

Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
and
Universidad de Chile - Chile
2001



Portable semantic alterations in Java

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Marc Ségura-Devillechaise

Promotor: Prof. Théo D'Hondt (Vrije Universiteit Brussel)
Co-Promotor: Prof. Luis Mateu (Universidad de Chile)

Abstract

In non-purely object-oriented languages some basic abstractions are not treated as objects. Their semantics cannot be refined using standard object-oriented techniques like polymorphism or inheritance. Metaobject protocols are a solution. However most of them are limiting the alterations to object-oriented constructions. Abstractions that are not handled as objects by the language can therefore not be altered. In the case of the Java language, there is however an important set of instructions interpreted by the virtual machine that closely match fundamental concepts of the language. By providing a mechanism to insert bytecode instructions (e.g, a method body) directly into existing bytecode, some fundamental concepts of the language could therefore be altered, although they remain hard-wired in the virtual machine. These hook insertions can be used to reify, on the client side, fundamental object-oriented concepts like: object initialization, method calls and casts for instance. This, per se, would represent a notable step forward since until now portable bytecode instrumentation tools like Javassist do not reify these concepts, or in a limited manner. Those concepts are clearly reified only in modified virtual machines (hence sacrificing portability). The purpose of this thesis is to explore the feasibility and implement at least some of these ideas. An open reflective extension like Reflex can directly benefit from this work, since it will provide it with code transformation entities allowing an increased expressiveness of the meta-object protocol.

Acknowledgments

I am glad to take this opportunity to thank the people who helped to make this work possible.

First, this work would not have been possible without all the persons behind the EMOOSE project: it was a great year where I have learned a lot and met lots of very interesting people. I am really grateful for the opportunity I have been given to study in the EMOOSE program. I do hope the program will be reconducted over years.

I am especially grateful to the teachers of EMOOSE. I hardly believe how much they made me evolve. Among many others interesting classes, I really enjoyed the discovery of Smalltalk and Pico%.

I am very happy to have a place to tell all Emoosers I studied with: Victor hugo Arroyo, Gustavo Bobeff (with Carla and Santi), Kristof De Vos, Sofie Goderis, Peng Liang, Patricio Salinas and David Würth how much I enjoyed working with them. I am even more happy to count them as friends.

Thanks to my promotor Prof. Dr. Théo D'Hondt for taking all EMOOSE students under his wings. Also thanks to my co-promotors Dr. Luis Mateu, and Eric Tanter for suggesting a subject, and allowing me to deviate from it and finally let me work on what I wanted. I especially appreciate the freedom I was given.

Thanks to the people of the supporting institutions, the Ecole des Mines de Nantes and the Universidad de Chile for helping me whenever they could, in particular during my stay in Chile.

Especially I would like to thank the people that motivated me and convinced me to finish this work when I was depressed: Eric Tanter, Kristof De Vos, Sofie Goderis, Victor-Hugo Arroyo, the Ibañez family, my own family, Annya Romanczuk, Andres Farias, friends met in Chile: Sumiko, Fiorella, Ron Pablo, Javiera, and Angela the daily people of Access Nova: David, Thomas and Eduardo and my promotors. I am grateful for all the help and ideas provided by Eric and Kristof.

As this is the conclusion of four years spent at the Ecole des Mines, I would like to take a few lines to thanks all the people and teachers I have been working with during this four years. Although I am already lacking of space, I owe a lot to Thomas Ledoux who, more or less, makes me born to computer science, Philippe David, Olivier Lhomme, and Patrice Boizumault whose introduction classes in first year attracted me to computer science, Mario Südholt for its smart classes on data structure in second year, Romuald Debruyne and Narendra Jussien for their wonderful classes on logic and constraint programming in third year, Jacques Noyé and Christian Colin both for their classes and help to choose between GL and GSI options. I am also grateful to Gossiaux Pol, Lionel Luquin, Carl Rauch, Ludovic Klein, and Richard Dallier, F. Tellier, F. Lallier and Jean-Paul Bourgeois, Rogatien Guihard, Sophie Dubuisson and Anne-France de Saint-Laurent, Philippe Houe, Christian Prins. Not to forget, I owe a lot to all the engineer students I have been working with like Jocker, Neg, GV, Volvic, K-Zim to mention a few of them.

I am already out of space and it is clear that many many people are still missing: I do not forget them. It is difficult to think to all the knowledge and affection I have been given. I just hope that one day I will merit these gifts.

Thanks to Veronica for being there.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 9 |
| 1.1 | An copy paste mechanism at a bytecode level | 9 |
| 1.2 | Goals | 10 |
| 1.3 | Document overview | 10 |
| 2 | An introduction to class files | 11 |
| 2.1 | Classes and class files | 11 |
| 2.1.1 | Overview | 11 |
| 2.1.2 | Source code and class files: different times | 11 |
| 2.2 | Constant pool | 14 |
| 2.3 | Attributes | 14 |
| 2.4 | Methods and related attributes | 14 |
| 2.4.1 | Code attribute | 14 |
| 2.4.2 | Local variables | 17 |
| 2.4.3 | Exception | 17 |
| 2.5 | Bytecode instructions: a low level language | 17 |
| 2.6 | An example | 18 |
| 2.7 | The stack | 21 |
| 2.7.1 | Types and stack | 21 |
| 2.7.2 | Stack depth | 21 |
| 2.8 | Summary | 22 |
| 3 | Macro languages | 23 |
| 3.1 | Generalities about macros | 23 |
| 3.2 | Some properties of macro languages | 23 |
| 3.3 | Macro languages: a small survey | 24 |
| 3.4 | Summary | 25 |
| 4 | Bytecode manipulation libraries | 27 |
| 4.1 | The prototype requirements | 27 |
| 4.2 | Bytecode manipulation libraries available | 28 |
| 4.3 | BCEL overview | 29 |
| 4.4 | JikesBT overview | 29 |
| 4.5 | Joie overview | 30 |
| 4.6 | Evaluation | 31 |

| | | |
|----------|--|-----------|
| 4.7 | Summary | 31 |
| 5 | Java reflective extensions using bytecode rewriting | 33 |
| 5.1 | Reflection in general | 33 |
| 5.2 | Reflection in Java | 34 |
| 5.3 | Kava | 37 |
| 5.4 | Javassist | 38 |
| 5.5 | Summary | 40 |
| 6 | Designing the copy paste mechanism | 41 |
| 6.1 | Nature and motivations of the tool | 41 |
| 6.2 | Design goals | 43 |
| 6.3 | Reified entities | 43 |
| 6.4 | Extent of alterations | 45 |
| 6.4.1 | Possible alterations | 45 |
| 6.4.2 | Not enabled alterations | 47 |
| 6.5 | Different kind of information | 49 |
| 6.5.1 | Dynamic information | 50 |
| 6.5.2 | Static information | 50 |
| 6.5.3 | Information passed for the implemented alterations | 51 |
| 6.5.4 | Strings and arrays versus objects | 51 |
| 6.6 | An example | 53 |
| 6.7 | Reinterpreted constructs | 57 |
| 6.7.1 | Local variables | 60 |
| 6.7.2 | Return reinterpretation | 60 |
| 6.7.3 | Exceptions reinterpretation | 60 |
| 6.7.4 | Self references reinterpretation | 61 |
| 6.8 | Summary | 61 |
| 7 | Naive implementation | 63 |
| 7.1 | The package structure | 63 |
| 7.2 | From Naive API objects to Naive implementation objects | 64 |
| 7.3 | From Naive implementation objects to BCEL objects | 65 |
| 7.4 | The inliner hierarchy | 67 |
| 7.4.1 | Using the inliners | 67 |
| 7.4.2 | The inliners and NaiveMethodImpl | 67 |
| 7.4.3 | Taking care of the stack depth | 69 |
| 7.4.4 | Passing and returning information to the pasted method | 71 |
| 7.5 | Memory policy | 74 |
| 7.6 | Summary | 74 |
| 8 | Future works | 75 |
| 8.1 | Offering more alterations | 75 |
| 8.2 | Revisiting the concept | 75 |
| 8.3 | Opening the framework | 76 |
| 8.4 | Summary | 76 |

| | |
|---|------------|
| <i>CONTENTS</i> | 5 |
| 9 Conclusion | 77 |
| A Some examples | 82 |
| A.1 General shape of the examples | 82 |
| A.2 Field read alterations | 82 |
| A.3 Some alternate implementations taking a field read alterations as example | 83 |
| A.4 Field write alterations | 85 |
| A.5 Cast alterations | 87 |
| A.6 Method invocation alterations | 89 |
| A.7 Constructor alterations | 97 |
| B Exceptions between metalevel and baselevel | 101 |
| B.1 An overview of Java exceptions facilities | 101 |
| B.2 Java metaprotocols and exceptions | 102 |
| B.3 Exceptions and assumptions | 104 |
| B.3.1 The shape of the example | 104 |
| B.3.2 Javassist 1.0 | 106 |
| B.3.3 Naive | 108 |
| B.4 Conclusion | 109 |

List of Figures

| | | |
|------|---|----|
| 2.1 | Class file structure | 12 |
| 2.2 | Different times, different level of alterations. | 13 |
| 2.3 | HelloWorld class: the source code. | 18 |
| 2.4 | State of the stack during main execution | 20 |
| 4.1 | JOIE layers | 30 |
| 5.1 | an example of Kava configuration binding | 37 |
| 6.1 | A block synchronization in a method body | 48 |
| 6.2 | ModifiedClass class | 53 |
| 6.3 | Design of the example: UML class diagram | 55 |
| 6.4 | Example class | 56 |
| 6.5 | MethodInvocationAlteratorObject class | 57 |
| 6.6 | Message flow performing the alteration | 58 |
| 6.7 | MethodBodyContainer class | 59 |
| 7.1 | The inliner inheritance tree | 68 |
| A.1 | Field read alteration: ModifiedClass class | 83 |
| A.2 | Field read alteration: Example class | 84 |
| A.3 | Field read alteration: FieldReadAlteratorObject class | 85 |
| A.4 | Field read alteration: MethodBodyContainer class | 86 |
| A.5 | Field write alteration: ModifiedClass class | 87 |
| A.6 | Field write alteration: Example class | 88 |
| A.7 | Field write alteration: FieldWriteAlteratorObject class | 89 |
| A.8 | Field write alteration: MethodBodyContainer class | 90 |
| A.9 | Cast alteration: ModifiedClass class | 91 |
| A.10 | Cast alteration: MethodBodyContainer class | 91 |
| A.11 | Cast alteration: Example class | 92 |
| A.12 | Cast alteration: CastAlterator class | 93 |
| A.13 | Method invocation alteration: ModifiedClass class | 93 |
| A.14 | Method invocation alteration: MethodBodyContainer class | 94 |
| A.15 | Method invocation alteration: Example class | 95 |
| A.16 | Method invocation alteration: MethodInvocationAlterator class | 96 |
| A.17 | Self method invocation alteration: ignored method invocation | 97 |

| | |
|---|-----|
| A.18 Constructor alteration: ModifiedClass class | 97 |
| A.19 Constructor alteration: MethodBodyContainer class | 98 |
| A.20 Constructor alteration: Example class | 99 |
| A.21 Constructor alteration: ConstructorInvocationAlteratorObject class | 100 |
| B.1 Main class of the example. | 105 |
| B.2 Auxiliary class of the example. | 106 |
| B.3 Metalevel with Javassist 1.0. | 107 |
| B.4 Javassist running the example | 108 |
| B.5 Metalevel with Naive. | 109 |
| B.6 Metalevel with Naive. | 110 |
| B.7 Naive running the example | 110 |

List of Tables

| | | |
|-----|--|----|
| 2.1 | Entries in the constant pool. | 15 |
| 2.2 | Predefined attributes | 16 |
| 2.3 | HelloWorld class : the constant pool. | 19 |
| 2.4 | HelloWorld class: bytecode instructions. | 20 |
| 3.1 | Characterization of some macro languages | 26 |
| 4.1 | Comparison of bytecode manipulation libraries | 31 |
| 5.1 | comparison of metaobject protocols for Java | 36 |
| 6.1 | Extent of the possible alterations | 46 |
| 6.2 | Information delivered | 52 |
| 7.1 | User and implementor view of the structural entities | 64 |
| 7.2 | Mapping from Naive implementor view to BCEL objects | 65 |

Chapter 1

Introduction

"No matter what [language] design was agreed upon, there would be time when a given user for entirely appropriate reasons would need this or that variant of it".

[KDRB91]

The primary mechanisms to alter a language semantic are macro systems and reflection. However, in Java the first mechanism is not available and the second is slightly limited. Here, first a different, but related, alteration mechanism, will first be proposed before presenting the goals of this work. Finally the structure of this document will be presented.

1.1 An copy paste mechanism at a bytecode level

The design of a macro system seems much simpler than the design of a reflective system, especially when the interpreter can not be modified. However, while macro systems are usually designed to work on source code, the Java runtime only knows a portable compiled representation of the different classes called bytecode.

To preserve portability, such a macro system should work on bytecode instead of source code. Our goal here was to design such a very rudimentary system whose sole purpose will be to inline some code in specific places.

However specific places is a vague term. From the working level, we can infer that "places" will ultimately be bytecode instructions. But, dealing with bytecode instruction does not make sense for most Java programmers. Since bytecode instructions are used to implement language semantic, "places" could be understood as language semantic.

The code to be expanded there should also be provided through a compiled class in order to remain at a bytecode level. A natural way for the user, is to describe the code to be expanded as a method of a compiled class.

Reaching an appropriate level of abstraction is also an important objective of this thesis. The reflection community has developed a representation of the different entities coming into play in object-oriented programming. We choose to borrow this representation in order to reach an abstraction level close to source code.

All in all, the goal of this thesis is a kind of advanced copy paste mechanism at the bytecode level akin to macro relying upon the reflection representation of the entities involved.

1.2 Goals

Our goal in this thesis, is to explore the direction described in 1.1 by developing a prototype acting as a technological demonstrator of this approach's feasibility. The prototype developed has been named N.A.I.V.E, an acronym for Natural Abstractions for the Virtual machinE. Possible applications of such a working prototype could be: production of new languages by Java language alterations like experimentation with new thread models or new language paradigms like aspect orientation, production of reflexive systems, and the optimization by inlining of delegation chains.

1.3 Document overview

This document is structured in nine chapters. The second chapter presents bytecode basics that are the ground of this work. Chapter 3 to 5 are part of the state of the art: chapter 3 studies how to characterize macro systems, chapter 4 analyzes the bytecode manipulation libraries available and chapter 5 presents the different Java reflective extensions. Chapter 6 describes the design of the copy paste mechanism while chapter 7 discusses the implementation of the prototype. Chapter 8 presents some perspectives. Chapter 9 concludes.

Chapter 2

An introduction to class files

”Bytecode: an architecture neutral intermediate format designed to transport code efficiently to multiple hardware and software platforms”.

[GM96]

This chapter tries to give an introduction to the runtime representation of class in Java: class files. First a general overview will be given before describing the main data structures that composes it: the constant pool and attributes. More attention will be then be paid to the data structures involved into the representation of methods. It will be followed by a simple example to fix the ideas. Finally, a number of constraints imposed on the stack structure used by methods will be enumerated.

2.1 Classes and class files

2.1.1 Overview

A class file is the representation of a given class used by the Java interpreter, or in Java parlance, used by the virtual machine. [LY97] standardizes the format of class files: it is defined as an array of bytes, although in most cases it ends up to be a file on disk. Figure 2.1 presents the structure of the class file. First comes versioning numbers that allow to identify the version of the class file format in use as well as the versions of the represented classes. It is followed by the constant pool, a data structure whose role will be presented in 2.2. Then it contains the access rights of the class: in source code terms, it roughly correspond to the modifiers `public`, `final`, `abstract` and `abstract` applied to a class. Further these access rights allows to differentiate a class from a Java interface. After, the class file structure defines the represented class and its super class. Next, the interfaces has implemented by the class are listed. Then, comes an enumeration of the fields and methods defined by this class. Finally, the structure ends by defining attributes. Attributes will be described in 2.3.

2.1.2 Source code and class files: different times

This thesis explores the alteration of class definition to alter their behaviors. Java source code, usually given as text files, is transformed into bytecode by compilation. as shown on 2.2, this leaves two places to alter the

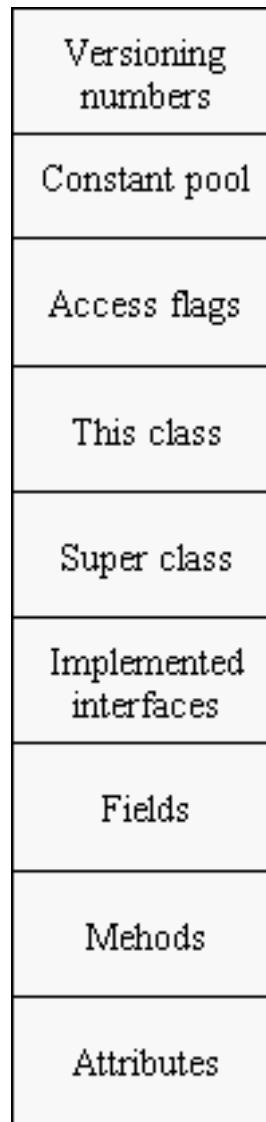


Figure 2.1: Class file structure.

This representation is based on chapter 4 of [LY97] that describes a class file with a C-like syntax.

The figure gives the impression that all attributes are stored at the end of the class file. This is only true for class attributes, the structures corresponding to fields and methods (corresponding to the Fields and Methods boxes on the figure) allocate space to store their own specific attributes.

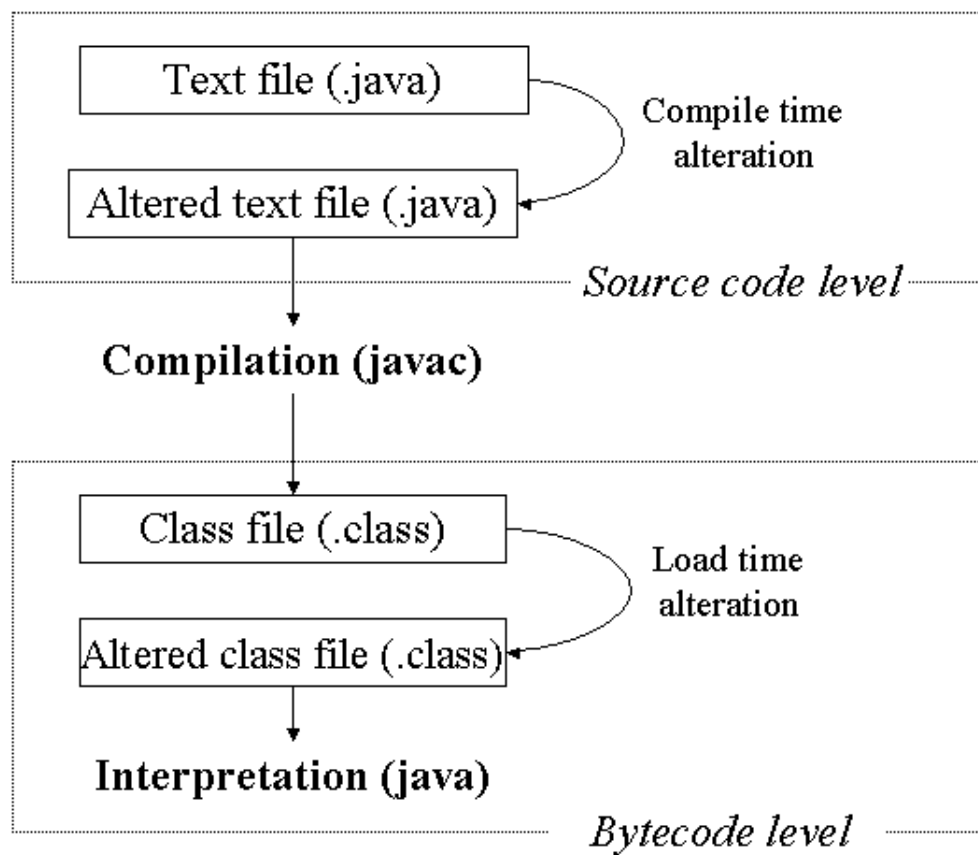


Figure 2.2: Different times, different level of alterations.

definition of a class and ultimately its behavior. If the alteration is performed, on the text files that contains the source code of the class, it is described as a *compile time* alteration. If the alteration is performed on the compiled form of the class (at the bytecode level) before the virtual machine loads it, it is said to be *load time* alteration.

2.2 Constant pool

The constant pool is a class scoped data structure that avoid any *Java virtual machine to rely on the runtime layout of classes, interfaces, class instances, or arrays* [LY97] hence ensuring portability. The constant pool is an array of entries whose types are summarized on Table 2.1. For example, the "This class" and "Super class" on figure 2.1 are defined as entries indexes in the constant pool. It is important to note that, depending on their type, constant pool entries may refer to each other.

2.3 Attributes

The data structures associated to the boxes corresponding to Fields and Methods on figure 2.1 are not directly describing more than a signature. The rest of information is stored in attributes. Attributes are a way to attach information to a whole class file, a field, a method or to another attribute itself. They are therefore contained in their related data structures: on figure 2.1, the data structures corresponding to the boxes "Fields", "Methods" and "Attributes" can contain attribute.

User specific can be defined but a virtual machine is free to ignore any user specific attribute it does not understand. [LY97] opposes to user specific attributes, predefined attributes described by the specification. These predefined attributes are summarized on table 2.2. But only a subset of this attributes (`ConstantValue`, `Code`, `Exceptions`, and `InnerClasses`) are required to be understood by all virtual machines.

2.4 Methods and related attributes

Because this thesis is interested into modifying the runtime definition of a method, this section will describe the most important attributes that can be attached to a method: `Code`, `LocalVariableTable` and `Exception`.

2.4.1 Code attribute

It is defined by [LY97] as *variable-length attribute used in the attributes table of method_info structures. A Code attribute contains the Java virtual machine instructions and auxiliary information for a single method, instance initialization method, or class or interface initialization method.* Saying it more simply, the `Code` attribute holds the method body of the method it is associated with.

Java virtual machines are stack based interpreters. The `Code` attribute defines the maximum stack depth that may be reach while executing its method body. It also defines the maximum number of local variables used,

| Entry | Representing | Referenced entry type | Referenced entry meaning |
|-----------------------------|--|--------------------------------|---|
| CONSTANT_Class | a class or an interface | CONSTANT_Utf8 | fully qualified class or interface name |
| CONSTANT_Fieldref | a field | CONSTANT_Class | fully qualified class or interface name of the class containing the field |
| CONSTANT_Methodref | a method on an object | CONSTANT_NameAndType | name and descriptor of the field |
| CONSTANT_InterfaceMethodref | a method on an interface | CONSTANT_Class | fully qualified class name of the class containing the method |
| | | CONSTANT_NameAndType | name and descriptor of the method |
| | | CONSTANT_Class | fully qualified class name of the interface containing the method |
| | | CONSTANT_NameAndType | name and descriptor of the method |
| CONSTANT_String | a chain of characters | CONSTANT_Utf8 | the character sequence |
| CONSTANT_Integer | an integer value | | |
| CONSTANT_Float | a float value | | |
| CONSTANT_Long | a long value | | |
| CONSTANT_Double | a double value | | |
| CONSTANT_NameAndType | represent a field or method without indicating which class or interface type it belongs to | CONSTANT_Utf8 CONSTANT_Utf8 | field or method name field or method descriptor |
| CONSTANT_Utf8 | a sequence of characters | | |

Table 2.1: Entries in the constant pool.

[LY97] defines descriptors as *strings representing the type of a field or method*. The encoding used to represent these strings is specified in details in [LY97].

| Name | Role | Used in data structure |
|--------------------|---|---|
| ConstantValue | represents the value of constant <code>static</code> field | representing fields |
| Code | contains the bytecode instruction information | representing methods |
| Exceptions | indicates which exceptions a method may throw | representing a method |
| InnerClasses | uses to describe inner classes | class attributes |
| Synthetic | uses to describe inner classes | class attributes representing a method representing a field |
| SourceFile | describes the file containing the Java source code | class attributes |
| LineNumberTable | maps between lines in Java source code and bytecode instructions | in Code attribute |
| LocalVariableTable | maps between the representation of local variable (a number) and the source code representation (a name) | in Code attribute |
| Deprecated | signals that this class or interface has been superseded signals that this field has been superseded signals that this method has been superseded | class attributes representing a field representing a method |

Table 2.2: Predefined attributes.

In this table, when an attribute is said to be used in the class attributes data structure, it should be understood that it is stored in the box called "Attributes" on figure 2.1.

In this table, when an attribute is said to be used in the data structure representing methods, it should be understood that it is stored in the box called "Methods" on figure 2.1.

In this table, when an attribute is said to be used in the data structure representing fields, it should be understood that it is stored in the box called "Fields" on figure 2.1.

representing each of them by a number. It defines an array containing the bytecode instruction sequence. Moreover it defines an exception table. It ends up by an array of the attributes that are associated to it.

The exception table corresponds at a source level to `try catch` statements. It is an array whose entries are composed of four elements: the first is the bytecode instruction just after the `try`, the second is the bytecode instruction right after the exception is not expected anymore (or in other words, the next instruction after the `{` closing the `try {}`), the third element is the bytecode instruction just after the `catch`, and finally the fourth is an index in the constant pool defining the exception being handled by the `try catch`.

In this subsection, we have seen that a `Code` corresponds to the runtime representation of a method body. It defines the bytecode instructions corresponding to that body, the local variables, the maximal stack depth used, and the `try catch` contained in that method.

2.4.2 Local variables

The `LocalVariableTable` is defined by [LY97] as *an optional variable-length attribute of a `Code` attribute. It may be used by debuggers to determine the value of a given local variable during the execution of a method.* For each local variable appearing in the `Code` attribute is associated with, the `LocalVariableTable` precises from and to which bytecode instruction the local variable is available (something akin to the scope at a source level) and provides an index in the constant pool defining its type.

2.4.3 Exception

The `Exceptions` is defined by [LY97] as *a variable-length attribute used in the attributes table of a `method_info` structure. The `Exceptions` attribute indicates which checked exceptions a method may throw. A checked exception is an exception subclassing `java.lang.Exception`.* This attribute roughly said represents the `throws` clause of a method declaration.

2.5 Bytecode instructions: a low level language

The bytecode instruction set currently consists of 212 instructions, 44 opcodes are marked as reserved and may be used for future extensions or intermediate optimizations within the virtual machine. The instruction set can be roughly grouped as follows:

- Stack operations: that push or pops values from the stack
- Arithmetic operations: allowing to do some basic operations on each primitive types of the virtual machine
- Control flow: branch instructions like `goto` like, `if` like instructions, and exceptions throwing. Branch targets are coded as offsets from the current byte code position
- Load and store operations for local variables and for arrays

- Field access
- Method invocation
- Object allocation: for object and array
- Conversion and type checking

Bytecode instructions may refer to the constant pool. In general, this is done by hard coding an integer corresponding to the refereed index in the constant pool in the bytecode instruction sequence in the `Code` attribute.

2.6 An example

```
public class HelloWorld {
    public static void main(String[] arguments) {
        System.out.println(arguments[0]);
    }
}
```

Figure 2.3: HelloWorld class: the source code.

In this section, a very simple example will be considered: the `HelloWorld` class whose source code is presented on figure 2.6 will be discussed from the bytecode level. This class only prints on the screen the first argument it is given to. The example is voluntary oversimplified, in particular, there is no error handling when no argument is given.

On this example, the author using the `javac` compiler coming with the standard `jdk 1.3` gets the constant pool presented in table 2.3.

With reference to the source code presented in 2.6, the compiler has generated a zero argument constructor that we will ignore for the sake of simplicity. Furthermore it has created a `SourceFile` attribute that indicates that the source file name is stored at position 13 in the constant pool.

There is no field defined in the source code, so there is no field structure (corresponding to the "Fields" box on figure 2.1) but there is one method defined: `main`. This method structure has a related `Code` attribute. This `Code` attribute specifies that the maximum stack depth while executing `main` is 3 and that no local variables are defined in it. It further contains the sequence of bytecode instructions presented in table 2.4.

The state of the stack during the execution of `main` is described on figure 2.4. An empty stack is allocated to the method before executing the first instruction. As shown in (a) on figure 2.4, the `getstatic` pushes the value of the field `System.out` on the stack. Then the `aload_0` pushes the value of the first argument: `arguments` on the stack. Next, an `iconst_0` pushes a zero on the stack. At that time, the maximal depth is reached. The

| Position | Type | Content | Refers |
|----------|----------------------|--|--------------------------------|
| 1 | CONSTANT_Methodref | void Object.<init> ()V() | 5 (class), 14 (name and type) |
| 2 | CONSTANT_Fieldref | System.out Ljava/io/PrintStream; | 15 (class), 16 (name and type) |
| 3 | CONSTANT_Methodref | void java.io.PrintStream.println (Ljava/lang/String;)V(String) | 17 (class), 18 (name and type) |
| 4 | CONSTANT_Class | HelloWorld | 19 (name) |
| 5 | CONSTANT_Class | Object | 20 (name) |
| 6 | CONSTANT_Utf8 | <init> | |
| 7 | CONSTANT_Utf8 | ()V | |
| 8 | CONSTANT_Utf8 | Code | |
| 9 | CONSTANT_Utf8 | LineNumberTable | |
| 10 | CONSTANT_Utf8 | main | |
| 11 | CONSTANT_Utf8 | ([Ljava/lang/String;) | |
| 12 | CONSTANT_Utf8 | SourceFile | |
| 13 | CONSTANT_Utf8 | HelloWorld.java | |
| 14 | CONSTANT_NameAndType | <init> ()V | 6 (name), 7 (signature) |
| 15 | CONSTANT_Class | System | 21 (name) |
| 16 | CONSTANT_NameAndType | out Ljava/io/PrintStream; | 22 (name), 23 (signature) |
| 17 | CONSTANT_Class | java.io.PrintStream | 24 (name) |
| 18 | CONSTANT_NameAndType | println (Ljava/lang/String;)V | 25 (name), 26 (index) |
| 19 | CONSTANT_Utf8 | HelloWorld | |
| 20 | CONSTANT_Utf8 | java/lang/Object | |
| 21 | CONSTANT_Utf8 | java/lang/System | |
| 22 | CONSTANT_Utf8 | out | |
| 23 | CONSTANT_Utf8 | Ljava/io/PrintStream; | |
| 24 | CONSTANT_Utf8 | java/io/PrintStream | |
| 25 | CONSTANT_Utf8 | (Ljava/lang/String;)V | |

Table 2.3: HelloWorld class : the constant pool.

The types are expressed using Strings in the internal format of the virtual machine. Without describing this format in details, constructors are renamed into <init> and the code corresponding to class initializers is stored in a method called <clinit>.

Position 4 describes the "This class" box of figure 2.1.

Position 5 describes the "Super class" box of figure 2.1.

Note that all Strings even those used by attributes are stored in the constant pool.

| Byte offset | Instructions | Argument in constant pool |
|-------------|---------------|--|
| 0 | getstatic | 15 (System.out Ljava/io/PrintStream;) |
| 3 | aload_0 | |
| 4 | iconst_0 | |
| 5 | aaload | |
| 6 | invokevirtual | 17 (java.io.PrintStream), 3 (println (Ljava/lang/String;)V(String):void) |
| 9 | return | |

Table 2.4: HelloWorld class : the bytecode instructions contained in the Code attribute of the main method.

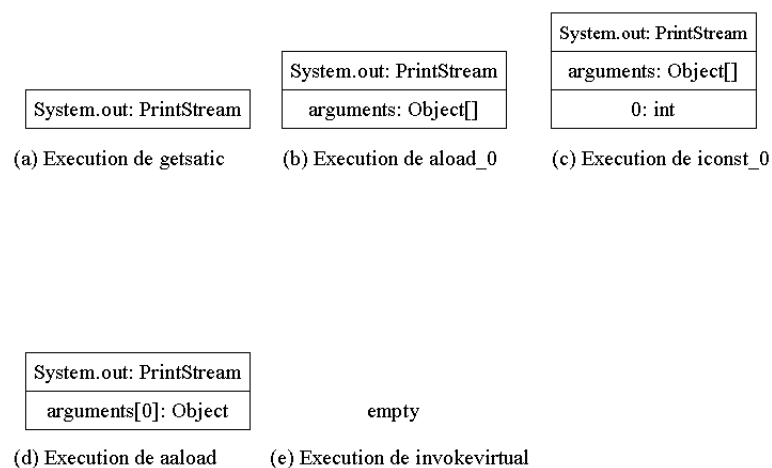


Figure 2.4: State of the stack during main execution

`aaload` instruction consumes the reference to the object array corresponding to `arguments` and the following integer to push the value contained in that array at that index on the stack. Finally the `invokevirtual` instruction invokes the `println` method. On the stack, it consumes the reference on which the method should be invoked: `System.out` and the arguments: `arguments[0]` to feed the method with. It pushes the result returned by the method on the stack: since `println` returns `void`, in that case, nothing is pushed on the stack. The `return` transfer control to the caller of `main`.

2.7 The stack

Each time method is executed, an associate stack is created to allow bytecode instructions to exchange results: they pop and push values from the stack.

2.7.1 Types and stack

While the Java virtual machine expects that nearly all type checking is done prior to runtime, the instruction set distinguishes its operand types using instructions intended to operate on values of specific types. Consequently, the stack is typed in *category 1* and *category 2*. Only `double` and `long` belongs to *category 2*. *category 1* consumes one position on the stack. *category 2* consumes two positions on the stack. Trying to manipulate two *category 1* as a *category 2* or a position on the stack corresponding to a *category 2* as a *category 1*, will cause the class file to be rejected by the class verifier.

2.7.2 Stack depth

To ensure the integrity of a class file and to prevent it from the attack of malicious classes, the virtual machine checks a number of properties of the class file. Class file not verifying these properties will not be loaded and therefore not be executed. Outside sanity checks ensuring that the interpreter will be able to interpret the class file, they are two very important constraints imposed on a class file:

- *"If an instruction can be executed along several different execution paths, the operand stack must have the same depth prior to the execution of the instruction, regardless of the path taken."*
- *"At no point during execution can the operand stack grow to a depth greater than that implied by the `max_stack` item."*

In others words, the first statements, ensures that for a given instruction the stack depth will always be the same regardless of the branch previously performed. The second statement clearly entails that the maximal stack depth of a given method has to be computed during the class file generation. Any class file generator has to meet these two constraints.

2.8 Summary

Classes are represented by class files at the runtime level. Class files are no more than an array of bytes composed of different structures. Among the most important, the constant pool that avoids any Java virtual machine to rely on the runtime layout of classes, interfaces, class instances, or arrays and attributes are a way to attach information to a whole class file, a field, a method or to another attribute itself. In essence, by comparison, the data structures corresponding to methods and fields are little more than a name and a signature. The `Code` holds runtime representation of a method body. The `LocalVariableTable` maps from the runtime representation of local variable machine to their source code representation. The `Exceptions` attribute corresponds to the `throws` clause of a method declaration. Each time a method is executed, an associated stack is created to allow bytecode instructions to exchange results: they pop and push values from the stack. `long` and `double` are occupying two positions on the stack. An attempt to split such a `long` or `double` will cause the class file to be rejected by the class verifier. For a given instruction the stack depth has to be always the same regardless of the branch previously performed. No execution of a method should consume a greater stack than declared in its code attribute.

Chapter 3

Macro languages

"From now on, a main goal in designing a language should be to plan for growth."

Guy Steele

Growing a Language, OOPSLA'98 invited talk

There exists systems that are already doing the kind of copy paste mechanism proposed in this thesis. But these systems: macro languages are modifying the source code while the central idea of this thesis is to modify the compiled code. Nevertheless, macros have been widely studied and understanding how to characterize them may be more than helpful to characterize our prototype.

Here generic notions on macro languages will first be introduced. Then a set properties allowing to characterize a macro language will be proposed. It will be followed by a short survey of well known macro languages. Our goal in this chapter is not to review all the macro languages but to identify some of their characterizing properties applicable to Naive as well. This justify that this chapter will not be focus on the Java language.

3.1 Generalities about macros

Macro are defined in the source code by specific keywords. This specific keywords can be though as a language extension. Then the source code is process by a preprocessor that reduces macro to source code of the original language. Finally the language compiler or interpreter can be used on this modified source code.

More precisely, a macro keyword is an identifier that has been associated with a macro definition. A macro definition is a user-definable transformation that tells how macro calls can be transformed into simpler forms. The result of a transformation may also contain macro calls, and these will also be transformed recursively, until the entire expression being expanded is expressed in terms of the primitive forms of the language.

3.2 Some properties of macro languages

A major characterization of a macro language is whether it operates at a lexical or syntactical language. Lexical languages proceed by arbitrary substitution of tokens. But the language independency of these macro languages

is repaid by their inability to ensure the correctness of the transformation: they just ignore the syntax of the underlying host language. On the other hand, macro languages working at a syntax level, uses an abstract syntax trees that allows them to ensure the correctness of the transformations they are performing.

Some macro languages working at syntactical level only provide pattern matching and substitution while others allows explicit programming on the abstract syntax tree. This last property is known as programmable macro.

While lexical systems seems to have been supplanted by lexical systems, a new working level is emerging. It is based on acknowledging the failure of syntax based systems when the macro programmer needs to deal with logical and/or contextual information. These macro languages are using reification of entities comparable to the well studied metaobjects of the reflection community to offer to the macro programmer a logical and contextual representation.

Type checking is an important problem occurring when the underlying language is strongly typed. Normally, at least the arguments and the return type of the macros should be checked in order to grant the correctness of the transformation performed by the macro language.

Recursion is another problem arising when dealing with macros. Direct recursion designs the fact that the body of a macro contain an invocation of itself. Indirect recursion covers the fact that a self invocation is created during the transformation. Usually both the macro language preprocessor and the macro user should take great care to avoid infinite loops. In system based on reflection-like entities, the macro expansion is type driven: macro expansion is managed by the reification of a type, in other words of a class¹.

There are several approaches to implement a macro languages. First, language can be distinguished on whether the macro body is expanded eagerly at its definition or lazily at each invocation. Since names clashes can occur, renaming is often used: macro languages that ensures that no such name clashes will occur are known as hygienic. While macro languages are designed to be transparent, meaning that the next phases of the compilation or interpretation process does not need of be aware of the macro systems, errors reporting is usually a headache.

In this section, several properties identifying a macro system have been, identified: the operation level, whether programmable macros are available or not, type checking and recursion capabilities, eager or lazily hygienic macro expansion, and error reporting.

3.3 Macro languages: a small survey

The purpose of this section is not to study the macro languages for themselves but to review wide spread macro languages towards the properties described before to check whether this set of properties can give a useful characterization. In that case, it might be interesting to discuss the previously described properties with Naive. The reviewed macro languages are: CPP: the C preprocessor [KR78], M4 [Ker81], T_EX [Knu01], Dylan [Sha96], C++ templates [Sha96], Scheme [KCR], Jakarta Tool Suite (JTS) [BLS92], bigwig [BS00] and OpenJava²

¹it seems that the terms type-driven was coined by [TCKI00].

²OpenC++ has been presented as a macro system too in [Chi98] and, despite the underlying language, is close to OpenJava.

[TCKI00].

Considering 3.1, it appears that no one of the macro languages considered offers the same answer to the properties identified in section 3.2. We can therefore conclude that this set of properties can be used to characterize macros languages.

3.4 Summary

This chapter has precised the notion of macros: in a reduced view, they appear as a tool performing an advanced copy paste of code in the source code before interpretation or compilation. It has isolated and proved on a small survey that a macro system can be roughly characterized by a small number of properties: the operation level, whether programmable macros are available or not, type checking and recursion capabilities, eager or lazily hygienic macro expansion, and error reporting.

| | CPP | M4 | TeX | Dylan | C++ templates | Scheme | JTS | bigwig | OpenJava |
|---------------------------|---------|------------|--------|---------------------|------------------|-------------|-------------|-------------|------------------|
| Level of operation | lexical | lexical | hybrid | syntactical | syntactical | syntactical | syntactical | syntactical | metaobjects like |
| Language dependent | no | no | yes | yes | yes | yes | yes | yes | yes |
| Programmable conditionals | n/a | arithmetic | yes | no | constant folding | yes | yes | no | type driven |
| Type checking | n/a | n/a | n/a | only argument types | yes | implicit | yes | yes | yes |
| Direct recursion | no | yes | yes | yes | no | yes | no | rejected | no |
| Indirect recursion | no | yes | yes | yes | yes | yes | n/a | n/a | n/a |
| Body expansion | lazy | eager | lazy | lazy | lazy | lazy | eager | eager | type driven |
| Hygienic | no | no | no | yes | no | yes | yes | yes | yes |
| Guaranteed termination | yes | no | no | no | no | no | yes | yes | yes |
| Transparent | yes | n/a | yes | yes | yes | yes | yes | yes | yes |
| Error reporting | n/a | n/a | no | no | no | no | no | yes | yes |

Table 3.1: Characterization of some macro languages.

Dylan mixes operations at lexical and syntactical level.

Constant folding is an optimization that replaces a call of constant arguments with the constant result of that call.

Chapter 4

Bytecode manipulation libraries

Surveying the libraries allowing to manipulate bytecode was a necessity with several objectives. At least, ensuring that no bytecode manipulation library already exists offering the features we wanted to develop was needed. From this viewpoint, this chapter is an integral part of the state of the art of this thesis. Moreover reifying the abstractions presented in chapter 2 is a huge work. Furthermore, the reification of these abstractions while necessary to implement a prototype was not the main goal of our work. It was therefore natural to use a third party library providing them.

Therefore this chapter should be understood from its double goal: ensuring that no bytecode manipulation library with similar features already exist and secondly reviewing them in order to choose one suitable to develop our prototype on top of it. This presupposes to have a better idea of what the prototype might expect from a bytecode manipulation library: this will be discussed before actually describing the libraries available.

4.1 The prototype requirements

The choice between the different libraries has been governed by four criteria: expressiveness, the programming language in which the library is implemented, the quality of the abstractions provided and licensing issues.

It may sound odd to present expressiveness as an important criterion to choose a bytecode manipulation library. However Java bytecode allow to express any -correct- construction appearing in Java source code. A partial or limited reification of the abstractions presented in chapter 2 used by the virtual machine may prevent a bytecode manipulation library either to describe, either to generate some construction appearing in the Java source code. For instance, a library neglecting to reify the exception table will prevent many alterations of the exceptions mechanism used in Java source code to handle errors. Therefore, one strong requirement in the choice of a bytecode manipulation library is to be able to express the same constructions appearing in Java source code.

Secondly, the implementation language of the bytecode library is important. To achieve our goal of portability, it is desirable that our prototype only uses portable libraries. Java distinguishes itself from other languages by its ability to cope with different platforms: [GM96] states that *Java is designed to support applications that will be deployed into heterogeneous network environments. In such environments, applications must be capable of executing on a variety of hardware architectures.* If we want to be able to run our program in every platforms where Java is available, it is natural to rely only on Java written libraries.

Moreover, the quality of the abstractions provided could not be neglected: there is an enormous gap between low-level objects close to the physical representation and a high level logical representation. In the first case, the programmer is in charge of maintaining and ensuring the correctness of the relationships between the different objects involved in the bytecode generation. For instance, a low-level representation may allow to add a bytecode instruction taking an index in the constant pool without even checking that this index is not outside of the constant pool¹. A logical representation eases the programming task by hiding low level details and by automatically maintaining the constraints imposed by the virtual machine between the different abstractions. For example, in a well design library most manipulations of the constant pool can be hidden.

Finally, the licensing problem corresponds to the fact that we prefer to work with open source tools rather than with commercial tools. This in turn allows us to distribute our work freely to any person interested.

It is important to be aware of the tensions between the goal of expressiveness and the qualities of abstractions criteria. Actually, it is difficult to reach a higher level of abstractions without presupposing the end user tasks. If the end user goals do not match exactly the presupposed activities, it is likely that the expressiveness will suffer. This does not mean that the library is bad; it only stresses that the library is not well adapted to the user needs. The following survey of bytecode manipulation libraries needs to be understood within our context: inlining a bytecode instruction sequence in place of another pre-existing instruction.

4.2 Bytecode manipulation libraries available

Although still few people trust it, bytecode manipulation is being used in a large spectrum of applications: BIT [Lee96] offers an instrumentation interface for Java classes modelled on ATOM [SE94]. Like the latter, it constrains the alteration to preserve the semantics of the instrumented code. For example, a method that will be inserted into another one has to be static and has to take a single argument. Binary Component Adaptation [KH97] is a bytecode transformation environment designed to solve the problem of integrating incompatible software components. It addresses external interface issues, such as method signatures and names, but not method implementations. Kimera [SGGB99] uses

¹Remember that the constant pool is no more than an array of entries

bytecode transformation to implement a distributed virtual machine infrastructure. Hyper/J [OT00] uses bytecode manipulation to implement a language extension for Java allowing the separate development of applications and aspects of code encapsulating features cross-cutting the existing methods. Unfortunately, neither of them is presented as a general-purpose transformation.

Fortunately, there are several general-purpose implementations of bytecode manipulation available: BCEL, JikesBT, and JOIE. All of them translate the class file data structure into an intermediate internal representation, let the user perform the modifications he wishes and then regenerate a valid class file data structure from the altered intermediate representation.

4.3 BCEL overview

BCEL stands for Byte Code Engineering Library. The purpose of BCEL as described in [Dah99] and in [Dah01] is *”to give the users a convenient possibility to analyse, create, and manipulate (binary) Java class files”*. BCEL achieved its goals by reifying all the abstractions and data structures specified in [LY97]. BCEL offers two levels of reification: a static one and a dynamic one. Each level of description has an associated package in BCEL clearly separating the scope of the different abstractions in use. The static level is used to describe a class from a virtual machine point of view while the dynamic level allows altering a class file. The static level can be roughly understood as the intermediate representation: reaching the dynamic level requires building first the abstractions representing a given class at a static level. Saving a dynamically modified class file requires to regenerate a static description from the dynamic description. This leads to a rather functional approach. On the other hand, BCEL way of improving the abstraction level it offered relies upon the use of object-oriented techniques and design patterns. This approach introduces no assumption on the activities of the user. BCEL does not offer any built in mechanism to safely copy paste a method body in place of a bytecode instruction.

4.4 JikesBT overview

JikesBT or Jikes Bytecode Toolkit has been proposed by Chris Laffra *”as a 100% Java class library which enables Java programs to create, read, and write binary Java class files”*. Its main difference with BCEL is its ability to represent the class file logically in memory. A logical representation focuses on the relationships between the abstractions that come into play in the virtual machine more than on their physical representation. The fact that the accent is put on the relations between abstractions allows to hide the constant pool to the user as long as he does not want explicitly deal with it. This slightly eases the programming task without making assumption on the activities the user is willing to perform. JikesBT does not offer any built in mechanism to safely copy paste a method body in place of a bytecode instruction.

4.5 Joie overview

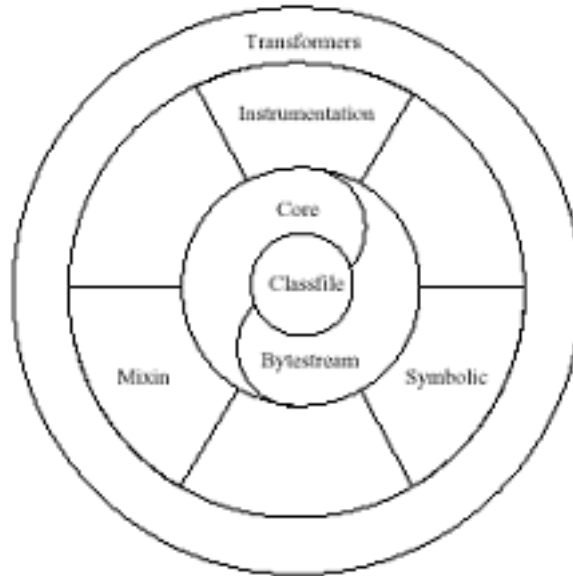


Figure 4.1: JOIE layers (taken from [CC01])

JOIE stands for Java Object Instrumentation Environment. JOIE as described in [CC01, CCK98] is a multilayered library: each layer address an identified level of abstraction. class file is the lowest level surrounded by two intermediate layers: Bytestream and Core. As shown on figure 4.1, three high level layers are available: Symbolic, Mixin and Instrumentation. class file is not meant to be used and direct usage of Bytestream and Core is as well discouraged. The Mixin interface allows to access and manipulate class-level structures, such as fields and methods, modifying them to copy features from externally supplied auxiliary classes. This provides a simple way to add fields, methods, or method prologues and epilogues without requiring knowledge of the bytecode. The Instrumentation layer allows the insertion of sequences of instructions at well-defined points of methods. The Instrumentation is limited to insertion; no replacement of a given instruction can be performed. The Symbolic interface is designed to create and manipulate instructions. It represents instructions as instances of classes that manage much of the bookkeeping complexity. The Symbolic interface interprets instruction operands and presents them as references to typed objects with useful methods and behavior.

While this layered architecture seems to be one of the major strength of JOIE, it raises specific problems: as stated by the designers themselves: upper layers may maintain private state that is unknown to the bottom layers. Especially JOIE does not attempt to maintain the consistency between the different layers: *”since Symbolic is layered above Bytestream, manipulations to instructions using Symbolic are reflected at the Bytestream level. However, the converse is not true: Symbolic supplements the intermediate representation with structures that are not updated by Bytestream. Thus transformers*

| | Expressiveness | Implementation language | Abstractions quality | Licence |
|---------|--|-------------------------|----------------------|---------------------------------|
| BCEL | Adequate | Java | Adequate | LGPL (free) |
| JikesBT | Adequate | Java | Excellent | AlphaWorks (limited to 90 days) |
| JOIE | Adequate in bottom layers Bounded in upper layers | Java | Adequate | Duke University license (free) |

Table 4.1: Comparison of bytecode manipulation libraries

cannot safely use both Bytestream and Symbolic to manipulate the same instructions”[CC01]. JOIE designers do not felt this as a problem because according to them: *”upper layers are specialized to meet common functional requirements without direct use of the bottom interfaces*”[CC01]. Unfortunately, there was no suitable upper layer higher than Bytestream suitable to copy paste a method body in place of a bytecode instruction. In other words, the higher level of abstractions provided by Mixin, Symbolic or Instrumentation restricts the user activities from replacing a bytecode instruction by a sequence of bytecode instruction. These hypothesis in turn are bounding the expressiveness of the possible alterations in such a way that the user was not able to replace an instruction by safely a method body².

4.6 Evaluation

The discussion on the adequation of BCEL, JikesBT and JOIE on the light of our four criteria chosen is summarized on Table 4.1: hopefully, it seemed that each of the libraries considered could allow us to implement a mechanism replacing a given bytecode instruction by a sequence of bytecode instructions. The choice can be reduced to a comfort question: while it is clear that JikesBT minimizes the development effort, its licence is problematic. JOIE while interesting suffers from layering problems and therefore does not offer an abstraction level hight than its bottom layers whose use is discouraged. It therefore appears that BCEL was the more adapted choice.

4.7 Summary

In short, in this section several bytecode manipulation libraries has been described in the light of evaluation criteria suitable to the goal of replacing one given bytecode instruction by a sequence of externally bytecode instructions. These criteria are expressiveness, implementation language, licence and abstractions quality. Among the others, the abstractions quality was probably the most important. We retained BCEL as most suited to our needs. While looking for a bytecode manipulation library,

²Another problem arising with JOIE is that all the upper layer described in [CC01] are not included in the main distribution. It further seems that there is not clear separation between the different layers actually making the problem of mixing the use of upper layers more burning

we discovered that no one offers a built in ability to perform the inlining of an externally provided bytecode instruction sequences with an abstraction level comparable to Java source code. From this viewpoint, the previous discussion goes beyond an implementation problem: it is an integral part of the state of the art.

Chapter 5

Java reflective extensions using bytecode rewriting

”Augmenting a language with a metaobject protocol does not need to be radical change. In many cases, problems with an existing language or implementation can be improved by gradually introducing metaobject protocol features.”

[KDRB91]

Reflection refers to the *”abilities to represent, operate, on and otherwise deal with itself in the same way that it represents, operates or and deals with its primary subject matter”*[Smi90]. Clearly, the alterations capabilities provided by a copy paste mechanism can help to develop a reflective system in Java. Conversely, one might wonder if existing reflective extensions of Java is not already offering this kind of copy paste mechanism . It is the purpose of this chapter to answer to this question.

First, general concepts about reflection will be introduced before describing the reflective extensions available for Java. Finally, the extensions based on bytecode rewriting techniques will be discussed.

5.1 Reflection in general

[Smi84] highlights that a reflective system can be understood as a system containing its own interpreter. The self modifying ability entails that the system is able to modify its own interpreter which in turns results in modifying the program execution. Since the system can reason on its own interpreter and modify it, the interpreter can be seen as a program executed by another interpreter. This second interpreter is part of the system and can therefore be modified. Consequently, it can be understood as a program executed by a third interpreter and so on. Each program interpreter plays the role of a program executed by another interpreter. This duality program/interpreters structures reflective systems in layers or meta-levels. In practice, however, only two levels are commonly considered: the first level is called base level describes the task that the system has to achieve (the *what*) while the second level called metalevel interprets the base level (the *how*).

Somehow, Java alone is already a reflective system. In fact, according to [BGW93], if *“reflection is the ability of a program to manipulate as data something representing the state of a program during its own execution, there are two aspects of such manipulation: introspection and intercession. Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to observe and therefore reason its own execution state or alter its own interpretation meaning.”* Therefore, Java alone appears somehow as a reflective system because it offers introspection capabilities¹.

The state of the art approach to implement reflective object-oriented systems offering introspection as well as intercession relies upon language based approach providing a metaobjects protocol. [KDRB91] describes them as resulting of three design stages:

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

However this approach is tightly integrated with the language design. Java was not designed with this goal in mind: this has lead implementors to use different techniques to augment Java with reflective abilities.

There is a difference between structural reflection: *“the ability to allow a program to alter the definitions of data structures such as classes and methods”*[Chi00]. Behavioral reflection can be derived from structural reflection as shown in [Chi00, Tan00]. [Chi00] provides a small metaobject² protocol providing behavioral reflection thanks to his implementation of a structural reflection: Javassist. Likewise [Tan00] describes a metaprotocol built on top of Javassist adapted to mobile agents.

5.2 Reflection in Java

[BS99] states that several approaches can be followed to add reflective capabilities to Java. These approaches are:

1. Modify the source code of the base level to glue the hole between the metalevel and the base level.

¹Through `java.lang.Class` and through the package `java.lang.reflect`.

²package `javassist.reflect`

2. Modify the bytecode of the base level with the same goals in mind.
3. Modify the virtual machine interpreting the Java byte-code.

Others approaches might be possible like building an interpreter of code source, or using partial evaluation. Nevertheless, we find no explorations of these possibilities in the literature. However, in a context more general than the implementation of a Java metaprotocol, Chiba in [Chi97] discusses various approaches to implement a metaobject protocol.

The following enumeration summarizes the metaobject protocols available to our knowledge:

- Examples of the first approach are Open Java[Tat99] and Proactive [Vay97].
- Examples of the second approaches are: Javassist[Chi00], Kava[WS00], and Reflex[Tan00]³.
- Examples of the third approach are: Metaxa[Gol97] and Guarana[OCGB98].

The major problem of the last two approaches is a portability problem while the problem of the first approach is to require an access to the source code of the base level.

It makes sense to compare the different implementations on the basis of the interceptions they allow to realize. In an object-oriented world, the main activities are methods calls, instance creation and accessing instance and class variables. Consequently, a metaobject protocols should offer the ability to refine this activities. Each of these activities can be understood from a sender viewpoint: for instance, one object wants to send a message to another or from a receiver viewpoint: for example, one object is asked to execute a given message. Each point of view is legitimate because it allows redefining the semantics of messages either per sender or per receiver.

Further, in an impure language like Java, it is interesting to be able to refine through the metaobject protocol some of the behavior that is not by default reified. In the case of Java, monitors⁴ and arrays are good candidate for such reification.

One of the major strengths of object-oriented technology is the ability to incrementally refine the protocol offered by subclasses specializing a given behavior. As pointed out by [KDRB91], a metaobject protocol is no more than a standard protocol. It is therefore wishable to let the user tailor it to its needs using the standard object-oriented construction offered by the language. However, the necessity to maintain the relationship between the runtime and the metaobject protocol may limit the user ability to refine the metaprotocol.

Finally, to evaluate the different reflective extensions four families of comparison criteria have been distinguished:

³Reflex is written on top of Javassist and therefore we will not described it more

⁴In Java each object instance has by default a monitor associated. But the monitor queue can not be directly manipulated.

| | Proactive | OpenJava | Javassist | Kava | Metaxa | Guarana |
|-----------------------------------|-----------|----------|-----------|----------|--------|---------|
| Message send | | yes | | yes | yes | |
| Message received | yes | yes | yes | yes | yes | yes |
| Constructors send | yes | yes | | | yes | |
| Constructors received | | yes | yes | yes | | yes |
| Fields read or write | yes | yes | | | yes | yes |
| Monitors | | | | | yes | yes |
| Arrays | | yes | | | | yes |
| Dynamic changes of metatreatments | yes | possible | yes | possible | yes | yes |
| Structural reflection | | yes | yes | | n/a | n/a |

Table 5.1: comparison of metaobject protocols for Java.

This an adaptation of a figure appearing in [BS99]

Note that we disagree with [BS99] when he states that OpenJava makes it difficult to maintain the metalink at runtime. We believe that depending on the user needs, the insertion of the appropriate hook may allow it as in any libraries providing structural reflection.

- the main activities performed in an object-oriend world: methods calls, instance creation and accessing to instance variables, all of them on a per sender or per receiver basis
- the ability to refine some first class object in particular monitors and arrays
- the ability of incrementally refine the metaprotocol using the standard object-oriented constructions offered by the language
- the degree of control on the hooks inserted given to the user

Our conclusions are summarized on Table 5.1. To our knowledge, in all reflective extensions of Java, the control of instance variables is limited to a sender viewpoint

Kava allows to refine the elements presented in Table 5.1 but only by providing before and after methods. This means that the refined activities in the base level will always takes place. This is slightly different from replacing the activity performed in the base level by an inserted code: the metatreatment. Javassist is quite powerful but still relatively oriented on a receiver point of view. Copying a method body in place of a language construct fits more with a sender viewpoint. All in all, by comparison with the other reflective extensions, it appears that the idea of replacing a given language semantic by a method body seems to be relatively complementary of existing reflective extensions. While this conclusion is interesting in itself, it might wondered if extensions built on bytecode rewriting do not use a copy paste mechanism hidden in an implementation. It is therefore needed to analyze Kava and Javassist in more details.

5.3 Kava

Kava is built on top of JOIE⁵. Kava rewrites method bodies using load time bytecode engineering. However, it is difficult to analyze Kava because the protocol presented in [WS00] and in [WS01] are not consistent with the distributed release⁶. In each of the previous papers, and in Kava 0.9, only before and after methods are provided. For example, in Kava 0.9, a given metaobject may be notified:

- Before or after finalization in other words the garbage collection of the base object,
- Before or after reading or writing a given field,
- Before or after sending or receiving a method⁷,

Thus Kava is only providing before and after methods in the metaprotocol: the only way to prevent a given operation to take place (like finalization or sending or receiving a message or reading or writing a field) is to throw an exception in the before method of the metaprotocol. Kava metaprotocol does not declare any specific exceptions in the signature of the metaobjects methods: this entails that due to Java language semantics, Kava metaprotocol can only throw `RuntimeException`. Actually, `RuntimeException` (and its subclasses) never needs to be part of a method signature: `RuntimeException`s can be thrown at any time, anywhere in any block of Java code. Thus, the only way to prevent a given operation to take place is to throw a `RuntimeException` in the related before method of the metaprotocol.

```
metaclass kava.MetaTrace {
    putfield( * , i , * , * );
    receivemethod( test , * );
} class * metaclass-is kava.MetaTrace ;
```

Figure 5.1: an example of Kava configuration binding. (file `META.CFG` provided with the examples of the Kava 0.9 distribution)

Because Kava does not provide structural reflection but behavioural reflection, it makes sense to consider the relationships between metaobjects and base objects in Kava. This binding is achieved by separately provided configuration file binding one class to one metaobject. For each metaobject, the configuration file specifies the after and before methods it is interested in. As it can be seen from figure

⁵Although [WS00] presents it as being written on top of BCEL, the distributed version of Kava 0.9 depends of JOIE.

⁶In particular, [WS00] presents an API based on an interface `IMetaObject` and context objects (like `IExecutionContext`, `IfieldContext`, `IInvocationContext` and `IexceptionContext`), but none of the latter objects seem to be part of the currently distributed version of Kava.

⁷It seems that Kava 0.9 does not include a method notified before or after an exception is thrown in the base level code contrary to the metaprotocol described in [WS00].

10, this binding language is relatively different from the Java language.

In each version of Kava protocols known to the author, the metaprotocol methods take various arguments whose classes are specific to Kava⁸. These objects are instantiated at runtime in the bytecode inserted by Kava in the base level. This introduces a tight relationship between the metalevel: Kava classes and the base level. In particular, it becomes impossible to run the base level in the absence of the metalevel classes, i.e. in the absence of Kava classes.

Finally, we have seen that Kava although relying on very expressive bytecode generation libraries only provides before and after like methods. Kava rewrites method bodies using load time bytecode engineering. Metaobjects are bound to base level objects by an externally provided configuration file written in a dedicated language. The only way to prevent a given operation to take place in the base level is to throw a `RuntimeException` in the before method of the metalevel. Moreover, base level classes made reflective by Kava become runtime dependant of Kava classes.

5.4 Javassist

javassist has been designed with two goals in mind: first provide source code level abstraction, and secondly provide efficiency and safety in terms of types.

As said previously javassist, enables structural reflection (we will not cover the aspect of behavioral reflection coming with javassist in the package `javassist.reflect`). As described in [Chi00] using javassist essentially means dealing with five objects: `javassist.CtClass`, `javassist.CtField`, `javassist.CtMethod`, `javassist.CtConstructor` and `javassist.CodeConverter`. A `CtClass` describes a class file, a `CtMethod` a method, a `CtField` a field, `CodeConverter` is used to perform some alterations on a method. With these objects a given user can:

- Perform the same introspection task that offers the standard Java with the difference that the given class file is not loaded by the virtual machine.
- Change modifiers of a method⁹ : visibility (`private`, `protected`, `public`) as well as `synchronized`
- Copy method from a given class into another, exchange method bodies
- Add simple methods (getter, setter and methods that returns immediately), and fields to a given class
- Change the super class of a given class

⁸`IExecutionContext`, `IFieldContext`, `IInvocationContext` and `IExceptionContext` in [WS00] `kava.Method` `kava.ValueArrayList` `kava.Value` `kava.Reference` in Kava 0.9.

⁹Some of the methods enabling this are now deprecated in Javassist 1.0.

- Instrument the body of a given method

The instrumentation of a method body allows:

- Replace the instantiation of a given class by another
- Replace a given method call by another
- Replace a given field access by another

Therefore, it can be said that the granularity of the structural reflection provided by Javassist is more or less the members of a given class. For instance, the instrumentation of a method body is too limited to be usefully used in the meta object protocol offering behavioral reflection that comes with Javassist: it is based on renaming techniques of different class members.

This and the restricted instrumentations abilities suggest that Javassist makes most of its modifications in the top-level structures of the class files where the members are defined and in the constant pool. Ensuring this, requires understanding the representation of the bytecode used by Javassist.

Javassist unlike Kava does not use a third party library of bytecode manipulation. Bytecode manipulation is achieved thanks to the package `javassist.bytecode`. The inheritance tree below `ConstInfo` completely reify the constant pool entries. Major abstractions such as class file, constant pool, attribute, field, methods and the exception table are reified by `ClassFile`, `ConstantPool`, `AttributeInfo`, `FieldInfo`, `MethodInfo`, `ExceptionTable` respectively. However, no reifications of the bytecode instructions are available. The `Bytecode` class provides a limited ability to add instructions in the code attribute containing the bytecode instruction flow of a given method. But the added instructions should not perturb the stack. For example, the maximal stack depth will not be recomputed.

This is an important restriction to Javassist ability to manipulate bytecode. It is therefore not surprising to discover that the `Transformer` inheritance tree, on which the `CodeConverter` is relying, only changes the indexes taken by bytecode instructions in the constant pool. Javassist exploits the fact that field read, field write, methods and constructors invocation all map to bytecode instructions whose indexes in the context pool indicates which field to read, write, method or constructor to invoke respectively, to reexpress them with source level concepts.

All in all, we have seen that Javassist:

- Allows structural reflection.
- The grain is more or less the member of a class.
- Ability to transform a method body is limited to changing the indexes of a bytecode instruction. This in source code terms allows to replace the instantiation of a given class by another, replace a given method call by another, and to replace a given field access by another.

Javassist metaprotocols are built using class members renaming schemes.

5.5 Summary

We have seen in this section that if there were several reflective extensions of Java available, few of them rely upon bytecode rewriting. Approaches relying on interpreter modifications suffer from a lack of portability. The problem could have been solved by source code approaches but the source code is usually not easily available when the virtual machine is loading a class. class file rewriting at load time emerges as a solution able to provide augment the language with reflection. Two libraries already exist exploring this path: Kava and Javassist.

Kava, on the other hand, is only providing after and before methods. Because in Kava the base semantic of the language is executed, because Kava only offers to refine the base semantic of the language before or after it did take place, there is no easy way to replace a given semantic attached to the language. Replacing a bytecode instruction corresponding to a given semantic attached to the language, can be thought as a complementary approach of the alterations that Kava can perform.

In Javassist, the granularity of the rewriting is more or less the member of a given class: this limits the families of metaprotocols deriving from Javassist to a receiver viewpoint based on renaming schemes. Replacing bytecode instruction corresponding to a given semantic attached to the language can provide the complementary sender viewpoint missing in Javassist.

Chapter 6

Designing the copy paste mechanism

”Courage: within the context of the first three values-communication, simplicity and feedback-it is time to go like hell.”

[Bec99]

In this chapter, the prototype: Naive is presented. First, its relations with reflection and macro systems are discussed, establishing its motivations. This leads to secondly enumerate the goals of the design. Thirdly, the entities allowing the alterations are presented before precisising their extents. But the altered semantic contain information and this information has to be passed to the user before and after performing the alteration. Next a full example will be presented. The fact that a copy paste mechanism is used entails that some language constructs do not make sense anymore: they will be finally discussed.

6.1 Nature and motivations of the tool

The idea behind Naive is very simple: there are bytecode instructions that directly map to source code language constructs. Unfortunately, Java ability to refine its language constructs are quite limited. Naive proposition is to replace the bytecode instruction corresponding to one of this constructs by an arbitrary sequence of bytecode effectively allowing to refine some of Java language constructs.

But bytecode is not a friendly playground for most programmers, and therefore reaching an abstraction level comparable to the source code level is a major goal. Because a method body can contain an almost arbitrary sequence of bytecode instructions and because methods are a familiar concept to any Java programmer, it leads us to rephrase the sentence: *”replacing the bytecode instruction corresponding to one of this language constructions by an arbitrary sequence of bytecode”* into *”inlining an externally user provided compiled method in place of a given semantic appearing in a given method”*. In particular, it entails that the user will only deal with the construct he is willing to refine in a given method. While this makes the prototype implementation responsible of mapping from a given semantic to the bytecode instructions to be replaced, it frees the user from low level details (the bytecode instruction to be replace and the sequence of instructions to be inlined) and provides him with a higher

source code like view (the construct to be altered and the method to be copied in place of this construct).

While this is a huge simplification, a useful metaphor is to reason on source code instead of bytecode. Considering the Java source code of the method body to be altered, what this thesis tries to achieve, is that each time a given Java language construct is encountered the Java source code of another method specified by the user is copied in replacement of the language construct. But instead of performing this copy in the source code, the prototype of this thesis does it at the bytecode level since the Java runtime environment relies only upon bytecode and never on source code.

Normally this kind of copy/paste mechanism works on the source code and is supported by macros. Unfortunately the Java runtime has almost no knowledge of the source code. This highlights the need of a macro system that does not work on source code but on the runtime data structures. In the latter case, the macro expansion is performed after the compilation at the bytecode level before interpretation contrary to the systems presented in chapter 3 where the macro expansion is performed at the source code level before compilation. While this is an abuse extension of the term, we propose to describe macro expansion performed at a bytecode level as *runtime level macro*.

This runtime level macro system: naive described below should increase the abstraction as much as possible: bytecode is not a friendly playground for most programmers. As in OpenJava [TCKI00] and OpenC++ [Chi98], we choose to borrow the representation that the reflection community studied for years. While however Chiba and Tsubori explicitly use the term of metaobject, we prefer to speak of structural entities rather than metaobject, since everything is merged at runtime.

We believe that the finer control possible should be given to the user. Therefore, instead of choosing for a type driven, we choose for a method driven where the structural entity describing a method is responsible of performing the copy paste inside the method. This is an attempt to answer to the needs of implementors of Aspect Oriented Programming system. Other desirable properties of Naive are: full type checking to be hygienic and to grant termination.

Clearly this type of macro system is close to a reflective system: it provides *”abilities to represent, operate, on and otherwise deal with itself in the same way that it represents, operates or and deals with its primary subject matter”* [Smi90]. This reflection capabilities, while strongly oriented on a sender viewpoint, are offered through modifying the compiled form defining classes: it can therefore be described as a structural reflective system. Nevertheless, it also allows to intercept basic operations such as message sends. It is therefore correct to describe it as a reflective behavioral system. One may try to find a compromise stating that it is a reflective system using structural reflection to enable limited behavioral reflective capabilities. Nevertheless, it is important not to get confused: macros and reflection are different things. Macros are powerful: it is significant that the entire CLOS object system [KDRB91] is implemented thanks to the macro system of Common Lisp. But a macro system is not a meta object protocol although it can help to implement one.

This section presented the ideas behind Naive: allowing to inline an externally user provided compiled

method in place of a given semantic appearing in a given method by actually replacing the bytecode instruction corresponding to one of this semantic by the bytecode instructions corresponding to the method body provided by the user. The system is described as a method driven runtime level macro system using reifications borrowed from the reflection field that can be used, but substantially different from, a reflective system using structural reflection to enable limited behavioral reflective capabilities. The system may fill the gap between the Java runtime and the macro systems based on source code as well as providing a sender viewpoint missing in most Java reflective extensions.

6.2 Design goals

Naive has been designed with three main goals in mind:

- abstraction,
- reusability,
- expressiveness,

Abstraction corresponds to the fact that the user should be free from the details arising from bytecode manipulation. In particular, a user should never have to deal with entities specific to bytecode but only with source code level code entities.

Reusability corresponds to the fact that the copied method should, as much as possible, not be forced to assume the place where it will be copied.

Expressiveness corresponds to the fact that enough information should be provided to allow a given user to copy methods in replacement of language constructions that perform exactly the same behavior than the replaced methods.

6.3 Reified entities

Naive reifies some *structural entities* borrowed from the reflection community that ultimately are common to the Java runtime environment and to the source code level: `NaiveClass`, `NaiveMethod` and `NaiveField`. They reify respectively a class, a method, and a field. Their instantiations is done by factory methods. Factory methods allow to hide the instantiation relations between the different entities. Each factory method takes a `String` in argument that uniquely identifies the structural entity.

In fact, it could have been possible to reach the same results with factory methods taking in argument something else than a `String` name. For example, the factory method of `NaiveClass` could have taken a `Class` in argument, the factory method of `NaiveMethod` could have taken a `Method` in argument, and the factory method of `NaiveField` could have taken a `Field` in argument. However,

using the Java standard structural entities (`Class` and `java.lang.reflect.*`) will automatically load the related class file in the virtual machine before `Naive` alters it. Since class files are cached by virtual machines, the same virtual machine could not have been able to load the class file after its alteration by `Naive`. Another problem arising with using Java standard structural entities (`Class` and `java.lang.reflect.*`) to instantiate `Naive` entities is that although they may refer to the same class file they are considered only equals if they have been loaded by the same `ClassLoader`.

Compared to other alternative, `Strings` are lightweight, are not `ClassLoader` dependant, and still make sense for the end user. The choice of `Strings` leads to the development of a specific jargon.

A *class member* of a class `SomeClass` is either a field, a method or an inner class declared in `SomeClass`.

The *simple name* of a class is the name of the class as it appears in its declaration after the class keyword. For instance, the simple name of the class `String` is `"String"`.

The *simple name* of a class member is the name of the member as it appears in its declaration inside the class. For instance, the simple name of the field `out` declared in the class `System` is `"out"`.

The *qualified name* or *canonical name* of a class is the package name to which the class belongs, followed by a dot, followed by the simple name of the class. For example the qualified name of the class `String` is `"java.lang.String"`.

Except for methods, the *qualified name* or *canonical name* of a member is the qualified name of the class to which the member belongs, followed by a dot, followed by the simple name of the member. For example the qualified name of the class field `out` declared in the class is `System` `"java.lang.System.out"`.

The *qualified name* or *canonical name* of a method is the qualified name of the class to which the member belongs, followed by a dot, followed by the simple name of the method, followed by a left parenthesis, followed by the class of each of the arguments taken separated by a coma and closed by a right parenthesis. For example the qualified name of the method `equals` declared in the class `Object` is `"java.lang.Object.equals(Object)"`.

This jargon helps to understand how to instantiate `Naive` structural entities:

- a `NaiveClass` is instantiated through the static method `NaiveClass.getNaiveClass(-String className)` where `className` is the qualified name of the class,
- a `NaiveMethod` is instantiated through the static method `NaiveMethod.getNaiveMethod(-String methodName)` where `methodName` is the qualified name of the method,
- a `NaiveField` is instantiated through the static method `NaiveField.getNaiveField(-String fieldName)` where `fieldName` is the qualified name of the Field,

These objects can be used in place of Java standard structural entities - `java.lang.Class` and these found in `java.lang.reflect.*` - to describe a given class file without loading a class file in the virtual machine.

Naive proceeds by replacing a given bytecode instruction with a well known semantic in the source level by the bytecode instructions sequence contained in a method provided by a user. It is therefore natural that the alterations provided by Naive are offered by the `NaiveMethod` class.

Naive provides entities reifying structural entities common to the bytecode level and the source code akin to those provided by standard Java (`Class` and `java.lang.reflect.*`). The provided reifications: `NaiveClass`, `NaiveMethod` and `NaiveField` are instantiated through factory methods taking in arguments the qualified name of the entity.

6.4 Extent of alterations

There are two ways of apprehending the extent of the alterations possible. The most obvious one is by enumerating the possible alterations. But it is at least equally interesting to understand which alterations remains impossible and why.

6.4.1 Possible alterations

The implementation principle behind Naive is based on *"inlining an externally user provided compiled method in place of a given semantic appearing in a given method"*. But for the Naive implementors, the *a given semantic* has to be mapped into a bytecode instruction because ultimately the only thing the Naive system knows to do is *"replacing the bytecode instruction corresponding to one of this language constructions by an arbitrary sequence of bytecode"*. This simple statement explains a lot the boundaries of the possible alterations that Naive enables: a given language semantic can be altered if and only if it is achieved by a single bytecode instruction in the runtime level.

Even at the beginning of this thesis work, this constraint appear to be far too restrictive. In particular, it was clear that they were bytecode instructions that did not map to a "full" language semantic of the source code but only to "specialized cases". For instance, the `GETSTATIC` instruction is used at the bytecode level to read a class variable of a given class while `GETFIELD` instruction is used at the bytecode level to read an instance variable of a class. This is what we call "specialized cases" of a "full" language semantic, in that case: reading the value stored in a class or instance variable.

But the constraint can be easily relaxed. For example, to refine the language semantic corresponding to "reading a class or instance value", the method provided by the user should be copied and pasted each time at `GETSTATIC` or `GETFIELD` is encountered. This just leads to state that a given language semantic can be altered if and only if it is achieved in the runtime level by a set of bytecode instructions,

| | Pseudo source code examples | Bytecode instructions | Implemented |
|----------------------|--|--|-------------|
| Messages sent | <code>object.toString()</code> | INVOKEVIRTUAL INVOKESTATIC INVOKESPECIAL INVOKEINTERFACE | yes |
| Constructor | <code>new Object()</code> | INVOKESPECIAL | yes |
| Exception throwing | <code>throw exception</code> | ATHROW | no |
| Field read | <code>point.x</code> | GETFIELD GETSTATIC | yes |
| Field written | <code>point.x=0</code> | PUTFIELD PUTSTATIC | yes |
| Cast | <code>(Object)point</code> | CHECKCAST | yes |
| RTTI | <code>object instanceof String</code> | INSTANCEOF | no |
| Local variable read | <code>int var=0; var=var+1;</code> | ALOAD, DLOAD FLOAD, ILOAD LLOAD | no |
| Local variable write | <code>int var; var=var+1;</code> | ASTORE, DSTORE FSTORE, ISTORE LSTORE | no |
| Get value from array | <code>array[index]= array[index]+1;</code> | AALOAD, BALOAD CALOAD, DALOAD FALOAD, IALOAD LALOAD, SALOAD | no |
| Write value in array | <code>array[index]= array[index]+1;</code> | AASTORE, BASTORE CASTORE, DASTORE FASTORE, IASTORE LASTORE, LASTORE | no |
| Getting array length | <code>array.length</code> | ARRAYLENGTH | no |
| Creating array | <code>new Object[3]</code> | NEWARRAY MULTIANEWARRAY ANEWARRAY | no |
| Returning | <code>return null</code> | ARETURN, DRETURN FRETURN, IRETURN LRETURN, RETURN | no |

Table 6.1: Extent of the possible alterations.

The first column names the semantic of the language that can be refined. RTTI is an abbreviation for Run-Time Type Identification.

The second column offers an example of the *refined semantic* as appearing in source code.

The third column enumerates the related bytecode instructions.

The fourth column states if the alterations are implemented in the prototype implementation companion of this thesis.

each of them realizing specialized cases of the language construct. The major point to be reminded is that in particular, it implies that language semantic achieved by a set of bytecode instructions working in cooperation can not be altered with Naive.

These considerations allow to enumerate the set of possible alterations that can be done with the technique proposed in this thesis. As presented in table 6.1, these alterations are: message sent, constructor sent, exception throwing, field read or write, type conversion between objects also known as cast, runtime type identification, local variable read or write, reading or writing a value in an array, getting an array length, creating an array, and returning from a method.

This section enumerated the various alterations possible with the technique proposed in this thesis considering that a given language semantic can be altered if and only if it is achieved in the runtime level by a set of bytecode instructions, each of them realizing specialized cases of the language construct.

6.4.2 Not enabled alterations

More or less, there are three reasons that can prevent a given semantic of the source language to be altered with the technique described here. First, there are language semantic implemented by a cooperation of instructions. Secondly some language constructions are not realized by bytecode instructions. Finally there are instructions whose results should conform to some constraints imposed by the virtual machine [LY97]. Furthermore, there are bytecode instructions that do not correspond to anything at a source code level. We will describe the four groups.

Semantic not realized by bytecode instruction

As described in section 2.4.1, exception trapping is not directly realized by bytecode instruction. Precisely speaking, the exceptions trapped in a block of code and the location of the exception handler¹ is not stored in the instruction flow of the `Code` attribute but in another part of the `Code` attribute. This prevents to refine the semantic associated with the `try-catch` statements.

Semantic implemented by a cooperation of instructions

Sadly all loops and conditional constructions appearing in the source code are realized by a cooperation of bytecode instructions. Therefore, the language constructions associated to the `while`, `for`, `if`, `break`, `cycle` Java keywords can not be altered.

Furthermore, `finally` statements are achieved at the bytecode level by a cooperation of what will appear at a source code level as a `try-catch` and jumping (using a `GOTO` instruction) to the piece of

¹an exception handler is the code appearing in a `catch` block in Java source code.

bytecode corresponding to the code of the `finally` statement. For these reasons, `finally` statements can not be currently refined.

Constraints preventing a correct refining

There are only one language construction that can not be refined using this technique because of constraints imposed on the bytecode instructions flow by the specification of the virtual machine given in [LY97]: the `synchronized` keyword when used to achieve block synchronization (`synchronized` can be used as a method modifier as well).

When `synchronized` is used as a modifier (like in `public synchronized Object method() { }`), no special instruction is needed inside the `Code` attribute containing the bytecode body of the method. Therefore, the alterations of the modifier `synchronized` applied to a method is a structural reflection concern and this section has nothing to do with our concerns here.

```
public Object method() {
    //code there - no synchronization
    Object lock = new Object();
    synchronized(lock) {
        //here is a block synchronization on lock
        //code there
    }
    //code there - no synchronization
}
```

Figure 6.1: A block synchronization in a method body

But, a given method body can contain a `synchronized` block like in figure 14. It allows to execute a block of code with the monitor of the object taken. In fact, at a bytecode level, this is achieved by at the beginning of the block taking the object lock with the bytecode instruction: `MONITORENTER` and releasing it at the end with the instruction `MONITOREXIT`. But [LY97] states that *although a compiler for the Java programming language normally guarantees structured use of locks, there is no assurance that all code submitted to the Java virtual machine will obey this property. Implementations of the Java virtual machine are permitted but not required to enforce both of the following two rules guaranteeing structured locking. Let T be a thread and L be a lock. Then:*

- *The number of lock operations performed by T on L during a method invocation must equal the number of unlock operations performed by T on L during the method invocation whether the method invocation completes normally or abruptly.*

- *At no point during a method invocation may the number of unlock operations performed by T on L since the method invocation exceed the number of lock operations performed by T on L since the method invocation.*

Stating it more simply, it means that a correct implementation of the semantic of the `synchronized` keyword may result in a cooperation of bytecode instructions.

Bytecode instructions without mapping to construct at a source level

There are two kinds of instructions groups that do not have even a partial mapping in the source level: those creating uninitialized objects² and those manipulating the stack. The stack only exists at a runtime level, it is therefore useless to provide the ability to touch it using our copy/paste technique. However, uninitialized objects manipulation are a feature needed by metaobject protocols.

The ability to manipulate uninitialized objects are a desirable feature that the Java language source level does not offer. However, uninitialized objects exist at a bytecode level: they are created by a `NEW` instruction and their constructor is invoked using a mechanism akin to private method invocation. During a while, this raises our hope to be able to provide the ability to manipulate uninitialized objects.

Unfortunately, [LY97] states that: *Only the invokespecial instruction is allowed to invoke an instance initialization method (this are the constructors or the class initializer). [...] When doing dataflow analysis on instance methods, the verifier initializes local variable 0 to contain an object of the current class, or, for instance initialization methods, local variable 0 contains a special type indicating an uninitialized object. After an appropriate instance initialization method is invoked (from the current class or the current superclass) on this object, all occurrences of this special type on the verifier's model of the operand stack and in the local variable array are replaced by the current class type. The verifier rejects code that uses the new object before it has been initialized or that initializes the object more than once. In addition, it ensures that every normal return of the method has invoked an instance initialization method either in the class of this method or in the direct superclass..* This clearly means that the bytecode verifier of the virtual machine will reject any classes that contain a method that tries to access to an uninitialized object.

6.5 Different kind of information

During this work it becomes quickly clear that the user may be interested into two types of information answering to different needs: *static information* and *dynamic information*. By *static information* we design information that could be provided by class file reasoning. For instance, the destination type of a cast operation is typically a static information. By *dynamic information* we design information that could only be provided at runtime. For instance, the instance on which a method is called is typically a

²in other words objects before the invocation of their constructors like objects created by the `basicNew` message in the Smalltalk language.

dynamic information. While dynamic information can only be provided to the user at runtime, in other words, after the inlining took place, static information can be provided before performing the inlining. It therefore becomes clear that static information can be used to guide the inlining mechanism: a user can use the static information to choose which method to inline.

6.5.1 Dynamic information

Designing a mechanism to feed the inlined code with the dynamic information could have been a problem but very hopefully it is possible to impose a given signature on the method containing the code inlined. Imposing a signature is a practical way of solving the problem, dynamic information is provided to the inlined code as if it was the arguments of the method. Clearly, this is only an impression and Naive implementation gives it by inlining some more glue code before actually pasting the bytecode sequence contained in the method provided by the user.

Imposing a signature on the inlined method implies to define the arguments of the method as well as the type it returns. As explained before, arguments are used to pass the dynamic information. The return type is used to let the inlined method produce a result in place of the result computed by the replaced bytecode instruction.

The fact that arbitrary values should be exchanged to and out the inlined code entails that a wrapping should be performed by the Naive implementation for the primitive types: `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`). When an argument to be passed to the inlined code is a primitive type, it is wrapped into a Java object (respectively `Boolean`, `Byte`, `Char`, `Double`, `Float`, `Integer`, `Long`, `Short`). The inlined code should use the static information to make the difference between a wrapped value and a non-wrapped one when it matters. When a returned value is expected from the inlined code, it should be returned as an object: the Naive implementation will ensure that a unwrapping will be performed if necessary, possibly ending by throwing a `CastException` if there is a type problem.

This subsection maps the need of exchanging information between the copied method and the containing method to the necessity of imposing a signature on the copied method. Arguments are used to feed the copied method with dynamic information while the return type is used as the result of the replaced construct if any. Wrapping and unwrapping are performed automatically by the prototype to handle the case of exchanging primitive types.

6.5.2 Static information

Static information should be provided before performing the copy to the user in order to let him choose which method to copy. Therefore `NaiveMethod` provides a set of method `accept` whose unique argument is an interface inheriting from `SemanticAlterator`. Each subinterface of `Semantic-`

`Alterator` only defines one method: this method takes the static information in argument and should return a `String` that will be interpreted as the qualified name of the method to be inlined.

Thus the standard usage scenario in `Naive` is to define an object implementing the subinterface of `SemanticAlterator` corresponding to a given semantic, then instantiates the `NaiveMethod` describing the method in which the alteration has to be performed and send it the message `accept` with the previously defined object. Finally the altered class file can be obtained by sending the message `getAsByteArray()` to the `NaiveClass`. This should be the preferred way of retrieving an altered class file since it allows to work entirely in memory. However for comfort a `save()` method is provided in `NaiveClass`: it writes the altered class file in the working directory. By construction this scenario grants termination.

This subsection maps the need to provide the user with static interface to an interface based design. These interfaces provide methods that take the static information in arguments and return the method to be copied. Objects implementing these interfaces can be given to a `NaiveMethod` with the message `accept` to actually perform the semantic alteration.

6.5.3 Information passed for the implemented alterations

While dynamic information and static information are different, they are also largely complementary of each other. For example, giving to an implementor of an Aspect Oriented Programming extension only a reference to the instance that sends a message at runtime is not enough: knowing the method that sends the message is needed. Since avoiding that the pasted copy needs to assume the place where it is pasted is desirable, it becomes clear that the pasted method should be feed both with static information and dynamic information.

The information passed is summarized on table 6.2. We neglect to give the name of the altered method before performing the paste (or in other words, we neglect to give the qualified name of the altered method to the object implementing a subinterface of `Alterator`) because the pasting mechanism requires to send the message `accept` to a `NaiveMethod` object describing the altered method. In particular it implies that the altered method is known when the paste is performed. This is all but no a surprise: when a user tries to do a copy paste, it is better that he knows where he want the paste to be done before using any copying system.

6.5.4 Strings and arrays versus objects

It may strange odd by reading the table 6.2 to pass information only through `String` and an array of objects. Especially in the latter case, it might be though than instead of using an array, using an object providing getter methods will be easier to manipulate for the end user.

| | Information given to <code>Alterator</code> | Information given at pasted method | delivered as |
|---------------|---|--|-----------------------|
| Message sent | qualified name of invoked method | qualified name of invoked method | <code>String</code> |
| | | qualified name of altered method | <code>String</code> |
| | | instance of invoker | <code>Object</code> |
| | | instance of invoked | <code>Object</code> |
| | | values of arguments | <code>Object[]</code> |
| Constructor | qualified name of constructor | qualified name of invoked constructor | <code>String</code> |
| | | qualified name of altered method | <code>String</code> |
| | | instance of invoker | <code>Object</code> |
| | | argument values | <code>Object[]</code> |
| Casts | qualified name of destination type | qualified name of destination type | <code>String</code> |
| | | qualified name of altered method | <code>String</code> |
| | | instance of altered object | <code>Object</code> |
| | | casted instance | <code>Object</code> |
| Field read | qualified name of field read | qualified name of field read | <code>String</code> |
| | | qualified name of altered method | <code>String</code> |
| | | instance of altered object | <code>Object</code> |
| | | instance on which the field is read | <code>Object</code> |
| Field written | qualified name of field written | qualified name of field written | <code>String</code> |
| | | qualified name of altered method | <code>String</code> |
| | | instance of altered object | <code>Object</code> |
| | | instance on which the field is written | <code>Object</code> |
| | | new field value | <code>Object</code> |

Table 6.2: Information delivered

The first column names the semantic of the language that can be refined. The second column describes the information passed to the class implementing an `Alterator` subinterface. The method required by an `Alterator` subinterface takes always one argument typed as `aString`.

The third column describes the information passed to the pasted method at runtime. Pasted methods have to take only one argument: an array of objects (i.e. `Object[]`). The information is passed through this array. Higher indexes in this array are placed at the bottom in this table.

The type of the information passed in that array is given in the fourth column.

The convention is the following: static information always precedes dynamic information.

Two points have to be considered there: first of all, such an object does not exist in the Java standard library because it is answering to a Naive specific need. Therefore a solution based on an object instead of an array will make the altered class file dependent of a Naive specific class to be interpreted successfully in a virtual machine. Rephrasing it in the macro parlance, it means that the macro expansion performed by Naive will not be transparent because the interpretation stage will be aware through classes dependencies of the alteration.

Secondly this is potentially a costly solution. Kava described in 5.3 passes information through Kava specific objects named in Kava parlance: context objects. But Kava authors in [WS01] points out that *” preliminary measurements of the performance of Kava indicate that the most expensive operation is the generation of the context. Presently, this expense is more than doubling the execution speed of a number of instructions. ”*. We did believe that a user worried by type safety could use an object whose constructor take our array in argument and that provide appropriate getter methods. This solution lets users choose between performance and type safety.

In this section, the use of arrays and `String` to exchange information has been presented as ensuring the system transparency. Beyond transparency performance problem may arise: the current implementation let the user choose between type safety and performance.

6.6 An example

```
package naive.thesisexamples.methodinvocation;
public class ModifiedClass {
    public String toString() {
        return "EMOOSE";
    }
    public String method() {
        System.out.println("hello");
        return this.toString();
    }
}
```

Figure 6.2: ModifiedClass class

This section will present a method alteration example to fix the ideas. `ModifiedClass` will contain a method `method()` whose sole purpose is to return the result of a self invocation of `toString()`. As shown on figure 6.6, `toString()` is refined to always return "EMOOSE".

The purpose of the alteration realized in this example will be to paste a method body in place of the `this.toString()` invocation contained in `method()` declared by `ModifiedClass`. The

pasted method body will use the dynamic information he is given to perform exactly the same method invocation using standard Java introspection capabilities.

To achieve this example, a class `MethodInvocationAlteratorObject` will implement the naive provided interface: `MethodInvocationAlterator`. The method `methodCallReplacer()` of `MethodBodyContainer` will contain the method body that will be pasted in place of the `this.toString()` invocation contained in `method()` declared by `ModifiedClass`. The `Example` will perform the alteration. An UML class diagram summarizing this design is presented in figure 6.3.

The message `modify` defined in `Example` actually performs the class file alteration. As shown on 6.6, it proceeds by instantiating a `NaiveMethod` describing the method `method()` declared by `ModifiedClass`. It will send to this `NaiveMethod` object the message `accept()` with in argument an instance of `MethodInvocationAlteratorObject`. Then `Example` will write the modified class file on disk using the method `save()` offered by the `NaiveClass` instance describing `ModifiedClass` and retrieved by sending the message `getDeclaringClass()` to the `NaiveMethod` object method `method()` declared by `ModifiedClass`. Finally, `Example` will instantiate a `ModifiedClass` and display the result of sending it the message `method()` in a console.

As shown on figure 6.6, The `MethodInvocationAlteratorObject` implements a subinterface: `MethodInvocationAlterator` of `SemanticALterator`. It's unique method: `getMethodToInlineOnMethodInvocation()` takes a `String` in argument. It will be called by the the `naiveMethod` local variable of `modify` method of `Example` class (figure 6.6). The `naiveMethod` will pass it through its `String` argument the static information. As stated in table 6.2, the `naiveMethod` will therefore pass it the the qualified name of the method invoked. `getMethodToInlineOnMethodInvocation()` has to return a `String` containing the qualified name of the method to paste in place of this method invocation or null if this method invocation should not be altered. This process is summarized on figure 6.6.

The pasted method described in figure 6.6 just perform the method invocation by introspection. This method: `methodCallReplacer()` takes an array in argument containing the dynamic information describing what the base level code wants to achieve: here invoking a method. As stated in table 6.2, the first element of the array is the qualified name of the method invoked given as a `String`, the second element is the qualified name of the invoker given as a `String`, the third element is the instance of the object invoking the method or null, the fourth element is the instance on which the method is invoke, and finally the fifth element is an object array containing the arguments passed to the invoked method or null if the invoked takes no arguments. If some argument value is a primitive type, it will be wrapped into an appropriate object.

As expected `Example` run in a console displays `EMOOSE` in output.

This section presented an example showing how a method invocation can be performed. The source

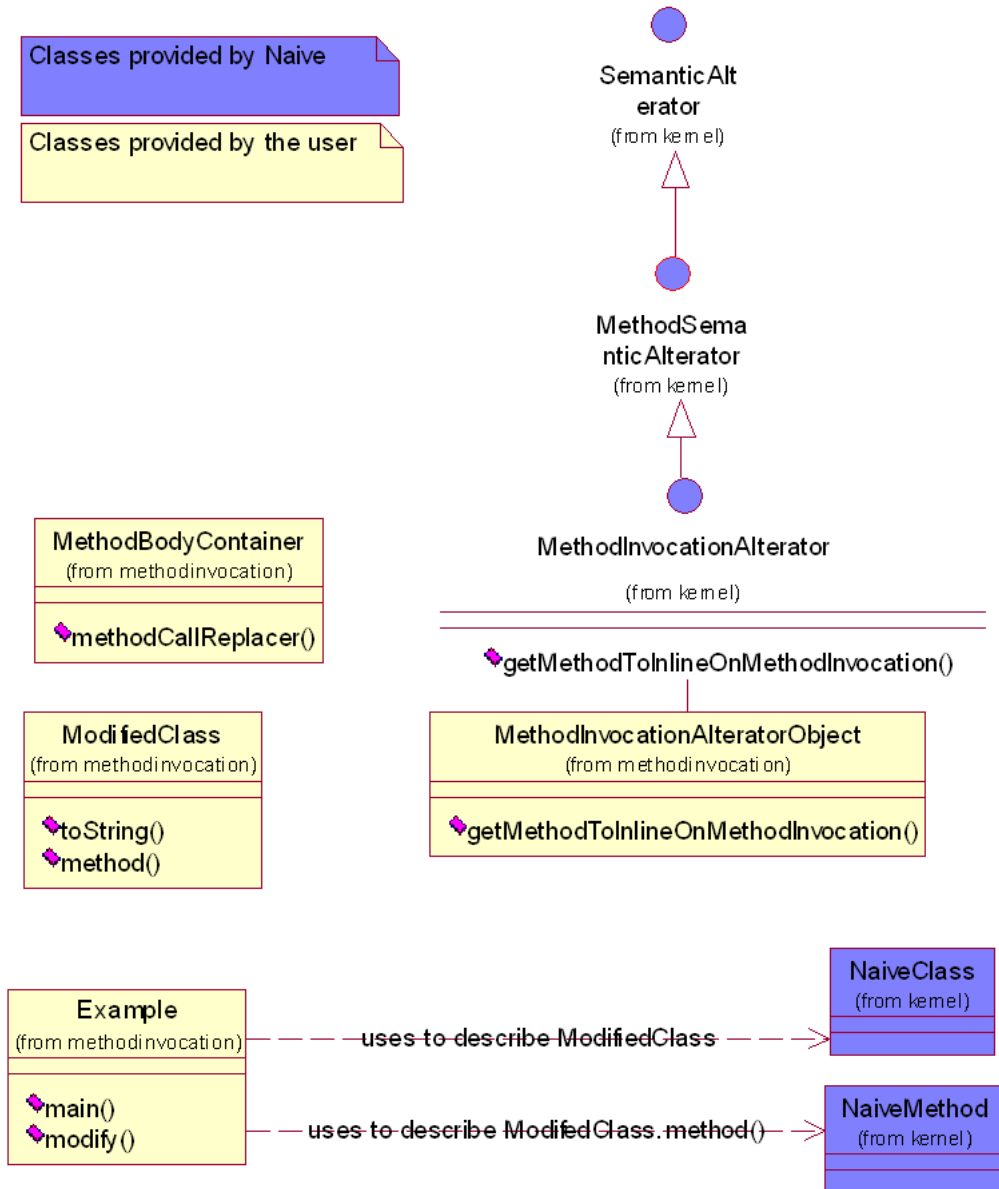


Figure 6.3: Design of the example: UML class diagram


```
package naive.thesisexamples.methodinvocation;
import naive.kernel.NaiveMethod;
import java.io.IOException;
public class Example {
    public static void main(java.lang.String[] args) {
        //do the modifications
        modify();
        //execute the modified code
        ModifiedClass m = new ModifiedClass();
        //display the result
        System.out.println(m.method());
    }
    public static void modify() {
        //creates the alterator
        MethodInvocationAlteratorObject alterator =
            new MethodInvocationAlteratorObject();
        //get the metaobject describing a method
        NaiveMethod method =
            NaiveMethod.getNaiveMethod(
                "naive.thesisexamples.methodinvocation.ModifiedClass.method()
            );
        //perform the alteration
        method.accept(alterator);
        //write the modified class file on disk
        try {
            NaiveClass naiveClass = method.getDeclaringClass();
            naiveClass.save();
        } catch(IOException exp) {
            System.err.println(exp);
            System.exit(-1);
        }
    }
}
```

Figure 6.4: Example class

```
package naive.thesisexamples.methodinvocation;

import naive.kernel.MethodInvocationAlterator;
public class MethodInvocationAlteratorObject
    implements MethodInvocationAlterator {
    public String getMethodToInlineOnMethodInvocation(
        String inReplacementOfMethodCall
    ) {
        return "naive.thesisexamples.methodinvocation.
            MethodBodyContainer.methodCallReplacer(Object[])
            ";
    }
}
```

Figure 6.5: MethodInvocationAlteratorObject class

code presented was based on the previous section.

6.7 Reinterpreted constructs

The fact that a method is actually inlined into another perturbs the semantics of a number of language constructs:

- local variables: how the inlined method avoids names clash,
- returns: how the inlined methods is returning a value,
- exceptions throwing: how the inlined method is signalling errors,
- self references,

Following the metaphor proposed in 6.1, each of these problems appears at a source level if someone copy and past a method into another. It makes clear that the name of the local variables should be different between the copied method and the method where it is past. In the same way, the return type of the inlined method should at least match the return type of the enclosing method. The same problem arises with the exceptions: the exceptions thrown by the inlined method should be caught or be declared in the signature of the enclosing method. And obviously if you do a copy and paste in the source code of two Java class, the `this` keyword used for self reference designs different objects. As it will be seen below, these problems are going deeper than ensuring the integrity of the modified method.

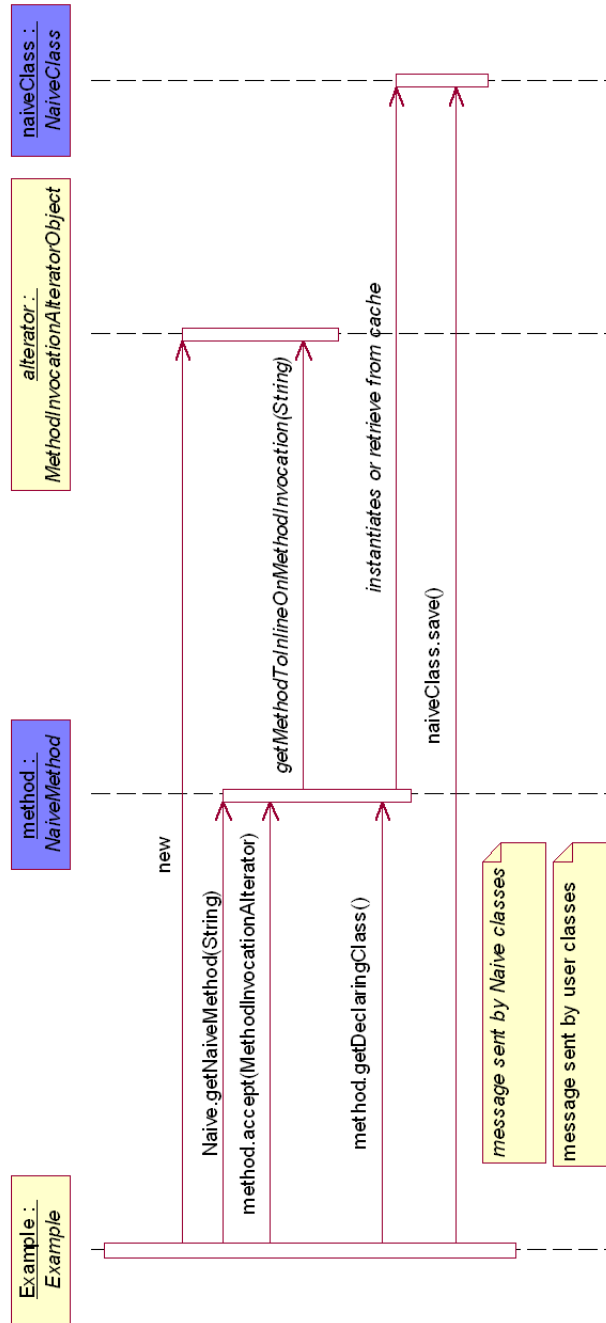


Figure 6.6: Message flow performing the alteration

```

package naive.thesisexamples.methodinvocation;
import java.lang.reflect.*;
import naive.kernel.NaiveMethod;
public class MethodBodyContainer {
    public Object methodCallReplacer(Object[] args)
        throws ClassNotFoundException,
            NoSuchMethodException,
            IllegalAccessException,
            Throwable {
        String qualifiedMethodName = (String) args[0];
        Object methodOwner = args[3];
        Object[] argumentValues = (Object[]) args[4];
        //get the Naive meta obj
        NaiveMethod naiveMethod =
            NaiveMethod.getNaiveMethod(qualifiedMethodName);
        //maps from Naive meta object to standard java meta object
        //Method langMethod=naiveMethod.getAsLangOnDisk() can
        //not be used because the class loader may be different
        Class[] argumentTypes = null;
        if (argumentTypes != null) {
            argumentTypes = new Class[argumentValues.length];
            for (int k = 0; k < argumentTypes.length; k++) {
                argumentTypes[k] = argumentValues[k].getClass();
            }
        }
        Class langClass = Class.forName(
            naiveMethod.getDeclaringClass().getCanonicalName()
        );
        Method langMethod = langClass.getMethod(
            naiveMethodx.getSimpleName(), argumentTypes
        );
        //perform the method call
        try {
            return resu = langMethod.invoke(
                methodOwner, argumentValues
            );
        } catch (InvocationTargetException exp) {
            throw exp.getTargetException();
        }
    }
}

```

Figure 6.7: MethodBodyContainer class

6.7.1 Local variables

The main problem with local variables is that the inlined method could refer to local variable with the same name that a preexisting local variable of the method where it is pasted. For example if at a source level one copy a piece code containing a local variable named `naive` and if the method where the paste is performed already a variable named `naive`, a name clash is likely to occur.

At the bytecode level, things are not very different: local variable does not have a name (a name can be available but only for debugging and is not mandatory) but are identified by a positive number. Therefore a name clash can occur exactly in the same way.

To prevent this, the current prototype, renames the different local variables contained in the pasted method. Actually it just add the greater integer identifying a local variable inside the method where the paste is performed to all local variables contained in the pasted method. This makes the system hygienic.

6.7.2 Return reinterpretation

The main problem with the `return` is the type of the `return` contained in the copied method should match the type of the return type declared in the signature of the method where the paste is achieved. Clearly we can expect not the user to provide a method to be copied with the right return type since it will force the copied method to always assume the place where it will be pasted. Thus, and as stated in 6.5.1, for generality, the inlined method should return an `Object` relying on an appropriate unwrapping facility. This considered, two alternatives have to be considered for the semantic attached to the `return` contained in the copied method:

- `return` in the copied method returns to the enclosing method,
- `return` in the copied method returns to the caller of the enclosing method,

Both semantics are legitimate and makes sense for different cases. However, the first approach enforces the idea of giving the impression that the inlining is like a kind of method calls. This semantics seemed the more logical one and this is the one we retained during the implementation.

6.7.3 Exceptions reinterpretation

Exceptions are a powerful ways of handling errors. In Java, they are completely integrated with the language: neglecting them while implementing our prototype is therefore closer to a mistake than to an acceptable tradeoff. The copied method as any piece of code may encounter error states and need to signal it. It means that the copied method need to be able to throw exceptions to the enclosing method.

However, contrary to the return type, the consistency between the exceptions thrown, the exceptions declared in the method signature, and the `try catch` blocks acting as exception handlers is only checked at compiled time. In other words, contrary to the source level, it is possible to throw any kind of exceptions at any time anywhere in the bytecode regardless of the method signature and regardless of the `try catch` blocks acting as exception handlers.

Nevertheless, as presented in B, despite this ability care must be taken not to break the assumptions made by the level on the exceptions thrown. This leads to restrict the exceptions that the copied method may throw to exceptions that the enclosing level expect.

We said that an exception is *expected* in a method if there is a `try` block handling the type of the exception thrown, or if this method declares a type compatible with the exception in its `throws` clause.

If the exception is expected inside the method where the paste is performed, the exception will be thrown as if. When the exception is not expected by the method where the paste is performed, Naive will wrap the exception thrown by the base level code into an `java.lang.reflect.UndeclaredThrowableException`. This subclass of `RuntimeException` will be eventually thrown in the method where the method is performed.

6.7.4 Self references reinterpretation

Finally, there is a problem arising with `self` (`this` keyword in Java) and `super` (`super` keyword in Java) references appearing in the copied method. We do not believe that it should part of Naive responsibility to handle it since the reinterpretation of these references can not be done correctly without presupposing the user activities. Therefore even if it has a taste of method call, the copied method is really copied in the object altered and the Naive user have to be aware of it.

6.8 Summary

Here the idea that a macro system performing its macro expansion after the compilation step has been defended. This approach solves the problem of source code availability encountered by previous macro systems. It can allow a relatively wide range of alterations that can help to implement a reflexive system using structural reflection to enable limited behavioral reflective capabilities, especially by providing a sender viewpoint.

Borrowing entities originally studied by the reflection communities: `NaiveClass`, `NaiveMethod` and `NaiveField`, it tries to raise the abstraction offered to the user by providing a logical and contextual representation of the manipulated concepts. These manipulated entities are reexpressed in a source code viewpoint meaning that the bytecode manipulations are hidden to the user.

In Naive a given language semantic can be altered if and only if it is achieved in the runtime level by a set of bytecode instructions, each of them realizing specialized cases of the language construct. In extenso, the possible alterations are: message sent, constructor sent, field read or write, type conversion between objects also known as cast, runtime type identification, exception throwing, local variable read or write, reading or writing a value in an array, getting an array length, creating an array, and returning from a method. The fourth first have been implemented in the prototype.

The altered semantic convey information that need to be passed to the alterator before and after performing the alteration. We distinguished two types of information answering to different needs: static information and dynamic information. By static information we design information that could be provided by class file reasoning. By dynamic information, we design information that could only be provided at runtime. Static information is used to guide the copy paste mechanism and is therefore provided to the user so that before performing a paste it could decide which method need to be pasted at the place of a given language semantic. Static and dynamic information are passed to the pasted method at runtime so that it does not need to assume the place where it is pasted. Information is conveyed by objects arrays and `String` in order to ensure system transparency.

Alterations (or macro expansions in other words) are performed by sending the message `accept` to a `NaiveMethod`. The process is therefore mostly method driven rather than type driven (or class driven in other words). The process by construction ensures termination.

The fact that a method is actually past into another perturbs the semantics of a number of language constructs: local variables, returns, exceptions throwing and self references. Automatic reinterpretation is provided for the first three but the reinterpretation of self references is left to the user. The reinterpretation of local variables ensures that the system is hygienic.

Chapter 7

Naive implementation

"If you wake up feeling no pain, you know you are dead".

Russian proverb

In this chapter, an overview of the implemented prototype is given. First the packages structure is presented. Then the mapping from structural entities to the implementation objects of the prototype, and the mapping of these implementation objects to BCEL objects are discussed. This leads to considering the hierarchy of inliners objects, the classes that are actually performing the copy paste. Finally, the memory policy adopted in Naive is described.

7.1 The package structure

Naive is structured in six packages: `naive.kernel`, `naive.kernel.impl`, `naive.kernel.impl.bytecode`, `naive.kernel.impl.bytecode.generator`, `naive.kernel.impl.inliner`, and `naive.kernel.impl.namespace`. `naive.kernel` contains the user API, the others package contains implementation classes. In particular all classes described in the chapter 6 belongs to that package. `naive.kernel.impl` contains an implementation view of the entities stored in `naive.kernel`. `naive.kernel.impl.bytecode` contains low level classes that either are performing bytecode analysis or are needed to deal with BCEL objects. The `naive.kernel.impl.bytecode.generator` contains classes that generate a bytecode sequence corresponding to a source code action. It is used by `naive.kernel.impl.inliner` which contains the classes that actually perform the bytecode inlining. `naive.kernel.impl.namespace` provides classes that allow to always provide a `java.lang.Class` up to date representing an altered class file.

| User view <code>naive.kernel</code> | Implementor view <code>naive.kernel.impl</code> | |
|--|--|--------------------------------------|
| | Interface | Abstract class |
| <code>NaiveClass</code> | <code>NaiveClassImpl</code> | <code>AbstractNaiveClassImpl</code> |
| <code>NaiveMember</code> | <code>NaiveMemberImpl</code> | <code>none</code> |
| <code>NaiveMethod</code> | <code>NaiveMethodImpl</code> | <code>AbstractNaiveMethodImpl</code> |
| <code>NaiveMethod</code> | <code>NaiveMethod</code> | <code>NaiveMethod</code> |

Table 7.1: User and implementor view of the structural entities: classes corresponding to structural entities, interfaces and abstract classes.

7.2 From Naive API objects to Naive implementation objects

In the early design stage, there was a sharp tension between the idea of offering the user a clean API and to clutter the structural entities: `NaiveClass`, `NaiveMethod` and `NaiveField` with implementation methods. The two needs were equally legitimate.

In the current prototype the problem is solved by providing two views of the structural entities: one is the user API located in `naive.kernel` and the other is an implementation view located in `naive.kernel.impl`. Nevertheless the user API view is more important than the other: as much as possible the use of implementation method should be avoided. Therefore, it was natural to provide a one way ability to switch from view. The message: `getImplLink()` sent to any structural entities of `naive.kernel` will return an appropriate class of `naive.kernel.impl`.

However this solution was still raising some problematic issues especially of visibility of the instance variables located in the structural entities. Therefore, `naive.kernel.impl` only provides a set of interfaces and abstract classes that are implemented by inner classes inside structural entities. The message `getImplLink()` returns an instance of these inner classes that the outer world only know through its implemented interfaces defined in `naive.kernel.impl`. This effectively solves the problem of offering different view of the same structural entities.

Strictly speaking, `naive.kernel.impl` could have only contain interfaces that would have been implemented by inner classes defined in the structural entities. However, this solution in itself was not satisfactory because if the API was not cluttered with implementation methods, the code source of the structural entities would have been mixing the different concerns associated to the different views we wanted to provide. Therefore, abstract classes were provided in `naive.kernel.impl` that implement most of the behavior of the implementor view. This allowed the structural entities of `naive.kernel` to implement only the API needed by the API while the inner classes they contain only add the definitions of a few getter method.

Table 7.1 summarizes the mapping between structural entities, interfaces, and abstract classes. All in all, defining a new implementation method requires to add the new method signature in the interface

| Concept | Reification | | |
|---------------|---|-----------------------------------|-------------------------------|
| | Naive | BCEL | |
| | Implementor view | Static view | Dynamic view |
| Class | NaiveClassImpl mapping through: | JavaClass getAsJavaClass() | ClassGen getAsClassGen() |
| Field | NaiveFieldImpl mapping through: | Field getAsJavaField() | FieldGen getAsFieldGen() |
| Method | NaiveMethodImpl mapping through: | Method getAsJavaMethod() | MethodGen getAsMethodGen() |
| Constant pool | NaiveClassImpl can be queried through: | ConstantPool getConstantPool() | ConstantPoolGen() |

Table 7.2: [Mapping from Naive implementor view to BCEL objects .

The constant pool is not a structural entities reified by Naive. But at a bytecode level, it is a class file scoped data structure. It is therefore natural to associate it with `NaiveClassImpl`. The mapping methods are defined on the Naive object (second column).

objects named in the second column of table 7.1 so that the new method becomes available to the outside world, and implement the method in the abstract classes identified in the third column of the table.

In this section, we have seen how the naive implementation decouples the user API offered in the package `naive.kernel` from an implementation view. This implementation view can be queried by sending the message `getImplLink()`. It will return an inner class defined inside the structural entities considered whose type is an interface declared in `naive.kernel.impl`. Additional abstract classes in `naive.kernel.impl` allows a true separation of concerns in the source code between the user view offered in `naive.kernel` and an implementor view offered thanks `naive.kernel.impl`.

7.3 From Naive implementation objects to BCEL objects

Since the structural entities proposed in `naive.kernel` are common to the bytecode level and to the source level as stated in 6.3, it is not surprising that BCEL offers some reification of them. There is however an important difference distinction to be done between the structural entities offered in `naive.kernel` and the reification of BCEL: the latter proposes objects with a bytecode concern while the former are operating at a level closer to source code.

Naturally the question arises to map the structural entities, or more precisely the implementation view of the structural entities, to their counterparts in BCEL. As stated in 4.3, BCEL provides two views of a class file: a static view and a dynamic view. The static view is the one provided by BCEL after parsing the class file. It can be used to provide bytecode logical and bytecode contextual information of a given class file. It is not possible to alter a class file using this static view. Alterations require to

switch to the dynamic view. Usually the classes of BCEL knows how to translate themselves into their dynamic counterparts and vice et versa.

However two problems appears there: first the translation process of BCEL between static and dynamic level have some side effects. For example, using the `getMethod()` more than once on a `MethodGen` to switch from a dynamic level to a static one, may cause the class file containing the method to have more than one `Code` attribute for that method. This causes the altered class file to be rejected by the bytecode verifier when a virtual machine tries to use it. The second problem is that once a reification is altered, the altered reification should be kept in memory until the user require the altered class file to be dump either in memory, either on disk. As described on table 7.2 Naive implementation objects define secure mapping methods that will provide their counterparts in BCEL taking all these constraints into account.

It is however interesting to note that the second problem: "keeping an altered reification in memory until the user either dump the related class file in memory or on disk" is the reason why Naive structural entities are instantiated by factory methods. These factory methods are relaying upon class members to keep track of the modified entities. A given `NaiveField`, `NaiveMethod` or `NaiveMethod` can register itself as being altered with regards to these class variables, by sending itself the message `addToPendingModificationsSet()` defined in the implementor view. Unregistration is achieved with the message `removeFromPendingModificationsSet()`. This frees the user to distinguishing when he has to reinstantiate a structural entities (using `new`) and when he has to reuse a previously altered reification.

This is however not enough: whatever the structural entities manipulated, only class maps to class file and are therefore dump in memory or written to disk. In clear, it means that `NaiveMethod` and `NaiveField` should have a kind of link with their `NaiveClass` so that the latter can query them for alterations before dumping itself in memory or on disk. To achieve this, `NaiveMethod` and `NaiveField` are defining inner classes implementing the interface `ClassAlterator`. When they perform an alteration, they register their related `ClassAlterator` on their declaring class file using the method `addAlterator(ClassAlterator)` of `AbstractNaiveClassImpl`.

In this section, we have seen that BCEL provides counterparts of the Naive structural entities, with however a bytecode concern. These counterparts exist in two view static and dynamic. The former is result of BCEL parsing and only allow a static description. To modify the described class file, switching to the dynamic view is needed. However, how the level switch should be achieved largely depends on the previous use of the different reification provided by BCEL. Naive implementor view provides safe translation methods. The necessity to kept altered structural entities in memory is the reason of being of factory methods. Altered `NaiveField` and `NaiveMethod` registered themselves on their declaring `NaiveClass` by sending the message `addAlterator(ClassAlterator)` on the implementor view using `ClassAlterator`.

7.4 The inliner hierarchy

The package `naive.kernel.impl.inliner` is at the heart of naive abilities: it provides the copy and paste mechanism. As shown on 7.1, it is structured in an inheritance tree that use subclassing at its roots to incrementally increase the copy paste features and at the bottom to specialize the different classes to perform specific alterations. First we will describe how to use them before discussing their relationship with `NaiveMethod`. Then the layering structure will be considered: one layer to take care of the stack, another to pass and return information to the pasted code.

7.4.1 Using the inliners

Inliners are typically instantiated by taking three arguments: the method in which the copy/paste takes place, a set of bytecode instructions belonging to the former method body, and a method that should be pasted in place of each bytecode instruction contain in the set. Once instantiated, any `MethodAlterator` can answer to the message `alter` that will actually perform the alteration. In fact, `alter` delegates the alteration work to the abstract method `proceed` declared in `MethodAlterator`.

7.4.2 The inliners and `NaiveMethodImpl`

As stated in 7.4.1, inliner classes are typically instantiated by taking three arguments: the method in which the copy/paste takes place, a set of bytecode instructions belonging to the former method body, and a method that should be pasted in place of each bytecode instruction contain in the set. It is the role of the message `accept` of `NaiveMethod` to instantiate the right inliner with the right number of arguments.

In fact, this task is not simple as it may seem to. The `accept` method has to enumerate the list of bytecode instruction on which the semantic alterator maybe interested in. For example, `accept(MethodInvocationAlterator)` has to enumerate all bytecode instructions invoking a method in the method it describes. Furthermore, the `accept` message has then to query its semantic alterator with the appropriate static information to get the method to inline. In the previous example, it means that that `accept(MethodInvocationAlterator)` has to analyze the enumerated bytecode instructions so as to send the message `getMethodToInlineOnMethodInvocation(String)` taking the qualified name of the method invoked by the bytecode instruction on its `MethodInvocationAlterator`. Moreover the alterator implemented by the user then return either the qualified name of the method paste or null if nothing has to be inlined. Clearly there is absolutely nothing in this process that grants that the alterator will always return the same method to be pasted.

This means that the `accept` method has to group the different results returned by the alterator in pairs where the first element is the set of bytecode instructions, and the second element is the method that

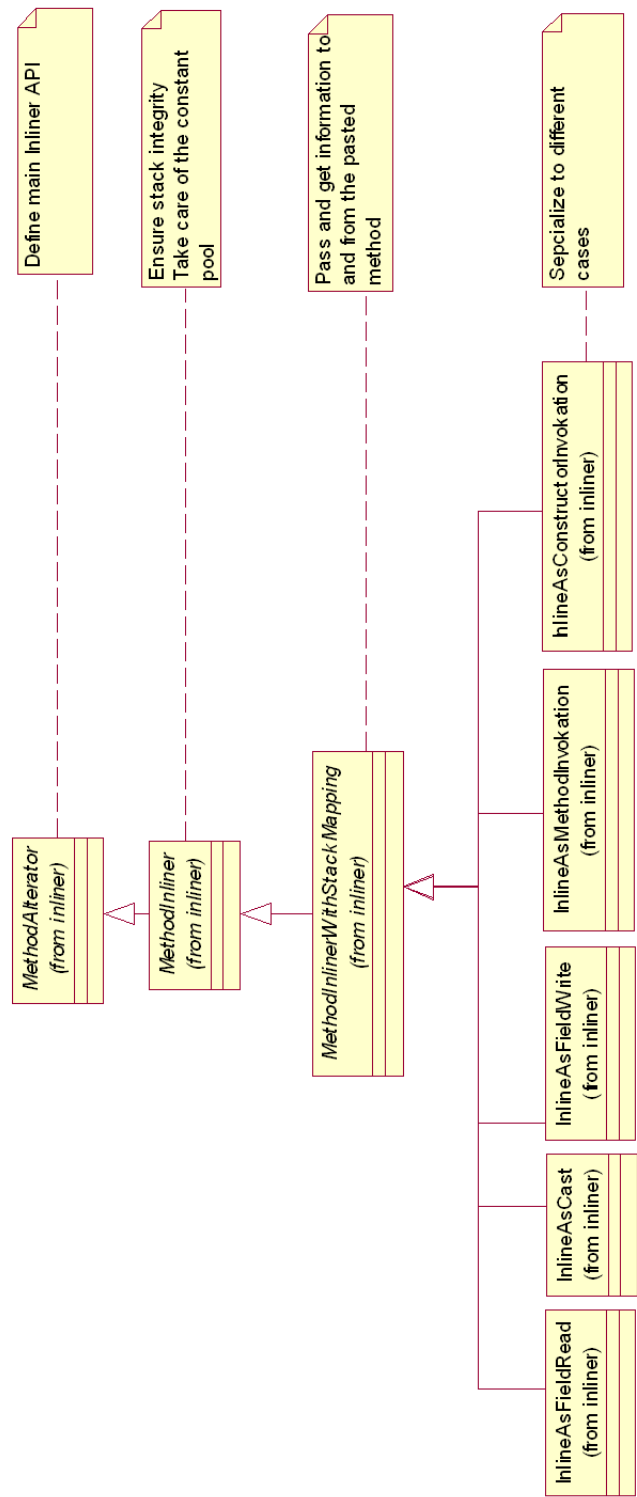


Figure 7.1: The inliner inheritance tree

should be pasted, and instantiate as much inliners than it has made different pairs. Since this process is relatively complex, close to the bytecode level, `accept` of `NaiveMethod` in fact are delegating their work to `accept` of `NaiveMethodImpl` that will actually perform all the task describe above.

In this section, the relationship of the inliners and `NaiveMethod` were studied. The method `accept` proceeds by delegation to `NaiveMethodImpl` that instantiates the inliner on behalf of the result returned by the alterator.

7.4.3 Taking care of the stack depth

As stated in 2.7.2, the stack depth has to remain the same whatever the path taken. This means in particular that the pasted method should not perturb the stack. To achieve this, `MethodInliner` translates the method body of the pasted method into a bytecode instruction sequence who last instruction has a stack depth just big enough to hold the returned value and who consumes the same number of slots on the stack than the replaced bytecode instruction. The idea there is to that the value returned by the pasted method should be pushed on the stack as if this value was pushed on the stack by the replaced instruction. This process is a bytecode to bytecode translation.

While it is dangerous to remove an instruction because other branch instructions may target it, BCEL when it dumps a class file in memory or on disk removes all NOP instructions. Therefore, the current prototype is adding a NOP instruction that just do nothing as the end of the pasted method. Then all return instructions are rewritten as jump to this NOP instruction. To equalize the stack depth, instructions are inserted before the old return (in other words before the new branch instructions targeting the newly added NOP).

Moreover the try catch statements should be translated too. This is a stack caring task because catching an exception may entail a branch to the exception handler, and as said previously the stack depth should be the same regardless of the stack taken. Try catch are stored the `Exception` attribute where instructions offsets defines from what to which bytecode instructions an exception should be catch. Since, we are copying a method into another, the `Exception` attribute of the pasted should be translated into `Exception` attribute of the method where the inlining is taking place.

While this is not a stack caring task, `MethodInliner` is also adding the needed entries to the constant pool of the class where the past is performed and rewrites the bytecode instruction of the pasted methods with regards to the new index in the constant pool of the class where the past is performed. This is mostly a direct use of BCEL objects. The constant pool is updated by the method `copyConstantPoolResources` declared in `MethodInliner`.

This whole process is managed by the `getInstructionsToInsert()` declared by `MethodInliner` that returns the translated bytecode instruction sequence. First of all, the technique used to equalize the stack depth will be presented before focusing on how returns and try/catch are rewritten.

Equalizing the stack depth

The general idea to equalize the stack depth is to use the `CachedStackDepthComputer` object that computes the stack depth at a given instruction. Then depending on the difference the stack can be equalized by inserting either `POP` instructions that will pop the top operand stack value or pushing some zeros on the stack using an `iconst`. As stated in 2.7.1, the virtual machine is a typed interpreter where long and double values occupies two slots on the stack. These slots should never been manipulated separately otherwise the bytecode verifier will complaint. Therefore, using `POP` instructions that pop a slot at a time can lead the bytecode verifier to reject the altered class file. To resolve this problem, we only uses `POP2` that pops two slots of the stack in one time. When the number of slots to be removed is not a multiple of two, we push an additional zero on the stack.

This idea is exploited in the `alignStack`, and `translateReturns` of `MethodInliner`. The former ensures that the stack depth is zero at the end of the instructions sequences while the latter translate returns. `alignStack` is a simple implementation of the idea described below. `getInstructionsToInsert()` will call `removeLastReturn`. This method will store the returned value in a local variable and adjust the stack depth by calling `alignStack` then push the returned value from the local variable on the stack.

Rewriting returns

The rewriting of all returns except the last one is performed by the `translateReturns()` method declared by `MethodInliner`. For each return instruction encountered, the value returned is stored in a local variable then some instructions are inserted to equalize the stack depth. Then the value returned is pushed on the stack. Finally a branch instruction to the last instruction is written in place of the return instruction.

Rewriting try/catch

Try/catch are rewritten by the method `translateExceptionHandler` of `MethodInliner`. There is no surprise in the algorithm used. It mostly relies directly upon BCEL objects. The idea is to use the `Exception` attribute of the pasted method to localize the different bytecode instructions that define an exception handler of a given type (in extenso the instruction where an exception starts to be expected, the instruction where the exception starts to be not expected anymore, and the instruction to jump to when an exception matching the given type is given). Once these instructions are known, they can be mapped in the new instruction sequence corresponding to the pasted method. Then, the type of the handled exception and the mapped instructions can be used to add a new exception handler in the `Exception` attribute of the method where the pasting is being performed.

In this section, how the class `MethodInliner` is ensuring the stack depth has been presented. The process is a bytecode to bytecode translation, rewriting return and try/catch. It is performed

by `getInstructionsToInsert()`. The stack depth is equalized thanks to `alignStack`, and `translateReturns` of `MethodInliner`, by inserting either zero with `iconst` instructions or either popping values from the stack using `POP2` instructions. `translateExceptionHandler` of `MethodInliner` rewrites the different `try/catch`.

7.4.4 Passing and returning information to the pasted method

While taking care of the stack depth was already a challenge in itself, passing and returning information from the pasted method, is quite interesting too. This role is achieved by the `MethodInlinerWithStackMapping` class. While first experiments at the beginning of this work, exhibit - unreliable - information passing without further alterations to the bytecode, it becomes quickly clear that the insertion of additional bytecode instructions to pass and retrieve information from the pasted method was needed.

The reason behind this need can be roughly grouped into two points:

- wrapping/unwrapping primitive types values,
- passing information that is not directly available in bytecode,

The first point roughly corresponds to the idea that the different values consumed on the stack by the replaced instruction should be provided as arguments to the pasted method and that the result produced by the method should be put on the stack as the result produced by the replaced instruction if any.

The second point might be less obvious. In 6.5.3 the need to feed the pasted method with static and dynamic information has been established. First, static information is usually not directly available in the bytecode. For example, there is no "magic" way to retrieve the qualified name of the method that is currently being executed in the current thread. Secondly, there are some dynamic information that is not on the stack but that need to be passed to the pasted method. For example, the current instance of the object is typically not available to the `CHECKCAST` instruction that performs a type conversion between objects.

This establishes the need for inserting some bytecode to pass the information to the pasted method and after to retrieve information from the pasted method. The former is know as the prologue and the latter as the epilogue. The prologue is generated by the `getPrologueToInstructionsToInsertOn()` declared in `MethodInlinerWithStackMapping`, the epilogue is generated by `getEpilogueToInstructionsToInsertOn()` declared in `MethodInlinerWithStackMapping`: both methods rely heavily upon the classes of the package `naive.kernel.impl.bytecode.generator`.

Passing information to the copied code: generating and wrapping

The generation of the prologue is probably the most complicated task. To achieve it, we took a pragmatic approach of the problem. `getPrologueToInstructionsToInsertOn()` has to map the stack put it in an array that may contain another array along with other information provided by static analysis. To achieve it, we took a very pragmatic approach. Considering that it was highly desirable to hide the task complexity, `getPrologueToInstructionsToInsertOn()` is a relatively long method that acknowledges the different kind of information to be passed described in table 6.2 by querying abstract and non abstract method various methods refined by its subclass. Still, this solution is akin to assuming all the need of all the different subclasses and we are strongly convinced that it is a *bad* solution. The next paragraphs will discuss the method queried and their effects on the array passed to the pasted method.

`getPrologueToInstructionsToInsertOn()` queries the `getStaticInfo()` method of `MethodInlinerWithStackMapping` for the static info to be passed to the pasted method. Only `String` can be passed as static info. This information will be passed to the pasted method as the first elements in the object array it takes in argument.

Depending on the result returned by the method `isSelfNeededToBeAddedToTheArgumentArray()`, it will add a reference to the instance object¹ running the method as next element, right after the static information in the object array taken in argument of the pasted method.

If the abstract method `isNullNeededBeforeStackMapping()` returns true, it will add a null as the next element of the object array taken in argument of the pasted method. This method is used while altering a static field read (`InlineAsFieldRead` encounters a `GETSTATIC` instruction), static field write (`InlineAsFieldWrite` encounters a `PUTSTATIC` instruction), or altering a static method invocation (`InlineAsMethodInvocation` encounters a `INVOKESTATIC` instruction). The point is that in all the previous cases, table 6.2 requires to pass either the instance owning the field, or the instance on which the method was invoked which does not make sense in these cases.

Finally the stack consumed by the replaced will be map into the array with an appropriate wrapping if necessary. The mapping also applies to fill the last element of the array with another object array if the method `isLastElementInArrayAnEnclosedArray()` returns true. Typically this method only returns true for `InlineAsConstructorInvocation` and `InlineAsMethodInvocation` classes. The types and the length of the objects to be stored in that enclosed array is retrieved through the abstract method `getLastPostionArrayTypes()`. The stack mapping performed by `getPrologueToInstructionsToInsertOn()` adjusts the enclosing array size by considering the length of the array returned by `getLastPostionArrayTypes()` and the number of values consumed on the stack by the replaced instructions. The mapping is further complicated by the fact that the data at the bottom of the stack corresponds to the greater indexes in the array of objects

¹it will generate a null if the method where the paste is performed is a class method.

taken in arguments of the pasted method.

In this subsection, we have seen that the bytecode sequence that passed information to the pasted method is generated by `getPrologueToInstructionsToInsertOn()`. It relies upon collaboration with various other methods to pass information that is not directly available and to map the information directly available into the types described in table 6.2. In fact, the collaboration with the different methods parameterizes the prologue generation. In other words, `getPrologueToInstructionsToInsertOn()` more or less assumes the shape of the prologue needed by all the subclasses of `MethodInlinerWithStackMapping`, and finally the shape of the information to be passed to the pasted method.

Getting information from the copied code: dewrapping

Fortunately getting information from the pasted is much less painful. This task is performed by the bytecode instructions sequence generated by `getEpilogueToInstructionsToInsertOn()` declared in `MethodInlinerWithStackMapping`. The result to be unwrapped is typically dependant of the instruction replaced and it is retrieved using the abstract method `typeToUnWrap()`. If this method returns "void" any results generated by the pasted method will be discarded. If necessary, an unwrapping will be performed.

Moreover, the generated unwrapping sequence takes care of mapping the exceptions. The replaced instruction is analyzed to infer which exceptions this instruction is allowed to throw. Then if the pasted method throws an exception whose type does not match to one of these allowed exceptions, the exceptions thrown will be wrapped into an `UndeclaredThrowableException` (from the package `java.lang.reflect`). This is done by generating at a bytecode level, a sequence of try catch blocks. The upper catches trap exceptions corresponding to the exceptions that the replaced instruction may have thrown and rethrow them as if. In source code, it will look like:

```
catch(AnExceptionType anException) {
    throw anException;
}
```

The last catch any exception and wraps them into `UndeclaredThrowableException` before rethrowing them. In source code, it will look like:

```
catch(Exception anException) {
    throw new
        UndeclaredThrowableException((Throwable) anException);
}
```

In this section, we have seen that the bytecode unwrapping sequence is generated by `getEpilogueToInstructionsToInsertOn()`. An unwrapping is performed both on the returned value and on the exceptions thrown by the pasted method.

7.5 Memory policy

Most objects in Naive, are instantiated through factory methods. Factory methods allows to do some caching. Cached values are stored into `SoftReference` (package `java.lang.ref`). Soft reference hold a reference on an object, which is cleared at the discretion of the garbage collector in response to memory demand.

In the same way, the instance variables that can be recomputed are hold in Naive objects through `SoftReference`. When an object wants to access such a field, it uses the related getter method. If the related getter determines that the reference has been cleared, it will reinitialize the reference kept in the field and returns the new computed value. Usually the values used to reinitialize the reference is performed by a method named `init` plus the name of the related field.

This process can be though as a cooperation between the garbage collector and lazy initialization. It is aimed is to find a compromise between caching and the cost of instantiating objects that gracefully adapt and profits at maximum of the memory available at runtime.

In this section we have seen that Naive is using lazy initialization, factory methods, and `SoftReference` to enable memory sensitive caching of instantiated objects.

7.6 Summary

Naive is structured in six packages each one with a sperate concern. The package `naive.kernel.impl` contains an implementor view of structural entities. A structural entities can be queried for its implementor view by sending the message `getImplLink()`. This implementor view provides safe getter method to map into BCEL counterparts reification. The copy paste is actually performed by the subclasses of `MethodAlterator` located in the package `naive.kernel.impl.inliner`. `MethodAlterator` mostly defines the public interface of the inliner hierarchy. Its first subclass: `MethodInliner` ensures the stack and constant pool integrity. Its subclass: `MethodInliner-WithStackMapping` adds the ability to pass and retrieve information to and from the pasted method. The memory policy in Naive tries to use a caching mechanism that gracefully and automatically adapt to the available memory.

Chapter 8

Future works

This chapter presents the different extensions that could improve, according to us, the current prototype. The most obvious one is to push the concept to its limits and implement all the alterations that it allows to perform. But it may also be possible, by analyzing the alterations it does not enable to revisit the concept. Finally, the framework can be refactored to remove the assumptions it is making on the user utilization of the prototype.

8.1 Offering more alterations

According to the alterations possibilities presented in table 6.1, it is remarkable that less than 40 % are implemented in the current prototype. While this figure was established by a quick look on table 6.1, it may even be decreased if one makes the difference between the alterations directly concerning self like a message sent to the `this` keyword and `super` keyword and the other one.

8.2 Revisiting the concept

In 6.1, we stated that Naive principle is: from an implementor viewpoint: "replacing the bytecode instruction corresponding to one of this language constructions by an arbitrary sequence of bytecode" and from an user viewpoint: "inlining an externally user provided compiled method in place of a given semantic appearing in a given method". Nevertheless, a number of alterations that can not be performed by Naive enumerated in 6.4.2 could be performed if the principles were not centered on *the bytecode instruction* that maps to the *given semantic* but on *a sequence of instructions*.

Achieving this, will require to provide the user with another structural entities, that does no exist in the reflection community, representing a sequence of instructions. This entity could be close to the structural entities that describe a method. It will in particular allow to alter language semantic that

results from a collaboration of bytecode instructions.

8.3 Opening the framework

Table 6.2 present the information passed to the `SemanticAlterator` and to the pasted method. This information is hard coded inside the prototype: a given user can not parameterize the information is want to be passed either to the `SemanticAlterator`, either to the pasted method. In clear, this design tries to be universal, at the price of efficiency since it is passing more information that a given user may need. We never believe very much in object-oriented designs that pretends to be universal with regards to their users. A much better approach will be to let the user specifies the information that it needs to be passed. Applying a pattern like the Command pattern may be a solution.

8.4 Summary

In this chapter, we have seen that the prototype could be improved by implementing all the alterations that the concept allows to perform. The concept in itself can be revisited by instead of replacing one bytecode instruction, replacing a sequence. This would lead to introduce a new structural entities representing a sequence of bytecode instructions. Finally, the framework could be opened up by letting the user parameterize the information he wants to be passed to the `SemanticAlterator` and to the pasted method.

Chapter 9

Conclusion

In this dissertation, we presented a new portable way of performing Java semantic alteration by runtime level macro. While the described prototype suffers from severe limitations and as described in chapter 8 more work is still need to reach full maturity, we believe that it gives serious hints that this approach is feasible and need to be explored.

The main contribution of this thesis is probably not into the description of yet another obscure system manipulating compiled forms of classes, but more in an attempt to merge the concepts developed by two communities: macro and reflection worlds. Both allows to perform semantic alterations, both may benefit of each other. On these basis, the constraints arising from the Java language leads us to propose runtime level macros to achieve portability.

Bibliography

- [Bec99] Kent Beck. *Extreme programming explained: embrace change*. Addison Wesley, 1999. 41
- [BGW93] D.G. Bobrow, R.G. Gabriel, and J.L. White. Object oriented programming: the clos perspective. *chapter CLOS in context The shape of the design space, in Object oriented programming MIT Press*, 1993. 34
- [BLS92] D. Batory, B. Lofaso, and Y. Smaragdakis. Jts: Tools for implementing domain-specific languages. *Proceedings of 5'th International Conference on Software Reuse, IEE*, 1992. 24
- [BS99] N. Bouraqadi-Sadani. Java and reflexion. Technical report, Ecole des Mines de Nantes, 1999. 34, 36
- [BS00] Claus "Brabrand and Michael I." Schwartzbach. "growing languages with metamorphic syntax macros". sep 2000. 24
- [Caz98] W. Cazzola. Evaluation of object-oriented reflective models. *ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98), 12th European Conference on Object-Oriented Programming (ECOOP'98)*, available at <http://www.disi.unige.it/person/CazzolaW/>, 1998.
- [CC01] G.A. Cohen and J.S. Chase. An architecture for safe bytecode insertion. *Duke University, submitted for publication available at http://www.cs.duke.edu/ari/joie/*, 2001. 30, 31
- [CCK98] G.A. Cohen, J. Chase, and D Kaminsky. Automatic program transformation with joie. *in Proceedings of the 1998 USENIX Annual Technical Symposium*, available at <http://www.cs.duke.edu/ari/joie/>, 1998. 30
- [Chi97] S. Chiba. Implementation techniques for efficient reflective languages. Technical Report Technical Report 97-06, Department of Information Science, University of Tokyo, 1997. 35
- [Chi98] Shigeru Chiba. Macro processing in object-oriented languages. *Proc. of Technology of Object-Oriented Languages and Systems (TOOLS Pacific '98) IEEE Press*, November 1998. 24, 42

- [Chi00] S. Chiba. Load-time structural reflection in java. *European Conference on Object-Oriented Programming (ECOOP'00)*, available at <http://www.hlla.is.tsukuba.ac.jp/chiba/Javassist/index.html>, 2000. 34, 35, 38, 102
- [Dah99] M. Dahm. Byte code engineering. in *Proceedings JIT'99, Berlin*, available at <http://bcel.sourceforge.net/documentation.html>, 1999. 29
- [Dah01] M. Dahm. Byte code engineering with the bcel api. *Technical Report B-17-98, Berlin*, available at <http://bcel.sourceforge.net/documentation.html>, 2001. 29
- [GM96] J. Gosling and H. McGilton. The java language environment white paper. *Sun Microsystems*, available at <http://Java.sun.com/docs/white/>, 1996. 11, 28
- [Gol97] M. Golm. Design and implementation of a meta architecture for java. available at <http://www4.informatik.uni-erlangen.de/Projects/PM/Java/>, Leipzig Germany, 1997. 35, 102
- [GS00] J. Gosling and B. Steele. *The Java Language Specification, Second Edition, the Java Series*, volume available at <http://Java.sun.com/docs/books/jls/>. Sun Microsystems, Inc, 2000. 101
- [KCR] R. Kelsey, W. Clinger, and J. (eds. Rees. Revised5 report on the algorithmic language scheme. *Computation*, vol. 11, no. 1, september, 1998 and *acm sigplan notices*, vol. 33, no. 9, october, 1998. available at <http://www.schemers.org/Documents/Standards/>. 24
- [KDRB91] G. Kiczales, J. Des Rivires, and D.G. Bobrow. *The art of the metaobject protocol*. MIT Press, 1991. 9, 33, 34, 35, 42
- [Ker81] Brian W. with contributions of P. J. Plauger Kernighan. *Software Tools in Pascal*. Addison-Wesley, 1981. 24
- [KFFD86] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. *Proceedings of the ACM Conference on LISP and Functional Programming*, 1986.
- [KH97] R. Keller and U. Hlzle. Binary component adaptation. Technical report trcs97-20, Computer Science Department, University of California, Santa Barbara, available at <http://www.cs.ucsb.edu/oocsb/papers/TRCS97-20.html>, 1997. 28
- [Knu01] Donald E. Knuth. *Computers and Typesetting*, volume A-E. Addison Wesley, 2001. 24
- [KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Inc, 1978. 24
- [Lee96] H. Lee. Bit: bytecode instrumenting tool. available at <http://www.cs.colorado.edu/han-lee/BIT/index.html>, University of Washington, 1996. 28

- [LS95] J.R. Larus and E. Schnarr. Eel: Machine-independent executable editing. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1995.
- [LY97] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, volume available at <http://Java.sun.com/docs/books/vmspec/>. Addison-Wesley, 1997. 11, 12, 14, 15, 17, 29, 47, 48, 49
- [Mae87] P. Maes. *Computational reflection*. PhD thesis, Artificial intelligence laboratory, Vrije Universiteit, Brussel, Belgium, 1987.
- [OCGB98] A. Oliva, I. Calciolari Garcia, and L.E. Buzato. The reflective architecture of guarana. available at <http://www.dcc.unicamp.br/oliva/guarana/index.html>, 1998. 35, 102
- [OT00] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. *In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, available at <http://www.research.ibm.com/hyperspace/Papers/index.htm>, 2000. 29
- [RVL⁺97] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of win32/intel executables using etch. *In Proceedings of the 1997 USENIX NT Conference*, 1997.
- [SE94] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. *In Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994. 28
- [SGGB99] E.G. Sirer, R. Grimm, A.J. Gregory, and B.N. Bershad. Design and implementation of a distributed virtual machine for networked computers. *in Proceedings of the Seventeenth Symposium on Operating Systems Principles, pages 202–216, Kiawah Island, South Carolina*, available at <http://www.cs.washington.edu/homes/egs/papers/>, 1999. 28
- [Sha96] Andrew Shalit. *Dylan Reference Manual, The: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, 1996. 24
- [Smi84] B.C. Smith. Reflection and semantics in lisp. *in Proceedings of the 14th Annual ACM Symposium on principles of programming languages, POPL'84*, pages 23–25, 1984. 33
- [Smi90] B.C. Smith. What do you mean meta? *In Workshop on reflection and metalevel architectures in OO programming, ECOOP/OOPSLA'90, Ottawa, Ontario Canada*, 1990. 33, 42
- [Str97] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition edition, 1997.

- [Tan00] E. Tanter. Reflex a reflective system for java: Application to flexible resource management in mobile object systems. available at <http://www.dcc.uchile.cl/~etanter/Reflex/>, Universidad de Chile, Chile, 2000. 34, 35, 102
- [Tat99] M. Tatsubori. An extension mechanism for the java language. available at <http://www.hlla.is.tsukuba.ac.jp/~mich/openjava/>, Tsukuba, Japan, 1999. 35, 102
- [TCKI00] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano. Openjava: A class-based macro system for java. *Springer-Verlag, Lecture Notes in Computer Science*, Reflection and Software Engineering(1826):117–133, 2000. 24, 25, 42
- [Vay97] J. Vayssire. Parallel and distributed programming in java: conception and implementation of java//. Technical report, DEA report in French, Sophia-Antipolis, France, 1997. 35, 102
- [WS00] I. Welch and R. Stroud. Kava - a reflective java based on bytecode rewriting. *Lecture Notes in Computer Science 1826 from Springer-Verlag (2000)*, available at <http://www.cs.ncl.ac.uk/research/dependability/reflection/>, 2000. 35, 37, 38, 102
- [WS01] I. Welch and R. Stroud. Kava - a reflective java based on bytecode rewriting. *Proceedings of USENIX Conference on Object-Oriented Technology*, available at <http://www.cs.ncl.ac.uk/research/dependability/reflection/>, 2001. 37, 53
- [Zim96] C. Zimmermann. *Advances in Object-Oriented Metalevel architectures and reflection, MOPs and what the Fuzz is All About*. CRC Press, 1996.

Appendix A

Some examples

The purpose of this appendix is to document by an example each alterations that is currently offered by the prototype. Each example will copied a method that will achieve the same behavior than the bytecode instruction replaced, or in less clear terms, than the altered semantics. For that purpose, the copied methods will heavily relay on Java instrospection features. However, introspection is not the only solution and alternative implementations are possible. A short description of these alternatives will be given but only with the field read example.

A.1 General shape of the examples

In the following examples, the class `Example` will first modify the method: `method()` declared by `ModifiedClass`. We will assume that `ModifiedClass` is stored in the working directory and that the working directory is included in the classpath. `Example` will perform the alteration by using an object implementing a subinterface of `SemanticAlterator` inlining the unique method of the class `MethodBodyContainer` in place of the semantic that the subinterface of `SemanticAlterator` allows to replace. `ModifiedClass` will be saved to disk. Finally `Example` will try to send the message `method` to `ModifiedClass`.

A.2 Field read alterations

In the example involving the alteration of reading a field, we will add a field: `i` to the `ModifiedClass` initialized to zero. Then, as shown on figure A.2, the `method()` of `ModifiedClass` will just return the value of `i`. The purpose of the inlined code will just be to return the value of `i` as well.

The class `Example` source code can be found in figure A.2. `Example` uses a `FieldReadAlterator`-`Object` to perform the alteration: as shown on A.2, it is no more than a simple implementation of the `FieldReadAlterator`. The method `getMethodToInlineOnFieldRead(String)` will

```
package naive.thesisexamples.fieldread;

public class ModifiedClass {
    public int i = 0;
    public int method() {
        return this.i;
    }
}
```

Figure A.1: Field read alteration: ModifiedClass class

be feed with the static information: in that case the qualified name of the field that will be read. Whatever the field the implementation in figure A.2 gives the method `fieldReadReplacer()` declared by `MethodBodyContainer` to inline in place of the field read.

This method: `fieldReadReplacer(Object[])` takes an array in argument containing the dynamic information describing what the base level code wants to achieve: here a field read. The first element of the array is the qualified name of the field given as a `String`, the second element is the qualified name of the method reading the field, the third element is the instance of the object reading the field and finally the fourth element is the instance on which the field is read.

A.3 Some alternate implementations taking a field read alterations as example

The implementation described in figure A.2 just perform the field read by introspection. It may encounter various error states: this leads to throw the exceptions `NoSuchFieldException`, `IllegalAccessException` and `ClassNotFoundException`. These exceptions are not excepted by the base level: `ModifiedClass` is not prepared to handle any of them. If one of the previously listed exceptions is thrown by the inlined, it will be wrapped in an `UndeclaredThrowableException` which is a subclass of `RuntimeException`. This will not perturb the `ModifiedClass.method()` since `RuntimeException` may occur at any time any place in a Java program.

The result run in a console hopefully does not encounter any error and displays a 0 in output. We believe that the implementation proposed in figure A.2 is the more general but alternative possible. For example, instead of the proposed implementation, one may write:

```
public Object fieldReadReplacer(Object[] args) {
    ModifiedClass instance =(ModifiedClass)args[3];
    return new Integer(instance.i);
}
```

```
package naive.thesisexamples.fieldread;
import naive.kernel.NaiveMethod;
import java.io.IOException;

public class Example {
    public static void main(Java.lang.String[] args) {
        //do the modifications
        modify();
        //execute the modified code
        ModifiedClass m = new ModifiedClass();
        //display the result
        System.out.println(m.method());
    }
    public static void modify() {
        //get the metaobject describing a method
        NaiveMethod method = NaiveMethod.getNaiveMethod("
            naive.thesisexamples.fieldread.ModifiedClass.method()");
        //perform the alteration
        method.accept(new FieldReadAlteratorObject());
        //write the modified class file on disk
        try {
            method.getDeclaringClass().save();
        } catch(IOException exp) {
            System.err.println(exp);
            System.exit(-1);
        }
    }
}
```

Figure A.2: Field read alteration: Example class

```

package naive.thesisexamples.fieldread;
import naive.kernel.FieldReadAlterator;

public class FieldReadAlteratorObject implements FieldReadAlterator {
    public String getMethodToInlineOnFieldRead(
        String inReplacementOfField) {
        return "naive.thesisexamples.fieldread.MethodBodyContainer.
            fieldReadReplacer(Object[])";
    }
}

```

Figure A.3: Field read alteration: FieldReadAlteratorObject class

```

}

```

The latter is however much less generic. Actually the inlined method: `: fieldReadReplacer` is now making the hypothesis that it will always be inlined in place of `ModifiedClass.i`. The implementation that used introspection in figure A.2 does not make any assumption.

A.4 Field write alterations

In the example involving the alteration of writing a value to a field, we will add a field: `i` to the `ModifiedClass` initialized to zero. Then, as shown on figure A.4, the method `()` of `ModifiedClass` will set its value to 1 and return value `i`. The purpose of the inlined code will just be to write the new value of `i`: 1 as well.

The class `Example` source code can be found in figure A.4. `Example` uses a `FieldWriteAlteratorObject` to perform the alteration: as shown on A.4, it is no more than a simple implementation of the `FieldWriteAlterator`. The method `getMethodToInlineOnWriteRead(String)` will be feed with the static information: in that case the qualified name of the field that will be written. Whatever the field the implementation in figure A.4 gives the method `fieldReadReplacer()` declared by `MethodBodyContainer` to inline in place of the field written.

This method: `fieldWriteReplacer()` takes an array in argument containing the dynamic information describing what the base level code wants to achieve: here writing a value to a field. The first element of the array is the qualified name of the field given as a `String`, the second element is the qualified name of the method writing the field, the third element is the instance of the object writing the field, the fourth element is the instance on which the field is written, and finally the fifth element is an object containing the new value of the field. If the new value is a primitive type, it will be wrapped

```
package naive.thesisexamples.fieldread;
import java.lang.reflect.*;
import naive.kernel.NaiveField;

public class MethodBodyContainer {
    public Object fieldReadReplacer(Object[] args)
        throws  NoSuchFieldException,
                IllegalAccessException,
                ClassNotFoundException {
        /*args contains
         [ qualifiedFieldNameAsString
           qualifiedMethodReadingFieldAsString
           InstanceOfSenderAsObject
           InstanceOfReceiverAsObject]
        */
        String qualifiedFieldName = (String)args[0];
        Object fieldOwner = args[3];
        //get the Naive meta obj
        NaiveField naiveField =
            NaiveField.getNaiveField(qualifiedFieldName);
        //maps from Naive meta object to standard Java meta object
        /* Field langField=naiveField.getAsLangOnDisk()
        can not be used because the class loader may be different*/
        Field langField = Class.forName( naiveField.
            getDeclaringClass().getCanonicalName())
            .getField(naiveField.getSimpleName());
        //load and return the field value
        //through standard Java meta object
        return langField.get(fieldOwner);
    }
}
```

Figure A.4: Field read alteration: MethodBodyContainer class

```
package naive.thexsisexamples.fieldwrite;

public class ModifiedClass {
    public int i = 0;
    public int method() {
        this.i = 1;
        return this.i;
    }
}
```

Figure A.5: Field write alteration: ModifiedClass class

into an appropriate object.

The implementation described in figure A.4 just perform the writing of the field by introspection. As expected Example run in a console displays a 1 in output.

A.5 Cast alterations

Java is strongly typed: classes defines types. While the operation of converting a class, respectively a type, into one of its superclass or superinterface, respectively into one of its supertype is performed automatically, in all other cases, the programmer has explicitly to request the conversion. This explicit conversion request is known as casting. Naive offers the ability to alter the semantic of cast operations between two objects¹.

In the example involving the alteration of cast, `ModifiedClass.method()` will now just instantiates a `String` casts it into an `Object` and return it as it can be seen from A.5. The inlined code presented in A.5 will perform the cast. Since we only want the cast to take place, introspection is of any help in that case and it is safe to just return the instance that the base code expect to cast relaying on the automatic unwrapping provided by Naive to perform the cast operation.

The class `Example` source code can be found in figure A.5. `Example` uses a `CastAlterator-Object` to perform the alteration: as shown on figure A.5, it is no more than a simple implementation of the `CastAlterator`. The method `getMethodToInlineOnCast(String)` will be feed with the static information: in that case the qualified name of the destination type² of the requested cast. Whatever the destination the implementation in figure A.5 gives the method `castReplacer()`

¹Conversion between primitive types can not be altered currently with Naive.

²Classes are types in Java.


```
package naive.thesisexamples.fieldwrite;
import naive.kernel.NaiveMethod;
import java.io.IOException;

public class Example {
    public static void main(Java.lang.String[] args) {
        //do the modifications
        modify();
        //execute the modified code
        ModifiedClass m = new ModifiedClass();
        //display the result
        System.out.println(m.method());
    }
    public static void modify() {
        //get the metaobject describing a method
        NaiveMethod method = NaiveMethod.getNaiveMethod(
            "naive.thesisexamples.fieldwrite.ModifiedClass.method()");
        //perform the alteration
        method.accept(new FieldWriteAlteratorObject());
        //write the modified class file on disk
        try {
            method.getDeclaringClass().save();
        } catch(IOException exp) {
            System.err.println(exp);
            System.exit(-1);
        }
    }
}
```

Figure A.6: Field write alteration: Example class

```

package naive.thesisexamples.fieldwrite;
import naive.kernel.FieldWriteAlterator;

public class FieldWriteAlteratorObject implements FieldWriteAlterator {

    public String getMethodToInlineOnFieldWrite(String
        inReplacementOfFieldWrite) {
        return    "naive.thesisexamples.fieldwrite.
                MethodBodyContainer.fieldWriteReplacer(Object[])" ;
    }
}

```

Figure A.7: Field write alteration: `FieldWriteAlteratorObject` class

declared by `MethodBodyContainer` to inline in place of cast conversion .

This method: `castReplacer()` takes an array in argument containing the dynamic information describing what the base level code wants to achieve: here casting a type into another. The first element of the array is the qualified name of the destination type given as a `String`, the second element is the qualified name of the method requesting the cast, the third element is the instance of the object performing the cast, and finally the fourth element is the object instance that is casted.

Running the example, just display "emoose" as expected. This alteration may seem silly: most of the casting task is performed on the user shoulder by the Naive implementation. This is because Naive has to offer a uniform API to the user. For example, once written a method that will alter the semantic of a field read, a user may want to inline it in various objects without regards of the types of these objects. Thus, the only assumption that the API can do on the classes it will be used with, is that they will be subclasses of `Object`. This, combined with the lack of parameterized types in Java, actually entails that the implementation has to perform cast operations when appropriate. Nevertheless, the ability to alter casts operation is still interesting: one may want to change the casted instance or impose other constraints than types matching to allow the conversion to succeed, raising a `RuntimeException` if needed.

A.6 Method invocation alterations

In the example involving the alteration of invoking a method, we will redefine the `toString()` of `ModifiedClass` so that it will always return "EMOOSE". Then, as shown on figure A.6, the `method()` of `ModifiedClass` will just the message `toString()` to self (i.e. `this`). The purpose of the inlined code will just be to invoke the `toString()` on self as well.

```

package naive.thesisexamples.fieldwrite;
import java.lang.reflect.*;
import naive.kernel.NaiveField;

public class MethodBodyContainer {

    public Object fieldWriteReplacer(Object[] args) {
        try {
            /*args contains
            [ qualifiedFieldNameAsString
              qualifiedMethodReadingFieldAsString
              InstanceOfSenderAsObject
              InstanceOfReceiverAsObject
              newFieldValueAsObject
            ]*/
            String qualifiedFieldName = (String)args[0];
            Object fieldOwner = args[3];
            Object newFieldValue = args[4];
            //get the Naive meta obj
            NaiveField naiveField =
                NaiveField.getNaiveField(qualifiedFieldName);
            //maps from Naive meta object to standard Java meta object
            /* Field langField=naiveField.getAsLangOnDisk()
            can not be used because the class loader may be different*/
            Field langField =
                Class.forName(naiveField.getDeclaringClass()
                              .getCanonicalName())
                    .getField(naiveField.getSimpleName());
            //set the field value
            langField.set(fieldOwner,newFieldValue);
            /*nothing to return
            (whatever the value returned it will be ignored)
            */
            return null;
        } catch( NoSuchFieldException exp) {
            System.err.println(exp);
        } catch( IllegalAccessException exp) {
            System.err.println(exp);
        } catch( ClassNotFoundException exp) {
            System.err.println(exp);
        }
        return null;
    }
}

```

Figure A.8: Field write alteration: MethodBodyContainer class

```
package naive.thesisexamples.cast;
public class ModifiedClass {
    public Object method() {
        String castMe = "emoose";
        Object result = (Object) castMe;
        return result;
    }
}
```

Figure A.9: Cast alteration: ModifiedClass class

```
package naive.thesisexamples.cast;
import java.lang.reflect.*;
import naive.kernel.NaiveField;
public class MethodBodyContainer {
    public Object castReplacer(Object[] args) {
        /*args contains
        [ destinationTypeAsString
          qualifiedMethodPerformingCastAsString
          InstanceOfSenderAsObject
          InstanceOfMethodPerformingCastAsObject
          castedInstanceAsObject
        ]*/
        Object castedInstance = args[3];
        //automatic unwrapping will perform the cast
        return castedInstance;
    }
}
```

Figure A.10: Cast alteration: MethodBodyContainer class

```
package naive.thesisexamples.cast;
import naive.kernel.NaiveMethod;
import java.io.IOException;

public class Example {
    public static void main(Java.lang.String[] args) {
        //do the modifications
        modify();
        //execute the modified code
        ModifiedClass m = new ModifiedClass();
        //display the result
        System.out.println(m.method());
    }
    public static void modify() {
        //get the metaobject describing a method
        NaiveMethod method = NaiveMethod.getNaiveMethod(
            "naive.thesisexamples.cast.
            ModifiedClass.method()");
        //perform the alteration
        method.accept(new CastAlteratorObject());
        //write the modified class file on disk
        try {
            method.getDeclaringClass().save();
        } catch(IOException exp) {
            System.err.println(exp);
            System.exit(-1);
        }
    }
}
```

Figure A.11: Cast alteration: Example class

```
package naive.thesisexamples.cast;
import naive.kernel.CastAlterator;
public class CastAlteratorObject implements CastAlterator {
    public String getMethodToInlineOnCast(String inReplacementCast) {
        return
            "naive.thesisexamples.cast.MethodBodyContainer
            .castReplacer(Object[])";
    }
}
```

Figure A.12: Cast alteration: CastAlterator class

```
package naive.thesisexamples.methodinvocation;
public class ModifiedClass {
    public String toString() {
        return "EMOOSE";
    }
    public String method() {
        System.out.println("hello");
        return this.toString();
    }
}
```

Figure A.13: Method invocation alteration: ModifiedClass class

```

package naive.thesisexamples.methodinvocation;
import java.lang.reflect.*;
import naive.kernel.NaiveMethod;
public class MethodBodyContainer {
    public Object methodCallReplacer(Object[] args)
        throws ClassNotFoundException,
            NoSuchMethodException,
            IllegalAccessException,
            Throwable {
        /*args contains
        [ qualifiedNameOfCalleeAsString
          qualifiedNameOfCallerAsString
          objectInstanceOfCallerAsObject
          objectInstanceOfCalleeAsObject
          argsGivenToCalleeAsObjectArray]*/
        String qualifiedMethodName = (String) args[0];
        Object methodOwner = args[3];
        Object[] argumentValues = (Object[]) args[4];
        //get the Naive meta obj
        NaiveMethod naiveMethod =
            NaiveMethod.getNaiveMethod(qualifiedMethodName);
        //maps from Naive meta object to standard java meta object
        //Method langMethod=naiveMethod.getAsLangOnDisk() can
        //not be used because the class loader may be different
        Class[] argumentTypes = null;
        if (argumentTypes != null) {
            argumentTypes = new Class[argumentValues.length];
            for (int k = 0; k < argumentTypes.length; k++) {
                argumentTypes[k] = argumentValues[k].getClass();
            }
        }
        Class langClass = Class.forName(
            naiveMethod.getDeclaringClass().getCanonicalName()
        );
        Method langMethod = langClass.getMethod(
            naiveMethod.getSimpleName(), argumentTypes
        );
        //perform the method call
        try {
            return resu = langMethod.invoke(
                methodOwner, argumentValues
            );
        } catch (InvocationTargetException exp) {
            throw exp.getTargetException();
        }
    }
}

```

Figure A.14: Method invocation alteration: MethodBodyContainer class

```
package naive.thesisexamples.methodinvocation;
import naive.kernel.NaiveMethod;
import java.io.IOException;
public class Example {
    public static void main(java.lang.String[] args) {
        //do the modifications
        modify();
        //execute the modified code
        ModifiedClass m = new ModifiedClass();
        //display the result
        System.out.println(m.method());
    }
    public static void modify() {
        //get the metaobject describing a method
        NaiveMethod method =
        NaiveMethod.getNaiveMethod(
            "naive.thesisexamples.methodinvocation.
            ModifiedClass.method()"
        );
        //perform the alteration
        method.accept(new MethodInvocationAlteratorObject());
        //write the modified class file on disk
        try {
            method.getDeclaringClass().save();
        } catch(IOException exp) {
            System.err.println(exp);
            System.exit(-1);
        }
    }
}
```

Figure A.15: Method invocation alteration: Example class


```

package naive.thesisexamples.methodinvocation;

import naive.kernel.MethodInvocationAlterator;
public class MethodInvocationAlteratorObject
    implements MethodInvocationAlterator {
    public String getMethodToInlineOnMethodInvocation(
        String inReplacementOfMethodCall
    ) {
        return "naive.thesisexamples.methodinvocation.
            MethodBodyContainer.methodCallReplacer(Object[])
            ";
    }
}

```

Figure A.16: Method invocation alteration: `MethodInvocationAlterator` class

The class `Example` source code can be found in figure A.6. `Example` uses a `MethodInvocationAlteratorObject` to perform the alteration: as shown on A.6, it is no more than a simple implementation of the `MethodInvocationAlterator`. The method `public String getMethodToInlineOnMethodInvocation(String)` will be feed with the static information: in that case the qualified name of the invoked method. Whatever the method the implementation in figure A.6 gives the method `methodCallReplacer()` declared by `MethodBodyContainer` to inline in place of the invoked method.

This method: `methodCallReplacer()` takes an array in argument containing the dynamic information describing what the base level code wants to achieve: here invoking a method. The first element of the array is the qualified name of the method invoked given as a `String`, the second element is the qualified name of the invoker given as a `String`, the third element is the instance of the object invoking the method or null, the fourth element is the instance on which the method is invoke, and finally the fifth element is an object array containing the arguments passed to the invoked method or null if the invoked takes no arguments. If some argument value is a primitive type, it will be wrapped into an appropriate object.

The implementation described in figure A.6 just perform the method invocation by introspection. As expected `Example` run in a console displays `EMOOSE` in output.

Note the above example will fail miserably if the invoked method was a super method redefined in `ModifiedClass`. This is because the introspection capabilities of Java do not offer the ability to invoke a such a method. This stresses the needs to be able to identify these methods performing

the cut and paste. Thus `SuperMethodInvocationAlterator` works exactly like `MethodInvocationAlterator` but is restricted to the invocation of super methods.

```
Object problem = (Object) this;
problem.toString();
```

Figure A.17: Self method invocation alteration: ignored method invocation

By nature, object-orientation makes a difference between object instances namely providing a distinguished instance: `this`. We do believe that making the difference still makes sense even for a copy paste mechanism such as the one proposed here. Thus, `SelfMethodInvocationAlterator` works exactly like `MethodInvocationAlterator` but is restricted to the invocation of methods on self. Because at the implementation level Naive only performs a rough data flow analysis, there are however limitations: for example the invocation of the `toString` shown on figure A.6 will be ignored. Nevertheless all messages explicitly sent to `this` as in `this.toString()` will be treated correctly. While we feel this as a limitation, it is still clear that in the first case, the programmer did not put the accent on sending the message to self, while on a second it emphasizes the fact that the message was sent to self.

A.7 Constructor alterations

```
package naive.thesisexamples.constructor;
public class ModifiedClass {
    public Object method() {
        return new Object();
    }
}
```

Figure A.18: Constructor alteration: `ModifiedClass` class

In the example involving the alteration of invoking a constructor, we will redefine, as shown on figure ??, the `method()` of `ModifiedClass` so that it will just create a new `Object` using a constructor without argument. The purpose of the inlined code will just be to create a new instance of an `Object` as well.

The class `Example` source code can be found in figure ?. `Example` uses a `ConstructorInvocationAlteratorObject` to perform the alteration: as shown on ??, it is no more than a simple implementation of the `ConstructorInvocationAlterator`. The method `public`

```

package naive.thesisexamples.constructor;
import java.lang.reflect.*;
import naive.kernel.NaiveMethod;
public class MethodBodyContainer {
    public Object constructorCallReplacer(Object[] args)
        throws ClassNotFoundException,
            NoSuchMethodException,
            IllegalAccessException,
            Throwable {
        /*args contains
        [ qualifiedNameOfObjectInstantiatedAsString
          qualifiedNameOfCallerAsString
          objectInstanceOfCallerAsObject
          argsGivenToConstructorAsObjectArray]*/
        String qualifiedMethodName = (String) args[0];
        Object[] argumentValues = (Object[]) args[3];
        //get the Naive meta obj
        NaiveMethod naiveMethod =
            NaiveMethod.getNaiveMethod(qualifiedMethodName);
        //maps from Naive meta object to standard java meta object
        //Method langMethod=naiveMethod.getAsLangOnDisk()
        //can not be used because the class loader may be different
        Class[] argumentTypes = null;
        if (argumentTypes != null) {
            argumentTypes = new Class[argumentValues.length];
            for (int k = 0; k < argumentTypes.length; k++) {
                argumentTypes[k] = argumentValues[k].getClass();
            }
        }
        Class langClass =
            Class.forName(naiveMethod.getDeclaringClass()
                .getCanonicalName());
        Constructor langConstructor =
            langClass.getConstructor(argumentTypes);
        Object resu;
        try {
            resu = langConstructor.newInstance(argumentValues);
        } catch (InvocationTargetException exp) {
            throw exp.getTargetException();
        }
        return resu;
    }
}

```

Figure A.19: Constructor alteration: MethodBodyContainer class

```
package naive.thesisexamples.constructor;
import naive.kernel.NaiveMethod;
import java.io.IOException;
public class Example {
    public static void main(java.lang.String[] args) {
        //do the modifications
        modify();
        //execute the modified code
        ModifiedClass m = new ModifiedClass();
        //display the result
        System.out.println(m.method());
    }
    public static void modify() {
        //get the metaobject describing a method
        NaiveMethod method = NaiveMethod.getNaiveMethod(
            "naive.thesisexamples.constructor.
            ModifiedClass.method()");
        //perform the alteration
        method.accept(new ConstructorInvocationAlteratorObject());
        //write the modified class file on disk
        try {
            method.getDeclaringClass().save();
        } catch (IOException exp) {
            System.err.println(exp);
            System.exit(-1);
        }
    }
}
```

Figure A.20: Constructor alteration: Example class

```
package naive.thesisexamples.constructor;
import naive.kernel.ConstructorInvocationAlterator;
public class ConstructorInvocationAlteratorObject
    implements ConstructorInvocationAlterator {
    public String getMethodToInlineOnConstructor
        (String inReplacementOfConstructor) {
        return "naive.thesisexamples.constructor
            .MethodBodyContainer.constructorCallReplacer(Object[])";
    }
}
```

Figure A.21: Constructor alteration: `ConstructorInvocationAlteratorObject` class

`String getMethodToInlineOnConstructorInvocation(String)` will be feed with the static information: in that case the qualified name of the invoked constructor. Whatever the method the implementation in figure A.7 gives the method `constructorCallReplacer()` declared by `MethodBodyContainer` to inline in place of the invoked method.

This method: `constructorCallReplacer()` takes an array in argument containing the dynamic information describing what the base level code wants to achieve: here writing here invoking a method. The first element of the array is the qualified name of the constructor invoked given as a `String`, the second element is the qualified name of the invoker given as a `String`, the third element is the instance of the object invoking the method or null, and finally the fifth element is an object array containing the arguments passed to the invoked constructor or null if the invoked takes no arguments. If some argument value is a primitive type, it will be wrapped into an appropriate object.

The implementation described in figure A.7 just perform the instance creation by introspection. As expected `Example` run in a console displays an object reference in output.

Appendix B

Exceptions between metalevel and baselevel

In this appendix, we discuss the choice made by Naive to handle exceptions. To enable a comparison with previously existing systems, a reflection viewpoint will be taken. Therefore, more precisely, this appendix examines the exception handling between the base level and the meta level. In particular, we will adopt the vocabulary of OpenJava [?], and called the reification of structural entities metaobject. First of all, an overview of Java exceptions facilities will be given, followed by a description of the metaprotocols introduced in chapter 5 with regards to their supports of exceptions handling. In a third part, an example will be taken showing that care must be taken to let the metalevel throws an exception without allowing it to break legitimate assumptions that the base level may do.

B.1 An overview of Java exceptions facilities

The Java language provides error handling facilities. These error handling facilities relays on the exception mechanism.[GS00] states that the *"language specifies that an exception will be thrown when semantic constraints are violated and will cause a non-local transfer of control from the point where the exception occurred to a point that can be specified by the programmer. An exception is said to be thrown from the point where it occurred and is said to be caught at the point to which control is transferred. Programs can also throw exceptions explicitly, using throw statements. Explicit use of throw statements provides an alternative to the old-fashioned style of handling error conditions by returning funny values, such as the integer value -1 where a negative value would not normally be expected. Experience shows that too often such funny values are ignored or not checked for by callers, leading to programs that are not robust, exhibit undesirable behavior, or both. Every exception is represented by an instance of the class Throwable or one of its subclasses; such an object can be used to carry information from the point at which an exception occurs to the handler that catches it. Handlers are established by catch clauses of try statements."* This has two consequences: first Java programmers throw exceptions when the semantic of the method in execution is violated. The definition of the correct semantic of the method is execution is left to the programmer: for instance it could be that for a file operation that the file given in argument is open. Secondly the error handling facilities is supported in the language through specific keywords and conventions: when an exception is thrown, a

catch statement may try to take actions to recover from the error state.

Moreover the language enforces two types of exceptions checked exceptions and unchecked exceptions. The latter are exceptions subclassing `java.lang.RuntimeException` or `java.lang.Error` and can occur at any time in a program. Other exceptions should be declared in the `throws` clause of methods signature.

B.2 Java metaprotocols and exceptions

As stated in 5.2, three approaches can be followed to add reflective capabilities to Java:

1. Modify the source code of the base level to glue the hole between the metalevel and the base level.
2. Modify the bytecode of the base level with the same goals in mind. Naive metaobject protocols fall in this category.
3. Modify the virtual machine interpreting the Java byte-code.

More or less all currently reflective extensions of Java fall in these categories:

- Examples of the first approach are Open Java[Tat99] and Proactive [Vay97]
- Examples of the second approaches are: Javassist[Chi00], Kava[WS00], and Reflex[Tan00]¹
- Examples of the third approach are: Metaxa[Go197] and Guarana[OCGB98]

Before discussing the behavior of all this extensions towards exception handling, it is probably worth knowing that although it only provides introspection, a problem akin to this one: - exception handling in Java reflective system - occurs in the standard Java distribution. In fact, in the case of the dynamic proxy shipped with the version 1.3 of the Java Development Kit, the package `java.lang.reflect` offers the ability to dynamically create a proxy of a given pre-existing interface. Proxy implementers should only provide one methods taking a reified description of the messages sent to the proxy in argument. This method is furthermore allowed to throw any kind of exception. This method may therefore throw an exception that is not declared by the pre-existing interface. But all method sent to the proxy are sent through the pre-existing interface. Since the pre-existing interface may declare method that do not throw any exception, the implementation method of the proxy may throw an exception which is not expected from the point of view of the message sender. However, the JDK specifies that: *"the exception thrown from the method invocation on the proxy instance. The exception's type must be assignable either to any of the exception types declared in the throws clause of the interface method or to the unchecked exception types `java.lang.RuntimeException` or `java.lang.Error`. If*

¹Reflex is written on top of Javassist and therefore we will not described it more

a checked exception is thrown by this method that is not assignable to any of the exception type declared in the throws clause of the interface method, then an `UndeclaredThrowableException` containing the exception that was thrown by this method will be thrown by the method invocation on the proxy instance.”

Open Java proceeds by reifying at compile time the different syntactic nodes corresponding to the Java language like class declarations for instance. Using the tree of nodes, the meta-level is free to insert what it wants exactly where it wishes. The insertion takes the form of strings insert in the source file. This approach makes no restriction on the exceptions thrown by the meta-level. The nature of the process: string insertions in the base level source code also ensures that the exceptions thrown by the meta-level can only be of classes or of subclasses of the exceptions handled by the base level code.

Proactive follows a different path: it generates a proxy. Each method message that was sent to the original object will be sent to the proxy. The proxy is free to handle the message as it wants. But the signature of the methods defined in Proactive allows only to the meta-level to throw exceptions of types: `java.lang.reflect.InvocationTargetException` or `java.lang.IllegalAccessException`.

Javassist in its release 0.8 does not allow the meta-level to throw exceptions in the base level. In its last version: 1.0, it allows the meta-level to throw any type of exceptions in the base level.

Kava only offers the ability for the meta-level to be notified when the base level will require some specific actions (like sending a message) after or before the action take place. There is no way in Kava to prevent the action required by the base level to take place. Kava does not allow the base level to throw an exception in the base level.

Naive allows any type of exceptions to be thrown by the meta level. However when an exception is thrown in the base level, it is checked in order to see if the base level can handle it. If the base level can handle it, it is thrown in the base level. Otherwise it is wrapped in `java.lang.reflect.UndeclaredThrowableException`. To determine if the base level code is able to handle an exception at that point, Naive proceeds by analyzing the exceptions that the intercepts instructions from the base level may throw.

MetaXa is no longer distributed, it will therefore not be discussed.

Guaran does not allow the meta level to throw exception into the base level.

To summarize, we have seen three types of behaviors:

- In Kava, in Javassist 0.8 and in Guaran, the meta-level is not allowed to throw any kind of exceptions into the base-level.
- In Proactive, the meta-level is allowed to throw only `java.lang.reflect.InvocationTargetException` or `java.lang.IllegalAccessException`.

- In Javassist 1.0, the meta-level is allowed to throw any kind of exceptions in the base level.
- In the JDK proxy API, and in Naive, the meta-level is allowed to send only exceptions in the base level that the base level expects. If the base level does not expect an exception thrown by the meta level, a wrapping of the exception into an unchecked exception is thrown before it is thrown.

It is important to note that in the two first approaches, the metalevel is still allowed to throw unchecked exception - in other words exceptions subclassing `java.lang.RuntimeException` or `java.lang.Error.Exception` - in the base level. This is because unchecked exceptions do not need to be declared in the throw clause of Java methods.

As it has been previously explained, we believe that the first two approaches impose unneeded restrictions on the meta-level programmer liberty. The question then arises to compare the last two approaches.

B.3 Exceptions and assumptions

The previous section has described the main metaobject protocols available with regards on their supports of exceptions. The aim of the present part is to determine whether the metalevel should be allowed to throw any exception in the base level or whether if a wrapping into an unchecked exception should occur when the exception thrown by the base level is not expected. We will therefore compare the behavior of the Java proxy, Javassist 1.0 and Naive -0.1 on the example presented below.

B.3.1 The shape of the example

The main classes involved in the example are presented in figure B.3.1 and B.3.1. What happens in this example? Mainly, the `main` function defined in figure B.3.1 try to do something (that something is precisely: `useBaseLevel(createBaseLevel())`). If during the execution of the something, any exception is thrown, `main` catches the exception and provides an interpretation from the trapped exception. If the exception is a checked exception, it is interpreted as an exception thrown by the instantiation of a `BaseLevel` object in the method `createBaseLevel()`, otherwise it is interpreted as an unchecked exception. The latter method: `createBaseLevel()` is in charge of creating and returning an instance of a `BaseLevel`, if any checked exception occurs during the instantiation, `createBaseLevel()` will infer that there was a security problem and will throw a `GeneralSecurityException` at the place of the exception that has been encountered. The method `useBaseLevel` will try to invoke method on the instance it has been given on argument. It will trap a `BaseLevelAwareException` which is a checked exception, and treated it as being non harmful. `BaseLevel` on Figure 2 is just an auxiliary class whose constructor may throw an unchecked exception: `IllegalArgumentException` and whose unique method can throw a

```
import java.lang.reflect.*;
import java.security.GeneralSecurityException;

public class BaseLevelUser extends Object {

    public static BaseLevel createBaseLevel()
        throws GeneralSecurityException {
        try {
            return new BaseLevel();
        } catch(IllegalArgumentException exp) {
            //reinterpret the exception to the context
            throw new
                GeneralSecurityException("creation was not allowed");
        }
    }

    public static void useBaseLevel(BaseLevel baseLevelObject) {
        try {
            baseLevelObject.method();
        } catch(BaseLevelAwareException exp) {
            System.out.println(
                "BaseLevelAwareException trapped"
            );
        }
    }

    public static void main(String[] args) {
        try {
            useBaseLevel(createBaseLevel());
        } catch(RuntimeException exp) {
            System.out.println(
                "unexpected runtime exception "+exp
            );
        } catch(Exception exp) {
            System.out.println(
                "Exception trapped while creating object "+exp
            );
        }
    }
}
```

Figure B.1: Main class of the example.

```

public class BaseLevel extends Object {

    //constructor
    public BaseLevel()
        throws IllegalArgumentException {
        super();
    }

    public void method()
        throws BaseLevelAwareException {
        System.out.println(
            "BaseLevel.method()"
        );
    }
}

```

Figure B.2: Auxiliary class of the example.

checked exception: `BaseLevelAwareException`. What should be reminded from this example is that:

- No checked exceptions can occur during the execution of `useBaseLevel`.
- A checked exception can occur during the execution of `createBaseLevel`.
- The programmer has used these two observations to infer in `main` that when a checked exception occurs in the statement `useBaseLevel(createBaseLevel())`; this is because the instantiation of `BaseLevel()` failed. It may of course take more complicated measures to recover from the instantiation problem than displaying the problem on the console.

Even if there are better ways to write a program with the same behavior, the example presented is legal in Java and the introduction of a metaobject protocol should not break it. The following example will use metaobject protocols to instrument `BaseLevel.method()` in an attempt to acknowledge whether or whether not throwing exceptions may break the inference made by `main`.

B.3.2 Javassist 1.0

We leave the previous example unchanged. We only add one class `MetaLevel` presented on Figure B.3.2.

```
import javassist.reflect.*;
import java.security.GeneralSecurityException;

public class MetaLevel extends Metaobject {

    //constructor required by javassist
    public MetaLevel(Object self, Object[] args) {
        super(self,args);
    }

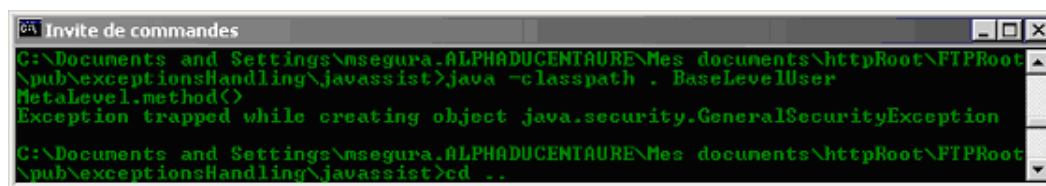
    public Object trapMethodcall(int identifier,
                                Object[] args)
                                throws Throwable {
        return this.method();
    }

    public Object method()
    throws Throwable {
        System.out.println(
            "MetaLevel.method()"
        );
        throw new GeneralSecurityException();
    }
}
```

Figure B.3: Metalevel with Javassist 1.0.

This object is in charge of trapping the method call `BaseLevel.method()` and instead to execute the code contains in `MetaLevel.method()`. This latter method throws a `GeneralSecurityException`.

We used a class loader to make the link between the meta level and the base level and we type in a console: `java -classpath . javassist.reflect.Compiler BaseLevel -m MetaLevel`. The result is shown on figure B.4.



```

Invite de commandes
C:\Documents and Settings\msegura.ALPHADUCENTAURE\Mes documents\httpRoot\FTPRoot\pub\exceptionsHandling\javassist>java -classpath . BaseLevelUser
MetaLevel.method()
Exception trapped while creating object java.security.GeneralSecurityException
C:\Documents and Settings\msegura.ALPHADUCENTAURE\Mes documents\httpRoot\FTPRoot\pub\exceptionsHandling\javassist>cd ..

```

Figure B.4: Javassist running the example

What happens? Actually, using javassist ability to throw any exception in the base level, we did instrument `BaseLevel.method()` so that it will always throw a `GeneralSecurityException`. Thus the message `BaseLevelUser.useBaseLevel()` throws a `GeneralSecurityException`. Finally `BaseLevelUser.main()` catches a checked exception and infers that the instantiation of `BaseLevel` did failed during the message `BaseLevelUser.createBaseLevel()`.

What we did learn here is that allowing the metalevel to throw any kind of exception is the base level may lead the base level to take wrong recovery actions from an error state.

B.3.3 Naive

We use Naive to instrument the `BaseLevel.method()` exactly as with javassist. The code replacing `BaseLevel.method()` is the code contained in `MetaLevelmethodClassReplacer(Object[])` shown on figure B.3.3.

Since Naive offers only structural reflection, more work is needed to bind the meta level with the base level. This is achieved by `Binder.main()`. The class `Binder` is shown on figure B.3.3. Then running in a console `Java -classpath bcel.jar;nave.jar;. exceptions.Binder` and after `Java -classpath bcel.jar;naive.jar;. exceptions.BaseLevelUser`, we reach the result shown on figure B.7.

What happens there? In fact, Naive analyzes the method call that it has replaced with `MetaLevelmethodClassReplacer(Object[])`. It has inferred that each checked exception throws that was not of type `BaseLevelAwareException` should be wrapped into an unchecked exception.

```
package exceptions;
import java.security.*;
public class MetaLevel {
    public MetaLevel() {
        super();
    }
    public void methodCallReplacer(Object[] args)
        throws Throwable {
        throw new GeneralSecurityException();
        return ;
    }
}
```

Figure B.5: Metalevel with Naive.

When the latter case occur, Naive wraps the unexpected exception into a `java.lang.reflect.UndeclaredThrowableException`. This is exactly the case of the example. By comparison with `javassist`, `BaseLevelUser.useBaseLevel()` now throws an unchecked exception. This does not break the assumption made by `main()`.

One might think that this example is biased. What if `main` was doing some assumptions on catching an `java.lang.reflect.UndeclaredThrowableException`. The truth is that if one can catch an unchecked exception, it can not make any hypothesis on where the exception has been thrown because an unchecked exception can occur at any time, any point in a Java program. Putting it another way, unchecked exceptions can be thought as exceptions that the base level always expect to occur. Therefore we believe that the example given is fair.

What we did learn here is that allowing the metalayer to throw exceptions in the base level is possible without breaking it. But unexpected exceptions from the point of view of the base level should be wrap in some unchecked exceptions.

B.4 Conclusion

The need of a meta-layer to throw exception into the base level can be satisfied on condition that exceptions unplanned by the base level are wrapped into unchecked exceptions. This is certainly not a surprise: you can recover from error states you did planned. Not more. Metaobject protocols allowing the meta-level to throw any kind of exception in the base level ask the base level to be able to recover from error states it did not planned. This is wrong but we believe this as being less problematic that metaobject that did not allow the metalevel to communicate their error states to the base level, because

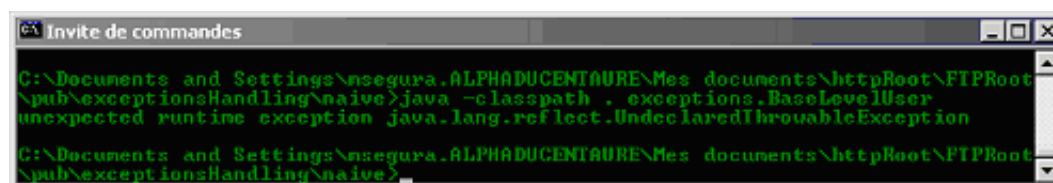
```

package exceptions;
import naive.kernel.*;
import naive.kernel.impl.*;
public class Binder implements MethodInvocationAlterator {
    public String getMethodToInline(String inReplacementOfMethod) {
        return "exceptions.MetaLevel.methodCallReplacer(
            Object[] args)";
    }
    public static void main(Java.lang.String[] args)
        throws Exception{

        //do the modifications
        modify();
    }
    public static void modify()
        throws Exception {
        NaiveMethod decorator = NaiveMethod.getNaiveMethod(
            "exceptions.BaseLevel.method()"
        );
        decorator.accept(new Binder());
        //just to make sure that it
        //will be written on disk before using it
        decorator.getDeclaringClass().save();
    }
}

```

Figure B.6: Metalevel with Naive.



The screenshot shows a Windows command prompt window titled "Invite de commandes". The command entered is:
 `C:\Documents and Settings\nsegura.ALPHADUCENTAURE\Mes documents\httpRoot\FTPRoot\pub\exceptionsHandling\naive>java -classpath . exceptions.BaseLevelUser`
 The output shows an error:
 `unexpected runtime exception java.lang.reflect.UndeclaredThrowableException`
 The prompt is now at:
 `C:\Documents and Settings\nsegura.ALPHADUCENTAURE\Mes documents\httpRoot\FTPRoot\pub\exceptionsHandling\naive>`

Figure B.7: Naive running the example

in the first approach, the metalevel can be made responsible of the consistency of the exceptions it throws while on the last approach, the only solution is to duplicate the measures taken by the base level to recover from an exception is to duplicate them in the metalevel. Something as beautiful as reflective systems should not lead to horrors like code duplications.