there are many methods being monitored, then there will be many wrapper methods created. Although, this has not shown itself to be a problem so far. In fact, all this extra work at setup means less overhead during the monitoring session, compared to the *Monitor Point* technique. The *Monitor Point* technique has to insert a *trap method context* into the call stack to trap the return value. This is done during the running of the monitored application.

There is another disadvantage to the wrapper method. It requires a lot more modifications to the Envy environment. We have worry about creating new methods and rename original ones, without effecting the Envy library. The *Monitor Point* technique need only worry about replacing existing methods.

In retrospect, if our proof of concept prototype were to be redone, we would use the
:
wrapper method technique rather than the *Monitor Point* technique.

References

- ¹ Goldberg, Adele, and Kenneth S. Rubin. Succeeding with Objects. Reading, Massachusetts: Addison-Wesley, 1995.
- ² Booch, Grady. Object Oriented Design with Applications. Redwood City, California: Benjamin/Cummings, 1991.
- ³ Goldberg, Adele, and Kenneth S. Rubin. Succeeding with Objects. Reading, Massachusetts: Addison-Wesley, 1995.
- ⁴ Yourdon, Edward. Decline & Fall of the American Programmer. Englewood Cliffs, New Jersey: Prentice Hall, 1992.
- ⁵ Jacobson, Ivar. Object-Oriented Software Engineering - A Use Case Driven Approach. Reading, Massachusetts: Addison-Wesley, 1992.
- ⁶ Jacobson, Ivar. Object-Oriented Software Engineering - A Use Case Driven Approach. Reading, Massachusetts: Addison-Wesley, 1992.
- ⁷ Jacobson, Ivar. Object-Oriented Software Engineering - A Use Case Driven Approach. Reading, Massachusetts: Addison-Wesley, 1992.
- ⁸ Bilow, Steven C. Journal of Object-Oriented Programming. New York, New York: SIGS Publication, June, 1994.
- ⁹ Hinkle, Bob, and Ralph E. Johnson. Breakpoints revisited. The Smalltalk Report, October, 1995.
- ¹⁰ Hinkle, Bob, and Ralph E. Johnson. ParameterizedCompiler: A case study in making code reusable. The Smalltalk Report, July-August, 1995.
- ¹¹ Alguire, Jeffrey C. On-line Dynamic Views of Applications Written in Smalltalk, M.C.S. Thesis. Ottawa, Ontario: School of Computer Science, Carleton University, 1994.
- ¹² Round-Trip Engineering with Rational Rose/Java Edition. Santa Clara, California: Rational Software Corporation, 1996.
- ¹³ Together/Professional. Stuttgart, Germany: Object International Software, 1997.
- ¹⁴ Probe-ADvance Connect. Elmshorn, Germany: IC&C GmbH, 1997.
- ¹⁵ Goldberg, Adele, and David Robson. Smalltalk-80 The Language and its Implementation. Reading, Massachusetts: Addison Wesley, 1985.
- ¹⁶ Jacobson, Ivar. Object-Oriented Software Engineering - A Use Case Driven Approach. Reading, Massachusetts: Addison-Wesley, 1992.
- ¹⁷ Jacobson, Ivar. Object-Oriented Software Engineering - A Use Case Driven Approach. Reading, Massachusetts: Addison-Wesley, 1992.
- ¹⁸ Rational Objectory Process manual. Santa Clara, California: Rational Software Corporation, 1996.
- ¹⁹ Rational Objectory Process manual. Santa Clara, California: Rational Software Corporation, 1996.
- ²⁰ Objectory - Design, Implementation and Testing. KISTA, Sweden: Objectory AB, 1994.
- ²¹ Objectory - Design, Implementation and Testing. KISTA, Sweden: Objectory AB, 1994.
- ²² Objectory - Design, Implementation and Testing. KISTA, Sweden: Objectory AB, 1994.
- ²³ Objectory - Design, Implementation and Testing. KISTA, Sweden: Objectory AB, 1994.
- ²⁴ Objectory - Design, Implementation and Testing. KISTA, Sweden: Objectory AB, 1994.

-
- ²⁵ Skublics, S., Kilmas E., and Thomas D. *Smalltalk with Style*. Englewood Cliffs, New Jersey: Prentice Hall, 1996.
- ²⁶ Goldberg, Adele, and David Robson. *Smalltalk-80 The Language and its Implementation*. Reading, Massachusetts: Addison Wesley, 1985.
- ²⁷ Goldberg, Adele, and David Robson. *Smalltalk-80 The Language and its Implementation*. Reading, Massachusetts: Addison Wesley, 1985.
- ²⁸ Goldberg, Adele, and David Robson. *Smalltalk-80 The Language and its Implementation*. Reading, Massachusetts: Addison Wesley, 1985.
- ²⁹ Goldberg, Adele, and David Robson. *Smalltalk-80 The Language and its Implementation*. Reading, Massachusetts: Addison Wesley, 1985.
- ³⁰ Goldberg, Adele, and David Robson. *Smalltalk-80 The Language and its Implementation*. Reading, Massachusetts: Addison Wesley, 1985.
- ³¹ Goldberg, Adele, and David Robson. *Smalltalk-80 The Language and its Implementation*. Reading, Massachusetts: Addison Wesley, 1985.
- ³² *VisualWorks 2.0 Cookbook*. Sunnyvale, California: ParcPlace Systems, 1994.
- ³³ Barry, B. Prototyping a Real-Time Embedded System in Smalltalk. *Proceedings of OOPSLA '89, ACM SIGPLAN*. New Orleans:, 1989.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc.
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved