An Extension Mechanism for the Java Language

By

Michiaki Tatsubori
mt@is.tsukuba.ac.jp

February 1999

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

In Partial Fulfillment of the Requirements
For the Degree of Master of Engineering

Michiaki Tatsubori

An Extension Mechanism for the Java Language

Master of Engineering Dissertation, Graduate School of Engineering, University of Tsukuba

February 1999

Abstract

This thesis presents the design and implementation of an extensible dialect of the Java language, named OpenJava. Although the Java language is well polished and dedicated to cover applications of a wide range of computational domain, it still lacks some mechanisms necessary for typical kinds of applications. Our OpenJava enables programmers to extend the Java language and implement such mechanisms on demand. It is an advanced macro processor based on the technique called compile-time reflection.

In this thesis, first, the problems of traditional compile-time reflection systems are pointed out; with those systems, it is difficult to write meta-level programs. Those reflective systems are not suitable for macro processing which are non-local, scattered, and spreaded in source programs, although such macro processing are typical in object-oriented languages. Also, those reflective systems do not provide sufficient supports of using different extensions together.

Then this thesis proposes a new compile-time reflection system for OpenJava. With OpenJava, translation of source code is indirectly performed through an abstract data structure called metaobjects. This data structure gives meta programmers an intuitive view in object orientation. Meta programmers in OpenJava can describe extensions of the base Java language more intuitively and safely than in traditional reflective systems. Finally, several examples are presented to show the good usability of OpenJava.

Acknowledgments

I profoundly thank Kozo Itano, who is my superviser in HLLA (High-Level Language and system software Architecrue) laboratory at University of Tsukuba, and this thesis was supervised by him. And, I would like to express my deep gratitude to Shigeru Chiba. He is the man who developped OpenC++, which served as the great stimulus for my study, OpenJava, and he gave me very useful advice around reflection and metaobject protocols. I also thank Ikuo Nakata and Yoshiyuki Yamashita, who was superviser when I belonged to Programming Language Laboratory. There, Teruo Koyanagi helped me to implement the eariest version of OpenJava.

Finally, I thank my thesis committee, Hideki Sakamoto, Yasushi Shinjo and other students in HLLA laboratory. Hidenori Miyamoto, who is also in our laboratory, Chowdhury Takdir Hossain, Yasutaka Sakayori, Marc-Olivier Killijian, Juan-Carlos Ruiz-Garcia and Yukie Itoh are the best frends of mine and put life into me. James Gosling gave me a chance to talk with him and it encourage me to contribute my work to the Java community.

This thesis project was conducted at University of Tsukuba, Ibaraki, Japan.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents the design and implementation of an extensible dialect of the Java language, named OpenJava. Our OpenJava enables programmers to extend the Java language and implement a number of language mechanisms on demand. It is an advanced macro processor based on the technique called compile-time reflection.

These days, the range of computer software is getting wider. The Java language[Gosling *et al.* 97, Kramer 97] is one of the most successful object-oriented languages adopted by a number of users. This language was born in 1995 and thus it is designed to run *anywhere* (on any platform) and to be dedicated to cover a wide range of applications. In fact, Java is a well-designed object-orientated language; it is simple and highly abstracted so that it is easy for programmers to learn. Also, its runtime environment, such as the Java virtual machine and the class loader, gives flexibility to Java programs and thereby application software written in Java is adaptable to various environment.

However, the Java language still lacks some userful mechanisms needed by some kinds of applications. The use of design patterns is a good motivating example to extend the Java language and make higher-level control/data abstractions available in the language. If design patterns are used, some programs are significantly complicated so that the overall structure of the programs is not easy to understand. The programmers have to write annoying and error-prone codes because the concept of design patterns are not directly supported by the Java language.

Furthermore, language supports of distributed computing is another example. To cover a variety of requirements of distributed computing, a lot of compilers including Sun's `rmic` have been developpped, which extend the Java language and provide various language constructs for distributed computing. The problem is that developers need to write their own compiler providing language constructs that are the most suitable for their applications though writing a new compiler and customizing an existing compiler is very difficult.

To address this problem, this thesis proposes an advanced macro processor based on the technique called compile-time reflection. However, traditional compile-time reflection systems has difficulties in writing meta-level programs. Those reflective systems are not suitable for macro processing which are non-local, scattered, and spread in source programs, although such macro processing are typical in object-oriented programming. Since the traditional systems only provide an abstract syntax tree of source programs, meta-level programs must reconstruct the concepts of object orientation appearing in the source programs from a given abstract syntax tree, which is irrelevant to object orientation. Also, they do not provide sufficient supports for using different extensions together. This may cause serious conflicts if multiple extensions are used togather.

To overcome these difficulties, we developed a new compile-time reflection system for OpenJava. With OpenJava, translation of source programs is indirectly performed through an abstract data structure called metaobjects. This data structure gives meta programmers an intuitive view of the source programs in object orientation. Meta programmers in OpenJava can describe extensions of the Java language more intuitively and safely than in traditional reflective systems.

This thesis also presents several examples of language extensions to show the good usability of OpenJava. The first example is language extensions for design patterns, which make it easy to write programs according to design pattern. The second example is language extensions for distributed programming.

From the next chapter, we present background, design, implementaion and application of OpenJava. In chapter 2, what motivates this research is described. chapter 3 discusses important issuses to design the OpenJava MOP. chapter 4 shows how to implement the OpenJava MOP is breafly shown. In chapter 5, we present examples how the OpenJava address to extend the Java language. Finally, we conclude this thesis in chapter 6.

Chapter 2

Motivation

Recently, the Java language[Gosling *et al.* 97] is one of the most successful object-oriented languages, which a number of users adopt. As can be seen in its catch phrase; write once run anywhere, it is machine independent and dedicated to applicability in wide area of computational domain. For the purpose of applicability, the language design is desired to have simplicity and flexibility. In fact, the language constructs are highly abstracted as an object-oriented language and very plain relatively to other practical object-orient languages like C++, and the mechanisms like class loading on Java VM (Virtual Machine)[Lindholm & Yellin 97, Meyer & Downing 97] bring its applications good flexibility. However, while the language has plainness, it lacks flexibility as a language though its applications can have flexibility. This forces programmers to write complex code in special domains.

To be applicable to wide area and to be easy to use, Java needs *flexibility as a language*. As in the past, libraries are flexible mechanism and good libraries can provide a programming environment easy to use such as `java.awt` package, which is included in Java as standard library for graphical user interface programming. But current library mechanism of Java is not enough applicable to all the domain such as transparent distributed programming. It is nearly impossible to write any good library for such environment because of the limitation of the Java language. Good libraries should provide useful and high-level control/data abstractions to improve readability of programs, and intuitive to avoid leading the users to misuse the libraries. In addition to easiness to use, efficiency is also criterion of good libraries.

In this chapter, the motivation of this research is described. First, in section 2.1, we show why extensible languages are needed, by presenting some application examples. Secondly, in section 2.2, we discuss about related researches to point the stage of this research out.

2.1 Extension of Languages

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 begining, 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 Programming according to Design Patterns

Although design patterns[Gamma *et al.* 94] 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 consist of several classes, any single class does not represent the overall structure of the programs, that is, any line 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[Bosch 97, Ducasse 97, Gil & Lorenz 98] 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.

Here we show an example. Suppose that a programmer has to adapt a class Vector (Listing 2.1) to an interface Stack (Listing 2.2), which are defined 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 found in a design patterns catalog[Gamma *et al.* 94] 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[Gamma *et al.* 94].

Figure 2.1 shows a structure of the Adapter pattern.

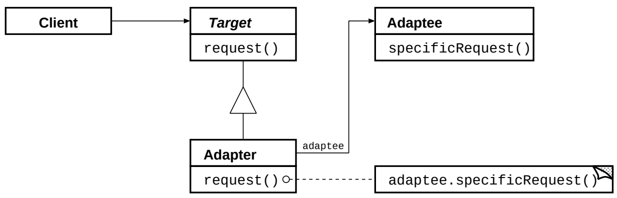

Figure 2.1: A structure of the Adapter pattern

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.[1] Although `isEmpty()` and `elements()` are shared between the class Vector and the class VectorStack, programmers must repeatedly write code for both of them.
3. 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 [Soukup 95, Schappert *et al.* 95, Bosch 96, Meijler *et al.* 97, Ducasse 97]. Bosch called the problem 1, 2 and 3, *traceability loss*, *self problem*, and *implementation overhead* in [Bosch 97].

2.1.2 Distributed Programming

Language extension for distributed programming[Bal 89], as it can be seen in RPC in ancient Unix operating system, are common technique helping programmers to write program easily. As for Java, there are several systems which support distributed object programming. Such a system provide a compiler to translate a source program written in ordinal way but into the one which works in distribute environment and run on the ordinal Java VM (Virtual Machine). In fact, RMI[JavaSoft 97b] by Sun Microsystems provides `rmic` compiler and HORB[Hirano 97] 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 archieve that transparency of programming remote objects, these compilers accepts a source program written as like local object in the ordinal

[1] If the class VectorStack as an innerclass[Kramer 97] of class Vector is defined, this problem is resolved. But this solution is not applicable when the source code of the class Vector is not modifiable.

way and produce a proxy class representing that actual communication to remote systems and a server skelton class working on remote systems.

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

2.2 Related Works

In this section, works related to this research are descibed. The *reflection* is often used as a model of language extension, and it was originally proposed by Simth[Smith 84] 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 comutation, intercession or introspection, is called *meta-level* program while the program which is performed reflective computation to 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 behaivior 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[Cointe 87].

The point is how to define an interface to these metaobjects, and how to realize these mechanisms, which is called MOP (Metaobject Protocol). 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.

First, in section 2.2.1, we reviews fully reflective MOPs founded to the model above but with enhanced execution though it still have some overheads of execution. Secondly, in section 2.2.2, we reviews the researches in systems with *compile-time MOPs*, in which reflective computations are fully performed at compile-time because our approach is based on this technique to achieve intercession without any runtime overhead at the cost of limitation in direct support of runtime alteration of program behavior. At last, in section 2.2.3 the ordinal reflection in regular Java is reviewed to discuss about its limitation because our extensible language, OpenJava, is based on Java language and executable code generated by OpenJava is bytecode which can be run on regular Java virtual machines.

2.2.1 Regular MOPs

CLOS

The CLOS (Common Lisp Object System) MOP[Kiczales 94] 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 employ class metaobjects instead of the metaobjects for objects.

Although the *currying*[Briot & Cointe] technique allows metaobjects in the CLOS MOP to mostly run at compile time, some 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

ABCL/R3[Masuhara *et al.* 95] is a compilation framework in object-oriented reflective languages. In their framwork, 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*[Futamura 82].

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, there seem not to be any effective one for Java.

2.2.2 Compile-time MOPs

Usually, programmers define how to compile program as meta level program through compile-time MOPs other than regular MOPs[Kiczales 94] such as CLOS MOP, through which they define how to execute program. From this point of view, macros like ANSI C preprocessor or macro system of lisp are probably oldest example of compile-time reflection since they are used in order to translate a source program at the meta level. But the problem is that such systems works well only in the case that the basic constructs of languages are simple like procedures and functions.

EPP

EPP[Ichisugi & Roudier 97] is an extensible preprocessor kit for Java and may be regarded as a kind of compile-time MOP. 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 plugins. If only plugins do not cause a collision, the end-user can incorporate multiple plugins into EPP simultaneously. In fact, it is powerful for locally limited translation though programmers must write recursive descent parser.

OpenC++

OpenC++ version 2[Chiba 95, Chiba 96] is the immediate ancestor of the OpenJava MOP. It provides an extensible C++ language, which is one of the most practical languages. 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. 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.

2.2.3 Reflection in Java

The Java already provides 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.

Java Reflection API

The Java Core Reflection API[JavaSoft 97a] 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 appliations 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.

Class Loader

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[Liang & Gilad 98] and Kierby[Kirby *et al.* 98].

Addtion to the execution overhead of loading and modifying, there is difficulty of programming because it is not easy to manipulate bytecodes directly.

MetaJava

MetaJava[Golm and Kleinőder 97] 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.

Chapter 3

Designing the OpenJava MOP

We designed a language named OpenJava, which is a dialect of the Java language and has an extensible mechanism, and implemented its proccessing system. The point of this research is the design of extension mechanism and inteface, which is called MOP (Metaobject Protocol). For the reason of efficient execution of application, our MOP is based on compile-time MOPs[Chiba 95, Ishikawa *et al.* 96], which has no overhead at runtime of application software.

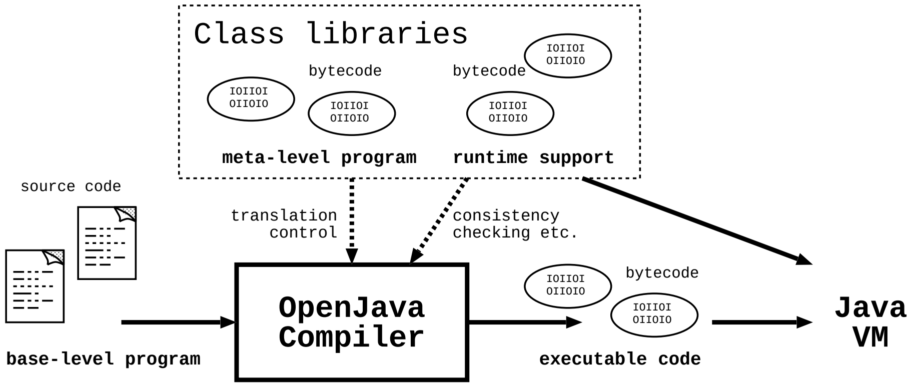

Figure 3.1: OpenJava Compiler Overview

For the end programmers who benefit from language extensions provided by meta programmers, the way of developping their application software is not different from the ordinal way. Similarly to other systems with compile-time MOPs, OpenJava compiler, `ojc`, accepts source programs written by programmers and generates executable code. The difference from regular Java compilers is that OpenJava compiler refers to metalevel libraries in addition to regular libraries. An overview of the OpenJava compiler processing is drawn in Figure 3.1. Moreover, the compiler generates bytecode for the Java VM.

Internally, our OpenJava compiler constructs two major modules, a translator and a regular Java compiler. The translator generates programs written in regular Java and then the compiler generates regular byte code for the Java VM according to the source code generated by the translator (Figure 3.2).

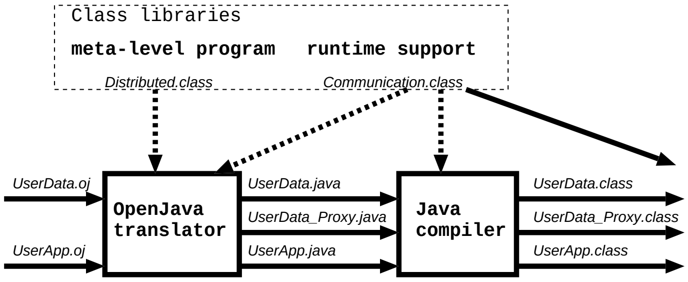

Figure 3.2: OpenJava Compiler Coarse Modules

As a research, the most important module of this system is the translator part. First, the translator takes source program written in the OpenJava language extended by the meta-level program specified in that source program and generates an AST (Abstract Syntax Tree). Then it transforms the AST according to the meta-level program. Finally, it generates source code in the regular Java language from the transformed AST. Figure 3.3 shows this flow.

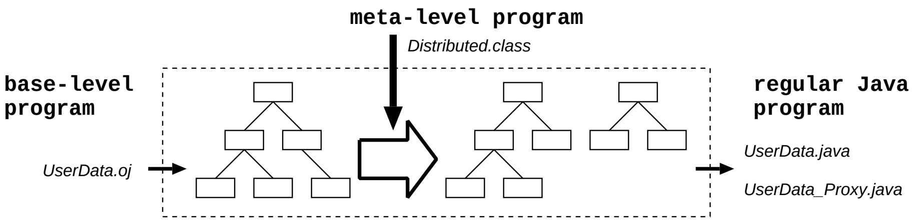

Figure 3.3: OpenJava Translator

The point of this research is how and what to provide a good interface for its meta-level programmers who write meta-level program in order to implement a new extended language feature. In the rest of this chapter, we discuss the important design issues of the OpenJava MOP. First, in section 3.1, why we choose a compile-time MOP instead of the runtime MOPs, is described. In section 3.2, how the system should control several extensions is described. Then, in section 3.3, we propose our new MOP after comparing with ordinal MOPs. In section 3.4, the API classes for our MOP are presented.

3.1 Use of Compile-time MOP

We choosed a compile-time MOP approach for implementing OpenJava because efficient execution of application software is our primary requirement. If we choose an approach of modifying the regular Java VM, the ability of language customization might be easily implemented. However, this implementation might make it difficult to apply runtime optimization techniques like just-in-time compiler, which are currently the mainstrem of optimization in Java. This is because such optimization techniques assume that programs should be statically given in general. But in runtime approach, methods and fields are dynamically altered, removed, and added by metalevel programs.

The OpenJava MOP is based on a compile-time MOP. Unlike runtime MOPs described in 2.2.1, the compile-time MOP lets meta programmers define how to compile given source program. In a runtime MOP approach, the execution performance is low if it is implemented simply. Thus, runtime MOPs must be implemented with a powerful partial evaluation mechanism like ABCL/R3[Masuhara *et al.* 95]. However, it's not easy to design and implement such an effective mechanism for languages like Java[Consel 93, Meyer, U. 91]. In fact, there seem to be no pretty result in partial evaluation techniques at compile-time for Java, for the present.

With our MOP, meta programmers handle how to translate source programs though there is still an alternative to lets programmers produce executable binary code for a source program. In OpenJava, programmers can handle only source code to source code translation. Even though MOPs handling how to generate executable code is very powerful to implement optimizer, especially in languages in which source code level optimization is relatively restricted like C++, such MOPs would be more complicated because of the interface to handle executable code. As for Java, the executable code is bytecode as intermediate language[Gosling 95] in which most of bytecodes simply correspond to source code; in fact, there's many disassembler for Java's bytecode. Thus, source code level optimization can be enough powerful in Java.

3.2 Scope of Translation

Controling the scope of translation is important. The 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 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.

3.2.1 Class-based Translation

OpenJava has a scope control mechanism with which translations are performed according to types of each object, that is, classes. An extension is defined as a metaclass and a class corresponds to an instance of a metaclass. In OpenJava, the default metaclass is provided as the class OJClass and OJClass is defined no to make any translation. Meta programmers define a new subclass of the class OJClass 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 very simple base level program in OpenJava.

Listing 3.1 *Hello.oj*

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

The notation:

```
class C instantiates M
```

specifys 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 Verbose 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 3.1 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++[Ishikawa *et al.* 96], system mixins in EPP[Ichisugi & Roudier 97] or pattern matching in A*[Ladd & Ramming 95]. Our scope controling method is founded upon OpenC++'s and it has been demonstrated to be very useful for many applications by Chiba[Chiba 95, Chiba 98].

3.2.2 Translation at Callee-side or 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 throuth its fields, methods or constructors of the class are performed.

To implement the example of methods which 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:

Listing 3.2 *Hello.java*

```
public class Hello instantiates Verbose
{
    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 Verbose, 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();
invoke_Hello_say( a );
```

using a function of Listing 3.3:

Listing 3.3 *invoke_Hello_say()*
```
String invoke_Hello_say( Hello obj ) {
    System.out.println( "say() is called." );
    String result = obj.say();
    System.out.println( "done." ) );
    return result;
}
```

Consequently, in order to change an object behavior by translating source program, there are two translation category of caller-side translation and callee-side translation. Some consideration from the point of this view are described below.

Advantages and Drawbacks

There are trade-offs between each translation thought the example above seems to be handled by both the translation at callee-side and the translation at caller-side. The difference in translating sides of source program makes

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[Gamma *et al.* 94]. Here, suppose a simple program providing this feature as follows:

Listing 3.4 *Flyweight*
```
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, henght );
    }
}
```

In this case, this program saves system memory and computation time by providing a method `genBitmapFont()` which recycles generated objects and by hidding 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 choosed by the active type of the object `obj` but not by the static type. This feature is a kind of method dispatch mechanisms usualy incorporated in object-oriented languages. 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 consitency 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 consitency for class (`static`) member accesses.

Consequently, to change 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.

Still Collision

The system with type-based scope controling works pretty better than the one without it because the system without scope controling is often inapplicable to several language extensions.

However the type-based system still has a possibility to cause a collision between two language extensions when it employ the two of translation policy, caller and callee side. This is because caller-side of translation for one type is always in callee-side of translation for another type.

The system with two side translation semantics must specify how that collision is solved.

3.2.3 Semantics-based Translation

With ordinal compile-time MOPs, it is not easy for meta programmers to write operations delivered in source code. For example, it is not easy to write a meta program to add a method of a certain name only in case that there is no such methods of the name. It is because of the design of ordinal MOPs; 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 MOP 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 parse tree directly.

Ordinal Compile-time MOPs

With compile-time MOPs, programmers 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 computer, programmers define a meta-level proglam 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 compile-time MOP, programmers would represent an algorithm of translation by transforming parse tree or AST while way of defining how to translate varies for each compile-time MOPs. Here suppose a language system with the simplest compile-time MOP. 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 another method `translate()` which returns parse tree representing desired behavior (Figure 3.4).

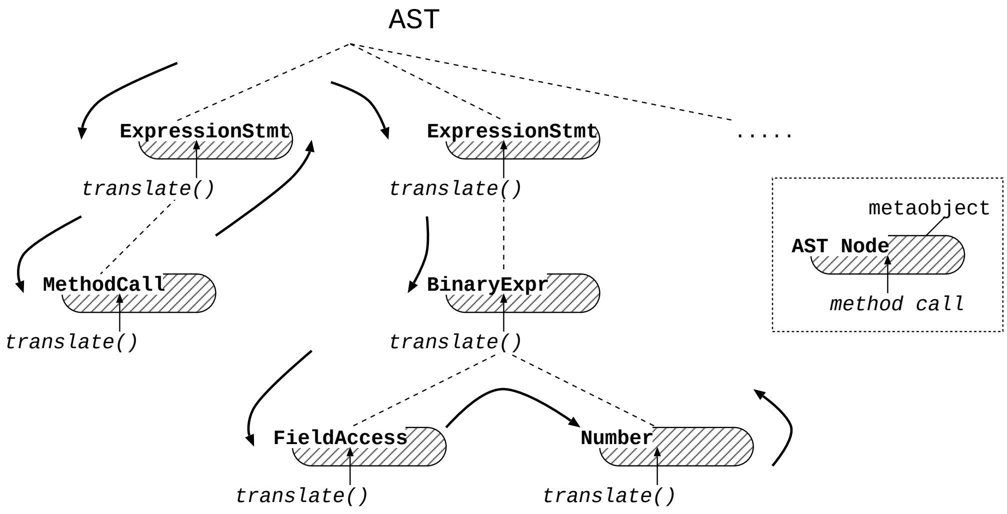

Figure 3.4: Translation with the Naive MOP

For instance, the above example can be implemented by defining a class RemoteMethodCall(Listing 3.6) substituting the regular class MethodCall(Listing 3.5).

Listing 3.5 *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 3.6 *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.

The OpenC++ MOP

As for the MOP in OpenC++ version 2, which has type-driven translation mechanism as described in subsection 3.2.1, it can handle the case above easily. Since the type of object `remote` differs from the type of the object `name`, the translation to be applied are easily switched appropriately for its type, that is, the translation for remote object can be only performed on the access to the object `remote`. Though the parts to be translated is scattered and spreads in the source program, class-based translation, which is semantical rather than syntactical, can be successfully applied to such applications.

However, this applicability is only about translation at caller-side described in subsection 3.2.2. As we claimed in subsection 3.2.2, translating the parts of source program at callee-side, which is class declaration part, is also important. Unfortunately, it falls into the problem similar to the problem of the naive MOPs above, in applying callee-side translation to the source program. In declarative language like Java, information related an object may not be available from the top to there, further more, in that file.

3.2.4 Problem of Ordinal MOPs

With the ordinal compile-time MOPs, it is not easy for meta programmers to write operations spreading in source code according to the information scattered in source code. For example, it is difficult to write a meta program to add a method of a certain name to a class only in case that there is no such methods of the name. This is because of the design of ordinal MOPs; The order to invoke each method `translate()` on node objects of parse tree is fixed in post-order or pre-order. As a result, it is difficult to translate a part of parse tree according to the information of another part of parse tree.

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 3.5).

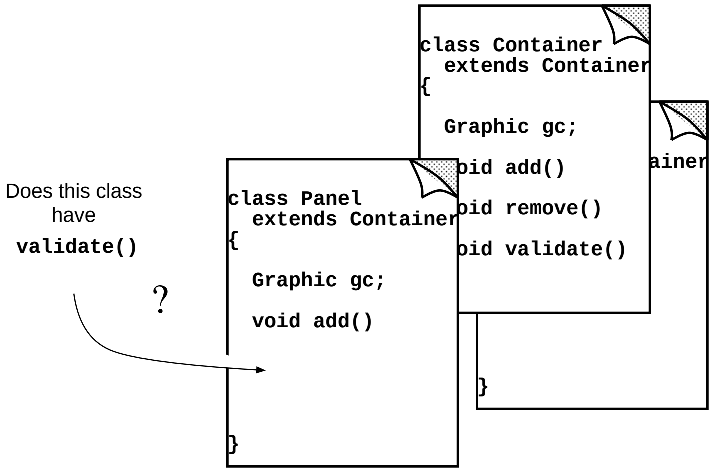

Figure 3.5: Scattered and Spreaded Information

Consequently, ordinal compile-time MOPs cannot handle examples shown above though they can handle application examples like what can be handled by Lisp macro and its advanced system. This is a reason why compile-time MOPs are not easy to use relatively to runtime MOPs.

3.3 The OpenJava MOP

The problem shown in subsection 3.2.4 points out that the model of parse tree for compile-time MOPs is not always addaptable to applications. In fact, the example in subsection 3.2.4, to add a method of a certain name only in case of no such methods of the name in that class, cannot be represented well in ordinal compile-time MOPs. But this kind of changing is very simple concept from the point of object-oriented systems and thus must be represented easily by meta programmers.

In OpenJava, we propose a new kind of compile-time MOP to address this problem. Its interface is very similar to that of runtime MOPs, with which it is claimed that programmers can intuitively describe extension of the language[Masuhara *et al.* 95]. In fact, except executional introspection for the source code under construction, the introspection mechanism found in Java Reflection API[JavaSoft 97a] is fully provided in OpenJava MOP, and the MOP naturally incorporates an intercession mechanism for translating the parts of class declarations in source program.

A unique feature is that our OpenJava translator internally creates a class metaobject for every base-level class in the processed source programs and the all refered bytecode classes. For instance, suppose the compiler is given two source programs which include class declarations of the classes VRect and Graph, and the classes refered to the classes String and VPoint anyway(Figure 3.6). The system creates each metaobject for them by each appropriate metaclass. Since the class VRect specified to be instantiated as Verbose metaobject, the metaobject for the class VRect is generated by the class Verbose. Also, since the class VPoint was compiled as an instance of a metaclass Verbose before, the metaobject for the class VPoint is generated as an instance of the metaclass Verbose.

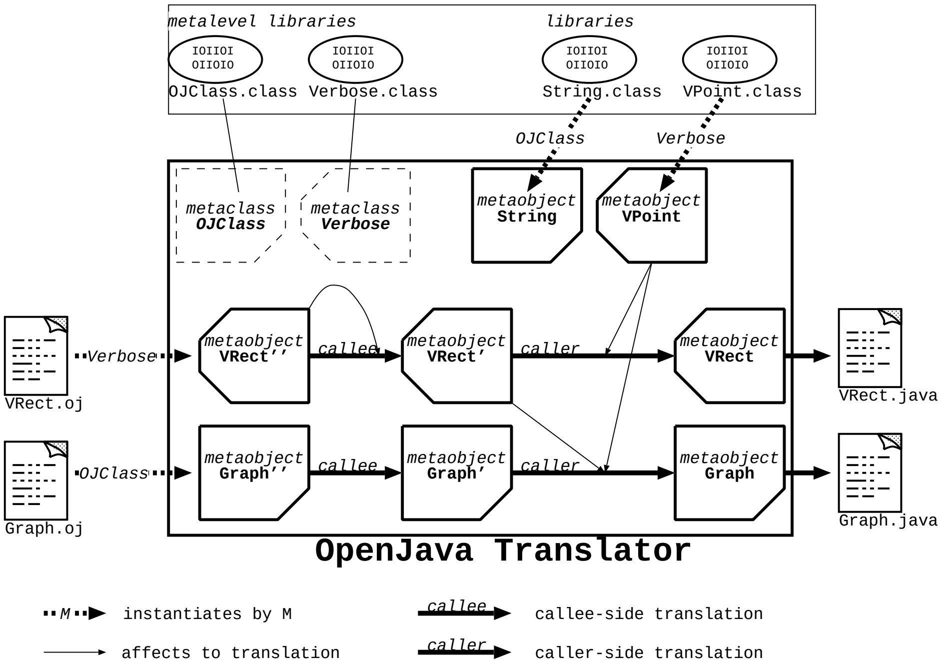

Figure 3.6: Class Metaobjects and Translation

First, callee-side translation is performed. The metaobject for the class VRect translates itself according to the definition in the metaclass Verbose, and so does the class Graph though it doesn't change because its metaclass is the default metaclass OJClass. Secondly, caller-side translation is performed. The regions where VPoint is used in the class VRect and the class Graph are translated by the metaobject for the class VPoint, which is an instance of the metaclass Verbose.

Finally, the translator generates regular Java codes from the translated metaobjects for VRect and Graph. And the system invokes regular Java compiler with the generated codes. Also the system generates the meta-level information including which metaclass instantiates the metaobject for the class VRect for the purpose of detecting the metaclass of the class in the future (Figure 3.7). This mechanism makes it possible to use libraries without their source programs.

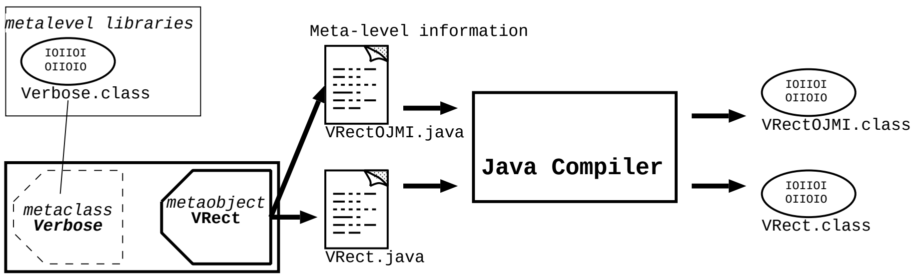

Figure 3.7: Meta-level Infomation

Syntax Extension

Syntax extension of languages is often useful to provide an appropriate interface to the newly extended language mechanisms. In OpenJava, the scope of syntax extension is also controled and restricted to the region of translation. The places are restricted where syntax extensions can be defined.

At callee-side, the allowed extensions for a class are as follows:

- Several modifiers consisting of a definitive word, for the member declarations, field, method, constructor and inner-class, and variable declarations in the class body
- Several suffixes consisting of a definitive word followed by context-free syntax, for the class declaration and the member declarations in the class body

For example, if a modifier `verbose` and a suffix which begins with `says` and consists of a following word are defined in a metaclass NoisyClass, the source program may be written as follows:

Listing 3.7 *The class NoisyObject of the metaclass NoisyClass*

```
class NoisyObject instantiates NoisyClass
{
    ...
    verbose String toString() {
        ...;
    }
    void addNotify() says Hello {
        ...;
    }
}
```

And at caller-side, the allowed extensions for a class are as follows:

- Suffixes consisting of a definitive token followed by context-free syntax, after the class name.

For example, if a suffix `< TYPENAME >`, where `TYPENAME` is a syntax representing a qualified class name, defined in a metaclass TemplateClass, a source program which use a class Vector whose metaclass is the TemplateClass may be written as follows:

Listing 3.8 *Using the class Vector of the metaclass TemplateClass*

```
class A
{
    Vector<String> strvec = null;
    void add( Vector<java.awt.Component> compvec ) {
        strvec = new Vector<java.lang.String>();
        ...
    }
}
```

Though they are still useful, syntax extensions are currently not so much powerful in OpenJava in order to avoid conflicting between extensions. This is future work.

3.4 OpenJava API

The most important part of OpenJava API is the default metaclass OJClass for class object. It provides methods to access information about the class. Additionally, the classes OJField, OJMethod, OJConstructor are important in OpenJava API to represent field, method, or constructor metaobjects which are available through the methods of OJClass.

3.4.1 Class

The class `openjava.mop.OJClass` represents a class object. Information about class is available through its methods.

First, programmers can use the `static` method `forName()` to get a class metaobject for its name (Table 3.1).

Table 3.1: static methods in OJClass

`public OJClass forName(String name)` Attepmpts to locate, load, and link a OJClass object representing the class of the given fully-qualified name. If it succeeds, returns that OJClass object.

The rest of methods in the class OJClass can be categorized into three:

1. to get information about the class (introspection)
2. to change information about the class (intercession)
3. to be overriden to translate (intercession)

Methods for Introspection of Class Declaration

The methods presented in this subsection would be used to get information about the class, that is, the name of class, its inheriting superclass, its implementing interfaces, its members, and so on. These methods correspond to the methods of the class `java.lang.Class` in Java Reflection API[JavaSoft 97a].

The methods in Table 3.2, Table 3.3 and Table 3.4 are used for introspection.

Methods for Modification of Class Declaration

The methods in Table 3.5 are used to modify the class only from the class metaobject itself. They cannot be used from other class metaobjects. Thus they are used in the overriden methods to translate the class declaration, callee-side.

Methods to Override

By overrideing the method in Table 3.6, programmers can perform their callee-side translation. And by overrideing the method in Table 3.7, programmers can perform their caller-side translation.

These methods are invoked by the OpenJava system at appropriate time of translation process.

Table 3.2: Methods in OJClass for primitive introspection (1)

`public OJModifier getModifiers()`
Returns a OJModifier object for the modifiers of this class object.

`public String getName()`
Returns the fully-qualified name of the class as a String.

`public String getPackage()`
Returns the package name of the class as a String.

`public boolean isInterface()`
Tests if this OJClass object represents an interface type.

`public boolean isArray()`
Tests if this OJClass object represents an array class.

`public boolean isPrimitive()`
Tests if this OJClass object represents a primitive type.

`public boolean isAssignableFrom( OJClass clazz )`
Determines if the class or interface can be represented by this class. Object is either the same as, or is a superclass or superinterface of, the class or interface represented by the given OJClass parameter.

Table 3.3: Methods in OJClass for primitive introspection (2)

`public OJClass getSuperclass()`
Returns the OJClass object which represents the superclass of the class.

`public OJClass[] getInterfaces()`
Returns an array of OJClass objects which represents the interfaces of the class. If this OJClass object represents an interface, returns an array containing objects representing the direct superinterfaces of the interface.

`public OJClass getDeclaringClass()`
Returns the OJClass object which represents the class declaring this class as an innerclass.

`public OJClass getComponentType()`
Returns the OJClass object representing the component type of an array. If this class does not represent an array class this method returns null.

Table 3.4: Methods in OJClass for primitive introspection (3)

`public OJClass[] getDeclaredClasses()`
Returns an array of all declared classes in this class, excluding inherited ones.

`public OJField[] getDeclaredFields()`
Returns an array of all declared fields in this class, excluding inherited ones.

`public OJMethod[] getDeclaredMethods()`
Returns an array of all declared methods in this class, excluding inherited ones.

`public OJConstructor[] getDeclaredConstructors()`
Returns an array of all declared constructors in this class, excluding inherited ones.

Table 3.5: Methods in OJClass for primitive intercession

`protected OJMethod addMethod( OJMethod method )`
Generates an OJMethod object with the same signature of the given OJMethod object and add it as a declared method of this class. This returns the generated OJMethod object.

`protected OJMethod removeMethod( Signature signature )`
Removes an method with the given signature from this class declaration. Returns the OJClass object which was removed or null if there was no correspongin method in this class declaration.

`protected OJMethod replaceMethod( Signature signature, OJMethod replacement )`
Replace an method of the given signature in this class declaration, with the given OJMethod object. Returns the OJClass object which was replaced or null if there was no correspongin method in this class declaration.

Table 3.6: Overridable Methods in OJClass for Callee-side Translation

`public void translateDeclaration( Environment env )`
To be overrided for translation the class declaration. This translation is applied before caller-side translation.

Table 3.7: Overridable Methods in OJClass for Caller-side Translation

`public Expression expandAllocation( AllocationExpression expr, Environment env )`
This returns the generated OJMethod object.

`public Expression expandArrayAllocation( AllocationExpression expr, Environment env )`
This returns the generated OJMethod object.

`public Expression expandVariableDeclaration( VariableDeclaration stmt, Environment env )`
This returns the generated OJMethod object.

`public Expression expandFieldRead( FieldAccess expr, Environment env )`
This returns the generated OJMethod object.

`public Expression expandFieldWrite( FieldAccess expr, Environment env )`
Generates an OJMethod object with the same signature of the given OJMethod object and add it as a declared method of this class.

`public Expression expandMethodCall( MethodCall expr, Environment env )`
Generates an OJMethod object with the same signature of the given OJMethod object and add it as a declared method of this class.

`public Expression expandExpression( Expression expr, Environment env )`
Generates an OJMethod object with the same signature of the given OJMethod object and add it as a declared method of this class.

3.4.2 Accessibility of Class Information

Like many other object-oriented languages, Java has a system for controling accessibily to members in each class. This kind of accessibily control is an important feature to maintain application software development. Programmers can define an accessibility of four levels for each member declaration by omitting or adding one of modifier notations, `private`, `protected` or `public`. Table 3.8 specifies each case of accessibility.

from which situation of class	public	protected	default	private
class itsself	OK	OK	OK	OK
class in same package	OK	OK	OK	NG
class in another package	OK	NG	NG	NG
subclass in same package	OK	OK	OK	NG
subclass in another package	OK	OK	NG	NG

Table 3.8: Accessibilities of Member by Modifier

Programmers can pick out only the methods which is available from the specified class by using methods in Table 3.9.

Table 3.9: Methods in OJClass for practical introspection

`public OJMethod[] getAllMethods()`
Returns an array of the all methods of this class, including all the inherited methods.

`public OJMethod[] getNonPrivateMethods()`
Returns `public`, `protected` or `package` methods, including the inherited `public`, `protected` or `package` methods. These methods are possibility to be accessed from other classes.

`public OJMethod[] getMethods( OJClass situation )`
Returns an array of the all methods of this class, including all the inherited methods.

`public OJMethod[] getInheritableMethods()`
Returns `public` or `protected` methods, including the inherited `public` or `protected` methods. Returned methods inheritable for any subclass.

`public OJMethod[] getInheritableMethods( OJClass situation )`
In addition to the methods inheritable for any subclass, this returns the methods with default `package` accessibility if the methods are declared in the package of the given situation of subclass.

3.4.3 Members

The class OJClass may return metaobjects for fields, methods, constructors, classes. They respectively corresponds to the class OJField, OJMethod, OJConstructor and OJClass. The class OJClass is already presented in this section. Since it is the class OJMethod which is the most complicated class for class member metaobjects except the class OJClass, we present only the class OJMethod here.

Table 3.10: Methods in OJMethod for primitive introspection (1)

`public OJModifier getModifiers()`
Returns a OJModifier object for the modifiers of this method object.

`public Signature signature()`
Returns a Signature object for the signature of this method object.

`public String getName()`
Returns the fully-qualified name of this method object as a String.

Table 3.11: Methods in OJMethod for primitive introspection (2)

`public OJClass[] getDeclaringClass()`
Returns the OJClass object which represents the class declaring this method object as a member.

`public String getReturnType()`
Returns the OJClass object which represents the type returned by this method object.

`public String getParameterTypes()`
Returns an array of OJClass objects which represents the parameter types which this method object accepts.

`public String getExceptionTypes()`
Returns an array of OJClass objects which represents the exception types which this method object throws.

Table 3.12: Methods in OJMethod for primitive intercession

`public void setReturnType( OJClass returnType )`
Sets the OJClass object which represents the type returned by this method object.

`public void setExceptionTypes( OJClass[] exceptionTypes )`
Sets the array of OJClass objects which represents the exception types which this method object throws.

Table 3.13: Methods in OJMethod for low-level intercession

`public Variable[] getParameters()`
Returns an array of the Variable objects which represent the parameter variables of this method object.

`public StatementList getBody()`
Returns the reconstructable list of Statement objects which represents the method body.

`public StatementList setBody( StatementList stmts )`
Sets the method body.

Chapter 4

Implementation

4.1 Class Diagram

Here construction of the major classes implementing the OpenJava MOP is briefly shown.

First, the classes OJClass, OJField, OJMethod and OJConstructor implement the interface OJMember (Figure 4.1) since classes, fields, methods and constructors may be members of a class.

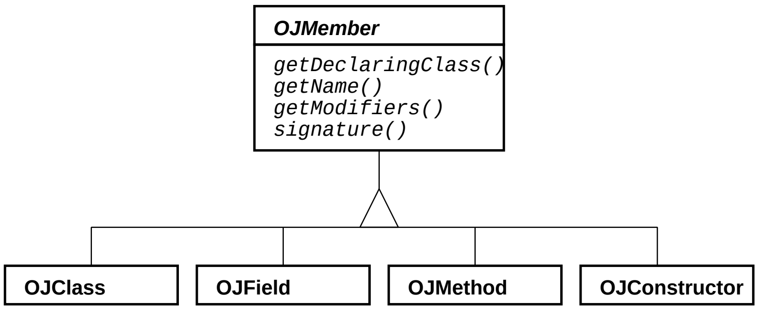

Figure 4.1: OJMember

While metaobject may represent a class defined in bytecode or a class defined in source code, the system gives unified interface to the meta programmers. To achieve this, OJClass internally has one of two states, bytecode or source code (Figure 4.2). And it is same for the classes OJField, OJMethod and OJConstructor (Figure 4.3, 4.4, 4.5).

4.2 Parse Tree

The classes for AST (Abstract Syntax Trees) are not so much unique comparatively to other systems. In order to avoid programmers access or con-

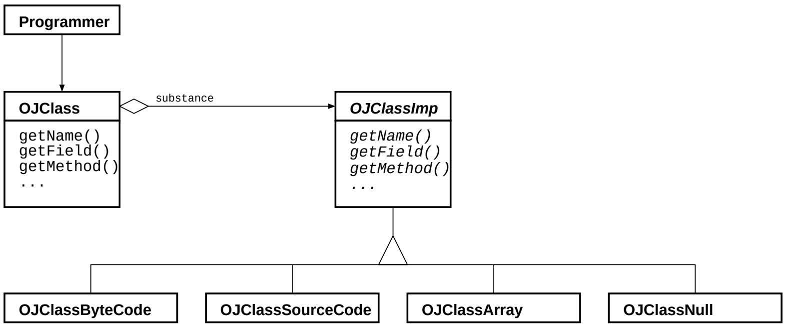

Figure 4.2: Implementation of OJClass

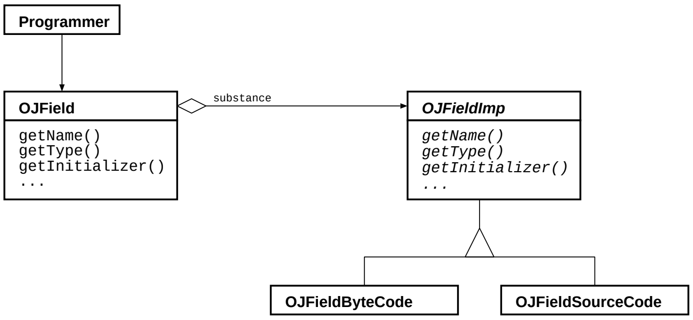

Figure 4.3: Implementation of OJField

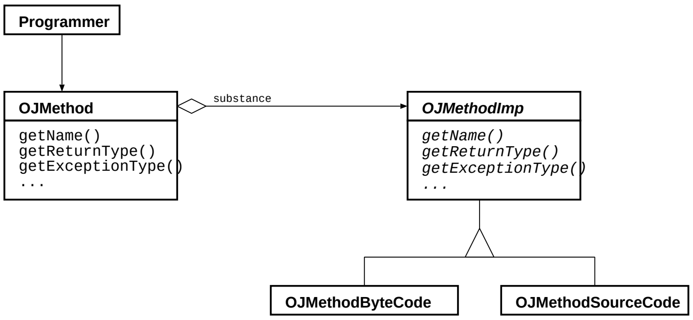

Figure 4.4: Implementation of OJMethod

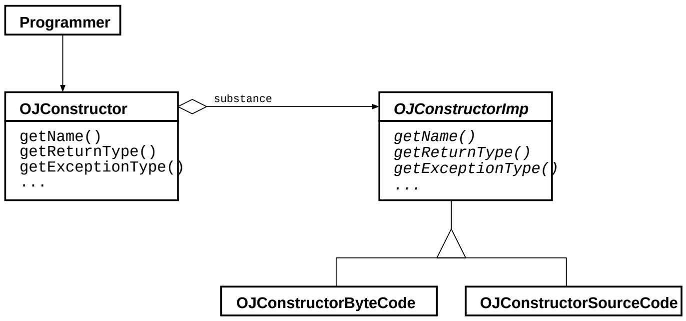

Figure 4.5: Implementation of OJConstructor

struct wrong trees, each node of AST is strongly typed in OpenJava.

Table 4.1 is a list of classes which represent statement part of AST. And Table 4.2 shows classes which represent expression part of AST.

Table 4.1: A List of Classes for statments

`public interface Statement` An interface as a statement for every class representing a statement.
`public class VariableDeclaration` Represents a variable declaration.
`public class Block` Represents a block with a pair of braces.
`public class EmptyStatement` Represents an empty statement only with semi colon.
`public class ExpressionStatement` Represents a statement with expression.
`public class DoWhileStatement` Represents a `do - while` statement.
`public class ForStatement` Represents a `for` statement.
`public class WhileStatement` Represents a `while` statement.
`public class IfStatement` Represents a `if` statement.
`public class SwitchStatement` Represents a `switch` statement.
`public class SynchronizedStatement` Represents a `synchronized` statement.
`public class TryStatement` Represents a `try` statement.
`public class LabeledStatement` Represents a labeled statement.
`public class BreakStatement` Represents a `break` statement.
`public class ContinueStatement` Represents a `continue` statement.
`public class ThrowStatement` Represents a `throw` statement.
`public class ReturnStatement` Represents a `return` statement.

Table 4.2: Expressions

`public interface Expression`
An interface as an expression for every class representing an expression.

`public class AllocationExpression`
Represents an allocation of an instance with `new`.

`public class ArrayAllocationExpression`
Represents an allocation of an array with `new`.

`public class AssignmentExpression`
Represents an assignment expression.

`public class BinaryExpression`
Represents a binary operational expression.

`public class CastExpression`
Represents a type casting expression.

`public class ConditionalExpression`
Represents a conditional expression like `a ? b : c`.

`public class InstanceofExpression`
Represents a type testing operation.

`public class UnaryExpression`
Represents an unary operational expression.

`public class ArrayAccess`
Represents an array access.

`public class FieldAccess`
Represents a field access.

`public class MethodCall`
Represents a mechod call.

`public class SelfAccess`
Represents an access for self object by `this` including `super`.

`public class Variable`
Represents a variable.

`public class Literal`
Represents a literal including `null`.

Chapter 5

Application Examples

Here we show actual practices with our OpenJava.

5.1 Design Patterns

We claims that compile-time MOPs (Meta-Object Protocols) provide a general framework for implementing those syntax extensions and extended language constructs. In our approach, programmers write a meta-level library which implements syntax extensions and extended language constructs for a design pattern. With these extensions, users of that library can explicitly declare in their programs what design patterns are used and what is the role of each class in that design pattern. Furthermore, they do not have to describe trivial behavior of objects because it is automatically generated by a MOP system according to that library. Thus their programs can be simple and easy to understand. To examine our idea, we have written libraries for design patterns in OpenJava.

As our basic concept, we apply a metaclass to a design pattern.

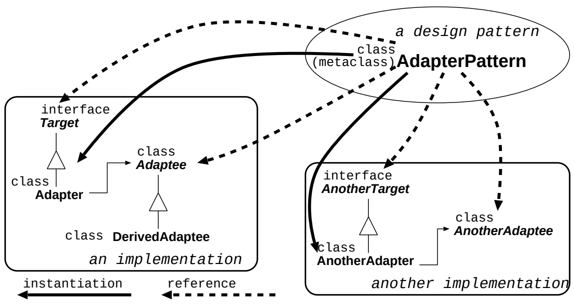

Figure 5.1: A metaclass as a design pattern

5.1.1 The Adapter Pattern

From the view point of the Adapter pattern, programmers have only to define which method of the class Vector corresponds to each methods of the interface Stack. This is because the Adapter only maps methods of the Target to methods of the Adaptee.

In an extended language supporting the Adapter pattern, programmer should be able to directly handle such *forwardings*. An implementation in that extended language should be as follows:

Listing 5.1 *VectorStack.oj*

```
public class VectorStack
    instantiates AdapterPattern
    adapts Vector in v to Stack
{
    Object peek() forwards lastElement;
    void push(Object o) forwards addElement;
    Object pop() { ....  }
}
```

To use the Adapter pattern, all that programmers have to do are:

1. to declare that the program uses the Adapter pattern
2. to declare an Adapter class
3. to specify an Adaptee class and a Target class
4. to specify mapping between methods of the Adapter class and methods of the Adaptee class

These things are easily described with the following language constructs:

1. A declaration
   `instantiates` P
   specifies that a design pattern P is used.
2. A declaration
   `adapts` A `in` F `to` T
   specifies that this class is an Adapter adapting the object of class A in the field F to the interface T.
3. A declaration at the end of a method signature
   `forwards` M
   specifies that the method forwards to the method M of the Adaptee.
4. An Adapter class have methods corresponding to all the methods of the Adaptee class. Even if they are not explicitly defined, they are automatically inserted in the Adapter class. They forward to the Adaptee's method with the same name and signature.

These lanugage constructs simplify programs written according to the Adapter pattern.

Implementation in OpenJava

In OpenJava, the language extension shown above is implemented by a metaclass AdapterPattern. Once this metaclass is written, other programmers can benefit from the metaclass and thereby write programs involving extended language constructs. The definition of the metaclass is as follows:

Listing 5.2 *AdapterPattern.oj*

```
public class AdapterPattern extends OJClass
{
    /* overrides for syntax extensions */
    static void init() {
        registerDeclarationSuffix( "adapts", ..  );
        registerMethodSuffix( "forwards", ..  );
    }

    /* overrides for translation */
    void translateDeclaration( Environment env )
        throws MOPException
    {
        ParseTree sfx = this.getSuffix( "adapts" );
        OJClass adaptee = OJClass.forName( ..sfx..  );
        OJClass target = OJClass.forName( ..sfx..  );

        /* implicit forwarding to same signature */
        OJMethod[] adapteds
            = adaptee.getDeclaringMethods();
        for (int i = 0; i < adapteds.length; ++i) {
            /* picks the method of same signature */
            OJMethod same_sign
                = target.lookupMethod( adapteds[i] );
            if (same_sign != null) {
                OJMethod imp_forwarder = ..same_sign..;
                this.addMethod( imp_forwarder );
            }
        }

        /* explicit forwarding */
        OJMethod[] forwarder
            = this.getSuffixedMethods( "forwards" );
        for (int i = 0; i < forwarder.length; ++i) {
            /* make forwardings */;
            forwarder[i].setBody( ..  );
        }

        /* adds a field to hold an Adaptee */
        this.addField( .."adaptee"..  );

        /* adds a constructor to accept an Adaptee */
        this.addConstructor( ..  );

        /* adds an interface as a Target */
        this.addInterface( target );
```

```
    }
}
```

According to this metaclass, the OpenJava system produces regular Java source code equivalent to the code in listing 2.3 from the code in listing 5.1.

5.1.2 The Visitor pattern

In this section, we show another example using the Visitor pattern. The Visitor pattern is used to *represent an operation to be performed on the elements of an objet structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates* [Gamma *et al.* 94]. Figure 5.2 shows a structure of the Visitor pattern.

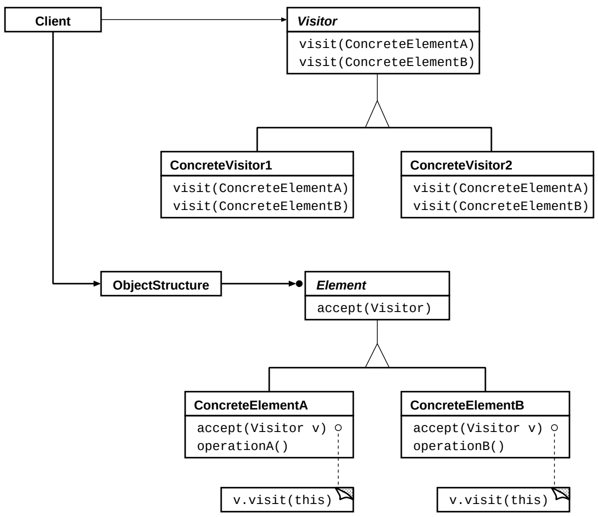

Figure 5.2: A structure of the Visitor pattern

Ordinal Implementation

Suppose that a programmer should implement a GUI class library according to the Visitor pattern. The Visitor and the Element in Figure 5.2 are represented by an interface GUIVisitor and an interface GUIElement:

Listing 5.3 *GUIElement.java*

```
public interface GUIElement
{
    void accept(GUIVisitor v);
}
```

Listing 5.4 *GUIVisitor.java*

```
public interface GUIVisitor
```

```
{
    void visit(Container e);
    void visit(Panel e);
    void visit(Label e);
}
```

Since all the elements must accept a visitor, a class for elements needs to have a method `accept()` which invokes `visit()` on the given GUIVisitor object. The following is the definition of an element class Panel:

Listing 5.5 *Panel.java*
```
public class Panel extends Container
    implements GUIElement
{
    void accept(GUIVisitor v) { v.visit( this ); }
    ....
}
```

Even if this class inherits from another element class Container which has a method `accept()`, it has to have their own version of `accept()`. Otherwise, the method `visit()` for the superclass would be wrongly invoked.

Writing a class implementing the interface GUIVisitor is also tedious:

Listing 5.6 *PartsCounter.java*
```
public class PartsCounter implements GUIVisitor
{
    void visit(Container e) { ....  }
    void visit(Panel e) { visit( (Container) e ); }
    void visit(Label e) { ....  }
}
```

It has to have distinct methods `visit()` for every class implementing GUIElement. In this example, the programmer has to write a method for Panel even though this method simply calls `visit()` for its superclass Container.

Extensions

If the above extension for the visitor pattern is available, programmers can make the classes Panel and PartsCounter simpler.

First, programmers can write interfaces GUIElement and GUIVisitor with explicit declaration representing the use of the Visitor pattern:

Listing 5.7 *GUIElement.oj*
```
public interface GUIElement
    instantiates VisitorPattern
    accepts GUIVisitor
{
    void accept(GUIVisitor v);
}
```

Listing 5.8 *GUIVisitor.oj*
```
public interface GUIVisitor
    instantiates VisitorPattern
    visits GUIElement
{
    void visit() on Container, Panel, Label;
}
```

Then, they do not have to write a method `accept()` anymore for every class implementing the interface GUIElement. A class Panel, for example, is now rewritten as follows:

Listing 5.9 *Panel.oj*
```
public class Panel instantiates VisitorPattern
    extends Container
    accepts as GUIElement
{
    ....
    //accept() is implicitly defined.
}
```

Also, they can simplify the definition of the class PartsCounter. They do not have to define a method `visit()` for the class Panel because `visit()` for the superclass Container is reused for Panel. Thus the class PartsCounter is written as the following:

Listing 5.10 *PartsCounter.oj*
```
public class PartsCounter
    instantiates VisitorPattern
    visits as GUIVisitor
{
    void visit(Container e) { ....  }
    void visit(Label e) { ....  }
}
```

Implementation in OpenJava

In OpenJava, the extension for the Visitor pattern is implemented by a metaclass VisitorPattern:

Listing 5.11 *VisitorPattern.oj*
```
public class VisitorPattern extends OJClass
{
    static void init() {
        registerDeclarationSuffix( "visits", ..  );
        registerDeclarationSuffix( "accepts", ..  );
        registerMethodSuffix( "on", ..  );
    }

    void translate() throws MOPException {
        if ( this.isInterface() ) {
            /* translation as an interface */
```

```
                OJClass visitee
                    = ..  this.getSuffix( "visits" ) ..;
                OJClass acceptee
                    = ..  this.getSuffix( "accepts" ) ..;
                if (visitee != null)
                    translateVisitor( visitee );
                if (acceptee != null)
                    translateElement( acceptee );
            } else {
                /* translation for a concrete class */
                OJClass visitor
                    = ..  this.getSuffix( "visits" ) ..;
                OJClass element
                    = ..  this.getSuffix( "accepts" ) ..;
                if (visitor != null)
                    translateVisitorAs( visitor );
                if (element != null)
                    translateElementAs( element );
            }
        }

        void translateVisitor( OJClass visitee ) { ....  }
        void translateElement( OJClass acceptee ) { ....  }

        /* translation for a concrete Element */
        void translateElementAs( OJClass element ) {
            OJSignature visit = element.getMethods()[0];
            this.addMethod( ..visit..  );
            this.addInterface( element );
        }

        /* translation for a concrete Visitor */
        void translateVisitorAs( OJClass visitor ) {
            OJClass element
                = ..visitor.getSuffix( "visits" )..;
            OJMethod[] visit = visitor.getMethods();
            for (int i = 0; i < visit.length; ++i) {
                if (this.lookupMethod( visit[i] ) == null) {
                    //implicit forwarding
                    OJMethod forwarder = ..visit[i]..;
                    this.addMethod( forwarder );
                }
            }
            this.addInterface( visitor );
        }
    }
```

According to this metaclass, the OpenJava system produces regular Java source code equivalent to the code in listing 5.3, 5.4, 5.5 and 5.6 from the code in listing 5.7, 5.8, 5.9 and 5.10.

5.2 Distributed Objects

In this section, we present a language extension providing a transparent distributed object programming. We assume there are already a class Client and a class Server which communicates each other through the network since it is beyond this thesis to present how to communicate through networks.

Suppose that a source program defined by programmers as follows:

Listing 5.12 *InfoCollector.java*

```
public class InfoCollector
{
    InfoCollector() {
        super();
    }

    String[] gatherInfo( File path ) {
        //access to localhost and
        //gather only the information available here
    }
}
```

To use objects of the class InfoCollector as remote objects, they must define a class like in listing 5.13. And it can be use as in listing 5.14.

Listing 5.13 *InfoCollectorProxy.java*

```
public class InfoCollectorProxy
{
    Client client;
    InfoCollectorProxy( URL url ) {
        client = new Client( url );
        client.invokeConstructor( "InfoCollector",
                                  new Object[0] );
    }

    String[] gatherInfo( File path ) {
        Object result = client.invoke( "gatherInfo",
                                       new Object[] { path } );
        return (String[]) result;
    }
}
```

Listing 5.14 *Using InfoCollectorProxy*

```
    InfoCollector collector
        = new InfoCollectorProxy( "www.apple.com" );
    String[] urls = collector.gatherInfo( "/index.html" );
```

By the program in listing 5.14, the InfoCollector object `collector` generate a Client object, then the Client object communicate with the remote Server object to generate a InfoCollector on the remote site. And each invocation on the object `collector` is sended to that remote object. The overview of such communication is shown in Figure 5.3.

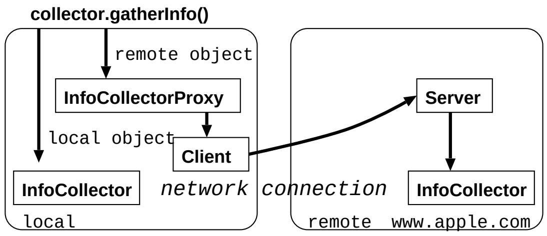

Figure 5.3: Proxy and Server

Extensions

In order to make remote invocations transparent and avoid annoying programming of the class InfoCollectorProxy for end-programmers, the system should automatically generate the class InfoCollectorProxy for the class InfoCollector defined by programmers as follows:

Listing 5.15 *InfoCollector.oj*

```
public class InfoCollector instantiates Distribution
{
    InfoCollector() {
        super();
    }

    String[] gatherInfo( File path ) {
        //access to localhost and
        //gather only the information available here
    }
}
```

And the usage of it should be as follows:

Listing 5.16 *Using Transparent InfoCollector*

```
    InfoCollector collector
        = new InfoCollector() on "www.apple.com";
    String[] urls = collector.gatherInfo( "/index.html" );
```

Implementation in OpenJava

In OpenJava, the extension for distributed object programming is implemented by a metaclass Distribution:

Listing 5.17 *Distribution.oj*

```
public class AdapterPattern extends OJClass
{
    /* overrides for syntax extensions */
    static void init() {
        registerTypeSuffix( "on", ..  );
    }
```

```
        /* overrides for translation */
        void translateDeclaration( Environment env )
            throws MOPException
        {
            /* implicit forwarding to same signature */
            OJClass proxy = generateProxy();
            OJSystem.env.addClass( proxy );     }

        /* overrides for translation at caller-side */
        void expandAllocation( AllocationExpression expr,
                               Environment env )
            throws MOPException
        {
            Literal site = (Literal) expr.getSuffix( "on" );
            if (site != null) {
                expr.setArguments( new ExpressionList( site ) );
            }
            return expr
        }

        /* generates proxy */
        OJClass generateProxy() {
            ...
        }
    }
```

According to this metaclass, the OpenJava system produces regular Java source code equivalent to the code in listing 5.12 and 5.13 from the code in listing 5.15. And The code fragments in listing 5.16 are translated into the code fragments in listing 5.14.

Chapter 6

Conclusion

This thesis has discussed the OpenJava MOP, our new language mechanism giving extensibility to the Java language. Through this MOP, programmers can intuitively write language extension libraries, which help programming in many kinds of application domains. This mechanism supplements a drawback of Java, which is flexibility that a good programming language should have.

The Java language still lacks some userful mechanisms needed by some kinds of applications though Java is a well-designed object-oriented language; it is simple and highly abstracted so that it is easy for programmers to learn and its runtime environment gives flexibility to Java programs and thereby application software written in Java is adaptable to various environment.

To address this problem, this thesis proposed an advanced macro processor based on the technique called compile-time reflection. Though traditional compile-time reflection systems have difficulties in writing meta-level programs for typical macro processing in object-oriented programming, we developed a new compile-time reflection system for OpenJava. With OpenJava, translation of source programs is indirectly performed through an abstract data structure called metaobjects. This data structure gives meta programmers an intuitive view of the source programs in object orientation. Meta programmers in OpenJava can describe extensions of the Java language more intuitively and safely than in traditional reflective systems.

This thesis presented an example extending the Java language to have higher-level control/data abstractions for design patterns. If design patterns are used without this language supports, some programs are significantly complicated so that the overall structure of the programs is not easy to understand. Also, the programmers have to write annoying and error-prone codes because the concept of design patterns are not directly supported by the Java language. Furthermore, this thesis presented an example of extending the Java language for supporting distributed computing. Meta programmers in OpenJava can describe extensions for distributed computing easily so that they can use the most suitable language constructs for their software.

Bibliography

[Bal 89] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum : Programming Languages for Distributed Computing Systems, *ACM Computing Surveys Vol.2, No.3*, pp.261-322, 1989.

[Bosch 96] Jan Bosch : Language Support for Design Patterns, In *TOOLS Europe '96*, 1996.

[Bosch 97] Jan Bosch : Design Patterns as Language Constructs, In *Journal of Object Oriented Programming*, SIGS Publications, 1997.

[Bracha *et al.* 98] G. Bracha, M. Odersky, D. Stoutamire and P. Wadler : Making the future safe for the past: Adding Genericity to the Java Programming Language, In *Proceedings of OOPSLA '98, ACM SIGPLAN Notices Vol.33, No.10*, pp.183-200, 1998.

[Briot & Cointe] J. P. Briot and P. Cointe : Programming with Explicit Metaclasses in Smalltalk-80, In *Proceedings of OOPSLA '89, ACM SIGPLAN Notices Vol.24, No.10*, pp.419-431, ACM, 1989.

[Cartwright & Steel 98] R. Cartwright and Guy L., Steele Jr. Compatible Genericity with Run-time Types for the Java Programming Language, In *Proceedings of OOPSLA '98, ACM SIGPLAN Notices Vol.33, No.10*, pp.201-215, 1998.

[Chiba 95] Shigeru Chiba : A Metaobject Protocol for C++, In *Proceedings of OOPSLA '95, ACM SIGPLAN Notices Vol.30, No.10*, pp.285-299, 1995.

[Chiba 96] Shigeru Chiba, Gregor Kiczales, and John Lamping : Avoiding Confusion in Metacircularity: The Meta-Helix, in *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS)*, pp.157-172, Springer, 1996.

[Chiba 98] Shigeru Chiba : Macro Processing in Object-Oriented Languages, In *Proceedings of TOOLS Pacific '98 Technology of Object-Oriented Languages and Systems*, IEEE Press, 1998.

[Chiba & Tatsubori 98] S. Chiba, and M. Tatsubori : Yet Another `java.lang.Class`, *ECOOP'98 Workshop on Reflective Object-Oriented Programming Systems*, 1998.

[Cointe 87] Pierre Cointe : Metaclasses are First Class : the ObjVlisp Model, In *Proceedings of OOPSLA'87, ACM SIGPLAN Notices Vol 22, No.12*, pp.156-162, 1987.

[Consel 93] C. Consel and O. Danvy : Tutorial Notes on Partial Evaluation, In *Proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.1-9, 1993.

[Ducasse 97] Stéphane Ducasse : Message Passing Abstractions as Elementary Bricks for Design Pattern Implementation, In *Object-Oriented Technology, ECOOP workshop Reader*, LNCS 1357, 1997.

[Futamura 82] Yoshihiko Futamura : Partial Computation of Programs, In *Proceedings of RIMS Symposia on Software Science and Engineering*, pp.1-35, LNCS 247, 1982.

[Gamma *et al.* 94] E. Gamma, R. Helm, R. Johnson, and J.O. Vlissides : *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Gil & Lorenz 98] J. Gil and D. H. Lorenz : Design Patterns and Language Design, *IEEE Computer Vol.31, No.3*, pp.118-120, 1998.

[Golm and Kleinöder 97] M. Golm and J. Kleinöder : MetaJava - A Platform for Adaptable Operating-System Mechanisms, In *Proceedings of the ECOOP '97 Workshop on Object-Orientation and Operationg Systems*, 1997

[Gosling 95] James Gosling : Oak Intermediate Bytecodes, In *ACM SIGPLAN IR '95 Workshop on Intermediate Representations*, 1995.

[Gosling *et al.* 97] J. Gosling, B. Joy, and G. Steele : *The Java Language Specification*, Addison-Wesley, 1997.

[Hirano 97] Satoshi Hirano : HORB : Distributed Execution of Java Programs, In *Proceedings of WWCA '97*, 1997.

[Ishikawa *et al.* 96] Y. Ishikawa, A. Hori, M.Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, and K. Kubota : Design and Implementation of Metalevel Architecture in C++ - MPC++ Approach -, In *Proceedings of Reflection 96*, pp.153-166, 1996.

[JavaSoft 97a] JavaSoft : *Java(TM) Core Reflection API and Specification*, Sun Microsystems, 1997.

[JavaSoft 97b] JavaSoft : *Java(TM) Remote Method Invocation Specification*, Sun Microsystems, Inc., 1997.

[Kiczales 94] G. Kiczales, J. des Rivières and D. G. Bobrow : *The Art of the Metaobject Protocol*, The MIT Press, 1991.

[Kirby *et al.* 98] G. Kierby, R. Morrison and D. Stemple : Linguistic Reflection in Java, *Software Practice and Experience Vol.28, No.10*, pp.1045-1077, John Wiley & Sons, 1998.

[Kramer 97] Doug Kramer : *JDK 1.1 Documentation*, Sun Microsystems, 1997.

[Ladd & Ramming 95] D. A. Ladd and J. C. Ramming : A* : A Language for Implementing Language Processors, *IEEE Trans. on Software Engineering Vol.21, No.11*, pp.894-901, 1995.

[Liang & Gilad 98] S. Liang and G. Bracha : Dynamic Class Loading in the Java Virtual Machine, In *Proceedings of OOPSLA'98, ACM SIGPLAN Notices Vol.33, No.10*, pp.36-47, 1998

[Lindholm & Yellin 97] T. Lindholm and F. Yellin : *The Java Virtual Machine Specification*, Addison Wesley, 1997.

[Maddox 89] William Maddox : Semantically-sensitive macroprocessing, *Master's thesis (ucb/csd 89/545)*, University of California, 1989.

[Masuhara *et al.* 95] H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa : Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Language Using Partial Evaluation, In *Proceedings of OOPSLA'95, ACM SIGPLAN Notices Vol.30, No.10*, pp.300-315, 1995.

[Meijler *et al.* 97] T.D. Meijler, S. Demeyer and R. Engel : Making Design Patterns Explicit in FACE, a Framework Adaptive Composition Environment, In *Proceedings of ESEC/FSE '97, Springer-Verlag, pp. 94-110*, 1997.

[Meyer & Downing 97] J. Meyer and T. Downing : *Java Virtual Machine*, O'Reilly, 1997.

[Meyer, U. 91] Uwe Meyer : Tecniques for partial evaluation of imperative programs, In *Proceedings of PEPM '91 the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, SIGPLAN Notices Vol.26, No 9*, pp.94-105, 1991.

[Mulet *et al.* 95] P. Mulet, J. Malenfant, and P. Cointe : Towards a methodology for explicit composition of metaobjects, In *Proceedings of OOPSLA'95, ACM SIGPLAN Notices Vol.30, No.10*, pp.316-330, 1995.

[Schappert *et al.* 95] A. Schappert, P. Sommerlad and W. Pree : Automated Support for Software Development with Frameworks, In *Proceedings of SSR '95 ACM SIGSOFT Symposium on Software Reusability*, pp.123-127, 1995.

[Smith 84] Brian C. Smith : Reflection and semantics in lisp, In *Proceedings of POPL '84 ACM Symposium on Principles of Programming Languages*, pp.23-35, 1984.

[Soukup 95] Jiri Soukup : Implementing patterns, In *Patterns Languages of Program Design*, pp.395-412, Addison-Wesley, 1995.

[Tatsubori & Chiba 98] M. Tatsubori and S. Chiba : Programming Support of Design Patterns with Compile-time Reflection, In *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998.

[Ichisugi & Roudier 97] Y. Ichisugi and Y. Roudier : Extensible Java Preprocessor Kit and Tiny Data-Parallel Java, In *ISCOPE'97*, 1997.

Appendix A

OpenJava Command Reference

NAME

ojc - the OpenJava compiler

SYNOPSIS

`ojc [ options ] [ sourcefiles ]`

Arguments maybe in any order.

options
Command-line options

sourcefiles
One or more source files to be compiled. (Such as Hello.oj)

DESCRIPTION

The `ojc` tool reads class and interface definitions, written in the OpenJava programming language, and compiles them into bytecode class files.

Source code file names must have .oj suffixes, class file names must have .class suffixes, and both source and class files must have root names that identify the class. For example, a class called MyClass would be written in a source file called MyClass.oj and compiled into a bytecode class file called MyClass.class.

OPTIONS

-classpath *classpath*

Set the user class path, overriding the user class path in the `CLASSPATH` environment variable. If neither `CLASSPATH` or **-classpath** is specified, the user class path consists of the current directory.

If the **-sourcepath** option is not specified, the user class path is searched for source files as well as class files.

User class path entries are separated by colons (:) and can be directories, JAR archives, or ZIP archives.

-sourcepath *sourcepath*

Specify the source code path to search for class or interface definitions. If packages are used, the local path name within the directory must reflect the package name.

Note that classes found through the classpath are subject to automatic recompilation if their sources are found.

-d *directory*

Set the destination directory for generated java files and class files. If a class is part of a package, ojc puts the java file and class file in a subdirectory reflecting the package name, creating directories as needed.

If -d is not specified, ojc puts the class file in the same directory as the source file.

Note that the directory specified by -d is not automatically added to your user class path.

-nowarn

Disable warning messages.

-verbose

Verbose output. This includes information about each class loaded and each source file compiled.

-J*javaoption*

Passes through the string *javaoption* as a single argument to the Java interpreter which runs the compiler. The argument should not contain spaces. Multiple argument words must all begin with the prefix `-J`, which is stripped. This is useful for adjusting the compiler's execution environment or memory usage.

-C*javacoption*

Passes through the string *javacoption* as a single argument qto the Java compiler which is employed by the compiler. The argument should not contain spaces. Multiple argument words must all begin with the prefix `-C`, which is stripped. This is useful for adjusting the compiler's execution environment or memory usage.

ENVIRONMENT VARIABLES

CLASSPATH

Used to provide the system a path to user-defined classes. Directories are separated by colons, for example,

```
.:/home/avh/classes:/usr/local/java/classes
```