Object-Oriented Modeling of Software Patterns

Takashi Kobayashi  Masahiko Kamo  Takayuki Sanui  Motoshi Saeki
Dept. of Computer Science, Tokyo Institute of Technology
Ookayama 2-12-1, Meguro-ku, Tokyo 152-8552, Japan
Tel: +81-3-5734-2192 / Fax:+81-3-5734-2917 / {tkobaya,saeki}@cs.titech.ac.jp

Abstract

This paper discusses a technique to model software patterns such as Gang-OF-Four (GOF) design patterns for supporting pattern based software development. Software patterns are general structures that frequently appear in the artifacts and encourage effective reuse of past successful experience. To support pattern based software development, we model the patterns from object-oriented view so that machine processing is possible. In our approach, we consider that a pattern consists of a pattern structure (a class diagram) and manipulation operations on the pattern structure in order to use it. These manipulations are classified into three types: pattern instantiation (applying a pattern to an actual problem, i.e. filling their hot spots), pattern combination (combining several patterns into larger one) and pattern refinement (refine a pattern into more concrete one). These manipulation operations are specific to patterns and each pattern is a class which they are encapsulated into. We have described 22 Gang-Of-Four design patterns with Java and developed a supporting tool for using patterns. The tool has been developed by using our approach itself and it shows the usefulness of our approach.

Keyword: Design Pattern, Object-Oriented Modeling, Software Pattern, Java

1 Introduction

Software patterns such as Gang-of-Four (GOF) design pattern[4], Fowler’s analysis pattern[3] and architectural patterns[1] are one of the promising techniques to develop software of high quality efficiently. They are general and abstract structures that frequently appear in past experiences in developing well-structured and maintainable artifacts and can be efficiently re-used for new software development projects.

Although many researchers and practitioners jointly have much efforts on collecting and cataloguing patterns [7], it may be difficult for software developers to select suitable patterns and use efficiently them, when we only provide informal descriptions of the patterns like textbooks. To support software development based on patterns, we should model the patterns as formally or semi-formally as computers can manage, so that a computerized tool like [2] can support software development processes based on software patterns.

In this paper, we propose a modeling technique of software patterns from object-oriented viewpoint, and describe them with the programming language Java, a formal language. Describing patterns with Java allows us to execute the patterns and the execution of the patterns means their applications during actual development processes to complete an artifact. It leads the enactment of software development processes based on patterns. The rest of the paper is organized as follows. In the next section, we show the basic idea of object-oriented modeling for software patterns. A pattern has a specific structure and specific manipulation operators to change its structure. During a pattern-based development process, the structure of the applied pattern is incrementally being changed as the process goes by. Manipulation operators specify how we can change the structure of a pattern when it is applied, and these operators are specific to a pattern. From object-oriented view, a pattern is modeled as a class encapsulating the operators that are for filling its hot spots when we apply it to software development, and we call this type of operation instantiation. Sections 3 and 4 discuss the other two types of manipulation operators for evolving patterns; combination (combining patterns to have a larger pattern) and refinement (refining a pattern to more concrete one). Pattern combination and refinement can be modeled as aggregation and specialization (inheritance) of patterns respectively. In section 5, we will introduce the experience in describing 22 GOF design patterns with Java and in developing a supporting tool for pattern-based software development. We have applied our modeling technique to the development of the tool itself. Finally we discuss this technique and list up the future work.

2 Object-Oriented Model for Software Patterns

2.1 Command Pattern Example

Before modeling software patterns, we consider a simple example of GOF design pattern. Figure 1 (a) and (b) illus-
abstract class command

(a) Command Pattern

Command Pattern

(b) Using Command Pattern

ConcreteCommandA

ConcreteCommandB

Figure 1. Command Pattern

trate Command Pattern and the way to use it, respectively. Command Pattern has several commands as classes, which has the same interface “execute()”. When Invoker wants to execute a command, it sends the message “execute()” to the object which is stored in the variable command, i.e. we write the statement “command.execute()” in the method of Invoker class. The class Command is an abstract class and has no implementation of the method “execute()”. To implement “execute()”, we add a concrete class as a subclass of the Command class and it has the implementation, i.e. the source code of “execute()”. As shown in Figure 1 (b), we can have ConcreteCommandA as a subclass. If we create an object of ConcreteCommandA and assign it to the variable command, Invoker can call “execute()” of ConcreteCommandA without any change of the source code of Invoker. If we would like to have a new command to be executed by Invoker, we only add the corresponding concrete class, say ConcreteCommandB, and update the variable command to an object of ConcreteCommandB before invoking its execution. The important point is that we do not need change the source code of Invoker because both of the commands has the same interface, i.e. “execute()”. This structure allows us to dynamically switch the contents of the commands to be executed.

To use Command Pattern, we should add as the subclasses of an abstract class the concrete classes that have the same interface to command execution. It is the only part where we can change the structure of Command Pattern when we use it. This kind of changeable parts are called hot spots. Which parts are hot spots in a pattern and how to fill the hot spots greatly depend on the pattern. In the example of Command Pattern, the way how to fill the hot spots is adding a subclass having “execute()” to an abstract class “Command”. “Adding a subclass to a class” can be considered as one of basic operations on the structure of a pattern and we can define how to fill the hot spots with a sequence of basic operations. We call this kind of operation to fill the hot spots instantiation operations. Since a pattern with a pair of its structure and the instantiation operations. The detailed definitions of the operations can be encapsulated into the definition of patterns. That is to say, we can model a pattern as a class (say, pattern class) from object-oriented viewpoint. In this modeling, the structure of a pattern is produced by executing its constructor (a method for object creation) and the operations are considered as methods of a pattern class (instance methods). See Figure 2.

When we use a pattern, first of all we execute a constructor of this pattern class and generate an instance of the class. The generated pattern instance has its initial structure as shown in Figure 1 (a). We incrementally change its structure, i.e. fill its hot spots by using its instantiation operations and get the final artifact of the system to be developed.

Although we can have various kind of representation techniques of pattern structures, e.g. use case diagrams for use case patterns [6], we use class diagrams because class diagrams are widely used and almost all of patterns are on class diagrams.

2.2 Basic Operations

As we discussed in the last section, the operations to change pattern structures are on class diagrams. The typical examples of basic manipulation operators on class diagrams are listed up below. We have implemented these operations with Java, All of the operations to change pattern structures can be defined by using these basic operations.

• createClass(’Name’)
  Creating a class whose name is ’Name’. (Constructor of Class)
• Class1.addSubClass(Class2)
  Adding a class Class2 as a subclass of Class1.
holds the information for the reference to the abstract class of basic operations. The global variable "acommand" has these definitions in addition to basic operations and association one respectively. The class "PatternClass" PMMethod and PMAssociation are used for method type class appearing in a class diagram of a pattern structure. For example, PMClass stands for the type of patterns. We can get the initial structure of Command Pattern as a constructor "CommandPattern()" by using basic operations as follows;

```java
public CommandPattern()
{
    acommand = createClass('Command') ;
    invokerTocommand = createClass('Invoker') ;
    PMMethod m1 = createMethod('execute()' + null) ;
    acommand.addMethod(m1) ;
    invokerTocommand.addAssociation(acommand) ;
    acommand.addSubclass(invokerTocommand) ;
}
```

How to use these basic operations will be discussed in the next subsection. Note that these basic operations can be considered as constructors of class diagrams and that the effects of executing the operations follow the usual semantics of Java programs.

### 2.3 Describing Pattern Structures

As discussed before, a pattern structure is generated as an instance when we use the pattern. In the case of Command Pattern, its initial structure has the classes Invoker and Command, the method execute() in Command class, and the association command. Thus we can describe the initial pattern structure as a constructor “CommandPattern()” by using basic operations as follows;

```java
CommandPattern extends PatternClass {
    public PMMethod cmethod ;
    public PMClass cclass ;

    public CommandPattern()
    {
        acommand = createClass('Command') ;
        invokerTocommand = createAssociation('Command') ;
        invokerTocommand.addAssociation(
            invokerTocommand, acommand) ;
        PMMethod m1 = createMethod('execute()' + null) ;
        acommand.addMethod(m1) ;
    }
}
```

Note that, to avoid the conflicts to the reserved words of Java such as “class”, in Java source codes we attached the prefix “PM” for the words denoting the constituents of patterns. For example, PMClass stands for the type of class appearing in a class diagram of a pattern structure. PMMethod and PMAssociation are used for method type and association one respectively. The class “PatternClass” has these definitions in addition to basic operations.

We can get the initial structure of Command Pattern shown in Figure 1 (a), by executing the above sequence of basic operations. The global variable “acommand” holds the information for the reference to the abstract class “Command” and the other operations can access it through this variable. “Class invoker” is the declaration of the variable “invoker” to which an object of “Class” is assigned as the constituent Invoker of the pattern. The operation createClass(‘Invoker’) generates an object of Class whose name is “Invoker”. That is to say, constituents of a class diagram such as classes, methods and associations are considered as the objects that a class diagram contains. Method m1 has a method name “execute()”, while its method body is none (null) because it is an abstract method. Note that the operator + stands for the concatenation of string data. For example, the expression "fooMethod(int A)" + fooMethodBody results in "foomethod(int A) { return A }" where the string "{" return A }" is assigned to the variable fooMethodBody.

### 2.4 Instantiation Operation

How to describe instantiation operations as methods of a pattern class is presented in this section by using the example of Command Pattern. The essential point of the instantiation operation on Command Pattern is adding a concrete class ConcreteClass having the implementation of execute(). Therefore we can get the following source code as an instance method of pattern class “CommandPattern”;

```java
createCommand(String new_command) {
    PMClass cclass = createClass(new_command + 'Command') ;
    acommand.addSubclass(cclass) ;
    PMMethod cmethod = createMethod('execute()' + null) ;
    cclass.addMethod(cmethod) ;
}
```

### 2.5 Meta Model of Patterns

Figure 3 shows the overview of the constituents and their relationships that our model of software patterns consist of. It is represented in a form of class diagram and called a meta model of software patterns. It is considered as a schema of object base with which a pattern catalog or library is implemented in a support tool for pattern based software development.

![Figure 3. Overview of a Meta Model of Software Patterns](image)
In the figure, the Role class expresses what roles the classes and the objects appearing in a pattern play and it is used for accessing constituents of the pattern, i.e. Class, Association, Method and Attribute objects in the pattern. It corresponds to global variables of a pattern class written in Java. Remember the global variables acommand, invoker and invokerTocommand of Command Pattern description in section 2.3.

3 Pattern Combination

The example of Command Pattern was a very simple pattern and was used just for the explanation of our modeling technique. In actual setting, we always design software combining several existing design patterns, and furthermore sometimes register the combinations of patterns as new patterns. These larger and more complicated patterns are useful to develop larger and/or domain-specific software systems [5]. The combination of patterns is also a pattern for composing coarse grain patterns from finer grain ones.

Consider a simple example of a pattern combination of Command pattern and Composite pattern of GOF design patterns. This pattern is often used to design graphic editors like Tgif, Xfig and Visio, for inputting and editing graphical objects such as polygons and circles. Figure 4 illustrates how to combine Command and Composite Patterns. Composite Pattern is frequently used to design the objects having recursive structure such as tree structure. In this example, it is used to represent graphic objects hierarchically. The graphic editor has a function of grouping several graphic objects into one and a group of the objects can be designed as a class “Composite” in the pattern. If an operation is invoked to a composite object, its execution is repeated on all of the elements in the composite object. For example, when Invoker is requested to perform the command of moving a selected graphic object on the display screen, it sends the message “execute()” to a concrete command MoveCommand, in order to invoke the move operation. MoveCommand sends the message “move()” to the selected object which is specified through the association refer from Command class to Component class. Composite class distributes it to all of its children objects (for each o ∈ children o.move()). These children objects know how to execute the move operation by themselves. For example, the circle objects change the X and Y coordinates of their centers, while in the case of the polygons, all of their vertices should be moved. Note that the hot spot of Composite Pattern is the subclass of the abstract class of Component and we can add various kind of classes of graphic objects such as Ellipse and Triangle.

The hot spots of a combined pattern result from the hot spots of that patterns it contains. Thus the pattern “Command-Composite Pattern” has as its hot spots both of Command and Composite patterns’ hot spots. For example if we need a new command of resizing graphic objects, we can add the concrete command class “ResizeCommand” as a subclass of Command by using the instantiation operation of Command Pattern as shown in section 2.4. We can also extend the design of the graphic editor so that it can manipulate polygon objects by adding a subclass “Polygon” to Composite class. It can be done by the instantiation operation of Composite Pattern.

Although we can fill the hot spots of Command Pattern and Composite Pattern independently by using their instantiation operations, we may frequently fill more than one hot spots, because each hot spot has a dependency relationship to the other spots in this combined pattern. For example, if the concrete class MoveCommand has been already added, any concrete class of graphic objects should have the operation “move()” that really moves the object. Assume that we add a new command that resizes the selected object, by using the instantiation operation of Command Pattern. The effect of adding the new command on the Command Pattern should be propagated to the combined Composite Pattern part. More concretely, we should add to each class the method “resize()” which changes the size of the graphic objects, and its contents, i.e. source code depend on the class.

Thus an instantiation operation “createCommand()” for the combined patterns 1) invokes the “createCommand()” to Command Pattern part, and then 2) performs the operation for adding a method “resize()” to all of the concrete classes and Composite class. Thus we can get the description of this combined pattern as follows;

```java
public createCommand(String new_command) { } // 1

PMMethod m1 = createMethod(new_command + null); // for each c in children o.move() 
for (i = 0; i < composite-pattern.leaf.size(); i++) {
	((composite-pattern.leaf).atElement(i)).addMethod(m1); // 2
}

PMMethod m2 = createMethod(new_command + null); // (composite-pattern.component).addMethod(m2); // 3

PMMethod m3 = createMethod(new_command + null); // (composite-pattern.compose).addMethod(m3); // 4
...}
```

If we send the message “createCommand(‘resize’)” to the instance of Command-Composite Pattern, we can get the instantiated pattern or the design that the resize command is embedded. In the code, we use the access to the constituents of the patterns such as “composite-pattern.component” and “composite-pattern.leaf”. The former expression is for accessing Component class of Composite Pattern part of Figure 4, and the latter for a set of concrete classes connected...
to Component as its subclasses. The concrete classes are stored in the variable *lea f* of type Vector and we use the method atElement to get the i-th element of the connected concrete classes.

The following is the part of the definition of Composite Pattern.

```java
CompositePattern extends PatternClass {

    /* Global Variables for Accessing
        Constituents of the Pattern */
    public PMClass component, composite;
    public Vector leaf = new vector();
    public PMAssociation children;

    /* Constructor */
    CompositePattern() {
        setName('CompositePattern');
        component = createClass('Component');
        composite = createClass('Composite');
        children = createAssociation('children');
        component.addAssociation(children, composite);
    }
    ...
}
```

Note that in this combined pattern we can also define the instantiation operation of adding graphic object classes to Composite class and it can have propagate the ripple effects caused by this addition.

Although how to combine patterns greatly depends on the dependency relationship between the two patterns and the relationships can be complicated, in our approach we can specify them because we write them as Java program from operational view to keep consistency among the patterns. To model pattern combination, we used aggregation relationships on patterns. Figure 5 illustrates this technique of pattern combination. Suppose that we combine the patterns #1 and #2. We introduce a new pattern and create the aggregation relationships from the new one to #1 and to #2. That is to say, the patterns #1 and #2 are constituents of the new pattern. This can be considered as a pattern evolution technique that can provide larger and coarse-grained patterns such as application framework.

### 4 Pattern Refinement

In this section, we introduce third type of pattern manipulation operations called refinement by using a simple example.

#### 4.1 What is Refinement?

Software development processes can be considered as the successive refinement of artifacts. For example, a re-
quirements specification is refined to its design one, and the design is to a source code. As mentioned before, we have various kind of software pattern, e.g. Fowler’s analysis pattern for requirements analysis phase, GOF design pattern for object-oriented design and architectural pattern for architecture design. Similar to the refinement of the artifacts, software patterns are refined along the progress of software development. Suppose that a developer uses analysis patterns to model his problem domain from object-oriented view in requirements analysis step. In his design step, he applies design patterns to the model and gets its design specification. In this process, there must be some guidelines or know-how to apply design patterns to the analysis patterns. That is to say, the analysis pattern seems be refined to the patterns than can be used in design step, according to some rules. We will illustrate more concrete example of pattern refinement in the next two sections.

4.2 Analysis Pattern Example

Figure 6 shows one of the simplified versions of Analysis Patterns (by Fowler) that are used in requirement analysis step, and called Party-Accountability Pattern. It is used for specifying an organizational structure and relationships (accountability relationships) among persons in the organization, and has a flexible structure that can handle with dynamic changes the organizations and the relationships. For example, it is possible to add new accountability relationships and delete the relationships in the organization during the developed system are being executed. In the figure, accountability type declares what kind of relationships among the organizational units (e.g. department and branch etc.) and persons (we call together them parties) exist, while accountability specifies which party types can participate in the accountability type.

![Figure 6. Party-Accountability Pattern (Simplified Version)](image)

For example, consider that we express the simple organizational structure shown in Figure 7 with the Party-Accountability Pattern. In the figure, you can find two accountability relationships “lecture_to” and “supervise”. To specify that they are the accountability relationships between the party types “Professor” and “Student”, the corresponding instances of Accountability (expressed with *** in the figure) have the links to “Professor” and “Student”.

![Figure 7. Specification](image)

The hot spots of the Party-Accountability Pattern are “Accountability Type”, “Accountability” and “Party”, and what kind of instantiation operations can be made on each hot spot depend on the hot spot. In the example of Figure 7, as shown in Figure 8, the instantiation operation is composed from a sequence of three operations; 1) adding as a subclass of Party an entity class that the organization consists of (adding the classes “Professor” and “Student” to the Party as its subclass), 2) creating accountability relationship types as instances of Accountability Type (creating the relationships “supervise” and “lecture_to”), and 3) creating the instances of Accountability corresponding to the generated Accountability Type instances, and linking them to Parties (creating two Accountability instances corresponding to “supervise” and “lecture_to” and linking them to “Student” and “Professor”).

![Figure 8. Instantiation of Party-Accountability Pattern](image)

The following is the description of Party-Accountability Pattern together with the instantiation operation.

```java
class Party-Accountability_AnalysisPattern extends PMPattern{
    PMClass accountability_Type;
    PMClass accountability;
    PMClass party;
    Vector concreteParty = new vector();
    int i;
    ...
    Instantiate_Party(string ConcreteParty-name) {
        concreteParty.atElement(i) =
        createClass(ConcreteParty-name);
        party.addSubClass(ConcreteParty);
        i = i + 1;
    }
```
Instantiate_Accountability(Relationship, Participant#1, Participant#2) {
    Accountability_Type x
        = new Accountability_Type(Relationship) ;
    Accountability y = new Accountability() ;
    y.addAssociation(createAssociation(null),x) ;
    y.addAssociation(createAssociation(null),
        Participant#1) ;
    y.addAssociation(createAssociation(null),
        Participant#2) ;
}
/* Definition of Inner Classes */
class Accountability {
    ...
}
class Accountability_Type {
    ...
}

For example, to make the accountability “supervise”, we create an instance of Party-Accountability_AnalysisPattern class and send it a message: Instantiate_Accountability('supervise', Professor, Student). Note that we had inner class definitions “class Accountability { ... }” and “class Accountability_Type { ... }”, because we need generating instances of Accountability and Accountability_Type classes.

4.3 Pattern Refinement Example

Let’s suppose that we proceed the design task on Party-Accountability Pattern. To make use of this Party-Accountability Pattern in the structure of a design specification, we refine Accountability Type and Accountability classes so that they possess the mechanism to create instances (e.g. creating Accountability Type instances supervise and lecture_to, etc.).

The refinement rule of the analysis model to a design model is that we add a class Creator that creates Accountability Type instances and associate it with Accountability class. This refinement process is depicted in Figure 9. In the pattern, i.e. Figure 9 (a), the class Creator is an abstract class, and its concrete class is made as its subclass when creating the instances of Accountability during the instantiation process. For example, by introducing Accountability Type instance “supervise”, we add a concrete class “CreatorSupervise” that is for generating the links between a Student instance and a Professor one. The reason why we use an abstract class is that we can write a single code of making links between party instances independently of Accountability Types. This is the same strategy of Factory Method of GOF design patterns.

The way how to refine an analysis model into a design model depends on what instantiations have been done on Party-Accountability Pattern in the analysis model, and the pattern can have several refinement operations according to design strategies.

Figure 9. Refinement Example of Party-Accountability Pattern

4.4 Modeling Pattern Refinement

The way how to refine patterns is also specific to a pattern, and can be specified as a series of manipulation operations on a pattern structure (class diagram) in the similar way to the definition of instantiation operations. Pattern refinement causes not only the change of a pattern structure but also the change of instantiation operations. Therefore adding methods and/or modifying source codes in the methods in a pattern class is also necessary for a pattern refinement.

Figure 10. Pattern Refinement

The refinement keeps whole of or a part of the pattern structure. Comparing Figure 9 (a) with Figure 6, the readers can recognize that the structure of Party-Accountability Pattern is still included in its refinement. Thus we use specialization relationship on pattern classes to model pattern
refinement. Suppose that pattern #1 is refined into pattern #2. The pattern #2 is one of the specializations of pattern #1, and the structure and the instantiation operations of pattern #2 is inherited from pattern #1. The pattern structure and the operations of pattern #2 can be changed. To achieve these changes, we override the definitions of pattern #1 in the same way as the override mechanism of super-subclass inheritance. That is to say, we override the constructors for structural changes and the methods for changes of instantiation operations. Figure 10 illustrates these two changes of a pattern structure and of the methods as instantiation operations.

In the example of Figure 9, we have the following changes; 1) change of a pattern structure: the class Creator (naming the Accountability instances is connected to Accountability class (overriding the constructors), 2) change of an instantiation operation InstantiateAccountability: the Party classes participating in generated Accountability instances are added, and the method for associating the accountability with the parties is added (overriding the instantiation operations). In this case, since the analysis pattern can appear in a design pattern as it is, we can get the design pattern by inheriting the analysis pattern. We capture a specialization (inheritance) relationship as an refinement relationship. That is to say, a pattern class is refined into the pattern that is its subclass. The constructor of Refined_Party-Accountability_AnalysisPattern (Figure 9 (a)) and its instantiation operation are newly defined, i.e. the definitions of Party-Accountability_AnalysisPattern are overridden.

```java
/* New Constructor of Pattern */
PMClass creator ;
/* producing the structure of Party-Accountability Pattern */
super() ;
/* Connecting Creator to Accountability Class */
creator = createClass('Creator') ;
Accountability.addAggregation(creator) ;
/* Add an abstract method CreatorAccountability */
PMMethod m = createMethod('createAccountability(Party p1,
    Party p2) { }') ;
creator.addMethod(m) ;

/* New Instantiation Operation */
/* Making up Relationship the relationship */
Instantiate_Accountability(relation,
    participant#1, participant#2) {
    super.Instantiate_Accountability(relation,
        participant#1, participant#2) ;
    /* Connecting ConcreteCreator as a subclass */
    concreteCreator = CreateClass('Creator' + relationship) ;
    
    creator.addSubclass(concreteCreator) ;
    PMMethod m = createMethod('addLink(Party p1) {
        ...
    }' ;
    accountability.addMethod(m)
    
    /* Adding a method to Creator */
    This new method is to associate Relationship with Participants#1 and Participant#2 */
    PMMethod m = createMethod('createAccountability(' + participant#1 + ' p1,' + participant#2 + ' p2) {
        Accountability rel = new Accountability() ;
        rel.addAccountabilityType {relationship} ;
        rel.addLink(p1) ;
        rel.addLink(p2) ;
    }' ;
    concreteCreator.addMethod(m)
}
```

The Party-Accountability DesignPattern class has the constructor Party-Accountability DesignPattern() and it calls the constructor of its super class Party-Accountability AnalysisPattern at first. After creating an instance of Party-Accountability AnalysisPattern, the class “Creator” is generated and embedded into the instance of the pattern. This operator is the result of overriding the constructor of the analysis pattern in the design pattern.

The class Accountability, which is one of the elements of the pattern, has the method “addAccountabilityType(relationship)”, and the method is used for relating Relationship (Accountability Type, i.e. the name of accountability) to the Accountability instance. By using this method, we can set the name of Accountability “supervise” to the generated Accountability instance. The method addLink is used for making a link between two instances. If we execute “rel.addLink(p1)” (strictly saying, send a message addLink(p1) to the object rel) and rel.addLink(p2), we establish the two links of Accountability instances to Party instances p1 and p2 respectively. When we perform the instantiation operation Instantiate Accountability (‘supervise’ Professor, Student) on the Party-Accountability AnalysisPattern, the method createAccountability (Professor p1, Student p2), where p1 and p2 are a Professor instance and a Student one respectively, is automatically installed to the concrete class CreatorSupervise. The concrete class CreatorSupervise is created at the invocation of creatorClass (‘Creator’ + relationship) where the variable “relationship” has the value ‘Supervise’.

5 Supporting Tool

5.1 Implementing a Supporting Tool

As one of the application of our modeling technique, we have developed a prototype of a tool for supporting software design based on GOF design patterns. The tool has two parts; one is an editor of class diagrams with graphical user interface and the other one is a manger of design patterns.
We have described 22 GOF design patterns with Java by using our description technique mentioned in section 2.

Figure 11 shows a snapshot of using the tool. A developer chooses a suitable design pattern by clicking the fifth button from the top in the left line. The initial structure of the chosen pattern is displayed on the screen. In this example, he chose Command Pattern (displayed on the left side) and then Composite Pattern (right side).

Whenever he wants to change the names of the classes, operations and attributes in the displayed class diagram, he can do that. When he has a double click on the class, a dialog box to enter new names appears. In the figure, one of ConcreteCommand classes has been re-named to MoveCommand. Furthermore he can add and delete classes, operations, attributes, associations, aggregation relationships and specialization ones as he uses a usual class diagram editor, because each of them can be considered as a pattern by itself. After choosing the two patterns, he added an association between the Command Pattern and the Composite Pattern.

If he wants to fill the hot spots of the pattern, he puts a mouse cursor on any constituents of the pattern and clicks it. He can see a list of allowable instantiation operations of the pattern. In the figure, createCommand and createCommandwithReceiver are shown as the allowable operations of the Command Pattern. When he selects one of them, the selected operation, written with Java, is executed and a concrete command class is automatically added and displayed on the screen.

5.2 Experience

After developed this prototype tool, we have re-designed it by using the tool itself. Figure 12 shows a part of the design of the tool by using it. It is the part of a pattern model and the readers can recognize the same structure as the meta model of patterns shown in Figure 3. The result of re-designing had 37 classes, and 34 classes of the 37 belong to the constituents of GOF design patterns. They have been derived only by the instantiation operations on the design patterns. Eight design patterns such as Command, Composite, State, Singleton, Observer etc. were applied and the hot spots of the 7 design patterns of the 8 were completely filled by using our pre-defined instantiation operations. This result means that applying our modeling technique to GOF design patterns works well and this kind of tool is useful to support pattern-based software development. The shortcomings of the tool is that it has no support of re-doing design tasks when a developer recognizes that he used unsuitable patterns during his design task. However, the tool can record a sequence of applying instantiation operations as the history of a design task. To analyze the sequence, it seems to be possible to support how the developer can go back and re-do his design task without abandoning all of the constituents in the design specification.

6 Conclusion

In our approach, we considered that a software pattern consists of a pattern structure (class diagram) and manipulation operations on the pattern structure. These operations are for pattern instantiation (filling hot spots). We modeled them with object-oriented technique encapsulating these operations into patterns, and described patterns with Java-like object-oriented language. In addition, we have explored two type of pattern manipulation; pattern combination and pattern refinement. These have been modeled as aggregation and specialization of pattern classes respectively.

We can pick up the research agenda for future work as follows.

- Supporting tools of refinement on pattern:
  By using the supporting tool, users can select patterns and combine them into the pattern suitable for their application domain. They are also supported to instantiate and to adapt the pattern with its instantiation operations. Furthermore the tool provides the candidate of patterns that can be used in the next step. It is done by applying the refinement operations of the pattern.

- Consistency check of pattern combination and pattern refinement:
  When we combine several patterns or evolve the pattern into a new one, we should check if the new pattern has no inconsistency, in particular behavioral aspects. For example, the behavioral property of the new pattern should be satisfied with the old pattern when we change the old pattern into the new one. We will consider how
to provide the formal semantics of pattern behavior by using a formal method such as $\pi$ calculus.

- Developing pattern base:
  Pattern base is a kind of database system for patterns together with three types of operation on patterns and plays an important role on the supporting tool for pattern based software development. It will be implemented on an object base system such as PCTE/OMS (Portable Common Tool Environment / Object Management System). And embedding the support for re-doing design tasks into our prototype tool is also future study.

References


