Evolutionary Patterns of Design and Design Patterns

Mikio Aoyama
Department of Information and Electronics Engineering
Niigata Institute of Technology
1719 Fujihashi, Kashiwazaki 945-1195, Japan
mikio@iee.niit.ac.jp

Abstract

Design patterns are considered well-formed language to represent software design. In this article, we propose several notions of design evolution and discuss the patterns of design evolution in terms of design patterns. First, we propose a classification of design patterns into pattern families. To model the design patterns and the evolutionary relationship between them, we propose a set of graphical notations of design patterns and their evolution, namely pattern type diagram and pattern evolution diagram. Then, we analyze evolutionary patterns of design within a family and across multiple families of design patterns. Based on the patterns of design evolution, we can navigate designers to select and compose design patterns for solving complex problems with design patterns. An example illustrates evolutionary design of framework by composing design patterns.

Keywords

Design evolution, software pattern, pattern family, component composition, and software architecture.

1. Introduction

Design for software evolution is interesting subject. Yet, it would be an integral task in software development since adaptation to rapid change of requirements and operating environments is the competitive edge in today's business climate. In this article, we propose a methodology for solving complex problems with evolutionary patterns of design in terns of design patterns.

Alexander's *pattern language* is a language to explicitly describe implicit design knowledge and best practices [1]. The idea was imported to software design [9] and has rapidly spread to various scenes in software development.

Like Alexander's pattern language, design patterns are considered well-formed language to represent software design. With design patterns, we can analyze design as well as design evolution. However, conventional works on design patterns focus on documenting individual patterns into catalogues. Few papers discuss the analysis and synthesis of design in terms of design patterns. In this article, we propose a methodology to represent and analyze design evolution in terms of design patterns [3]. To model the design patterns and the evolutionary relationship between them, we provide a set of graphical notations of design patterns and their evolution, namely pattern type diagram and pattern evolution diagram.

Another goal of this article is to provide a methodology for mining, selecting and applying design patterns. Following the first design pattern book [9], many so-called pattern catalogues have been published [5, 10]. However, this phenomena created a burden to pattern practitioners. They need more time to explore a forest of patterns to find an appropriate pattern. Although a systematic way to use patterns, such as *pattern system*, was suggested [5], little research has been done on the subject. In this article, we propose a notion of pattern family and a method to classify the patterns into families. With the pattern family, we found the evolutionary patterns of design within a family and across multiple families of design patterns. Based on the patterns of design evolution, we can navigate designers to select and compose design patterns for solving complex problems with design patterns. An example illustrates evolutionary design of framework by composing design patterns.

The structure of this article is as follows. In section 2, we discuss principles of design patterns and propose a graphical notation of design patterns, namely pattern type diagram. In section 3, we explore the concept of pattern family with an example of Factory patterns. To represent the structure of design evolution, the pattern evolution diagram is proposed in section 4. In section 5, a set of patterns of design evolution is formalized with an example of evolution of Factory patterns. As an example, an evolutionary design of pattern-based framework is illustrated in section 6. Related works and conclusions follow in section 7 and 8, respectively.

2. Principles of Design Patterns

2.1 Model of Design Patterns

We view design patterns a set of mapping from a problem space to a solution space as illustrated in Figure 1. As indicated, problem space is not a single space but is composed of multiple problem spaces. So, if we can decompose a complex problem space into a set of primitive problem spaces from R_1 to R_n, then we can find multiple solution spaces from S_1 to S_n through the primitive pattern spaces from P_1 to P_n. The total solution, S, against to the original problem, R, can be elaborated by composing the primitive solutions from S_1 to S_n. Although the real problems are much more complicated, the solution process abovementioned can provide a principle of design patterns for solving complex problems.

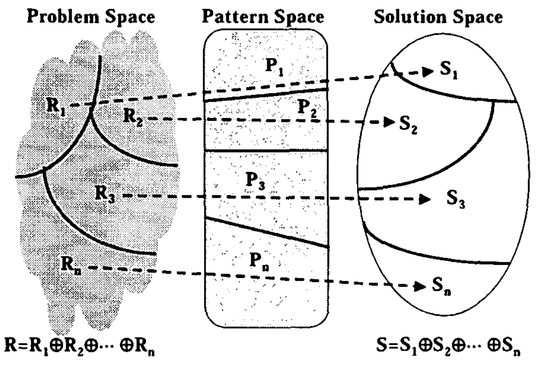


Fig. 1 Principle of Design Patterns

2.2 Software Design with Design Patterns

The abovementioned principle of software design with design patterns can be represented as follows.

(1) Decompose problem space, R, into n primitive problem spaces:

$$R=R_1\oplus R_2\oplus\cdots\oplus R_n$$

(2) Find design pattern, P_i, for each primitive problem space, R_i:

$$P_i=F(R_i, S_i) \text{ for } i=1\cdots n$$

(3) Compose n primitive solution spaces, S_i, by reordering them in an appropriate sequence:

$$S= S_1\oplus S_2\oplus\cdots\oplus S_n$$

Here $\oplus$ denotes disjoint.

It should be noted that design is based on the decomposition of problem space, instead of solution space. Since design activities start in problem space, it's desirable to navigate designer in problem space. Although, in general, the decomposition of problem space is arbitrary and the primitive problem spaces may not be disjointed, the mapping from problem space to solution space through design patterns is an integral activity of design. Therefore, the mission of this research is to extend the principle to be useful for practitioners in their daily work.

2.3 Model of Design Patterns

The power of design pattern lies in its representation method, that is, pattern language. Although there are some variants, the following 3-tuple is essential ingredient in any design patterns. Thus, we use the following representation.

$P= (R, S, F)$

Here, R, S, and F respectively denote problem, solution, and a set of forces. The forces include design goals and constraints. Thus, forces bind R and S.

To represent patterns, conventional pattern catalogues employ narrative descriptions in a natural language with diagrams such as UML class diagram and sequence diagram. However, to analyze the design evolution with patterns, we need a more formal and high-level representation. Thus, we propose a model of design patterns.

We view a pattern an artifact of design. Thus, a pattern is a *type of design*. Here, we propose a simple diagram notation, *Pattern Type Diagram* (*PTD*), to represent one pattern with a box as exemplified in Figure 2. This is an extended representation of the class diagram of UML [4] or an extension of type diagram of Catalysis [6]. The box consists of four parts, namely, name, problem (R) architecture of solution (S), and protocol to the pattern.

The third box is important since it represents the abstract structure of solution. In general, the class diagram of a design pattern consists of two layers. For example, Figure 3 illustrates the class diagram of Abstract Factory pattern. The upper layer represents the architecture of the pattern in terms of collaboration among abstract classes and interfaces. The lower layer represents implementation examples. We view design pattern as a parameterized design. Thus, in the third box of the PTD, we employ an abstracted representation of the upper layer of class diagram for representing the architecture of solution.

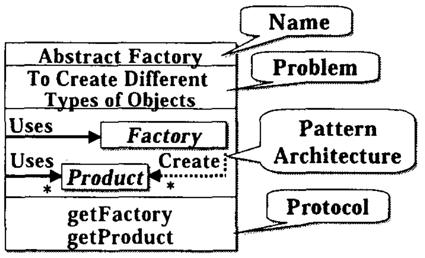


Fig. 2 Patterns Type Diagram

3. Pattern Family

3.1 Notion of Pattern Family

We propose a concept of *pattern family* that is a design level notion of program family [15]. A pattern family is a set of patterns sharing the same central *forces*, or design goals and constraints. Although each design pattern is elaborated to meet different design forces, there are similar but slightly different design patterns sharing primary design forces.

For example, let's take a look at factory patterns. There are several different Factory patterns including Abstract Factory, Factory Method, Builder, Prototype, and Singleton [9]. These patterns do the same thing, i.e. creating one or multiple objects, but in different manners as illustrated in Figure 4. Abstract Factory pattern is intended to create different objects of different structure, while Prototype pattern is to create copies of the same structure. We call them Factory pattern family.

3.2 Structure of Pattern Family

A family of design patterns can be structured along with the relationship among patterns as indicated in Figure 4. Variation of forces, or design goals and constraints, creates the variation of patterns. However, we can observe these patterns share the same central mission named primary forces. Thus, we can assume there is a *base pattern*, P_0, in a family, which meets the central forces.

$P_0=(R_0, S_0, F_0)$

Here, R_0, S_0, F_0 respectively denote the base problem, base solution and a set of primary forces.

The base pattern can be regarded as the most general solution in the family.

From P_0, we can create a set of *derived patterns*, P_i, depending on the variation of the problems, solutions and forces.

$P_i=(R_i, S_i, F_i)$

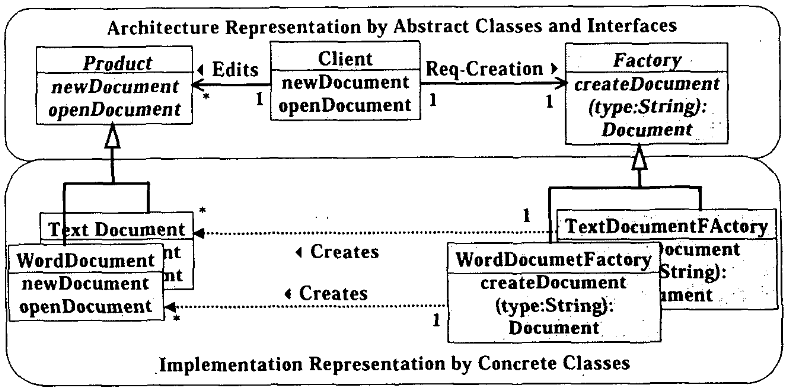


Fig. 3 A Class Diagram of Abstract Factory Pattern

4. Patterns of Design Evolution

4.1 Two Types of Design Evolution

Like an isomorphic relationship of software evolution [12], we define design evolution with a simple isomorphic relationship between patterns as illustrated in Figure 5. Here, we classify design evolution into two categories, that is, *intensive evolution* and *extensive evolution*.

(1) Intensive evolution is caused by some internal change of problem space, from R_1 to R_1', such as requirements change, bug fix and design improvement.

(2) Extensive evolution is caused by extension of problem space, from R to R+Re. Re is exemplified by adding new requirements and accommodating new operating

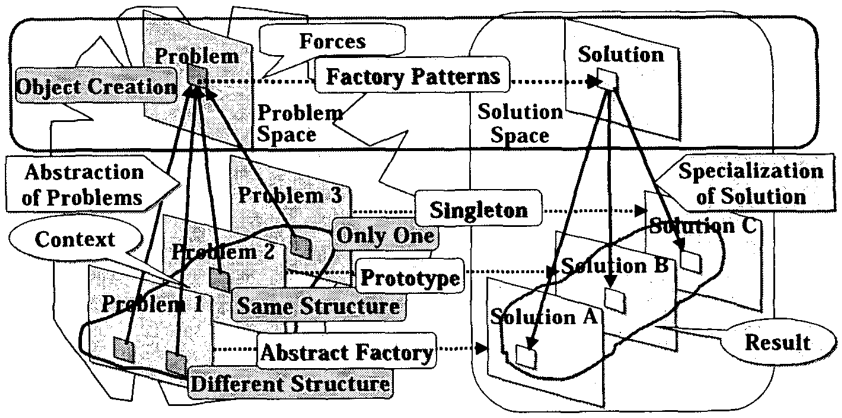


Fig. 4 Example of Pattern Family: Factory patterns

environment.

With design patterns, we represent the design evolution in terms of the relationship among design patterns. The two types of evolution indicate the two types relationships among design patterns: *intensive* and *extensive*. And, it should be noted that finding appropriate design patterns and composing them are the key tasks of designing software evolution.

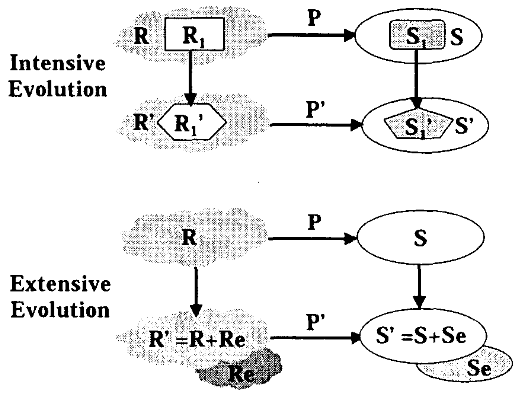


Fig. 5 Two Models of Design Evolution

4.2 Pattern Evolution Operations

To represent the intensive and extensive evolution of patterns precisely, we introduce *Pattern Evolution Operations* (*PEO*). Similar to software architecture [18], the architecture of the design pattern is defined with a set of components and connectors connecting them. Thus, evolution may emerge in either components or connectors, or both. We can define a set of primitive operations on components and connectors consisting of the design patterns as illustrated in Figure 6.

(1) *Replacement operation*: A primitive operation replaces a component with a set of components: rep (x, y) indicates replacement of component x with y. If y=ϕ, it indicates to eliminate component x.

(2) *Connection operation*: A primitive operation connects components in a cascading form. Thus, connection may arise either before a component or after a component along with the connection path. Connection operation is represented by 4-tuple, con (x, y, c, z). If z="pre", it connects component x before y with connector c. If z="post", it connects components x after y with connector c.

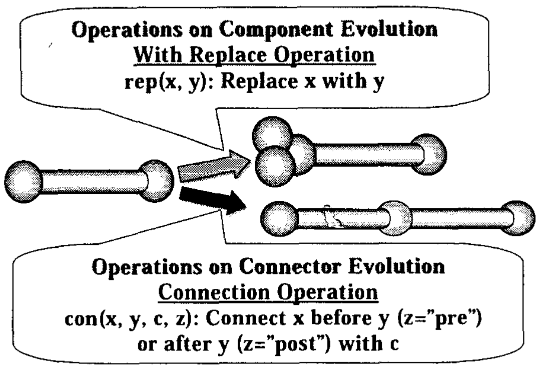
4.3 Pattern Evolution Diagram

Design evolution is an organized collective changes to design. Since design patterns represent design, we can view the relationship between two design patterns as *a design evolution*. To represent the evolution relationship among the design patterns, we need to represent direction of evolution and structure of collective changes. Thus, we employ a directed graph to represent the direction of evolution, and an ordered set of PEOs to represent the structure of collective changes as illustrated in Figure 7. We call the diagram *Pattern Evolution Diagram* (*PED*). A PED consists of the following entities:

(1) A set of patterns comprising nodes: each pattern is represented in a PTD as illustrated in Figure 2.

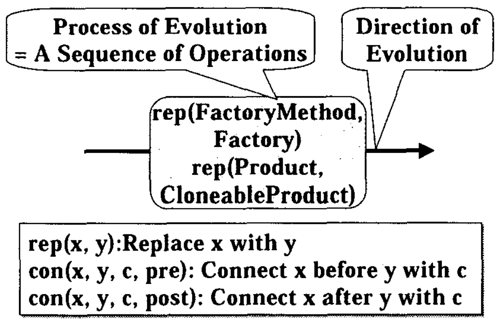
(2) Evolution path is a directed graph represented by arrow-headed line.

(3) Evolution Relationship (ER) indicates the difference between two patterns in terms of design. The ER is represented by an ordered set of PEOs in a rounded box on the evolution path as illustrated in Figure 7.



Operations on Component Evolution
With Replace Operation
rep(x, y): Replace x with y
Operations on Connector Evolution
Connection Operation
con(x, y, c, z): Connect x before y (z="pre") or after y (z="post") with c

Fig. 6 Pattern Evolution Operations



Process of Evolution = A Sequence of Operations
Direction of Evolution
rep(FactoryMethod, Factory)
rep(Product, CloneableProduct)
rep(x, y):Replace x with y
con(x, y, c, pre): Connect x before y with c
con(x, y, c, post): Connect x after y with c

Fig. 7 Pattern Evolution Diagram

5. Modeling Design Evolution with Patterns

With the PED, we can represent evolution of design in terms of design patterns.

5.1 Design Evolution within a Family

We start with a classification of design with pattern families. We can classify design evolutions into two types, *micro evolution* and *macro evolution*, as illustrated in Figure 8.

(1) Micro evolution indicates an evolution within a family such that: $E_1 = A_i \rightarrow A_j$. The patterns in this category share the same base problem and primary forces. There should be a base pattern, P_0, and a set of derived patterns, P_i, comprising the family. Those derived patterns can be regarded as micro design evolution from the base pattern.

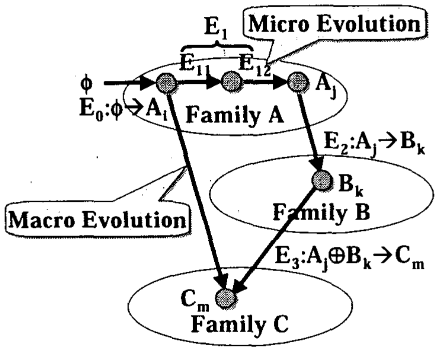


Fig. 8 Two Types of Evolutions of Design Patterns

(2) Macro evolution indicates an evolution across the families. This evolution emerges from the following causes:

1) Transition to another family: An evolution of a pattern may go beyond the boundary of a family and move to another family: $E_2 = A_j \rightarrow B_k$.

2) Composing patterns across multiple families to solve a complex problem: $E_3 = A_i \oplus B_k \rightarrow C_m$

3) Forming a new family: As a special case of E_1 or E_2, a new family may emerge: $E_4 = \phi \rightarrow A_i$.

5.2 Example: Micro Evolution of Factory Family

Figure 9 illustrates a set of patterns in the Factory pattern family and their evolution represented in PED. Note that we assume Abstract Factory as the base pattern since it is the most general solution to create objects.

It should be also noted that PED provides navigational paths in order to find appropriate design patterns within the family. Currently, we are developing design navigator with the PED.

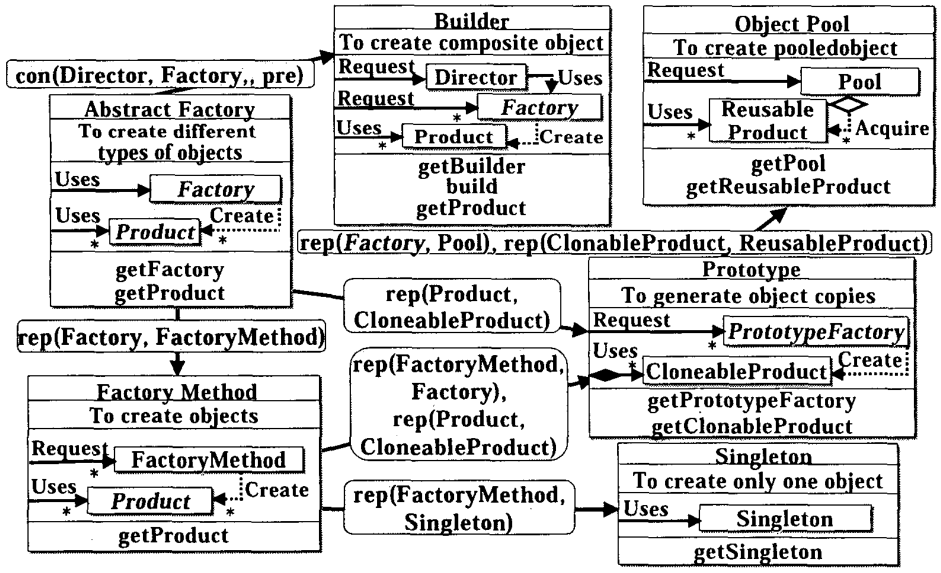


Fig. 9 Evolution in Factory Pattern Family

6. Application to Pattern-Based Framework Design

One practical application of the design patterns is to develop a application framework by composing patterns. We have exercised two examples of designing frameworks with patterns. One is file reader framework [20] and another is POS (Point Of Sales) framework [3].

We will briefly review the process of solving file reader framework along with the process described in Section 2.2. The description of the original problem and solution can be found in [20].

First, the original problem is decomposed into the following three primitive problems.

1) To handle recursively structured record.
2) To handle record of arbitral length until the end of record is detected by readObject method.
3) Store the record into corresponding object depending on the input record structure.

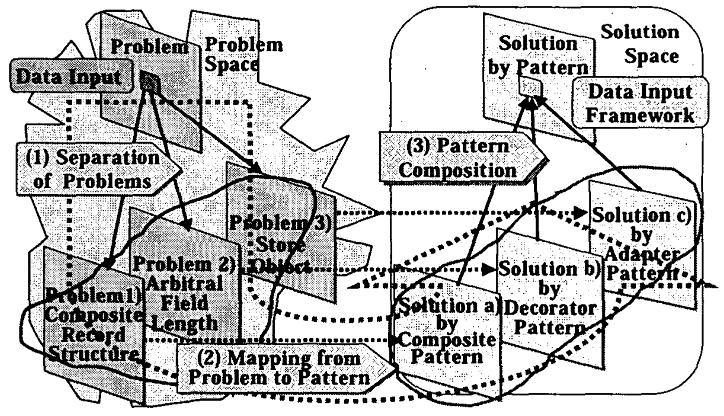

Problem Space
Problem
Data Input
(1) Separation of Problems
Problem 1) Composite Record Structure
Problem 2) Arbitral Field Length
Problem 3) Store Object
(2) Mapping from Problem to Pattern
Solution Space
Solution by Pattern
Data Input Framework
(3) Pattern Composition
Solution a) by Composite Pattern
Solution b) by Decorator Pattern
Solution c) by Adapter Pattern

Fig. 10 Decomposition of Problem Space and Mapping to Solution Space by Design Patterns

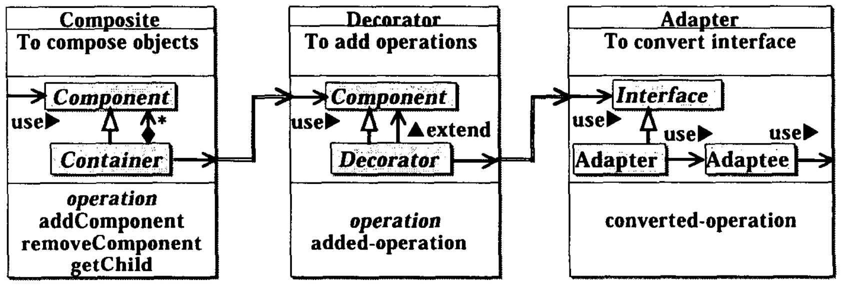


Fig. 11 Composition of Design Patterns in a Cascade Pattern

After mining patterns, we identified three patterns, Composite, Decorator and Adapter, to respectively solve each of the above primitive problems 1), 2), and 3) as illustrated in Figure 10.

By composing three patterns of a) b) and c) in Figure 10 with Cascade pattern [3, 8], we can build the file reader framework as illustrated in Figure 11.

7. Related Works

There are two related area of this work, formalization of design patterns and design evolution.

In the formalization of design patterns, there are tow categories of approaches in modeling patterns. One category of works adopted object-oriented modeling techniques to represent design patterns. For example, Pree proposed a class-like box notation, called meta pattern, to represent a pattern [16]. He also suggested pattern-based design methodology by specializing patterns by inheritance relationship. D'Souza and Wills proposed Catalysis with a process and a set of sophisticated graphical notations [6]. The notations and methodology can be used to represent design patterns. The UML does not provide specific notation to represent design patterns, but suggests use of collaboration diagram [4]. These approaches may have some limitation to modeling design patterns. Although the specialization relationship can be observed between some patterns, such as the base pattern and the derived patterns in a family, the structure of the derived patterns may be different from the base pattern since the design patterns are open to the implementation.

Another category of work employed various kinds of formal methods including temporal logic and algebraic specification. For example, Helm et al. proposed a semi-formal language based on the first order logic [11]. Mikkonen proposed a formal representation of design patterns based on temporal logic [14]. His technique provides rigorous reasoning on temporal behavior of design patterns. Lano et al. adopted a version of Object Calculus, a formal language based on temporal logic [13]. These formalisms have some advantages to provide rigorous reasoning and transformation of design patterns. However, since they are textual formal languages, they provide little intuitive understanding on the structure and relationship among multiple design patterns. Yet, it's hard to use these formalisms to provide concrete design guidelines to the practitioners.

In the area on evolution of design and design patterns, few works have been reported.

8. Conclusions

In this article, we proposed a set of concepts, representations, and methodology to understand design evolution in terms of design patterns. Like patterns in natural evolution [7], we see patterns of design evolution in terms of evolution of design patterns.

Although the proposed framework is relatively simple, the representation method provides an intuitive understanding both design patterns and design evolution. As briefly illustrated in the design exercise of file reader framework, this method can be used to navigate designers to find appropriate design patterns and compose them in order to solve complex problems such as building application frameworks with design patterns.

We may employ more formal way to represent and analyze patterns. However, we took a semi-formal approach since pattern language is inherently semi-formal because design is an exploratory process and open to experience.

It's also interested to explore the relationship of proposed approach with some related works such as classification of Composite patterns [17], patterns for component-based development [2], and multi-dimensional separation of concerns [19].

Currently, we are developing a pattern-based design navigation system. We will look at applying the system to assist designers for pattern-based framework design. We are also exploring the software evolution with the evolutionary patters of design patterns.

References

[1] C. Alexander, et al., *The Timeless Way of Building*, Oxford University Press, 1979.

[2] M. Aoyama, Component-Based Software Engineering: Can it Change the Way of Software Development?, *Proc. 20th ICSE, Vol. II*, Apr. 1998, pp. 24-27.

[3] M. Aoyama, Models of Software Patterns and the Patterns of Patterns Evolution, *Proc. IPSJ SIGSE Workshop*, No. 124-6, Oct. 1999, pp. 35-42 (In Japanese).

[4] G. Booch, et al., *The Unified Modeling Language User Guide*, Addison Wesley, 1999.

[5] F. Buschmann, et al., *Pattern-Oriented Software Architecture*, John Wiley & Sons, 1996.

[6] D. F. D'Souza and A. C. Wills, *Objects, Components and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1998.

[7] N. Eldredge, *The Pattern of Evolution*, W. H. Freeman and Company, 1999.

[8] T. Foster and L. Zhao, Cascade, *J. of OOP*, Feb. 1999, pp. 18-24.

[9] E. Gamma, et al., *Design Patterns*, Addison-Wesley, 1994.

[10] M. Grand, *Patterns in Java*, Vol. 1, John Wiley & Sons, 1998.

[11] R. Helm, et al., Contracts: Specifying Compositions in Object Oriented Systems, *Proc. ACM OOPSLA/ECOOP '90*, pp. 169-180.

[12] T. Katayama, A Theoretical Framework of Software Evolution, *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE '98)*, Apr. 1998, pp. 1-5.

[13] K. Lano, et al., Formalising Design Patterns, *Proc. 1st BCS-FACS Northern Formal Methods Workshop*, Electronic Workshops in Computer Science, Springer-Verlag, 1996, http://www.cs.concordia.ca/~faculty/eden/precise_and_formal/.

[14] T. Mikkonen, Formalizing Design Patterns, *Proc. 20th ICSE*, Apr. 1998, pp. 115-124.

[15] D. L. Parnas, On the Criteria to be Used in Decomposing Systems into Modules, *CACM*, Vol. 15, No. 12, Dec. 1972, pp. 1053-1058.

[16] W. Pree, Design *Patterns for Object-Oriented Software Development*, Addison-Wesley, 1994.

[17] D. Riehle, Composite Design Patterns, *Proc. ACM OOPSLA 97*, Oct. 1997, pp. 218-228.

[18] M. Shaw and D. Garlan, *Software Architecture*, Prentice Hall, 1996.

[19] P. Tarr, et al., N Degree of Separation: Multi-Dimensional Separation of Concerns, *Proc. 21st ICSE*, May 1999, pp. 107-119.

[20] B. Woolf, Framework Development Using Patterns, M. E. Fayad, et al (eds), *Implementing Application Frameworks*, John Wiley & Sons, 1999, pp. 621-629.