Geppetto: Enhancing Smalltalk's Reflective Capabilities with Unanticipated Reflection

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

David Röthlisberger

2005

Leiter der Arbeit: Prof. Dr. S. Ducasse
Institut für Informatik und angewandte Mathematik

The address of the author:

David Röthlisberger
Neueneggstrasse 9
CH-3172 Niederwangen
roethlis@iam.unibe.ch

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern

Abstract

Reflection is an important tool to extend and modify the semantics or runtime of applications. However, lot of approaches to support reflection are based on up-front fully reflective or load-time based reflection mechanisms. Using these approaches, it is not possible to apply reflective techniques on running systems without stopping them, unless the system is fully reflection which is very costly. Because many applications and systems exist that cannot be halted and stopped but have to be always on and running, such as web applications, real-time systems or mobile systems, the ability to apply reflective features at runtime is a crucial and important property. Our solution to achieve this possibility is unanticipated reflection. With unanticipated reflection, we can design the MOP required for the problem we want to solve, introduce it in the language, activate the reflective mechanisms and possibly remove them once they are not necessary anymore at runtime, without halting the system or the application in which we want to apply and use the reflective mechanisms.

In this thesis we motivate first the need for unanticipated reflection, allowing us to adapt dynamically applications and the system itself. We compare already existing solutions to support reflection in different languages, such as Reflex [TANT 03] or Iguana/J [REDM 02], and learn that these proposals have several shortcomings. Hence we develop our own proposal, an approach to unanticipated reflection in the context of dynamically-typed languages. To provide unanticipated reflection in an efficient as well as expressive way we reuse the partial behavioral reflection model of Reflex. We believe that this model supports best our demands and requirements for unanticipated reflection and fits well into the concept of dynamic languages.

We also propose a concrete implementation of unanticipated reflection by providing a reflective framework for Squeak/Smalltalk, called GEPPETTO, which allows us to install behavioral reflective features unanticipated into the running Squeak system ubiquitously. This reflective framework supporting unanticipated partial behavioral reflection is particulary useful to debug or profile running applications or to temporarily test extensions or adaptations of a system that cannot be halted. Another purpose is the implementation of cross-cutting behavior of an application in a causally connected but separated metalevel, similar as in aspect-oriented programming.

Acknowledgements

I would like to thank Marcus Denker for his ongoing and encouraged work on supporting me during the development of this master thesis. The numerous and valuable discussions with him helped me a lot to find solutions for the different problems popped up during this work. I want also to thank him for his great and important work on BYTESURGEON, a tool without which my work would not have been possible.

I would like to thank my supervisor Prof. Dr. Stéphane Ducasse for his many vital suggestions and hints as well as for reading my writings and giving me crucial feedback. Especially I want to thank him for motivating me to work on this interesting topic.

Furthermore, I would like to thank Prof. Dr. Oscar Nierstrasz, head of the SCG, for giving my the opportunity to work on this thesis in his group.

Finally I would like to thank all the people and friends at the SCG and other places that supported me during this work and didn't stop to encourage and motivate me.

Thank you all, I will never forget.

David Röthlisberger
December 2005

"Our doubts are traitors, and make us lose the good we oft might win by fearing to attempt."
William Shakespeare

"People who like this sort of thing will find this the sort of thing they like."
Abraham Lincoln

Contents

Abstract i

Acknowledgements iii

1 Introduction 1
- 1.1 Problem discussion . . . 1
- 1.2 Thesis . . . 2
- 1.3 Structure of this Document . . . 2

2 Reflection and Open Implementations 5
- 2.1 Reflection . . . 5
  - 2.1.1 Reflection in object-oriented languages . . . 7
  - 2.1.2 Partial Reflection . . . 9
- 2.2 Open Implementations . . . 10
- 2.3 Summary . . . 12

3 Unanticipated Partial Behavioral Reflection 13
- 3.1 Introduction . . . 13
- 3.2 Unanticipated Reflection . . . 14
  - 3.2.1 Example: Profiling a Webserver . . . 14
  - 3.2.2 Requirements to Use Unanticipated Reflection . . . 15
- 3.3 Analysis of Existing Reflective Solutions . . . 19
- 3.4 Partial Behavioral Reflection . . . 23
  - 3.4.1 Spatial and Temporal Selection . . . 24
- 3.5 Supporting Unanticipated Reflection in a Dynamic System . . . 31
- 3.6 Summary . . . 32

4 Geppetto: Design 35
- 4.1 Fundamental Design Aspects . . . 35
- 4.2 Selection Possibilities . . . 37
  - 4.2.1 Spatial Selection . . . 38
  - 4.2.2 Operation selection . . . 39
  - 4.2.3 Temporal Selection . . . 39
  - 4.2.4 Example: Array Enhancement . . . 40

4.3 Unanticipated Usage 42
4.4 Summary 44

5 Geppetto: Examples 45
5.1 Illustration 1: Profiling 45
5.2 Illustration 2: Code Coverage 49
5.3 Illustration 3: Proceedable metaobject 51
5.4 Summary 54

6 Geppetto: Implementation 55
6.1 Bytecode Transformation to Insert Hooks 55
6.1.1 ByteSurgeon 55
6.2 Geppetto-Core Package 57
6.3 Geppetto-Hookset Package 59
6.3.1 Class GPHookset 60
6.3.2 Hook installer 64
6.4 Geppetto-Link Package 68
6.4.1 Class GPLink 68
6.4.2 Important Methods of GPLink 75
6.5 Geppetto-Operation Package 76
6.5.1 Class-side of Operation Classes 77
6.5.2 Instance-side of Operation Classes 79
6.6 Geppetto-MOP Package 79
6.6.1 Class GPCallDescriptor 80
6.6.2 Class GPParameter 84
6.6.3 Passing Mode Classes 85
6.7 Geppetto-Library Package 86
6.7.1 Usage 86
6.8 Other Packages 89
6.9 Summary 89

7 Benchmarks 91
7.1 Installation performance 91
7.2 Execution performance 94
7.3 Comparison with MetaclassTalk 96
7.4 Summary 99

8 Conclusions 101
8.1 Contributions 101
8.2 Perspectives 103

Chapter 1

Introduction

Reflection is an important tool to extend and modify the semantics or runtime of applications. However, lot of approaches to support reflection are based on up-front fully reflective or load-time reflection mechanisms. Using these approaches, it is not possible to apply reflective techniques on running systems. Because many applications and systems exist that cannot be halted and stopped but have to be always on and running, such as web applications, real-time systems or mobile systems, the ability to apply reflective features at runtime is a crucial and important property. Our solution to achieve this possibility is unanticipated reflection. With unanticipated reflection, we can design the MOP required for the problem we want to solve, introduce it in the language, even in systems where just the binary is available but not the source code, activate the reflective mechanisms and possibly remove them once they are not necessary anymore at runtime, without halting the system or the application in which we want to apply and use the reflective mechanisms.

1.1 Problem discussion

We analyzed several existing proposals for such reflective systems (*e.g.,* Reflex [TANT 03], Iguana/J [REDM 02], MetaclassTalk [BOUR 00]), but they have some shortcomings and do not provide the necessary facilities to achieve unanticipated reflection and to dynamically adapt binary components without requiring the source code. Either these proposals are not efficient and expressive enough or they do not allow us to introduce reflection unanticipated into a running system in a disciplined way. Some approaches perform better and provide indeed unanticipated reflection, but rely on an adapted and dedicated virtual machine, which is not a solution for us either, because we do not want to sacrifice portability. Instead our approach has to run on a standard virtual machine.

These shortcomings of existing solutions tempt us to seek for a better approach to unanticipated reflection. We did our work in Squeak/Smalltalk because this dynamic language fulfills many requirements needed to implement our solution for

unanticipated reflection, such as a powerful environment to access dynamically applications or the system itself. Nonetheless we believe that our approach to unanticipated reflection is also applicable to other dynamic languages such as Python, Ruby or Self, as long as these languages meet some requirements which we cover in detail later on.

1.2 Thesis

In this thesis we propose unanticipated reflection, an approach allowing us to dynamically adapt applications on the fly without halting them to introduce adaptations and also without having prepared them previously. To be able to introduce MOPs into running applications we only need to have access to the bytecode but do not require their source code. We also study the exact requirements we have to use reflection dynamically and unanticipated and analyze existing solutions following a similar goal but not providing exactly what we want.

The cornerstone of this thesis is a framework for Squeak/Smalltalk called GEPPETTO which exactly supports unanticipated reflection and thus allows us to dynamically build and introduce new MOPs into applications in an unanticipated way. We present our work on this framework, explain its design and open implementation with many examples. We report also on its efficiency by providing several benchmarks demonstrating the performance to install and run adaptations of applications or the Squeak system itself.

The thesis concludes with a discussion of unanticipated reflection and its implementation in Squeak and by denoting perspectives for future work.

1.3 Structure of this Document

In chapter 2 we present the basic concepts of reflection and give a short introduction into the area of open implementations as well as partial reflection.

The next chapter 3 is devoted to the presentation of our approach to unanticipated reflection. We discuss some adaptations and extensions of this approach used to optimize and improve its runtime performance and expressiveness. We also cover in detail the requirements a programming language has to fulfill in order to be able to serve as a host language for unanticipated reflection.

In chapter 4 we present GEPPETTO, an implementation of a reflective framework supporting unanticipated reflection in Squeak/Smalltalk. We explain the design of this framework in detail as well as in chapter 6 its implementation along with several examples and illustrations. In the following chapter 5 we discuss how to install GEPPETTO in a Squeak image and illustrate some advanced examples where GEPPETTO proves its power and expressiveness. Next we show some benchmarks in chapter 7 to illustrate the efficiency of GEPPETTO.

The last chapter 8 is a summary of the presented thesis, followed by a section about related work done in the area of reflection, and finally we talk about

perspectives for future studies.

Chapter 2

Reflection and Open Implementations

This chapter is dedicated to present the concepts behind reflection, metaprogramming and open implementations.

The first section of this chapter discusses reflection in general and reflection in the context of object-oriented languages in particular. We also present some metaprogramming techniques in this section. Afterwards we give a short introduction into the art of open implementations in section 2.2. The last section 2.3 concludes this chapter.

2.1 Reflection

Since the studies of John von Neumann who first brought up the idea to store program instructions in the same memory as data of this program, representing programs as data and hence having the possibility to manipulate a program by another program is an active research topic. Already in 1958 von Neumann was interested in studying the manipulation of programs represented as data by other programs.

Even in theoretical computer science we can find the idea of processing programs by other programs, for instance in the Church lambda calculus where programs and data are represented by higher order functions. Another example is the universal Turing machine which is able to process another Turing machine.

Computational System.

The distinction between a program and a computational system has to be drawn: A program is a textual description, while a computational system is a running program [STEY 94]. Thus a program describes a computational system. A computational system has a representation of its domain, this representation is causally connected with the computational system, which means that the changes in the domain are reflected in the computational system, and vice-versa [TANT 04]. These

considerations lead to the term *metasystem*, a computational system whose domain is another computational system. Finally, a reflective system is a metasystem which is causally connected to itself, a metasystem whose domain is itself [MAES 87a].

Computational System.

Reflection is defined as *"the ability of a program to manipulate as data something representing the state of the program during its own execution"* [BOBR 93]. This means a program deals with itself in the same way as it deals with its primary subject matter [SMIT 82].

A reflective system can therefore act and reason about itself. A reflective program thus describes a computational system that accesses its own metasystem.

Other Definitions.

Some more definitions are required to better understand the subject of reflection.

Reification is the process *"by which the state of a program is passed to the program itself in a form that the program can manipulate it"* [WAND 88]. Reified data is then this "part of the state" that is passed to the program during the reification phase. Often all the reified data together is called a reification. A programming language must support two requirements to be able to reify information and thus to be reflective, as stated in [SMIT 84]: First, the language needs *"an account of itself embedded within it"*, which means that the representation of a language must be accessible from within itself. Second, this self-representation must be *causally connected* to the reflective system [MAES 87a].

It is actually not possible for any programming language to support full reflection, which would mean that a program can observe or modify everything of itself [WAND 88]. But there is a clear distinction between true reflective languages and languages that provide just a few reflective mechanisms, such as Java which merely offers introspective facilities.

Reflective Mechanisms.

Several reflective mechanisms are identified in the reflection community. Steyaert defines reflective mechanisms as *"language facilities, offered by the programming language, that allow programs to access the metasystem with which they are executed"* [STEY 94].

We can distinguish the different reflective mechanisms by what we can access with reification, what entities or operations of a programming language we can reify.

Structural reflection is the ability of a program to access a representation of its structure, as it is defined in the programming language.

Behavioral reflection is the ability of a program to access a dynamic representation of itself, that is to say, of the operation execution of the program as it is defined by the programming language implementation [TANT 04].

In an object oriented language, structural reflection allows us to access classes and their members, whereas message sends, temporay variable access and other base level operations are accessible with behavioral reflection, because the last mechanisms define the behavior, whereas the former represents the structure of a program.

Another important distinction of the reflective mechanisms is what we can do with the reified data, only reading or also modifying this data:

Introspection is the ability of a program to simply reason about reifications of otherwise implicit aspects of itself or of the programming language implementation.

Intercession is the ability of a program to actually act upon reifications of otherwise implicit aspects of itself or of the programming language implementation [BOBR 93].

Simply said, with introspection we can just read and access reifications, whereas intercession allows us to also modify these reifications.

These two kinds of dimensions of reflective mechanisms are orthogonal and can be connected together: An inspector in an Integrated Development Environment (IDE) just needs behavioral introspection if it only provides a view to the objects, and also behavioral intercession if it even offers the ability to change the objects, as opposed to a class browser which requires only that the programming language in question supports structural introspection.

2.1.1 Reflection in object-oriented languages

Reflection is not only used in object-oriented languages, but most research was done in the object-oriented world, because the concepts of reflection and object-oriented programming fit well [KICZ 91]. In this subsection we cover the relation between reflection and object orientation and talk about metaobject protocols, the glue between the base level and the metalevel of a program, in the context of OOP.

The main property of object-oriented programming is the fact that data and procedures are packed in the same entity, called an object. Different objects can communicate to each other by sending messages. Further properties of OOP are abstraction and encapsulation, both together properly applied yield a lose coupling between interdependent objects. Reflection does fit well in the concepts of object orientation, because an object could not only perform a computation, but could also have knowledge about *how* to fulfill this computation [MAES 87a].

Furthermore, the abstraction and encapsulation properties are important requirements to be able to separate the meta computations from the base computations and to engineer the metalevel independently from the base level, which is probably the most important requirement for using reflection as a technique to adapt software systems. Object orientation allow the base objects and the so-called metaobjects, that form the metalevel and perform meta computations, to communicate to each other over a well-defined protocol, the metaobject protocol

(MOP) [KICZ 91, FERB 89].

Metaobject Protocol.

A metaobject protocol is a well-defined interface used for the communication between the base level and the meta level, similar as a normal object can communicate with other objects through their public interface.

Kiczales et al. defined *metaobject protocol* in the context of CLOS (object-oriented LISP) as follows:

"First, the basic elements of the programming language - classes, methods and generic functions - are made accessible as objects. Because these objects represent fragments of a program, they are given the special name of metaobjects*. Second, individual decisions about the behavior of the language are encoded in a protocol operating on these metaobjects - a* metaobject protocol*. Third, for each kind of metaobjects, a default class is created, which lays down the behavior of the default language in the form of methods in the protocol."* [KICZ 91]

Several types of metaobject protocols are identified in the literature: explicit or implicit MOPs and inter-metaobject protocols [ZIMM 96]. We use explicit metaobject protocols for the communication of a base level object with the metalevel, when the base object has an explicit knowledge that a metaobject does change its behavior. When using implicit metobject protocols, the base level object is not aware of the metalevel, the call to the metalevel does occur transparently. Finally, intra-metaobject protocols are used in cases where a metaobject has to communicate with other metaobjects.

When referring to metaobject protocols (MOPs) later on we mean always explicit metaobject protocols, unless stated otherwise.

Examples of Reflective Object-Oriented Languages

Several object-oriented programming languages supporting reflection are available. Some of these just have a very limited support of reflective capabilities, often added later to the languages in form of an extension, as in Java. We call a programming language to be reflective *if it recognizes reflection as a fundamental programming concept and thus provide tools for handling reflective computation explicitly* [MAES 87b]. Object-oriented languages obeying to this definition include ObjVLisp, CLOS and of course Smalltalk.

Although Smalltalk is not fully reflective because it does not reify messages and message lookup due to efficiency reasons, it supports many reflective mechanisms and is one of the originators of reflection in object-oriented languages, so we can still call it a reflective language. After all, it supports the most important reflective mechanisms such as structural introspection and structural intercession of classes and, although limited, behavioral introspection and intercession of message sends or the stack frame. Many prominent and wide-spread languages such as C++ and Java cannot pretend to be reflective. C++ does not have any reflective features included, Java was successively extended to support some reflective mechanisms

such as structural introspection, but its support for behavioral reflection is still very limited.

To come back to Smalltalk we shortly describe the reflective model implemented in this language. In Smalltalk-80 structural reflection is achieved with the concept of metaclasses. Every class in the Smalltalk image has an automatically generated metaclass associated. The single instance of a metaclass is exactly the normal class, so the formula that in pure class-based language everything is an object is hold also for classes: Every class is an object, the unique instance of its metaclass. Such a metaclass describes the structure as well as the behavior of its sole instance, the class. This metaclass model is used to form the static structural part of reflection, although it offers also some behavioral reflective aspects.

However, the support of Smalltalk-80 for behavioral reflection is limited in several ways. The unique control structure in Smalltalk is message sending [RIVA 96]. But a message send is only reified when an error occurs. Normally, message sending is not reified due to pragmatic reasons of efficiency [RIVA 96]. The same is true for instance or temporary variable accesses, both are never reified.

Other shortcomings that make it hard to use behavioral reflection to adapt dynamically applications in a standard Smalltalk system are the limited capabilities offered by the objects representing methods, which are instances of class CompiledMethod. This class has a very fixed and hard-wired design and is thus not extensible. The current implementation does not allow us to modify the methods, we cannot even query instances of CompiledMethod for important information about a method, such as its class.

Although we can access and modify the compiler easily, as proven in [RIVA 96], this is not a solution either, because modifying the compiler would only allow us to adapt application's behavior at compile-time, but not dynamically. Smalltalk gives also access to the stack frame and the method context (with class MethodContext, accessible with the pseudo-variable *thisContext*). Using these mechanisms we have some means for behavioral introspection, but adapting the behavior in a disciplined, organized and unanticipated way is not feasible at runtime.

2.1.2 Partial Reflection

Because reflection is very costly, it is necessary to find a solution for this efficiency problem to be able to use reflective features in real world situations. If we reified every message send occurring in a Smalltalk-80 system we would get a performance delay of around factor fifty or even more, as some simple benchmarks demonstrate. This is a clear indication that a full reflection is not feasible and that optimizations are required.

In 1990, Ibrahim [IBRA 91] motivated the idea of partial reflection as a possible solution. Instead of applying full reification where "everything", every entity and every operation in a programming language is reified, partial reflection tries to just reify "some" operations. What we need are expressive means to select in a very fine-grained manner which entities and operations we want to reify. We call

this possibility *selective reification*. The programmer uses for example metaobject protocols or annotations to programming text to explicitly select the operations he want to have reified. Other proposals offer configuration files or even kind of scripting languages to specify the reifications and the links between base level and metalevel, *e.g.,* [REDM 02].

By selecting in a very fined-grained manner where refications are required in our application, we drastically limit the price to pay for using reflection, compared to systems where everything would be reified. These selection mechanisms are therefore an effective solution for the efficiency problem raised by the usage of reflection.

Several levels of selective granulation are available: Some proposals just offer the possibility to select reflective classes [OLIV 99], some can also select members (instance variables, methods, etc.) of these classes, others also offer support to specify exactly which operations should be reified, *e.g.,* instance variable access, message sending or message receiving. Advanced selective possibilities (*e.g.,* available in Reflex [TANT 03]) allows the programmer to even select particular operation occurrences, *e.g.,* just invocations of method #foo. Even specific objects are selectable, for instance we only reify instance variable accesses in object *a* of class A, but not in instance *b* of the same class.

2.2 Open Implementations

In this section we describe shortly what the term *open implementation* means and why it is important to provide open implementations in the context of reflection.

Definition.

Rao [RAO 91] defined the concept of open implementation as follows:

"A system with an open implementation provides (at least) two linked interfaces to its clients: a base-level interface to the functionality of the system similar to the interface of other such systems, and a metalevel interface that reveals some aspects of how the base-level interface is implemented."

The metalevel interface plays the important role here: Providing such an interface basically means that the implementation of the system as well as its interface is revealed and that we are are able to adapt or extend this interface. This definition is very close to the definition of reflection itself: With reflection we have, generally spoken, the ability to view and modify a system in itself. An open implementation basically provides means and facilities to modify and adapt its own implementation in itself, although these possibilities are of course limited, because they have to be well defined and elaborated. Therefore, it is not possible to change everything of an open system, because the means used to define these open functionality are not changeable by this system for which they define the open functionality. This property of open implementations is similar to the fact that systems providing full reflection are not possible, only systems with "some" reflective facilities.

Black-box and White-Box Approach.

The definition of open implementation may sound strange at first glance, because exposing the implementation is controversial to the black-box abstraction proposed by traditional software engineering principles [TANT 04]. But because a complex system implemented in conventional closed-world manner tends to fix many properties, it is not possible to implement such a closed system in a way that all the different needs of a large-scale user group are met. Any high level system requires fixing a number of tradeoffs, and the higher the complexity of the systems is, the more tradeoffs have to be fixed, as mentioned in [KICZ 92].

But the opposite strategy to the black-box approach, called white-box abstraction, is not a solution either, because exposing the entire implementation of a systems with all its details would probably overstrain the users, and it will therefore cost them too much time to achieve their tasks with such a system. This is similar to the framework concept: Most frameworks do hide their internal implementation to their users due to a high inner complexity, so white-box abstraction is not applied. Instead users communicate through well-defined interfaces when working with a framework, such as Seaside [DUCA 04], a sophisticated framework for developing web applications.

Opening an Implementation.

The correct approach to open implementations and open systems is therefore to open just some well-defined parts of the implementation in a disciplined way, not to give users access to the whole internal definition in detail, but to let users just control and adapt implementation strategies [RAO 91, KICZ 92]. Some aspects of the implementation are opened and accessible, others rest implicit.

Object-oriented programming is the predestinated paradigm to develop such open implementations, because by utilizing object-oriented development techniques we can easily provide a well-defined interface through which we can influence and adapt the implementation [TANT 04]. This way, it is possible to give an implementation of a system that meets the needs of a large group of users, because thanks to its metalevel interface the system can still support very special functionality not required by most users most of the time. The few users that need this special functionality can simply implement it themselves using this metalevel interface. As we will see later on this is of particular interest when implementing MOPs: Most of the time we need just a simple standard MOP, but in very rare cases it could come in handy to be able to define a special MOP, for instance to not waste performance by reifying to much information or to engineer a complex metalevel.

Issues.

Designing such a powerful metalevel interface is not as simple as providing just a closed implementation or a predefined metaobject protocol (MOP). We find several information about the issues of designing metalevel interfaces and identified interfaces styles, for instance in [TANT 04], such as the locality issue or the

tradeoffs between generality and granularity. Here we just give some basic information about this task. When designing an open system we have a whole range of implementations, namely all these implementations users should be able to specify [TANT 04]. It is very difficult to anticipate all the various implementations users might be interested in, therefore the only solution is to apply an iterative refinement of the implementation design, with respect to the feedback users give.

For more information about the different styles to design open implementation interfaces we refer to [TANT 04] or [KICZ 97].

2.3 Summary

In this chapter we studied the different concepts of reflection, gave some vocabulary and presented the main problems and issues the raise from the use of reflection in application, such as the efficiency problem. We also shortly analyzed some object-oriented programming languages such as Java and in particular Smalltalk to learn and understand how they support reflection and how they solved the mentioned issues, *e.g.,* the efficiency problem.

The next section was dedicated to partial reflection, which address the efficiency problem by minimizing costly reifications occurring in a system to provide reflective facilities. Precise means to exactly select where an application requires reifications are one way to provide an efficient and effective approach to reflection.

Finally, we also discussed open implementations and open systems, and how they are related to reflection. Opening up the implementation of a framework or a system offering reflection is a powerful way to let the users of reflection freely define their specific needs and to let them implement and design customizable MOPs.

In the next chapter we introduce unanticipated reflection, an enhancement of traditional reflection permitting to adapt dynamically applications, without having them initially prepared for the use of reflective capabilities.

Chapter 3

Unanticipated Partial Behavioral Reflection

In this section we present unanticipated partial behavioral reflection. We begin with an introduction to state our motivation to develop and use unanticipated reflection. Next we analyze existing approaches to behavioral reflection to see how well they support unanticipated as well as dynamic reflection. We learn that existing solutions have many shortcomings and do not offer all the facilities required to support a clean, fast and expressive approach to unanticipated reflection. This motivates our work to implement our own solution which we present thereafter. We also discuss several optimization strategies to be able to provide a fast and useful implementation of unanticipated reflection. Finally, we conclude with a discussion of the presented approach to unanticipated reflection.

3.1 Introduction

Many applications could profit from dynamic adaptation directly in their execution context. Such applications are for instance server applications that cannot be halted, such as web application running on a web server. Very few proposals offer the ability to insert the required reflective mechanisms unanticipated at runtime, because in most programming languages this goal is not achievable without implementing an adapted interpreter for this language or a dedicated runtime environment to insert the reflective facilities at runtime into an application. In the context of Java for example Iguana/J [REDM 02] supports insertion of reflection at runtime by providing an adapted interpreter and thus sacrificing portability. Redmond and Cahill call this form of behavioral reflection *unanticipated*, because it allows us to dynamically apply reflective behavior to running applications or the system itself without adding or modifying any application code before the start up and without halting the system. For debugging or experimenting with applications it is very useful and convenient to be able to use reflection unanticipated, even for applications that were not prepared for the use of reflection before their start up.

We first present unanticipated reflection, its possibilities and mechanisms. We identify several demands an approach to unanticipated reflection has to achieve to be useful in real-world scenarios. Next, we propose important requirements a programming language has to support to elaborate a reflective extension providing unanticipated reflection for this language.

We show the usage and usefulness of unanticipated reflection in a concrete, realistic example which proves the importance of the stated demands for dynamic and unanticipated reflection. We also study several existing proposals supporting reflection and learn that they have different shortcomings, which motivates us to work on our own proposal for unanticipated reflection, overcoming these manifold shortcomings of existing solutions.

We finally discover some advanced facilities a smart implementation of unanticipated reflection can provide to offer more convenience to user. For example, reflective definitions should also affect future programming entities (*e.g.,* new methods, new classes) in a dynamic system.

3.2 Unanticipated Reflection

Unanticipated reflection is the ability to install and remove reflective functionality dynamically even in cases where an application was not initially developed to use reflective capabilities. We are not forced to stop an application to insert or remove reflective behavior and we do also not have to prepare the application before we start them to be able to add and use any reflection in this application later on. Instead we are able to suddenly decide to design and introduce a new metaobject protocol (MOP) into a running application or even the whole system. We are also able to completely remove a previously installed MOP entirely from applications or from the system. This means that unanticipated reflection is not introduced at compile-time nor at load-time, but directly at runtime.

3.2.1 Example: Profiling a Webserver

This ability to use unanticipated reflection is of particular interest for applications running on a server which cannot be halted, such as a web server serving dynamic websites such as a Wiki or a Forum system. We serve these web applications from a Squeak system and have implemented them using the Seaside framework [DUCA 04]. Unfortunately, our Wiki system suffers somewhere from a performance bottleneck, our users complain that the website reacts slowly, and they have to wait a long time for reading or writing content in the Wiki. We wish to find and solve this bottleneck, but we do not have a clear understanding where exactly the problem lies.

We know that some methods in our Wiki are poorly implemented and that we lose a lot of execution time in these methods. Therefore, we want to know which methods consume most of the time. We want to apply a simple profiler to

our Wiki application, but we cannot afford to shutdown our server to install this profiler, because we have a large user base and our Wiki has to be permanently accessible. It is crucial for us to be able to profile our Wiki without needing to stop and restarting it, during the profiling our users should still be able to use the Wiki system as usual.

We have the strict requirement to profile the application in its natural environment and context, because the performance bottleneck does not occur in any other environment, it is bound to the circumstances we encounter in this special setup on the running server. Hence we cannot do the profiling offline on another system or server.

After having gathered all the necessary information we want to remove the profiler entirely from the system, also without being forced to halt the Wiki.

This scenario is a good example for the use of unanticipated reflection. Our Wiki system was not prepared for the use of reflection, but we can nonetheless add such a profiler with behavioral reflection and remove it later when we are finished with the profiling at any point in time.

The profiler itself is quite simple and offers just a basic functionality although it may be extended. We intercept every execution of a method (a so-called message receive) and measure the time elapsed between the execution of the first and the last statement of the method.

This measurement is done on the metalevel in a dedicated metaobject. We use this metaobject to measure the execution time of every method in our Wiki and to gather and store this information in a database. After we run this profiler for a while we simply remove the reflective behavior entirely from our Wiki to bring it back to exactly that state in which it was before we installed this profiling metalevel. We analyze afterwards the gathered data, stored in a database or in a file, offline whenever we want.

We present in detail the technical realization of this profiling example in chapter 5.

3.2.2 Requirements to Use Unanticipated Reflection

We identified several demands our approach to unanticipated reflection has to meet to be useful and applicable onto real-world and complex applications. Based on these demands we anticipate the different requirements a programming language has to fulfill in order to be able to build a framework supporting unanticipated reflection on top of this language.

We first explain these demands and describe second the specific requirements.

Requirements for a Proposal To Unanticipated Reflection

As we learned from the previous example we have to be able to dynamically change existing or add new behavior to running applications. We are mainly interested in

intercepting message sending and message receiving, but we also have to be able to intercept *e.g.,* instance variable access or temporary variable access. These are the base level operations for which we want to add dynamically behavior defined on the metalevel, either before or after, or even instead of the base level operation. For instance, we replace every message send to the method #foo in our application with functionality defined in a metaobject.

Being capable to intercept these four operations (*i.e.,* message sending, message receiving, instance variable access, temporary variable access) is just the baseline our solution has to support. Outgoing from this baseline we make several further demands on our approach to unanticipated reflection to be useful and applicable in real-world situations:

- We must be able to install the reflective behavior at runtime without having prepared the applications or the system before
- We must be able to completely remove previously installed reflective behavior
- To install and remove reflective behavior we do not rely on the source code, but just on the bytecode of an application
- The reflective functionality has to be as fast as possible to be able to apply it also in time-critical applications
- The MOPs we utilize have to be expressive and customizable
- We should be able to adapt every part of the system, even fundamental system classes
- A concrete implementation of our approach must be portable and therefore not rely on an adapted interpreter or virtual machine. Instead we just use a standard dynamically-typed language such as Smalltalk or Ruby.

We summarize our goal by saying that we want to be capable to install (and remove) customizable, expressive and efficient MOPs dynamically into a system without requiring its source code in a portable manner.

Example for the Importance of These Requirements.

After we have finished profiling our running Wiki system and removed the profiler using the mechanisms described in the previous example we can work on fixing the efficiency problem we experience on our Wiki application. Luckily, we found a very expensive and costly method called #expensiveCalculation in class WikiPage. This method is invoked from several places in our Wiki application. We now want to experiment with a different, hopefully more efficient implementation of this method. Because we have only access to the binary of our Wiki we cannot simply change the source of this method and recompile it. Instead, we replace

every invocation of #expensiveCalculation, this is every message send calling #expensiveCalculation, by behavior defined in a metaobject.

In this metaobject we implement a simpler and less expensive implementation of the algorithm in #expensiveCalculation. This optimized algorithm is experimental, we have to analyze if it really provides exactly the same behavior as the old algorithm and how much faster it is. For that we have to be able to experiment in exactly the same environment as the original implementation of the algorithm was running to get reliable results.

Because reifying information is costly we minimize to a great extent the data that is reified at runtime to not bias our results too much with costs for the reification. Instead, we should be able to estimate accurately how much faster the optimized algorithm is compared to old implementation. It is therefore important to be able to specify exactly what information is reified. In our case, for instance, we just need the receiver in the metaobject, we do not care about all the other information about this message send.

We learn several things from this example. First, it is important to be able to introduce our changes just *temporarily* and without having prepared the application for these changes. Because if our new implementation is not really better than the old one, we have to remove it again. Second, experimenting with adapted behavior has to be done in exactly the same environment in which the original behavior was running to get reliable and trustworthy results. Third, if the applied changes are introduced to provide optimized functionality, the mechanisms to introduce these changes should interfere as less as possible with the performance. This means that unanticipated reflection has to be as fast as possible to be useful for experimenting with temporary changes and optimizations. A way to achieve efficient reflection are expressive means to precisely specify the MOPs, *e.g.,* what has to be reified and where.

This example again motivates the mentioned demands on our approach to unanticipated reflection to be useful. Outgoing from these demands we formulate the following requirements a programming language has to meet.

Requirements on a Programming Language

Accessing the Bytecode.

We rely on bytecode transformation techniques to dynamically add reflective functionality. Because the source code of an application is often not available we only have access to the binary of this application, to its bytecode. We therefore have to introduce the desired reflective behavior into this bytecode by inserting so-called hooks, small pieces of bytecode which are responsible for reifying at runtime information we require on the metalevel to fulfill our reflective actions.

By using bytecode manipulation to provide reflection we guarantee that we indeed not require the source code of an application to be available and that our approach to reflection is portable. Bytecode manipulation is applicable using a

standard virtual machine, we do not have to adapt the interpreter of a language in any way or to even extend the language itself to support behavior reflection.

Hence one requirement to be able to support our approach to unanticipated reflection is that the "host" language provides a means to access and manipulate its bytecode. Such facilities are often not natively implemented in a language but are provided by special tools and frameworks which allow us to manipulate bytecode on a higher and more convenient level. Such tools are Javassist [CHIB 03] for the Java language or BYTESURGEON for Squeak/Smalltalk [DENK 05]. For our reflective framework, GEPPETTO, we use BYTESURGEON which we describe in chapter 4 in detail.

Accessing Running Applications.

Another important requirement a language needs to fulfill to host our approach for unanticipated reflection is a means to access running applications or the system itself. To be indeed able to install dynamically hooks into the bytecode of an application we must have an environment which supports accessing the bytecode of running applications.

Most programming languages, dynamic as well as static languages, are capable to give access to the binary of running applications. Squeak/Smalltalk for example has the necessary mechanisms already included in the standard Integrated Development Environment (IDE), so we are able to design MOPs in this IDE and to install these MOPs even into running applications without any problems. Other languages do not offer such a powerful environment out of the box, but it is nonetheless possible to develop such environments also for these languages. Eclipse for Java is an example of such an environment which also supports accessing running applications, although with limited possibilities.

Reification of Structural Entities.

A last requirement is a certain degree of structural reflection a language has to offer, either directly provided by the language itself or at least by an additional tool such as by a bytecode reading and modification tool. For instance, being able to access classes as reified objects is crucial to get knowledge about the methods, subclasses or the superclass a given class has. We require this knowledge about classes in order to be able to install hooks for specific classes and methods.

Not only classes should be reified, but also methods. At least we must be able to access the bytecode of a method, because all the changes we dynamically add to an application are applied into the bytecode of methods. Therefore, we rely on the ability to access a reified representations of methods. In Smalltalk for example we can simply access a method with this code: `AClass>>#aMethod`. Instrumenting the bytecode by accessing methods as normal objects is then straightforward, because a tool supporting bytecode manipulation can access these method objects and thereby their bytecode or intermediate representation.

Conclusion.

We believe to be able to introduce and implement our approach to unanticipated reflection in any dynamically-typed programming language that meets the herein presented requirements. We prove this claim at least for Squeak/Smalltalk in the following chapters, supporting other languages such as Ruby with similar constraints and conditions as Smalltalk is future work.

In the next section we analyze already existing solutions for reflective systems. We explain what the shortcomings of these solutions are and what facilities we are missing to achieve our goal of providing a powerful proposal for unanticipated reflection.

3.3 Analysis of Existing Reflective Solutions

We study in the following several different solutions to provide a reflective system: Smalltalk-80 [GOLD 89, RIVA 96, DUCA 99], MetaclassTalk [BOUR 00], Reflex [TANT 03] and Iguana/J [REDM 02]. The former two are systems based on Smalltalk, the latter two are based on Java.

We now describe these different approaches to reflective systems according to important criteria such as if they support unanticipated reflection, how expressive and efficient they are, what techniques they applied to provide their reflective capabilities, if these techniques are freely portable, or if they require the source code of the applications.

Smalltalk-80.

In Smalltalk-80 structural reflection is based on the concept of metaclasses. Every class in the Smalltalk image has an automatically generated metaclass associated. The single instance of a metaclass is exactly the class, so the formula that in pure class-based language everything is an object holds also for classes: Every class is an object, the unique instance of its metaclass. Such a metaclass describes the structure as well as the behavior of its sole instance, the class. This metaclass model is used to form the static structural part of reflection, although it offers also some behavioral reflective aspects.

Structural reflection, introspection as well as intercession, is very well supported by every Smalltalk-80 dialect. However, behavioral reflective support is quite limited in many ways. Message sending is the most important metaphor to define behavior in Smalltalk: If we invoke a method of a class we send a message denoting the selector of the method (*i.e.,* its name) and the arguments we want to pass to a receiver (*i.e.,* an object). This concept of message sending is uniformly applied in the whole Smalltalk language, it is the unique control structure of Smalltalk [RIVA 96]. The behavior of an application is basically built up with message sending. But Smalltalk-80 does normally not reify a message send due to efficiency reasons. Only when an error occurs the message send causing the error is reified by creating an instance of class Message which holds the important information about the message sent such as the receiver, the arguments and the se-

lector. Reifying such a message object is costly, benchmarks show that a message send is up to hundred times slower if it is reified. Reifying every message send by default is therefore not affordable.

Because the sole control structure, message sending, is not reified in Smalltalk-80 we cannot dynamically adapt the behavior of applications in a standard Smalltalk system in a disciplined way. Existing techniques, such as message passing control techniques presented in [DUCA 99], are ad-hoc solutions and do not offer enough expressiveness and precision for our concern. To use unanticipated reflection as a means for the dynamic adaptation of applications in Smalltalk we have to extend the reflective facilities of this language. Not only message sending is not properly reified, but also instance or temporary variable access. We have to reify these operations with a dedicated tool to be able to intercept them, using techniques such as manipulating bytecode or annotating the abstract syntax tree.

Adding support for unanticipated introduction of reflective functionality and MOPs into a Smalltalk system is not complex, because Smalltalk provides a powerful runtime environment which allows the programmer to communicate to running applications and to access their bytecode conveniently. This situation is pleasant compared to other languages where we would have to implement such an environment first to gain access to running applications.

Of course this reflective system is portable in any way, because the reflective facilities are directly written in standard Smalltalk-80 code, and thus we can use the standard Smalltalk-80 interpreter.

MetaclassTalk.

Implemented on top of Squeak/Smalltalk MetaclassTalk is a reflective extension of this Smalltalk dialect. It is based on explicit metaclasses and allows us to "transparently control and redefine the method evaluation process" [BOUR 00]. Instance variable access, message sending and message receiving can be intercepted and controlled. When an object is created MetaclassTalk automatically associates a metaobject with this new object. This metaobject controls the behavior of the base level object. Metalevel behavior adapting the behavior of base level objects is always implemented in the explicit metaclasses.

Newer versions of MetaclassTalk do not extend the virtual machine of Squeak anymore to provide these advanced reflective functionality, but extend the compiler and use the concept of Method Wrappers [BRAN 98]. Because of these extensions to the compiler MetaclassTalk requires the source code to introduce its reflective capabilities into the system.

Another issue of MetaclassTalk is performance: Although many optimizations were made by introducing only reifications into the system where these are really required the overhead due the use of reflective behavior is still remarkable. For instance, an intercepted message send is around 50 times slower to execute than a not intercepted one.

The expressiveness of the MetaclassTalk MOP is limited, it is not possible to specify what information has to be reified at runtime, instead we always reify the

same amount of information for the different types of operations. For a message send, for example, we reify the receiver, the sender, all the arguments and the selector name.

Newer versions of MetaclassTalk are fully portable between different Squeak images, because they only require extensions to the compiler.

Reflex.

Reflex is an open reflective system for Java developed by Eric Tanter [TANT 03]. It relies on bytecode transformation to insert hooks, relatively small pieces of bytecode, into the binary of an application. These hooks reify during their execution information about a base level operation and pass this information to the metalevel by invoking dedicated metaobjects. The places in the base level code where these hooks are inserted can be precisely selected. For example, we define that we need reflection in class A for every message send to method #foo of a class B.

Due to technical limitations of the Java platform these hooks have to be inserted at load-time of the application. After an application is completely loaded into the virtual machine, we cannot change it anymore. This means that unanticipated reflection is impossible to achieve using Reflex and the standard Java virtual machine.

Reflex features a great expressiveness to specify precisely the reflective needs. We are able to select exactly which operation occurrences (*e.g.,* message sending to object b, invoking method #bar) we want to reify and in which entity (*i.e.,* class or object) this reification should occur. Furthermore, the information to be reified is also precisely selectable. For example, we just want to reify the first argument and the receiver of a message send and are not interested in all the other reifiable data. These selection possibilities are of great advantage to limit costly reifications of information. Thus, the reflective functionality provided by Reflex is quite efficient because unnecessary reifications and shifts to the metalevel are avoided as much as possible. Instead we just reify what we really require.

Because Reflex uses bytecode transformation to introduce hooks adapting the behavior of applications, it runs on a standard Java virtual machine and is thus fully portable.

Iguana/J.

Iguana/J [REDM 02] is also a reflective framework for Java. It shares several similarities with Reflex such as expressive means to specify the desired reflective behavior. It does not only rely on bytecode transformation techniques to insert this reflective behavior into an application, but requires also an adapted and extended virtual machine which is not compatible with a standard Java virtual machine anymore. Although the required adaptations of the virtual machine are small Iguana/J nonetheless sacrifices portability. In fact, this proposal does not directly change the source code of the Java VM, but it modifies the interpreter by making use of the JIT Compiler Interface.

Iguana/J is the only reflective system we analyze in this section that supports

	Smalltalk-80	MetaclassTalk	Reflex
behavioral reflection	very limited	yes	yes
structural reflection	yes	yes	no, added in newer versions
unanticipated reflection	limited	limited	no
control flow reification	very limited	limited	yes, but limited
expressiveness	poorly, ad hoc	coarse-grained	very good
performance	poorly	improved, but still slow	well
portability	yes	yes	yes
source code required	yes, in most cases	yes	no
technique	standard Smalltalk facilities	adapted VM	bytecode transformation
dynamic language	yes	yes	no

	Iguana/J	GEPPETTO
behavioral reflection	yes	yes
structural reflection	no	no
unanticipated reflection	yes	yes
control flow reification	yes, but limited	limited
expressiveness	okay	very good
performance	well, but not as Reflex	well
portability	no, dedicated VM	yes
source code required	no	no
technique	extended VM and bytecode transformation	bytecode transformation
dynamic language	no	yes

Figure 3.1: Comparing different proposals for reflective systems.

true unanticipated reflection. We are capable of adapting Java applications at runtime without being forced to shut them down and without having to prepare them before their start up by using Iguana/J. Actually, this reflective framework supports both: up-front insertion and definition of reflective behavior at load-time like in Reflex, and unanticipated addition of dynamically defined reflection into running applications. This feature set is powerful and inspired us to work on a similar concept. However, the use of unanticipated reflection with Iguana/J has some drawbacks: One is performance, because we get usually a slowdown of more than factor twenty when we reify information at runtime, which might not be efficient enough for several scenarios such as in real-time systems.

Another drawback is the limited expressiveness to specify where and when reflection is required. Iguana/J does not allow us to freely and precisely select our reflective requirements as for instance Reflex does. The possibility to select specific operation occurrences (*i.e.,* intra-operation selection), for example, is missing in the Iguana approach.

Conclusion.

In Table 3.1 we summarize the different features supported by the herein presented proposals for reflective systems.

To conclude our analysis of these four reflective systems we pinpoint that none of these proposals provide exactly the needed functionality. Either they do not

support expressive means to specify the required reflective behavior and are thus often not efficient enough, or they rely on an adapted or extended virtual machine and interpreter for their language and are hence not portable. Furthermore, only Iguana/J brings a true support for unanticipated reflection, but has several shortcomings such as a certain lack of expressiveness or performance. Others require the source code to be able to introduce all their reflective behavior, which is not a solution for us either.

All these shortcomings of existing proposals motivate us to seek for a better and more complete approach to unanticipated reflection which is portable, expressive, efficient and does not require the source code to be available.

In the next sections we describe the different problems we faced during the development of our proposal for unanticipated reflection and how we solved these problems, such as the efficiency problem or the issue of dynamic changes of applications or the system itself.

3.4 Partial Behavioral Reflection

One issue when working on an approach to unanticipated reflection is surely the efficiency of the resulting implementation. Reification of information at runtime and doing frequently shifts to the metalevel to invoke behavior defined in metaobjects is costly. Preliminary benchmarks show us clearly that it is simply not possible to reify all operations occurring in a program in form of full-fledged objects. If we reified "everything", for instance every message send in an application, we would suffer from a slowdown of at least factor fifty up to factor one hundred or even more during the execution of this application. Such a slowdown is simply not affordable, so we have to find a way to optimize our reflective proposal.

One strategy to optimize the efficiency is to minimize the reification of information as much as possible. If we only reify in situations where we really require to have access to the reified information we greatly limit the costs for using reflection. On the one hand, we minimize the number of operation occurrences we reify, on the other hand, we limit the information that is reified per operation occurrence. For instance, we reify just the first argument and the receiver of a message send, instead of also reifying the selector, the sender, all the arguments, the context and other information about this message send.

Selecting *what* (and *when*) to reify information instead of reifying just everything, is referred to as *partial reflection*, as stated in chapter 2.

We studied several proposals achieving partial reflection (see also section 3.3) and believe that the partial behavioral reflection approach provided by Reflex [TANT 03], based on the model of hooksets, is the most powerful and meets best our specific requirements. We therefore use this model of partial behavioral reflection as the basis for an efficient implementation of unanticipated reflection, because it is very expressive and flexible, efficient, and applicable to achieve unanticipated reflection in dynamically-typed languages, even though it was initially developed for Java,

a statically-typed language. We have to adapt this model to fit our demands and to make it ready for unanticipated reflection, but these adaptations are straightforward and easy to perform in a Smalltalk-80 system.

As opposite to full reflection we just shift to the metalevel in cases where reflective behavior provided by the metalevel is really required. Instead of invoking the metalevel on every executed operation the metaobjects are just called for only certain operations, as illustrated in the "reflectogram" on figure 3.2. We see that many shifts to the metalevel occur using full reflection, but only a few if using partial reflection.

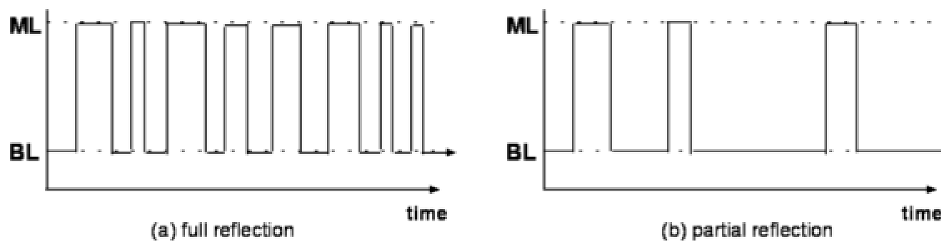

Figure 3.2: "Reflectogram" spotting the different control flow using full and partial reflection.

The key aspect behind the model of partial behavioral reflection are the flexible and expressive means to specify exactly the reflective needs. We now describe these selection possibilities in their spatial and temporal dimension.

3.4.1 Spatial and Temporal Selection

Because reification is a costly process, one way to minimize the costs for reflection is to exactly determine where reification should occur and what should be reified. If we can avoid every useless reification of any piece of information, we gain a lot of performance compared to the situation where we would reify everything. Such a selection implies for instance defining exactly in which method of which class we want to reify a message send to the method #foobar.

We can further select what piece of information we require when reifying such a message send. Possibly we are just interested in the first argument passed to #foobar but not all the other arguments. We also do not need the receiver object of the message send, so we simply specify that we just want to reify this first argument instead of creating and passing an array containing all reifiable information of this message send.

Another dimension in which we want to precisely select the necessary reifications is time: Maybe we only require the reification of an operation during a special phase of our application, *e.g.,* at start time, or only as long as a certain condition is met. Thus, we also lack a means to just activate reification temporarily.

We first introduce a vocabulary of important terms used in the proposal of partial behavioral reflection and explain later on the different selection mechanisms in detail.

Vocabulary: Hook, Hookset and Link.

A *hook* is a piece of code inserted in the base level bytecode responsible for doing the reification and the shift to metalevel. By defining where these hooks are installed we specify where reification occurs. The goal is to select very precisely the places where these hooks have to be installed to avoid any not required, but expensive reification of information.

A *hookset* is a collection of hooks providing the same functionality. We use hooksets to define all the places where reification has to occur to achieve a certain goal. If we for example want to replace every message send occurring in class Bar and invoking a method #foo we define a hookset that selects this class Bar and every therein occurring message send invoking method #foo. Behind the scenes, many hooks have to be installed, one for every operation occurrence matching the defined criteria. But because all these hooks do the same job, we do not identify every single hook, but one hookset containing all the required individual hooks.

A hookset may therefore *"gather execution points scattered in various objects"* [TANT 04], as opposite to most other traditional reflective systems where hooks are just defined on a per class or object basis. The main benefit of the concept of hooksets is a simpler and better assignment of metaobjects to hooksets instead of directly assigning them to single hooks.

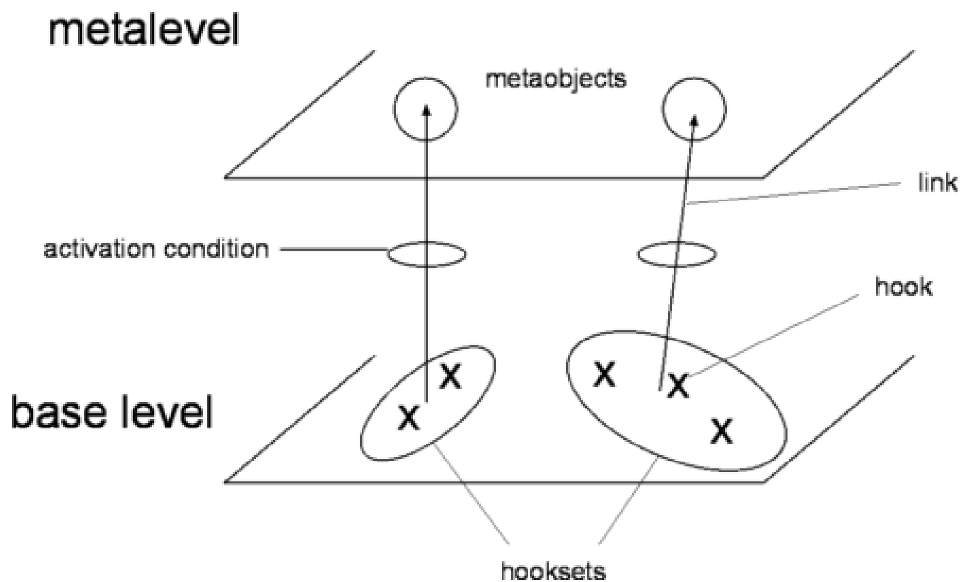

Figure 3.3: Base level and metalevel with hooksets, links and metaobjects

The link is the central entity in this proposal of partial behavioral reflection. Its

responsibility is to causally connect the hooks in the base level with the metaobjects in the metalevel. We associate every hookset with at least one link. The link entity specifies how the shift to metalevel has to take place when a hook is executed, what information is reified in the hook, how this information is passed to the metalevel and which metaobject is invoked. Further responsibility of the link is to maintain the activation condition, which determines when the link is active (see the next section about temporal selection), to define the scope of the metaobject, or to manage how the metaobject controls the base level operation.

We shortly come back to links after covering the two dimensions of selection, spatial and temporal. With these two dimensions of selection, *spatial selection* and *temporal selection*, we avoid many useless level shifts that would occur if using full reflection.

Spatial selection

We use the spatial selection mechanism to specify which operations and which occurrences of these operations are reified in an application. Thus spatial selection determines *what* is reified. We distinguish three different levels of spatial selection:

- Entity selection is used to select the reflective classes and objects. For instance, we can select the whole class A or just one single instance of class B.

- Operation selection specifies which operations are reified for a given entity. Such an operation could be message sending in class A, instance variable access in class B, or every temporary variable access in a whole package of classes.

- Using intra-operation selection we select particular operation occurrences, for instance only message sending occurring in method #bar of class A (caller-side) or only message receiving in method #foo of class B if the sender object is an instance of class A.

In Figure 3.4 we illustrate the spatial selection mechanism: We have an application with three classes A, B and C. We select class B and in this class method #bar and #bla: (entity selection). We decide to reflect message sending (operation selection), but only when the message send invokes a method #factorial (intra-operation selection).

Intra-operation selection, the finest-grained selection level, is of great value to minimize the places in the base level where hooks have to be installed and reification occurs, therefore it is an important property of this model of partial behavioral reflection, offered by very few other proposals.

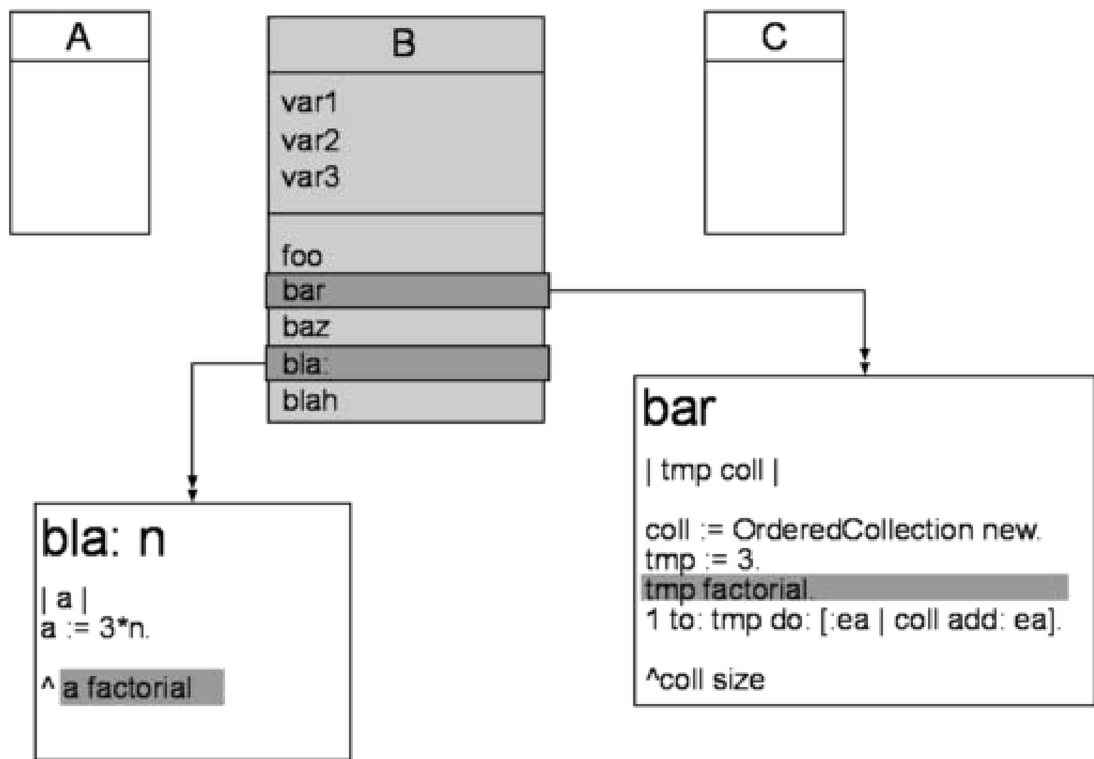

Figure 3.4: Spatial selection in a small application with three classes.

Temporal selection

With temporal selection we specify *when* a link from the base level to the metalevel is active. Only if a link is active, a shift to the metalevel occurs. Thus, temporal selection has a very dynamic dimension because it allows us to change the reflective needs of an object during its lifetime. If a link is deactivated the execution of the associated hook does cost less than for the active link, but of course still more if no hook is installed at all, because some parts of the hook has to be evaluated to determine if the link is active or not.

This selection possibility is not that important anymore in our proposal of unanticipated reflection. Our reflective architecture, GEPPETTO, is capable to remove or reinsert hooks at runtime. Hence we remove links entirely instead of deactivating them, when we do not have a use for those links anymore. This mechanism to dynamically install reflective functionality into a running system and remove it again entirely is an important step to unanticipated reflection.

An Example Illustrating the Selection Mechanisms

A short example clarifies these selection mechanisms. In the base level we have a method #calculate in class Math. This method is complex and consists out of many instructions. We heavily use several collections in it and want to track message sends to collections invoking either the methods #add:, #remove: or #size just to determine how many times these methods are executed in our #calculate method to estimate the time this algorithm spends in collection-related operations. Furthermore, this tracking has to occur only when our application is in a measurement mode, not during its normal execution. This means that we want to activate the reification of these operations just temporarily when this condition is met.

```
Math>>#calculate
  | col1 col2 col3 |
  ...
  0 to: col3 size do: [col1 add: a].
  ...
  col1 do: [:ea | col2 remove: ea].
  ...
```

The first step is to select the operation occurrences we want to reify, this is every invocation of either method #add:, #remove: or #size in Math>>#calculate. This selection is referred to as spatial selection.

To specify this selection we have to define a hookset. This hookset selects first the method Math>>#calculate in which we want to install reflective behavior and thus hooks. Second, we select the operation *message send*. We have to further redefine this selection to just some occurrences of the message send operation,

namely only invocations of either selector #add:, #remove: or #size, referred to as intra-operation selection.

The next code snippet present the accordingly arranged hookset:

```
hookset := Hookset inClass: Math inMethod: #calculate.

hookset operation: MsgSend.

hookset operationSelector: [:send
        | #(#add: #remove: #size) includes: send selector].
```

The second part of the spatial selection is the specification of the information we have to reify. In this example we are only interested in the selector and the receiver of the message send. This specification is done in the link by providing a MOP descriptor, which characterizes the reification as well as the call to the metaobject. This link is further used to define the control which states when the hook is executed, before, after or instead of the base level operation. In our case we take the before control. We also use the link to hold the activation condition in which we specify that the link is only active when our application is in measurement mode.

In the MOP descriptor we specify the metaobject, an instance of class CounterMO, the method we invoke on this metaobject, #countCollectionInvocations:, and the parameters to reify, here the receiver and the selector of the message send. If we do not state otherwise, these parameters are passed in an array to the metaobject method.

Note that the last statement in the following listing denoting the link definition finally install the link. During this installation all the required hooks, as defined in the hookset, are installed into the system transparently.

```
link := Link hookset: hookset.

link control: Control before.

link moCall: (CallDescriptor metaobject: CounterMO new
            selector: #countCollectionInvocations: .
            parameters: {Parameter receiver. Parameter selector}).
link activationCondition: [:object | Application mode =
#measurementMode].
link install.
```

The metaobject itself can be arbitrarily complex and is implemented in normal Smalltalk code. In this example we simply increase a counter if the receiver is indeed an instance of a collection class.

```
CounterMO>>#countCollectionInvocations: anArray
   | receiver selector |

   receiver := anArray first.
   selector := anArray second.

   self assert: (#(#add: #remove: #size) includes: selector).

   (receiver isKindOf: Collection) ifTrue: [self increaseCounter].
```

Because links have a high complexity and are customizable in many ways we cover the functionality offered by the link entity in the next section comprehensively.

More on Links

In our model the link entity has the important function of bridging the gap between the base level and the metalevel. Every installed hookset is associated with a link and this link then defines how the hooks of this hookset call the metaobject(s) and which information they reify and pass to the metalevel. A link is the primary entity to which we communicate during the configuration, the installation and the use of behavior provided by unanticipated reflection. We can use it also to define the properties of the hookset and to identify the installed hooksets.

Links are very important in our model to maximize the reuse of metaobjects and to decouple the work of selecting the places in an application where we need reflection. Thanks to the concept of links we can design and engineer the metalevel completely separated and independently from the base level. Links bridge this gap between base- and metalevel, making it possible to reuse hookset definitions and metaobjects separately.

We describe in the following the different attributes a link knows about. These attributes manage the complex behavior of a link, such as controlling the execution of hooks or characterizing the reification and the call to the metaobjects.

- the *control* defines when the metaobject is given control over an operation occurrence, we can set it either to before, after, before and after, or replace, which means that the metaobject is executed instead of the original operation

- we use *scope* to define for which entity a metaobject is valid. This can be hookset scope if a single metaobject is valid for every hook in a hookset, or class scope if every class involved in a hookset has its own metaobject, or even object scope if every object has a dedicated metaobject.

- the *activation condition* dynamically determines if a link is active. We specify this condition either for the whole hookset, for a single class or even for a

single object (similar to the scope attribute). We use this activation condition to achieve temporal selection.

- the *updatable* attribute specifies if the link is able to dynamically change its associated metaobject or not.

- with a *MOP descriptor* we specify exactly which metaobject we want to call from within a hook and which information has to be reified. Furthermore, we define the passing mode which specifies how this information is passed to the metaobject, either packed in an array or in an object to ease the access to the data, or in plain mode, where every piece of reified information is passed in its own method parameter to the metaobject. A "none" passing mode is also available for cases where we do not reify any data.
  These MOP descriptors provide simple but yet powerful means to select in a fine-grained manner what information has to be reified and how we access this information in the metalevel. This helps to further gain performance, *e.g.,* by not reifying any not required information or by abandoning any creation of arrays or objects to hold all the reified information when simple method parameters are enough. For more information on MOP descriptors see [TANT 05a].

Perspectives

As we mentioned in the introduction this model of partial behavioral reflection was initially motivated in [TANT 03]. We have now presented the key ideas of this model that we reuse to support unanticipated partial behavioral reflection. Eric Tanter et. al. extended their approach to partial behavioral reflection recently to also support structural reflection and to be able to use this as a basis for an aspect-oriented kernel (AOP kernel) [TANT 05a]. A lot of research was also done in the area of link composition. We do not cover these progressive features and evolutions of the model in the following, although these additions might be interesting to study in future research.

3.5 Supporting Unanticipated Reflection in a Dynamic System

Dynamic Changes.

If we have adapted an application dynamically we have probably installed several hooks in different methods of application classes. If we update this application, *e.g.,* extend and recompile one or more methods, the hooks installed in the affected methods disappear. This is often problematic, because the behavior implemented in the metalevel relies on these hooks. It is therefore crucial that installed hooks do not disappear accidentally from a system upon changes such as method recompilations.

Our proposal to unanticipated reflection hence provides mechanisms that observe all methods and classes in which hooks are installed to be able to react on changes to these entities and to reinstall any accidentally removed hooks.

New Entities.

In addition, because we can dynamically extend or update an application, the introduction of new methods, classes or even packages is also possible. If we have for example installed once a link that intercepts every instance variable access in a given class, we expect that this is still true after adding new methods containing new instance variable accesses into this class. This means that hooks reifying instance variable accesses have to be installed automatically in newly defined methods.

If we dynamically add a new subclass of this class for which we reflect every instance variable access, we expect that instance variables defined in the superclass, but used in this new subclass are also reflected. So our definitions of links and hooksets selecting entities and operations we want to reify should also affect future entities, method or classes, that are not yet defined when we install these links for the first time.

Conclusion.

To summarize these two advanced capabilities of unanticipated reflection we say that if hooks are accidentally removed from the system, by recompiling methods or classes, our system has to trap this situation and react accordingly by reinstalling the removed hooks to not get any weird behavior on the metalevel. Furthermore, if definitions of reifications affect newly added entities, such as an extended method or a complete new class, the hooks performing these reifications should automatically be installed also into these new entities to guarantee the correct behavior as initially defined during the configuration of hooksets and links.

Generally, after every change to the system, *e.g.,* recompilations, additions of methods or classes, the installed reflective behavior, mainly achieved by hooks, has to be in a proper state and perform exactly how the user initially defined.

3.6 Summary

We first presented unanticipated reflection, an enhancement of traditional reflection making it possible to dynamically adapt applications and the system itself. We enumerated the several demands we put on a proposal for unanticipated reflection to be useful and powerful, such as the expressiveness, acceptable runtime performance, or portability. Next, we discussed the requirements we believe that a dynamically-typed languages has to meet in order to be able to support unanticipated reflection.

We analyzed existing solutions supporting reflection in general and unanticipated reflection in particular. We learned that none of these proposals yet provide our desired features and functionality which motivates us to work on a better and

more complete approach to unanticipated reflection.

An important step to a powerful proposal was the usage of the model of partial behavioral reflection as stated in [TANT 03] and implemented in Reflex for Java. This model is based on the concept of hooksets and links. With hooksets we basically define where to install reflective functionality in the base level of our application, and with links we causally connect these places to behavior implemented on the metalevel. This model supports very fine-grained and powerful selection mechanisms to specify exactly the required reflective functionality in spatial and temporal dimensions.

We finally discovered that Squeak/Smalltalk is an ideal platform to implement in it a framework, called GEPPETTO, to provide unanticipated partial behavioral reflection, consequently following the proposal herein presented. Even though we did our concrete work mostly in Squeak, we believe nonetheless that the implemented solution is also applicable for other dynamically-typed languages such as Ruby.

The next chapters are dedicated to the presentation of GEPPETTO, its design and implementation. We discuss in detail how we implemented the GEPPETTO framework to support unanticipated reflection in Squeak.

Chapter 4

Geppetto: Design

This chapter presents GEPPETTO, an open reflective framework, implementing unanticipated partial behavioral reflection in Squeak/Smalltalk.

We first describe the important parts of the design of GEPPETTO and show how we implemented the unanticipated reflection in Squeak. We talk also about the advanced possibilities to precisely select where reification should occur we have in Squeak compared to what is available in *e.g.,* Java. Finally, we summarize our work on GEPPETTO in the last section.

The details of the implementation, *e.g.,* how the different classes are structured internally, how they fulfill their tasks and how different packages and classes communicate to each other is the topic of the next chapter 6.

4.1 Fundamental Design Aspects

The two most important entities in GEPPETTO are hookset and link. In this section we show the important parts of the design around these two entities.

On Figure 4.1 we see the relation between hookset and link. Every link is associated with just one hookset, whereas a hookset may have different links, although this is not common. In most cases, we have a 1:1 relation between hooksets and links. As we can see the hookset has not many associated attributes, basically just the association to the link. The link in contrary has several attributes: The control, the scope and an activation condition, to just mention the most important attributes. As we see later on when talking about the metaobject protocol (MOP) a link has also an associated metaobject descriptor.

Hookset and Link.

A hookset simply represents a collection of hooks. We have different means to select the places where we have to install hooks, such as class selectors or class enumerators. By defining a hookset we actually specify these places where we want to install hooks. A hookset manages all the hooks that were introduced into the system for the same purpose, a hookset therefore groups hooks installed at

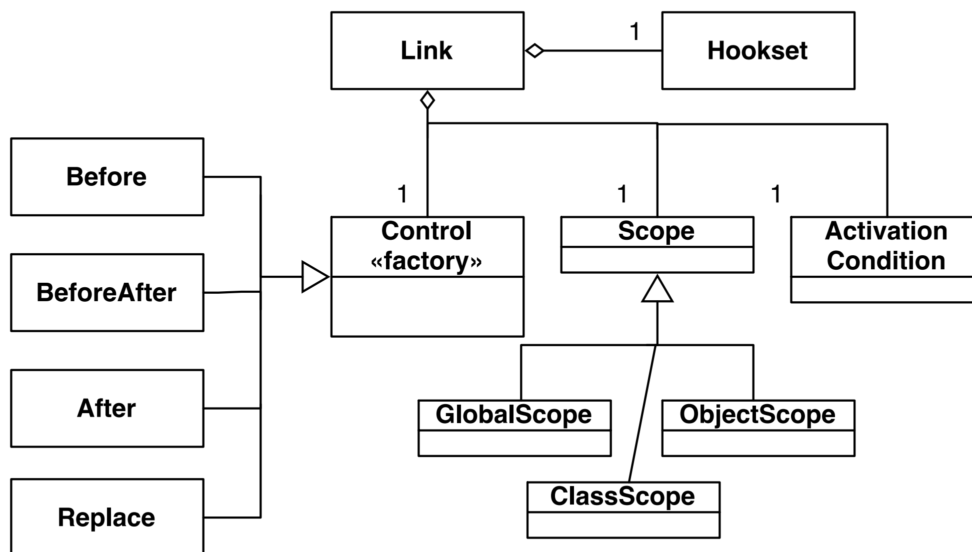

Figure 4.1: Hookset and link

different places, but providing the same behavior. If we want to log every message send in class A, we have to install one hook for every message send occurring in this class. But we define just one hookset representing all the required hooks.

In the following example we select class Object and its method #asString using the appropriate selector mechanisms. We further select the operation *message sending* in this single method if the send invokes a method #printString. For every such message send in Object>>#asString the so-defined hookset specifies the hook. The same hook will be installed for every operation matching the criteria, and all these installed hooks are specified by one single hookset.

```
hookset := GPHookset new.
hookset classSelector: [:class | class = Object].
hookset methodSelector: [:method | method = #asString].
hookset operation: GPMsgSend.
hookset operationSelector: [:send| send isSend and: [send selector=#printString]].
```

By using class enumerators or class selectors we select all the classes in which we want to install hooks. A class in which we have installed at least one hook is called a *reflective class*. We can also select just some methods of a reflective class by defining method enumerators or method selectors, this means that we install hooks only into these specified methods.

An important selection mechanism is the operation selection. An operation is either message sending, message receiving, instance variable access or temporary variable access. We can also define an intra-operation selector which further refines

the selection to just some operation occurrences, *e.g.,* we are just interested in message sends invoking a method #foo.

In the following example we present how we set up a hookset and a link by just using the methods provided by GPLink, instead of defining the hookset directly as in the example above:

```
link := GPLink id: #exampleLink.
link inClass: Object.
link inMethod: #asString.
link operation: GPMsgSend.
link operationSelector: [:send| send isSend and: [send selector=#printString]].
link control: GPControl replace.
link moCall: (GPCallDescriptor
          object: self
          selector: #mo:
          parameters: {GPParameter selector.
                    GPParameter receiver.
                    GPParameter arguments}
          passingMode: GPPassingMode array).
```

With this link we specify that we want to install a hookset affecting the method Object>>#asString. In this method we intercept the message send operation and, as specified with the operation selector, just message sends invoking method #printString. The link object internally creates a hookset with these definitions. The other definitions such as the control attribute or the MOP descriptor which we pass to method #moCall:, are the attributes directly associated with the link. As we stated in section 3.4.1 the link has five important attributes: control, scope, activation condition, updatable and MOP descriptor. We explain these attributes and their use in detail in the implementation chapter in Section 6.

In the next section we present the several possibilities to select the places where we want to use reflection, because these selection possibilities form a crucial part of the model of partial behavioral reflection applied in GEPPETTO.

4.2 Selection Possibilities

An important aspect of partial behavioral reflection are the different possibilities to select *where* and *when* we want to apply reflection in our system. In this chapter we describe how we can specify the two selection dimensions, *spatial selection* and *temporal selection*, in GEPPETTO and how fine-grained these selection mechanisms are.

4.2.1 Spatial Selection

Spatial selection gives an answer to the crucial question *where* we need reflective functionality on our system and is defined with the concept of hooksets. When we configure a hookset the first step is to specify the classes that should become reflective. This can be done using a *class enumerator*, which simply enumerates all the reflective classes, for instance all classes in a given package or all subclasses of a given class or all classes whose name matches a certain criteria. This selection is *extensionally*. An intentional definition of the reflective classes is also available in GEPPETTO, called *class selector*. Such a class selector selects all classes in the system that match a given condition:

```
hookset := GPHookset new.
hookset classSelector: [:class | (class name first: 2) = 'GP'].
```

A class selector is in fact a block expecting one single parameter which is a class. This block will then be evaluated for every class in the system, and all the classes are selected for that the selector evaluates to *true*. Thus, the class selector in the above example selects all classes whose name starts with the letters 'GP'.

The next step is to select all the reflective methods in the previously defined reflective classes. If no methods are selected all methods of each reflective class become reflective. To explicitly select just some methods the same means are available as for classes: Method enumerators and method selectors. The former works analogously to class enumerators: One can simply enumerate all the affected methods, for instance all methods in a given method protocol or all methods those selector starts with a certain string.

Method selectors are similar to class selectors, they are also defined using a block which expects the name of a method, the *selector*, as its single parameter:

```
hookset := GPHookset new.
hookset methodSelector: [:methodName | methodName = 'size'].
```

This block is then evaluated for every method in all the defined reflective classes. Therefore, all methods called #size become reflective in this example.

These two means for selecting classes and methods, enumerators and selectors, can also be combined in one single hookset, in such a case the union of all selected classes resp. methods defined using enumerators or selectors become reflective.

Affecting the future.

One important and interesting feature concerning this spatial selection mechanism is that it will also affect new classes and methods that are not yet defined in the system when the hooksets and links are installed for the first time, as described in 3.5. Because every new class and method that is added to the system at any point

in time will be also checked against all the defined enumerators and selectors for selecting classes and methods in any installed hookset, it is also possible to automatically install hooks in a new class or method as long as this class or method match the conditions defined in any of the installed hooksets.

4.2.2 Operation selection

There are five operations that we can reify in an application: Message sending, message receiving, instance variable access, temporal variable access, and value returning from methods. Except message receiving which is callee-side all operations are caller-side. For all these operations it is important to explicitly declare which occurrences of an operation has to be reified. This is called intra-operation selection. For message receiving it is enough to just select all the methods for which a reflective functionality is required as soon as they are invoked.

For all other operations we can specify intra-operation selection by using an *operation selector* to explicitly select all operations that have to be reified:

```
hookset := GPHookset new.
hookset operation: GPMsgSend.
hookset operationSelector: [:instr | instr selector = 'bar'].
```

In this example we select the operation message send. With the operation selector we can further restrict the selection of the reified operations to only message sends to the method #bar. An operation selector is a block that receives an instruction as its single parameter. Several instructions are defined, such as IRSend, IRReturn, IRTempRead or IRInstVarStore that together basically form a compiled method. When selecting the operation 'message send' the operation selector block is evaluated for every IRSend in every previously selected method. An IRSend does for instance understand the message #selector to find out to which method this instruction does a message send. For the other operations, instance variable access and temporal variable access, we define operation selectors the same way, except that the instructions passed to the block are instances of IRInstVarAccess and IRTempVarAccess, respectively.

Operation selectors do the static part of the intra-operation selection mechanism, the part that only depends on programming text. But intra-operation selection has in fact also a dynamic aspect, for instance to select only operation occurrences in some objects. We can achieve that by using an activation condition in links in the context of temporal selection.

4.2.3 Temporal Selection

Doing temporal selection means defining when a link is active. We can define link those active status is changeable at runtime, and links for which we cannot change the active status as soon as they are installed, they are always active then. For links

that may change their active status at runtime, an activation condition can be set which will be evaluated during the execution of the hook to determine if the given link is active and should be executed.

The constant activation conditions ON and OFF are available, but user-defined conditions can be provided. Such a condition has to be an object understanding the message evaluate: anObject which has to answer a boolean determining if the link is active or not. This method expects the current object as its single parameter, therefore it's possible to execute links only for some objects to achieve intra-operation selection on a per-object basis, but at the price of additional runtime costs.

4.2.4 Example: Array Enhancement

In the following we present a complete example showing how to apply the spatial selection mechanisms explained above. We enhance the standard Array class of Smalltalk in this illustration.

In Squeak/Smalltalk we can use the class Array to represent arrays. But class Array has some shortcomings, for instance, we have to fix the size of the array at creation time and cannot add more elements as soon as we reached the limit. We would also like to add elements without specifying an explicit index, instead we expect that Array internally uses the next higher index. In Smalltalk this functionality is provided by OrderedCollection, but users coming from other languages might expect the same behavior for arrays too.

In this example we create an array containing four elements:

```
| arr |
arr := Array new: 4.
arr at: 1 put: '1'.
arr at: 2 put: '2'.
arr at: 3 put: '3'.
arr at: 4 put: '4'.
```

Note that we had to specify the size of the array, four, already when we created the array instance. To add the four elements we have to explicitly use a numeric index between 1 and 4 for our elements.

But we do not want to limit the size of the array, but want to be able to add as many elements as we wish to an array object. It should be possible to write the following code:

```
| arr |
arr := Array new.
arr add: '1'.
arr add: '2'.
```

```
arr add: '3'.
arr add: '4'.
```

The resulting array structure should be the same as in the previous example, the same string should have the same index from 1 to 4.

We can use GEPPETTO to achieve this functionality by changing the behavior of class Array. Normally, when sending the message #add: to array we get an error, because this operation is not supported for arrayed collections. We now use the message receive operation in GEPPETTO to provide a different implementation for the method #add: for arrays. In fact, we replace the standard #add: method with behavior implemented in a metaobject.

The method #add: is implemented in class ArrayedCollection, the superclass of Array, so we have to install our hook in ArrayedCollection>>#add:. We use the replace control, so if an instance of Array receives the message #add: we execute instead of ArrayedCollection>>#add: a method in our metaobject on the metalevel.

In the following we present the required code for the configuration of the link. We specify the spatial selection using the convenient methods #inClass: and #inMethod: that allow us to select just one reflective class and one method of this class. Internally in the hookset GEPPETTO creates automatically class and method enumerators to hold the reflective class and method.

Further spatial selection includes the choosing of the message receive operation, as specified with GPMsgReceive passed to the method #operation: of the link.

```
| link |
link := GPLink id: #arrayAdd:.
link inClass: ArrayedCollection.
link inMethod: #add:
link operation: GPMsgReceive.
link control: GPControl replace.
link moCall: (GPCallDescriptor object: GPArrayMO new
            selector: #add:for:
            parameters: {GPParameter arg1.
                         GPParameter self}
            passingMode: GPPassingMode plain).
link install.
```

We just use the link object to also define the hookset. With the installation of the link, we also trigger the installation of the hookset, so a hook is placed in ArrayedCollection>>#add: to perform a shift to the metalevel if this method is called. We describe this shift using a call descriptor. The metaobject is an instance of class GPArrayMO which has a method #add:for: implementing the reflective

behavior. We pass the first (and sole) argument passed to the method ArrayedCollection>>#add: and self, representing the array object, to the metaobject using the plain passing mode.

The method GPArrayMO>>#add:for: in the metaobject expects as its first argument the element we try to add to the array object, and as second argument the array object itself. The implementation of GPArrayMO>>#add:for: is simple:

```
add: element for: array
  | coll |
  coll := element asOrderedCollection.
  coll add: element.
  array become: coll asArray.
  ˆelement
```

We convert the obtained array into an ordered collection and add the passed element to this collection. Then we convert the collection back to an array and let this new array become the old array, using the #become: method which basically swaps the object pointers of the two arrays. Finally, we answer the added element.

This simple but interesting example shows us clearly how easy it is to change the behavior of system classes transparently. The change we introduced here is available for all classes in the running system as soon as we have installed our configured link. With just a couple lines of code we extended and adapted the behavior of the Array class without modifying its source code. We are also able to remove that change again by simply deinstalling the link from the system to bring back the original behavior.

4.3 Unanticipated Usage

We are able to use the reflective features provided by GEPPETTO in an unanticipated manner which means that we can define new hooksets and links and apply them to running applications at any point in time.

We do not have to prepare these applications to be able to introduce any reflective behavior. Instead we just access an arbitrary application dynamically and introduce any reflective functionality supported by GEPPETTO, such as intercepting message sends to specific methods. Applications adapted this way do not have to be developed in a language or a system supporting reflection, but can still profit of the reflective features provided by GEPPETTO.

The support for unanticipated reflection is natively implemented in GEPPETTO. For instance, we support the recompiling of methods with installed hooks. These hooks do not get lost after a recompilation, because GEPPETTO traps every recompilation event and triggers the re-installation of any hooks that belong to a recompiled method to guarantee the correct behavior of the defined reflective functionality.

As we mentioned in Section 3.3 Smalltalk provides genuinely the required means to communicate to running applications, such as the possibility to manipulate and transform the bytecode of methods while the application to which they belong is running. Thus GEPPETTO has only to assure that hooksets and links can be inserted into methods at runtime. Necessary steps to achieve this goal is to make sure that different hooks inserted into the same method do not conflict, or that we do not loose any hooks upon a recompilation of a method.

Conflicting Hooks.

GEPPETTO automatically detects if several hook conflict in a method. A conflict occurs if more than one hook affects the same operation occurrence in a method. The strategy applied to resolve such a conflict is very basic but effective: GEPPETTO inserts one single hook that executes every link defined for all the conflicting hooks. Thus we have to reify the union of all the required information of all conflicting hooks.

This is surely not so efficient and also not expressive, because we cannot specify the order in which these links are executed. But it is an easy and working approach to conflict resolution. We plan to come up with a more expressive detecting and resolving mechanism for conflicting hooks in the near future.

Recompilation Issue.

Always when a method is recompiled in a system with GEPPETTO installed we get notified by the system notifier. We then analyze if the recompiled method is affected by any hookset. If this is the case we re-install all the hooks belonging to this method again, because they get lost after the recompilation. It is of course possible that this method was changed in a way that we cannot insert a previously installed hook again, because *e.g.,* the operation occurrence intercepted by this hook disappeared from the method. These hooks are not installed again, but all other hooks do not get lost after a recompilation.

Thanks to this automatical re-installation of hooks upon changes to the system we guarantee the constant existence of all the reflective mechanisms required in the metalevel also when the adapted application itself evolves.

Future Entities.

Because changes to an application are not limited to recompilations of methods, but include also the addition of complete new entities, such as a new method or an entire new class, we apply the definition of our reflective requirements also to these newly added entities.

For example we have installed a MOP intercepting every instance variable access of the variable called foobar in class Baz. We dynamically add the method #newBehavior containing two accesses to instance variable foobar into class Baz. Because we rely on the metalevel that really every access to foobar occurring in class Baz is trapped we also expect that this is still true after changes to the system. Hence we have to also intercept these two newly added accesses to this variable in

the new method #newBehavior.

Affecting also future entities, methods or classes that do not exist when we first define and install a MOP is crucial to be able to assure that the constraints implicitly or explicitly stated at installed time of hooksets and links are still hold when the system evolves. Such a constraint is for instance that every instance variable access in a given class is trapped.

It is a key ability of unanticipated reflection in a dynamically changeable and extendable system that the reflective behavior can also affect future base level behavior.

4.4 Summary

In this chapter we presented how GEPPETTO has implemented the model of unanticipated partial behavioral reflection described in 3. We described the most important design issues of GEPPETTO that are closely related to hooksets and links, the core and fundamental entities in the model of partial behavioral reflection.

We also presented how this design allows us to use behavioral reflection unanticipated to dynamically adapt applications without having to stop them. We discovered that the design for partial behavioral reflection is well capable to be used unanticipated.

In the next chapter we show how to use GEPPETTO to achieve some realistic real-world goals and scenarios. These scenarios serve as a proof-of-concept and as a description of the power and possibilities GEPPETTO offers.

Chapter 5

Geppetto: Examples

In this chapter we present several examples and illustrations how to install hooks and links into a running Squeak system, how to implement a metalevel used to adapt dynamically the behavior of the running system.

The first illustration is dedicated to profiling: We are suffering from a serious performance bottleneck in a running application which we cannot afford to halt, so we install at runtime some hooks into these application. Their associated metaobjects measures the time spent in the different methods of our application.

In the next example we show how GEPPETTO is useful to implement a simple code coverage analysis tool on the metalevel, for instance to find out which methods in an application are not covered by the test suite we elaborated. Finally, we illustrate in the last example the use of a proceedable metaobject which caches the calculation of Fibonacci numbers. To fill the cache with the correct values the metaobject proceeds the replaced base level operation to get the results, so the replaced behavior from the base level is still accessible in the metalevel.

5.1 Illustration 1: Profiling

When we finished the development of a program and deployed the application it gets often very complicated and error-prone to change or extend this software later on without breaking anything. Especially it is hard to add reflection to an application when this was not initially designed with that intention in mind. To have the possibility to add reflective features even to a running system just when we need it is a very interesting and demanding ability. As soon as we not longer require these reflective capabilities we should be able to completely remove them from the system to bring it back to its initial state. We call this kind of reflection *unanticipated*, which means that we can add or remove every reflective functionality entirely at any point in time from a running system transparently.

Problem Statement.

This ability to use reflection unanticipated is of particular interest for appli-

cations running on a server which cannot be halted. This can be a web server providing dynamic websites such as a Wiki or a Forum system. We serve these web applications from a Squeak system and implemented them using the Seaside framework [DUCA 04].

Unfortunately, our Wiki system suffers somewhere from a performance bottleneck, our users complain that the website reacts slowly and they have to wait a long time for reading or writing content in the Wiki. We wish to spot and solve this bottleneck, but we do not have a clear understanding where exactly the problem lies. We know that some methods in our Wiki are poorly implemented and that we spend a lot of execution time in this method. Therefore, we want to know which methods consume the most time to be executed. We want to apply a simple profiler to our Wiki application, but we cannot afford to shutdown our server to install this profiler, because we have a large user base and our Wiki has to be permanently accessible. It is crucial for us to be able to profile our Wiki without having to stop and restart it, during the profiling our users should still be able to use the Wiki system as usual.

After we have gathered all the necessary information we want to remove the profiler entirely from the system, also without being forced to halt the Wiki. We have also the strict requirement to profile the application in its natural environment, because the performance bottleneck does not occur in any other environment, it is bound to the circumstances we encounter in this special setup on the running server.

Reflective Needs.

To find out how much time we spend to execute which method of our Wiki system we can use simple reflective functionality. We install hooks in our Wiki system which allow us to reify runtime information, such as the name of the currently executed method or the class to which this method belongs. For every method in our application we install one hook just *before* the method and one hook right *after* the method. Every time when we call a method we execute the first hook, and as soon as the method has finished its execution, we execute the second hook. To determine the time spent for the evaluation of the whole method we simply measure the current timestamp in the first as well as in the second hook. The difference between these two timestamps is the execution time of this method.

The operation we reflect in this scenario is called message receive. An object receives a message send and executes the method denoted in this message. We trap this event to execute a hook before and after this method. In this hook we are able to reify information about the operation occurrence, for instance the name of the method or the arguments that were passed to this method.

In our situation we just have to know the class and the name of the method to find out which methods are really slow in our application. We also specify how to *control* the base level operation. In this scenario we used a before and after control, this means we execute “something”, a hook, before and after the base level operation.

The hooks installed into the system perform a shift to the metalevel when being

executed. On the metalevel we implement our metaobjects like we engineer any other objects in the base level.

To gather the statistical data required to find the performance bottleneck we use just one simple metaobject which expects information about the method being executed and which calculates the time we spent in this method. We store this information in a database or any other persistent storage to be able to analyze the data as soon as we finished the profiling of our Wiki. When we have gathered enough data, we remove all the installed hooks and thus also all reflective behavior we used during the profiling. Our system is afterwards exactly the same state as it was before we introduced any hooks.

Solution.

We now present a possible solution for this profiling issue. We have to install hooks for every method in our Wiki application, so we select all classes in the whole package Wiki. Then we use the operation *message receive*, because we have to intercept the execution of every method. Because we want to measure the time required to execute a method, we use a *before and after* control and thus installing hooks at the beginning and at the end of each method in the Wiki package. We do not need to specify an activation condition, because our profiling link should always be active once it is installed. When we have finished the profiling, we remove the link entirely from the system. We just need one single metaobject for every hook installed in our Wiki package, so we define global scope, thus the single metaobject is valid for the whole hookset.

Finally, when we specified the installation of the hooks into the system, we can elaborate the call to the metalevel. We provide a special class called ProfilingMO for the metaobject. This class has a method #profileMethod: which is called on the metaobject in the hook and to which we pass all the reified information packed into an array. We just reify the selector of the executed method and the object which receives the message send invoking this method.

The code snippet below illustrates the configuration of the link required for our profiling task.

```
link := GPLink id: #profiling.
link classEnumerator: (GPPackage packageName: 'Wiki').
link operation: GPMsgReceive.
link control: GPControl beforeAfter.
link moCall: (GPCallDescriptor
          object: ProfilingMO new
          selector: #profileMethod:
          parameters: {GPParameter selector.
                       GPParameter self}
          passingMode: GPPassingMode array).

link install.
```

The hooks defined with this code are installed into all methods of our Wiki application. As soon as we installed the link, the metalevel method #profileMethod: will be called when a Wiki method is invoked. We can therefore gather the statistical data to profile the methods of our application. For instance, we want to know the twenty methods requiring the most time to be executed, in average. We store this information in the metaobject or write it into a database - the implementation of the metalevel can be arbitrarily complex.

Next we provide a very simple implementation of ProfilingMO>>#profileMethod: where we just measure the time to execute a method once. We do not calculate the execution time for the same method twice, as soon as we have a value for the execution time of this method, we ignore any further executions of the same method.

```
ProfilingMO>>#profileMethod: anArray
  | receiver selector startTime endTime executionTime |
  selector := anArray first.
  receiver := anArray second.
  method := receiver class>>selector.

  "skip if we have measured this method already"
  (self methodDict includesKey: method) ifTrue: [^self].

  control = #BEFORE ifTrue: [
    startTime := TimeStamp millisecondClockValue.
    self tempMethodDict at: method put: startTime.
  ] ifFalse: [
    startTime := self tempMethodDict at: method.
    endTime := TimeStamp millisecondClockValue.
    executionTime := endTime - startTime.
    self methodDict at: method put: executionTime.
    self tempMethodDict removeKey: method.
  ]
  ^self
```

The array passed to #profileMethod: contains the reified information. The first element in the array is the selector of the method executed in the base level, the second the receiver of the message send that executed the method. We can construct an object denoting this method with the code receiver class>>selector.

Then we check if we already have gathered a value for the execution time of this method. If this is not the case, we check if the metalevel method was executed by the *before* hook or by the *after* hook. If we are in the *before* hook we store the start time, the current timestamp in milliseconds, in a temporary dictionary with the method as a key. If we executed the *after* hook instead we fetch again the start time from the temporary dictionary and calculate the difference between the end

time of the method execution, the current time stamp, and the start time. This value is then stored in the persistent dictionary, again with the method as the key.

Validation.

Of course, the resulting execution times of all the measured methods are not accurate, because we have to spend much time to reify the needed information and to actually measure the execution time in the metaobject. If we have not installed these hooks, the methods would execute much faster. But we are mainly interested in the proportions between the different methods in our Wiki application, it does not matter, if we cannot measure the execution time exactly as it is. The time we lose for the measuring is for every method almost the same, more or less, so the proportions between different methods are still correct and accurate.

5.2 Illustration 2: Code Coverage

By following test-driven development process we implemented many test cases during the implementation of an application. Nonetheless, we will never achieve a complete test base which covers every method and every possible execution branch of our whole application. But we are always interested in improving the test coverage of our systems, the goal is to cover as many classes and methods as possible with tests. It is therefore important to get information about the test coverage of a given application, to find out which methods are covered directly or indirectly with one or more test cases and which methods are not tested at all and hence need a test case.

We can profit of the possibilities provided by reflection to actually gather the information about method covered or not covered by test cases. The idea is simple and straightforward: Before we run our tests for a certain application we install hooks into every method belonging to our application, afterwards we run our tests. The installed hooks have the responsibility to log every method that was called during the execution of the tests. At the end, we know every method of our application which was covered directly or indirectly with at least one test. We now present our solution for this task.

Basically, we have to introduce a hook for every method in our application which we can do by selecting the whole package containing our application with the package enumerator. The operation to intercept is *message receive*. We decided to use a *before* control, so we log that a certain method is executed just before we start its evaluation.

The configuration of the hookset and link is similar to what we used in the previous example in section 5.1 where we also installed a hookset for the message receive operation in a whole package. We reify the same information here, the object receiving the method invocation and the name of the selector. Having reified this data we know in the metalevel which method was actually executed in the base level.

Here is a listing of the code used to configure hookset and link.

```
hookset := GPHookset new.
hookset operation: GPMsgReceive.
hookset classEnumerator: (GPPackage packageName: 'OurApp')

link := GPLink id: #codeCoverage hookset: hookset.
link control: GPControl before.
link moCall: (GPCallDescriptor
        object: self
        selector: #log:
        parameters: {GPParameter self.
                GPParameter selector}
        passingMode: #ARRAY).
link install.
```

We have implemented this configuration in a method called GPCodeCoverage>>#install, so self which we used for the metaobject in the code above refers to an instance of GPCodeCoverage.

This class GPCodeCoverage is implemented as a singleton. On the class-side we have the method #forPackage: to which we pass a string denoting the name of our package, in the above example *OurApp*. When we create an instance of GPCodeCoverage like this:

```
| codeCoverage |
codeCoverage := GPCodeCoverage forPackage: 'OurApp'.
```

The method GPCodeCoverage>>#install is executed and all hooks we defined with the hookset above are installed into our application, this means that every method in every class in package *OurApp* has a *before* hook installed.

After the installation of all the necessary hooks we can run our test suite implemented for this application. While the tests are running we constantly log information about executed methods in our codeCoverage object which serves as the metaobject. The hooks executed before the evaluation of every method in our package invoke the method #log: on codeCoverage and pass the reified information, receiver of the method call and selector, to this log method. Internally in codeCoverage we write every executed method into a log.

Finally, after the execution of the tests we have all the information about the executed methods in the codeCoverage object. When we send the message #simpleStatistics to this object we get the result printed out to the Transcript. We can see in the statistics which methods of our application are indeed directly or indirectly covered with tests, but also which methods were not invoked with any of the test cases.

We can directly access the methods covered or not covered with tests by sending the message #methodsWithTests or #methodsWithoutTests, respectively, to the codeCoverage object. A method is denoted by the class to which it belongs and the name of its selector.

Validation.

This example shows how a very basic and simple test coverage analysis tool can be implemented with reflection by using GEPPETTO as the reflective architecture.

We can evolve this simple example to a more elaborated coverage tool. For instance, we could divide the methods covered with tests into methods directly called in a tests and those indirectly invoked during the execution of a test case by using subjective features provided by GEPPETTO. When using subjectivity we analyze the stack frame to find out *who*, which method, called a given method. This means that we can find out if a method of our application was called directly in a test or just indirectly. But the presented example is already a good illustration for the power of our reflective model.

5.3 Illustration 3: Proceedable metaobject

In this last illustration we describe how we proceed in the metalevel a replaced operation of the base level by using a special metaobject called GPProceedMO. The example we present caches the results of a Fibonacci calculation on the metalevel. The Fibonacci algorithm is recursively implemented in a base level method. We replace this method with behavior defined on the metalevel, but in our metaobject we want to use this replaced behavior to actually calculate the Fibonacci numbers for input values not already contained in the cache. We then store the result of the calculation in the cache to speed up further calculations for the same input value.

A recursively defined Fibonacci calculation has a very poor time complexity (basically $O(1.5^n)$), by using a caching mechanism we calculate for every input value *n* the result of the algorithm just once which drastically speeds up the calculation of the Fibonacci number for given *n*.

Realization.

We explain now the Fibonacci algorithm we implemented in the base level:

```
GPFibonacci >> #fib: n
   (n = 0 or: [n = 1]) ifTrue: [^n].
   ^(self fib: (n - 1)) + (self fib: (n - 2))
```

In class GPFibonacci we provide the method #fib: which calculates recursively the Fibonacci numbers for a given n. This algorithm is well-known in computer science and needs no further explanation.

More interesting is the link definition we use to set up our caching mechanism on the metalevel. Basically we utilize the message receive operation to intercept every call to GPFibonacci>>#fib:. By using the replace control we exchange this algorithm with behavior defined in the metaobject. Here is the link configuration code for that purpose:

```
link := GPLink id: #fibonacci.
link operation: GPMsgReceive.
link inClass: GPFibonacci.
link inMethod: #fib:.
link control: GPControl replace.
link moCall: (GPCallDescriptor
        object: self
        selector: #cachedFib:
        parameters: {GPParameter arg1}
        passingMode: #PLAIN).
```

Class GPFibonacciMO represents the metaobjects and implements the caching mechanism. GPFibonacciMO has a method #cachedFib: in which we calculate the Fibonacci numbers based on a cache to improve the performance of this calculation. We pass just the sole argument of #fib:, the n, in plain mode to this metalevel method.

Method GPFibonacciMO>>#cachedFib: is implemented as follows:

```
GPFibonacciMO >> #cachedFib: n
  | result |
  (self cacheContains: n) ifTrue: [^self cacheAt: n].

  result := self proceed.
  self cacheAt: n put: result.
  ^result
```

If the cache, which is a dictionary, contains a result for the given n we answer the value stored in the cache. Otherwise, we calculate the result by proceeding the replaced base operation GPFibonacci>>#fib:. We store the retrieved result in the cache and answer it.

Class GPProceedMO.

The call to the #proceed method in the metaobject is central. If we create a subclass of GPProceedMO we can proceed any replaced operation in metaobjects of the resulting metaobject class. GPProceedMO represents proceedable metaobjects, the method #isProceedable, which is also defined in Object, answers *true* in GPProceedMO.

input (n)	without cache (ms)	with cache in metaobject (ms)
20	4	0.01
30	432	0.01
40	51785	0.01

Figure 5.1: Time spent to calculate Fibonacci numbers with and without caching mechanism.

During the reification in the hooks we save the replaced operation into the metaobject, but only if this metaobject is defined as proceedable. Thus we are able to proceed any replaced operation in the metaobject by just sending the message #proceed to the metaobject. This proceed operation evaluates the replaced operation which the executed hook has stored in this metaobject. Such an operation is for example a MessageSend or a BSInstVarAccess, depending on the base level operation we replaced. In method #proceed of the metaobject we just send the message #value to the stored operation to execute it and to get the result of this execution which we answer. Providing objects representing replaced operations and proceeding them by sending #value are features supported by BYTESURGEON.

Validation.

This proceed mechanism is powerful in all cases where we require access to the value of the original, replaced operation. Unfortunately, reifying the information needed to create an object for the replaced operation is time-consuming, because we have to reify everything concerning this replaced operation, such as the receiver, the selector and the arguments in the case of a message send. Executing this reified operation later on in the metalevel is also not cheap, but affordable.

Note that such a proceed mechanism is of course just available for replaced operations, not when using a before or after control, where the intercepted operation is still normally executed in the base level.

To conclude this example we show the amazing differences in execution time between using a cache in the metalevel and just using the original recursive implementation of the Fibonacci algorithm for some input values in Figure 5.1.

These measurements are impressive: The time consumption of the original, recursive Fibonacci algorithm jumps up enormously with increasing n, whereas the time consumption of the Fibonacci algorithm in the metalevel where we applied a simple cache mechanism is almost the same for every n. This proves that the performance penalty we get by reifying the information required to provide a proceed mechanism of the replaced base level operation is not huge compared to the very inefficient Fibonacci algorithm we replaced.

Note that we cannot afford to measure the time taken by the Fibonacci calculation for bigger input values, because the calculation with original algorithm would just take too long. But the metalevel implementation of Fibonacci can easily calculate the Fibonacci numbers for very large input values without requiring much time

for that, *e.g.,* for n = *10000* we just spend 1 ms to calculate the Fibonacci number.

5.4 Summary

In this chapter we presented several illustrations showing how to use GEPPETTO and what different goals we can achieve with this framework. These manifold examples give a comprehensive introduction into the possibilities GEPPETTO offers. They serve as a proof-of-concept or as a documentation for a programmer who wants to learn if GEPPETTO is useful for his purpose and how to run it. Of course we covered not every possible aspect of GEPPETTO in these illustrations, many more scenarios, where the use of GEPPETTO is useful or even indicated, are conceivable.

The next chapter gives answers to the question how we concretely implemented the design presented in chapter 4 in Squeak by analyzing and explaining each important package and class of GEPPETTO.

Chapter 6

Geppetto: Implementation

In this chapter we cover the implementation of GEPPETTO. We analyze in detail the interfaces of the important core classes and how they collaborate with other classes. We also explain the internal implementation of these classes.

We divided the GEPPETTO project into several packages, *e.g.,* Geppetto-Core, Geppetto-Hookset or Geppetto-Link. We now describe these packages and the classes they hold. But first we illustrate the mechanisms and tools used to insert hooks into the bytecode of methods.

6.1 Bytecode Transformation to Insert Hooks

Our approach to unanticipated partial behavioral reflection requires the insertion of so-called hooks, relatively small pieces of bytecode, directly into the original bytecode of methods. Hooks are placed in all methods where we want to reify operations, such as a message send or an temporary variable access. This hook insertion can occur at any time in any method of any class in the whole running system, even in system classes.

6.1.1 ByteSurgeon

Because bytecode modification is a complex, error-prone and cumbersome business we have not implemented the bytecode related facilities directly in GEPPETTO, instead we decided to use a bytecode transformation tool called BYTESURGEON [DENK 05]. BYTESURGEON is a powerful and easy-to-use framework for runtime manipulation of bytecode in Squeak/Smalltalk. With this tool we do not have to manipulate directly the bytecode which would be very tedious. Instead we write our hooks directly in normal Smalltalk code which we then pass in a string to BYTESURGEON. Internally, BYTESURGEON compiles this Smalltalk code to bytecode and inserts the code at the desired places.

Example.

Here is a short example showing how we use BYTESURGEON to insert a simple piece of Smalltalk code into the method #foo of class Bar.

The method Bar>>#foo contains four lines of code:

```
(Bar>>#foo)
  | coll |
  coll := OrderedCollection with: 'a' with: 'b' with: 'c'.
  self assert: coll size = 7.
  ^coll
```

This assert condition obviously fails. We now modify the bytecode of this method to make sure that the assertion holds. We have to replace the message send to the coll object, invoking method #size in line 3, to return the constant 7.

We do this by instrumenting method Bar>>#foo with BYTESURGEON. We are interested in message sends, but only those invoking selector #size. Because an invocation of a method #size just occurs once in Bar>>#foo it is safe to just look for message sends invoking method #size and to replace these sends with the constant 7.

The following code achieves exactly this behavior:

```
  (Bar>>#foo) instrumentSends: [:send |
      send selector = #size ifTrue: [
        send replace: '7']]
```

We first select the operation in which we are interested (*e.g.,* message sending, instance variables access or temporary variables access) by calling the correct instrumentation method, *e.g.,* #instrumentSends:, #instrumentInstVars: or #instrumentTempVars:. This is referred to as *operation selection.* Second we select a particular instance of this operation, an operation occurrence, namely just message sends invoking a method #size. We call this *intra-operation selection.*

Instrumenting Methods.

To instrument a method we use the different instrument-methods provided by BYTESURGEON. All of them expect a block as their sole parameter. This block is evaluated for every operation in the given method, so we are sure that our block is checked against every operation occurrence of interest. We do intra-operation selection by specifying a condition in the block, *e.g.,* checking if an IRSend invokes a specific method. We change this operation occurrence, either by replacing it or by inserting code before or after this operation, only if this condition is met.

To actually change an operation occurrence we send either the message #replace:, #insertBefore: or #insertAfter: to the operation object, *e.g.,* to send in

the example above. We pass a string to these methods in which we write normal Smalltalk code denoting the behavior to insert. In this string we have also access to meta variables to use runtime information in our code, *e.g.,* <meta: #*receiver*> or <meta: #*arguments*> to get the receiver or the arguments of a method, respectively.

Conclusion.

This short introduction into the possibilities offered by BYTESURGEON shows how GEPPETTO is doing the selection of all the places where we have to install hooks, and how the insertion of these hooks is achieved. To get a more comprehensive impression of all the interesting features of BYTESURGEON and how to use them we refer to [DENK 05].

Besides BYTESURGEON we do not require any special tools for running GEPPETTO in a Squeak system.

6.2 Geppetto-Core Package

In this package we find some important core classes of the GEPPETTO framework which we use also in other packages.

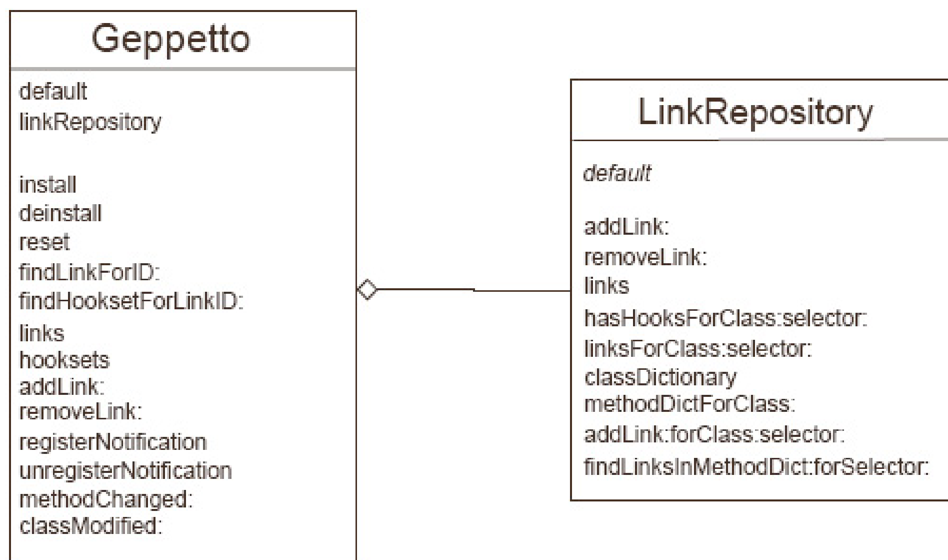

Figure 6.1: Package Geppetto-Core

The main class of GEPPETTO is simply called Geppetto. This class contains knowledge that we have to access from different points in the GEPPETTO framework and acts as a facade. For instance we need to know what links we have installed in the system to be able to access or change or to even remove them from

the system.

Notification.

Another important issue is the notification mechanism of changes in the system, as described in 3.5. Because Squeak/Smalltalk is a runtime environment where a user can dynamically change and recompile methods, we need a mechanism to re-insert hooks that were accidentally removed during the recompilation of a method. The notification mechanism of Squeak informs us when and where such a recompilation happened, so we can check if the recompiled method needs any hooks to be inserted. If so, we dynamically reinstall all defined hooks that we have had installed before this method was recompiled. This has the advantage that the system does not loose any hooks we once installed, if we do not explicitly remove them using methods provided by GEPPETTO, such method #deinstall of the hookset class.

We use this notification mechanism also to get informed when a new method was added to the system. With class and method selector we are able to also include classes and methods that do not exist at the moment when we write the definition. But as soon as methods are added to the system that indeed match the conditions defined with class and method selectors we dynamically add hooks to these methods. This way we can also install hooks for future methods which do not exist at the time we install a link but might be added later on. This is a step forward to a more dynamic approach for defining unanticipated reflective means. Of course, we can also specify that we do not want to apply our definition of hooksets to future entities but just to currently existing ones. The default setting is to apply reflective definitions also to future entities.

Class Geppetto

The Geppetto class is implemented as a singleton: Only one instance can live in a system. We access this sole instance by sending the message #default to Geppetto. The registration mechanism mentioned above always notifies this sole instance of class Geppetto. We unregister this object from receiving these system notifications by calling the method #unregisterNotification on Geppetto default. With #registerNotification we register our sole instance again into the system notifier. We define that we are interested in several system events, such as adding new methods or changing existing methods.

For every change (*i.e.,* recompilation) to a method we execute Geppetto>>-#methodChanged: which expects a *method changed event* as an argument. We ask this event for the class and the selector of the changed method. With this information we can easily check if we have installed any hooksets that affect this method. If this is the case, we reinstall all the hooks belonging to this method to make sure that our reflective definitions are still fulfilled even after changes to the system.

Of course, it is possible that a method is changed in a way that we cannot install

a hook again after this change. For instance, we have removed a message send in a method for which we have installed a hook before. After our change to the method the hook also disappears with the message send and is not installed again.

Accessing Links.

In class Geppetto we also provide methods to look up installed links in the system. With #findLinkForID: we look up a link in the system with a given linkID. If this method does not find any installed link with this ID, it answers nil. Another method is #findHooksetForLinkID:. This method also expects a linkID but answers the hookset associated with the link belonging to the given ID. Using these finding methods we can easily retrieve information about installed hooksets and links from anywhere in the system. The methods #links and #hooksets also provide access to a collection with all currently installed links and hooksets, respectively.

Installation of Links.

We can also install and deinstall hooksets with the Geppetto class. For this purpose we have implemented two methods #install and #deinstall. By calling them we simply install all defined links or deinstall all installed links, respectively.

To add a link to the system we use the method Geppetto>>#addLink:. The passed link is then stored in global link repository. As soon as we call the method Geppetto>>#install we install every link stored in this global link repository that is not already installed. With Geppetto>>#deinstall we remove all links marked as installed in this link repository from the system.

Class GPLinkRepository.

This global link repository is implemented in the class GPLinkRepository. Just one instance of this class should exist in the system, this sole instance is already associated with the Geppetto class. In the link repository we provide convenient methods to find out which class and method has associated links. We store for every class and also for every method of a reflective class all the associated links. Therefore, we can easily determine if we need to reinstall hooks in a changed method by asking the method #hasHooksForClass:selector: of the link repository. This method looks up if we have defined links for the given class and for the given selector. If this is the case, the method answers *true* and we reinstall all hooks in this method. Other useful method of GPLinkRepository are #linksForClass:selector: which answers all installed links in the given method, or #methodDictForClass: returns all methods of a class that have links installed.

6.3 Geppetto-Hookset Package

We describe now the package *Geppetto-Hookset*, starting with class GPHookset.

6.3.1 Class GPHookset

To define hooksets we use the class GPHookset. This class represents one hookset which basically defines all the places where hooks have to be installed. GPHookset further knows how to install these hooks by invoking GPHookInstaller with the appropriate parameters. This hook installer installs the right hooks at the right places, using BYTESURGEON internally to transform the bytecode as needed. We just pass the hookset itself to the hook installer, the installer then asks the hookset for all the data required to install the hooks correctly.

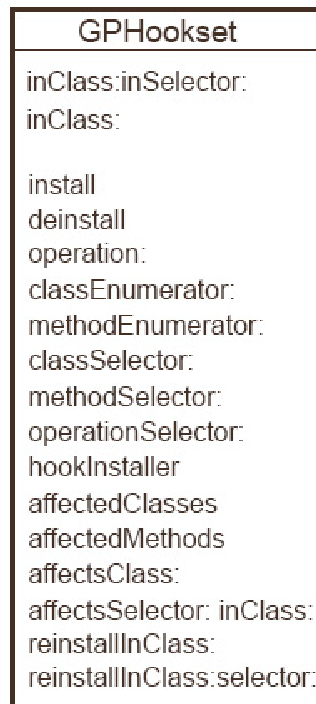

Figure 6.2: Hookset class in detail.

Every hookset is at least associated with one link, which is stored in an instance variable. We require access to the link in every hookset to know, for instance, if we have to install our hooks before, after or instead of the original operation.

We create a hookset by using the convenient constructor method #inClass:inSelector: to which we pass a class and a selector. This hookset affects just this single method, no other method or classes. Internally, GPHookset creates appropriate class and method enumerators for that purpose. We can use GPHookset class>>#inClass: to select just one single class.

The implementation of these spatial selection mechanisms called entity enumerators, entity selectors and operation selectors are presented in the next sections.

Example for Spatial Selection.

In the next example below we present how we define and use these spatial selection mechanisms.

We select the operation GPMsgSend, denoting all message send operations. By specifying an operation selector, we further redefine the selection to just message sends invoking the method #isLiteral.

We also define precisely in which entities these operation occurrences take place. With the class selector we choose class Array, with the method selector the method #printOn: of class Array. This means that we finally just selected message sends calling method #isLiteral in Array>>#printOn: with the hookset configuration below.

```
hookset := GPHookset new.
hookset operation: GPMsgSend.
hookset classSelector: [:class | class = Array].
hookset methodSelector: [:selector | selector = #printOn:].
hookset operationSelector: [:send | send selector =
#isLiteral].
```

To install the whole hookset we simply evaluate hookset install. We can also just install hooks for one single class or even one single method by using either the method #reinstallInClass: or #reinstallInClass:selector: of class GPHookset.

During the installation of a hookset all the affected classes and methods are stored in collections, and we can access these collections with the methods #affectedClasses and #affectedMethods, respectively. We have some testing methods to find out if a class or a method is affected by this hookset, *e.g.,* #affectsClass: or #affectsSelector:inClass:.

Selectors and Enumerators

We also define in a hookset the possibilities to select all the entities (*i.e.,* classes and methods) in which hooks have to be installed. For that purpose we specify enumerators and selectors to select classes and methods. A hookset can have a class enumerator, a class selector, a method enumerator and a method selector. The selectors are blocks, whereas the enumerators are instances of either GPClassEnumerator or GPMethodEnumerator, or of subclasses of one of these two classes. We can use both means to select classes or methods simultaneously, the hook installer selects automatically all classes and methods that are either affected by the definition of the selector or of the enumerator. This means that when we define selectors and enumerators for the same hookset the union of all affected entities are selected at the end.

Entity Enumerators.

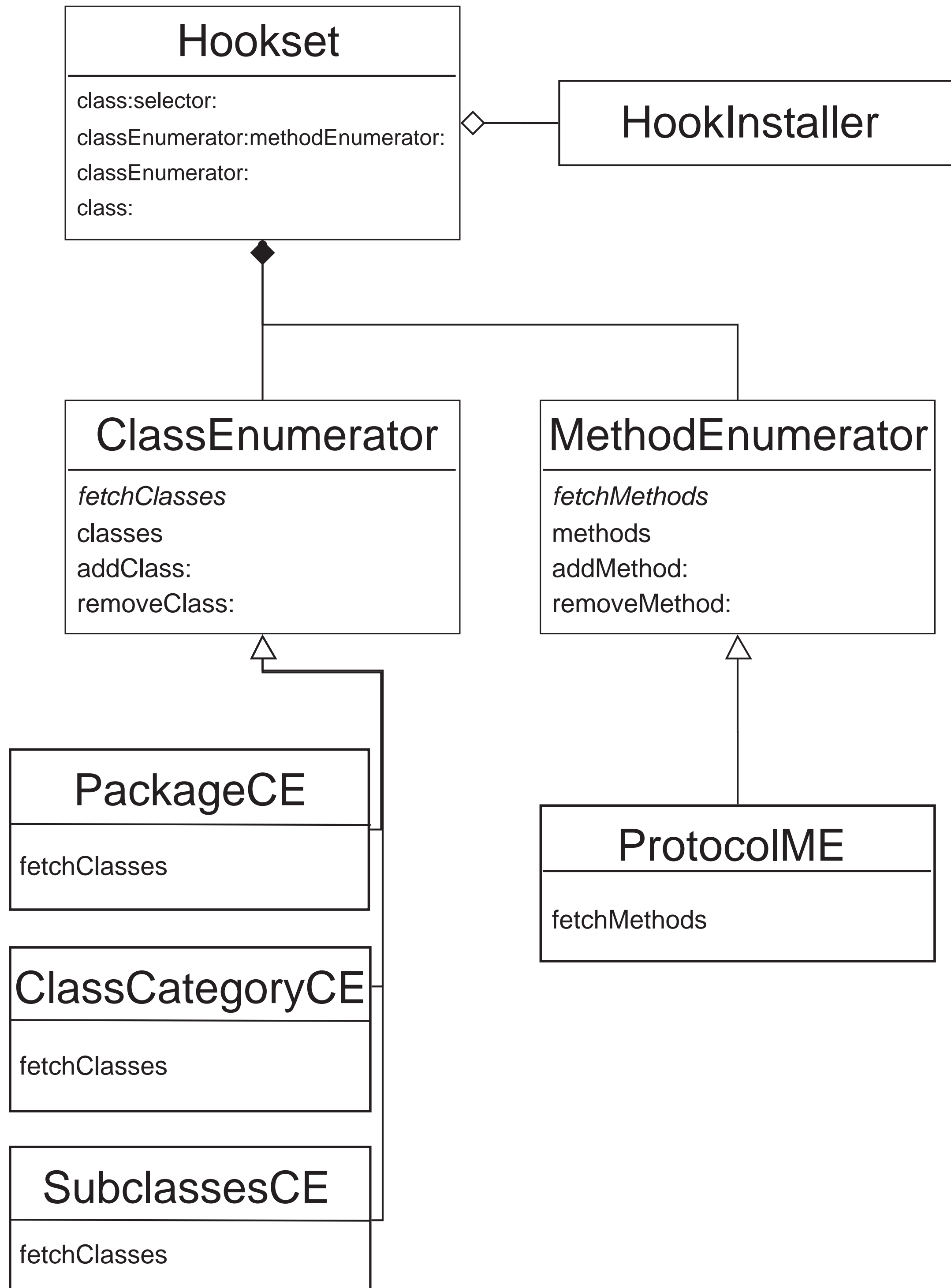

Figure 6.3: Package Geppetto-Hookset

Class enumerators and method enumerators are instances of either GPClassEnumerator or GPMethodEnumerator, or of subclasses of these two classes. These two base classes represent basically a collection of classes or selectors, respectively. Subclasses overwrite the method #fetchClasses or #fetchMethods, respectively, to enumerate the desired entities. For instance, we have implemented class GPPackageCE which enumerates all classes in a given package. GPProtocolME instead enumerates all methods in a given protocol of a given class.

We use these two classes as follows:

```
ce := GPPackageCE packageName: 'Geppetto'.
me := GPProtocolME inClass: GPHookset protocolName: 'accessing'.
```

We have also implemented a class enumerator for all subclasses of a given class, called GPSubclassesCE, or a class enumerator to collect all classes in a class category, called GPClassCategoryCE.

Entity Selectors.

Class and method selectors are simple blocks. Both expect one argument, a class or a selector, respectively. A class selector is evaluated for every class in the whole system. If the block evaluates to *true* for a class, this class is selected and turned into a reflective class. Method selectors are similar: For every reflective class we evaluate the method selector block and pass every method of these classes as a parameter to that block. If the method selector evaluates to *true* this method is selected.

If we do not specify a method selector or a method enumerator, we turn all methods of all selected classes into reflective methods. Of course, we install only hooks in any of these selected methods if there is an operation occurrence which matches our defined conditions for operation selecting.

Class and method selectors are evaluated during the installation phase of a hookset, in class GPHookInstaller.

As explained in section 3.5 it is important that these definitions for the entities in which hooks are installed also affect the "future". If either a method is recompiled or a new method is added to the system, GEPPETTO has to assure that hooks are installed in these methods if they are selected with the entity selection mechanisms.

Operation selector.

Other selection mechanisms are operation selection and intra-operation selection. Operation selection is done in a hookset by defining the type of operation we want to reify. We can reify message sending, message receiving, instance variable access and temporary variable access. These operations are represented with dedicated classes, such as GPMsgSend or GPTempVarAccess. If we do not specify an operation explicitly, GPHookset chooses GPMsgSend as the default operation.

The operation selector further redefines the selection to particular operation occurrences and is specified with a block. An operation selector does the static part of intra-operation selection, we cover the dynamic part later when describing the link class.

An operation selector is a block like class or method selector, but expects an IRInstruction as its sole parameter. Such an instruction is either a message send, an instance variable access, a temporary variable access, or several other instructions we not reify in GEPPETTO, such as jumps or returns.

If we want to reify for instance message sending as an operation, we can define an operation selector exactly specifying which operation occurrence we have to reify. For example, we just want to reify a message send in method Array>>#printOn: if this message invokes a method called #isLiteral. For that purpose we define an operation selector which is evaluated with every send instruction existing in method #printOn:. We check in the operation selector block if the given message send invokes the method #isLiteral with this piece of code, as already seen at the beginning of this section:

```
operationSelector := [:send | send selector = #isLiteral].
```

When we have selected the operation GPMsgSend, referring to every message send, GEPPETTO assures that only send instructions are passed to the operation selector, no other type of instructions of a method. A send instruction, an instance of class IRSend, understands the message #selector, referring to the method invoked by this message send. So we simply check if this selector is equals to #isLiteral to find every operation occurrence in all reflective methods matching our criteria.

Note that we cannot specify in which class #isLiteral has to be defined, this would be a dynamic intra-operation selection where we have to analyze the receiver of the message send to find out if this receiver is an instance of the desired class. We can achieve this behavior by defining activation conditions in a link, which we cover shortly in detail. When just specifying the desired method in an operation selector we mean simply every call to a method named #isLiteral, no matter in which class this method is defined.

6.3.2 Hook installer

This section illustrates the internals of the hook installer class by giving a short introduction into the functionality of this important class.

We create a hook installer by passing our hookset to class-side method GPHookInstaller class>>#install:. This method automatically installs all the hooks defined in the given hookset. A class-side method #deinstall: is also available which deinstalls all hooks of the given hookset.

On the instance-side of GPHookInstaller there are many methods holding all the logic used to install hooks at the right place in the system. These methods are

not dedicated to be used from outside class GPHookInstaller, they are mainly private methods.

Installation of a Hookset.

The first task of the hook installer during the installation of a hookset is to iterate over all classes defined in the hookset to turn them into reflective classes. If we specified a class selector in the hookset, the installer has to evaluate this selector for all classes defined in the system and to select every class for which the class selector, a block, evaluates to *true*. The reflective classes defined with an enumerator are easier to select, because they are stored in a collection in the enumerator. For every selected class the hook installer either analyzes every method selected in the hookset using method enumerators or selectors, or, if we have not specified any method enumerators or selectors, every method defined in the reflective class(es).

If we have found all these methods in which hooks have to be installed, the distinction between caller-side and callee-side operations has to be drawn. If we decided to use a caller-side operation, *e.g.,* message sending, the hook installer has to consider the operation selector we defined in the hookset. By specifying such a selector we choose just certain operation occurrences in a method, *e.g.,* not every message send, but just those invoking the method #foo. This operation selector has to be evaluated for every instruction in all the selected methods.

BYTESURGEON provides methods to instrument any method in the system and defines several instrument methods for that purpose. The different instrument methods depend on the operation they manipulate. For instance, there is #instrumentSend: or #instrumentInstVarAccess: available directly in class CompiledMethod. The first manipulates message send, the second instance variable accesses occurring in the receiver, a compiled method. Instrumenting a method includes the iteration over all instructions of a method to find out for which instructions we actually have to introduce a hook.

With MyClass>>#foo we obtain the CompiledMethod object for the method #foo in class MyClass.

We pass a block to these instrument methods and this block is then evaluated for every instruction in the instrumented method. In the following example, aBlock is evaluated for every message send occurring in MyClass>>#foo:

```
(MyClass>>#foo) instrumentSend: aBlock
```

We define in this *instrument block* that if the current instruction, which we obtain as a parameter in the block, meets the condition defined in our operation selector, a hook is installed for this operation. We also define *where* we want to have the hook: before or/and after the original operation or instead of it, by invoking the appropriate operation manipulation methods.

In the example below we instrument every message send in MyClass>>#foo for which our operationSelector evaluates to *true*, then we insert our hook after

such a message send. The hook installer asks the control attribute of the link to obtain the correct method it has to call on the send object, either #insertBefore:, #insertAfter: or #replace:, these three methods are the operation manipulation methods provided by BYTESURGEON.

```
(MyClass>>#foo) instrumentSend: [:send |
        (send evaluate: operationSelector) ifTrue: [
            send insertAfter: hook]].
```

Building the Hook.

Next, the hook installer has to build the hook, basically a string which we pass to one of the three methods mentioned above (#insertBefore:, #insertAfter: or #replace:). This string is transformed to bytecode by BYTESURGEON and this bytecode is then placed into the bytecode of the instrumented method.

A hook has basically two important tasks: First, it has to reify the information required in the metalevel about the reified operation occurrence. Second, the hook calls the metaobject defined in the link. Because we can deactivate links we have to check if the link for the hook is active, before we fulfill these two tasks. By sending the message #isActiveFor: to the link, passing the current object in which the hook is executed as a parameter, we can determine directly in the hook, if the link is active. If the answer is yes, we evaluate indeed the hook and reify and pass the required information to the metaobject. Otherwise, we skip the hook and evaluate the next operation, or, in the case of a replace control, we proceed with the original operation. Fortunately, we have always access to the corresponding link object in the hook, so it is easily to ask the link for information, such as its active status.

Reification.

To handle our first task in the hook, the reification of the required data, we ask in the hook installer the passing mode object defined in the link for the necessary code to reify the information in the schema required for the selected passing mode. If we for instance use passing mode *Array*, hence passing the reified information in an array to the metaobject, we create this array directly in the hook and fill it with the reifications we get at runtime. The passing mode object answers the necessary code to create the array and to reify the desired information.

Internally, the passing mode asks the parameters for the code needed to reify a certain piece of information. As we learned in chapter 4 the parameters are defined in a MOP descriptor and specify all the reified information we require in the metaobject. BYTESURGEON provides some meta-variables which we write directly in the string we pass to *e.g.,* the #insertAfter: method. Such a meta-variable is for instance <meta: #*receiver*> or <meta: #*selector*>. When we ask a parameter for the reification code it answers us such a meta-variable. GPParameter receiver for instance answers <meta: #*receiver*> when sending the method #evaluate to it. Using the intelligence of the passing mode object and the param-

eters, it is not complex for the hook installer to build the reification code.

Shift to the Metalevel.

As soon as the reification is finished, we pass the reified information to the metaobject. Therefore, we have to be able to access the metaobject in the hook and have to know which message we have to send to the metaobject. How to obtain the metaobject in the hook is dependent on the scope setting we specified in the link. It is naturally to ask the scope object for the code to insert into the hook.

The object scope, for instance, answers the metaobject defined for the current object in which the hook is executed. The global scope gives us the metaobject specified in the link, because here we have one single metaobject for the whole hookset, which is set in the link.

As soon as we can access the metaobject in the hook we just have to know which method to call on it. We get this information from the link, because we defined the metaobject selector within a MOP descriptor associated with the link. The passing mode object, which is part of the MOP descriptor, knows how to pass the reified information to the metaobject, hence this object answer us the correct metaobject selector when we send #moSelector to it.

Now we put these two pieces, the reification and the shift to the metalevel, together and send the message determined by the metaobject selector to the metaobject. The reified information are the arguments of that message. Either they are packed into an array or an object, or we pass them as plain arguments to the metaobject, if we specified plain passing mode.

An Example.

In the next example we obtain the metaobject from the link, because we use global scope. The metaobject selector is called afterMsgSendTo:withArgs:selector:.

We specify passing mode plain, so we write the meta-variables which reify the information at runtime directly into this metaobject selector, without wrapping them into an array or an object. Finally, we insert this hook after any message send matching the criteria defined in the operation selector into method MyClass>>#foo.

```
hook := '(<meta: #link> isActiveFor: self) ifTrue: [
        <meta: #link> metaobject afterMsgSendTo: <meta: #receiver>
          withArgs: <meta: #arguments> selector: <meta: #selector>.
    ]'.

(MyClass>>#foo) instrumentSend: [:send |
        (send evaluate: operationSelector) ifTrue: [
            send insertAfter: hook]].
```

This short introduction into the complexity of the hook installer shows us how easy we can transform bytecode in Squeak using BYTESURGEON. In fact, we have to deal nowhere directly with bytecode, we always specify our reflective requirements in normal Smalltalk code, BYTESURGEON manages transparently the bytecode manipulation of methods.

6.4 Geppetto-Link Package

This section introduces the Geppetto-Link package which contains some very important class, such as GPLink itself or the scope and control hierarchy. These classes form a cornerstone of the functionality provided in GEPPETTO.

6.4.1 Class GPLink

A link causally connects the base level with the metalevel. In GEPPETTO a link is represented with class GPLink. Every link instance has a hookset associated which basically determines the places in the base level where reification occurs and where this link does a shift to the metalevel. The link is responsible for specifying what information has to be reified and how this information is passed to the metalevel, concretely to which metaobject and to which method in this metaobject. As mentioned in section 3.4.1 a link has several important attributes used to configure the reification process and the shift to the metalevel. We describe these attributes shortly one after the other.

Common attributes

An important attribute of the link is the so-called *linkID* which has to be unique in the whole system. We need to specify such a unique ID which is a symbol for every link.

Because we have to specify a *linkID* and at least one hookset for every link before we can install it, we provide some constructor methods to simplify the initialization process of a link. We have the methods GPLink class>>#id: and GPLink class>>#id:hookset: that offer convenient means to create new links.

In the following we present the complex attributes of a link that are an integral part of our standard MOP.

Control attribute

With the control attribute of a link we define when the shift to the metalevel has to occur: before, after or instead of the original operation in the base level. The hooks we install in our system are placed in the bytecode according to the setting of this control attribute. If we for instance set the control to *After*, all the hooks in the hookset associated to this link are placed after the original operation into the bytecode. Hence the hook installer asks the control attribute of the link to know

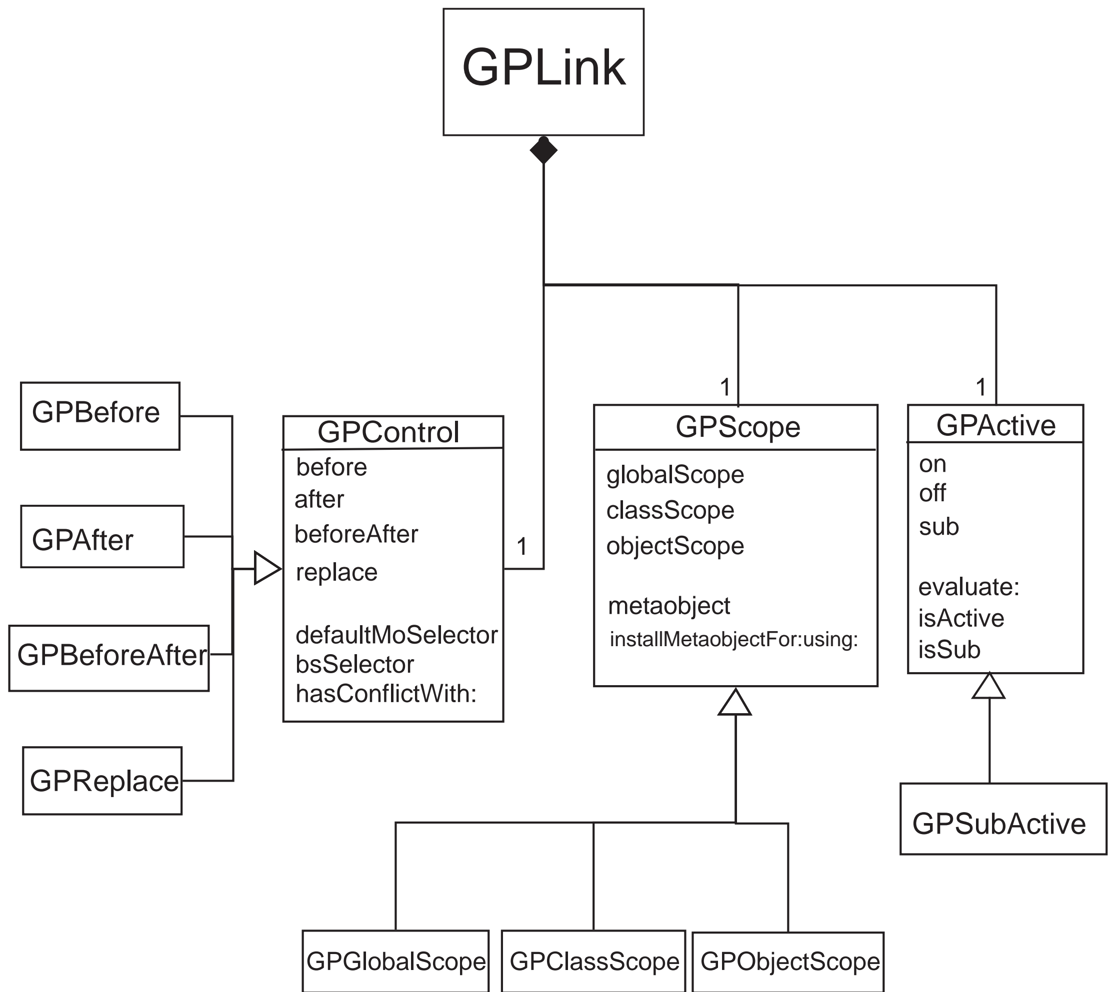

Figure 6.4: Package Geppetto-Link

where it has to install the hooks, before or after the operation, or if it has to replace the base level operation.

The control attribute is always an instance of any of the four subclasses of GPControl, either GPBefore, GPAfter, GPBeforeAfter or GPReplace. Only one instance of every of these four classes should exists in the system, therefore we provide factory methods on the class-side of GPControl that are used to identify these four objects. These methods are called #before, #after, #beforeAfter and #replace.

Class GPControl.

On the instance-side of GPControl we define several methods, for instance test methods to check if the current control is a before, an after or a replace control. Then we have methods called #moSelector and #moSelectors to provide the default metaobject selectors. When we use for instance a before control, we assume that the default selector in the metaobject is called #before: and expects an array with all the reified information as an argument. This default selector is of course only used when we do not define explicitly a method to be invoked when a hook is executed.

Furthermore, we have to define how a specific control is installed into the bytecode. As mentioned in section 6.3.2 the hookset installer internally uses BYTESURGEON which provides methods to install code before, after or instead of an operation. These different methods are called #insertBefore:, #insertAfter: and #replace:. We write in the method #bsSelector overridden in every control subclass how the appropriate operation manipulating method of BYTESURGEON is called. GPReplace>>#bsSelector for example answers #replace:.

We have defined all these methods mentioned above as "abstract" in GPControl, which means that we have not provided an implementation. In every subclass we have then implemented these methods appropriately according to the requirements of the specific control. Hence these controls hold behavior used in the hookset installer or in the link, and we can hence use these attributes of the link in a generic way from different places.

Scope attribute

With the scope attribute we specify for which entity a metaobject should be valid, so we define the scope of a metaobject. Such an entity is either the whole hookset, one class or even one single object. If a metaobject is valid for the whole hookset we say that it has global scope, otherwise it has either class scope or object scope. A metaobject with global scope is directly associated with the corresponding hookset. When using class scope the metaobject is stored in a class variable of this class, and for object scope we store the metaobject in an instance variable of the class. Hence we can have a dedicated metaobject for every reflective class affected by a hookset or even for every object of a reflective class.

We have to dynamically add these class or instance variables to reflective classes

when we decide to use class or object scope. These variables are named accordingly to the unique *linkID* every link has. If the *linkID* is 'myID', we install a class or instance variable called 'metaobject_myID' in the reflective class affected by a hookset whose link has either class scope or object scope defined as the validity range for a metaobject.

The reason why we prefer to directly store the metaobject in a variable instead of using for instance a global accessible dictionary is performance: Fetching the metaobject from a class or an instance variable is very fast and we can omit an expensive lookup in a dictionary. The drawback of this solution is of course the installation of these variables which requires that we recompile the class and also all of its subclasses. But it is better to pay this price once, at installation time, instead of loosing time whenever we execute a hook. The goal of GEPPETTO is always to be a fast as possible during the runtime of the system, even though the costs for the installation of the hooks are then higher.

Similar as for the control attribute we have a base class called GPScope and one subclass for every kind of scope, called GPGlobalScope, GPClassScope and GPObjectScope. The base class defines some abstract methods such as an accessor for the metaobject, simply called #metaobject, or methods to install the metaobject in either the hookset, a class or an instance variable. Some test methods to check if we have a global, a class or an object scope are also available.

Installation of the Metaobject.

Installing and accessing the metaobject if it has global scope is very easy: We just store the metaobject in an instance variable of the link called *metaobject*. For class and object scope more work is required: We have to check at installation time if there is already a class or an instance variable for the metaobject in all the reflective classes of our hookset. If not, we install first the class or instance variable into the reflective class, then we store the metaobject into this variable. When accessing the metaobject later on we have to read always the content of either the class or the instance variable for the metaobject.

For the installation of an instance variable in an given object we use the following code snippet:

```
anObject class addInstVarName: 'metaobject_', aLink idString.
```

To place the metaobject into the previously created instance variable we use:

```
anObject instVarNamed: 'metaobject_', aLink idString
    put: (aLink metaobjectForObject: anObject)
```

The link knows at installation time all the metaobjects for the different objects that we want to use. By evaluating aLink metaobjectForObject: anObject we fetch the correct metaobject out of the dictionary hold in aLink.

To read the metaobject from the instance variable we write:

```
anObject instVarNamed: 'metaobject_', aLink idString
```

This last line of code is directly written into the hook to have access to the metaobject there. It is important to invoke the metaobject right in the hook to not loose any time for looking up the metaobject in the link or a global dictionary. By using class or instance variables we make sure that this lookup is very fast and can be done directly in the hook.

Active attribute

The active attribute basically determines if a link is active or not. An inactive link is not executed, no shift to the metalevel occurs. To find out if a link is active we have to reify at least the object in which the hook is executed. This object is then passed to the active which holds an activation condition expecting this object. If the activation condition evaluates to *true* then the hook is further executed and the link does the shift to the metalevel. Otherwise, the hook is not executed, no more information is reified.

It depends on the control attribute what exactly happens if a hook is inactive. For a *before* or *after* control we just skip the hook and move on with the next operation. For a *replace* we do not execute the replaced operation but instead proceed with the original operation. The following pseudo code illustrates this procedure:

```
if (link is active)
  execute hook, reify and do shift to metalevel
else
  proceed with normal operation
```

The *else* part is only required for a replace control.

Executing an inactive link still costs more than having no link installed at all, but clearly it is not as expensive as executing an active link, because no shift to the metalevel occurs and we do not reify as much information as for an active link. But if we are sure that we do not need the link anymore, we better deinstall the link entirely from the system. In chapter 7 we show that an active link consumes three times more resources than an inactive link, but ten times more than having installed no link at all.

Temporal Selection.

As mentioned in section 3.4.1 we call the mechanism of specifying an activation condition for a link *temporal selection*. But temporal selection does not only cover the temporarily deactivation of a link, but also allows us to do dynamic intra-operation selection. Because we have access to the current object in which the hook is evaluated we can further restrict the operation occurrences that we want

to reify by defining an activation condition. For instance, we define that we just want to call our metaobject when the hook is evaluated in a certain object, not in every object. This further minimizes the situations where we effectively evaluate the whole hook and reify all the required information, which would be very costly.

To define a new activation condition dedicated to a certain purpose we have to create a subclass of GPActive. In this subclass we have to implement method #evaluate: which expects as its argument the current object in which the hook is executed. This method has to answer a boolean, determining if the link should be active or not. The activation condition can be arbitrarily complex, but we can only use the current object to determine the active status of the link, nothing else.

Subactives.

Class GPActive represents some generic activation condition, named *on*, *off*. These actives are accessible by methods on the class-side of GPActive, *e.g.,* GPActive class>>#off. The actives for on and off simply do what their name suggests: They just specify that a link is always active or inactive, respectively, they do not implement a real activation condition but just answer the corresponding boolean.

The third pre-implemented active, called *sub* active, is quite special: First, it is an instance of class GPSubActive, second, GPActive class>>#sub always answers the same instance of GPSubActive. We can use this sub active to tell the system that we want to check the active on the next lower level. There are three levels in which we can define actives: Hookset level, referred to as global, class level and object level (similar to the scope attribute). We can define an active condition on every level. All these levels are checked in the order noted above, therefore we have to define a sub active for a higher level if we do not want to specify explicit actives on these higher levels.

For instance, if we want to define actives just on the object level, we define sub actives on hookset and on class level, which denotes that the sub level has to be evaluated. Because if there would be no active on hookset level GEPPETTO assumes that no activation condition was set and would not check the lower level to not loose any time. Therefore, we have to explicitly fill all levels with actives until we reach the finest grained level in which we want to specify activation conditions. But normally it should be enough to define a global active. As soon as we find an explicit active on a level, we evaluate that and answer the result of this evaluation, thus not evaluating any actives on a lower level.

Storing the Actives.

The actives are stored on the different levels similarly to scope attributes: The global active is stored in an instance variable of the link, the class active and the object active in a class and an instance variable of the given reflective class.

An Example for an Active.

A useful activation condition is already predefined in class GPScopeActive. Here we check in method #isActive if the current hook is executed from within a

given scope. Currently, a scope is just a package. We can find out if we call any method from within a given package by analyzing the call stack. If we find anywhere in this call stack a class that is defined in the given package, then we assume that we call indeed the current method from within that package. This approach to provide scope-dependent reflection is quite expensive, but it nonetheless works.

Updatable attribute

The updatable attribute denotes if the metaobjects of this link are dynamically updatable or not. It is a simple boolean, either *true* or *false*. If updatable is set to true we change at runtime the metaobjects associated with this link. Depending on the setting of the scope attribute, this means that we change the metaobject of the link, of any reflective class or of any object of a reflective class. These entities use afterwards the newly specified metaobjects and drop the old ones for any future evaluation of the hooks belonging to this link. Otherwise, if updatable is set to false, exchanging metaobjects is not permitted as soon as they are once set.

MOP Descriptor

MOP descriptors provide expressive means to describe what information should be reified and how this information is passed to the metalevel. We can exactly specify every reifiable piece of information, such as the arguments or the receiver of a message send, or the value and the name of an instance variable. Furthermore, we can specify to which metaobject and to which method of this metaobject we want to pass this information and how they should be packed: in an array, in a dedicated object which provides accessor methods to the reified information, or if we pass the different pieces of the reification just as normal parameters to the method in the metaobject, the so-called plain passing mode.

We set such a MOP descriptor with the method GPLink>> #moCall: which expects an instance of GPCallDescriptor. We describe this class in detail when covering the package Geppetto-MOP. For the moment, it is enough to know that we can specify exactly the call to the metalevel with such a call descriptor. We can specify up to four parameters for a call descriptor: The metaobject itself, the method of this metaobject to which we pass the reified information, an array for all data we need to reify, such as selector, arguments or receiver and a passing mode which denotes how this data is passed to the method of the metaobject. Only the first parameter, the metaobject, is required, the other settings are optional, if we do not specify them we reify all default information and pass that in an array to metaobject.

The metaobject selector, if not explicitly specified, is derived from the control we use. If we for instance use the after control, we anticipate a method called #after: in the specified metaobject.

For more information on call descriptors we refer to section 6.6.

6.4.2 Important Methods of GPLink

Finally, we describe some important public methods of class GPLink.

Building Hooksets in a Link.

First of all we have some convenient methods to build up a hookset from within a link. Instead of first initializing a hookset and then passing this hookset to the link we directly use the link class which then internally builds up the hookset automatically. For that purpose we provide a wide range of methods that are part of the public interface of hookset. In the following example we show their usage:

```
link := GPLink id: #myLink.
link classEnumerator: (GPPackageCE packageName: 'Geppetto').
link methodSelector: [:selector | selector = #install].
link operation: GPMsgSend.
link operationSelector: [:send | send selector = #hookset].
```

Using these hookset related methods of GPLink we deal just with a link instance during the configuration of our reflective requirements instead of also creating hookset instances ourself.

Setting Metaobjects.

Next we have a whole bunch of methods to set the metaobjects. We set the sole metaobject for the link with #setMetaobject:. This has only an effect if we use global scope. If we configured the link to use class scope, we set the metaobjects with #setMetaobject:forClass: or #setMetaobjects:forClasses:. When using object scope we have #setMetaobject:forObject: or #setMetaobjects:forObjects:.

For class and object scope we can of course set several metaobjects, one for every class or object, respectively. These objects are then stored internally in a dictionary with the class or object as a key and the corresponding metaobject as the value. During the installation phase we read these metaobjects from this dictionary to store them either in a class variable or in an instance variable. If we have an updatable link and we set a new metaobject for either a class or an object, we dynamically update the class or the instance variable, respectively, with this new metaobject.

Furthermore, we have also read methods for the different types of metaobjects, such as #metaobjectForLink, #metaobjectForClass: and #metaobjectForObject:. We provide also methods to answer every class or every object that has an associated metaobject, they are called #classesWithMetaobjects and #objectsWithMetaobjects, respectively.

Setting Actives.

Similar methods exist for setting and accessing actives. Here we have to deal also with the three levels *global*, *class* and *object* to specify activation conditions. For these three levels we provide the methods #setActive:, #setActive:forClass: and #setActive:forObject: to set one active. To set several actives for several classes and objects, respectively, we have implemented #setActives:forClasses: and #setActives:forObjects:.

Readers for actives are also available: #activeForLink to access the active for the whole link, #activeForClass: to get the active for the specified class, and finally #activeForObject: to read the active for the given object. Internally all these actives are stored in a dictionary with the class or object as the key and the active as the value, except the link active, which is stored in the instance variable *active*. During the installation of the link the actives for classes and objects are installed in a class variable or an instance variable of the reflective classes.

Testing Methods.

Several testing methods are available in GPLink, *e.g.,* #isInstalled, used to test if this link object is already installed in the system. #isUpdatable checks if we can update metaobjects of this link (see updatable attribute). With #isActiveFor: we check of this link is active for the given object, as we explained in detail above when presenting the active attribute.

Installing and Deinstalling.

To install a link we simply call method #install, to remove it later on entirely from the system we use the method #deinstall. We can also reinstall the link by calling #reinstall which checks if the link is installed at all, and if so, this method deinstalls and installs the same link again. We also provide method #installMetaobjectsForClass: which dynamically installs all metaobjects for the given class. These includes metaobjects for the class itself, thus stored in a class variable, or metaobjects for objects of this class, stored in an instance variable. If we specified object scope we have to iterate over all objects with associated metaobjects and have to store the metaobject in the appropriate instance variables of every object. The possibility to install metaobjects on demand in class and instance variables, respectively, is important when we want to dynamically set new metaobjects in new classes or objects.

6.5 Geppetto-Operation Package

The Operation package holds all the classes representing reifiable operations in GEPPETTO, such as message sending, instance variable access or message receiving. For each of these operations we have created one class for the static and the dynamic part of that particular operation. In the static part, hold on the class-side of the operation class, we define what pieces of information are reifiable for this operation and also how to achieve this reification. In the dynamic part, hold on the

instance-side, we provide accessors for the concrete reified information for a given operation occurrence. To summarize: the static part is responsible for defining the operation as such, and the dynamic part represents a concrete occurrence of such an operation.

With GEPPETTO we can reify currently four operations: Message sending, message receiving, instance variable access (read and write) and temporary variable access (read and write). For these four operations corresponding classes named GPMsgSend, GPMsgReceive, GPInstVarAccess and GPTempVarAccess exist. All these classes are directly or indirectly subclasses of GPOperation.

6.5.1 Class-side of Operation Classes

On the class-side of GPOperation we have implemented testing methods to check if the current operation is caller-side or callee-side. Only message receive is a callee-side operation, because here we are on the "surface" of an object and receive there a message send. The other three operations represent "something" in a object, something is "done" in this object, *e.g.,* a message is sent or a variable is accessed. Subclasses of GPOperation have to override #isCalleeSide to answer *true*, if they are callee-side operations, because the default points to *false*. #isCallerSide always returns the negation of #isCalleeSide.

Another important method is GPOperation class>>#convert:. We use this method if we want to convert an array filled with reified data of an operation occurrence into an object of this type of operation, *e.g.,* to conveniently access the data of an operation occurrence. If we have for instance an array with reified data of a message send, we evaluate GPMsgSend convert: anArray to obtain a message send object, an instance of GPMsgSend. We then send the message #receiver or #arguments to this object to get the corresponding information. Accessing the same data from the array is more difficult because we have to know the indices under which a piece of information is accessible in this array.

Creating Reifications.

To install the hooks performing the reification of operation occurrences in the bytecode BYTESURGEON provides different methods for every type of operation, *e.g.,* #instrumentSend: or #instrumentTempAccess:, as stated in 6.3.2. We store the information which BYTESURGEON selector to use for which operation in method #bsCode on the class-side of every operation class. For instance, in GPTempVarAccess the method #bsCode is implemented as follows:

```
GPTempVarAccess>>#bsCode
  ^#instrumentTempAccess:
```

To reify and to store information in an array GEPPETTO takes care to use always the same indices for the same information when creating this array. These indices are hold into methods in the operation classes. Every operation class provides

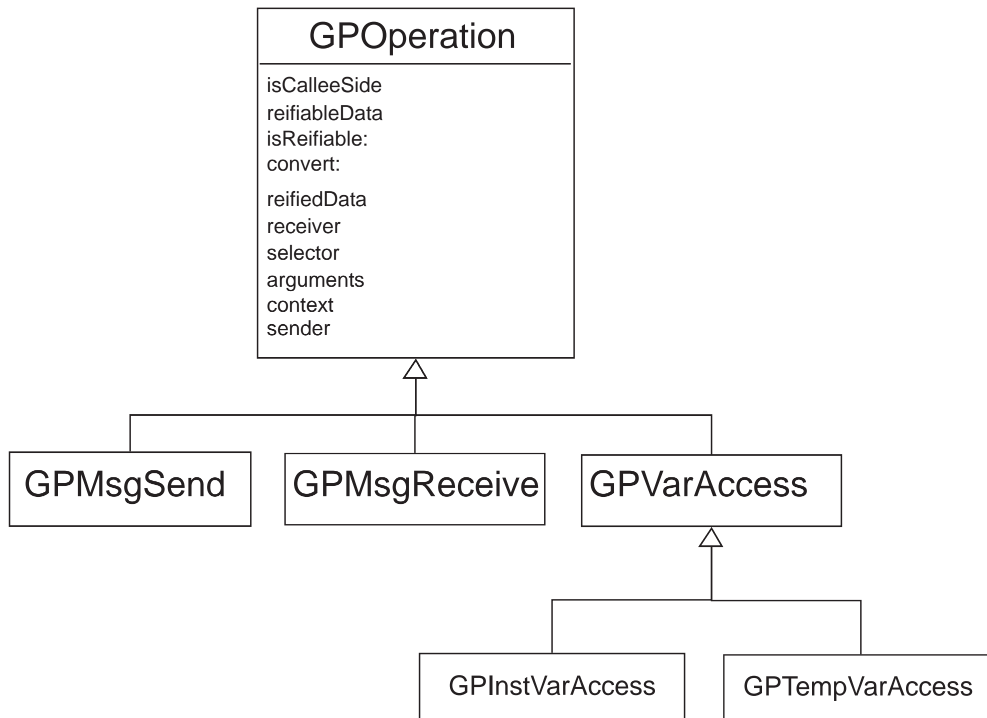

Figure 6.5: Package Geppetto-Operation

just these indices to access information that is reifiable for this kind of operation. In GPOperation we have therefore methods containing array indices for context, senderObject (the object which sent the message invoking the method where this operation occurrence is reified) and senderSelector (the selector of the method in which the message invoking the current method was sent), because this data is reifiable for all operations. In subclasses of GPOperation we extend the list of array indices accordingly to what is reifiable for this operation.

6.5.2 Instance-side of Operation Classes

On the instance-side of every operation class we have implemented accessors to the reified data for a given operation occurrence. Of course, we just provide accessors to information that is also reifiable for this type of operation. In GPTempVarAccess for instance we can access the name of the variable, its offset, the value stored in it and, for write accesses, also the new value that we want to store in this variable. We name these accessors #varName, #offset, #oldValue and #newValue. Because exactly the same information is also available for instance variable accesses, we have abstracted these methods into a common superclass of GPTempVarAccess and GPInstVarAccess, called GPVariableAccess. For GPMsgSend and GPMsgReceive, we have similar methods to access the reified data, such as #selector, #arguments or #receiver.

Directly in GPOperation we provide accessors for data that is reifiable for every operation. As already noted above when talking about the class-side of GPOperation we can reify context, senderObject and senderSelector for every operation, so we provide the corresponding accessor methods directly in GPOperation.

Internally in an operation class, all the reified data is stored in a dictionary with the array indices for the reified data as keys. If we do not reify a piece of information for a given operation occurrence, because we do not require this data in the metaobject, the corresponding method of the operation object answers *nil*.

Note that operation classes also contain setter methods for the reified data. But normally these setters are only used internally to fill the dictionary containing the data and should therefore not be called from outside an operation class.

6.6 Geppetto-MOP Package

In this section we describe all classes used to define how to do the shift from the base level to the metalevel. These classes are part of the metaobject protocol (MOP) of GEPPETTO, called *standard MOP*. They provide means to customize and adapt this standard MOP to achieve a specific goal, *e.g.,* to reify selector and receiver of a message send and to pass this information in an array to a metaobject.

Basically, we can customize the MOP in two ways: First, we can define what pieces of information we exactly require from the base level and thus what infor-

mation has to be reified at runtime. Second, we can determine which metaobject we want to call on the metalevel and how the reified data is passed to this metaobject. We specify both possibilities using a special entity implemented for that purpose, a so-called *MOP descriptor*.

6.6.1 Class GPCallDescriptor

We have implemented a simple MOP descriptor in class GPCallDescriptor. With this entity we define basically how the call to the metaobject has to take place. We also specify what information we want to reify with this call descriptor, using parameter objects.

GPCallDescriptor expects four arguments to work correctly: The metaobject, the name of the method we want to call on this metaobject, information about the data reified in the hook belonging to the link, and the passing mode which specifies how the reified data is passed to the method of the metaobject. We create a call descriptor by using one of the constructor methods on the class-side of GPCallDescriptor and pass these four arguments to these constructor methods. Only the specification of the metaobject is really required, we can omit to specify the three other arguments, the call descriptor relies on default settings for the omitted arguments.

We now explain these four arguments expected by the call descriptor in detail.

Metaobject

We either set the metaobject with the constructor methods provided on the class-side of GPCallDescriptor or use the #metaobject: method on the instance-side. All these methods expect either a class or an object for the metaobject. If we pass a class, GPCallDescriptor automatically instantiates this class using its #new method and uses then the created instance as the metaobject. If we pass an object instead of a class this object becomes the metaobject.

If we want to use class or object scope for the metaobject, we have to directly set the desired metaobjects with the appropriate methods provided in GPLink which we described in the section on the link package in 6.4. To update the metaobjects dynamically when the link is installed we have to use the methods provided by the link, not the call descriptor. The MOP descriptor is only used to statically describe what has to be reified and how this data has to be passed to the metaobject. The metaobject itself can dynamically change, but all the other parameters, *e.g.,* selector of the method on the metalevel or the passing mode of the reified data, are not changeable as soon as we have installed the link.

Note that we must pass a metaobject when creating a call descriptor, because otherwise it would be possible to install a link without an associated entity in the metalevel which would surely result in an error at runtime. All the other parameters we explain shortly are optional.

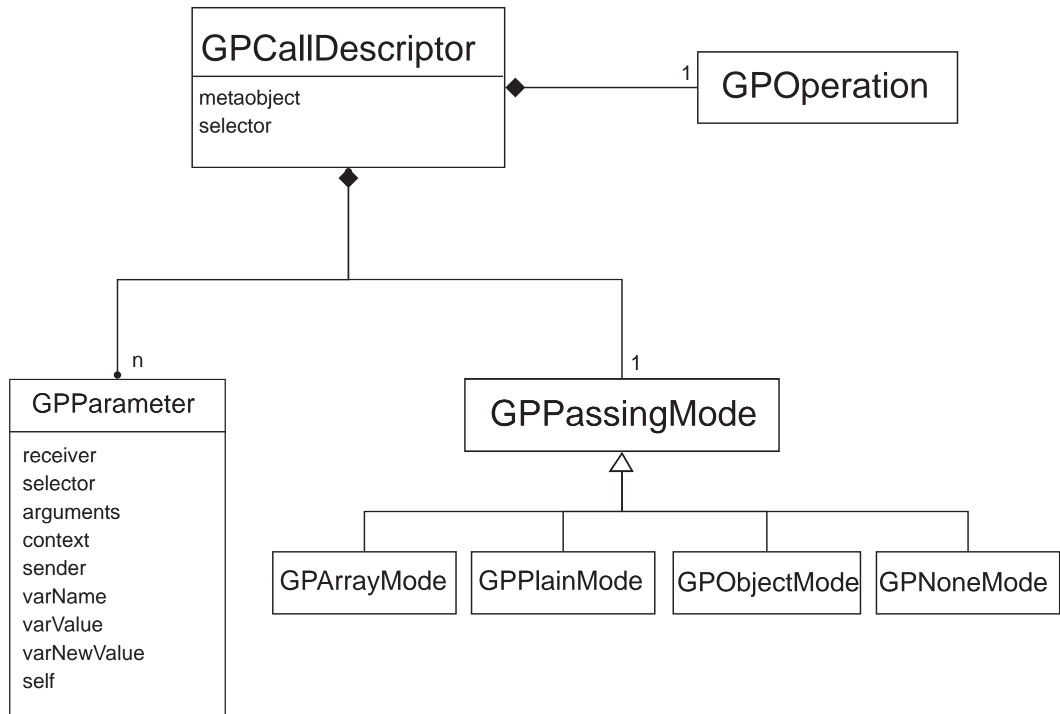

Figure 6.6: Package Geppetto-MOP

Metaobject selector

We also have to know the name of the method to call on the metaobject in the hook, referred to as metaobject selector. We simply pass this selector to one of the constructor methods of GPCallDescriptor, or use the instance-side method #moSelector: to set it. Of course, a method with this selector has to exist in the class of the metaobject.

If we do not specify explicitly such a metaobject selector, GEPPETTO assumes the presence of a default selector in the metaobject. The name of this default selector is dependent on the control attribute we specified in the link. If we use a before control the default selector is called #before: and expects one parameter which is the array with the reified data in it. The other default selectors are #after: and #replace:. If we use a *Before_After* control GEPPETTO assumes that we provide a #before: as well as an #after: method in our metaobject.

Note that as soon as we rely on these default metaobject selectors we only use the array passing mode, also if we have explicitly set any other passing mode in the call descriptor.

Parameters

Next we have to define the so-called parameters. One parameter represents one piece of information we want to reify. If we are, for instance, interested in the reification of the selector, the arguments and the receiver of a message send, we have to define three instances of GPParameter.

On the class-side of GPParameter we find selectors for every reifiable information GEPPETTO currently supports. To reify the data mentioned above we have to create these three parameter objects: GPParameter selector, GPParameter arguments and GPParameter receiver. We pack these three parameters into an array and pass it to a constructor method or to the instance-side method #parameters: of the call descriptor.

If we do not define explicitly one or more parameters GEPPETTO reifies all reifiable data of the selected operation occurrence.

We present shortly the GPParameter class in detail and report why this class is important.

Passing Mode

By setting a passing mode to the call descriptor we instruct GEPPETTO to pass the reified data in a certain manner to the metaobject. We can choose between four passing modes: Array mode, object mode, plain mode and the so-called none mode.

Array Mode.

Array mode is the default passing mode, here we pack all the reified data in one array and pass that array to the specified method of the metaobject. Every reifiable

piece of information has always the same index in this array, so it is possible to unambiguously identify the information in the metalevel. These indices are stored on the class-side of the operation classes.

Object Mode.

When using object mode we do not create an array but an object holding the reified information. This object is an instance of the corresponding operation class. If we *e.g.,* have reified the operation message receive we create an instance of GPMsgReceive in the hook, if using object passing mode. We can easily access all the reified information of a message receive by sending messages to this instance of GPMsgReceive, because the operation classes provide accessors methods with intention revealing names, such as #selector, #arguments or #receiver. Having access to such an object instead of just an array in the metaobject simplifies the retrieving of reified information, but the creation of this object costs more than just using an array. The costs for creating an object instead of an array are between ten and twenty percent higher.

Note that you can also turn an array into such an operation object in the metalevel by using the method GPOperation class>>#convert: which expects the array as a parameter and answers the operation object filled with the reified data of the array.

Plain Mode.

The plain passing mode is useful in situations where we want to reify just two or three pieces of information, because here we pass every information as an argument to the method in the metaobject. Therefore, we have to provide a method that expects exactly as many arguments as we have specified parameters in the call descriptor. The reified data is then passed to the metaobject in the same order as we specified these parameters.

In the following we define a call descriptor using plain mode:

```
callDescriptor := GPCallDescriptor object: mo
            selector: #afterSendToSelector:receiver:arguments:
            parameters: {GPParameter selector.
                     GPParameter receiver.
                     GPParameter arguments}
            passingMode: GPPassingMode plain.
```

This call descriptor uses the object *mo* as the metaobject. In the hook we then send the message #afterSendToSelector:receiver:arguments: to *mo*. This send has three arguments and GEPPETTO uses the reified data specified with the parameters in the call descriptor as values for these arguments, in the order we defined them in the parameters array. This means that the first argument is the selector of the reified operation, here of a message send, the second argument the receiver,

the third the arguments of that message send. This way we can easily access the reified information in the metaobject method, hence the plain passing mode is very efficient and easy to use in cases where we want to know just a few information about an operation occurrence.

None Mode.

Finally we use the none mode in cases where we do not reify anything. The none mode means that we just execute a method in the metalevel without passing any information to that method and thus not reifying anything in the hooks. This is useful if we for example want to notify the metalevel that a certain operation has been executed in the base level or if we can fulfil our task in the metalevel without requiring information from the base level.

We analyze the internals of the passing mode classes in section 6.6.3.

Installation

When we have defined these four settings of the call descriptor, metaobject, metaobject selector, parameters and passing mode, we set it with the #moCall: method: link moCall: aCallDescriptor. During the installation of the link this call descriptor is asked to make sure that all the specified information provided by the parameters list is reified in the hook and passed to the metaobject with the desired passing mode. The hook installer directly asks the link for the information given in the link's call descriptor, such as the parameter list or the passing mode.

Besides the call descriptor class, we have some other classes in the MOP package, such as GPParameter and several classes representing passing modes which we explain now in detail.

6.6.2 Class GPParameter

GPParameter is responsible to hold all knowledge used to represent a concrete piece of reifiable information, such as the arguments of a message send. A parameter knows how to reify this information using BYTESURGEON. For every reifiable information we have to use either a meta variable or a pseudo-variable in the string we pass to BYTESURGEON. Such a meta-variable is for instance <meta: #*receiver*> or <meta: #*arguments*>, the pseudo-variables are the three standard pseudo-variables available in Smalltalk: self, super and thisContext. Therefore, we have to use different codes to reify different information, and every parameter knows this code, accessible with the method GPParameter>>#bsCode. To reify the receiver of a message send, #bsCode answers 'receiver', this is the code inserted into the meta-variable to access the receiver object.

We also specify if a parameter is static, thus representing a pseudo-variable. For that purpose we provide two methods called #beStatic and #isStatic to set or read the *static* property.

Important is the method GPParameter>>#evaluate. The hook installer calls this method to find out what is exactly the string it has to pass to BYTESURGEON to reify a piece of information. #evaluate constructs this string by using the information of this parameter stored in #bsCode and the *static* attribute. The parameter representing the receiver of a message send for example returns the meta-variable denoting the receiver in the bytecode, this is <meta: #*receiver*>.

As we mentioned above when talking about Array passing mode every type of reifiable information has a fixed index in that reification array. We store this index also in the parameter to simplify the creation process of this array in the hooks. To set and get this array index we have the methods #index and #index:, respectively.

On the class-side of GPParameter we have implemented constructor methods for every reifiable type of information, such as #senderObject, #receiver or #varName. We can also customize the code and index by using the constructor method #code:index: where we explicitly specify the code we use in BYTESURGEON and the array index of that parameter, but in most cases we are fine by using the pre-implemented parameter constructor methods.

6.6.3 Passing Mode Classes

We have created a dedicated class for every available passing mode, called GPArrayMode, GPObjectMode, GPPlainMode, GPNoneMode, and an abstract base class named GPPassingMode. In this super class we find constructor methods for every of the four passing modes on the class-side. These methods always answer the same instance of every of the four passing mode classes.

On the instance-side of a passing mode class we just find the method #reificationCodeFor: which is defined abstract in GPPassingMode but is overridden in every subclass. This method expects a link as its sole argument and is responsible for building the whole code achieving the call to the metaobject. This includes also the code for the reification of all specified parameters.

In #reificationCodeFor: we iterate over all these parameters and evaluate them to gain the code required by BYTESURGEON to reify a certain kind of data. During this iteration we build up the whole string which the hookset installer then passes to BYTESURGEON. Because we have very different reification codes to build for every type of passing mode we provide an independent implementation of the #reificationCodeFor: method in every passing mode subclass. We can therefore polymorphically ask the passing mode attribute to give us the reification code during the installation of the hooks in the hook installer.

During the reification we also create the array or the object, if using either array or object passing mode. This means that we have to specify the creation of the array or object directly in the #reificationCodeFor: by building up a string which consists out of standard Smalltalk code.

In the case of Array passing mode #reificationCodeFor: creates an array using the #with: class methods of the Array class. The reification of, for instance, name, value and offset of an instance variable access is denoted by the following string,

which is created in #reificationCodeFor:. This string begins with the metaobject selector, followed by the code creating the array, including the reification of the parameters.

```
' moSelector: (Array with: <meta: #varname> with: <meta: #value>
with: <meta: #offset>)'.
```

The hook installer inserts this string directly after the string representing the metaobject. Both strings together represent the entire shift to the metalevel: The metaobject is the receiver, #moSelector: the selector and the reification array the arguments of the message send to the metalevel.

Besides this #reificationCodeFor: method we have not implemented any other behavior in these passing modes classes.

6.7 Geppetto-Library Package

The libraries implemented in the Geppetto-Library package are useful to simplify the configuration of hooksets and links. Because we can set many parameters to define our requirements for spatial selection or the shift to the metalevel, it gets quite complex to know the syntax and spelling used to specify all these parameters. By using the library classes instead of directly talk to hookset and link objects we obtain much cleaner, shorter and simpler code to define all the necessary hooksets and links for a given problem.

For that purpose we have created several configuration classes. The base class of all these configuration classes is called GPConfiguration. We have implemented one subclass for the configuration of every reifiable operation, *i.e.,* message send, message receive, instance variable access and temporary variable access. We named these classes GPMsgSendConfiguration, GPMsgReceiveConfiguration, GPInstVarConfiguration and GPTempVarConfiguration.

The reason why we provide different configuration classes for every type of reifiable operation is simple: We have many differences to consider when configuring hooksets and links for an operation or another. For instance, we do not have to define operation selectors for callee-side operations. By providing dedicated configuration classes for every type of operation we can abstract much of this complexity in every configuration class and therefore hide it from the user.

6.7.1 Usage

We now describe how the configuration classes are used. On the class-side of GPConfiguration we have implemented many methods to set up the desired configuration. Some methods expect just a few parameters, whereas others have a huge interface to specify many parts of the entire possible configuration. Basically, we

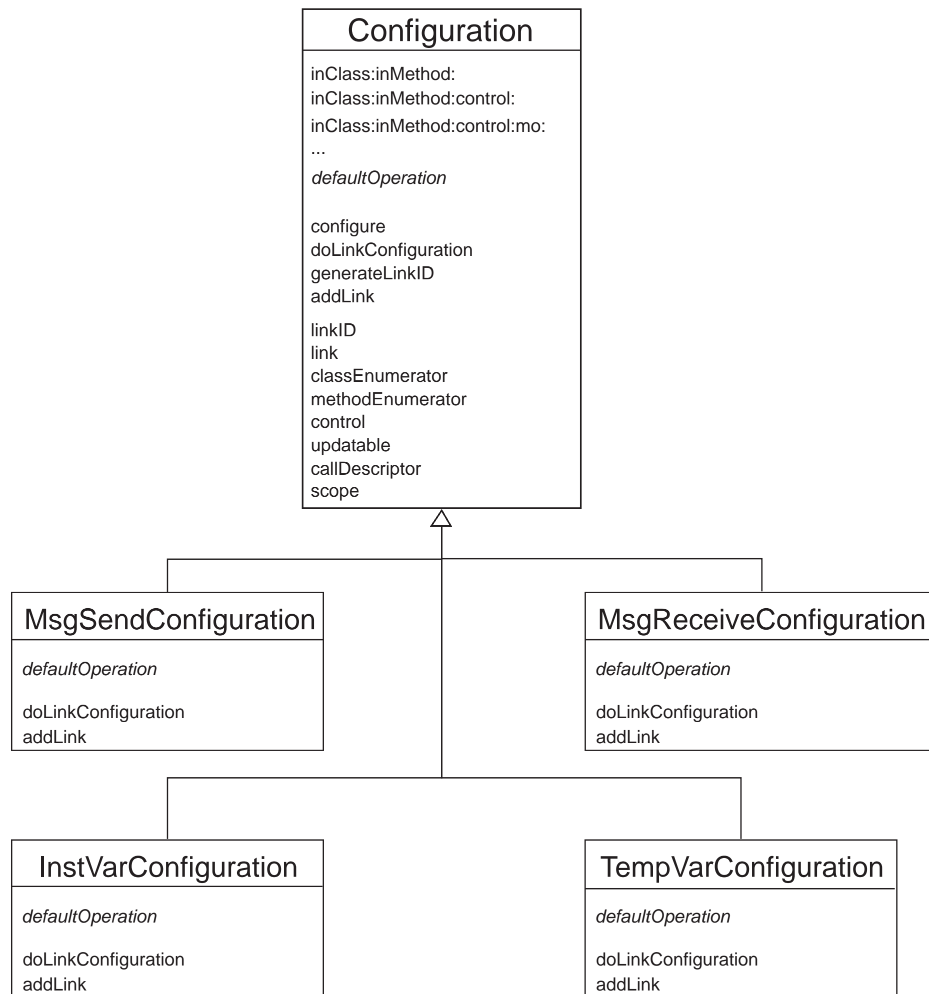

Figure 6.7: Package Geppetto-Library

should have the possibility to affect every parameter that we can also specify when using directly the hookset and link class.

Parameters not specified explicitly are automatically set to some default values. Currently, we use these default values: Replace as the control attribute for the link, global scope, passing mode array and a non-updatable link. If we do not specify any information about what data we want to reify, all the reifiable data for a given operation is reified. If no explicit metaobject is set, the class in which we issued the configuration is used to create new metaobjects. We just send the method #new to this class to get our metaobject.

Thanks to these default values we have to define only one argument in a minimal variant of a configuration: The class in which we want to have reflection, using GPConfiguration>>#class:. In that case we reify every occurrence of the selected operation and pass the information in an array to the metaobject which is an instance of that class which holds the configuration code.

We select the operation to be reified by using one of the dedicated configuration classes, *e.g.,* GPMsgSendConfiguration automatically selects message send operations.

Of course, such a minimal configuration where we specify only the reflective class and the operation explicitly is not sufficient for most purposes, therefore we provide many other configuration methods such as #inClass:inMethod:control:mo: parameters:. Here we can specify the reflective class and method, the control attribute, the metaobject, and also all the information we want to reify.

An even more complex configuration method is #inClass:inMethod:operationSelector:control:mo: parameters:passingMode:updatable: where we can also influence the operation selector, the passing mode and the updatable attribute of the link. Other methods allow us to set different attributes. For almost every possible and reasonable combination of the different attribute we have created such a configuration method.

After specifying our reflective requirements using these configuration methods the configuration class internally creates the appropriate hookset and link. Afterwards this links is installed into the system transparently to the user. The ID of this link is created automatically and is named #generatedId*n*, where *n* is the first number, starting with 1, that is not already in use for a *linkID*. So the links built with these configuration classes have *linkIDs* following this pattern: #generatedId1, #generatedId2, #generatedId3, ...

By sending the method #linkID to the configuration instance we get the symbol denoting the *linkID*, by sending #link we directly access the link object. Further methods of the configuration classes are among others #control, #classEnumerator, #methodEnumerator or #callDescriptor.

The creation and installation of the hookset and link is done in the methods #doLinkConfiguration, where we create and configure the link, and in #addLink, where we add the link to the global link repository and install it into the system. These methods are overridden in subclasses of GPConfiguration to adapt the creation and installation procedure accordingly to the requirements of the operation

that is configured with the given subclass.

6.8 Other Packages

GEPPETTO has some other, less important packages that we do not cover in detail, because they contain additional functionality not belonging to the core of the GEPPETTO framework, such as examples or tests.

For instance we have the package Geppetto-Tests which includes several TestCases to test different parts of GEPPETTO. We have tests for the core classes, tests for every reifiable operation, dedicated tests for MOP descriptors, for several examples such as subjective scope or persistence.

Another interesting package is Geppetto-Examples-Metaobject where we provide implementations of several metaobjects, *e.g.,* a metaobject for persistence, for an extension of the standard Array class, for multiple dispatch or subjective scope, for caching results of a Fibonacci calculation, and so on.

These classes are very interesting for people learning how to use GEPPETTO, because they give a comprehensive introduction into the possibilities and the usage of GEPPETTO.

Furthermore, we have the package Geppetto-Benchmarks in which we collected several benchmarks for GEPPETTO, showing *e.g.,* the differences between activated or deactivated links or presenting a comparison between GEPPETTO and MetaclassTalk [BOUR 00].

6.9 Summary

In this chapter we gave a comprehensive and complete description of the internals of GEPPETTO that are interesting for a user of our framework as well as for a person who wants to adapt or extend the GEPPETTO framework to fulfill specific needs.

This implementation chapter presented the important functionality of every package of the GEPPETTO framework and showed how the different classes in these packages interact and collaborate. We also described the internal behavior of crucial classes such as hookset, hook installer, link or operation.

The next chapter is dedicated to report on the efficiency of GEPPETTO by providing several benchmarks with which we measure how fast GEPPETTO is to install hooks, to reify information in these hooks and to execute behavior on the metalevel. With these benchmarks we get a clear feeling if we can apply reflection with GEPPETTO to solve a problem, or if the use of reflection would be too expensive in a given situation.

Chapter 7

Benchmarks

In this chapter we report on the efficiency of GEPPETTO by presenting several benchmarks. Basically we have to benchmark two different types of mechanisms: First, we are wondering how fast we can install and deinstall hooks and links into a system. Then second, we want to know how fast it is to actually execute these hooks and links at runtime. Because the execution performance of an application is more important than a fast installation of hooks into this application, which happens only once, we turned our attention mainly to measure this execution performance.

We have also optimized GEPPETTO to accelerate the hook code to a great extent. For instance, we have moved as much behavior as possible directly into the hooks instead of implementing this functionality in the link, because it is always faster to execute behavior placed directly into the bytecode than on a higher level in the link or even in the metaobject. An example of such an optimzation is the call to the metaobject or the creation of arrays used to pass information to the metalevel which is all done directly in the hooks and not in the link.

First, we present benchmarks for the installation and deinstallation of hooks and links, and second for the execution of a hook and of a metaobject. Then we also show the difference between an activated and deactivated link and present the effect on execution performance when we deinstall a link entirely from the system compared to just deactivating the link.

At last we compare the slowdowns we get when using reflective capabilities provided by GEPPETTO with the slowdowns other proposals experience, such as Reflex [TANT 03], Iguana/J [REDM 02] or MetaclassTalk [BOUR 00].

7.1 Installation performance

The first benchmark for the installation performance demonstrates how much time it takes to place a hook in a single method which contains just one message send. The second benchmark measures the time required for installing a *before* hook for every temporary variable access in class Array.

First benchmark: Message send reification.

We have implemented a method #simpleMethod in class Bench which contains only a few lines of code and just one message send:

```
Bench>>#simpleMethod
   | counter |
   counter := 0.
   self doAction.
   counter := counter + 1.
   ^self
```

We now install one hundred times a hook just after the message send to #doAction and remove it right after every installation. This hook reifies the selector, the receiver and the arguments of that message send.

In the next listing we see the configuration of the hookset and the link. As we define with the operation selector we are just interested in message send with a selector called #doAction. This operation selector has to be evaluated for every instruction in Bench>>#simpleMethod which contains 14 instructions.

```
link := GPLink id: #installObject.
link activation: GPActivation enabledStartOn.
link inClass: Bench.
link inMethod: #simpleMethod.
link operation: GPMsgSend.
link operationSelector: [:send| send isSend and: [send selector=#doAction]].
link control: GPControl after.
link moCall: (GPCallDescriptor
               object: self
               selector: #mo:
               parameters: {GPParameter selector.
                            GPParameter receiver.
                            GPParameter arguments}
               passingMode: GPPassingMode array).
```

The benchmark itself is quite simple. We repeat the installation and deinstallation of the link one hundred times and measure the elapsed time in milliseconds.

```
[100 timesRepeat: [link install. link deinstall]] timeToRun
```

In average we get a value of 1430 ms to install and deinstall the link one hundred times. [1]

[1] We did all the benchmarks in this chapter on a Windows PC with an Intel Pentium 4 CPU 3.4 GHz and 3 GB RAM, in Squeak 3.9

We cannot just measure the repeated installation of this link isolated, because as soon as we install a hook in a method we change its bytecode, so further installations of the same link do not encounter the same method. Therefore we deinstall the link right after its installation before we install it again. This ensures that every installation of the link encounters the method in the same state. To deinstall a link we basically just recompile all the methods in which we installed this link. Therefore, we can estimate that the deinstallation of the link just takes as much time as the recompilation of the method Bench>>#simpleMethod which we measure with the following code:

```
[100 timesRepeat: [Bench recompile: #simpleMethod]] timeToRun
```

Recompiling Bench>>#simpleMethod lasts around 500 ms, so the pure installation of this link one hundred times takes around 930 ms, less than a second. This is quite fast when we imagine that the installation of links in a system is a task we perform very rarely.

Second benchmark: Instrumenting class Array.

In the next benchmark measuring the installation performance we install hooks in class Array for every access to a temporary variable. There are 242 accesses to temporary variables occurring in Array. In total, Array has 27 methods containing totally 730 instructions.

To configure the link we use the following code. We specify that every temporary variable access is reified and that we want to know the offset of the temporary variable and the value it contains.

```
link := GPLink id: #tempVars.
link activation: GPActivation enabledStartOn.
link inClass: Array.
link operation: GPTempVarAccess.
link operationSelector: [:instr | true].
link control: GPControl before.
link moCall: (GPCallDescriptor
            object: self
            selector: #tempVar:
            parameters: {GPParameter varOffset. GPParameter varValue}
            passingMode: GPPassingMode array).
```

We benchmark the installation of all required hooks in Array with this line of code:

```
[link install] timeToRun
```

We get a value of 1758 ms to install all the 242 hooks before every temporary variable access in Array.

We conclude from these two measurements for the installation performance of hooksets and links that we can install even a lot of hooks pretty fast into methods. It takes some time to install them, but because we do not frequently install new hooks and links into a system this performance is acceptable.

7.2 Execution performance

More important than a fast installation of hooks is a fast execution of them, because we do not want to suffer from a drastic slowdown of our application when we use reflective features in it. Of course, as we stated several times in the previous chapters, applying reflection is very time-consuming, but we want to make it as fast as possible by optimizing the way we implement our hooks. We optimized the code of the hooks we insert into applications as much as possible, *e.g.,* by storing and invoking the metaobject directly in the hook, by reifying only the required information or by embedding runtime checks for activation conditions directly into the hook code.

We now present some benchmarks which prove that we managed to come up with a fast implementation of unanticipated partial behavioral reflection. We use an example where we reify a message send in a single, small method with just a few lines of code. We install one hook into this method and execute this hook afterwards one million times and measure the time this execution takes. Next we quantify how fast the execution of this hook is if the link is deactivated. Finally, we deinstall the link entirely and measure the execution time again to prove that the link really disappeared and no hook is executed anymore.

We install our hook in the small method Bench>>#simpleMethod we already used for the installation performance benchmark above. This method contains just one single message send to method #doAction for which we install an *after* hook. We also reuse the same link configuration as for the installation benchmark, so we reify the selector, the receiver and the arguments of this message send with selector #doAction when we execute the hook in Bench>>#simpleMethod. We pass this data in an array to the method #mo: of our metaobject, so the hook has to create internally the array for the reified data. In the metaobject we just access the information from the array, as stated below.

```
mo: anArray
   | selector receiver arguments |

   selector := anArray first.
   receiver := anArray second.
   arguments := anArray third.
```

	time (ms)	factor
link active	3646	10.85
link deactivated	1439	4.28
link deinstalled	337	1.00
no link	336	1

Figure 7.1: Execution performance for different link states.

To perform the benchmark itself we use the following code. We execute one million times the method Bench>>#simpleMethod containing our hook and thus also calling one million times our metaobject and passing the reified data to it.

```
link install.
bench := Bench new.

[1000000 timesRepeat: [bench simpleMethod]] timeToRun.
```

We reuse the same code to also measure how long it takes to execute the same method when this link is deactivated and when the link is deinstalled.

We can deactivate a link by evaluating the following code:

```
link deactivate.
```

Finally, we compare the results we get with the required time to execute Bench>>#simpleMethod before we installed any hooks into this methods. Table 7.1 contains the results.

These numbers present well that the removal of the link was indeed successful, there is almost no difference in the measured time before we introduced a link into this method and after the deinstallation of the link. Compared to the time elapsed to execute a hook for a deactivated link it is much more efficient to totally remove the link instead of just deactivating it, because the price we pay for a deactivated link is more than four times higher than for a deinstalled link.

It is clear that the execution of an active link takes the most time. In this case we actually execute the entire hook and thus reify all the defined information and even create the array for this information, and finally call the metaobject. The difference to the situation where we have no hook installed at all is huge (factor 11). But nonetheless we have to be aware that calling a method one million times is a lot and we still spend just three seconds for this, even when the link is active.

Benchmarking other systems offering behavioral reflection reveals that they get slowdowns often worse than those of GEPPETTO. We collected typical slowdowns for message invocation reification in Figure 7.2 for the reflective systems Iguana/J [REDM 02], Guarana [OLIV 99], PROSE [POPO 01] and MetaclassTalk [BOUR 00].

System	slowdown factor
Geppetto	10.85
Iguana/J	24
Guarana	ca. 40
PROSE	ca. 70
MetaclassTalk	ca. 20

Figure 7.2: Slowdowns of different reflective systems for the reification of message invocations.

7.3 Comparison with MetaclassTalk

MetaclassTalk [BOUR 00] is also a reflective system based on Squeak/Smalltalk. Early versions of MetaclassTalk rely on an extended virtual machine to gain the reflective possibilities and follows the concept of explicit metaclasses. Newer version can provide the same functionality without needing to adapt the virtual machine. MetaclassTalk supports the reification of message sending, message receiving and instance variables access, but do not allow the programmer to select in a very fine-grained manner which occurrences of these operations have to be actually reified. More details about MetaclassTalk are stated in section 3.3.

Because MetaclassTalk nonetheless shares several similarities to GEPPETTO we come up with a comparison between these two systems concerning their execution performance. We implemented one example for every operation both systems support, these are message sending, message receiving and instance variable access.

We first explain how we set up the metalevel in GEPPETTO, afterwards we implement a metalevel providing the same behavior with MetaclassTalk. Both reflective systems use the same class called Test as the sole base level entity of interest.

Defining the Metalevel in Geppetto

We illustrate the link definitions required to reify the three different operations separately, starting with the message send operation.

Message Sending.

First we present the example for message sending. The base level method in which we replace a message send is very simple, it contains just one message send to the Transcript:

```
simpleMethod: aString
  Transcript show: aString
```

Then we use the following link configuration code for setting up the hookset and the link. We reify receiver, selector and arguments of the sole message send to the Transcript in GPTest>>#simpleMethod:. This message send is replaced by metalevel behavior, as we specify by using the replace control.

```
link := GPLink id: #sendReplace.
link operation: GPMsgSend.
link inClass: GPTest.
link inMethod: #simpleMethod:.
link operationSelector: [:i | i isSend].
link control: GPControl replace.
link moCall: (GPCallDescriptor
        object: self
        selector: #replaceMsgSendFor:selector:args:
        parameters: {GPParameter receiver.
                GPParameter selector.
                GPParameter arguments}
        passingMode: #PLAIN).
```

Message Receiving.

The second operation, message receive, is similar to reify. We reuse the method GPTest>>#simpleMethod: and replace a the invocation of this method with behavior on the metalevel.

In the following we provide the code configuring the link. We also reify the receiver of the message, the selector and the arguments.

```
link := GPLink id: #receiveReplace.
link operation: GPMsgReceive.
link inClass: GPTest.
link inMethod: #simpleMethod:.
link control: GPControl replace.
link moCall: (GPCallDescriptor
        object: self
        selector: #replaceMsgReceiveFor:selector:args:
        parameters: {GPParameter self.
                GPParameter selector.
                GPParameter arguments}
        passingMode: #PLAIN).
```

Instance Variable Access.

Finally, we set up the reification of an instance variable access. The base level method in which we reify an instance variable access just contains one single read of an inst var:

```
readField
  ^myField isNil
```

We now install a hook to replace this read of the variable myField. We reify the name of the instance variable and the object in which the access occurs by defining the following link:

```
link := GPLink id: #fieldAccess.
link inClass: GPTest.
link inMethod: #readField.
link operation: GPInstVarAccess.
link operationSelector: [:i | i isInstVarRead].
link control: GPControl replace.
link moCall: (GPCallDescriptor
            object: self
            selector: #field:of:
            parameters: {GPParameter varName.
                         GPParameter self}
            passingMode: #PLAIN).
```

In the metaobjects to which we pass the reified information we just test if the the correct information was reified, we do not implement any behavior on the metalevel.

These are all the required steps to configure the reification of these three operations in GEPPETTO. Next, we have to set up the necessary functionality to achieve the same behavior on the metalevel in MetaclassTalk.

Defining the Metalevel in MetaclassTalk

We use version 0.3beta of MetaclassTalk to install and run this benchmark. Setting up a metalevel in this version of MetaclassTalk is very easy and convenient. On every execution of an operation the MetaclassTalk interpreter calls a certain method on the class-side of that class in which the operation occurs. These methods are named according to the operation #send: selector from: sender to: receiver arguments: args superSend: superFlag originClass: originCl for message send, #receive: selector from: sender to: receiver arguments: args superSend: superFlag originClass: originCl for message receive and #atIV: ivIndex of: instance for instance variable access. We implement these method on the class-side of Test right in the same way as we implemented the corresponding methods in the metaobject in the GEPPETTO version, this means that we just test if the correct information is reified.

Note that we reify basically the same amount of information in GEPPETTO as we have available in MetaclassTalk to be fair. In MetaclassTalk we do not have the

	MetaclassTalk (ms)	GEPPETTO (ms)	Difference
message send	108	46	-57%
message receive	3	55	1733%
instance variable read	272	92	-66%

Figure 7.3: GEPPETTO compared to MetaclassTalk.

possibility to explicitly specify what data of an operation has to be reified and how to pass this information to the metalevel. Instead the system always automatically reifies the same information for every type of operation, for instance in case of an instance variables access this is the offset and the object in which this access takes place.

Comparison

After we have set up both metalevels in GEPPETTO and MetaclassTalk we execute the methods for every operation one thousand times to get a reliable benchmark. The results we get when running these %benchmarks are presented in Figure 7.3.

We learn from these results that GEPPETTO is much faster in reifying information and calling the metalevel than MetaclassTalk, at least for the reification of message sends and instance variable accesses. However, MetaclassTalk is extremely fast in reifying the message receive operation. Because it has a different architecture than GEPPETTO it does not really need to reify a message receive. Instead MetaclassTalk always executes the metalevel method before it actually calls the real method. Thus, it can execute this metalevel method as any other Smalltalk system normally executes a method, an explicit reification of the message parameters (receiver, selector, arguments, etc.) is not required. This mechanism is very different from what GEPPETTO performs to reify a message receive, where the information is actually reified from the stack which is much slower. These differences in the reification process of a message receive explain why MetaclassTalk is much faster in reifying this operation. For the other operations where MetaclassTalk also reifies information from the stack, the situation is completely different and GEPPETTO is much faster, between factor two and three.

7.4 Summary

After all these presented benchmarks we can say that the price to pay for having reflection is quite high, but not so high that we cannot afford to pay it in cases where reflection is really required. A slowdown of about factor 10 if reflection is active is remarkable, but compared to other reflective systems which are suffering from delays of at least factor 20 this result is still enjoyable.

Thanks to the advanced possibilities GEPPETTO provides to deactivate and even remove links entirely from the system we can safely apply reflective features

to running applications unanticipated, because we are sure that we can always turn our application back to its initial state. This is something that other proposals cannot support.

The results we get for the installation performance reveal that installing hooksets and links is quite fast. We could think of some optimization strategies to make it even faster, but because installing, and also deinstalling, of links from the system is something we perform very rarely, we did not yet invested much time in the optimization of the installation routines.

As opposite to MetaclassTalk GEPPETTO offers very fined-grained mechanisms to precisely select what is actually reified in the base level, therefore the programmer has facilities at hand to further minimize the price to pay for using reflection by minimizing the places where reflective features are installed and by limiting the information reified at these places.

Chapter 8

Conclusions

In this last chapter we conclude our work by summarizing the most important contributions we elaborated during our studies on this topic and by enumerating some perspectives we encounter for future work in the area of reflection and open implementations in the context of Smalltalk.

8.1 Contributions

The main contributions of this thesis are threefold:

- **Motivate the need for unanticipated reflection and define a proposal:** Already existing proposals providing behavioral reflection have several shortcomings: Either they require to halt running applications or to prepare them on start up in order to be able to insert any reflective functionality at runtime, and hence they do not support unanticipated reflection. An example for such a proposal is Reflex [TANT 03]. Other approaches do not provide an expressive means to specify the reflective requirements and are therefore often inefficient in executing the reflective functionality, such as Iguana/J [REDM 02].

  To overcome these manifold shortcomings we proposed unanticipated reflection which allows the programmer to insert reflective behavior dynamically into a system without halting it. Our approach to unanticipated reflection is expressive, flexible, efficient, and openly implemented to support the adaptation and customization of the reflective system itself.

  We analyzed in detail our different demands on a reflective system, such as portability, efficiency or applicability without requiring the source code of applications. We also presented several requirements a programming language has to fulfill in order to be able to implement a system providing unanticipated reflection for this language.

- **Usage of partial behavioral reflection as an optimization for unanticipated reflection:** We chose the model of partial behavioral reflection of Reflex [TANT 03] to optimize our approach to unanticipated reflection and to

provide expressive means for defining reflective requirements. We believe, after comparing different proposals, that the Reflex model is most powerful to fit our specific needs.

To be able to use partial behavioral reflection together with unanticipated reflection we have to adapt the Reflex model in several ways. This adaptation includes a simplification of several concepts and entities available in the original model, because Squeak is a reflective language in itself we do not need the same, often complex mechanisms to apply reflection as in Java. In Squeak we are permanently able to change and adapt classes whereas in Java this is only possible at load-time. Therefore we can for instance use an intentional and an extensional mechanism to select classes and methods in which we want to apply reflection, because all classes and methods are well-known at the time when we install our reflective features into the system.

What is more, Squeak offers mechanisms to directly access classes in the system, because a class is also just a normal object. After all we can say that all the adaptations we applied to the original variant of partial behavioral reflection lead to a simplified, cleaner and easier to understand approach. This proves that the concept of partial behavioral reflection naturally fits into the Smalltalk environment.

- **Implementation of a framework for unanticipated behavioral reflection in the context of Squeak/Smalltalk:** We developed a framework, called GEPPETTO, which supports unanticipated partial behavioral reflection in Squeak/Smalltalk. This framework is openly implemented and can be therefore extended and adapted in many ways quite easily. For instance, it is possible to change and customize the metaobject protocol (MOP) to a great extent to be able to exactly specify what we have to reify and how we want to pass this information to the metalevel.

  Another example of the openness of GEPPETTO is the possibility to support other mechanisms to introduce hooks into a system. The standard hook installer class which uses BYTESURGEON can be exchanged with another installer class supporting a different bytecode manipulation tool or even a tool operating on another abstraction level, for instance at the abstract syntax tree (AST) level.

  The GEPPETTO framework also supports expressive means to precisely select where and when reflection should be active in a system. By defining hooksets we can specify where we want to have reflection, by specifying activation conditions for links we say when this reflection is active, for instance when the application is in a certain state.

  The possibilities to select the desired reflective entities, classes and methods, are also elaborated: We can select these entities extensional by enumerating them, and intensional by defining a selector which iterates over all entities

(either all classes in a system or all methods of a given class) to choose the entities matching the given condition.

We proved the usefulness and the simplicity of GEPPETTO by presenting several realistic examples where this framework shows its effectiveness and efficiency.

8.2 Perspectives

The presented ideas and concepts of unanticipated partial behavioral reflection and its implementation open a wide ranged of perspectives for future work. In this section we briefly mention those perspectives.

Structural Reflection.

So far our model is dedicated to just support behavioral reflection. It is interesting to see if we can elaborate and extend this model to also support structural reflection and to therefore complete it to provide means for both kinds of reflection. The resulting model and its implementation would be a comprehensive approach to unanticipated reflection which allows us to fulfill changes of the behavior and of the structure of an application or the whole system with the same concepts, interfaces and expressions.

AOP.

Another demanding area for future research is aspect-oriented programming (AOP). Reflection and AOP share many similarities in concepts, possibilities and applied techniques, therefore it is worth to study how we can evolve our proposal to also support the advanced capabilities of an AOP system such as AspectJ [KICZ 01] for Java. Reflex for instance evolved recently into a versatile kernel for aspect-oriented programming which merges reflection and AOP tightly together in one and the same system [TANT 05b].

Bytecode or AST.

Currently, GEPPETTO uses bytecode manipulation techniques to dynamically introduce hooks into methods. These hooks reify information when they are evaluated and pass this information over the associated link to the metalevel. However, bytecode manipulation offers limited possibilities and has several shortcomings, for instance it is hard to debug manipulated bytecode or install "hooks of hooks". Therefore, we are looking for a better abstraction level in which we can define and install these hooks more efficiently and effectively. First studies prove that the abstract syntax tree (AST) of methods could indeed be the right representation level for the manipulation of methods, solving the shortcomings from which the bytecode level suffers. GEPPETTO could therefore profit from a tool working on the AST representation.

Link Composition.

GEPPETTO currently supports just a basic and simple mechanism to compose links by linearly chaining them. Future work will surely improve this unpleasant situation by providing more elaborated and expressive means to compose links, either explicitly specified by the user or implicitly and automatically solved by the GEPPETTO framework. A mechanism for link composition could be to define sequences how colliding links should be applied, or to nest the several links in a way that one link is applied in the other. In the theory many sophisticated procedures for these composition issues are already identified. Reflex for instance has recently implemented some of these procedures [TANT 04].

Bibliography

[BOBR 93] D. Bobrow, R. Gabriel, and J. White. *CLOS in Context — The Shape of the Design*. In A. Paepcke, editor, Object-Oriented Programming: the CLOS perspective, pages 29–61. MIT Press, 1993.

[BOUR 00] N. Bouraqadi. *Concern Oriented Programming using Reflection*. In Workshop on Advanced Separation of Concerns – OOSPLA 2000, 2000.

[BRAN 98] J. Brant, B. Foote, R. Johnson, and D. Roberts. *Wrappers to the Rescue*. In Proceedings ECOOP '98, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[CHIB 03] S. Chiba and M. Nishizawa. *An Easy-to-Use Toolkit for Efficient Java Bytecode Translators*. In Proceedings of GPCE'03, volume 2830 of *LNCS*, pages 364–376, 2003.

[DENK 05] M. Denker, S. Ducasse, and É. Tanter. *Runtime Bytecode Transformation for Smalltalk*. In Proceedings of ESUG Research Track 2005, pages 75–98, August 2005.

[DUCA 99] S. Ducasse. *Evaluating Message Passing Control Techniques in Smalltalk*. Journal of Object-Oriented Programming (JOOP), vol. 12, no. 6, pages 39–44, June 1999.

[DUCA 04] S. Ducasse, A. Lienhard, and L. Renggli. *Seaside — a Multiple Control Flow Web Application Framework*. In Proceedings of ESUG Research Track 2004, pages 231–257, September 2004.

[ZIMM 96] C. Zimmermann. (ed.). Advances in Object-Oriented Metalevel Architectures and Reflection. CRC Press, 1996.

[FERB 89] J. Ferber. *Computational Reflection in Class-Based Object-Oriented Languages*. In Proceedings OOPSLA '89, ACM SIGPLAN Notices, volume 24, pages 317–326, October 1989.

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

[IBRA 91] M. H. Ibrahim. *Reflection and metalevel architectures in object-oriented programming (workshop session)*. In OOPSLA/ECOOP '90: Proceedings of the European conference on Object-oriented programming addendum : systems, languages, and applications, pages 73–80, New York, NY, USA, 1991. ACM Press.

[KICZ 91] G. Kiczales, J. des Rivières, and D. G. Bobrow. The Art of the Metaobject Protocol. MIT Press, 1991.

[KICZ 92] G. Kiczales. *Towards a New Model of Abstraction in the Engineering of Software*. In Proc. of IMSA '92 Workshop on Reflection and Meta-Level Architecture, 1992.

[KICZ 97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-Oriented Programming*. In M. Aksit and S. Matsuoka, editors, Proceedings ECOOP '97, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[KICZ 01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. *An overview of AspectJ*. In Proceeding ECOOP 2001, number 2072 in LNCS, pages 327–353. Springer Verlag, 2001.

[MAES 87a] P. Maes. *Computational Reflection*. PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels Belgium, January 1987.

[MAES 87b] P. Maes. *Concepts and Experiments in Computational Reflection*. In Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, pages 147–155, December 1987.

[OLIV 99] A. Oliva and L. E. Buzato. *The Design and Implementation of Guarana*. In USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99), 1999.

[POPO 01] A. Popovici, T. Gross, and G. Alonso. *Dynamic Homogenous AOP with PROSE*. Technical report, Department of Computer Science, Federal Institute of Technology, Zurich, August 2001.

[RAO 91] R. Rao. *Implementational Reflection in Silica*. In P. America, editor, Proceedings ECOOP '91, volume 512 of *LNCS*, pages 251–267, Geneva, Switzerland, July 1991. Springer-Verlag.

[REDM 02] B. Redmond and V. Cahill. *Supporting Unanticipated Dynamic Adaptation of Application Behaviour*. In Proceedings of European Conference on Object-Oriented Programming, volume 2374, pages 205–230. Springer-Verlag, 2002.

[RIVA 96] F. Rivard. *Smalltalk : a Reflective Language*. In Proceedings of REFLECTION '96, pages 21–38, April 1996.

[SMIT 82] B. C. Smith. *Reflection and Semantics in a Procedural Language*. Research Report TR-272, MIT, Cambridge, MA, 1982.

[SMIT 84] B. Smith. *Reflection and Semantics in Lisp*. In Proceedings of POPL '84, pages 23–3, 1984.

[STEY 94] P. Steyaert. *Open Design of Object-Oriented Languages. A Foundation for Specialisable Reflective Language Frameworks*. PhD thesis, Vrije Universiteit Brussel, Belgium, 1994.

[TANT 03] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. *Partial Behavioral Reflection: Spatial and Temporal Selection of Reification*. In Proceedings of OOPSLA '03, ACM SIGPLAN Notices, pages 27–46, nov 2003.

[TANT 04] É. Tanter. *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*. PhD thesis, University of Nantes and University of Chile, nov 2004.

[TANT 05a] É. Tanter, K. Gybels, M. Denker, and A. Bergel. *Context-aware aspects*. Research Report TR/DCC-2005-12, University of Chile, 2005.

[TANT 05b] É. Tanter and J. Noyé. *A Versatile Kernel for Multi-Language AOP*. In Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005), volume 3676 of *LNCS*, Tallin, Estonia, sep 2005.

[WAND 88] M. Wand and D. Friedman. *The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower*. In P. M. North-Holland and D. Nardi, editors, Meta-level Architectures and Reflection, pages 111–134, 1988.