A Class-Object Model for Program Transformations

Doctoral Program in Engineering
University of Tsukuba, Japan

January 2002

Michiaki Tatsubori

A Dissertation Submitted to the
Graduate School of Engineering
University of Tsukuba

In Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy in Engineering

Abstract

The recent expert programmers who have been forced to develop large and complicated programs have strong desire to write "good codes" from the viewpoints of both runtime efficiency and understandability, and they expect a translator to generate a good code just to fit their own needs if possible. This fitting usually requires complicated customization of the translator which could only be done by the experienced compiler experts, and the programmers cannot reflect their ideas on the translators so easily, especially when the programmer's demands to the quality of practical programming tools are severe. These situations lead us to the necessity to develop a more flexible approach by which the specific programmers can reflect their experience and knowledge on the translators by themselves. The technology that enables the reuse of larger software components have become available with the emergence of the object-oriented paradigm, which have widened the applicability of reusable code pieces. However, the class-based object-oriented modularization cannot always encapsulate every design decisions perfectly; especially when they crosscut the module structure of a program, code pieces crosscutting a program tangle with other codes and scatter over the entire program; hence, class-based modularization is eventually violated. This kind of violation may happen modern network programming involving distribution or security issues.

This thesis addresses how to solve these crosscutting problems without losing object-oriented framework's appearance or runtime performance. To cope with both of these two problems, transformational system is to be used to embed the crosscutting code among the entire program automatically and hide the scattered code from the appearance. Runtime performance can be achieved so that the transformational approaches could inherently produce only the necessary and elaborate code based on the expert programmer's experiences. For this purpose, we propose a class-object model for transforming object-oriented programs, and develop powerful transformational systems not only for compiler experts but also for object-oriented programming experts. The proposed class-object model is the abstract data model representing the logical structure of an object-oriented program and its logical alternations. In order to ease the description, the transforma-

tional systems allow users to describe transformations of programs in the intuitive notions of object-oriented programming rather than the notions of compiler implementations. The proposed class-object model makes powerful transformations not only of the compiler experts but also of the object-oriented programming experts expected to exist more than compiler experts, who must also be familiar with object-oriented programming. The reuse of well-defined program code modules could be regarded to reduce the cost of software development, keeping programmers away from the continuous rediscovery and reinvention of concepts across the software industry.

Also, this thesis discloses the design and implementation of practical program-transformation systems based on the proposed class-object model; namely, OpenJava and Javassist as general-purpose transformational systems, and Addistant as a special-purpose system for the support of distributed programming. OpenJava is an object-oriented macro system for transformations of source-code program written in Java. Javassist is a Java bytecode manipulating tool for transformations of binary programs for Java virtual machines. Addistant is a bytecode translator built on Javassist as an application case-study of Javassist. By the use of Addistant, legacy Java software can be modified so that it can be translated to the programs which can be executed in the distributed environment. In these applications, programmers can select how remote references are implemented for each class. From the viewpoint of reflection, these transformational systems are regarded as reflective systems providing architectures for structural reflection, avoiding the runtime overhead of general reflection. These systems are freely available and have been used widely in the world.

Acknowledgments

I profoundly thank Professor Kozo Itano (University of Tsukuba), who is my supervisor at the University of Tsukuba. And, I would like to express my deep gratitude to Professor Shigeru Chiba (Tokyo Institute of Technology), who also supervised this thesis. He is the one behind OpenC++, which served as the great stimulus for my study around reflection and metaobject protocols.

Professor Doug Lea (SUNY Oswego), Doctor Brent Hailpern (IBM Watson Research Center), Professor James Noble (Victoria University of Wellington), Professor Mary Beth Rosson (Virginia Tech), Doctor Richard Gabriel (Sun Microsystems), and Professor Ron Goldman (Stanford University) gave me a valuable guidance for this thesis construction at the OOPSLA 2001 doctoral symposium.

Doctor Marc-Olivier Killijian (LAAS/CNRS) reviewed a draft of this thesis, suggesting many improvements. The thesis comittees reviewed the submitted version of this thesis and the final version reflects their comments. The comittees are organized by Professor Kozo Itano, Professor Jiro Tanaka, Professor Nobuo Ohbo, Professor Seiichi Nishihara, Professor Kazuhiko Kato (University of Tsukuba), and Professor Ikuo Nakata (Hosei University).

Professor Ikuo Nakata (now Hosei University) and Professor Yoshiyuki Yamashita (now Saga University) were supervisors when I belonged to the LANG (Programming Language) group at the University of Tsukuba from 1996 to 1998. Doctor David Powell and Doctor Jean-Charles Fabre (LAAS-/CNRS) managed my stay at LAAS in 1999.

I also thank colleagues in the HLLA (High-Level Language and Software Architecture) group and the LANG group at the University of Tsukuba, especially Professor Hisashi Nakai (now University of Library and Information Science), Professor Yasushi Shinjo (University of Tsukuba), Professor Hideki Sakamoto (now Tsukuba International University), Teruo Koyanagi (now IBM Tokyo Research Lab.), and Toshiyuki Sasaki (now Hitachi Ltd.).

Part of my friends who are somehow related to my research are good companies for me to disscuss with or just to share fun with. Especially Kenichi Kourai (University of Tokyo), Yasutaka Sakayori (University of Tsukuba),

Juan-Carlos Ruiz-Garcia (LAAS/CNRS), Hidenori Miyamoto (now IBM), Daisuke Yokota (University of Tsukuba), Chanika Hobatr (Clemson University), and Eric Tanter (University of Chile) must have given me an indirect but great influence on my way of thinking.

Doctor James Gosling (Sun Microsystems) gave me a chance to join him at a table of a breakfast and the talk there encouraged me to contribute my work to the Java community.

Financially, ACM supported me to join the OOPSLA ACM conferences from 1998 to 2001. LAAS/CNRS supported me to research at Toulouse, France in 1999. Sun Microsystems Inc. supported me to visit their firm at Silicon Valley, USA in 1997.

This thesis project was conducted mainly at the University of Tsukuba, Ibaraki, Japan, and partly at Tokyo Institute of Technology, Tokyo, Japan.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The recent expert programmers who have been forced to develop large and complicated programs have strong desire to write "good codes" from the viewpoints of both runtime efficiency and understandability, and they expect a translator to generate a good code just to fit their own needs if possible. However, such fitting usually requires complicated customization of the translator which could only be done by the experienced compiler experts, and the programmers cannot reflect their ideas on the translators so easily, especially when the programmer's demands to the quality of practical programming tools are severe. These situations lead us to the necessity to develop a more flexible approach by which the specific programmers can reflect their experience and knowledge on the translators by themselves. The technology that enables the reuse of larger software components is one of the means to reflect those experiences on the actual programming [54]. The reuse technologies themselves have become available with the emergence of the object-oriented paradigm since SIMULA-67 [24], and dynamic-binding and polymorphism of *classes* have widened the applicability of reusable code pieces. These mechanisms enable *inversion of control* for the reuse of components, which allows to build a framework [45]. A larger component of the main system design is now reusable as an object-oriented framework in addition to the reusable sub-systems. Thus, the reuse of well-defined program code *modules* could be regarded to reduce the cost of software development, keeping programmers away from the continuous rediscovery and reinvention of concepts across the software industry.

However, the class-based object-oriented modularization cannot always encapsulate every design decisions perfectly; especially when they *crosscut* the module structure of a program, code pieces crosscutting a program tangle with other codes and scatter over the entire program; hence, class-based modularization is eventually violated. This kind of violation may happen modern network programming involving distribution or security is-

sues. We address how to solve these crosscutting problems without losing object-oriented framework's appearance or runtime performance. To cope with both of these two problems, transformational system is to be used to embed the crosscutting code among the entire program automatically and hide the scattered code from the appearance. Runtime performance can be achieved so that the transformational approaches could inherently produce only the necessary and elaborate code based on the expert programmer's experiences. For this purpose, we propose a class-object model for transforming object-oriented programs, and develop powerful transformational systems not only for compiler experts but also for object-oriented programming experts. The proposed class-object model is the abstract data model representing the logical structure of an object-oriented program and its logical alternations. In order to ease the description, the transformational systems allow *metaprogrammers* as users to describe transformations of programs in the intuitive notions of object-oriented programming rather than the notions of compiler implementations.

This thesis discloses the design and implementation of practical program-transformation systems based on the proposed class-object model; namely, OpenJava [80, 82, 81] and Javassist [20] as general-purpose program-transformation systems, and Addistant [83, 79] as a special-purpose system for the support of distributed programming. From the viewpoint of *reflection*, these transformational systems are regarded as reflective systems providing architectures for structural reflection, avoiding the runtime overhead of general reflection.

1.1 Limitations in Object-Orientation

Though the object-oriented modularization using classes is powerful, it sometimes fails to encapsulate some design decisions that *crosscut* the module structure of the program. Code pieces crosscutting a program tangle with other code and scatters over the program. An example of a crosscutting design decision is objects distribution in a network. Code related to concerns of objects distribution is often tangled with other code and scatters over a number of classes. Lack of modularization implies low reusability and low maintainability of code pieces. When programmers change the decomposing points of program in distributed environment for reducing the overhead of network communication, this change of design decision brings about the modification of remote or local object allocation code scattering over the program. They have to modify a number of code pieces for implementing a simple change of distributed design decision.

The responsibility of this low reusability weighs not only on the designer of a problematic program but also the language with which programmers

describes the program. Actually, it is often impossible to give a settlement of design as long as they use the language. Even though design patterns[32] give you well-sophisticated designs for solving a design problem but a design given by a pattern which offers benefits often implies additional drawbacks to the program.

For example, to preserve the maintainability of the design structure and algorithm of a non-distributed part of a program, it is typical to represent a remote object by a proxy object. This design is known as the PROXY pattern and enables writing simple clients code using remote objects as if the remote objects were local objects. However, describing a proxy class is a tedious task and programmers must provide lots of proxy classes for every class whose instances are remotely accessed. Special compilers who automatically generates proxy classes according to the original classes have been proposed for addressing this modularization limitation of object-oriented languages.

Another example in distributed programming is distributed allocation coding. The ABSTRACT FACTORY pattern allows you to write a centered code controlling instance allocations as a Factory class. You can change the policy of distribution of objects by overriding factory methods in a subclass of the Factory class. But, with this design, a class of the Factory role must provide a number of factory methods for all the combinations of instantiated classes and contexts creating instances. You must write very redundant code and it is a tedious and error-prone task.

The software industry needs a more flexible and fine-grained modularization mechanism for software because the complexity of software increases drastically as computing power, massive storage, pervasive network infrastructures and growing heterogeneity of computing platforms become available. One of the most important challenges is to enhance the fundamental support for reuse beyond object-orientation in order to draw out the abilities of frameworks, design patterns, and other techniques for reuse of software. Since application frameworks and design patterns are techniques to make most of the existing reuse mechanism, object-oriented application frameworks and object-oriented design patterns suffers from the limitation of their base object-oriented models provided by object-oriented programming languages.

The limitation of simple object-oriented languages with class mechanisms is well-known and domain specific language extensions (DSLE) are known to solve this problem. A language with a DSLE is a specialized, problem-oriented language and it allows solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Thus a modularization mechanism supplied by a DSLE can be quite suitable to reuse. A multi-purpose super-rich language providing DSLEs for many specific application domain cannot be a solution for every application domain. Though it di-

rectly supports programmers in some specific application domains, it cannot be expressive enough for all the applications. In order to achieve suitability for all the applications, a language is desired to have several facilities supporting primitives in every application domain. However, a multi-paradigm language, which supports tons of language primitives from the beginning, tends to have too complex specifications to learn. Moreover, it cannot support applications which are unknown when the language is designed.

1.2 Problems in Program Transformations

One choice to be free from the limitation of a language is to extend the language with the desired features for yourself. You can implement this extension as a translator. A transformation, a mapping from programs to programs, is often an implementation of a language extension, and we can implement it on demand. According to Krueger [54], transformational systems are one of the eight different approaches to software reuse. Implementation of a transformation can be a kind of software library which can be applied to another software. The approach to reusable software by fully-featured transformations can provide more flexible reusability than other approaches since the decomposing points of software for modules are freely definable with the transformational approach.

1.2.1 Development Costs of a Translator

Developing a transformation as a reusable unit, however, tends to cost relatively more than other approaches while the reusability of developed units can be very good. A naive system with a lifted abstraction level of transformation reduces that difficulty but it also spoils the flexibility of modularization, which is the advantage of the transformational approach to software reuse. For instance, the `#define` macro of C/C++ is very simple to use but what we can do with it is very limited to just a substitution of a keyword or a keyword followed by parenthesizes with parameter variables.

Developing translators as stand-alone programs from scratch is the most effort-intensive way of building translators. You have to design and implement the internal source representation as well as the code performing analysis, optimization, and generation from scratch. In case the translator expects input in text form, you will need to build a lexer and a parser for converting the source text into the internal representation. The lexer and the parser can be generated using tools, such as lex and yacc. But this is the only piece of reusable infrastructure that is available in this approach. In addition to a large effort needed to build a translator from scratch, this approach also impairs the integration of different notations and development

tools. This is because each translator uses its own internal source representation and does not provide any interfacing facilities. As a result, we get a language landscape consisting of islands of noninteroperable domain-specific languages.

Typically, we implement a translator as follows:

1. *Input for the internal representation:* If the source is given in the form text, the translator needs to parse the text into the internal representation. In many cases, the internal program representation has the form of an abstract syntax tree. Other examples of representations are data and control flow graphs.

2. *Code analysis:* It often analyzes code to obtain some information of program such as data and control flow. They are needed to check the structure of the input program and also to choose where to apply adequate transformations.

3. *Transformation:* It performs required transformations to the internal program representation.

4. *Output from the internal representation:* It needs to transform the internal representation back into a textual representation. One class of output facilities are code-generation backends, which generate the machine code for a given target platform.

1.2.2 Transformational Systems

Transformational systems are support system for the development of translators. A transformational system provides toolkits and frameworks for partial transformation processes commonly used by several translators.

Transformational systems consist of the following elements:

- *A data-structure for the internal program representation:* A built-in data structure for representing program save metaprogrammers (translator developers) from providing it by them-selves.

- *Code analysis facilities:* Transformational systems often provide code analysis facilities, such as data and control flow analysis. They are needed to check the structure of the input program and also to guide the selection of appropriated transformations.

- *A transformation engine (also called a rewrite engine):* A transformation engine applies user-provided transformations to the internal program representation. One of the responsibilities of a transformation engine is scheduling transformations, that is, determining the order of applying transformations.

- *Input and output facilities for the internal representation:* If the source is given in the form text, we need a parser to transform it into the internal representation. We can also use various unparsers to transform the internal representation back into a textual representation. One class of output facilities are code-generation backends, which generate the machine code for a given target platform. Another class of input/output facilities are editors, which allow us to directly edit the internal representation and to render it in different ways. The latter are provided by the Intentional Programming system.

1.3 Goal — Higher Abstractions for Powerful Transformations

The goal of this thesis is to provide a design model with high-level abstractions for building object-oriented transformational systems. Preserving the most of expressive power in a fully featured transformational system, a transformational system should give metaprogrammers a way to simply and intuitively describe object-oriented program transformations.

A transformational system for simple descriptions of object-oriented program transformations should provide:

- *A declarative data format for the internal program representation:* The format of abstract syntax tree or the format of procedural data/control flow graphs are not suitable for representing an object-oriented logical structure of a program.
- *Capsularation analysis facilities:* Transformational systems should provide code analysis facilities suitable for capturing the capsularation mechanism of the base object-oriented language.
- *An event-driven style rewrite engine:* Determining where to apply transformations are complicated task thus it should be automated.
- *Abstracted input and output facilities:* Fetching the source files of a program should be hidden.

1.4 A Class-Object Model Approach

A higher abstraction design model has been required for a system of program transformation with which programmers can simply write metaprogram dealing with large and complex code transformation. Especially, a number of transformations typical in object-oriented program require a highly abstracted data model with which they can directly deal with the object-oriented logical structure of program.

The class-object model proposed in this thesis gives metaprogrammers a logical class representation for a class declaration in program source. This design model captures declarative construct of classes including the inheritance mechanism and access controlling by data hiding. Supporting the callee-side transformation, it also addresses the polymorphic mechanism of dynamic binding.

In our approach, a translator program can directly manipulate important object-oriented language constructs; *Declaration* (Inheritance), *Encapsulation* (Access control), and *Polymorphism* (Dynamic binding). Thus metaprogrammers can simply and intuitively describe object-oriented transformations with a transformational systems designed with our model.

1.5 Related Work

A translator, or a generator, is a program that takes a higher-level specification of a piece of software and produces its implementation. The piece of software could be a large software system, a component, a class, a procedure, and so on. This approach to software reuse has been researched as generators [5] and cutting-edge implementation technologies include C++ templated metaprogramming [64], aspect-oriented programming [48], intentional programming [72], and others. Krzysztof Czarnecki and Ulrich W. Eisenecker tied these kinds of closely-related researches as the concept of generative programming [23]. The work of this thesis is also in this stream.

1.5.1 Macro Systems and Preprocessor Toolkits

Macro systems have been typical systems manipulating source-text to source-text transformation from early days [13]. Lisp macros [75, 53] and Scheme macros[27] are known as powerful macro systems with which programmers can describe the transformation process in a procedural manner in the languages themselves. In the traditional Lisp-style macro systems, a metaprogram as a macro needs to handle the abstract syntax tree (AST) of the source text piece in the original program. They were designed for implementing small, localized transformations.

Java Syntax Extender [3] (JSE) adds Java language a macro facility similar to the one of Dylan[71] but offers a fully procedural macro engine. With JSE, programmers must begin with a keyword for writing code in an extended syntax. They define a translator implementing the extension, and the translator is bound to the keyword. The translator transforms the code written in the extended syntax. We call this type in applying macros *keyword-driven* macro application. This is a typical macro which is a lisp-style macro system applied to Java. With JSE, metaprogram handles code in the form of *skeleton syntax tree* which they insist has fewer categories

than a typical AST and instead represents the basic shapes and distinctions necessary for macro processing. A translator transforms a code piece using a library providing syntactical pattern matching and code construction utilities.

The Jakarta Tool Suite [6] is a set of precompiler-compiler tools for extending Java with domain-specific constructs. Its main tools are Jak and Bali. Jak supports the definitions of AST constructors. ASTs are created using typed code quotes and are manipulated using *AST cursors* which is a library for tree traversing from left to right. Bali is a parser generator with which its users can create syntactic extensions in a more familiar BNF style with regular-expression repetitions. The result of parsing is an AST which can be further modified through a tree walk. A large AST is hard to handle with a fixed traversal style.

EPP [42] is an extensible preprocessor kit for Java. It is an application framework for preprocessor type language extension systems. The parser of EPP is written by recursive descent style and provides many hooks for extensions. By using these hooks, the extension programmer can introduce new features, possibly associated with new syntax. Because all grammar rules are handled in a modular way, it is also possible to remove some original grammar rules from standard Java. EPP enables preprocessor programmers to write an extension as a separate module, called EPP plug-ins. If only plug-ins do not cause a collision, the end-user can incorporate multiple plug-ins into EPP simultaneously. In fact, it is powerful for locally limited translation though programmers must write *recursive descent parser*.

1.5.2 Reflection

The *reflection* is often used as a model of language extension. The concept of reflection was originally proposed by Smith [73] as 3-Lisp. It can be generally parted into two kinds of functions. One is *introspection* and another is *intercession*. The introspection is the mechanism to obtain information of program and use it in program. And the intercession is the one to change the behavior and implementation of program in program. The program which performs reflective computation, intercession or introspection, is called *meta-level* program while the program on which reflective computation is performed is called *base-level* program. Generally, it is difficult to archive fully available intercession without execution overheads of meta-level computation. If it were not for reflection mechanisms, programmers could not handle the behavior of program since it is not a first-class in the language, unlike string, integer, boolean, and so on. In reflection of object-oriented programming, it is usual to provide a class (*metaclass*) representing instance objects (*metaobjects*) for these non-first level things [22].

From a software engineering viewpoint, reflection is a tool for separa-

tion of concerns and thus it can be used for letting programmers write a program with higher-level abstraction and with good modularity. This is because reflection is a technique for changing the program behavior according to another program. For example, a number of reflective systems provide metaobjects for intercepting object behavior, that is, method invocations and field accesses. Those metaobjects can be used for *weaving* several programs separately written from distinct aspects, such as an application algorithm, distribution, resource allocation, and user interface, into a single executable program.

However, previous reflective systems do not satisfy all the requirements in software engineering. Although the abstraction provided by the metaobjects for intercepting object behavior is easy to understand and use, they can be used for implementing only limited kinds of separation of concerns. Moreover, this type of reflection often involves runtime penalties. Reflective systems should enable more fine-grained program weaving and perform as much reflective computation as possible at compile time for avoiding runtime penalties.

The point is how to define an interface to these metaobjects, and how to realize these mechanisms, which is called *metaobject protocol* (MOP). [50] If MOPs are simply designed and implemented, it would provide interpreters on the executional environment. The source program run on one of these interpreter and it can modify the interpreter to change its behavior. Though reflection mechanisms are fully provided by this method, such a design and implementation causes too serious overhead of execution.

Runtime Reflection

The CLOS (Common Lisp Object System) MOP [49] is an exemplary model of how to provide fully-functional reflective support in a language. It was an open and adaptable implementation which could be modified to provide features that were not part of standard CLOS behavior. It employs class metaobjects instead of the metaobjects for objects. Although the *currying* [12] technique allows metaobjects in the CLOS MOP to partly run at compile time, the rest of computation by the metaobjects is still performed at runtime. At least, which metaobject is selected for given source program is determined at runtime.

ABCL/R3 [61] is a compilation framework in object-oriented reflective languages. In their framework, the meta-level of the language is exposed to the programmer as a pure meta-circular interpreter organized in an object-oriented way, as is with traditional approaches. The interpretation overhead is effectively eliminated by the compiler with the technique based on *partial evaluation* [31]. Programmers can write meta program more easily on this system since they can consider how to execute other than how to compile.

But implementing an effective partial evaluator is very difficult. In fact, it seems that there is no effective one for Java.

Reflection in Java

The Java provides limited reflection mechanisms. One is the Java Reflection API which enables introspection. Another is the class loader API which enables intercession. And, there is other researches on reflection in Java.

The Java Core Reflection API [44] provides a type-safe API that supports introspection about the classes and objects in the current Java VM(Virtual Machine) at runtime. This API can be used to:

- construct new class instances and new arrays
- access and modify fields of objects and classes
- invoke methods on objects and classes
- access and modify elements of arrays

Programmers might want to easily handle classes unknown at programming time in order to provide applications like debugger, JavaBeans or Java Object Serialization. And these applications have needs:

- getting information about classes and its members
- using classes and its members

But the kind of information above are often unavailable at compile-time and it is impossible to write program using unknown classes in strongly typed languages without this API.

Through the Java Reflection API, programmers can handle classes, fields, methods and constructors as objects. For instance, with these metaobject, they can get the name of a class, invoke a method on a object, and so on like following code:

```
Object unknown = ...
Class clazz = unknown.getClass();
Field field = clazz.getField( "name" );
String name = (String) field.get( p );
```

This API is refined for introspection at runtime, especially for security issues at runtime, but it does not have intercession mechanism.

As of version 1.3 of the standard Java2, `java.reflect.Proxy` is provided.

The Java VM uses class loaders to load class files and create class objects. Since class loaders are instances of subclasses of the class ClassLoader

provided as Java API, programmers can define new subclasses of it in Java program. In a subclass of ClassLoader, programmers might change the behavior of program by modifying loaded bytecode. Though execution overheads of loading and modifying bytecodes are not small, it is still useful. Several applications are demonstrated by Liang [56] and Kirby [51].

Even though changing the source of class files (e.g. to remote host beyond a network) is easy to do with this dynamic class loading mechanism, there is difficulty of programming to do more because it is not easy to manipulate bytecodes directly.

MetaXa [52] is an extended Java interpreter that allows structural and behavioral reflection. The system consists of the OS, the application program (the base system), and the meta system. The computation in the base system raises events and that events are delivered to the meta system. The meta system evaluates the events and reacts in a specific manner. All events are handled synchronously. Base-level computation is suspended while the meta object processes the event. This gives the meta level complete control over the activity in the base system. What actually happens depends entirely on the meta object used. A base object also can invoke a method of the meta object directly. This is called explicit meta interaction and is used to control the meta level from the base level.

By limiting the point of alteration only in behavioral reflection, it succeeded in achieving efficient execution of its applications comparatively as runtime MOP. Also, it does not allow syntax extensions in language.

1.5.3 Compile-time MOPs (Meta-Object Protocols)

OpenC++ version 2 [15, 18] provides an extensible C++ language. Its translation is performed according to each type of objects, that is, classes. Since, in higher level languages, the basic constructs are more complicated for compilers, namely, the basic constructs of class-based object-oriented languages are objects, classes and methods, this kind of translation controlling is very effective to extending the behavior of objects, which needs local translation scattered in program.

However, it is not easy to write translation of class declaration with OpenC++. This is because it gives programmer a part of AST (abstract syntax tree) to translate. Though it also gives contextual information with parse tree, its not suitable for handling object-oriented semantics.

1.6 Thesis Organization

The rest of this thesis are organized as follows. First, in Chapter 2, we give the concept of our class-object model for program transformations.

We implemented two program transformation frameworks with this model for Java [36]. The first is for transforming program in source text written in the Java programming language, and the other is for transforming program in Java byte code, which is the compiled code for direct execution on Java virtual machines. The former framework is named OpenJava [82, 81], and the latter is Javassist [17, 20]. Chapter 3 describes the design and implementation of OpenJava, which is a macro system employing the class-object model for source-text to source-text transformation. Chapter 4 describes the design and implementation of Javassist, which is a bytecode editing tool employing the class-object model for bytecode to bytecode transformation.

While these two systems are designed for general purpose of transformations of object-oriented program, we designed a transformational system specially targeting a support of distributed programming. This system is named Addistant and build on top of Javassist. Chapter 5 describes the design and implementation of Addistant. Addistant is also a proof of the expressing power of transformational systems with the class-object model is enough for constructing a non-trivial relatively-large application.

The thesis concludes with a projection into the future in Chapter 6.

Chapter 2

A Class-Object Model

The class-object model proposed in this thesis is the representation of program transformations in a higher-level abstraction. This abstract data model directly represents a logical structure of a class and the type-driven application of transformation.

What we are focusing in this thesis is object-oriented languages with the mechanism of class and inheritance. Including SIMULA-67 [24], which is the founder of object-oriented languages, most of object-oriented languages share these language constructs. Popular object-oriented languages such as Smalltalk [34], C++ [28] and Java [36] also do. The core concept of the proposing model is applicable to transformations of programs in these object-oriented languages with classes and inheritance though we use Java for explanation of object-oriented program and the systems we implemented are based on Java,

Demands for a support of programming according to design patterns are a good motivation for object-oriented transformations. Design patterns help developers choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. A description of communicating objects and classes in a pattern has been polished and used a number of times for a long time. Only if programmers or designers apply patterns to their program adequately, we can say that the design using a pattern should be well sophisticated for the particular context. The problems like coding overhead in implementing a design pattern are what we must address with a programming support but not what we can address by changing the design of program.

In this chapter, we first show motivating examples which represents the typical requirements of object-oriented program transformations. Then, we describe the core concept and design of the class-object model.

2.1 Motivating Examples

Although design patterns [32] are useful guidelines for writing good object-oriented programs, some of the programs written according to design patterns are complex and errorprone and the overall structure of the programs is not easy to understand. First, programmers using design patterns have to write annoying code to implement the patterns because the concept of a design pattern is orthogonal to programming languages such as Smalltalk and C++. Moreover, since most of design patterns are just descriptions apart from code, any line of the code explicitly represents neither which design pattern is used in that program nor which role in that design pattern each class plays. A number of researchers [9, 26, 33] have argued these problems and they have proposed that syntax extensions and extended language constructs help design pattern users write programs and improve the readability of programs written with design patterns.

In order to support programming in some kinds of specific application domains such as distributed programming, extended languages are very useful. But such languages do not have all-round power for all the applications though they are very suitable for applications in each domain. Achieving suitability for all the applications, a language is desired to have several mechanisms supporting primitives in every application domain. However, a multi-paradigm language, which supports many mechanisms of language primitives from the beginning, tends to have too complex specifications to learn. Thus an extensible language, in which programmers can choose appropriate language mechanisms on demand, meets. Moreover, programmers may add a new extension for a new application domain.

The rest of this section shows example applications which need support by extended language mechanisms and motivate us to provide an extensible language.

2.1.1 Implementing Adapter Classes

Suppose that a programmer has to adapt a class Vector (Listing 2.1) to an interface Stack (Listing 2.2), which are declared as follows:

Listing 2.1 *Vector.java*

```
public class Vector
{
    boolean isEmpty();
    Enumeration elements();
    Object lastElement() { ....  }
    void addElement(Object o) { ....  }
    ....
}
```

Listing 2.2 *Stack.java*

```
public interface Stack
```

```
{
    boolean isEmpty();
    Enumeration elements();
    Object peek();
    void push( Object o );
    Object pop();
}
```

The ADAPTER pattern should be used in this case, to:

> convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces [32].

Figure 2.1 shows a structure of the ADAPTER pattern.

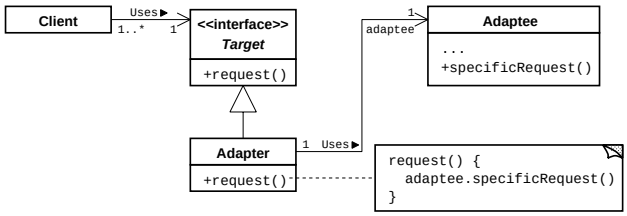

Figure 2.1: A structure of the Adapter pattern.

A class Vector and an interface Stack corresponds to the Adaptee and the Target respectively in Figure 2.1. According to the Adapter pattern, programmers must write a class VectorStack correspondent to the Adapter:

Listing 2.3 *VectorStack.java*

```
public class VectorStack implements Stack
{
    private Vector v;
    VectorStack(Vector v) { this.v = v; }
    boolean isEmpty() { return v.isEmpty(); }
    Enumeration elements() { return v.elements(); }
    Object peek() { return v.lastElement(); }
    void push(Object o) { return v.addElement( o ); }
    Object pop() { ....  }
}
```

The class VectorStack extends the class Vector to have the interface Stack. Here, the class VectorStack is not a subclass of the class Vector so that a single Adapter may work with several Adaptees, that is, the Vector itself and all of its subclasses.

In the case above, programmers are faced with some problems when writing the class VectorStack which plays the role of the Adapter. The problems are:

1. Although the class VectorStack is written for the Adapter of the Adapter pattern, it is difficult to find out this fact from the source code. Which design pattern is used? What is the role of the class VectorStack?
2. The programmers must add a field which holds a reference to an Vector object and a constructor to accept it. Although `isEmpty()` and `elements()` are shared between the class Vector and the class VectorStack, programmers must repeatedly write code for both of them.
3. Programmers cannot reuse any part of this implementation when they apply the Adapter pattern to another coding though the design of a pattern is reusable.
4. In the body of the method `peek()`, only the method `lastElement()` is invoked on the Vector object and the value obtained by this invocation is returned intactly. Such a trivial operation of object also appears in the method `push()`. Describing those operations is a boring task and errorprone.

The above problems are also found in most of other design patterns and these problems have been reported by a number of researchers [74, 69, 8, 63, 26]. Bosch called these problems *traceability loss*, *self problem*, *no reusability* and *implementation overhead* [9].

2.1.2 Implementing Proxy Classes

The PROXY pattern is often used in distributed programming to:

> Provide a surrogate or placeholder for another object to control access to it. [32]

Figure 2.2 shows a structure of the PROXY pattern.

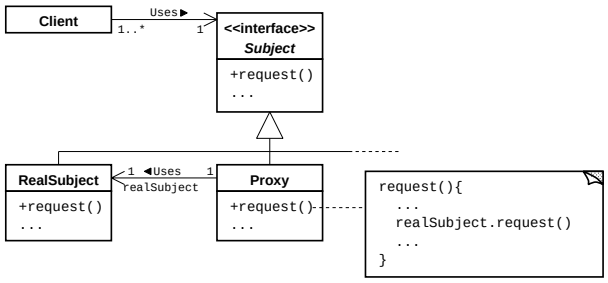

Figure 2.2: A structure of the Proxy pattern.

For making a class Window accessible remotely, a Proxy class WindowProxy must be provided in the PROXY pattern as follows:

Listing 2.4 *Window.java*

```
public class Window extends Component
{
    void addComponent(Component c, int direction) {
        ...
    }
    boolean isVisible() {
        ...
    }
    ....
}
```

Listing 2.5 *WindowProxy.java*

```
public class WindowProxy extends Window
{
    void addComponent(Component c, int dir) {
        int classID = .., objectID = .., methodID = ..;
        Object[] args = new Object{c, new Integer(dir)};
        broker.invoke(classID, objectID, methodID, args);
    }
    boolean isVisible() {
        int classID = .., objectID = .., methodID = ..;
        Object[] args = new Object[]{};
        return broker.invoke(classID, objectID, methodID, args);
    }
    boolean equals(Object obj) {
        int classID = .., objectID = .., methodID = ..;
        Object[] args = new Object[]{obj};
        return broker.invoke(classID, objectID, methodID, args);
    }
    ....
}
```

For being a surrogate of a Window object, the WindowProxy class must provide the same interface of Window, which include the `equals()` method which Window implicitly inherited from its superclass Component. Each method of WindowProxy must implement the packing of method arguments and communication through an object request broker. Even though we can see a lot of similarity between methods, programmers had to write these one by one from scratch if there were not for programming supports.

Code translator support for distributed programming [4] is a common technique helping programmers to write program easily. As for Java, there are several systems which support distributed object programming. Such a system provides a compiler transforming a source program written in ordinal Java into the one which works in distribute environment and run on the ordinal Java VM (Virtual Machine). For instance, the standard Java RMI developed by Sun Microsystems provides `rmic` compiler, and HORB [39] provides `horbc` compiler.

With such systems, programmers can describe a class representing an object which are produced and works on remote systems as if that object

exists on the local system. They can write program handling remote objects without considering how to communicate with these objects through the network. To achieve that transparency of programming remote objects, these compilers accepts a source program written as like local object in the ordinal way and produce a proxy class representing that actual communication to remote systems and a server skeleton class working on remote systems.

The problem is that they must provide each new compiler for every new system supporting distributed programming. In the research level, there are many proposal of such distributed systems. The researchers have to implement a new compiler to provide their proposing systems. In the industry, there are also a lot of distributed systems.

2.1.3 Ordinal Transformational Systems

Ordinal Abstract Syntax Tree Representation

It is not easy for metaprogrammers to implement transformations spreading in source program as a localized translator. For example, it is not easy to write a metaprogram adding a method of a certain name only in case that there is no such methods of the name in a class. It is because of the design of preprocessors; The order to invoke each method `translate()` on node objects of parse tree is fixed in post-order or pre-order. Such design of a preprocessor makes it difficult for meta programmers to translate a part of parse tree according to the information of another part of parse tree.

Because the definition of fields or methods in a class is declarative in most of object-oriented languages, the availability of information can not be fixed neither in post-order nor in pre-order of parse tree. Furthermore, inherited fields and methods are not described in the local parse tree directly. Whether a class has a method or not may be distinguishable after processing the whole source program.

For example, in translating a part of class Panel, it is not easy to test whether the class Panel has a method `validate()` or not because the method `validate()` may be defined after the part being translated or may be inherited from the superclass Container of the class Panel (Figure 2.3).

Consequently, localized translators cannot handle examples shown above though they can handle application examples like what can be handled by Lisp macro and its advanced system.

Syntax-driven Translation

Metaprogrammers must define how to translate program in order to implement their desiring behavior. In the case to change a regular method invocation as an invocation of a method of object on another remote com-

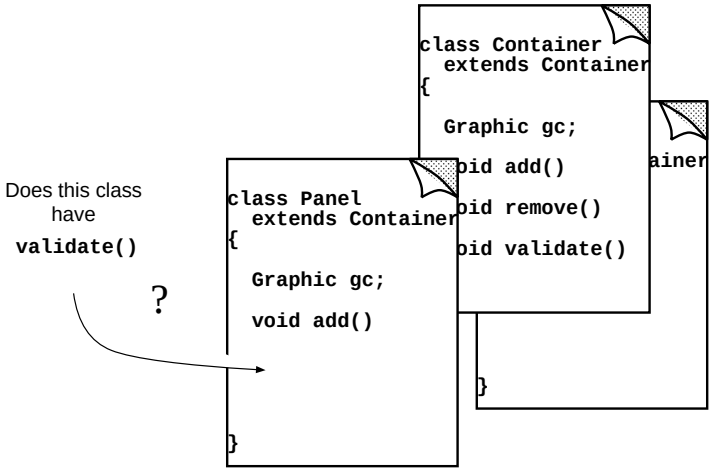

Figure 2.3: Scattered and spreading information over a program.

puter, programmers define a meta-level program implementing an algorithm of source code translation through which the program:

```
p.setName( "Thomas" );
```

are translated into the program which call the method `invoke()` of an object remoteObject, which make a network connection and access to a remote server :

```
remote.invoke( p, "setName", new Object[]{ "Thomas" } );
```

With most of current preprocessors, programmers would represent an algorithm of translation by transforming parse tree or AST while way of defining how to translate varies for each system. Here suppose a language system with the simplest compile-time MOP (Meta-Object Protocol). The MOP should have a model as follows:

- Each node of parse tree would be a metaobject. And classes varies for kinds of syntactic elements such as a variable declaration, expression, or statement.
- Each metaobject has a method `translate()` which transforms the corresponding part of parse tree and returns transformed one.

Before compiling it into an executable code, the system invokes the method `translate()` of the metaobject at the root of the given parse tree and each method `translate()` recursively from the root to its leaves. In order to implement a language extension, programmers can redefine classes with

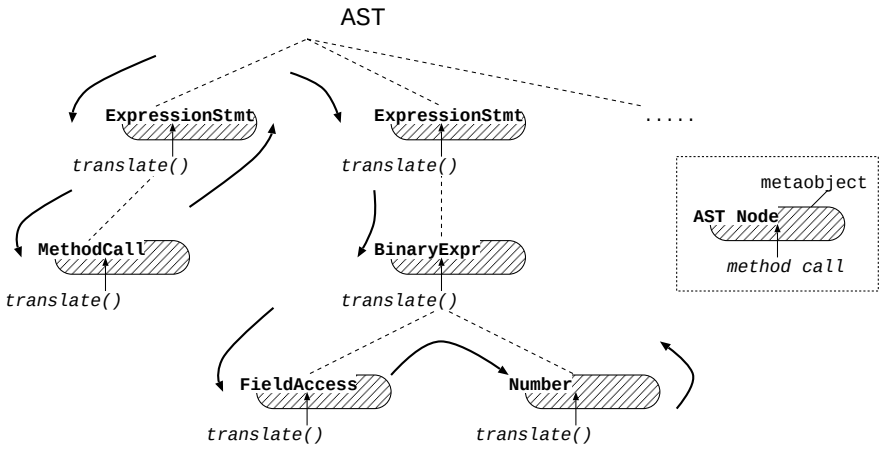

Figure 2.4: Translation with the naive MOP.

another method `translate()` which returns parse tree representing desired behavior (Figure 2.4).

For instance, the above example can be implemented by defining a class RemoteMethodCall(Listing 2.7) substituting the regular class MethodCall(Listing 2.6).

Listing 2.6 *A class for default method call metaobjects*

```
class MethodCall implements ParseTree
{
    Expression ref;
    String name;
    Expressions args;
    MethodCall( Expression ref, String name, Expressions args ) {
         this.ref = ref; this.name = name; this.args = args;
    }
    Expression translate() {
        this.arguments = arguments.translate();
        return this;
    }
}
```

Listing 2.7 *A class for customized method call metaobjects*

```
class RemoteMethodCall extends MethodCall
{
    ...
    MethodCall translate() {
        Expression expr = new ClassName( "Remote" );
        ArrayAllocation aryalloc
            = new ArrayAlloc( "Object", this.args );
        Expressions virtualargs
            = new Expressions( this.ref, this.name, aryalloc );
        MethodCall result
```

```
                = new MethodCall( expr, "invoke", virtualargs );
            return result.translate();
        }
    }
```

It is natural that end users should want to use both remote objects and non remote, local, objects in the same source program. However, switching several translations with the naive MOPs described above is very difficult because they only distinguish syntactical difference but its semantical difference. For example, there are two method call expressions in the code below:

```
String name = info.getName()
remote.setName( name )
```

With the naive MOPs, it is difficult to make the access to `info` be remote method invocation and not to make the access to `remote` be regular one.

2.2 Modeling Object-Oriented Programs as Class-Objects

A translator must handle *object-orientation* in order to provide code generation or modification for addressing the problems stated in the previous section with transformations. The class-object model proposed in this thesis is a model for metaprogramming interfaces, through which metaprogram can directly state this object-orientation. This section describes the object-orientation treated by the model.

Classes play one of the most important roles in the logical structure of program in object-oriented program written in class-based languages like Java. Classes are declarative language constructs and they provide declarative methods or fields as their attributes. Thus, the data structures of abstract syntax tree or raw bytecode instructions are not suitable for directly describing object-oriented high-level transformation in a metaprogram. In addition to the ability of directly handling data flows which were addressed by ordinal program transformation systems, object-oriented program transformation requires the direct support of declarative language constructs.

The class-object model is the abstract data model for representing transformations of an abstract data type denoted by a class. A class-object, which is an instance of class-object model, has two-aspects. One is a logical structure of a class, and the other is a transformation framework suitable for object-oriented program transformations.

2.2.1 Representing Abstract Data Type

Object-oriented programs are made up of objects. An *object* packages both data and the procedures that operate on that data. The procedures are called *methods*. An object performs an operation when it receives a *message* from a *client*. Messages are the only way to get an object to execute an operation.

Interfaces are fundamental in object-oriented systems. Objects are known only through their interfaces. There is no way to know anything about an object or to ask it to do anything without going through its interface. An object's interface says nothing about its implementation — different objects are free to implement requests differently. That means two objects having completely different implementations can have identical interfaces.

Types

Every operation declared by an object specifies the operation's name, the objects it takes as parameters, and the operation's name, the objects it takes as parameters, and the operation's return value. This is known as the operation's *signature*. The set of all signatures defined by an object's operations is called the interface to the object. An object's interface characterizes the complete set of requests that can be sent to the object. Any request that matches a signature in the object's interface may be sent to the object.

A *type* is a name used to denote a particular interface. We speak of an object as having the type "Window" if it accepts all requests for the operations defined in the interface named "Window." An object may have many types, and widely different objects can share a type. Part of an object's may have many types, and widely different objects can share a type. Part of an object's interface may be characterized by one type, and other parts by other types. Two objects of the same type need only share parts of their interfaces. Interfaces can contain other interfaces as subsets. We say that a type is a *subtype* of another if its interface contains the interface of its *supertype*. Often we speak of a subtype *inheriting* the interface of its supertype.

In a class-based object-oriented language, programmers describe a particular interface by declaring a *type*. We speak of an object as having the type "Window" if it accepts all requests for the operations defined in the interface named "Window." An object may have many types, and widely different objects can share a type. Part of an object's interface may be characterized by one type, and other parts by other types. Two objects of the same type need only share parts of their interfaces. Interfaces can contain other interfaces as subsets. We say that a type is a *subtype* of another if its interface contains the interface of its *supertype*. Often we speak of a subtype

inheriting the interface of its supertype.

In Java, programmers use the reserved words `class` and `interface` for declaring types. Every object has a class that defines its data and behavior. Each class has two kinds of members except member classes:

- Fields are data variables associated with a class and its objects. Fields store results of computations performed by the class.
- Methods contain the executable code of a class. Methods are built from statements. The way in which methods are invoked, and the statements contained within those methods, is what ultimately directs program execution.

Inheritance

One of the major benefits of object orientation is the ability to *extend*, or *subclass*, the behavior of an existing class and continue to use code written for the original class when acting upon an instance of the subclass. The original class is known as the *superclass*. When you extends a class to create a new class, the new extended class *inherits* fields and methods of the superclass.

If the subclass does not specifically override the behavior of the superclass, the subclass inherits all the behavior of its superclass because it inherits the fields and methods of its superclass. In addition, the subclass can add new fields and methods and so add new behavior.

Let's look at an example of extending a class. Here we extends a Point class to represent a pixel that might be shown on a screen. The new Pixel class requires a color in addition to `x` and `y` coordinates declared in Point:

```
class Pixel extends Point {
    Color color;

    public void clear() {
        super.clear();
        color = null;
    }
}
```

Pixel extends both the data and behavior of its Point superclass. Pixel extends the data by adding a field named `color`. Pixel also extends the behavior of Point by overriding Point's `clear()` methods.

Pixel objects can be used by any code designed to work with Point objects. If a method expects a parameter of type Point, you can hand it a Pixel object and it just works. All the Point code can be used by anyone with a Pixel in hand. This feature is known as *polymorphism* — a single object like

Pixel can have many (*poly*) forms (*-morph*) and can be used as both a Pixel object and a Point object.

Pixel's behavior extends Point's behavior. Extended behavior can be entirely new (adding color in this example) or can be a restriction on old behavior that follows all the original requirements. An example of restricted behavior might bePixel objects that live inside some kind of Screen object, restricting `x` and `y` to the dimensions of the screen. If the original Point class did not forbid restrictions for coordinates, a class with restricted range would not violate the original class' behavior.

An extended class often *overrides* the behavior of its superclass by providing new implementations of one or more of the inherited methods. To do this the extended class defines a method with the same signature and return type as a method in the superclass. In the Pixel example, we override `clear()` to obtain the proper behavior that Pixel requires. The `clear()` that Pixel inherited from Point knows only about Point's fields but obviously can't know about the new `color` field declared in the Pixel subclass.

Dynamic Binding

When a request is sent to an object, the particular operation that's performed depends on both the request and the receiving object. Different objects that support identical requests may have different implementations of the operations that fulfill these requests. The run-time association of a request to an object and one of its operations is known as *dynamic binding*.

Dynamic binding means that issuing a request doesn't commit you to a particular implementation until run-time. Consequently, you can write programs that expect an object with a particular interface, knowing that any object that has the correct interface will accept the request. Moreover, dynamic binding lets you substitute objects that have identical interfaces for each other at run-time. This substitutability is known as *polymorphism*, and it's a key concept in object-oriented systems. It lets a client object make few assumptions about other objects beyond supporting a particular interface. Polymorphism simplifies the definitions of clients, decouples objects from each other, and lets them vary their relationships to each other at run-time.

Capsulation

If every member of every class and object was accessible to every other class and object then understanding, debugging and maintaining programs would be an almost impossible task. The contracts presented by classes could not be relied upon because any piece of code could directly access a field and change it in such a way as to violate the contract. One of the strengths of object-oriented programming is its support for *encapsulation* and *data-*

hiding. To achieve this we need a way to control who has access to what members of a class or interface, and even access to the class or interface itself.

In Java, this control is specified by using *access modifiers* on class, interface and member declarations.

private members declared `private` are accessible only in the class itself.

package members declared with no access modifier are accessible in classes in the same package, as well as in the class itself.

protected members declared `protected` are accessible in subclasses of the class, in classes in the same package, and in the class itself.

package members declared `public` are accessible anywhere the class is accessible.

The available methods differs depending on from which class the class (objects) is accessed.

2.2.2 How to Apply Transformations

Controlling the scope of translation is important. A system should provide the ability to apply translation to pieces of programs only when they satisfy given conditions and only in a restricted region of programs. To incorporate several extensions in a language, the system should control the scope of its translation by each extension otherwise unexpected collisions among extensions occurs. Without any scope control, programmers must carefully define their meta program for the compatibility against other extensions. At least, it must be specified how the system behaves when any collision occurs.

Type-driven Translation

For extending the behavior of a class, it is useful for programmers if a translation can be applied only to code pieces related to the class. The *type-driven translation* is the mechanism of a translation scope control for that. With this mechanism, Translations are performed according to types of each object, that is, classes.

A possible design for this mechanism is to let metaprogrammers to describe a *metaclass* which define the transformations of its instances, base classes. Every base class corresponds to an instance of a metaclass. A default metaclass is defined not to make any translation. Meta programmers define a new subclass of the default class to implement their desiring extensions and specify the relation of this new metaclass and appropriate classes. Then, the system applies that translation only to program pieces of the objects which instantiates the class related to the metaclass.

The following is a simple base level code declaring a class Hello.

Listing 2.8 *Hello.oj*

```
public class Hello instantiates VerboseClass
{
    public String say() {
        return "Hello World.";
    }
}
```

The notation:

```
class C instantiates M
```

specifies the class C is related to the metaclass M, that is, the class object representing the class C is an instance of the class M. As a result, the translation around objects of type C will be performed according to the definition in the class M.

Here, the class VerboseClass is defined to change the behavior of method call on its instance class object to the one which prints out the called method's name, for the purpose of debugging or something. Then, the notation in Listing 2.8 makes Hello objects have an additional behavior of printing out the method's name when it is called.

From the point of view in extending Java, an object-oriented language, it is natural to switch the extension by the type of objects. We believe this method of scope control is one of the best ways, though there's several alternatives for the choice of the translation scope controls, such as delegation in MPC++ [43], system mixins in EPP [42] or pattern matching in A* [55]. Our scope controlling method is founded upon OpenC++'s and it has been demonstrated to be very useful for many applications by Chiba [15, 16].

Translation at Callee-side and Caller-side

Here, what region of source code is to be translated as the part related to an object is discussed. The parts of source code are categorized into three from this point of view. The categories of relation to an object are:

1. callee-side: the declaration of the class
2. caller-side: where accesses to the object performed occurs
3. non related parts

Parts of source code in the category 3 are to be protected from translation around the object. The part of 1 is *callee* side, where the class is declared with its field declarations, method declarations and constructor declarations described. And the rest, 2 is *caller* side, where accesses to the object through its fields, methods or constructors of the class are performed.

To implement a transformation which makes a method print out their name for each invocation, one candidate is to translate the method declaration in the declaration of the class Hello into the program as follows:

```
public class Hello
{
    public String say() {
        System.out.println( "say() is called." );
        String result = original_say();
        System.out.println( "done." );
        return result;
    }
    private String original_say() {
        return "Hello World.";
    }
}
```

In order to change the behavior of Hello objects, another candidate is to translate the part of each program where methods of the class Hello are called. It is also possible to achieve the purpose of the metaclass VerboseClass, as same as callee-side translation, by translating the code below:

```
Hello a = new Hello();
String str = a.say();
```

into the code below:

```
Hello a = new Hello();
String str = invoke_Hello_say( a );
```

using a function:

```
String invoke_Hello_say( Hello obj ) {
    System.out.println( "say() is called." );
    String result = obj.say();
    System.out.println( "done." ) );
    return result;
}
```

The applicability of two kind of translation is different though the example above seems to be able to be achieved by any side of translation. In fact, there are trade-offs between the caller-side translation and the callee-side translation.

First, we present a limitation of callee-side translation. In order to use large numbers of fine-grained objects efficiently, a programming technique to give rather shared objects than objects to be generated each time if the

shared objects can be used interchangeably. And such a technique is well known as the Flyweight design pattern [32]. Here, suppose a simple program providing this feature as follows:

```
public class BitmapFont
{
    Image bitmap;
    private FontFace(Font f, int height) {
        bitmap = generateImage( f, height )
    }

    static Font[] fontcache = null, null, ..;

    public static genBitmapFont(Font f, int height) {
        if (height < 15) {
            if (fontcache[height] == null)
                fontcache[height] = new BitmapFont(f, height)
            return fontcache[height]
        }
        return new FontFace( f, height );
    }
}
```

In this case, this program saves system memory and computation time by providing a method `genBitmapFont()` which recycles generated objects and by hiding the constructor of the class BitmapFont.

In order to implement this optimization transparent against its users, a caller-side translation can replace the constructor invocation by a method call for `genBitmapFont()`. However, such implementation seems to be impossible with callee-side translations.

Then, we present the problem of caller-side translation. Suppose that a class Hello is a subclass of a class Object. The class Object has an instance method `toString()` which return a String object representing the identical string of this Object object and the class Hello overrides that method to return a String object `"Hello"`. If we execute the program below, which is compiled and run on the regular Java environment:

```
Object obj = new Hello();
System.out.println( obj.toString() );
```

the Java VM prints out as follows:

```
"Hello"
```

This means the method `toString()` is chosen by the active type of the object `obj` but not by the static type. This causes thanks to the dynamic

binding mechanism of the object-oriented language. However, at compile-time, the type of `obj` can only be detected to be the superclass Object at least since the variable `obj` is binded to it in this example. Generally, it is impossible to determine the active type of `obj` at compile-time. Thus even if a caller-side translation defined for the class Hello, the system cannot apply it to `obj`.

The consistency of changing behavior is lost in the translation of instance member accesses at caller-side, though it is still useful for the purpose of optimization and it can keep consistency for class (`static`) member accesses. For changing behavior of object according to its type, callee-side translation is useful to keep the consistency of translation. Thus the system must provide powerful callee-side translation in addition to translating at caller-side.

Chapter 3

OpenJava

This chapter presents OpenJava, which is a macro system that we have developed for Java. With traditional macro systems designed for non object-oriented languages, it is difficult to write a number of macros typical in object-oriented programming since they require the ability to access a logical structure of programs. One of the drawbacks of traditional macro systems is that abstract syntax trees are used for representing source programs. This chapter first points out this problem and then shows how OpenJava addresses this problem. A key idea of OpenJava is to use metaobjects, which was originally developed for reflective computing, for representing source programs.

Reflection is a technique for changing the program behavior according to another program. From software engineering viewpoint, reflection is a tool for separation of concerns and thus it can be used for letting programmers write a program with higher-level abstraction and with good modularity. For example, a number of reflective systems provide metaobjects for intercepting object behavior, that is, method invocations and field accesses. Those metaobjects can be used for *weaving* several programs separately written from distinct aspects, such as an application algorithm, distribution, resource allocation, and user interface, into a single executable program.

However, previous reflective systems do not satisfy all the requirements in software engineering. Although the abstraction provided by the metaobjects for intercepting object behavior is easy to understand and use, they can be used for implementing only limited kinds of separation of concerns. Moreover, this type of reflection often involves runtime penalties. Reflective systems should enable more fine-grained program weaving and perform as much reflective computation as possible at compile time for avoiding runtime penalties.

On the other hand, a typical tool for manipulating a program at compile time has been a macro system. It performs textual substitution so that a

particular aspect of a program is separated from the rest of that program. For example, the C/C++ macro system allows to separate the definition of a constant value from the rest of a program, in which that constant value is used in a number of distinct lines. The Lisp macro system provides programmable macros, which enables more powerful program manipulation than the C/C++ one. Also, since macro expansion is done at compile time, the use of macros does not imply any runtime penalties. However, the abstraction provided by traditional macro systems is not sophisticated; since macros can deal with only textual representation of a program, program manipulation depending on the semantic contexts of the program cannot be implemented with macros.

This chapter proposes a macro system integrating good features of the reflective approach, in other words, a compile-time reflective system for not only behavioral reflection but also structural reflection. A key idea of our macro system, called *OpenJava*, is that macros (meta programs) deal with class-objects representing logical entities of a program instead of a sequence of tokens or abstract syntax trees (ASTs). Since the class-objects abstract both textual and semantic aspects of a program, macros in OpenJava can implement more fine-grained program weaving than in previous reflective systems. They can also access semantic contexts if they are needed for macro expansion. This chapter presents that OpenJava can be used to implement macros for helping complex programming with a few design patterns.

In the rest of this chapter, section 3.2 presents a problem of ordinary macro systems and section 3.3 discusses the design and implementation of OpenJava, which addresses this problem. We compare OpenJava with related work in section 3.4. Finally, section 3.5 concludes this chapter.

3.1 Problems with Ordinary Macros

Macro systems have been typical language-extension mechanisms. With C/C++'s `#define` macro system, programmers can specify a symbol or a function call to be replaced with another expression, although this replacement is simple token-based substitution. In Common Lisp, programmers can write more powerful macros. However, even such powerful macros do not cover all requirements of object-oriented languages programming.

3.1.1 Programmable Macros

Macros in Common Lisp are programmable macros. They specify how to replace an original expression in Common Lisp itself. A macro function receives an AST (abstract syntax tree) and substitutes it for the original expression. Since this macro system is powerful, the object system of Common Lisp (CLOS) is implemented with this macro system.

Programmable macros have been developed for languages with more complex syntax like C. MS^2 [84] is one of those macro systems for C. Macro functions are written in an extended C language providing special data structure representing ASTs. The users of MS^2 can define a new syntax and how it is expanded into a regular C syntax. The parameter that a macro function receives is an AST of the code processed by that macro function.

One of the essential issue in designing a programmable macro system is a data structure representing an original source program. Another essential issue is how to specify where to apply each macro in a source program. For the former, most systems employed ASTs. For the latter, several mechanisms were proposed.

In Common Lisp and MS^2, a macro is applied to expressions or statements beginning with the trigger word specified by the macro. For example, if the trigger word is `unless`, all expressions beginning with `unless` are expanded by that macro. In this way, they cannot use macros without the trigger words. For instance, it is impossible to selectively apply a macro to only `+` expressions for adding string objects.

Some macro systems provide fine-grained control of where to apply a macro. In A^* [55], a macro is applied to expressions or statements matching a pattern specified in the BNF. In EPP [42], macros are applied to a specified syntax elements like `if` statements or `+` expressions. There's no need to put any trigger word in front of these statements or expressions.

3.1.2 Representation of Object-Oriented Programs

Although most of macro systems have been using ASTs for representing a source program, ASTs are not the best representation for all macros: some macros typical in object-oriented programming require a different kind of representation. ASTs are purely textual representation and independent of logical or contextual information of the program. For example, if an AST represents a binary expression, the AST tells us what the operator and the operands are but it never tells us the types of the operands. Therefore, writing a macro is not possible with ASTs if the macro expansion depends on logical and contextual information of that binary expression.

There is a great demand for the macros depending on logical and contextual information in object-oriented programming. For example, some of design patterns [32] require relatively complicated programming. They often require programmers to repeatedly write similar code. [9]. To help this programming, several researchers have proposed to extend a language to provide new language constructs specialized for particular patterns. [9, 33] Those constructs should be implemented with macros although they have been implemented so far by a custom preprocessor. This is because macros implementing those constructs depend on the logical and contextual infor-

mation of programs and thus they are not implementable on top of the traditional AST-based macro systems.

Suppose that we write a macro for helping programming with the OBSERVER [32] pattern, which is for describing one-to-many dependency among objects. This pattern is found in the Java standard library although it is called the event-and-listener model. For example, a Java program displays a menu bar must define a listener object notified of menu-select events. The listener object is an instance of a class MyMenuListener implementing interface MenuListener:

```
class MyMenuListener implements MenuListener {
   void menuSelected(MenuEvent e) { .. }
   void menuDeselected(MenuEvent e) { return; }
   void menuCanceled(MenuEvent e) { return; }
}
```

This class must declare all the methods for event handling even though some events, such as the menu cancel event, are simply ignored.

We write a macro for automating declaration of methods for handling ignored events. If this macro is used, the definition of MyMenuListener should be re-written into:

```
class MyMenuListener follows ObserverPattern
    implements MenuListener
{
    void menuSelected(MenuEvent e) { .. }
}
```

The `follows` clauses specifies that our macro ObserverPattern is applied to this class definition. The declarations of `menuDeselected()` and `menuCanceled()` are automated. This macro first inspects which methods declared in the interface MenuListener are not implemented in the class MyMenuListener. Then it inserts the declarations of these methods in the class MyMenuListener.

Writing this macro is difficult with traditional AST-based macro systems since it depends on the logical information of the definition of the class MyMenuListener. If a class definition is given as a large AST, the macro program must interpret the AST and recognize methods declared in MenuListener and MyMenuListener. The macro program must also construct ASTs representing the inserted methods and modify the original large AST to include these ASTs. Manipulating a large AST is another difficult task. To reduce these difficulties, macro systems should provide logical and contextual information of programs for macro programs. There are only a few macro systems providing the logical information. For example, XL [59] is one of those systems although it is for a functional language but not for an object-oriented language.

3.2 OpenJava

OpenJava is our advanced macro system for Java. In OpenJava, macro programs can access the data structures representing a logical structure of the programs. We call these data structure class-objects. This section presents the design of OpenJava.

3.2.1 Macro Programming in OpenJava

OpenJava produces an object representing a logical structure of class definition for each class in the source code. This object is called a class-object. A class-object also manages macro expansion related to the class it represents. Programmers customize the definition of the class-objects for describing macro expansion. We call the class for the class-object *metaclass*. In OpenJava, the metaprogram of a macro is described as a metaclass. Macro expansion by OpenJava is divided into two: the first one is macro expansion of class declarations (callee-side), and the second one is that of expressions accessing classes (caller-side).

Applying Macros

Fig. 3.1 shows a sample using a macro in OpenJava. By adding a clause `instantiates M` in just after the class name in a class declaration, the programmer can specify that the class metaobject for the class is an instance of the metaclass M. In this sample program, the class-object for MyMenuListener is an instance of ObserverClass. This metaobject controls macro expansion involved with MyMenuListener. The declaration of ObserverClass is described in regular Java as shown in Fig. 3.2.

```
class MyMenuListener
   instantiates ObserverClass
   extends MyObject
   implements MenuListener
{ .... }
```

Figure 3.1: Application of a macro in OpenJava

Every metaclass must inherit from the metaclass OJClass, which is a built-in class of OpenJava. The `translateDefinition()` in Fig. 3.2 is a method inherited from OJClass, which is invoked by the system to make macro expansion. If an `instantiates` clause in a class declaration is found, OpenJava creates an instance of the metaclass indicated by that `instantiates` clause, and assigns this instance to the class-object representing that de-

```
class ObserverClass
   extends OJClass
{
   void translateDefinition() { ... }
   ....
}
```

Figure 3.2: A macro in OpenJava

clared class. Then OpenJava invokes `translateDefinition()` on the created class metaobject for macro expansion on the class declaration later.

Since the `translateDefinition()` declared in OJClass does not perform any translation, a subclass of OJClass must override this method for the desired macro expansion. For example, `translateDefinition()` can add new member methods to the class by calling other member methods in OJClass. Modifications are reflected on the source program at the final stage of the macro processing.

Describing a Metaprogram

The method `translateDefinition()` implementing the macro for the OBSERVER pattern in section 3.1.2 is shown in Fig. 3.3. This metaprogram first obtains all the member methods (including inherited ones) defined in the class by invoking `getMethods()` on the class-object. Then, if a member method declared in interfaces is not implemented in the class, it generates a new member method doing nothing and adds it to the class by invoking `addMethod()` on the class-object.

```
void translateDefinition() {
   OJMethod[] m = this.getMethods(this);
   for (int i = 0; i < m.length; ++i) {
      OJModifier modif = m[i].getModifiers();
      if (modif.isAbstract()) {
         OJMethod n = new OJMethod(this,
            m[i].getModifiers().removeAbstract(),
            m[i].getReturnType(), m[i].getName(),
            m[i].getParameterTypes(),
            m[i].getExceptionTypes(),
            makeStatementList("return;"));
         this.addMethod(n);
      }
   }
}
```

Figure 3.3: `translateDefinition()` in ObserverClass

As a class is represented by a class-object, a member method is also represented by a method metaobjects. In OpenJava, classes, member methods,

member fields, and constructors are represented by instances of the class OJClass, OJMethod, OJField, and OJConstructor, respectively. These metaobject represent logical structures of class and member definitions. They are easy to handle, compared to directly handling large ASTs representing class declarations and collecting information scattered in these ASTs.

3.2.2 Class-Objects

As shown in section 3.1, a problem of ordinary macro systems is that their primary data structure is ASTs (abstract syntax trees) but they are far from logical structures of programs in object-oriented languages. In object-oriented languages like Java, class definitions play an important role as a logical structure of programs. Therefore, OpenJava employs the class-object model, which was originally developed for reflective computing, for representing a logical structure of a program. The class-objects make it easy for meta programs to access a logical structure of program.

Hiding Syntactical Information

In Java, programmers can use various syntax for describing the logically same thing. These syntactical differences are absorbed by the metaobjects. For instance, there are two notations for declaring a String array member field:

```
String[] a;
String b[];
```

Both a and b are String array fields. It would be awkward to write a metaprogram if the syntactical differences of the two member fields had to be considered. Thus OJField provides only two member methods `getType()` and `setType()` for handling the type of a member field. `getType()` on the OJField metaobjects representing a and b returns a class-object representing the array type of the class String.

Additionally, some elements in the grammar represent the same element in a logical structure of the language. If one of these element is edited, the others are also edited. For instance, the member method `setName()` in OJClass for modifying the name of the class changes not only the class name after the `class` keyword in the class declaration but also changes the name of the constructors.

Logically Structured Class Representation

Simple ASTs, even arranged and abstracted well, cannot properly represent a logical structure of a class definition. The data structure must be carefully designed to corresponded not only to the grammar of the language but also

to the logical constructs of the language like classes and member methods. Especially, it makes it easy to handle the logical information of program including association between names and types.

For instance, the member method `getMethods()` in OJClass returns all the member methods defined in the class which are not only the methods immediately declared in the class but also the inherited methods. The class-objects contain type information so that the definition of the super class can be accessible.

3.2.3 Class-Object API in Details

The root class for class-objects is OJClass. The member methods of OJClass for obtaining information about a class are shown in Tab. 3.1 and Tab. 3.2. They cover all the attributes of the class. In OpenJava, all the types, including array types and primitive types like `int`, have corresponding class-objects. Using the member methods shown in Tab. 3.1, metaprograms can inspect whether a given type is an ordinary class or not.

Tab. 3.3 gives methods for modifying the definition of the class. Metaprograms can override `translateDefinition()` in OJClass so that it calls these methods for executing desired modifications. For instance, the example shown in Fig. 3.3 adds newly generated member methods to the class with `addMethod()`.

Table 3.1: Member methods in OJClass for non-class types

`boolean isInterface()` Tests if this represents an interface type.
`boolean isArray()` Tests if this represents an array type.
`boolean isPrimitive()` Tests if this represents an primitive type.
`OJClass getComponentType()` Returns a class-object for the type of array components.

Metaobjects Obtained through Class-Objects

The method `getSuperclass()` in OJClass, which is used to obtain the superclass of the class, returns a class-object instead of the class name (as a string). As the result, metaprogram can use the returned class-object to directly obtain information about the superclass. OpenJava automatically generates class metaobjects on demand, even for classes declared in another

Table 3.2: Member methods in OJClass for introspection (1)

`String getPackageName()`
: Returns the package name this class belongs to.

`String getSimpleName()`
: Returns the unqualified name of this class.

`OJModifier getModifiers()`
: Returns the modifiers for this class.

`OJClass getSuperclass()`
: Returns the superclass declared explicitly or implicitly.

`OJClass[] getDeclaredInterfaces()`
: Returns all the declared superinterfaces.

`StatementList getInitializer()`
: Returns all the static initializer statements.

`OJField[] getDeclaredFields()`
: Returns all the declared fields.

`OJMethod[] getDeclaredMethods()`
: Returns all the declared methods.

`OJConctructor[] getDeclaredConstructors()`
: Returns all the constructors declared explicitly or implicitly.

`OJClass[] getDeclaredClasses()`
: Returns all the member classes (inner classes).

`OJClass getDeclaringClass()`
: Returns the class declaring this class (outer class).

Table 3.3: Member methods in OJClass for modifying the class

`String setSimplename(String name)` Sets the unqualified name of this class.
`OJModifier setModifiers(OJModifier modifs)` Sets the class modifiers.
`OJClass setSuperclass(OJClass clazz)` Sets the superclass.
`OJClass[] setInterfaces(OJClass[] faces)` Sets the superinterfaces to be declared.
`OJField removeField(OJField field)` Removes the given field from this class declaration.
`OJMethod removeMethod(OJMethod method)` Removes the given method from this class declaration.
`OJConstructor removeConstructor(OJConstructor constr)` Removes the given constructor from this class declaration.
`OJField addField(OJField field)` Adds the given field to this class declaration.
`OJMethod addMethod(OJMethod method)` Adds the given method to this class declaration.
`OJConstructor addConstructor(OJConstructor constr)` Adds the given constructor to this class declaration.

source file or for classes available only in the form of bytecode, that is, classes whose source code is not available.

The returned value of the member method `getModifiers()` in Tab. 3.2 is an instance of the class OJModifier. This class represents a set of class modifiers such as `public`, `abstract` or `final`. Metaprograms do not have to care about the order of class modifiers because OJModifier hides such useless information.

The class OJMethod, which is the return type of `getDeclaredMethods()` in OJClass, represents a logical structure of a method. Thus, similarly to the class OJClass, this class has member methods for examining or modifying the attributes of the method. Some basic member methods in OJMethod are shown in Tab. 3.4. Any type information obtained from these methods is also represented by a class-object. For instance, `getReturnType()` returns a class-object as the return type of the method. This feature of OJMethod is also found in OJField and OJConstructor, which respectively represent a member field and a constructor.

The class StatementList, which is the return type of the member method `getBody()` in the class OJMethod, represents the statements in a method body. An instance of StatementList consists of objects representing either expressions or statements. StatementList objects are AST-like data struc-

Table 3.4: Basic methods in OJMethod

`String getName()`
: Returns the name of this method.

`OJModifier getModifiers()`
: Returns the modifiers for this method.

`OJClass getReturnType()`
: Returns the return type.

`OJClass[] getParameterTypes()`
: Returns the parameter types in declaration order.

`OJClass[] getExceptionTypes()`
: Returns the types of the exceptions declared to be thrown.

`String[] getParameterVariables()`
: Returns the parameter variable names in declaration order.

`StatementList getBody()`
: Returns the statements of the method body.

`String setName(String name)`
: Sets the name of this method.

`OJModifier setModifiers(OJModifier modifs)`
: Sets the method modifiers.

`OJClass setReturnType()`
: Sets the return type.

`OJClass[] setParameterTypes()`
: Sets the parameter types in declaration order.

`OJClass[] setExceptionTypes()`
: Sets the types of the exceptions declared to be thrown.

`String[] setParameterVariables()`
: Sets the parameter variable names in declaration order.

`StatementList setBody()`
: Sets the statements of the method body.

tures although they contain type information. This is because we thought that the logical structure of statements and expressions in Java can be well represented with ASTs.

Logical Structure of a Class

Tab. 3.5 shows the member methods in OJClass handling a logical structure of a class. Using these methods, metaprograms can obtain information considering class inheritance and member hiding. Although these member methods can be implemented by combining the member methods in Tab.3.2, they are provided for convenience. We think that providing these methods is significant from the viewpoint that class-objects represent a logical structure of a program.

Table 3.5: Member methods in OJClass for introspection (2)

`OJClass[] getInterfaces()` Returns all the interfaces implemented by this class or the all the superinterfaces of this interface.
`boolean isAssignableFrom(OJClass clazz)` Determines if this class/interface is either the same as, or is a superclass or superinterface of, the given class/interface.
`OJMethod[] getMethods(OJClass situation)` Returns all the class available from the given situation, including those declared and those inherited from superclasses/superinterfaces.
`OJMethod getMethod(String name, OJClass[] types, OJClass situation)` Returns the specified method available from the given situation.
`OJMethod getInvokedMethod(String name, OJClass[] types, OJClass situation)` Returns the method, of the given name, invoked by the given arguments types, and available from the given situation.

In considering the class inheritance mechanism, the member methods defined in a given class are not only the member methods described in that class declaration but also the inherited ones. Thus, method metaobjects obtained by invoking `getMethods()` on a class-object include the methods explicitly declared in its class declaration but also the methods inherited from its superclass or superinterfaces.

Moreover, accessibility of class members is restricted in Java by member modifiers like `public`, `protected` or `private`. Thus, `getMethods()` returns only the member methods available from the class specified by the argument. For instance, if the specified class is not a subclass or in the same package, `getMethods()` returns only the member methods with `public` modifier. In

Fig. 3.3, since the metaprogram passes `this` to `getMethods()`, it obtains all the member methods defined in that class.

3.2.4 Type-Driven Translation

As macro expansion in OpenJava is managed by metaobjects corresponding to each class (type), this translation is said to be type-driven. In the above example, only the member method `translateDefinition()` of OJClass is overridden to translate the class declarations of specified classes (callee-side translation).

In addition to the callee-side translation, `OJClass` provides a framework to translate the code related to the corresponding class spreading over whole program selectively (caller-side translation). The parts related to a certain class is, for example, instance creation expressions or field access expressions.

Here, we take up an example of a macro that enables programming with the FLYWEIGHT [32] pattern to explain this mechanism. This design pattern is applied to use objects-sharing to support large numbers of fine-grained objects efficiently. An example of macro supporting uses of this pattern would need to translate an instance creation expression of a class Glyph:

```
new Glyph('c')
```

into a class method call expression:

```
GlyphFactory.createCharacter('c')
```

The class method `createCharacter()` returns an object of Glyph correspondent to the given argument if it was already generated, otherwise it creates a new object to return. This way, the program using Glyph objects automatically shares an object of Glyph representing a font for a letter `c` without generating several objects for the same letter. In ordinary programming using Glyph objects with the FLYWEIGHT pattern, programmers must explicitly write `createCharacter()` in their program with creations of Glyph objects. With a support of this macro, instance creations can be written in the regular `new` syntax and the pattern is used automatically.

In OpenJava, this kind of macro expansions are implemented by defining a metaclass FlyweightClass to be applied to the class Glyph. This metaclass overrides the member method `expandAllocation()` of OJClass as in Fig.3.4. This method receives a class instance creation expression and returns a translated expression. The system of OpenJava examines the whole source code and apply this member method to each Glyph instance creation expression to perform the macro expansion.

The member method `expandAllocation()` receives an AllocationExpression object representing a class instance creation expression and an Environment object representing the environment of this expression. The Environ-

```
Expression expandAllocation(AllocationExpression expr, Environment env) {
   ExpressionList args = expr.getArguments();
   return new MethodCall(this, "createCharacter", args);
}
```

Figure 3.4: Replacement of class instance expressions

ment object holds name binding information such as type of variable in the scope of this expression.

OpenJava uses type-driven translation to enable the comprehensive macro expansion of partial code spreading over various places in program. In macro systems for object-oriented programming languages, it is not only needed to translate a class declaration simply but translating expressions using the class together is also needed. In OpenJava, by defining a methods like `expandAllocation()`, metaprogrammers can selectively apply macro expansion to the limited expressions related to classes controlled by the metaclass. This kind of mechanism has not been seen in most of ordinary macro systems except some systems like OpenC++ [15]. Tab. 3.6 shows the primary member methods of OJClass which can be overridden for macro expansion at caller-side.

Table 3.6: Member methods for each place where the macro-expansion is applied

Member method	Place applied the macro expansion to
`translateDefinition()`	Class declaration
`expandAllocation()`	Class instance allocation expression
`expandArrayAllocation()`	Array allocation expression
`expandTypeName()`	Class name
`expandMethodCall()`	Method class expression
`expandFieldRead()`	Field-read expression
`expandFieldWrite()`	Field-write expression
`expandCastedExpression()`	Casted expression from this type
`expandCastExpression()`	Casted expression to this type

3.2.5 Translation Mechanism

Given a source program, the processor of OpenJava:

1. Analyzes the source program to generate a class-object for each class.
2. Invokes the member methods of class-objects to perform macro expansion.
3. Generates the regular Java source program reflecting the modification made by the class-objects.

4. Executes the regular Java compiler to generate the corresponding byte code.

The Order of Translations

Those methods of OJClass whose name start from `expand` performs caller-side translation, and they affect expressions in source program declaring another class C. Such expressions may also be translated by `translateDefinition()` of the class-object of C as callee-side translation. Thus different class-objects affect the same part of source program.

In OpenJava, to resolve this ambiguousness of several macro expansion, the system always invokes `translateDefinition()` first as callee-side translation, then it apply caller-side translation to source code of class declarations which was already applied callee-side translation. Metaprogrammers can design metaprogram considering this specified order of translation. In this rule, if `translateDefinition()` changes an instance creation expression of class X into Y's, `expandAllocation()` defined in the metaclass of X is not performed.

Moreover, the OpenJava system always performs `translateDefinition()` for superclasses first, i.e. the system performs it for subclasses after superclasses. As a class definition strongly depends on the definition of its superclass, the translation of a class often varies depending on the definition of its superclass. To settle the definition of superclasses, the system first translates the source program declaring superclasses. Additionally, there are some cases where the definition of a class D affects the result of translation of a class E. In OpenJava, from `translateDefinition()` for E, a metaprogrammer can explicitly specify that `translateDefinition()` for D must be performed before.

In the case there are dependency relationships of translation among several macro expansions, consistent order of translation is specified to address this ambiguousness of translation results.

Dealing with Separate Compilation

In Java, classes can be used in program only if they exist as source code or byte code (.class file). If there is no source code for a class C, the system cannot specify the metaclass of C, as is. Then, for instance, it cannot perform the appropriate `expandAllocation()` on instance creation expressions of C.

Therefore, OpenJava automatically preserves meta-level information such as the metaclass name for a class when it processes the callee-side translation of each class. These preservation are implemented by translating these information into a string held in a field of a special class, which is to be

compiled into byte code. The system uses this byte code to obtain necessary meta-level information in another process without source code of that class. Additionally, metaprogrammers can request the system to preserve customized meta-level information of a class.

Meta-level information can be preserved as special attributes of byte code. In OpenJava, such information is used only at compile-time but not at runtime. Thus, in order to save runtime overhead, we chose to preserve such information in separated byte code which is not to be loaded by JVM at runtime.

3.2.6 Syntax Extension

With OpenJava macros, a metaclass can introduce new class/member modifiers and clauses starting with the special word at some limited positions of the regular Java grammar. The newly introduced clauses are valid only in the parts related to instances of the metaclass.

In a class declaration (callee-side), the positions allowed to introduce new clauses are:

- before the block of member declarations,
- before the block of method body in each method declaration,
- after the field variable in each field declaration.

And in other class declarations (caller-side), the allowed position is:

- after the name of the class.

Thanks to the limited positions of new clauses, the system can parse source programs without conflicts of extended grammars. Thus, metaprogrammers do not have to care about conflicts between clauses.

```
class VectorStack instantiates AdapterClass
   adapts Vector in v to Stack
{
   ....
}
```

Figure 3.5: An example of syntax extension in OpenJava

Fig. 3.5 shows an example source program using a macro, a metaclass AdapterClass, supporting programming with the ADAPTER pattern [32]. The metaclass introduces a special clause beginning with `adapts` to make programmers to write special description for the ADAPTER pattern in the class declaration. The `adapts` clause in the Fig. reffig:VectorStack VectorStack is the adapter to a class Stack for a class Vector. The information by this clause

is used only when the class-objects representing VectorStack performs macro expansion. Thus, for other class-objects, semantical information added by the new clause is recognized as a regular Java source code.

```
static SyntaxRule getDeclSuffix(String keyword) {
   if (keyword.equals("adapts")) {
      return new CompositeRule(
         new TypeNameRule(),
         new PrepPhraseRule("in", new IdentifierRule()),
         new PrepPhraseRule("to", new TypeNameRule()) );
   }
   return null;
}
```

Figure 3.6: A meta-program for a customized suffix

To introduce this `adapts` clause, metaprogrammers implement a member method `getDeclSuffix()` in the metaclass AdapterClass as shown in Fig. 3.6. The member method `getDeclSuffix()` is invoked by the system when needed, and returns a SyntaxRule object representing the syntax grammar beginning with the given special word. An instance of the class SyntaxRule implements a recursive descendant parser of LL(k), and analyzes a given token series to generate an appropriate AST. The system uses SyntaxRule objects obtained by invoking `getDeclSuffix()` to complete the parsing.

For metaprogrammers of such SyntaxRule objects, OpenJava provides a class library of subclasses of SyntaxRule, such as parsers of regular Java syntax elements and synthesizing parser for tying, repeating or selecting other SyntaxRule objects. Metaprogrammers can define their desired clauses by using this library or by implementing a new subclass of SyntaxRule.

3.2.7 Metaclass Model of OpenJava

A class must be managed by a single metaclass in OpenJava. Though it would be useful if programmers could apply several metaclasses to a class, we did not implement such a feature because there is a problem of conflict of translation between metaclasses. And, a metaclass for a class A does not manage a subclass A' of A, that is, the metaclass of A does not perform the callee-side and caller-side translation of A' it is not specified to be the metaclass of A' in the source program declaring A'.

For innerclasses such as member classes, local classes, anonymous classes in the Java language, each of them are also an instance of a metaclass in OpenJava. Thus programmers may apply a desired metaclass to such classes.

3.3 Related Work

There are a number of systems using the class-object model for representing a logical structure of a program: 3-KRS [60], ObjVlisp [22], CLOS MOP [49], Smalltalk-80 [34], and so on. The reflection API [44] of the Java language also uses this model although the reflection API does not allow to change class-objects; it only allows to inspect them. Furthermore, the reflection API uses class metaobjects for making class definition accessible at runtime. On the other hand, OpenJava uses class-objects for macro expansion at compile-time.

OpenC++ [15] also uses the class-object model. OpenJava inherits several features, such as the type-driven translation mechanism, from OpenC++. However, the data structure mainly used in OpenC++ is still an AST (abstract syntax tree). MPC++ [43] and EPP [42] are similar to OpenC++ with respect to the data structure. As mentioned in section 3.1, an AST is not an appropriate abstraction for some macros frequently used in object-oriented programming.

3.4 Summary

This chapter described OpenJava, which is a macro system for Java providing a data structure called class-objects. A number of research activities have been done for enhancing expressive power of macro systems. This research is also in this stream. OpenJava is a macro system with a data structure representing a logical structure of an object-oriented program. This made it easier to describe typical macros for object-oriented programming which was difficult to describe with ordinary macro systems. To show the effectiveness of OpenJava, we implemented some macros in OpenJava for supporting programming with design patterns.

Chapter 4

Javassist

This chapter presents Javassist, which is a bytecode editor that we have developed for Java. Like OpenJava, Javassist employs the class-object model proposed in this thesis. In Javassist, we apply the design of the class-object to transformations of Java bytecode, which are compiled, binary representation of Java program. Since Javassist handles bytecode, transformations with Javassist are not limited at compile-time.

Java is a programming language supporting reflection. The reflective ability of Java is called the reflection API. However, it is almost restricted to introspection, which is the ability to introspect data structures used in a program such as a class. The Java's ability to alter program behavior is very limited; it only allows a program to instantiate a class, to get/set a field value, and to invoke a method through the API.

To address the limitations of the Java reflection API, several extensions have been proposed. Most of these extensions enable behavioral reflection, which is the ability to intercept an operation such as method invocation and alter the behavior of that operation. If an operation is intercepted, the runtime systems of those extensions call a method on a *metaobject* for notifying it of that event. The programmer can define their own version of metaobject so that the metaobject executes the intercepted operation with customized semantics, which implement a language extension for a specific application domain such as fault tolerance [30].

However, behavioral reflection only provides the ability to alter the behavior of operations in a program but not provides the ability to alter data structures used in the program, which are statically fixed at compile time (or, in languages like Lisp, when they are first defined). The latter ability called structural reflection allows a program to change, for example, the definition of a class, a function, and a record on demand. Some kinds of language extensions require this ability for implementation and thus they cannot be implemented with a straightforward program using behavioral

reflection; complex programming tricks are often needed.

To simply implement these language extensions, this chapter presents *Javassist*, which is a class library for enabling structural reflection in Java. Since portability is important in Java, we designed a new architecture for structural reflection, which can be implemented without modifying an existing runtime system or compiler. Javassist is a Java implementation of that architecture. An essential idea of this architecture is that structural reflection is performed by bytecode transformation at compile-time or load time. Javassist does not allow structural reflection after a compiled program is loaded into the JVM. Another feature of our architecture is that it provides source-level abstraction: the users of Javassist do not have to have a deep understanding of the Java bytecode. Our architecture can also execute structural reflection faster than the compile-time metaobject protocol used by OpenC++ [15] and OpenJava [82].

In the rest of this chapter, we first overview previous extensions enabling behavioral reflection in Java and point out limitations of those extensions. Then we present the design of Javassist in Section 3 and show typical applications of Javassist in Section 4. In Section 5, we compare our architecture with related work. Section 6 is conclusion.

4.1 Extensions to the Reflection Ability of Java

The Java reflection API dose not provide the full reflective capability. It does not enable alteration of program behavior but it only supports introspection, which is the ability to introspect data structures, for example, inspecting a class definition. This design decision was acceptable because implementing the full capability was difficult without a decline in runtime performance. An implementation technique using partial evaluation has been proposed [62, 11] but the feasibility of this technique in Java has not been clear.

However, several extensions to the Java reflection API have been proposed. To avoid performance degradation, most of these extensions enable restricted behavioral reflection. They only allow alteration of the behavior of specific kinds of operations such as method calls, field accesses, and object creation. The programmers can select some of those operations and alter their behavior. The compilers or the runtime systems of those extensions insert *hooks* in programs so that the execution of the selected operations is intercepted. If these operations are intercepted, the runtime system calls a method on an object (called *a metaobject*) associated with the operations or the target objects. The execution of the intercepted operation is implemented by that method. The programmers can define their own version of metaobject for implementing new behavior of the intercepted operations.

The runtime overheads due to this restricted behavioral reflection are low

since only the execution of the intercepted operations involves a performance penalty and the rest of the program runs without any overheads. Especially, if hooks for the interception are statically inserted in a program during compilation, the runtime overheads are even lowered. To statically insert hooks, Reflective Java [86] performs source-to-source translation before compilation and Kava [85] performs bytecode-level transformation when a program is loaded into the JVM. MetaXa [52, 35] internally performs bytecode-level transformation with a customized JVM. It uses a customized just-in-time compiler (JIT) for improving the execution speed of the inserted hooks. This hook-insertion technique is well known and has been applied to other languages such as C++ [19].

Although the restricted behavioral reflection is useful for implementing various language extensions, there are some kinds of extensions that cannot be intuitively implemented with that kind of reflection. An example of these extensions is binary code adaptation (BCA) [47], which is a mechanism for altering a class definition in binary form to conform changes of the definitions of other classes. Suppose that we write a program using a class library obtained from a third party. For example, our class Calendar implements an interface Writable included in that class library:

```
class Calendar implements Writable {
  public void write(PrintStream s) { ... }
}
```

The class Calendar implements method `write()` declared in the interface Writable.

Then, suppose that the third party gives us a new version of their class library, in which the interface Writable is renamed into Printable and it declares a new method `print()`. To make our program conform this new class library, we must edit the definitions of all our classes implementing Writable, including Calendar:

```
class Calendar implements Printable {
  public void write(PrintStream s) { ... }
  public void print() { write(System.out); }
}
```

The interface of Calendar is changed into Printable and method `print()` is added.

BCA automates this adaptation; it automatically alters class definitions in binary form according to a configuration file specifying how to alter them. Note that the method body of `print()` is identical among all the updated classes since `print()` can be implemented with the functionality already provided by `write()` for the old version. If that configuration file is supplied by the library developer, we can run our program without concern about evolution of the class library.

Unfortunately, implementing BCA with behavioral reflection is not intuitive or straightforward. Since behavioral reflection cannot directly provide the ability to alter data structures such as a class definition or construct a new data structure, these reflective computation must be indirectly implemented. For example, the implementation of BCA with behavioral reflection defines a metaobject indirectly performing the adaptation specified by a given configuration file. For the above example, this metaobject is made to be associated with Calendar and it watches method calls on Calendar objects. If the method `print()` is called, the metaobject intercepts that method call and executes the computation corresponding to `print()` instead of the Calendar object. The metaobject also intercepts runtime type checking so that the JVM recognizes Calendar as a subtype of Printable. Recall that Java is a statically typed language and the original Calendar is a subtype of Writable.

The ability to alter data structures used in a program is called structural reflection, which has not been directly supported by previous systems. Although a number of language extensions are more easily implemented with structural reflection than with behavioral reflection, the previous systems have not been addressing those extensions. They have been too much focused on language extensions that can be implemented by altering the behavior of method calls and so on.

4.2 Javassist

To simply implement language extensions like BCA shown in the previous section, we developed Javassist, which is our extension to the Java reflection API and enables structural reflection instead of behavioral one. Javassist is based on our new architecture for structural reflection, which can be implemented without modifying an existing runtime system or a compiler.

4.2.1 Implementations of Structural Reflection

Structural reflection is the ability to allow a program to alter the definitions of data structures such as classes and methods. It has been provided by several languages such as Smalltalk [34], ObjVlisp [22], and CLOS [49]. These languages implement structural reflection with support mechanisms embedded in runtime systems. Since the runtime systems contain internal data representing the definitions of data structures such as a class, the support mechanisms allow a program to directly read and change those internal data and thereby execute structural reflection on the correspondent data structures.

We could not accept this implementation technique for Javassist since it needs to modify a standard JVM but portability is important in Java. Furthermore, a naive application of this technique to Java would cause serious

performance degradation of the JVM because this technique makes it difficult for runtime systems to employ optimization techniques based on static information of executed programs. Since a program may be altered at runtime, efficient dynamic recompilation is required for redoing optimization on demand. For example, method inlining is difficult to perform. If an inlined method is altered at runtime with structural reflection, all the inlined code must be updated. To do this, the runtime system must record where the code is inlined. This will spend a large amount of memory space. Another example is the "v-table" technique used for typical C++ implementations [29]. This technique statically constructs method dispatch tables so that invoked methods are quickly selected with a constant offset in the tables. If a new method is added to a class at runtime, then the dispatch tables may be updated and all offsets in the tables may be recomputed. Since the dynamic recompilation technique has been used so far for gradually optimizing "hot spots" of compiled code at runtime [40], it has been assuming that a program is never changed at runtime. Effectiveness of dynamic recompilation without this assumption is an open question.

Another problem is correctness of types. Since Java is a statically typed language, a variable of type X must be bound to an object of X or a subclass Y of X. If a program can freely access and change the internal data of the JVM, it may dynamically change the super class of Y from X to another class. This change causes a type error for the binding between a variable of type X and an object of Y. To address this problem, extra runtime type checks or restrictions on the range of structural reflection are needed.

4.2.2 Load-time Structural Reflection

To avoid the problems mentioned above, we designed a new architecture for structural reflection; it does not need to modify an existing runtime system or a compiler. On the other hand, it enables structural reflection only before a program is loaded into a runtime system, that is, at load time. Javassist is a class library enabling structural reflection based on this architecture. In Java, the bytecode obtained by compilation of a program is stored in *class files*, each of which corresponds to a distinct class. Javassist performs structural reflection by translating alterations by structural reflection into equivalent bytecode transformation of the class files. After the transformation, the modified class files are loaded into the JVM and then no alterations are allowed after that. Thereby, Javassist can be used with a standard JVM, which may use various optimization techniques.

Javassist is used with a user class loader. Java allows programs to define their own versions of class loader, which fetch a class file from a not-standard resource such as a network. A typical definition of the class loader is as follows:

```
class MyLoader extends ClassLoader {
  public Class loadClass(String name) {
    byte[] bytecode = readClassFile(name);
    return resolveClass(defineClass(bytecode));
  }

  private byte[] readClassFile(String name) {
    // read a class file from a resource.
  }
}
```

The methods `defineClass()` and `resolveClass()` are inherited from ClassLoader. They request the JVM to load a class constructed from the bytecode given as an array of `byte`. The returned value is a Class object representing the loaded class. Once a class X is manually loaded with an instance of MyLoader, all classes referenced by that class X are loaded through that class loader. The JVM automatically calls `loadClass()` on that class loader for loading them on demand.

Javassist helps `readClassFile()` shown above obtain the bytecode of a requested class. It can be regarded as a class library for reading bytecode from a class file and altering it. However, unlike similar class libraries such as the BCEL [25] and JOIE [21], Javassist provides source-level abstraction so that it can be used without knowledge of bytecode or the data format of the class file. Also, Javassist was designed to make it difficult to wrongly produce a class file rejected by the bytecode verifier of the JVM.

4.2.3 The Javassist API

We below present the overview of the Javassist API.

Reification and Reflection

The first step of the use of Javassist is to create a CtClass (compile-time class) object representing the bytecode of a class loaded into the JVM. This step is for reifying the class to make it accessible from a program. If `stream` is an InputStream for reading a class file (from a local disk, memory, a network, etc.), then:

```
CtClass c = new CtClass(stream);
```

creates a new CtClass object representing the bytecode of the class read from the class file, which contains enough symbolic information to reify the class. Also, the constructor of CtClass can receive a String class name instead of an InputStream. If a String class name is given, Javassist searches a class path and finds an InputStream for reading a class file.

One can call various methods on the CtClass object for introspecting and altering the class definition. Changes of the class definition are reflected on

the bytecode represented by that object. To obtain the bytecode for loading the altered class into the JVM, method `toBytecode()` is called on that object:

```
byte[] bytecode = c.toBytecode();
```

Loading the obtained bytecode into the JVM is regarded as the step for reflecting the CtClass object on the base level. Javassist provides several other methods for this step. For example, method `compile()` writes bytecode to a given output stream such as a local file and a network. Method `load()` directly loads the class into the JVM with a class loader provided by Javassist. It returns a Class object representing the loaded class. Recall that Class is included in the Java reflection API while CtClass is in Javassist.

Note that Javassist does not provide any framework for specifying how and what classes are processed with Javassist. The programmer of the class loader has freedom with respect to this framework. For example, the class loader may process classes with Javassist only if they are specified by a configuration file read at the beginning. It may process them according to a *hard-coded* algorithm.

Javassist allows a user class loader to define a new class from scratch without reading any class file. This is useful if a program needs to dynamically define a new class on demand. To do this, a CtClass object must be created as follows:

```
CtClass c2 = new CtNewClass();
```

The created object `c2` represents an empty class that has no methods or fields although methods and fields can be added to the class later through the Javassist API shown below. If `toBytecode()` is called on this object, then it returns the bytecode corresponding to that empty class.

Introspection

Javassist provides several methods for introspecting the class represented by a CtClass object. This part of the Javassist API is compatible with the Java reflection API except that Javassist does not provide methods for creating an instance or invoking a method because these methods are meaningless at load time. Table 4.1 lists selected methods for introspection.

CtClass objects returned by `getSuperclass()` and `getInterfaces()` are constructed from class files found on a class path. They represent the original class definitions and thus accept only introspection but not alteration. To alter a class, another CtClass object must be explicitly created with the `new` operator. Modifications to this object have no effect on the CtClass object returned by `getSuperclass()` or `getInterfaces()`. For example, suppose

Table 4.1: Methods in CtClass for introspection

Method	Description
`String getName()`	gets the class name
`int getModifiers()`	gets the class modifiers such as `public`
`boolean isInterface()`	determines whether this object represents a class or an interface
`CtClass getSuperclass()`	gets the super class
`CtClass[] getInterfaces()`	gets the interfaces
`CtField[] getDeclaredFields()`	gets the fields declared in the class
`CtMethod[] getDeclaredConstructors()`	gets the constructors declared in the class
`CtMethod[] getDeclaredMethods()`	gets the methods declared in the class

Table 4.2: Methods in CtField and CtMethod for introspection

Method in CtField	Description
`String getName()`	gets the field name
`CtClass getDeclaringClass()`	get the class declaring the field
`int getModifiers()`	gets the field modifiers such as `public`
`CtClass getType()`	get the field type
Method in CtMethod	**Description**
`String getName()`	gets the method name
`CtClass getDeclaringClass()`	get the class declaring the method
`int getModifiers()`	gets the method modifiers such as `public`
`CtClass[] getParameterTypes()`	gets the types of the parameters
`CtClass[] getExceptionTypes()`	gets the types of the exceptions that the method may throw
`boolean isConstructor()`	returns `true` if the method is a constructor
`boolean isClassInitializer()`	returns `true` if the method is a class initializer

that a class C inherits from a class S. If a CtClass object for S is created with `new` and a method `m()` is added to that object, this modification is not reflected on the object returned by `getSuperclass()` on a CtClass object for C. The class C inherits `m()` from S only if the CtClass object created with `new` is converted into bytecode and loaded into the JVM.

The information about fields and methods is provided by objects separate from the CtClass object; it is provided by CtField objects obtained by `getDeclaredFields()` and CtMethod objects obtained by `getDeclaredMethods()`, respectively. The information about a constructor is also provided by a CtMethod object. Table 4.2 lists methods in CtField and CtMethod for introspection.

Table 4.3: Methods for alteration

Method in CtClass	Description
`void bePublic()`	make the class `public`
`void beAbstract()`	make the class `abstract`
`void notFinal()`	remove the `final` modifier from the class
`void setName(String name)`	change the class name
`void setSuperclass(CtClass c)`	change the super class
`void setInterfaces(CtClass[] i)`	change the interfaces
`void addConstructor(...)`	add a new constructor
`void addDefaultConstructor()`	add the default constructor
`void addAbstractMethod(...)`	add a new `abstract` method
`void addMethod(...)`	add a new method
`void addWrapper(...)`	add a new wrapped method
`void addField(...)`	add a new field
Method in CtField	**Description**
`void bePublic()`	make the field `public`
Method in CtMethod	**Description**
`void bePublic()`	make the method `public`
`void instrument(...)`	modify a method body
`void setBody(...)`	substitute a method body
`void setWrapper(...)`	substitute a method body

Alteration

A difference between Javassist and the standard Java reflection API is that Javassist provides methods for altering class definitions. Several methods for alteration are defined in CtClass (Table 4.3). These methods are categorized into methods for changing class modifiers, methods for changing class hierarchy, and methods for adding a new member. They were carefully selected to satisfy our design goals.

Our design goals are three. (1) The first goal is to provide source-level abstraction for programmers. Javassist was designed so that programmers can use it without knowledge of the Java bytecode. (2) The second goal is to execute structural reflection as efficiently as possible. (3) The last goal is to help programs perform structural reflection in a safe manner in terms of types.

As for the first goal, the most significant design decision was how programmers specify a method body. Suppose that a new method is added to a class. If a sequence of bytecode is used for specifying the body of that method, the programmers would get great flexibility but have to learn details of bytecode. To achieve the first goal, Javassist allows to copy a method body from another existing method although this design decision restricts the flexibility of the added method. The copied bytecode sequence is adjusted to fit the destination method. For example, the bytecode for

accessing a member through the `this` variable contains a symbolic reference to the type of `this`. This reference is replaced with one to the class declaring the destination method.

Despite the well-known quasi-equivalence between Java source code and bytecode, the correspondence between source-level and bytecode-level alterations are not straightforward. Hiding the gap between the two levels from programmers is also a part of the first goal.

For example, `setName()` renames a class but it also substitutes the new name for all occurrences of the old name in the definition of that class, including method signatures and bodies. Modifying a single constant-pool item never performs this substitution. If a constructor calls another constructor in the same class (if it executes `this()`), then the bytecode of the former constructor is modified since the bytecode contains a symbolic reference to the name of the class declaring the latter constructor. This reference must be modified to indicate the new name.

`setSuperclass()` performs similar substitution. If it is called, all occurrences of the old super class name is replaced with a new name and all constructors are modified so that they call a constructor in the new super class. However, there is an exception to this substitution. If the name of the original super class is java.lang.Object (the root of the class hierarchy), `setSuperclass()` does not perform the substitution except it modifies constructors. This is because java.lang.Object is often used for representing any class. For example, although `addElement()` in java.util.Vector takes a parameter of class java.lang.Object, which is the super class of java.util.Vector, this never means that `addElement()` takes an instance of the super class.

The second design goal is to reduce overheads due to class loading with Javassist. Since we will use Javassist for implementing a mobile-agent system, in which Javassist inserts security-check code into bytecode, Javassist must transform bytecode received through a network as efficiently as possible. Mobile agents frequently move among hosts and thus we cannot ignore the loading time of the bytecode implementing the mobile agents.

Our design decision on how programmers specify a method body was influenced by the second goal as well as the first one. Javassist does not use source code for specifying the body of an added method. If source code is used, it must be compiled *on the fly* when a class is loaded into the JVM. A naive implementation of this source-code approach would produce a complete class definition including the added method at source level and then compile it with a Java compiler such as `javac`. As we show later, however, this implementation implies serious performance penalties. To achieve practical efficiency, we need a special compiler that can quickly compile only a method body. We did not adopt the source-code approach because of limitations of our resources. Instead, Javassist allows to copy a pre-compiled

method body from a class to another. This approach does not imply overheads due to source-code compilation at load time.

The third design goal is to prevent programs to wrongly produce a class including type incorrectness. To achieve this goal, Javassist allows only limited kinds of alteration of class definitions. In general, reflective systems should impose some restrictions on structural reflection so that programs do not falsely collapse themselves with reflection. Suppose that a reflective system allows to remove a field from a class at runtime. If there are already instances of that class, is it appropriate that the system simply discards the value of the removed field of those instances?

Since erroneous bytecode produced with Javassist is rejected by the bytecode verifier, it can never damage the JVM. However, restricting the reflective capability of Javassist is still necessary because it is often awkward to correct a program producing erroneous bytecode. For this reason, Javassist does not provide methods for removing a method or a field from a class because they cause type incorrectness if there is a method accessing the removed method or field. Javassist also imposes restrictions on the class passed to `setSuperclass()`, which is a method for changing a super class. The new super class must be a subclass of the original super class since there may be methods that implicitly cast an instance of that class to the original super class. Of course, the new super class must not be `final`. Furthermore, Javassist does not provide a method for changing the parameters of a method. Programmers are recommended to add a new method with the same name but with different parameters.

Adding a New Member

Javassist provides methods for adding a new method to a class. To avoid the abstraction and performance problems mentioned above, `addMethod()` receives a CtMethod object, which specifies a method body. The signature of `addMethod()` is as shown below:

```
void addMethod(CtMethod m, String name, ClassMap map)
```

`name` specifies the name of the added method. The method body is copied from a given method `m`. Since a method body is copied from an existing compiled method, no source-code compilation is needed at load time or no raw bytecode is given to `addMethod()`. Programmers can describe a method body in Java and compile it in advance. Javassist reads the bytecode of the compiled method and adds it to another class. This improves execution performance of Javassist since a compiler is not run at load time.

When a method body is copied, some class names appearing in the body can be replaced according to a hash table `map`.[1] For example, programmers can declare a class XVector:

```
public class XVector extends java.util.Vector {
  public void add(X e) {
    super.addElement(e);
  }
}
```

and copy the method `add()` into a class StringVector:

```
CtMethod m = /* method add() in XVector */;
CtClass c = /* class StringVector */;
ClassMap map = new ClassMap();
map.put("X", "java.lang.String");
c.addMethod(m, "addString", map);
```

The class name java.lang.String is substituted for all occurrences of the class name X in `add()`. The added method is as follows:

```
public void addString(java.lang.String e) {
  super.addElement(e);
}
```

Javassist provides another method `addWrapper()` for adding a new method. It allows more generic description of a method body:

```
void addWrapper(int modifiers, CtClass returnType, String name,
                CtClass[] parameters, CtClass[] exceptions,
                CtMethod body, ConstParameter constParam)
```

The first five parameters specify the modifiers, the return type, the method name, the parameter types, and the exceptions that the method may throw. The body of the added method is copied from the method specified by `body`. No matter what the signature of the added method is, the method specified by `body` must have the following signature:

```
Object m(Object[] args, value-type constValue)
```

To fill the gap between this signature and the signature of the added method, `addWrapper()` implicitly wraps the copied method body in *glue* code, which constructs an array of actual parameters passed to the added method and assigns it to `args` before executing the copied method body. The glue code also sets `constValue` to a constant value specified by `constParam` passed to `addWrapper()`. In the current version of Javassist, an integer value or

[1] At least, `addMethod()` replaces all occurrences of the name of the class declaring the copied method. Even if that class name does not appear at source level, the corresponding bytecode may include references to it.

a String object can be specified for the constant value. For example, this constant value can be used to pass the name of the added method.

The value returned by the copied method body is an Object object. The glue code also converts it into a value of the type specified by `returnType`. Then it returns the converted value to the caller to the added method. If type conversion fails, then an exception is thrown. Although methods added by `addWrapper()` involve runtime overheads due to type conversion, a single method body can be used as a template of multiple methods receiving a different number of parameters. Examples of the use of `addWrapper()` are shown in Section 4.3.

Javassist also provides a method for adding a new field to a class:

```
void addField(int modifiers, CtClass type, String fieldname,
              String accessor, FieldInitializer init)
```

If `accessor` is not `null`, this method also adds an accessor method, which returns the value of the added field. The name of the accessor is specified by `accessor`. Moreover, the last parameter `init` specifies the initial value of the added field. The initial value is either one of parameters passed to a constructor, a newly created object, or the result of a call to a static method.

Altering a Method Body

Although Javassist does not allow to remove a method from a class, it provides methods for changing a method body. `setBody()` and `setWrapper()` in CtMethod substitute a given method body for an original body:

```
void setBody(CtMethod m, ClassMap map)
void setWrapper(CtMethod m, ConstParameter param)
```

They correspond to `addMethod()` and `addWrapper()` respectively. `setBody()` copies a method body from a given method `m`. Some class names appearing in the body are replaced with different names according to `map`. `setWrapper()` also copies a method body from `m` but it wraps the copied body in glue code. The signature of `m` must be:

```
Object m(Object[] args, value-type constValue)
```

Javassist also provides a method for modifying expressions in a method body. `instrument()` in CtMethod performs this modification:

```
void instrument(CodeConverter converter)
```

The parameter `converter` specifies how to instrument a method body. The CodeConverter object can perform various kinds of instrumentation. Table 4.4 lists methods provided by the current implementation of Javassist.

Table 4.4: Methods in CodeConverter

Method	Description
`void redirectFieldAccess()`	change a field-access expression to access a different field.
`void replaceNew()`	replace a `new` expression with a `static` method call.
`void replaceFieldRead()`	replace a field-read expression with a `static` method call.
`void replaceFieldWrite()`	replace a field-write expression with a `static` method call.

They direct a CodeConverter object to replace a specific kind of expressions with *hooks*, which invoke static methods for executing the expressions in a customized manner. The idea of CodeConverter came from C++'s operator overloading. CodeConverter was designed for safely altering the behavior of operators such as `new` and `.` (dot) independently of the context.

For example, expressions for instantiating a specific class can be replaced with expressions for calling a static method. Suppose that variables `xclass` and `yclass` represent class X and Y, respectively. Then a program:

```
CtMethod m = ... ;
CodeConverter conv = new CodeConverter();
conv.replaceNew(xclass, yclass, "create");
m.instrument(conv);
```

instruments the body of the method represented by the CtMethod object `m`. All expressions for instantiating the class X such as:

```
new X(3, 4);
```

are translated into expressions for calling a static method `create()` declared in the class Y:

```
Y.create(3, 4);
```

The parameters to the `new` expression are passed to the static method.

Reflective Class Loader

The class loader provided by Javassist allows a loaded program to control the class loading by that class loader. If a program is loaded by Javassist's class loader L and it includes a class C, then it can intercept the loading of C by L to self-reflectively modify the bytecode of C. For avoiding infinite recursion, while the loading of a class is intercepted, further interception is prohibited. The `load()` method in CtClass requires that a program is

loaded by Javassist's class loader although the other methods work without Javassist's class loader.

Java's standard class loader never allows this self-reflective class loading for security reasons. If it is allowed, a program may change some `private` fields to `public` ones at load time for reading hidden values. Furthermore, in Java, if a program creates a class loader and loads a class C with that class loader, the loaded class is regarded as a different one from the class denoted by the name C appearing in that program. The latter class is loaded by the class loader that loaded the program.

Using Javassist without a Class Loader

Javassist can be used without a user class loader. There are three kinds of usage of Javassist: with a user class loader, with a web server, and off line.

For security reasons, an applet is usually prohibited from using a user class loader. However, we can write an applet working with Javassist if we use a web server as a replacement of a user class loader. Since classes used in an applet are loaded from a web server into the JVM of a web browser, we can customize the web server so that it runs Javassist for processing the classes before sending them to the web browser. Javassist includes a simple web server written in Java as a basis for such customization. We can extend it to perform structural reflection with Javassist. The program of the customized web server would be as follows:

```
for (;;) {
  receive an http request from a web browser.
  CtClass c = new CtClass(the requested class);
  do structural reflection on c if needed.
  byte[] bytecode = c.toBytecode();
  send the bytecode to the web browser.
}
```

Before sending a requested class to a web browser, it performs structural reflection on the class according to the algorithm, for example, given as a configuration file.

Another usage of Javassist is "off line". We can perform structural reflection on a class and overwrite the original class file of that class with the bytecode obtained as the result. The altered class can be later loaded into the JVM without a user class loader. The following is an example of the off-line use of Javassist:

```
CtClass c = new CtClass("Rectangle");
do structural reflection on c if needed.
c.compile();    // writes bytecode on the original class file.
```

This program performs structural reflection on class Rectangle and overwrites the class file of that class with the bytecode obtained by `c.toBytecode()`.

4.3 Examples

This section shows three applications of Javassist. We illustrate that Javassist can be used to implement non-trivial alteration required by these applications despite the level of the abstraction.

4.3.1 Binary Code Adaptation

The mechanism of binary code adaptation (BCA) [47] automatically alters class definitions according to a file written by the users, called a delta file:

```
delta class implements Writable {
  rename Writable Printable;
  add public void print() { write(System.out); }
}
```

This delta file specifies adaptation that we mentioned in Section 4.1.

If Javassist is used, the implementor of BCA has only to write a parser of delta file and a user class loader performing adaptation with Javassist. For example, the parser translates the delta file shown above into the Java program shown below:

```
class Exemplar implements Printable {
  public void write(PrintStream s) { /* dummy */ }
  public void print() { write(System.out); }
}

class Adaptor {
  public void adapt(CtClass c) {
    CtMethod printM = /* method print() in Exemplar */;
    CtClass[] interfaces = c.getInterfaces();
    for (int i = 0; i < interfaces.length; ++i)
      if (interfaces[i].getName().equals("Writable")) {
        interfaces[i] = CtClass.forName("Printable");
        c.setInterfaces(interfaces);
        c.addMethod(printM, new ClassMap());
        return;
      }
  }
}
```

The class Exemplar is compiled together with Adapter in advance so that `adapt()` can obtain a `CtMethod` object representing `print()`. `adapt()` uses the reification and introspection API of Javassist for obtaining it. It first constructs a `CtClass` object representing Exemplar and then obtains the `CtMethod` object by `getDeclaredMethods()` in CtClass. The class file for Exemplar is automatically found by Javassist on the class path used for loading Adapter.

The user class loader calls `adapt()` in Adaptor whenever a class is loaded into the JVM. It creates a CtClass object representing the loaded class and calls `adapt()` with that object. The method `adapt()` performs adaptation

if the loaded class implements Writable. Then the user class loader converts the CtClass object into bytecode and loads into the JVM.

Note that this implementation is more intuitive than the implementation with behavioral reflection. Moreover, it is simpler than the implementation without reflection since the implementor does not have to care about low-level bytecode transformation. If the users of BCA can directly write the classes Exemplar and Adaptor instead of a delta file, then the implementation would be much simpler since we do not need the parser of delta file.

4.3.2 Behavioral Reflection

Behavioral reflection enabled by MetaXa [52, 35] and Kava [85] can be implemented with an approximately 750-line program (including comments) using Javassist. A key idea of their implementations is to insert *hooks* in a program when a class is loaded into the JVM. We below see an overview of a user class loader performing this insertion with Javassist.

Let a metaobject be an instance of MyMetaobject, which is a subclass of Metaobject:

p

```
public class MyMetaobject extends Metaobject {
  public Object trapMethodcall(String methodName, Object[] args) {
    /* called if a method call is intercepted. */ }
  public Object trapFieldRead(String fieldName) {
    /* called if the value of a field is read. */ }
  public void trapFieldWrite(String fieldName, Object value) {
    /* called if a field is set. */ }
}
```

If field accesses and method calls on an instance of C:

```
public class C {
  public int m(int x) { return x + f; }
  public int f;
}
```

are intercepted by the metaobject, then the user class loader alters the definition of the class C into the following:[2]

```
public class C implements Metalevel {
  public int m(int x) { /* notify a metaobject */ }
  public int f;
  private Metaobject _metaobject = new MyMetaobject(this);
  public Metaobject _getMetaobject() { return _metaobject; }
  public int orig_m(int x) { return x + f; }
  public static int read_f(Object target) {
      /* notify a metaobject */ }
  public static void write_f(Object target, int value) {
      /* notify a metaobject */ }
}
```

[2]For simplicity, this implementation ignores `static` members although extending the implementation for handling `static` members is possible within the ability of Javassist.

```
class Exemplar {
  private Metaobject _metaobject;

  public Object trap(Object[] args, String methodName) {
    return _metaobject.trapMethodcall(methodName, args);
  }

  public static Object trapRead(Object[] args, String name) {
    Metalevel target = (Metalevel)args[0];
    return target._getMetaobject().trapFieldRead(name);
  }

  public static Object trapWrite(Object[] args, String name) {
    Metalevel target = (Metalevel)args[0];
    Object value = args[1];
    target._getMetaobject().trapFieldWrite(name, value);
  }
}
```

Figure 4.1: Class Exemplar

where the interface Metalevel declares the method `_getMetaobject()`.

This alteration can be performed within the ability of Javassist. The interface Metalevel is added by `setInterfaces()` in CtClass. The field `_metaobject` and the accessor `_getMetaobject()` are added by `addField()` in CtClass.

For intercepting method calls, the user class loader first makes a copy of every method in C by calling `addMethod()` in CtClass. For example, it adds `orig_m()`[3] as a copy of `m()`. Then it replaces the body of every method in C with a copy of the body of the method `trap()` in Exemplar (see Figure 4.1). This modification is performed by `setWrapper()` in CtMethod. The gap between the signatures of `m()` and `trap()` is filled by `setWrapper()`. The substituted method body notifies a metaobject of interception. The first parameter `args` is a list of actual parameters and the second one `name` is the name of the copy of the original method such as `"orig_m"`. These two parameters are used for the metaobject to invoke the original method through the Java reflection API.

For intercepting field accesses, the user class loader instruments the bodies of methods in all classes. All accesses to a field `f` in C are translated into calls to a `static` method `read_f()` or `write_f()`. This instrumentation is performed by `instrument()` in CtMethod and `replaceFieldRead()` and `replaceFieldWrite()` in CodeConverter. The methods `read_f()` and `write_f()` notify a metaobject of the accesses. They are added by `addWrapper()` in CtClass as copies of `trapRead()` and `trapWrite()` in Exemplar. The gap between the signatures of `read_f()` (or `write_f()`) and

[3] If a method name is overloaded, a copy of each method must be given a different name such as `orig_m1()`, `orig_m2()`, ...

`trapRead()` (or `trapWrite()`) is filled by `addWrapper()`. For example, actual parameters to `read_f()` are converted into the first parameter `args` to `trapRead()`. The second parameter `name` to `trapRead()` is the name of the accessed field such as `"f"`.

4.3.3 Remote Method Invocation

Generating stub code for remote method invocation is another application of Javassist. A Java program cannot directly call a method on a remote object on a different computer. It needs the Java RMI tools generating stub code, which translates a method call into lower-level network data transfer such as TCP/IP communication. However, the Java RMI tools are compile-time ones; a program must be processed by the RMI compiler, which generates and saves stub code on a local disk. Also, a program using the Java RMI must be subject to a protocol (i.e. API) specified by the Java RMI.

Javassist allows programmers to develop their own version of the RMI tools, which specify a customized protocol and produce stub code at either compile-time or even runtime. Suppose that an applet needs to call a method on a Counter object on a web server written in Java. For remote method invocation, the applet needs stub code defining a proxy object of the Counter object, which has the same set of methods as the Counter object. If the Counter object has a method `setCount()`, the proxy object also has a method `setCount()` with the same signature. However, the method on the proxy object serializes given parameters and sends them to the web server, where `setCount()` is invoked on the Counter object with the received parameters.

This stub code can be generated at runtime with Javassist at the server side and it can be sent on demand to the applet side. The applet programmer can easily write the applet without concern about low-level network programming. The stub code for accessing the Counter object is as follows:

```
public class ProxyCounter {
  private RmiStream rmi;
  public ProxyCounter(int objectRef) {
    rmi = new RmiStream(objectRef);
  }
  public int setCount(int value) { /* remote method invocation */ }
}
```

An instance of ProxyCounter is a proxy object. An RmiStream object handles low-level network communication. The class RmiStream is provided by a runtime support library.

ProxyCounter can be defined within the confines of Javassist. The field `rmi` is added by `addField()` in CtClass and the initialization of `rmi` in a constructor can be specified by a FieldInitializer object passed to `addField()`.

The method `setCount()` is added by `addWrapper()` in CtClass as a copy of the method `invoke()` in Exemplar shown below:

```
class Exemplar {
  private RmiStream rmi;
  Object invoke(Object[] args, String methodName) {
    return rmi.rpc(methodName, args);
  }
}
```

The gap between the signatures of `setCount()` and `invoke()` is filled by `addWrapper()`. If `setCount()` is called, the actual parameter `value` is converted into an array of Object and assigned to `args`. `methodName` is set to a method name `"setCount"`[4]. Then `rpc()` is called on the RmiStream object for serializing the given parameters and sends them to the web server. Note that the parameters can be serialized within the ability of the standard Java if they are converted into an array of Object.

Stub code generation is another example, which is not straightforward to implement with behavioral reflection. In a typical implementation with behavioral reflection, a proxy object is an instance of the class Counter although all method calls on the proxy object are intercepted by a metaobject and forwarded to a remote object; the class ProxyCounter is not produced. Therefore, if the proxy object is created, a constructor declared in Counter is called and may cause fatal side-effects since the class Counter is defined as a class at the server side but the proxy object is not at that side.

4.4 Related Work

Reflection in Java

MetaXa [52, 35] and Kava [85] enable behavioral reflection in Java whereas Javassist enables structural reflection. They are suitable for implementing different kinds of language extensions. However, Javassist indirectly covers applications of MetaXa and Kava since a class loader providing functionality equivalent to MetaXa and Kava can be implemented with Javassist as we showed in Section 4.3.2.

Although Kava performs bytecode transformation of class files before the JVM loads them as Javassist does, they only insert hooks for interception in bytecode but do not run metaobjects at that time. They enable reflection at runtime and their ability is not structural reflection but the restricted behavioral reflection.

The Java reflection API was recently extended in the JDK 1.3 beta to partially enable behavioral reflection [78]. The new API allows a program

[4] If a method name is overloaded, this should be `setCount1`, `setCount2`, ... for distinction.

to dynamically define a proxy class implementing given interfaces. An instance of this proxy class delegates all method invocations to another object through a type-independent interface.

Javassist is not the first system enabling structural reflection in Java. For example, Kirby et. al. proposed a system enabling structural reflection (they called it linguistic reflection) in Java although their system only allows to dynamically define a new class but not to alter a given class definition at load time [51]. With their system, a Java program can produce a source file of a new class, compile it with an external compiler such as `javac`, and load the compiled class with a user class loader. They reported that their system could be used for defining a class optimized for a given runtime condition.

Compile-time MetaObject Protocol

The compile-time metaobject protocol [15] is another architecture enabling structural reflection without modifying an existing runtime system. OpenJava [82] is a Java implementation of this architecture. As Javassist does, it restricts structural reflection within the time before a class is loaded into the JVM although it was designed mainly for off-line use at compile time. However, OpenJava is source-code basis although Javassist is bytecode basis; OpenJava reads source code for creating an object representing a class, a method, or a field. Alteration to the object is translated into corresponding transformation of the source code. The bytecode for the altered class is obtained by compiling the modified source code. Since OpenJava is source-code basis, it can deal with syntax extensions within a framework of structural reflection. For example, one can extend the syntax of class declaration and make it possible to add an annotation to a class declaration.

On the other hand, the source-code basis means that OpenJava needs the source file of every processed class whereas Javassist needs only a class file (compiled binary). This is a disadvantage because source files are not always available if the class is provided by a third party. OpenJava also involves a performance overhead due to handling source code; the source file of every class must be parsed for reification and compiled for reflection. Although this overhead is compensation for the capability for fine-grained transformation of source code (including syntax extension), it is not negligible if OpenJava is used by a class loader for altering a loaded class. Some kinds of applications such as a mobile agent system do not need fine-grained transformation but fast class loading.

Although the implementations of OpenJava or Javassist have not been tuned up, the performance difference between OpenJava and Javassist is notable with respect to reification and reflection. If a class loader can be implemented with either OpenJava or Javassist, Javassist achieves shorter loading time. To show this performance difference, we compared Javassist

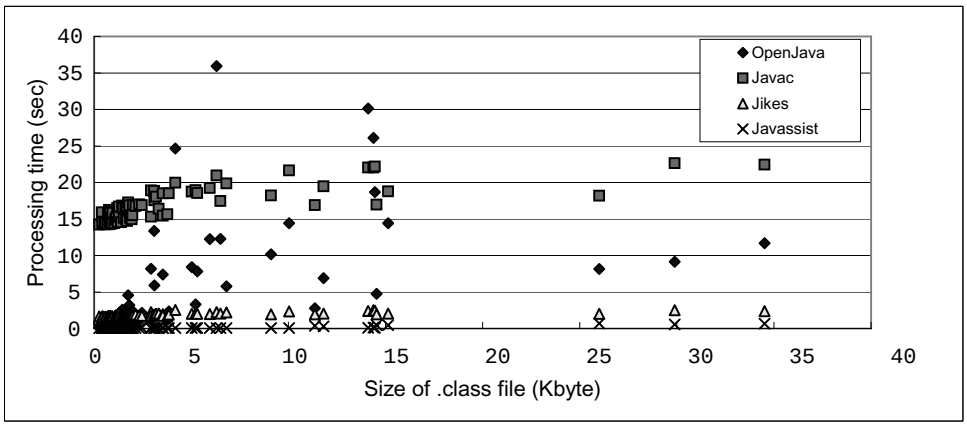

Figure 4.2: Execution time of reification and reflection

and OpenJava for classes in SPECjvm98 which is supplied with source text. We measured the time needed for reifying a given class so that it can be altered.

Figure 4.2 lists the results. The execution time is the minimum of three trial. For each trial, we repeated reification continuously and we mesured the second repetition. Since a program is gradually loaded into the JVM during the first repetition, the first one is tremendously slow. For compiling a modified source file, OpenJava requires to run a Java compiler. Thus the actual time for reification in OpenJava takes the compiling time of a Java compiler like Jikes compiler or the standard Javac compiler in addition to the processing time of the OpenJava parser.

Javassist processed a class more than ten times faster than OpenJava. Note that the execution time by Javassist is shorter than the time needed only for compiling a modified source file. This is because Javassist can move compilation penalties to an earlier stage. Even a method body is not compiled while Javassist is running; it is pre-compiled in advance and the resulting bytecode is directly copied to a target class at run time.

Bytecode Translators

Bytecode translators such as JOIE [21] and the BCEL [25] provide a functionality similar to Javassist. They enable a Java program to alter a class definition at load time. However, they are toolkits for directly dealing with bytecode, that is, the raw data structure of a class file. For example, classes included in JOIE are ClassInfo, Code, and Instruction. They show that JOIE was designed for experienced programmers who have a deep understanding of the Java bytecode and want to implement complex transformation. On

the other hand, Javassist was designed to be easy to use; it does not require programmers to have knowledge of the Java bytecode but instead it provides source-level abstraction for manipulating bytecode in a relatively safe manner. Although a range of instrumentation of a method body is restricted, we showed that Javassist can be used to implement non-trivial applications. Javassist can be regarded as a front end for easily and safely using a bytecode translator like JOIE; it is not a replacement of the bytecode translators.

Using bytecode instrumentation for implementing a reflective facility is a known technique in Smalltalk [10]. A uniqueness of Javassist against this is the design of the API providing source-level abstraction. The Javassist API was carefully designed to avoid wrongly producing a class definition containing type incorrectness.

X-time MOPs

OpenJIT [66] is a just-in-time compiler that allows a Java program to control how bytecode are compiled into native code. It provides better flexibility than Javassist with respect to instrumenting a method body while OpenJIT does not allow to add a new method or field to a class. However, using OpenJIT is more difficult than using Javassist because OpenJIT requires programmers to have knowledge of both the Java bytecode and native code. Although OpenJIT can be used without knowledge of the Java bytecode if programmers use a mechanism of OpenJIT for translating bytecode into a parse tree of an equivalent Java program, overheads due to that translation has not been reported.

The idea of enabling reflection only at load time for avoiding performance problems is found in the CLOS MOP [49]. For example, the CLOS MOP allows a program to alter the algorithm of determining the super classes of a given class but the super classes are statically determined when the class is loaded; the program cannot dynamically change the super classes at runtime.

4.5 Summary

This chapter presented Javassist, which is an extension to the Java reflection API. Unlike other extensions, it enables structural reflection in Java; it allows a program to alter a given class definition and to dynamically define a new class. A number of language extensions are more easily implemented with structural reflection than with behavioral reflection.

For avoiding portability and performance problems, the design of Javassist is based on our new architecture for structural reflection. Javassist performs structural reflection by instrumenting bytecode of a loaded class.

Therefore, it can be used with a standard JVM and compiler although structural reflection is allowed only before a class is loaded into the JVM, that is, at load time. Since a standard JVM is used, the classes processed by Javassist are subject to the bytecode verifier and the SecurityManager of Java. Javassist never breaks security guarantees given by Java.

The followings are important features of Javassist:

- Javassist is portable. It is implemented in only Java without native methods and it runs with a standard JVM. It does not need a platform-dependent class library. Portability is significant in Java programming.

- Javassist provides source-level abstraction for manipulating bytecode in a safe manner while bytecode translators, such as JOIE [21] and the BCEL [25], provide no higher-level abstraction. The users of Javassist do not have to have a deep understanding of the Java bytecode or to be careful for avoiding wrongly making an invalid class rejected by the bytecode verifier.

- Javassist never needs source code whereas OpenJava [82], which is another system for structural reflection with source-level abstraction, does. Since OpenJava performs structural reflection by transforming source code, it must parse and compile source code for reifying and reflecting a class. Thus a class loader using Javassist can load a class faster than one using OpenJava. However, OpenJava enables fine-grained manipulation of class definitions so that the resulting definitions may be smaller and more efficient than ones by Javassist.

The architecture that we designed for Javassist can be applied to other object-oriented languages if a compiled binary program includes enough symbolic information to construct a class object. However, the API must be individually designed for each language so that it allows a program to alter class definitions in a safe manner with respect to the semantics of that language.

Chapter 5

Addistant

This chapter proposes a system named Addistant, which enables the distributed execution of "legacy" Java bytecode. Here "legacy" means the software originally developed to be executed on a single Java virtual machine (JVM). For adapting legacy software to distributed execution on multiple JVM, developers using Addistant have only to specify the host where instances of each class are allocated and how remote references are implemented. According to that specification, Addistant automatically transforms the bytecode at load time. A technical contribution by Addistant is that it covers a number of issues for implementing distributed execution in the real world. In fact, Addistant can adapt a legacy program written with the Swing library so that Swing objects are executed on a local JVM while the rest of objects are on a remote JVM.

Object-oriented distributed software can be developed with various programming tools and environments. For example, a number of object request brokers have been proposed[37, 38, 39, 76], just to mention a few, and they allow programmers to easily make an object accessed by a remote host through a network. The programmers only have to define the interface of the object in an interface definition language. Another example is to use a distributed programming language like Emerald[7]. Such a language provides language constructs for creating objects on remote hosts, migrating them to another host, and so on.

However, these programming tools and environments are mainly for developing new distributed software from scratch; they are not for adapting "legacy" software to distributed execution on multiple hosts. Here, "legacy" means that the software was originally developed with intent to be executed on a single host. The existing tools or environments are not helpful in modifying the legacy software so that part of the software can be executed on a remote host. The programmers have to manually modify the source text of the program to follow a programming conventions specified by the tools, or

to use special language constructs. This modification takes long time and it is error-prone. It is even impossible if the program text is not available or modifiable. Practical demands for adapting legacy software to distributed execution will never disappear. While there is already a number of such legacy software, programmers will continue to develop legacy software since non-distributed software is easier to develop than distributed software.

To support distributed execution of legacy software written in Java[36], we have developed a system named *Addistant*. Addistant helps developers modify legacy Java programs to run on multiple Java virtual machines (JVM). It performs:

- Letting developers specify where to allocate the instances of each class among multiple hosts, in a policy file separated from the original program. All the instances of a class must be subject to the same allocation policy. Since real software contains a large number of objects, it is not realistic to individually specify where each object is allocated.

- Translating the bytecode of the legacy Java software according to the specification above so that specified classes are executed on the JVM running on a remote host. Addistant does not need source code for the translation. The translated bytecode is the regular Java bytecode. No custom JVM is needed for execution. And,

- Delivering the translated bytecode to remote JVM. This delivery is also performed by the runtime system of Addistant.

The translation by Addistant has been implemented by a synthesis and re-engineering of ideas found in existing programming tools and environments for distributed software. It is based on the proxy-master model, in which a proxy object forwards method invocations to a remote object through a network although the generation of the classes for proxy objects automatically managed by Addistant; it is hidden from the developers. A technical contribution by Addistant is rather that it covers all the issues that we encounter if applying the proxy-master model to real software development in Java. For example, since the JVM does not allow modifying the bytecode of the system classes at load time, the proxy-master model cannot be implemented with only a well-known straightforward translation, which requires the bytecode translation of all related classes including the system classes. To avoid these problems, Addistant provides multiple implementation approaches, which developers can choose for each class.

A typical application of Addistant is to apply functional distribution to a legacy Java program so that some modules of that program are executed on a remote host suitable for the functionality of those modules. For example, Addistant can be used to adapt a legacy program using the Swing class

library[77], which is Java's graphical user interface (GUI) library, so that GUI objects are executed on a host in front of the user while other objects are on a remote high-performance host. The resulting program produced by Addistant achieves good performance. Although the same effects can be achieved by using the X Window System[70], which enables the program to show windows on a remote display, our experiments showed that Addistant could achieve better response time of the GUI than the X Window System. This is because the X Window System implements distribution at the level of runtime library and thus it needs network communication for every drawing primitive. On the other hand, Addistant implements distribution by translating a whole program including both library code and user code. This higher-level distribution significantly reduces the amount of network communication. This fact suggests that a distributed program developed with a program translator can give better performance than one with a runtime library.

In the rest of this chapter, Section 2 presents the architecture of Addistant. Section 3 describes Java-related implementation issues. In Section 4, we show how Addistant can be used for adapting legacy software using the Swing class library to distributed execution. Section 5 discusses related work. Finally, section 6 concludes the chapter.

5.1 Addistant

Addistant is a Java programming tool for adapting legacy software, which was developed with intent to be executed on a single JVM, to distributed execution so that some objects of that software are executed on a remote host. This adaptation is performed by a bytecode translator at load time. This section first mentions design issues of the tools like Addistant, and then presents how Addistant deals with them.

5.1.1 Design Goal

Unlike developing distributed software from scratch, adapting legacy software written in Java to distributed execution needs special tool support. Without such tool support, programmers would have to read the program of that software and modify it so that some objects should be allocated on a remote host and method invocations be specially treated as if they are across a network. Since manual modification is troublesome and error-prone, a programming tool should automate this modification.

Although a number of researchers have been proposing Java-based distributed languages[46, 65, 67], those languages are not suitable for this purpose. Using such a distributed language, the programmer needs to obtain the source code of the program, which is usually unavailable if supplied by a

third party. Moreover, she has to modify the program to use special syntax provided by that language. For example, in case of a language proposed by Nagaratnam[65], a regular Java statement for creating an object:

```
Frame f = new Frame("The Great Encyclopedia");
```

must be replaced with a statement:

```
Frame f = remotenew Frame("The Great Encyclopedia");
```

using special syntax `remotenew`. She has to obtain the source code and edit all such statements.

Existing object request brokers (ORB)[14, 37, 38, 39, 76] are not suitable as well. They are mainly for making legacy software as a component of larger distributed software. To use such an ORB for distributing some modules of that software to a remote host, a programmer has to manually split the software into several modules and modify the program so that interactions among the modules are subject to the protocols of the ORB. For example, the Java RMI requires that all remote method invocations be performed through interface types. Suppose a method `show()` is called on a remote instance `f` of a class Frame. First, the programmer must declare a new interface DistributedFrame and modify the declaration of the class Frame so that the class Frame implements the interface DistributedFrame. Then she has to substitute DistributedFrame for occurrences of the class name Frame in the program. Also, she has to care about a number of issues such as remote object creation and polymorphism.

An ideal tool for adapting legacy software to distributed execution must provide the following features:

- **Remote reference:** The tool must hide implementation details of remote-object references from the programmers. The programmers should not have to modify the program so that remote references in the program follow a particular protocol specified by the tool.
- **Policy of object allocation:** The tool must allow the programmers to easily specify whether each object is allocated on a local host or a remote host. Since the programmers may not know details of the program of legacy software, the object allocation should be specified at an appropriate abstract level.
- **Program delivery:** The tool must be able to automatically deliver the program of modules to a remote host if those modules are executed on that host.

In the rest of this section, we first focus on the implementation of the remote reference. Then we describe how the users of Addistant specify the policy of object allocation. We also describe how Addistant implements the program delivery.

5.1.2 Remote Reference

Addistant implements remote references by bytecode translation at load time. To run the translated software, no custom JVM is needed; Addistant only needs that the regular JVM is running on every host.

Addistant employs the proxy-master model, which is also known as the Remote Proxy pattern[32, 68], so that a remote method can be transparently invoked with the same syntax as a local method. In this model, an object whose methods can be invoked from a remote host is associated with an object called proxy existing on that remote host. For distinction, we call the former object master. A proxy provides the same set of methods as its master and delegates every method invocation to its master. It encapsulates details of network communication necessary for the remote method invocations.

Unfortunately, any single implementation approach of the proxy-master model cannot deal with all kinds of classes. Each approach covers only the classes satisfying the criteria peculiar to that approach. Since we design a programming tool for legacy software, which someone else may have written, we cannot choose a single approach and enforce the criteria on the whole program. For example, one of the approaches needs to modify the declaration of the class of master objects. Since the JVM does not accept modified system classes, if an instance of a system class is a remote object, that approach cannot be used. A different approach must be used for that case.

To avoid this problem, Addistant provides several different approaches for implementing the proxy-master model. It currently provides four approaches: *replace*, *rename*, *subclass*, and *copy*. The developers can choose one from the four for each class of master. The differences among the four approaches are mainly how a proxy class is declared, how caller-side code, that is, expressions of remote method invocations, is modified, and how a master class is modified. The four approaches cover most of cases in practical development according to our experiences with the Swing library. To choose one from these four approaches, the developers must know whether a given class of master meets some of the following features or not:

- **Call by reference:** The master object must be passed to a remote method as a parameter in the call-by-reference manner. It cannot be passed in the call-by-value manner.

- **Heterogeneity:** A variable with that class type must be able to hold both local and remote references. Some kinds of master objects do not require this feature. For example, all the instances of a GUI class would exist on the same host in front of the user. If so, all the references to those instances are local on that host while they are remote on the other host. In this case, local and remote references do not coexist on a single host.
- **Unmodifiable bytecode:** The implementation must be done without transforming the unmodifiable bytecode. This is required as JVM prohibits the developers from modifying or replacing the bytecode of the system classes such as java.util.Vector. This feature is divided into three sub-features: the class declaration of the master objects (*original class*) is unmodifiable, other master classes accessing the master objects (*referrer classes*) are, or other master classes creating the master objects (*factory classes*) are, respectively.

The remainder of this subsection presents details of the four approaches, and conditions in which the approaches can be used. The summary of the conditions is listed in Table 5.1.

Table 5.1: Applicability of the four approaches. The mark of x ([x]) indicates that the approach is (probably) unavailable in case the feature is required.

		Replace	Rename	Subclass	Copy
Call by reference					x
Heterogeneity		x	x		
Unmodifiable bytecode of	original class	x		[x]	[x]
	referrer classes		x		
	factory classes		x	x	

Replace Approach.

The first approach is the *replace* approach. It is available unless the heterogeneity feature is required or the original class is unmodifiable. Developers should apply this approach to non-system classes whose masters are allocated at the same host.

Suppose that the class of a master object is Widget. Since the heterogeneity feature is not required, all the references to the Widget objects are either local or remote. Therefore, Addistant uses the original Widget class for local references on one host while it generates another version of the Widget class for remote references on the other hosts (Table 5.2). This version corresponds to a proxy class for the original Widget class. Addistant sends the bytecode of this proxy class to the host where there are remote references to Widget masters.

Table 5.2: The feature of a proxy class for a class Widget used by each approach. Here, we assume that the original Widget is a subclass of Object.

	Replace	Rename	Subclass
Proxy class	Widget†	WidgetProxy	WidgetProxy
Superclass of proxy	Object	Object	Widget
Variable type for proxy	Widget†	WidgetProxy	Widget

†A different version of the Widget class.

Rename Approach.

The second approach is the *rename* approach. The replace approach is not available if the declaration of the original class is not modifiable. The rename approach can be used in that case although it requires that the referrer classes and the factory classes are modifiable. As the replace approach, the rename approach is not available if the heterogeneity feature is required. Developers should apply this approach to classes like java.awt.Window.

In the rename approach, Addistant generates a proxy class of the original class Widget with a different name such as WidgetProxy. Then Addistant uses that proxy class for remote references. It modifies the bytecode of all the referrer classes on the hosts where references to the Widget objects are remote so that all the occurrences of the original class name Widget are replaced with the proxy class name WidgetProxy. Addistant does not modify the other referrer classes on the host where references to the Widget objects are local.

Addistant also modifies the factory classes if they are used on the host where references to the Widget objects are remote. Since the Widget objects must be created on the other host, Addistant also replaces all the occurrences of Widget with WidgetProxy in the bytecode of the factory classes. For example, it translates the following statement:

```
Frame w = new Frame();
```

into this statement:

```
FrameProxy w = new FrameProxy();
```

The latter statement creates a proxy object, which requests a remote host to create a master object.

Subclass Approach.

The third approach is the *subclass* approach. It is available even if the heterogeneity feature is required. Developers should apply this approach to classes like java.util.Vector.

In this approach, a proxy class WidgetProxy is a subclass of the original class Widget. Both local and remote references have the reference type to Widget so that they can coexist on the same host. If a reference is local, it points to a Widget object. If a reference is remote, it points to a WidgetProxy object.

However, as the rename approach, this approach needs to modify the factory classes if they are used on the host on which references to master objects are remote. Furthermore, this approach may require that the original class is modifiable. First, if the original class is a `final` class or it includes a `final` method, it must be modified to be a non-`final` class and to include no `final` method. Otherwise, the proxy class cannot be a subclass of the original class or override methods declared in the original class. Second, if the constructor of the original class causes inappropriate side-effects and fails to create an object, Addistant must add to that class another constructor performing nothing so that the constructor of the proxy class can call it. Remember that a constructor must call a constructor of the super class in Java. For example, the original constructor of the class Widget may access a local graphic device. If it is called by the constructor of WidgetProxy (since WidgetProxy is a subclass of Widget), it may throw an error because of the absence of the graphic device on the host where the WidgetProxy object is created.

Note that the subclass approach does not require that the original class is modifiable if the original class is not a `final` class and the constructors do not cause inappropriate side-effects. For instance, the bytecode of a system class java.io.File is unmodifiable. Since that class is used by other system classes, the referrer classes are also unmodifiable. Thus, even if the heterogeneity feature is not required, either the replace or rename approaches cannot be used for java.io.File. On the other hand, the subclass approach can be used for that class.

Copy Approach.

The last is the *copy* approach. This approach can be used for primitive types such as `int` and classes like java.lang.String, instances of which are immutable.

If the copy approach is chosen for a class C, a remote reference to an instance of C cannot exist. If a local reference to an instance of C is passed to a remote method, Addistant makes a *shallow* copy of that instance on the remote host. A local reference to that copy is passed to the method. Thus, the copy approach cannot be used if a reference must be passed to a remote method in the call-by-reference manner. However, the copy approach does not need to modify bytecode at all.

Addistant also provides a slightly different version of the copy approach:

the *write-back* copy approach. If this approach is chosen, the contents of the copy passed to the remote method are written back to the master object after executing that remote method. For example, suppose that the write-back copy approach is chosen for an array of `byte`. Then in the following code:

```
byte[] buf = ... ;
inputstream.read(buf);
```

the call to `read()` on a remote object `inputstream` makes a copy of `buf` on the remote host. A local reference to that copy is passed to `read()`. Since the write-back copy approach is chosen, the contents of that copy are written back to `buf` after executing `read()`. Therefore, the byte data read from the input stream are eventually stored in `buf`.

5.1.3 Object Allocation

Addistant allows the developers to specify a policy of object allocation for each class. It does not allow to use a different policy for each object because Addistant is a tool for modifying legacy software; it is not realistic for the developers to specify a policy for every occurrence of "`new`" (the operator of object creation) appearing in a program, which someone else may have written.

The developers can declare that all the instances of a class are allocated on a specific host. If a host D is specified for a class `C`, an expression "`new C()`" (create an instance of `C`) executed on any host is interpreted as that an instance of `C` is created on the (probably remote) host D. On the other hand, if any host is not specified for the class `C`, an expression "`new C()`" executed on a host D' is interpreted as that an instance of `C` is locally created on the host D'.

The declaration by the developers is written in a policy file, which Addistant reads at startup time. The policy file is written in an XML-like syntax. For example, a declaration below:

```
<import proxy="rename" from="display">
    java.awt.*
</import>
```

means that all the instances of classes included in the `java.awt` package are allocated on a host specified by a variable `display`. Remote references to these instances are implemented with the rename approach. The variable `display` is bound to a real host name at run time. If the "`from`" attribute is not given, the instances of a class `C` are allocated on a host where an expression "`new C()`" is executed.

Note that `java.awt.*` means all the classes included in the `java.awt` package. It does not mean sub-packages of `java.awt`, such as `java.awt.image` because sub-packages are irrelevant to the parent package with respect to the language semantics. For example, the access rights of a class in a sub-package are equivalent to ones in other packages than the parent package of that sub-package. To specify all the classes and sub-packages in `java.awt`, `java.awt.-` should be used.

Besides all classes included in a package, all subclasses of a class in a package can be specified. For example, a declaration below:

```
<import proxy="rename" from="display">
    subclass@java.awt.Component
</import>
```

means that the rename approach is used for all the subclasses of the class Component, including Component itself. To specify only the subclasses excluding the parent class, `exactsubclass` should be used instead of `subclass`.

Some implementation approaches of remote references restrict policies of object allocation. Since the replace and rename approaches require that local and remote references do not coexist, the "`from`" field must be specified so that instances are created on the same host. On the other hand, the copy approach does not allow the developers to specify the "`from`" field since it does not deal with remote references.

5.1.4 Bytecode Delivery

Addistant provides a mechanism for automatically distributing bytecode from a host to other hosts. The users have to only run a class loader of Addistant on every host. If a program starts on a host A and creates an object on a remote host D, the class loader on the host A sends necessary bytecode to the class loader on the host D so that the object can be created on the host D. If the bytecode must be modified, it is modified by the class loader on the host A before it is sent to the host D. If the bytecode is of system classes, the class loader on the host D loads it from a local file system instead of the host A.

Although the regular class loader of Java fetches bytecode on demand, the class loader of Addistant may fetch the bytecode of certain classes in advance. For example, suppose that the rename approach is specified for all subclasses of a class C. If the class loader of Addistant loads a class U, it must read the bytecode of the class specified by every name appearing in the bytecode of U and examine whether each class is a subclass of C. If so, the class name must be replaced with the name of the proxy class. Thus, while the class loader of Addistant loads a class U, it may fetch a number of other classes as well as the class C and subclasses of C.

5.2 Implementation Issues

5.2.1 Single System Image

There are several implementation issues for keeping the semantics of the Java language in distributed program execution, that is, providing a single system image with multiple JVMs. This sub section describes how Addistant deals with those issues.

Remote Field Access.

Although a naive implementation of the proxy-master model cannot support remote field accesses, Addistant translates a field access at the bytecode levelinto a static method invocation on that class and thereby enables remote field accesses. Suppose that a class Point declares a field `x` and the field is accessed as follows:

```
Point p;
..  = p.x ..
..  p.x = 100 ..
```

If remote references to Point objects are implemented by the rename approach, the code above is translated into the code below:

```
PointProxy p;
.. = PointProxy.read_x(p) ..
.. PointProxy.write_x(p, 100) ..
```

The static methods `read_x()` and `write_y()` implement the remote field accesses. They are declared in the proxy class produced by Addistant.

The translation above must be applied to all the remote field accesses. Therefore, Addistant cannot deal with remote field accesses embedded in the unmodifiable bytecode, for example, the bytecode of the system classes.

Equality between Remote References.

Addistant preserves the semantics of the equality operators such as "`==`" and "`!=`" with respect to remote references. To do that, Addistant maintains a table of proxy objects on every host so that there exists only a single proxy object referencing to each master object. Addistant gives a unique identifier to every master object and sends this identifier when a reference to the master object is passed as a parameter across the network to a remote method. Then it looks up the corresponding proxy object in the table and passes a reference to that proxy object to the destination method. If the proxy object is not found in the table, Addistant creates and registers it in the table.

Self Deadlock Avoidance.

In Addistant, any host can invoke a method on a remote object and receive a method invocation from a remote object. Therefore, a remote method call from a host A to a host D may cause another method call back from D to A. In this case, the latter method call must be handled by the same thread that requested the former method call on the host A. Otherwise, a deadlock may occur if the methods are synchronized ones.

Suppose that a Button object and a Listener object exist on different hosts D (display host) and A (application host), respectively. The declarations of class Button and Listener are as follows:

```
class Button {
    Listener listener;
    synchronized void push() {
        listener.pushed(this);
    }
    synchronized ButtonState getState() { ... }
}

class Listener {
    void handlePush(Button button) {
        .. button.getState() ..
    }
}
```

If `push()` is invoked on the Button object, it calls `handlePush()` on the remote Listener object. Then `getState()` is called back on the Button object. If `push()` and `getState()` are executed by different threads, a deadlock occurs since the two threads try to lock the Button object at the same time. The deadlock never occurs if the two objects exist on the same host because all the methods are executed by the same thread.

In order to ensure the same thread executes all the methods called back, Addistant establishes a one-to-one communication channel between the thread executing `push()` on D (T_D^i) and the one executing `handlePush()` on A (T_A^i). This communication channel is stored in a thread local variable implemented with java.lang.ThreadLocal. A thread always uses the same channel for every remote method invocation and it waits for not only the result of the invocation but also another request of invocation from a remote thread sharing the same channel. When `handlePush()` calls `getState()`, the thread T_A^i sends a request of `getState()` to the remote thread T_D^i connected through the communication channel, which is the thread executing `push()` on D and blocking to wait for the result of `handlePush()`.

The thread T_D^i invokes the requested `getState()` to send the result of `getState()` through the channel, and then it continues to wait for the result of the original `handlePush()`. Thereby, both `push()` and `getState()` are executed by the same thread T_D^i. A deadlock is avoided.

Distributed Garbage Collection.

Addistant maintains a table of objects exported to a remote host. While there exists a proxy object on a remote host, the master object is recorded in that table so that it is not garbage collected. If all the proxy objects are garbage collected, then the master object is removed from the table. If there are no other references to the master object, then the master object is garbage collected.

The table of proxy objects for checking the equality between remote references is implemented with the weak reference mechanism of Java[2]. An element of the table is a weak reference to a proxy object. Thus, the proxy object is garbage collected when the garbage collector determines that nothing except that table refers to the proxy object.

Currently, Addistant cannot collect all objects if remote references make cycles. Although several algorithms are known for dealing with distributed cycles, efficiently implementing those algorithms is not straightforward without modifying the JVM. For example, if using a distributed mark-sweep algorithm, we would need a mechanism for tracing object references. However, Java's reflection API does not provide such a mechanism. We expect that weak references and object finalizers might help to solve this problem but implementation details are still open.

5.2.2 Bytecode Modification

Bytecode Translation Toolkit.

One of the research aims of the development of Addistant was to examine the expressive power of Javassist[17, 20], which is our toolkit for implementing a bytecode translator for Java. Unlike other similar toolkits, Javassist provides a source-level view of bytecode for the developers, who can manipulate bytecode without detailed knowledge of the bytecode specifications. Javassist is easier to use than other naive toolkits as a source-level debugger is easier to use than an assembly-level debugger. On the other hand, Javassist restricts the ability to modify bytecode. It does not allow bytecode modification that is difficult to express with a source-level view.

To show that the expressive power of Javassist is powerful enough to implement a real application, we have developed Addistant within the confines of the Javassist API (Application Programming Interface). No undocumented low-level API was used. All the bytecode modification that Ad-

distant needs could be easily implemented with a source-level abstraction provided by Javassist.

Bootstrap Classes.

If we use a command-line option provided by Sun's JVM, we can modify the bytecode of system classes and have the JVM load the modified bytecode at bootstrap time. Hence using this option extends the range of the classes that the approaches provided by Addistant for implementing the proxy-master model are applicable to. However, we did not modify the bytecode of the system classes because Sun's license terms prohibit the modification. Even if we could modify, consistently modifying the system classes is difficult since runtime systems such as a system class loader depends on the definition of the system classes.

5.3 Distributed Swing Applications

This section presents that Addistant can adapt legacy software using the Swing class library so that GUI objects are allocated on a remote host and the users can interact with the software through the GUI shown on a remote display. The Swing class library is a GUI library included in the standard Java runtime environment. Although the same effects can be achieved with the X Window system[70], Addistant can achieve better performance since drawing operations are directly performed on the host with a display. This is typical benefit of functional distribution. The X Window system needs network communication for every primitive drawing operation and hence communication overheads tend to be a performance bottleneck.

In this section, we first present a policy file for adapting legacy software using the Swing class library to distributed execution. Then we show the results of our performance measurement.

5.3.1 Policy File

The following is a typical policy file for adapting software using the Swing class library:

```
<policy>
    <import proxy="rename" from="display">
        subclass@java.awt.-
        subclass@javax.swing.-
        subclass@javax.accessibility.*
        subclass@java.util.EventObject                    </import>
    <import proxy="rename" from="application">
        exactsubclass@java.io.[InputStream|OutputStream|Reader|Writer]
        exactsubclass@javax.swing.filechooser.*           </import>
    <import proxy="subclass">
        subclass@java.util.[Dictionary|AbstractCollection|AbstractMap]
```

```
        subclass@java.util.BitSet                          </import>
    <import proxy="writeBackCopy">
        array@-                                            </import>
    <import proxy="replace" from="application">
        user@-                                             </import>
    <import proxy="copy">
        -                                                  </import>
</policy>
```

Here, the variable `display` indicates the host where the GUI objects are allocated. The variable `application` indicates the other host where the rest of the objects are allocated. An `import` declaration listed above has a higher priority.

This policy file specifies that GUI objects are allocated on the `display` host and remote references to those objects are implemented with the rename approach. Any array type (`array@-`) is processed with the write-back copy approach. The instances of classes except the system classes (`user@-`) are allocated on the `application` host and remote references to them are implemented with the replace approach. The rest of the classes are processed with the copy approach.

5.3.2 Performance Measurement

For performance measurement, we used two host computers. One is a machine with a 500MHz PentiumIII and Linux 2.2. It is a display server for executing GUI objects. The other is a machine with a 440MHz UltraSparcII and Solaris 2.7. It is an application server for executing the other objects. We used the HotSpot JVM (JDK 1.3) for both machines. For connecting the two machines, we used two kinds of network: 100Base-TX full-duplex and 10Base-T half-duplex.

Remote Method Invocation.

Before measuring the performance of a GUI, we compared the execution time of remote invocations of empty methods among Addistant (AD) and other Java-based object request brokers (ORB), which are HORB[39] version 2.0.1, Java RMI (JRMI) included in JDK 1.3, and Java Class Broker[37] (JCB) version 1.2. We changed the number and types of parameters and measured the elapsed time of each remote method invocation. We also changed the network connecting the two hosts.

Table 5.3 lists the results. The results showed that Addistant achieved a comparable performance to other ORB except the case that a `byte` array was passed. This is because the parameter encoder/decoder of Addistant had not been tuned and because Addistant used the write-back copy approach for passing an array as a parameter although the other ORB did not write the updated contents of the array back after executing a method. In the

Table 5.3: Elapsed Time (milliseconds) for a remote method invocation. AD indicates Addistant.

(ms)	(100Base-TX full-duplex)				(10Base-T half-duplex)			
	HORB	JRMI	JCB	AD	HORB	JRMI	JCB	AD
`void f()`	0.33	0.52	0.71	0.28	0.48	0.69	0.86	0.40
`void f(int)`	0.33	0.53	0.78	0.48	0.48	0.69	0.93	0.61
`int f(int)`	0.34	0.54	1.20	0.54	0.49	0.71	1.42	0.68
`void f(int,int,int)`	0.34	0.54	0.75	0.76	0.49	0.71	0.91	0.91
`int f(int,int,int)`	0.34	0.55	1.17	0.83	0.50	0.72	1.40	0.99
`void f(String)`	0.34	0.58	0.83	0.36	0.50	0.75	1.00	0.48
`String f(String)`	0.35	0.63	0.94	0.37	0.52	0.81	1.11	0.50
`void f(String[])`	0.58	0.87	1.26	0.66	0.77	1.08	1.46	0.85
`String[] f(String[])`	0.84	1.22	1.66	0.79	1.10	1.50	1.94	0.99
`void f(byte[])`	0.69	0.94	1.26	2.76	1.51	1.73	2.08	2.93

measurement, all String type parameters included 10 ASCII characters. The size of String array was three. The size of `byte` array was 1024.

Window Drawing.

To measure the performance of a GUI, we prepared three Java programs. The first one displays a single window (a java.awt.Frame object) containing no components. The second displays a single empty internal window (a javax.swing.JInternalFrame object) in a window (a javax.swing.JFrame object). The third displays a single internal window containing twenty buttons (a javax.swing.JButton object) in a window. The size of the window is 600 by 600 while the size of the internal window is 500 by 500.

We compared the X Window system[70], Rawt[41], and Addistant by measuring the elapsed time that each program took for creating and drawing a window (and internal windows) on a remote display. The X Window system showed a window on a remote display by connecting a remote X server. Rawt is a GUI library that is compatible to the Swing class library but enables to show a window on a remote display. Addistant showed a window on a remote display by allocating GUI objects on the remote host with that display.

Table 5.4 listed the results. As a drawing image becomes more complex, Addistant showed better performance against Rawt because Rawt allocates only part of instances of the Swing classes on the remote host with a display and thus it needs a larger number of remote method invocations for drawing a window. On the other hand, Addistant allocates all instances of the Swing classes on the remote host and thus the interactions among the Swing objects are local method invocations. This is because Addistant is a general-purpose bytecode translator and it allows the developers to easily customize object allocation for maximizing performance. Rawt cannot do that since the implementation of Rawt is a black box.

Table 5.4: The elapsed time (seconds) for drawing a window.

	X Window	Rawt	Addistant
(100base-TX full-duplex)			
No components	0.005	0.041	0.044
1 internal window	0.156	1.814	0.276
20 buttons	0.873	17.599	0.955
(10base-T half-duplex)			
No components	0.006	0.041	0.045
1 internal window	0.612	1.988	0.281
20 buttons	1.895	21.322	0.971

Since the X Window system asynchronously executes an X server and an X client, the elapsed time listed in the table indicates the time needed for sending all the requests from the client to the server. It does not indicate the actual elapsed time of drawing a window. In fact, we observed that the response time of the GUI implemented on top of the X Window system was considerably slower than one on top of Addistant.

To confirm our observation above, we conducted another experiment. We wrote a Java program that displays a button in a window and, if that button is clicked, then a graphic image (1148 by 778) is shown in the window. Table 5.5 listed the results of our experiment. We measured the elapsed time after the button was clicked by mouse until the image was shown. The time was measured by hand. 0.0 means that the response time was too short to measure. Since the Swing class library caches a drawn image, Rawt and Addistant responded quicker than the X Window to a mouse click at the second time. The X Window must transfer the drawn image every time from the client to the server. Even at the first time, Addistant achieved the best performance if the network is 10Base-T since the X Window system and Rawt had to transfer a larger amount of data between the hosts. Table 5.6 listed the results of our measurement of the size of the data exchanged through a network during the above interaction. The X Window system needs a few megabytes whereas Addistant does less than a hundred kilobytes. The large amount of exchanged data can be a performance bottleneck.

Table 5.5: The response time (seconds) to a mouse click.

	(100Base-TX full-duplex)			(10Base-T half-duplex)		
(sec.)	X Window	Rawt	Addistant	X Window	Rawt	Addistant
1st	1.6	2.6	2.0	5.6	3.2	2.0
2nd	1.4	0.0	0.0	5.6	0.0	0.0

Table 5.6: The size of the data (Kbyte) exchanged through a network.

	X Window	Rawt	Addistant
1st	3493.57	116.20	81.88
2nd	3438.96	10.95	0.06

5.4 Related Work

Transparent Distribution

To run a Java program on a distributed environment, several extended Java virtual machines have been developed. These virtual machines such as cJVM[1], Java/DSM[87], and JESSICA[58] provide a single-machine image on several network-connected computers, that is, a workstation/PC cluster. Thus, multiple threads are executed in parallel as if they were running on a multi-processor machine with shared memory. These virtual machines do not need to modify a program at all to run it. A difference between Addistant and these virtual machines is that Addistant uses the standard JVM and hence it is mainly for functional distribution, where objects run on the most suitable host for the computation by the objects.

JavaParty[67] extended the Java language for parallel distributed computing. They introduced only the extended modifier "`remote`" for class declarations. Although the users of Addistant do not have to modify a program, the users of JavaParty have to append an extended modifier "`remote`" to a class declaration if an instance of that class is accessed through a remote reference.

There are a number of object request brokers for Java. Most of them, including the Java RMI[76], require that a remote object be accessed through an interface type. Thus, developers may have to largely modify programs if they adapt legacy software to distributed execution. Java Class Broker[37] avoids this problem by a technique similar to our subclass approach. However, it requires developers to modify a program to follow another programming convention. For example, the following regular Java program:

```
Frame f = new Frame("The Great Encyclopedia");
Button b = new Button();
f.add(b);
```

must be translated into a program using a runtime distribution manager object `objectBroker`:

```
Object[] params = {"The Great Encyclopedia"};
Frame f = (Frame) objectBroker.create("Frame", params);
Button b = (Button) objectBroker.getProxy("Button", new Button());
f.add(b);
```

Remote Display

The X Window System[70] enables a Java program to show a graphical output on the display of a remote host. Like Addistant, the X Window System does not require developers to modify their programs to use a remote display. However, as shown in Section 5.3, the X Window System is often less efficient than Addistant.

Rawt[41] is a GUI library that is compatible to the standard Java GUI library. If substituting Rawt for the standard library, developers can extend their programs without any other modifications to use a remote display for output. Underlying network communication is encapsulated by that library. Addistant can be regarded as a tool for semi-automatically producing a library like Rawt from the standard Java GUI library. Since the production by Addistant is based on both the library and user code, however, the resulting software can often achieve better performance than Rawt.

Aspect-Oriented Programming

With Addistant, developers describe a policy file for adapting software to distributed execution. This policy file can be considered as a separate description of a distribution aspect in the context of aspect-oriented programming (AOP). In this context, Addistant is a tool for *weaving* a Java program written for a single JVM and a description separately written about a distribution aspect.

Proposing a distribution aspect is not new. For example, D[57] provides an aspect language for distribution. However, it allows programmers to separately describe how a parameter is passed to a remote procedure whereas Addistant allows to describe where objects are allocated and how proxy objects are implemented. Furthermore, it seems that the design goal of D is to support the development of distributed software from scratch. The goal of Addistant is to add a new aspect on existing software for adaptation. Thus, the description in a policy file is not a part of program text but rather *meta-level* instructions to modify an existing program.

5.5 Summary

This chapter presented Addistant, which is a programming tool for adapting legacy Java software to distributed execution. Addistant performs this adaptation by bytecode translation at load time. No source code is needed for the adaptation. The users of Addistant have only to write a policy file for specifying where the instances of each class are allocated and how remote references to those instances are implemented. The users can select an implementation approach from the four provided by Addistant.

Although the four implementation approaches are not new, a contribution of this chapter is that it reveals that letting developers select an implementation approach for each class is necessary for adapting legacy Java software in the real world to distributed execution. This chapter presented several practical issues that we must consider for the adaptation. However, the ability of Addistant still has a few limitations. Although the developers using Addistant do not need to read or modify source code, they must have some knowledge of source code, for example, which class of objects should be allocated on a remote host. Moreover, Addistant provides only class-based distribution: all the instances of a class must be allocated on the same host. These limitations are acceptable in our GUI examples although it is an open question in other contexts.

This chapter also showed that Addistant could adapt a Java program using the Swing class library so that GUI objects could be allocated on a remote host with a display. This functional distribution with Addistant showed better response time of the GUI than the distribution with the X Window System and the Rawt class library. This fact suggests that library-level functional distribution could not give good performance since only the library code is split and distributed to multiple hosts. On the other hand, Addistant can split a whole program including both user and library code and then it can distribute objects so that the maximum performance could be obtained.

Chapter 6

Conclusion

This thesis proposed a class-object model which is a new abstract data model for the transformation of object-oriented programs. The key concept of the class-object model is to capture the object-orientation of programs. Designing metaobject protocols considering the mechanism of abstract data types in the targeting object-oriented language enables direct manipulation of object-oriented constructs. Declarative language constructs, such as classes and inheritances, and the capsulation mechanism must be considered. This model allows a transformational system to provide a sophisticated interface to metaprograms through which metaprogrammers can simply and intuitively describe transformation of object-oriented language constructs in programs translated.

This contribution of the proposed model suggests a new design approach for programming support, especially for programming languages. Systems using the proposed design model make it more commonplace to provide transformations as reusable software artifacts. Now, powerful transformations are not only of the compiler experts but also of the object-oriented programming experts, who can be expected to have a lot of knowledge worth reusing in software engineering.

This thesis also contributed by giving three practical applications of the class-object model for program transformations. We built three transformational systems using this model. The design space of three systems differs to each other's in the perspective of generality or the format of targeting programs. OpenJava and Javassist supply their users with generic metaobject protocols for transforming object-oriented programs, while Addistant supplies their users a programming interface that is specific to distributed domain. OpenJava transforms source-text programs while Javassist and its application Addistant transform bytecode programs.

In addition to a contribution as practical applications of the class-object model, there are contributions by each transformational system. The first

system is OpenJava. OpenJava is an object-oriented macro system employing the class-object model for transformations of source-text program written in Java. Instead of abstract syntax tree, it provides an abstract data structure which represents a logical structure of object-oriented program. This made it easier to describe typical macros for object-oriented programming which was difficult to describe with ordinary macro systems. To show the effectiveness of OpenJava, we implemented some macros in OpenJava for supporting programming with design patterns. From the point of reflection, OpenJava provides a mechanism for a compile-time structural reflection with a limited syntax extensibility. With structural reflection, metaprogrammers change the structure of a program while they change the behavior of a program with ordinal runtime reflection, which we call behavioral reflection.

The second system is Javassist. Javassist is a Java bytecode manipulating tool employing the class-object model for transformations of binary programs of Java. Javassist provides source-level abstraction for manipulating bytecode in a safe manner while bytecode translators, such as JOIE [21] and the BCEL [25], provide no higher-level abstraction. The users of Javassist do not have to have a deep understanding of the Java bytecode or to be careful for avoiding wrongly making an invalid class rejected by the bytecode verifier. Like OpenJava, Javassist provides a mechanism for a structural reflection but its reflective computation is done at load-time at which classes are loaded on Java virtual machines. We call this a load-time structural reflection. It allows a program to alter a given class definition and to dynamically define a new class. A number of language extensions are more easily implemented with structural reflection than with behavioral reflection.

The third system is Addistant. Addistant is a programming tool for adapting legacy Java software to distributed execution. It is built upon Javassist and performs this adaptation by bytecode translation at load time. No source code is needed for the adaptation. The users of Addistant have only to write a policy file for specifying where the instances of each class are allocated and how remote references to those instances are implemented. The users can select an implementation approach from the four provided by Addistant. Although the four implementation approaches are not new, a contribution of Addistant is that it reveals that letting developers select an implementation approach for each class is necessary for adapting legacy Java software in the real world to distributed execution. We presented several practical issues that we must consider for the adaptation.

A future direction of this research is to construct frameworks for AOP (aspect-oriented programming) support. AspectJ [48] provides a general-purpose AOP support. With AspectJ, programmers can describe separated code called *aspects* which was originally difficult to separate without the

AOP support. However, aspects are often less reusable in AspectJ. Domain-specific aspect languages can solve this problem and a transformational system with the proposed model can be a generic framework for domain-specific aspect language. In fact, Addistant can be regarded as a distributed domain-specific aspect-oriented programming language.

Bibliography

[1] Aridor, Y., Factor, M., and Teperman, A. CJVM: a single system image of a JVM on a cluster. In *Proceedings of ICPP '99* (1999), IEEE.

[2] Arnold, K., Gosling, J., and Holmes, D. *The Java Programming Language*, 3 ed. Addison Wesley, 2000, ch. Chapter 12, Garbage Collection and Memory, pp. 313–327.

[3] Bachrach, J. The Java syntactic extender. In *Proceedings of OOPSLA 2001* (Tampa, Florida, USA, October 2001), no. 10 in SIGPLAN Notices vol.36, ACM, pp. 31–42.

[4] Bal, H. E., Steiner, J. G., and Tanenbaum, A. S. Programming languages for distributed computing systems. *ACM Computing Surveys 2*, 3 (1989), 261–322.

[5] Balzer, R. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering 11*, 11 (November 1985), 1257–1268.

[6] Batory, D., Lofaso, B., and Smaragdakis, Y. JTS: Tools for implementing domain-specific languages. In *Proceedings of ICSR'98, Fifth International Conference on Software Reuse* (Victoria, B.C., Canada, June 1998), IEEE Press.

[7] Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L. Distribution and abstract types in emerald. *IEEE Transactions on Software Engineering SE-13*, 1 (January 1987), 65–76.

[8] Bosch, J. Language support for design patterns. In *TOOLS Europe '96* (Paris, France, Feburary 1996).

[9] Bosch, J. Design patterns as language constructs. *Journal of Object Oriented Programming 11*, 2 (May 1998), 18–32.

[10] Brant, J., Foote, B., Johnson, R. E., and Roberts, D. Wrappers to the rescure. In *ECOOP'98 - Object Oriented Programming* (1998), LNCS 1445, Springer, pp. 396–417.

[11] Braux, M., and Noyé, J. Towards partially evaluating reflection in java. In *Proc. of Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)* (1999), no. 11 in SIGPLAN Notices vol. 34, ACM, pp. 2–11.

[12] Briot, J.-P., and Cointe, P. Programming with explicit metaclasses in smalltalk-80. In *Proceedings of OOPSLA '89* (New Orleans, Louisiana, USA, October 1989), N. K. Meyrowitz, Ed., no. 10 in SIGPLAN Notices vol.24, ACM, pp. 419–431.

[13] Brown, P. J. *Macro Processors and Techniques for Portable Software*. Wiley, 1974.

[14] Caromel, D., Klauser, W., and Vayssièra, J. Towards seamless computing and metacomputing in java. *Concurrency: Practice & Experience 10*, 11-13 (November 1998), 1043–1061.

[15] Chiba, S. A metaobject protocol for C++. In *Proceedings of OOPSLA'95* (1995), no. 10 in SIGPLAN Notices vol.30, ACM, ACM Press, pp. 285–299.

[16] Chiba, S. Macro processing in object-oriented languages. In *Proceedings of TOOLS Pacific '98* (Australia, November 1998), IEEE, IEEE Press.

[17] Chiba, S. Load-time structural reflection in Java. In *ECOOP 2000 - Object Oriented Programming* (Sophia Antipolis and Cannes, France, June 2000), LNCS 1850, Springer-Verlag, pp. 313–336.

[18] Chiba, S., Kiczales, G., and Lamping, J. Avoiding confusion in metacircularity: The meta-helix. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS)* (1996), LNCS 1049, Springer, pp. 157–172.

[19] Chiba, S., and Masuda, T. Designing an extensible distributed language with a meta-level architecture. In *Proc. of the 7th European Conference on Object-Oriented Programming* (1993), LNCS 707, Springer-Verlag, pp. 482–501.

[20] Chiba, S., and Tatsubori, M. Structural reflection by Java bytecode instrumentation. *IPSJ Journal 42*, 11 (2001), 2752–2760. (In Japanese).

[21] Cohen, G. A., Chase, J. S., and Kaminsky, D. L. Automatic program transformation with JOIE. In *USENIX Annual Technical Conference '98* (New Orleans, Louisiana, USA, June 1998), USENIX.

[22] Cointe, P. Metaclasses are first class : the ObjVlisp model. *SIGPLAN Notices 22*, 12 (December 1987), 156–162.

[23] Czarnecki, K., and Eisenecker, U. W. *Generative Programming — Methods, Tools, and Applications*. Addison-Wesley, June 2000.

[24] Dahl, O.-J., Myhrhaug, B., and Nygaard, K. SIMULA 67 common base language. Tech. Rep. S–2, Norwegian Computer Centre, Oslo, Norway, 1970.

[25] Dahm, M. Byte code engineering with the javaclass api. Techincal Report B-17-98, Institut für Informatik, Freie Universität Berlin, Berlin, Germany, July 1999.

[26] Ducasse, S. Message passing abstractions as elementary bricks for design pattern implementation. In *Object-Oriented Technology, ECOOP workshop Reader* (1997), LNCS 1357, Springer.

[27] Dybvig, R. K., Friedman, D. P., and Haynes, C. T. Expansion-passing style: A general macro mechanism. *Lisp and Symbolic Computation 1*, 1 (June 1988), 53–75.

[28] Ellis, M., and Stroustrup, B., Eds. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[29] Ellis, M., and Stroustrup, B., Eds. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[30] Fabre, J.-C., and Pérennou, T. A metaobject architecture for fault tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers 47*, 1 (1998), 78–95.

[31] Futamura, Y. Partial computation of programs. In *Proceedings of RIMS Symposia on Software Science and Engineering* (1982), LNCS 247, Springer, pp. 1–35.

[32] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[33] Gil, J., and Lorenz, D. H. Design patterns and language design. *IEEE Computer 31*, 3 (March 1998), 118–120.

[34] Goldberg, A., and Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison Wesley, 1983.

[35] Golm, M., and Kleinöder, J. Jumping to the meta level, behavioral reflection can be fast and flexible. In *Proc. of Reflection '99* (1999), LNCS 1616, Springer, pp. 22–39.

[36] Gosling, J., Joy, B., and Steele Jr., G. L. *The Java Language Specification*. Addison-Wesley, 1997.

[37] Har'El, Z., and Rosberg, Z. Java class broker - a seamless bridge from local to distributed programming. *Parallel and Distributed Computing 60*, 11 (2000), 1223–1237.

[38] Hicks, M., Jagannathan, S., Kelsey, R., Moore, J. T., and Ungureanu, C. Transparent communication for distributed objects in Java. In *Proceedings of the ACM 1999 conference on Java Grande* (Palo Alto, CA USA, June 1999), pp. 160–170. June 12-14, 1999,.

[39] Hirano, S. HORB: Distributed execution of Java programs. In *Proceedings of WWCA '97* (1997).

[40] Hölzle, U., and Ungar, D. A third generation self implementation: Reconciling responsiveness with performance. In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications* (1994), no. 10 in SIGPLAN Notices vol. 29, pp. 229–243.

[41] IBM. Remote abstract windowing toolkit (rawt). `http://www.s390.-ibm.com/java/rawt.html`, July 2000.

[42] Ichisugi, Y., and Roudier, Y. Extensible Java preprocessor kit and tiny data-parallel Java. In *Proceedings of ISCOPE'97* (California, December 1997).

[43] Ishikawa, Y., Hori, A., Sato, M., Matsuda, M., Nolte, J., Tezuka, H., Konaka, H., and Kubota, K. Design and implementation of metalevel architecture in C++ - MPC++ approach -. In *Proceedings of Reflection'96* (April 1996), pp. 153–166.

[44] JavaSoft. Java core reflection api and specification. online publishing, January 1997.

[45] Johnson, R. E., and Foote, B. Designing reusable classes. *Journal of Object-Oriented Programming 1*, 2 (1988), 22–35.

[46] Kalé, L. V., Bhandarkar, M., and Wilmarth, T. Design and implementation of parallel Java with global object space. In *Proceedings of PDPTA '97, Conference on Parallel and Distributed Processing Technology and Applications* (LasVegas, Nevada, USA, 1997), pp. 29–42.

[47] Keller, R., and Hölzle, U. Binary component adaptation. In *ECOOP'98 - Object Oriented Programming* (1998), LNCS 1445, Springer, pp. 307–329.

[48] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Grisworld, W. G. An overview of AspectJ. In *ECOOP 2001 - Object*

Oriented Programming (Budapest, Hungary, June 2001), L. Knudsen, Ed., LNCS 2072, Springer-Verlag, pp. 327–353.

[49] Kiczales, G., and Lamping, J. Issues in the design and specification of class libraries. In *Proceedings of OOPSLA'92* (1992), pp. 435–451.

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

[51] Kierby, G. N. C., Morrison, R., and Stemple, D. W. Linguistic reflection in Java. *Software — Practice and Experience 28*, 10 (August 1998), 1045–1077.

[52] Kleinöder, J., and Golm, M. MetaJava: An efficient run-time meta architecture for java. In *Proc. of the International Workshop on Object Orientation in Operating Systems (IWOOS'96)* (1996), IEEE.

[53] Kohlbecker, E., Friedman, D. P., Felleisen, M., and Duba, B. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA, August 1986), ACM, ACM Press, pp. 151–161.

[54] Krueger, C. W. Software reuse. *ACM Computing Surveys 24*, 2 (1992), 131–183.

[55] Ladd, D. A., and Ramming, J. C. A* : A language for implementing language processors. *IEEE Transactions on Software Engineering 21*, 11 (November 1995), 894–901.

[56] Liang, S., and Bracha, G. Dynamic class loading in the Java virtual machine. In *Proceedings of OOPSLA '98* (1998), no. 10 in SIGPLAN Notices vol.33, ACM, ACM Press, pp. 36–44.

[57] Lopes, C. V., and Kiczales, G. D: A language framework for distributed programming. Technical Report SPL97-010, Xerox Palo Alto Research Center, Palo Alto, CA, USA, 1997.

[58] Ma, M. J. M., Wang, C.-L., and Lau, F. C. M. JESSICA: Java-enabled single-system-image computing architecture. *Journal of Parallel and Distributed Computing 60*, 11 (2000), 1194–1222.

[59] Maddox, W. Semantically-sensitive macroprocessing. Tech. Rep. ucb/csd 89/545, University of California, Berkeley, California, USA, 1989.

[60] Maes, P. Concepts and experiments in computational reflection. In *Proceedings of OOPSL'87* (Orland, Florida, USA, October 1987), no. 12 in SIGPLAN Notices vol.22, ACM, ACM Press, pp. 147–155.

[61] Masuhara, H., Matsuoka, S., Asai, K., and Yonezawa, A. Compiling away the meta-level in object-oriented concurrent reflective language using partial evaluation. In *Proceedings of OOPSLA'95* (Austin, Texas, USA, October 1995), no. 10 in SIGPLAN Notices vol.30, ACM, pp. 300–315. October 15-19.

[62] Masuhara, H., and Yonezawa, A. Design and partial evaluation of meta-objects for a concurrent reflective languages. In *ECOOP'98 - Object Oriented Programming* (1998), LNCS 1445, Springer, pp. 418–439.

[63] Meijler, T. D., Demeyer, S., and Engel, R. Making design patterns explicit in FACE, a framework adaptive composition environment. In *Proceedings of ESEC/FSE '97* (September 1997), Springer-Verlag, pp. 94–110.

[64] Musser, D. R., and Stepanov, A. A. Algorithm-oriented generic libraries. *Software — Practice and Experience 24*, 7 (July 1994), 623–642.

[65] Nagaratnam, N., Srinivasan, A., and Lea, D. Remote objects in Java. In *Proceedings of IASTED '96, International Conference on Networks* (January 1996).

[66] Ogawa, H., Shimura, K., Matsuoka, S., Maruyama, F., Sohda, Y., and Kimura, F. Openjit : An open-ended, reflective jit compiler framework for java. In *Proc. of ECOOP'2000* (2000), Springer Verlag. To appear.

[67] Philippsen, M., and Zenger, M. JavaParty - transparent remote objects in Java. *Concurrency: Practice & Experience 9*, 11 (1999), 1225–1242.

[68] Rohnert, H. *The Proxy Design Pattern Revisited.* Addison-Wesley, 1995, pp. 105–118.

[69] Schappert, A., Sommerlad, P., and Pree, W. Automated support for software development with frameworks. In *Proceedings of SSR '95 ACM SIGSOFT Symposium on Software Reusability* (1995), pp. 123–127.

[70] Scheifler, R., and Gettys, J. The X window system. *ACM Transactions on Graphics 5*, 2 (1986), 79–109.

[71] Shalit, A. *The Dylan Reference Manual.* Addison Wesley Longman, 1996.

[72] Simonyi, C. The death of computer languages, the birth of intentional programming. Tech. Rep. MSR-TR-95-52, Microsoft Research, Microsoft Corporation, Redmond, WA 98052, September 1995.

[73] Smith, B. C. Reflection and semantics in lisp. In *Proceedings of POPL '84 ACM Symposium on Principles of Programming Languages* (1984), pp. 23–35.

[74] Soukup, J. Implementing patterns. In *Pattern Languages of Program Design*. Addison-Wesley, 1995, ch. 20, pp. 395–412.

[75] Steele Jr., G. L. An overview of common lisp. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming* (Pittsburgh, PA, USA, August 1982), ACM, pp. 98–107.

[76] Sun Microsystems, Inc. The Java remote method invocation specification. `http://java.sun.com/products/jdk/rmi/`, 1997.

[77] Sun Microsystems, Inc. Java foundation classes. `http://java.sun.-com/products/jfc/`, 1998.

[78] Sun Microsystems, Inc. Java^{TM} 2 SDK documentation. version 1.3, 1999.

[79] Tatsubori, M. Separation of distribution concerns in distributed java programming. In *Addendum to the 2001 Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001 Addendum), Doctoral Symposium* (Tampa Bay, Florida, USA, October 2001), ACM, pp. 19–20.

[80] Tatsubori, M., and Chiba, S. Programming support of design patterns with compile-time reflection. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java* (1998), pp. 56–60.

[81] Tatsubori, M., Chiba, S., and Itano, K. A macro system with class objects for the Java language. *IPSJ Journal 41*, 8 (2000), 2327–2338. (In Japanese).

[82] Tatsubori, M., Chiba, S., Killijian, M.-O., and Itano, K. OpenJava: A class-based macro system for Java. In *Reflection and Software Engineering* (July 2000), W. Cazzola, R. J. Stroud, and F. Tisato, Eds., LNCS 1826, Springer-Verlag, pp. 119–135.

[83] Tatsubori, M., Sasaki, T., Chiba, S., and Itano, K. A bytecode translator for distributed execution of "legacy" Java software. In *ECOOP 2001 - Object Oriented Programming* (Budapest, Hungary, June 2001), L. Knudsen, Ed., LNCS 2072, Springer-Verlag, pp. 236–255.

[84] Weise, D., and Crew, R. Programmable syntax macros. *SIGPLAN Notices 28*, 6 (1993), 156–165.

[85] Welch, I., and Stroud, R. From dalang to kava — the evolution of a reflective java extension. In *Proc. of Reflection '99* (1999), LNCS 1616, Springer, pp. 2–21.

[86] Wu, Z. Reflective java and a reflective-component-based transaction architecture. In *Proc. of OOPSLA'98 Workshop on Reflective Programming in C++ and Java* (1998), J.-C. Fabre and S. Chiba, Eds.

[87] Yu, W., and Cox, A. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice & Experience 9*, 11 (1997), 1213–1224.