OpenJava: A Class-based Macro System for Java

Michiaki Tatsubori[1], Shigeru Chiba[2], Marc-Olivier Killijian[3], and Kozo Itano[2]

[1] Doctral Program in Engineering, University of Tsukuba,
Tennohdai 1-1-1, Tsukuba, Ibaraki, Japan,
mt@is.tsukuba.ac.jp
[2] Department of Information Science and Electronics, University of Tsukuba
[3] Laboratory for Analysis and Architecture of Systems,
Centre National de la Recherche Scientifique

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

1 Introduction

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

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

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

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

In the rest of this paper, section 2 presents a problem of ordinary macro systems and section 3 discusses the design and implementation of OpenJava, which addresses this problem. We compare OpenJava with related work in section 4. Finally, section 5 concludes this paper.

2 Problems with Ordinary Macros

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

2.1 Programmable Macros

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

Programmable macros have been developed for languages with more complex syntax like C. MS^2[19] is one of those macro systems for C. Macro functions are written in an extended C language providing special data structure representing ASTs. The users of MS^2 can define a new syntax and how it is expanded into a regular C syntax. The parameter that a macro function receives is an AST of the code processed by that macro function.

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

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

Some macro systems provide fine-grained control of where to apply a macro. In A^* [14], a macro is applied to expressions or statements matching a pattern specified in the BNF. In EPP[9], macros are applied to a specified syntax elements like `if` statements or `+` expressions. There's no need to put any trigger word in front of these statements or expressions.

2.2 Representation of Object-Oriented Programs

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

There is a great demand for the macros depending on logical and contextual information in OO programming. For example, some of design patterns[6] require relatively complicated programming. They often require programmers to repeatedly write similar code.[1] To help this programming, several researchers have proposed to extend a language to provide new language constructs specialized for particular patterns [1, 7]. Those constructs should be implemented with macros although they have been implemented so far by a custom preprocessor. This is because macros implementing those constructs depend on the logical and contextual information of programs and thus they are not implementable on top of the traditional AST-based macro systems.

Suppose that we write a macro for helping programming with the Observer[6] pattern, which is for describing one-to-many dependency among objects. This pattern is found in the Java standard library although it is called the event-and-listener model. For example, a Java program displays a menu bar must define a listener object notified of menu-select events. The listener object is an instance of a class MyMenuListener implementing interface MenuListener:

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

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

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

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

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

Writing this macro is difficult with traditional AST-based macro systems since it depends on the logical information of the definition of the class MyMenuListener. If a class definition is given as a large AST, the macro program must interpret the AST and recognize methods declared in MenuListener and MyMenuListener. The macro program must also construct ASTs representing the inserted methods and modify the original large AST to include these ASTs. Manipulating a large AST is another difficult task. To reduce these difficulties, macro systems should provide logical and contextual information of programs for macro programs. There are only a few macro systems providing the logical information. For example, XL[15] is one of those systems although it is for a functional language but not for an OO language.

3 OpenJava

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

3.1 Macro Programming in OpenJava

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

Applying Macros Fig. 1 shows a sample using a macro in OpenJava. By adding a clause `instantiates M` in just after the class name in a class declaration, the programmer can specify that the class metaobject for the class is an instance of the metaclass M. In this sample program, the class metaobject for MyMenuListener is an instance of ObserverClass. This metaobject controls macro expansion involved with MyMenuListener. The declaration of ObserverClass is described in regular Java as shown in Fig. 2.

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

Fig. 1. Application of a macro in OpenJava

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

Fig. 2. A macro in OpenJava

Every metaclass must inherit from the metaclass OJClass, which is a built-in class of OpenJava. The `translateDefinition()` in Fig. 2 is a method inherited from OJClass, which is invoked by the system to make macro expansion. If an `instantiates` clause in a class declaration is found, OpenJava creates an instance of the metaclass indicated by that `instantiates` clause, and assigns this instance to the class metaobject representing that declared class. Then OpenJava invokes `translateDefinition()` on the created class metaobject for macro expansion on the class declaration later.

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

Describing a Metaprogram The method `translateDefinition()` implementing the macro for the OBSERVER pattern in section 2.2 is shown in Fig. 3. This metaprogram first obtains all the member methods (including inherited ones) defined in the class by invoking `getMethods()` on the class metaobject. Then, if a member method declared in interfaces is not implemented in the class, it generates a new member method doing nothing and adds it to the class by invoking `addMethod()` on the class metaobject.

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

Fig. 3. `translateDefinition()` in ObserverClass

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

3.2 Class Metaobjects

As shown in section 2, a problem of ordinary macro systems is that their primary data structure is ASTs (abstract syntax trees) but they are far from logical sturctures of programs in OO languages. In OO languages like Java, class definitions play an important role as a logical structure of programs. Therefore, OpenJava employs the class metaobjects model, which was originally developed for reflective computing, for representing a logical structure of a program. The class metaobjects make it easy for meta programs to access a logical structure of program.

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

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

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

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

Logically Structured Class Representation Simple ASTs, even arranged and abstracted well, cannot properly represent a logical structure of a class definition. The data structure must be carefully designed to corresponded not only to the grammer of the language but also to the logical constructs of the language like classes and member methods. Especially, it makes it easy to handle the logical information of program including association between names and types.

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

3.3 Class Metaobjects in Details

The root class for class metaobjects is OJClass. The member methods of OJClass for obtaining information about a class are shown in Tab. 1 and Tab. 2. They cover all the attributes of the class. In OpenJava, all the types, including array types and primitive types like `int`, have corresponding class metaobjects. Using the member methods shown in Tab. 1, metaprogramms can inspect whether a given type is an ordinary class or not.

Tab. 3 gives methods for modifying the definition of the class. Metaprograms can override `translateDefinition()` in OJClass so that it calls these methods for executing desired modifications. For instance, the example shown in Fig. 3 adds newly generated member methods to the class with `addMethod()`.

Table 1. Member Methods in OJClass for Non-Class Types

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

Metaobjects Obtained through Class Metaobjects The method `getSuperclass()` in OJClass, which is used to obtain the superclass of the class, returns a class metaobject instead of the class name (as a string). As the result, metaprogram can use the returned class metaobject to directly obtain information about the superclass. OpenJava automatically generates class metaobjects on demand, even for classes declared in another source file or for classes available only in the form of bytecode, that is, classes whose source code is not available.

The returned value of the member method `getModifiers()` in Tab. 2 is an instance of the class OJModifier. This class represents a set of class modifiers such as `public`, `abstract` or `final`. Metaprogramms do not have to care about the order of class modifiers because OJModifier hides such useless information.

The class OJMethod, which is the return type of `getDeclaredMethods()` in OJClass, represents a logical structure of a method. Thus, similarly to the class OJClass, this class has member methods for examining or modifying the attributes of the method. Some basic member methods in OJMethod are shown in Tab. 4. Any type information obtained from these methods is also represented by a class metaobject. For instance, `getReturnType()` returns a class metaobject as the return type of the method. This feature of OJMethod is also found in OJField and OJConstructor, which respectively represent a member field and a constructor.

The class StatementList, which is the return type of the member method `getBody()` in the class OJMethod, represents the statements in a method body. An instance of StatementList consists of objects representing either expressions or statements. StatementList objects are AST-like data structures although they contain type information. This is because we thought that the logical structure of statements and expressions in Java can be well represented with ASTs.

Table 2. Member Methods in `OJClass` for introspection (1)

`String getPackageName()`	Returns the package name this class belongs to.
`String getSimpleName()`	Returns the unqualified name of this class.
`OJModifier getModifiers()`	Returns the modifiers for this class.
`OJClass getSuperclass()`	Returns the superclass declared explicitly or implicitly.
`OJClass[] getDeclaredInterfaces()`	Returns all the declared superinterfaces.
`StatementList getInitializer()`	Returns all the static initializer statements.
`OJField[] getDeclaredFields()`	Returns all the declared fields.
`OJMethod[] getDeclaredMethods()`	Returns all the declared methods.
`OJConctructor[] getDeclaredConstructors()`	Returns all the constructors declared explicitly or implicitly.
`OJClass[] getDeclaredClasses()`	Returns all the member classes (inner classes).
`OJClass getDeclaringClass()`	Returns the class declaring this class (outer class).

Table 3. Member Methods in `OJClass` for modifying the class

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

Table 4. Basic Methods in OJMethod

`String getName()`	Returns the name of this method.
`OJModifier getModifiers()`	Returns the modifiers for this method.
`OJClass getReturnType()`	Returns the return type.
`OJClass[] getParameterTypes()`	Returns the parameter types in declaration order.
`OJClass[] getExceptionTypes()`	Returns the types of the exceptions declared to be thrown.
`String[] getParameterVariables()`	Returns the parameter variable names in declaration order.
`StatementList getBody()`	Returns the statements of the method body.
`String setName(String name)`	Sets the name of this method.
`OJModifier setModifiers(OJModifier modifs)`	Sets the method modifiers.
`OJClass setReturnType()`	Sets the return type.
`OJClass[] setParameterTypes()`	Sets the parameter types in declaration order.
`OJClass[] setExceptionTypes()`	Sets the types of the exceptions declared to be thrown.
`String[] setParameterVariables()`	Sets the parameter variable names in declaration order.
`StatementList setBody()`	Sets the statements of the method body.

Logical Structure of a Class Tab. 5 shows the member methods in OJClass handling a logical structure of a class. Using these methods, metaprograms can obtain information considering class inheritance and member hiding. Although these member methods can be implemented by combining the member methods in Tab.2, they are provided for convenience. We think that providing these methods is significant from the viewpoint that class metaobjects represent a logical structure of a program.

Table 5. Member Methods in OJClass for introspection (2)

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

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

Moreover, accessibility of class members is restricted in Java by member modifiers like `public`, `protected` or `private`. Thus, `getMethods()` returns only the member methods available from the class specified by the argument. For instance, if the specified class is not a subclass or in the same package, `getMethods()` returns only the member methods with `public` modifier. In Fig. 3, since the metaprogram passes `this` to `getMethods()`, it obtains all the member methods defined in that class.

3.4 Type-Driven Translation

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

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

Here, we take up an example of a macro that enables programming with the FLYWEIGHT[6] pattern to explain this mechanism. This design pattern is applied to use objects-sharing to support large numbers of fine-grained objects efficiently. An example of macro supporting uses of this pattern would need to translate an instance creation expression of a class Glyph:

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

into a class method call expression:

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

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

In OpenJava, this kind of macro expansions are implemented by defining a metaclass FlyweightClass to be applied to the class Glyph. This metaclass overrides the member method `expandAllocation()` of OJClass as in Fig.4. This method receives a class instance creation expression and returns a translated expression. The system of OpenJava examines the whole source code and apply this member method to each Glyph instance creation expression to perform the macro expansion.

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

Fig. 4. Replacement of class instance expressions

The member method `expandAllocation()` receives an AllocationExpression object representing a class instance creation expression and an Environment object representing the enviroment of this expression. The Environment object holds name binding information such as type of variable in the scope of this expression.

OpenJava uses type-driven translation to enable the complehensive macro expansion of partial code spreaded over various places in program. In macro systems for OO programing languages, it is not only needed to translate a class declaration simply but translating expressions using the class togather is also needed. In OpenJava, by defining a methods like `expandAllocation()`, metaprogrammers can selectively apply macro expansion to the limited expressions related to classes controlled by the metaclass. This kind of mechanism has not been seen in most of ordinary macro systems except some systems like OpenC++[3]. Tab. 6 shows the primary member methods of OJClass which can be overridden for macro expansion at caller-side.

Table 6. Member Methods for Each Place Applied the Macro-Expansion to

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

3.5 Translation Mechanism

Given a source program, the processor of OpenJava:

1. Analyzes the source program to generate a class metaobject for each class.
2. Invokes the member methods of class metaobjects to perform macro expansion.
3. Generates the regular Java source program reflecting the modification made by the class metaobjects.
4. Executes the regular Java compiler to generate the corresponding byte code.

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

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

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

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

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

Therefore, OpenJava automatically preserves metalevel information such as the metaclass name for a class when it processes the callee-side translation of each class. These preservation are implemented by translating these information into a string held in a field of a special class, which is to be compiled into byte code. The system uses this byte code to obtain necessary metalevel information in another process without source code of that class. Additionally, metaprogrammers can request the system to preserve customized metalevel information of a class.

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

3.6 Syntax Extension

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

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

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

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

– after the name of the class.

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

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

Fig. 5. An example of syntax extension in OpenJava

Fig. 5 shows an example source program using a macro, a metaclass AdapterClass, supporting programming with the ADAPTER pattern[6]. The metaclass introduces a special clause beginning with `adapts` to make programmers to write special description for the ADAPTER pattern in the class declaration. The `adapts` clause in the Fig. reffig:VectorStack VectorStack is the adapter to a class Stack for a class Vector. The information by this clause is used only when the class metaobjects representing VectorStack performs macro expansion. Thus, for other class metaobjects, semantical information added by the new clause is recognized as a regular Java source code.

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

Fig. 6. A meta-program for a customized suffix

To introduce this `adapts` clause, metaprogrammers implement a member method `getDeclSuffix()` in the metaclass AdapterClass as shown in Fig. 6. The member method `getDeclSuffix()` is invoked by the system when needed, and returns a SyntaxRule object representing the syntax grammer begenning with the given special word. An instance of the class SyntaxRule implements a recursive descendant parser of LL(k), and analyzes a given token series to generate an appropriate AST. The system uses SyntaxRule objects obtained by invoking `getDeclSuffix()` to complete the parsing.

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

3.7 Metaclass Model of OpenJava

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

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

4 Related Work

There are a number of systems using the class metaobjects model for representing a logical structure of a program: 3-KRS[16], ObjVlisp[5], CLOS MOP[13], Smalltalk-80[8], and so on. The reflection API[11] of the Java language also uses this model although the reflection API does not allow to change class metaobjects; it only allows to inspect them. Furthermore, the reflection API uses class metaobjects for making class definition accessible at runtime. On the other hand, OpenJava uses class metaobjects for macro expansion at compile-time.

OpenC++[3] also uses the class metaobject model. OpenJava inherits several features, such as the type-driven translation mechanism, from OpenC++. However, the data structure mainly used in OpenC++ is still an AST (abstract syntax tree). MPC++[10] and EPP[9] are similar to OpenC++ with respect to the data structure. As mentioned in section 2, an AST is not an appropriate abstraction for some macros frequently used in OO programming.

5 Conclusion

This paper describes OpenJava, which is a macro system for Java providing a data structure called class metaobjects. A number of research activities have been done for enhancing expressive power of macro systems. This research is also in this stream. OpenJava is a macro system with a data structure representing a logical structure of an OO program. This made it easier to describe typical macros for OO programming which was difficult to describe with ordinary macro systems.

To show the effectiveness of OpenJava, we implemented some macros in OpenJava for supporting programming with design patterns. Although we saw that class metaobjects are useful for describing those macros, we also found limitations of OpenJava. Since a single design pattern usually contains several classes, a macro system should be able to deal with those classes as a single entity[17]. However it is not easy for OpenJava to do that because macros are applied to each class. It is future work to address this problem by incorporate OpenJava with techniques like aspect-oriented programming[12].

References

[1] Bosch, J. Design patterns as language constructs. *Journal of Object Oriented Programming* (1997).

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

[3] Chiba, S. A metaobject protocol for c++. *SIGPLAN Notices 30*, 10 (1995), 285–299.

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

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

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

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

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

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

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

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

[12] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J. M., and Irwin, J. Aspect-oriented programming. *LNCS 1241* (June 1997), 220–242.

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

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

[15] Maddox, W. Semantically-sensitive macroprocessing. Master's thesis, University of California, Berkeley, 1989. ucb/csd 89/545.

[16] Maes, P. Concepts and experiments in computational reflection. *SIGPLAN Notices 22*, 12 (October 1987), 147–155.

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

[18] Steel Jr., G. L. *Common Lisp: The Language*, 2nd ed. Digital Press, 1990.

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