Software Development Based on Software Pattern Evolution *

Takashi Kobayashi 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
E-mail: {tkobaya,saeki}@cs.titech.ac.jp

Abstract

This paper discusses a technique to model software patterns for supporting pattern based software development. Software development can be considered as the evolution of the artifacts to be produced. Software patterns are general structures that frequently appear in the artifacts and the patterns are also being evolved as the artifacts are being done. By specifying how to evolve software patterns as software processes progress, we can get a support for developing an artifact from the artifacts that were produced in the previous steps. In our approach, we consider that a software pattern consists of a pattern structure (a class diagram and/or an object diagram) and manipulation operations on the pattern structure. These operations are for pattern instantiation (applying a pattern to an actual problem) and for pattern evolution (evolving the artifacts of the previous steps into a new one). We model them with object-oriented technique encapsulating these operations into patterns.

Keyword: Software Pattern, Software Evolution, Object-Orientation

1. Introduction

Software development can be considered as producing various kinds of documents (called artifacts). Developers elicit requirements from their customers and/or users, and compose a requirements specification document. Based on the requirements specification, they do a design task to produce a design specification. The activity of coding is for transforming the design specification into a source code. Thus the process of evolving customers' requirements into a final product (i.e. the source code) can be considered as a software development process[5]. From this viewpoint of artifact evolution, the essential point of software development is the structural change of the artifacts that are being produced. Although the way of evolving the artifacts greatly depends on the application domain and on the steps of software development processes such as requirements analysis step, we can have the styles of artifact evolution that frequently appear and that are reusable to the other development processes.

For example, although they are not patterns of artifact evolution, Analysis Pattern[3] and Design Pattern[4], are general and abstract structures that frequently appear in past experiences in developing well-structured artifacts. 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. This example suggests to us that there are some guidelines or ways how to apply Design Patterns to an Analysis Pattern. Therefore we can consider that artifacts can have common structures in the development and that software patterns[1, 6, 8, 11] and artifact evolution can be formalized as rules of structural changes on the patterns.

In this paper, we consider software development processes as the processes of changing and evolving the artifacts, and propose the modeling technique for these processes by using evolution operations on the patterns. The rest of the paper is organized as follows. In the next section, we show the basic idea of pattern evolution in software development by illustrating Analysis Pattern and Design Pattern. Section 3 presents how to describe these evolution operations on software patterns.

We use object-oriented model to specify the evolution rules as operations to change the structure of the patterns. Basic operations for manipulating patterns will be listed in the section. As the artifacts to be developed are more complicated, the patterns to be used are also larger and more complicated. To model these patterns, we introduce a hierarchical decomposition technique. In this technique, a pattern comprise the combination of smaller patterns, and we sketch it in section 4. Finally we discuss this technique

*Published in the proceedings of the Sixth Asia Pacific Software Engineering Conference - APSEC'99.

and list up the future work.

2. Pattern Evolution in Development Processes

2.1. Software Development using Patterns

In the case that we use software patterns such as Analysis Patterns and Design Patterns to develop software systems, we decide an appropriate pattern from a pattern catalog, and instantiate and adapt it into a concrete artifact. After this step, based on the artifact developed by using the patterns, we produce a new artifact. We express the step of instantiating and adapting the selected patterns as the function ins and the step of producing the new artifact as the function $develop$. Figure 1 shows a development process from the viewpoint of these functions. The usual process of developing $artifact\#2$ from $artifact\#1$ (without using patterns) can be represented as follows;

$$artifact\#2 = develop(artifact\#1)$$

On the other hand, we can define the development process of the $artifact\#1$ using the patterns $pattern\#1$ as;

$$artifact\#1 = ins\#1(pattern\#1)$$

In the case of the product#2, we can get the similar expression, i.e. $artifact\#2 = ins\#2(pattern\#2)$ where the function $ins\#2$ is the step of instantiating and adapting the $pattern\#2$ to $artifact\#2$. We assume that there are some rules or guidelines for evolving and that they can be formally defined as a function $evol$. We can specify the relationships between the instantiation steps and between the patterns by using $evol$ respectively as follows.

$$ins\#2 = evol(ins\#1)$$
$$pattern\#2 = evol(pattern\#1)$$

We can also have the description of developing the $artifact\#2$;

$$artifact\#2 = evol(ins\#1)(evol(pattern\#1))$$

The above expression expresses a style of pattern based software development. In this development process, a developer selects a suitable pattern ($pattern\#1$)and instantiates it so that it is adaptable to his problem and so that he gets an artifact ($artifact\#1$). As the development process progress, he can obtain a next artifact (semi-)automatically by using evolution function $evol$. In this research, we aim at the technique for supporting the development of the artifacts in the next step, based on the patterns and their instantiations used in the current step.

In waterfall lifecycle model of software development, the artifacts are evolved from requirements analysis step to maintenance step through design, coding and test/debugging steps. We can consider software patterns in each step and Table 1 illustrates them. The patterns in a step are evolved in the latter steps.

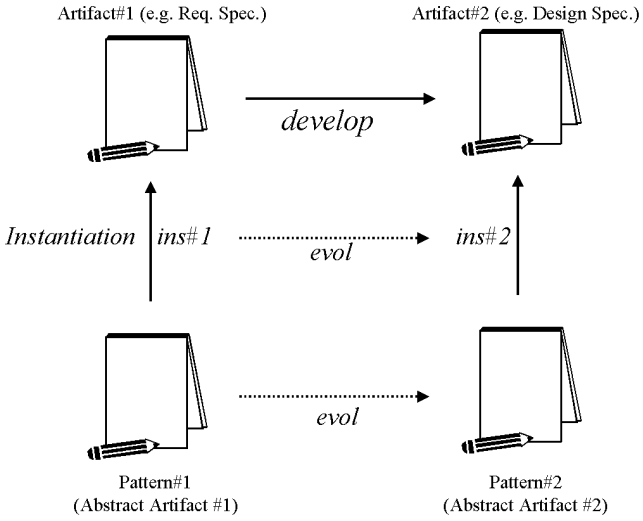

Figure 1. Software Development Based on Patterns

2.2. Example of Pattern Based Development

In this subsection, lets' illustrate a simple example of software patterns and its evolution. Figure 2 shows one of the simplified versions of Analysis Patterns 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.

For example, consider that we express the simple organizational structure shown in Figure 3 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 cor-

Requirement Analysis		Design		Coding	Test	Maintenance
Requirements Elicitation	Requirements Specification	Architectural Design	Detail Design			
-Interview Pattern -Questionnaire Pattern	-Use Case Pattern -Analysis Pattern	-Architecture Pattern	-Design Pattern	-Application Framework		-Impact Propagation Pattern

Table 1. Examples of Patterns Used in Each Step

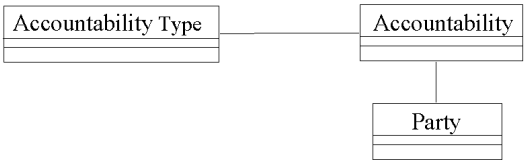

Figure 2. Party-Accountability Pattern (Simplified Version)

responding instances of Accountability (expressed with *** in the figure) have the links to "Professor" and "Student".

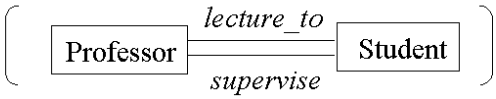

Figure 3. Specification

As mentioned above, it is necessary to adapt the structure of the patterns and to add some information to them according to the actual system to be developed, when we practically apply the patterns. We call these operations of adaptation and addition *instantiation operations*. What parts we should and can instantiate in a pattern depends on the pattern. These changeable parts are called *hot spots*. 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 3, as shown in Figure 4, 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").

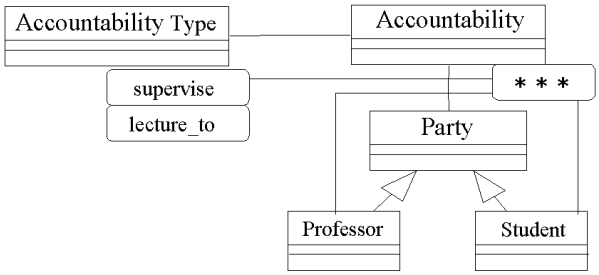

Figure 4. Instantiation of Party-Accountability Pattern

An allowable series of the instantiation operations depends on patterns. Thus we should specify the instantiation operations together with the structure of the patterns.

Lets' suppose that we proceed the design task of this example. To make use of this Party-Accountability Pattern in the structure of a design specification, we evolve Accountability Type and Accountability classes so that they possesses the mechanism to create instances (e.g. creating Accountability Type instances "supervise" and "lecture_to", etc.).

The evolution 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 evolution process is depicted in Figure 5. In the pattern, i.e. Figure 5 (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 design pattern.

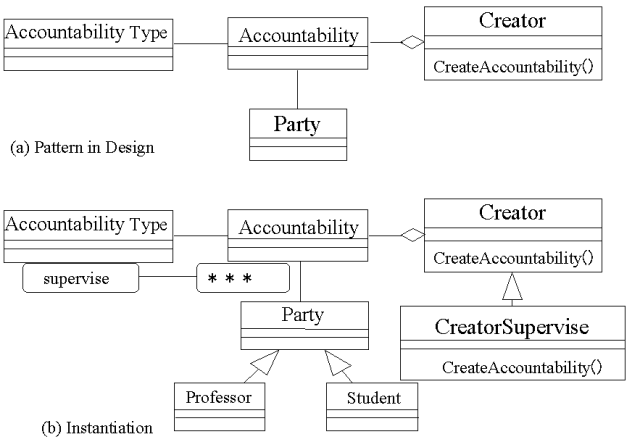

Figure 5. Evolution Example of Party-Accountability Pattern

The way how to evolve 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 evolution operations according to design strategies. We can also have different evolution operations if the pattern has different ways of instantiation.

3. Modeling Pattern Evolution

3.1. Modeling Patterns

In this section, we discuss how to model software patterns with instantiation operations at first. We frequently specify the structures of patterns with class diagram and/or object diagram. Thus we can define the instantiation operations as manipulation operations on class and object diagrams. Since these instantiation operations are specific to patterns, we can model a pattern with a pair of its structure and the instantiation operations on it as follows;

Software Pattern
= Pattern Structure + Instantiation Operation

The detailed definitions of the instantiation 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 specified as attributes (a set of instance variables) and the instantiation operations are considered as methods of a pattern class. See Figure 6.

3.2. Modeling Instantiation Operations

An instantiation operation, i.e. filling hot spots in a pattern, consists of manipulations of a class or object diagram. The examples of the basic manipulation on a diagram are adding a subclass (addSubclass), adding a method to a class, creating an object and so on. The list of the basic manipulations will be listed up in the subsection 3.3. For example, with Java-like language, we can express a part of the instantiation operations of Party-Accountability Pattern shown in Figure 2 as follows;

```
Party-Accountability_AnalysisPattern{
 Class Accountability_Type ;
 Class Accountability ;
 Class Party ;
 ...

 Instantiate_Party(ConcreteParty-name) {
    ConcreteParty =
         CreateClass(ConcreteParty-name);
    Party.addSubClass(ConcreteParty);
 }

 Instantiate_Accountability(Relationship,
                      Participant#1,
                      Participant#2) {
   x = new Accountability_Type(Relationship) ;
    y = new Accountability() ;
    y.addAssociation(x) ;
    y.addAssociation(Participant#1) ;
    y.addAssociation(Participant#2) ;
}
```

where y.addAssociation(x) is a manipulation operation for adding an association between x and y. To make the accountability "supervise", for an instance we create an instance of Party-Accountability_AnalysisPattern class and send it a message Instantiate_Accountability('supervise', Professor, Student) . As mentioned in this subsection, the instantiation functions $ins\#1$ and $ins\#2$ in Figure 1 can be defined as a series of manipulations for modifying the class or object diagram that expresses a pattern structure.

3.3. Manipulation on Patterns

The typical examples of basic manipulation operations on class and object diagrams are listed up below. The instantiation operations and evolution on patterns can be defined.

- createClass('Name')
  Creating a class whose name is 'Name'. (Constructor of Class)
- $Class_1$.addSubclass($Class_2$)
  Adding a class $Class_2$ as a subclass of $Class_1$.
- $Class_1$.addSuperclass($Class_2$)
  Adding a class $Class_2$ as a superclass of $Class_1$.
- $Class_1$.addAssociation($Class_2$, 'Name')
  Adding an association 'Name' between $Class_1$ and $Class_2$.
- $Class_1$.addAggregation($Class_2$)
  Adding an aggregation 'Name' between $Class_1$ and $Class_2$.
- createMethod('definition')
  Creating a method whose code is 'definition'. (Constructor of Methods)
- $Class_1$.addMethod($Method$)
  Adding a method $Method$ to $Class_1$
- $Class_1$.addAttribute($Attribute_1$)
  Adding an attribute $Attribute_1$ to $Class_1$.

These basic operations can be considered as constructors of class and object diagrams. Note that the effects of executing the operations follow the usual semantics of object-oriented model. For example, when we execute the following sequence of operations;

```
superclass = createClass('A') ;
subclass = createClass('B') ;
superclass.addSubclass(subclass) ;
m1 = createMethod('method1' + MethodBody1) ;
m2 = createMethod('method2' + MethodBody2) ;
superclass.addMethod(m1) ;
superclass.addMethod(m2) ;
m1new =
  createMethod('method1' + MethodBody1New) ;
subclass.addMethod(m1new) ;
```

we can get the subclass B that has the method1 and the method2. However the body of the method2 of the subclass B is MethodBody1New, while that of the class A is MethodBody1. This results from the sematic rule of inheritance and overloading of methods between a superclass and its subclasses. Note that the operator + stands for the concatenation of string data. For example, the expression 'fooMethod' + fooMethodBody results in 'foomethod(int A) { return A }' where the string '(int A) { return A }' is assigned to the variable fooMethodBody.

3.4. Modeling Evolution

The way how to evolve patterns, i.e. the function $evol$ in Figure1, is also specific to a pattern, and can be specified as a series of manipulation operations on a pattern structure (class or object diagram) in the similar way to the definition of instantiation operations. Pattern evolution causes not only the change of a pattern structure but also the change of instantiation operations ($ins\#2 = evol(ins\#1)$ in Figure 1). Therefore adding methods and/or modifying codes in the methods in a pattern class is also necessary for a pattern evolution.

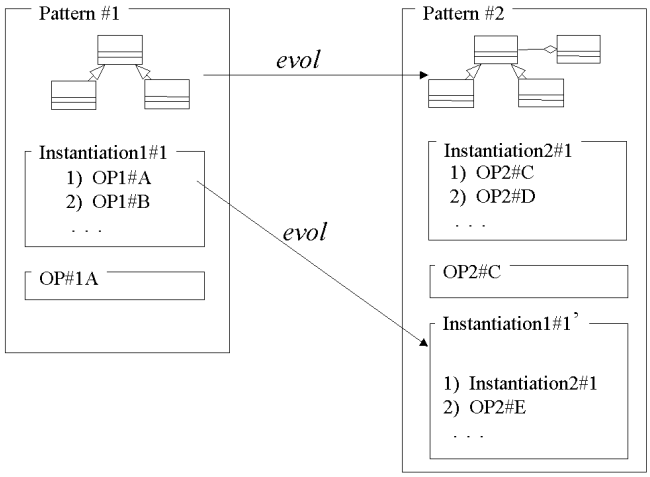

Figure 6. Pattern Evolution

Figure 6 illustrates these two changes of a pattern structure and of the methods as instantiation operations. Thus we can define a pattern evolution as

> Pattern Evolution = Evolution of a Pattern Structure + Evolution of Instantiation Operations.

In the example of Figure 5, we have the following changes; 1) change of a pattern structure : the class Creator that generates Accountability instances is connected to Accountability class, and 2) change of an instantiation operation Instantiate_Accountability : the Party classes participating in generated Accountability instances are added and the method for associating the accountability with the parties is added. 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. In this example, we capture an inheritance relationship as an evolution relationship. That is to say, a pattern class is evolved into pattern that is its subclass.

```
Party-Accountability_DesignPattern
     extends Party-Accountability_AnalysisPattern{

 Party-Accountability_DesignPattern(){
```

```
   super();

   /* Connecting Creator to Accountability Class */
   creator = createClass('Creator') ;
   Accountability.addAggregation(creator) ;

   /* Add an abstract method CreatorAccountability
                              (Interface only) */
   m = CreateMethod(
        'createAccountability(Party p1,
                        Party p2) { }') ;
   Creator.addMethod(m) ;
 }

 /* Making up Relationship the relationship
       between Participant#1 and Participant#2  */
 Instantiate_Accountability(relationship,
                            participant#1,
                            participant#2){
   super.Instantiate_Accountability(relationship,
                                    participant#1,
                                   participant#2);

   /* Connecting ConcreteCreator as a subclass    */
   concreteCreator =
            CreateClass('Creator' + relationship);
   Creator.addSubclass(concreteCreator);

   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      */
   m = createMethod('createAccountability('
                    + participant#1 + ' p1,'
                    + participant#2 + ' p2) {
           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 superclass 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 evolving the constructor of the analysis pattern in the design pattern, and it is one of the evolution directions or strategies. If we have a different evolution direction, we should define another evolution operation corresponding to this direction.

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.add(p2), we establish the two links of Accountability instances to Party instances p1 and p2 respectively. When we operate the instantiation 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 instance 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'. is assigned

3.5. Meta Model of Patterns

Figure 7 shows the entities and their relationships that a software pattern consists 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 in a support tool for pattern based software development.

In the figure the Role class expresses what roles the classes and the objects appearing in a pattern play.

4. Combining Patterns and Their Evolution

The example that we have used until this section was a very simple pattern and was used just for explaining our modeling technique. In actual development processes, much larger and more complicated patterns are used because the size of the software systems to be practically developed are large. We frequently combine many patterns into a larger pattern or application frameworks[7, 9, 10] (simply framework) are applied according to problem domains. A framework can be considered as a kind of patterns, a coarse-grained or large size pattern specialized in a certain problem domain. To specify a large-size pattern or a framework comprehensively, we hierarchically decompose it into a set of finer grained patterns and it is defined by using the definitions of the patterns in the lower level of the hierarchy. More precisely, the pattern structure, the instantiation operations and the evolution operations are specified by using those of the lower level patterns. Figure 8 shows this hierarchical decomposition technique to specify large and complicated patterns. The bottom level of the hierarchical structure has operations to operate on each element of a class and an object diagram, such as

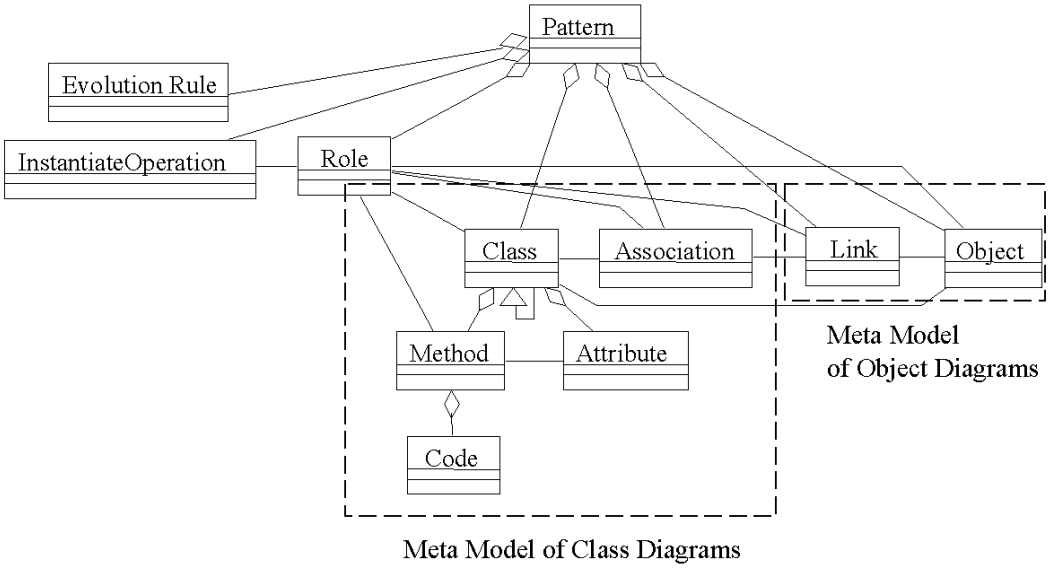

Figure 7. Meta Model of Software Patterns

adding a subclass, adding a method and so on, which were listed up in the subsection 3.3. We can collect fine-grained patterns reusable for constructing coarse-grained patterns.

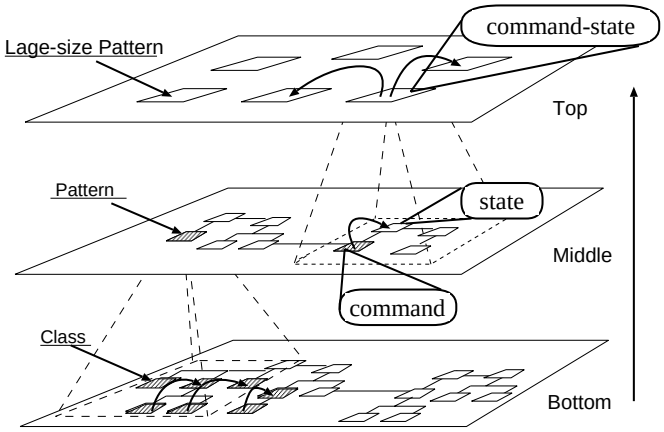

Figure 8. Hierarchical Structure on Patterns

Consider a simple example pattern that consists of Command pattern and State pattern of G-O-F design patterns. This pattern is often used to design interactive editors. Assume that we instantiate a new command that manipulates edited objects, such as text and model its implementation by using a state transition machine. Since the state transition machine is designed based on State Pattern, The effect of instantiating the new command on the Command Pattern should be propagated to the State Pattern. Thus an instantiation operation of the pattern invokes the instantiation operation of Command pattern and then performs the operation of State patterns. How to combine the two invoked operations depends on the relationship between the two patterns – Command and State. We can specify the instantiation operation of the higher level pattern as sequential invocations as follows;

```
Command-State_Pattern
  extends Command_Pattern and State_Pattern{
 ....
 /* Definition of Pattern Structure */

 Command-State_Pattern(){
   ...
   command-pattern = new Command_Pattern() ;
   ...
   state-pattern = new State_Pattern() ;
   ...
 }

 Instantiate_Command(Command){
   ...
   command-pattern.Instantiate_NewCommand(
                                  Command.name) ;
   ....
   state-pattern.Instantiate_NewState(state) ;
   ....
 }
}
```

The operations Instantiate_NewCommand and Instantiate_NewState are the instantiation operations on Command and State patterns respectively. Since the Command-

State_Pattern consists of Command and State patterns, it is defined as a subclass of Command and State. Therefore we can refer to the pattern structures of Command and State and use their operations to define Command-State_Pattern.

5. Conclusion

In our approach, we considered that a software pattern consists of a pattern structure (class diagram and object diagram) and manipulation operations on the pattern structure. These operations are for pattern instantiation (applying a pattern to an actual problem) and for pattern evolution (evolving the artifacts of the previous steps into a new one). We modeled them with object-oriented technique encapsulating these operations into patterns, and described patterns with Java-like object-oriented language.

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

- Supporting tools of evolution 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 evolution operations of the pattern.

- Consistency check of pattern combination and pattern evolution:
  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 evolve 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 π calculus.

- Developing pattern base:
  Pattern base is a kind of database system for patterns together with instantiation and evolution operations 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[12].

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture , A System of Patterns*. John wiley & Sons, 1996.

[2] G. Florijn, M. Meijers, and P. Winsen. Tool support for object-oriented patterns. In *ECOOP'97 - Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science, pages 472–495, 1997.

[3] M. Fowler. *Analysis Patterns: Reusable Object Modeling*. Addison Wesley, 1997.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Pattern: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] T. Katayama. A theoretical framework of software evolution. In *Proceeding of IWPSE98*, pages 1–5, 1998.

[6] R. Martin, D. Rehle, and F. Buschmann, editors. *Pattern Languages of Program Design 3*. Software Patterns Series. Addison Wesley, 1998.

[7] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1995.

[8] C. Schmidt, editor. *Pattern Languages of Program Design 1*. Software Patterns Series. Addison Wesley, 1996.

[9] Taligent Inc. Leveraging object-oriented frameworks. A Taligent White Paper, 1993.

[10] Taligent Inc. Building object-oriented frameworks. A Taligent White Paper, 1994.

[11] J. Vlissides, J. Coplien, and N. Kerth, editors. *Pattern Languages of Program Design 2*. Software Patterns Series. Addison Wesley, 1996.

[12] L. Wakeman and J. Jowett. *PCTE The Standard For Open Repositories*. Prentice Hall, 1993.