Programming Support of Design Patterns with Compile-time Reflection [*]

Michiaki Tatsubori [†] and Shigeru Chiba
Institute of Information Sciences and Electronics,
University of Tsukuba, Japan
{mt,chiba}@is.tsukuba.ac.jp

Abstract

This paper presents that compile-time MOPs can provide a general framework resolving implementation problems of design patterns. The problems come from the fact that some programs written according to design patterns are too complicated and errorprone and that their overall structure is not easy to understand. This problem can be resolved by syntax extensions and extended language constructs that simplify description of the patterns and improve the readability of the programs. In our approach, programmers can use a MOP to write a library which implements syntax extensions and extended language constructs for supporting each design pattern. We illustrate this approach with examples written in OpenJava, which is our self extensible version of the Java language with a compile-time MOP. The Adapter pattern and the Visitor pattern are used as examples.

1 Introduction

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 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 [Bosch 97, Ducasse 97, Gil & Lorenz 98].

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[Tatsubori 97, Chiba & Tatsubori 98], which is an self extensible version of the Java language[Gosling *et al.* 97, Kramer 97] with a class based compile-time MOP succeeding to OpenC++[Chiba 95].

In the rest of this paper, how to resolve the problems with a compile-time MOP is described in the section 2 with an example of the Adapter pattern. Then, in the section 3, another example using the Visitor pattern is shown.

2 Compile-time MOP for Design Patterns

In this section, we show an example using the Adapter pattern. The Adapter pattern is used to *convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces* [Gamma *et al.* 94]. Figure 1 shows a structure of the Adapter pattern.

Suppose that a programmer has to adapt a class Vector to an interface Stack, which are defined as fol-

[*]Submitted for OOPSLA'98 Workshop Reflective Programming in C++ and Java, Vancouver, Canada, Oct 18, 1998.

[†]Tennohdai 1-1-1, Tsukuba, Ibaraki 305-8573, Japan. Phone: +81-298-53-5349 Fax: +81-298-53-5206

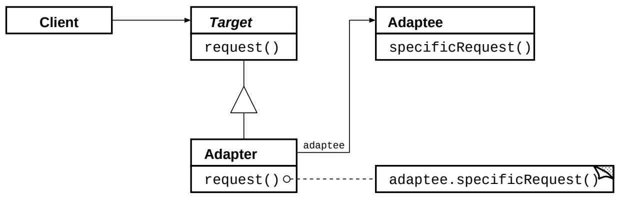

Figure 1: A structure of the Adapter pattern

lows:

Listing 2.1 *Stack.java*
```
public interface Stack
{
  boolean isEmpty();
  Enumeration elements();
  Object peek();
  void push(Object o);
  Object pop();
}
```

Listing 2.2 *Vector.java*
```
public class Vector
{
  boolean isEmpty();
  Enumeration elements();
  Object lastElement() { ....  }
  void addElement(Object o) { ....  }
  ....
}
```

A class Vector and an interface Stack corresponds to the Adaptee and the Target respectively in figure 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.

Implementation Problems

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 other design patterns and this fact has 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].

Solution by Special Annotations

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 2.4 *VectorStack.oj*
```
public class VectorStack
    instantiates AdapterPattern
    adapts Vector 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, a Target class, and an Adaptee class

[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.

3. 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 `to` T
   specifies that this class is an Adapter adapting the class A 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 with Compile-time MOP

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

  /* overrides for translation */
  void translate() 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 2.4.

3 Another Example

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 2 shows a structure of the Visitor pattern.

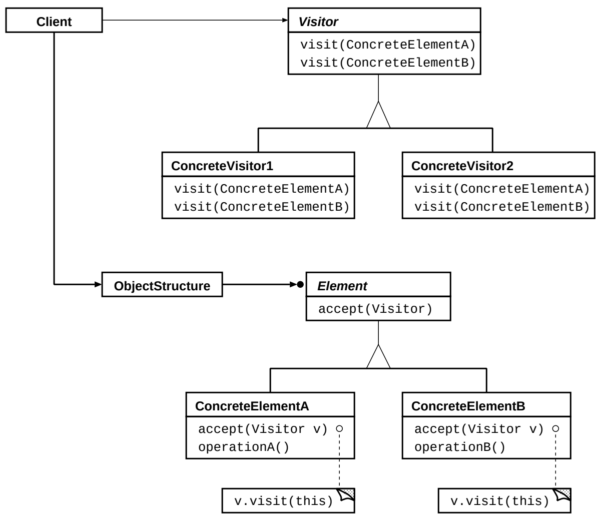

Figure 2: A structure of the Visitor pattern

Ordinal Implementation

Suppose that a programmer should implement a GUI class library according to the Visitor pattern. The

Visitor and the Element in Figure 2 are represented by an interface GUIVisitor and an interface GUIElement:

Listing 3.1 *GUIElement.java*
```
public interface GUIElement
{
  void accept(GUIVisitor v);
}
```

Listing 3.2 *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 3.3 *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 3.4 *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 3.5 *GUIElement.oj*
```
public interface GUIElement
    instantiates VisitorPattern
    accepts GUIVisitor
{
  void accept(GUIVisitor v);
}
```

Listing 3.6 *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 3.7 *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 3.8 *PartsCounter.oj*
```
public class PartsCounter
    instantiates VisitorPattern
    visits as GUIVisitor
{
  void visit(Container e) { ....  }
  void visit(Label e) { ....  }
}
```

Metalevel program

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

Listing 3.9 *VisitorPattern.oj*
```
public class VisitorPattern extends OJClass
{
  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 3.1, 3.2, 3.3 and 3.4 from the code in listing 3.5, 3.6, 3.7 and 3.8.

4 Conclusion

When implementing design patterns, programmers are faced with problems because some programs written according to design patterns tend to be too complicated and errorprone and their overall structure is not easy to understand.

We claims that compile-time MOPs provide a general framework for implementing syntax extensions and extended language constructs which help design pattern users to write their programs and improve the readability of programs written with design patterns. Also, this paper has illustrated how programmers write a meta-level library for every design pattern with compile-time MOPs.

Two examples of the Adapter pattern and the Visitor pattern have been described using our OpenJava, which is an self extensible version of the Java language with a compile-time MOP.

References

[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.

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

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

[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.

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

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

[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.

[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.

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

[Tatsubori 97] Michiaki Tatsubori : OpenJava WWW page, `http://www.softlab.is.tsukuba.ac.jp/~mich/openjava/index.html` , 1997-1998.