JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6, No. 10, November-December 2007

A Classification of Design Pattern Evolutions

Jing Dong, Department of Computer Science, University of Texas at Dallas
Sheng Yang, Department of Computer Science, University of Texas at Dallas
Yongtao Sun, Information Technology Service, American Airlines

Abstract

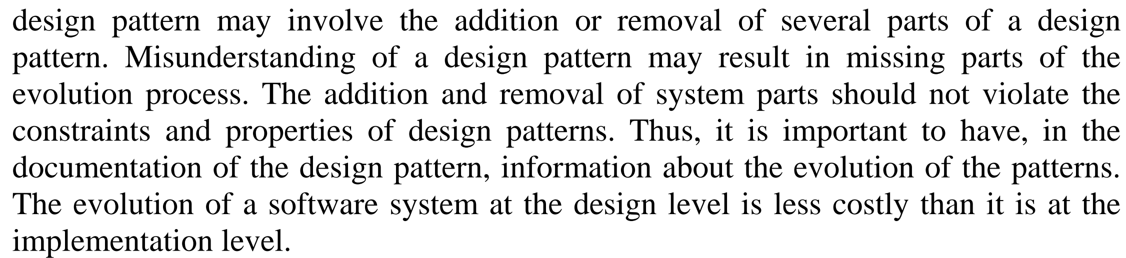
Design for change is one of the important themes of design patterns. Each design pattern normally embeds some specific ways for future changes. Currently, such evolution information is typically documented in each design pattern implicitly. In this paper, we classify design pattern evolutions into two levels: the primitive-level and pattern-level evolutions. Each pattern-level evolution is represented by several primitive-level evolutions. In this way, we can describe the possible changes of each design pattern in terms of a number of pattern-level evolutions.

1 INTRODUCTION

Software systems are generally not fixed and may evolve over time because of constant changes of user requirements, platforms, technologies and environments. Unlike other engineering products, such as automobiles and electronic devices, software systems are normally more amenable to changes. It can be a disaster if a single change may cause huge impact in the software systems. It is important to localize the changes such that minimum efforts are needed. This requires the initial designers of a software system to be aware of potential changes. Thus, the resulting software systems are flexible and agile to future evolutions.

Designing a software system is hard. Designing a changeable software system is even harder. Design patterns [8] capture expert design experience by partitioning software designs into stable part and changeable part. By separating and encapsulating both parts, the change impact of a software design can be minimized. One of the important goals of design patterns is design for change. Thus, most of design patterns encapsulate future changes that may only affect limited part of a design pattern. This evolution process can be achieved by adding or removing design elements in existing design patterns. In the document of each design pattern, however, the evolution information is generally not explicitly specified. When changes are needed, a designer has to read between the lines of the document of a design pattern to figure out the correct ways of changing the design. More importantly, the evolution process of a

Cite this article as follows: Jing Dong, Sheng Yang, Yongtao Sun: “A Classification of Design Pattern Evolutions”, in *Journal of Object Technology*, vol. 6, no. 10, November – December 2007, pp. 95-109 http://www.jot.fm/issues/issue_2007_10/article3/



design pattern may involve the addition or removal of several parts of a design pattern. Misunderstanding of a design pattern may result in missing parts of the evolution process. The addition and removal of system parts should not violate the constraints and properties of design patterns. Thus, it is important to have, in the documentation of the design pattern, information about the evolution of the patterns. The evolution of a software system at the design level is less costly than it is at the implementation level.

In this paper, we explicitly capture the evolution information of each design pattern in two levels: the primitive level and the pattern level. The primitive-level evolutions are the addition or removal of modeling elements, such as classes and relationships. The pattern-level evolutions characterize the recurring evolutions of design patterns based on the primitive-level evolutions. Thus, this classification allows us to describe the evolution process of each design pattern in terms of a number of pre-defined pattern-level evolutions. In this way, designers no longer need to extract the implicit evolution information from pattern document. Both levels of evolution processes are also amendable for automation with tool support.

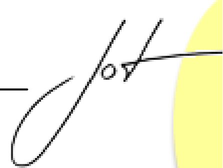
The reminder of this paper is organized as follows: the next section presents an example to motivate our approach. Section 3 describes the primitive-level and pattern-level evolutions. The last two sections are related work and conclusions.

2 MOTIVATING EXAMPLE

In this section, we use an example to present the motivation of our approach. Let us consider an example of the Abstract Factory pattern as shown in Figure 1. It originally includes two concrete factories: ConcreteFactory1 and ConcreteFactory2. These two concrete factories may create two families of concrete products. Thus, there are two create operations in each concrete factory class: createProductA and createProductB. The createProductA operation is used to create the ProductA family, whereas the createProductB operation is for the ProductB family. In particular, the createProductA and createProductB operations in the ConcreteFactory1 class are used to create the ProductA1 and ProductB1, respectively. Similarly, these two operations in the ConcreteFactory2 class are for ProductA2 and ProductB2, respectively.

In the Abstract Factory pattern, there are two possible ways to evolve the design. One way is to add a different kind of concrete product in each product family. For example, we may add the ProductA3 and ProductB3 classes in the ProductA and ProductB families, respectively. The other way is to put in a new product family: ProductC. Both ways of changes have some impact on the factory class hierarchy. The first way of evolution may require the addition of a new concrete factory class (ConcreteFactory3), whereas the second way may result in the insertion of a new create operation (createProductC) in each of the existing concrete factories. Both changes are shown in Figure 2 and Figure 3, respectively.

In current document of design patterns, unfortunately, such evolution information is not explicitly specified. Thus, the designers may not immediately identify, e.g., the two possible ways of evolutions in the Abstract Factory pattern. Even though the designers may know the two possible ways of changes, they still can make mistakes since the changes of factories and products are correlated, especially when the Abstract Factory pattern is applied in a large software design with many



classes. Such correlated changes may be mis-conducted. In the following sections, we introduce a solution to this problem in terms of two-level evolutions: primitive level and pattern level. As a result, the evolution processes of each design pattern can be classified in terms of a number of pattern-level evolutions.

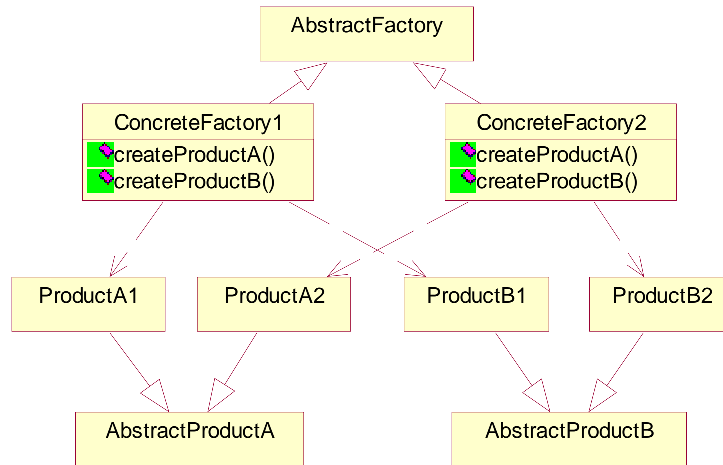


Figure 1 Abstract Factory Pattern with Two Kinds of Products Created by Two Concrete Factories

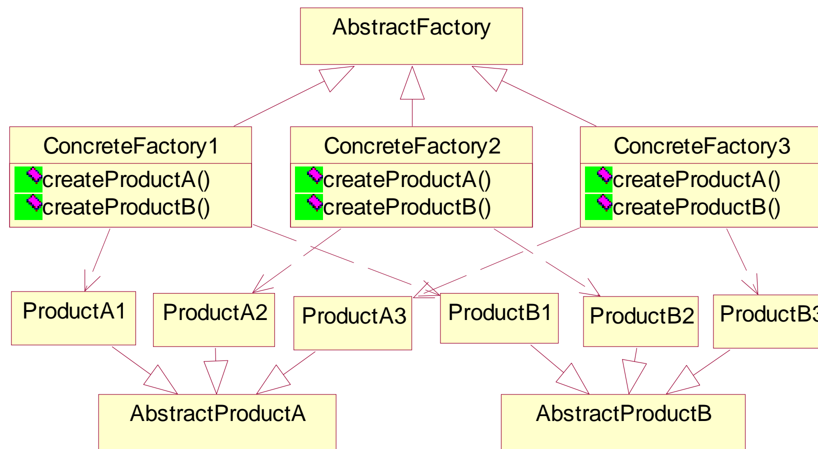


Figure 2 Abstract Factory Pattern with Three Kinds of Concrete Products Created by Three Concrete Factories

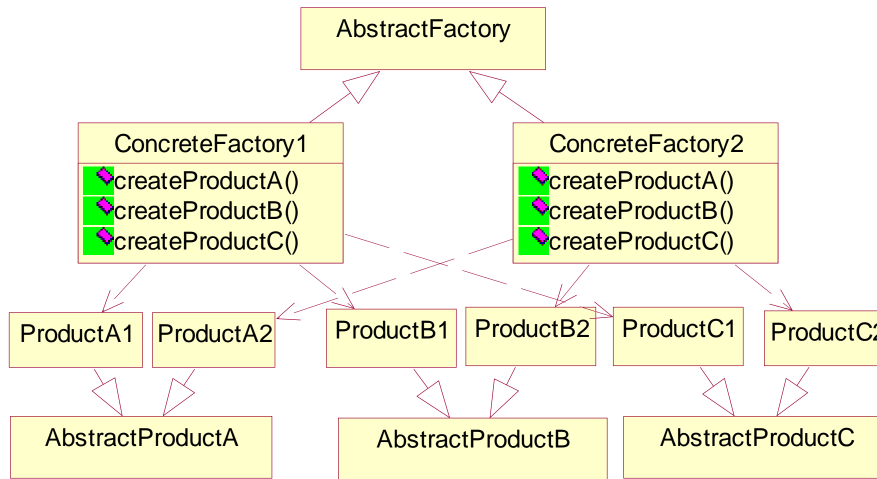


Figure 3 Abstract Factory Pattern with Three Kinds of Products Created by Two Concrete Factories

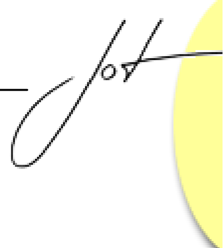
Model Elements	Parameter List	Descriptions
Class	className	Add or remove a class with name "className" into a pattern
Attribute	attributeName, className, type, accessibility	Add or remove an attribute with name "attributeName", type of "type", accessibility of "accessibility" into the class "className"
Operation	operationName, className, returnType, accessibility, para1, paraType1…	Add or remove an operation with name "operationName", type of "type", accessibility of "accessibility", and arguments list para1 with type "paraType1" into the class "className"
Association	className1, className2	Add or remove an association between classes "className1" and "className2" into a pattern
Generalization	child, parent	Add or remove a generalization relationship into a pattern, with subclass "child" and superclass "parent"
Aggregation	part, whole	Add or remove an aggregation relationship into a pattern, "part" class is a part of "whole" class
Composition	part, whole	Add or remove a composition relationship into a pattern, "part" class is a part of "whole" class
Realization	fromName, toName	Add or remove a realization relationship from class "fromName" to class "toName" into a pattern
Dependency	fromName, toName	Add or remove a dependency relationship from class "fromName" to class "toName" into a pattern

Table 1 Primitive-Level Evolutions

3 CLASSIFICATION OF DESIGN PATTERN EVOLUTIONS

In this section, we investigate different kinds of evolutions in design patterns and provide a classification of these evolutions. We describe these evolutions in terms of two-level evolutions: the primitive-level evolution and the pattern-level evolution.

The primitive-level evolution describes the basic transformations that can be performed during the evolution process of a design pattern. These basic transformations include the addition or removal of a modeling element, such as class, operation, attribute, association, generalization, aggregation, composition, realization,



and dependency. These basic transformations become the building blocks of the pattern-level evolution.

The pattern-level evolution characterizes the recurring evolution processes which occur in many design patterns. It is described in terms of a sequence of the basic transformations. Each design pattern may perform some of the pattern-level evolutions, which can be added in the document of the pattern. Thus, the designer may choose a potential pattern-level evolution and apply the corresponding transformations when changes are required.

Primitive-Level Evolutions

We identify nine modeling elements that can be added or deleted as the basic transformations in the pattern evolution processes. The general format of adding a modeling element is Add (ME (PL)). The model elements (ME) and the parameter list (PL) are shown in Table 1. For example, adding a class named "Leaf" can be specified: Add (Class (Leaf)). Similarly, the removal of a modeling element can be specified: Delete (ME (PL)). The replacement of a model element with another is conducted by first removing the modeling element and then adding a new modeling element. It can be defined: Delete (ME1 (PL1)) + Add (ME2 (PL2)).

Pattern-Level Evolutions

In this section, we characterize five pattern-level evolutions which are recurring in different design patterns. Each design pattern may encapsulate some of the pattern-level evolutions, which can be explicitly documented in the descriptions of the pattern. Thus, a designer can simply follow the prescribed evolution processes when the corresponding changes are needed. Note that we do not claim this is a complete list of all possible pattern-level evolutions. Nevertheless, new pattern-level evolutions can be easily added into the list specified by the primitive-level evolutions.

The first pattern-level evolution is called independent change which is a simple addition or removal of one independent class and the corresponding relationships between this class and the classes in the original pattern. This class is independent in the sense that the addition or removal of the class does not cause any effects on the existing classes of the design. This kind of pattern-level evolution can be expressed in the primitive level evolutions as follows[1]:

```
Add ( Class (className)) +
Add ( Relationship (className, existingClassName))
```

where className is the name of the class which is added into the pattern. Relationship includes association, generalization, aggregation, composition, realization, and dependency. The existingClassName is the name of the class from the original pattern. There may be multiple relationships added into the pattern with the addition of a class.

This kind of evolution appears in several design patterns as, for example, in the Mediator and Facade patterns. Figure 4 is the class diagram of the Mediator pattern

[1] Since the addition and removal have the same format and the only difference is the transformation names (Add and Delete), we omit the evolutions of removing modeling elements.

describing a possible application containing two ConcreteColleague classes. A potential evolution of this pattern application is to add a new ConcreteColleague class, which can be defined as the following transformation in terms of the primitive-level evolutions:

```
Add ( Class (ConcreteColleague)) +
Add ( Generalization (ConcreteColleague, Colleague)) +
Add ( Dependency (ConcreteMediator, ConcreteColleague))
```

where a new ConcreteColleague class is added with two new relationships: generalization and dependency. The generalization relationship is with the Colleague class. The dependency relationship is on the ConcreteMediator class. The result of this evolution is shown in Figure 5.

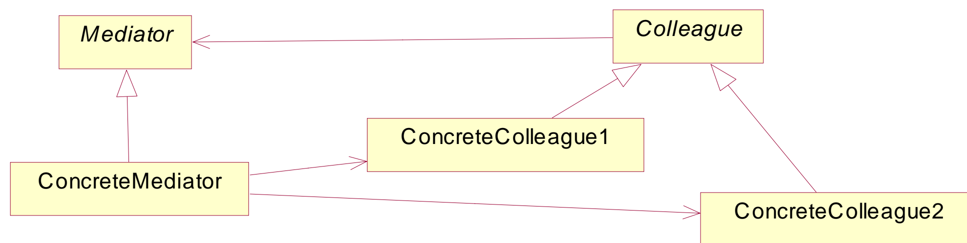


Figure 4 Mediator Pattern with Two Concrete Colleagues

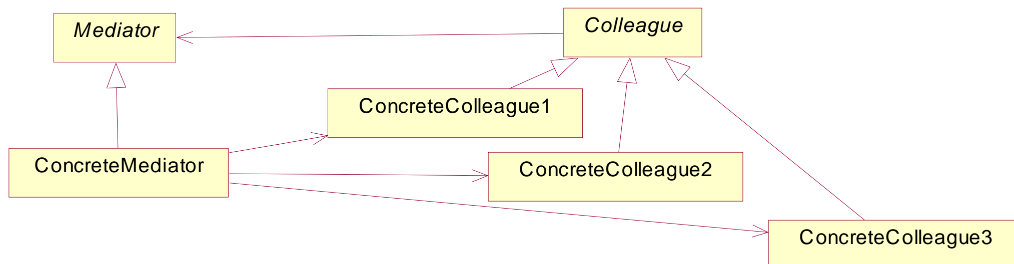
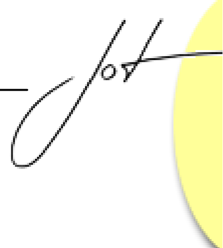


Figure 5 Mediator Pattern with Three Concrete Colleagues

The second pattern-level evolution is called packaged change which is the addition or removal of one independent class and the corresponding relationships between this class and the classes in the original pattern. In addition, certain attributes and/or operations of this class are added and removed accordingly. This kind of pattern-level evolution can be expressed in the primitive level evolutions as follows:

```
Add ( Class (className)) +
Add ( Relationship (className, existingClassName)) +
Add ( Attribute (attributeName, className, type, accessibility)) +
Add ( Operation (operationName, className, returnType, accessibility,… ))
```

where className is the name of the class which is added into the pattern. Relationship includes association, generalization, aggregation, composition,



realization, and dependency. The existingClassName is the name of the class from the original pattern. The attributeName is the name of the attribute of the class to be added whereas the operationName is the name of the operation of the class. There may be multiple relationships, attributes and/or operations added into the pattern with the addition of a class.

This kind of evolution can be found in several design patterns as, for example, in the Composite, Bridge, State, Strategy, Chain of Responsibility, and Observer patterns. Figure 6 is the class diagram of the Observer pattern describing a possible application containing one ConcreteSubject and two ConcreteObserver classes. A potential evolution of this pattern application is to add a new ConcreteObserver class (ConcreteObserver3) with its attributes (s1 and s2) as shown in Figure 7, which can be defined as the following transformation in terms of the primitive-level evolutions:

```
Add ( Class (ConcreteObserver3)) +
Add ( Generalization (ConcreteObserver3, Observer)) +
Add ( Attribute (s1, ConcreteObserver3, Undefined, private)) +
Add ( Attribute (s2, ConcreteObserver3, Undefined, private))
```

where a new concrete observer (ConcreteObserver3) is added with a generalization relationship between this new class and the Observer class. Two attributes (s1 and s2) of this new class are also added accordingly, where "Undefined" refers to the unknown types of these two attributes.

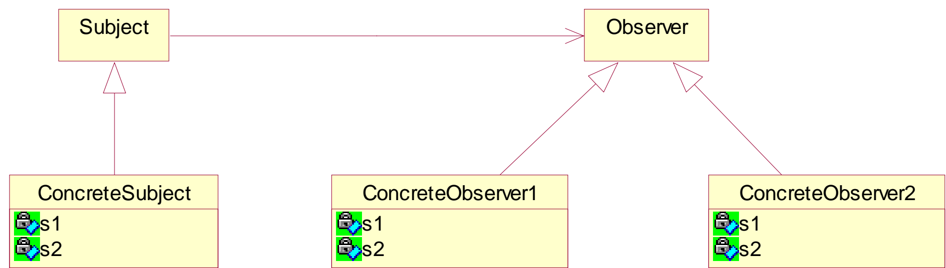


Figure 6 Observer Pattern with Two Concrete Observers

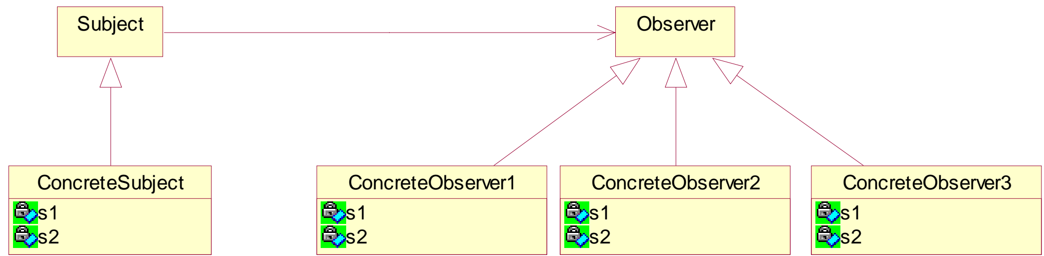


Figure 7 Observer Pattern with Three Concrete Observers

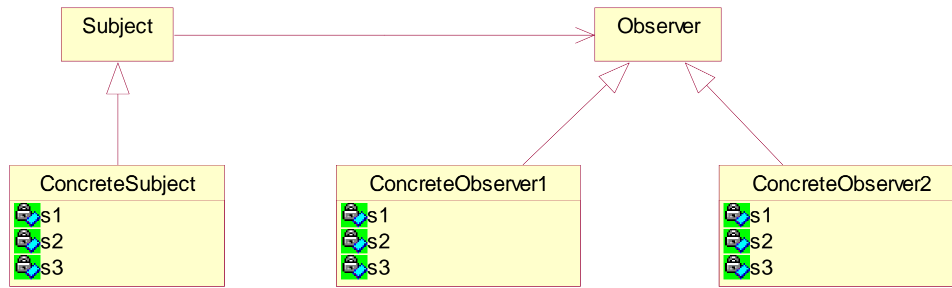


Figure 8 Observer Pattern with Three Attributes

The third kind of pattern-level evolution is called class group change which is the addition or removal of one attribute/operation in several different classes consistently. In this case, a certain set of classes, instead of a single class, are affected by the addition or removal of the attribute or operation. This kind of pattern-level evolution can be expressed in the primitive level evolutions as follows:

$$\sum_i \text{Add (Attribute(attributeName, className}_i\text{, type, accessibility))} +$$
$$\sum_j \text{Add(Operation(operationName, className}_j\text{, returnType, accessibility,...))}$$

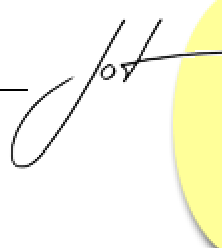
where attributeName is the name of the attribute of the adding class named $className_i$ whereas operationName is the name of the operation of the adding class named $className_j$. There may be multiple relationships, attributes and/or operations added into the pattern with the addition of a class.

This kind of evolution is common in several design patterns as, for example, in the Decorator and Observer patterns. Figure 6, for instance, shows an application of the Observer pattern with one ConcreteSubject and two ConcreteObserver classes. One potential evolution is to add one attribute called s3 as a new data to be observed by the observers. Thus, this attribute needs to be added in all ConcreteSubject and ConcreteObserver classes, which can be defined as the following transformation in terms of the primitive-level evolutions:

Add (Attribute(s3, ConcreteSubject, Undefined, private)) +
Add (Attribute(s3, ConcreteObserver1, Undefined, private)) +
Add (Attribute(s3, ConcreteObserver2, Undefined, private))

which indicates the attribute s3 is added into the ConcreteSubject, ConcreteObserver1, and ConcreteObserver2 classes. The resulting class diagram of this evolution is shown in Figure 8.

The fourth kind of pattern-level evolution is called correlated classes change which is the addition or removal of a group of correlated classes. When certain classes are added or removed, some other classes have to be added or removed accordingly. These correspondence relations are important since missing transformations may cause inconsistency. In addition, the corresponding relationships between this group of classes and other classes are added or removed. The attributes and operations of this group of classes are also added or removed. The addition or removal of this group



of classes may not affect the internal of other classes in the original design pattern applications. This kind of pattern-level evolution can be expressed in the primitive level evolutions as follows:

$$
\begin{aligned}
&\sum_{i} \text{Add (Class(className}_i\text{))} + \\
&\sum_{i,j} \text{Add (Relationship (className}_i\text{, className}_j\text{))} + \\
&\sum_{i,j} \text{Add (Attribute(attributeName}_i\text{, className}_j\text{, type}_i\text{, accessibility}_i\text{))} + \\
&\sum_{i,j} \text{Add (Operation(operationName}_i\text{, className}_j\text{, returnType}_i\text{, accessibility}_i\text{,...))}
\end{aligned}
$$

where Class($className_i$) refers to the ith class is added with the name of $className_i$. Similarly, relationships are also added between the classes $className_i$ and $className_j$. The $attributeName_i$ is the name of the ith attribute of the adding class named $className_j$ whereas $operationName_i$ is the name of the ith operation of the added class named $className_j$. There may be multiple relationship, attributes and/or operations added into the pattern with the addition of a class.

This kind of evolution can be seen in several design patterns as, for example, in the Builder, Factory Method, Command, Interpretor, Iterator, Visitor, Abstract Factory patterns. Figure 1 shows an application of the Abstract Factory pattern with two kinds of products (AbstractProductA and AbstractProductB). Each kind of products has two concrete products: ProductA1/ProductB1 and ProductA2/ProductB2, respectively. Thus, there are two concrete factories: ConcreteFactory1 and ConcreteFactory2. A potential evolution can be the addition of a new kind of concrete products (ProductA3 and ProductB3). This requires the addition of a new concrete factory (ConcreteFactory3) to create the corresponding newly added concrete products. This new concrete factory class also has the same operations (createProductA and createProductB) as the other two concrete factory classes as shown in Figure 2. This evolution can be defined in terms of the primitive-level transformations as follows:

```
Add ( Class (ConcreteFactory3)) +
Add ( Class (ProductA3)) +
Add ( Class (ProductB3)) +
Add ( Generalization (ProductA3, AbstractProductA)) +
Add ( Generalization (ProductB3, AbstractProductB)) +
Add ( Generalization (ConcreteFactory3,AbstractFactory)) +
Add ( Realization (ConcreteFactory3, ProductA3)) +
Add ( Realization (ConcreteFactory3, ProductB3)) +
Add ( Operation (createProductA, ConcreteFactory3, null, public)) +
Add ( Operation (createProductB, ConcreteFactory3, null, public))
```

which indicates that three classes, ConcreteFacotory3, ProductA3, and ProductB3 are added into the pattern application. ProductA3 and ProductB3 are subclasses of AbstractProductA and AbstractProductB, respectively. ConcreteFacotory3 is a subclass of AbstractFactory. The realization relationships are also added between the

ConcreteFactory3 and ProductA3/ProductB3 classes. These relationships show that ConcreteFactory3 creates ProductA3 and ProductB3. The createProductA() and createProductB() operations are added into the ConcreteFactory3 classes.

The fifth kind of pattern-level evolution is called correlated attributes/operations change which is the addition or removal of a group of classes. This change also requires the addition or removal of some attributes or operations in the classes of the original pattern applications.

The expression of this kind of pattern-level evolution is the same as the fourth kind pattern-level evolution as follows:

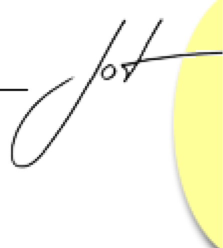
$$\sum_{i} \text{Add (Class(className}_i\text{))} +$$
$$\sum_{i,j} \text{Add (Relationship (className}_i\text{, className}_j\text{))} +$$
$$\sum_{i,j} \text{Add (Attribute(attributeName}_i\text{, className}_j\text{, type}_i\text{, accessibility}_i\text{))} +$$
$$\sum_{i,j} \text{Add (Operation(operationName}_i\text{, className}_j\text{, returnType}_i\text{, accessibility}_i\text{,...))}$$

Nevertheless, they have different semantic meaning. In the fourth kind of pattern-level evolution, the className_j in the "Add" attributes and operations transformations only includes those classes which are newly added into the pattern. In the fifth kind of pattern-level evolution, in contrast, className_j includes the newly added classes as well as the existing classes, i.e., the addition of a group of classes results in the addition of the attributes and operations of the existing classes in original pattern.

This kind of evolution can be seen in several design patterns as, for example, in the Abstract Factory and Adapter patterns. For the same example shown in Figure 1, another potential evolution can be the addition of a new kind of product (AbstractProductC with ProductC1 and ProductC2). This requires the addition of the createProductC operation in all concrete factory classes (ConcreteFactory1 and ConcreteFactory2). The corresponding generalization and realization relationships are also added as shown in Figure 3. This evolution can be defined in terms of the primitive-level evolutions as follows:

Add (Class (AbstractProductC)) +
Add (Class (ProductC1)) +
Add (Class (ProductC2)) +
Add (Generalization (ProductC1, AbstractProductC)) +
Add (Generalization (ProductC2, AbstractProductC)) +
Add (Realization (ConcreteFactory1, ProductC1)) +
Add (Realization (ConcreteFactory2, ProductC2)) +
Add (Operation (createProductC, ConcreteFactory1, null, public)) +
Add (Operation (createProductC, ConcreteFactory2, null, public))

which indicates that three classes, AbstractProductC, ProductC1, and ProductC2 are added into the pattern application. ProductC1 and ProductC2 are subclasses of AbstractProductC. The createProductC() operation is added into both ConcreteFactory1 and ConcreteFactory2 classes. The realization relationships are also



added between the ConcreteFactory1 and ProductC1 classes and between the ConcreteFactory2 and ProductC2 classes, respectively. These relationships show that ConcreteFactory1 and ConcreteFactory2 create ProductC1 and ProductC2, respectively.

All five kinds of pattern-level evolutions are summarized in Table 2.

#	Evolution Names	Description
1	Independent	Addition or removal of one independent class and the corresponding relationships between this class and the classes in the original pattern.
2	Packaged	Addition or removal of one independent class with attributes and/or operations and the corresponding relationships between this class and the classes in the original pattern.
3	Class group	Addition or removal of one attribute/operation in several different classes consistently.
4	Correlated classes	Addition or removal of a group of correlated classes.
5	Correlated attributes/operations	Addition or removal of a group of classes and addition or removal of some attributes or operations in the classes of the original pattern applications.

Table 2 Pattern Level Evolutions

Design Pattern Name	Pattern-Level Evolutions
Abstract Factory	4,5
Builder	4,5
Factory Method	4
Prototype	2
Singleton	N/A
Adapter	4,5
Bridge	2
Composite	2
Decorator	2,3
Façade	1
Flyweight	2
Proxy	4
Chain of Responsibility	2
Command	4
Interpreter	2
Iterator	4
Mediator	1
Memento	3
Observer	2,3
State	2
Strategy	2
Template Method	2,3
Visitor	2,5

Table 3 Evolutions of GoF Design Patterns

Categorization of Design Pattern Evolutions

We studied the types of pattern evolutions of the design patterns listed in [8]. The result is shown in Table 3. For each design pattern, all possible evolution types of the design pattern are listed in the "Pattern-Level Evolutions" column. Consider the

Abstract Factory pattern, for instance, one possible evolution is shown in Figure 2, which is classified as the fourth type of pattern-level evolution (correlated classes in Table 2). In this type of evolution, the addition of a new set of concrete products (ProductA3 and ProductB3) results in the addition of ConcreteFactory3 class. The other possible evolution is the fifth type (correlated attributes/operations) of pattern-level evolution as, for example, depicted in Figure 3. The addition of a new set of concrete products (ProductC1 and ProductC2) results in the addition of AbstractProductC class and the addition of operation (createProductC()) in the existing classes, ConcreteFactory1 and ConcreteFactory2. Thus, the Abstract Factory pattern may have the fourth and fifth types of possible pattern-level evolutions. Since the application of the Singleton pattern is not typically intended to evolve, it is labeled "N/A".

4 RELATED WORK

The evolution processes of design patterns have been studied in [1], where Prolog [4] is used to capture the structural evolution processes of design patterns. The structural aspect of a design pattern is described in terms of Prolog facts. Thus, the evolution and change of a design pattern application can be achieved by the addition or removal of new or old Prolog facts. The evolution processes are defined as Prolog rules. In this paper, we further characterize two-level pattern evolutions.

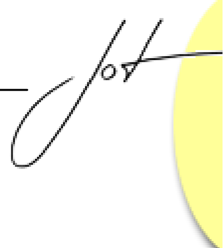
Design pattern evolutions in software development processes are also discussed in [9], where software development processes are considered as the evolutions of analysis of design patterns. The evolution rules are specified in Java-like operations to change the structure of patterns. Although some primitive-level evolution rules are introduced, there is no discussion on pattern-level evolution rules.

Noda et al. [10] consider design patterns as a concern that is separated from the application core concern. Thus, an application class may assume a role in a design pattern by weaving the design pattern concern into the application class using Hyper/J [11]. Due to the separation of concerns, an application class may assume different roles in different design patterns. The change of roles that an application class plays, i.e., the change of design patterns, becomes a relative simple task. The main goal of their evolution of design pattern is the replacement of one pattern by another. In contrast, our design pattern evolution refers to the internal changes of a design pattern application. In addition, the practical application of their approach is left as a mystery.

Improving software system quality by applying design patterns in existing systems has been discussed in [3]. When the user selects a design pattern to be applied in a chosen location of a system, automated application is supported by applying transformations corresponding to the mini-patterns. The main goal of their software evolution is to apply design patterns in existing systems, whereas our evolution goal is to change the design patterns that have already applied in a system.

5 CONCLUSIONS

Currently, the evolution information of each design pattern is generally implicit in the descriptions of the pattern. A designer has to dig into the pattern descriptions and try



to understand the particular ways of evolutions encapsulated in the design patterns. There are several problems when the evolution information is implicit: first, it is hard for the designer to take advantage of the benefits of using a design pattern when changes are needed. Second, the evolution of a design pattern generally involves several classes and relationships. Missing one part may cause inconsistencies and errors in the design which are difficult to find and correct. Third, the evolution processes are not reusable if not documented. As discussed previously, many of the evolution processes recurs in different patterns.

In this paper, we characterize two-level evolutions: the primitive level and the pattern level and explicitly describe the evolution of design patterns using these two-level evolutions. This classification not only provides explicit documentation of design pattern evolutions, but also opens the door for automation of these evolution processes. We may consider the primitive-level and pattern-level evolutions as model transformations in the Model Driven Architecture (MDA) [12]. Thus, we may provide techniques and tools for automating the pattern evolution processes as model transformation.

Design patterns are usually represented in the Unified Modeling Language (UML) [2] which is considered to be the de facto standard for object-oriented modeling. The Model Driven Architecture supports developing software systems based on models as primary artifacts. Thus, the level of abstraction of software development is raised from implementation (writing code) to model transformation. By raising the level of abstraction, the level of reuse is raised accordingly since high-level software models can be reused as well as software programs (libraries). In this way, models become assets in MDA. Consequently, technology that supports the transformation of models is considered as a key enabler of MDA. While the application of a design pattern can be represented in a design model in UML, the evolution of the design pattern may be considered as a transformation of the design model.

The XML Metadata Interchange (XMI) [13] is an interchange format for metadata in terms of the Meta Object Facility (MOF) [12]. XMI specifies how UML models are mapped into a XML file. By representing a UML models in XML format, the UML model can be manipulated since there are rich collections of XML related techniques and tools available. The extensible stylesheet language transformation (XSLT) [14] provides the transformation from XML document to other types of document (including XML). The use of XMI and XSLT helps on the automated model transformation process and enforces constraints of model implicitly.

Based on MDA, we can map our primitive-level and pattern-level evolutions into model transformations and automate these model transformations based on the XSLT techniques [6]. In this way, the users can choose a pattern-level evolution given a design pattern application and perform the evolution automatically. In addition, we have investigated the model transformation techniques based on Query, View, Transformation (QVT) that is an OMG standard allowing users to query, establish and maintain views, and transform MOF models [7]. In our investigation, we take advantage of available model transformation tools, such as Model Transformation Framework from IBM [16].

In the future, we will characterize the constraints of evolutions of each design pattern and provide techniques and tools for checking such constraints after

evolutions. Our pattern evolution techniques can be also naturally integrated with our pattern visualization techniques discussed in [5] since both are based on MDA.

REFERENCES

[1] P. Alencar, D. Cowan, J. Dong, and C. Lucena, A Pattern-Based Approach to Structural Design Composition, the Proceedings of the IEEE 23rd Annual International Computer Software & Applications Conference (COMPSAC), pp160-165, Phoenix USA, October 1999.

[2] G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language User Guide, Addison-Wesley, 1999.

[3] M. Ó Cinnéide and P. Nixon. Automated Software Evolution Towards Design Patterns, Proceedings of the International Workshop on the Principles of Software Evolution, pp162-165, Vienna, Austria, September, 2001.

[4] W. F. Clocksin and C.S. Mellish. Programming in Prolog. Berlin : Springer-Verlag, 1987.

[5] Jing Dong, Sheng Yang and Kang Zhang, Visualizing Design Patterns in Their Applications and Compositions, IEEE Transaction on Software Engineering (TSE), Volume 33, Number 7, pp. 433-453, July 2007.

[6] Jing Dong, Sheng Yang and Kang Zhang, A Model Transformation Approach for Design Pattern Evolutions, Proceedings of the Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS), pp 80-89, Germany, March 2006.

[7] Jing Dong, Sheng Yang, Yongtao Sun, and W. Eric Wong, QVT Based Model Transformation for Design Pattern Evolutions, Proceedings of the Tenth IASTED International Conference on Internet and Multimedia Systems and Applications (IMSA), pp16-22, USA, August 2006.

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

[9] T. Kobayashi and M. Saeki. Software Development Based on Software Pattern Evolution, Proceedings of the Sixth Asia-Pacific Software Engineering Conference (APSEC), pp 18-25, Takamatsu, Japan, 1999.

[10] Natsuko Noda, Tomoji Kishi. Design pattern concerns for software evolution, Proceedings of the 4th International Workshop on Principles of Software Evolution, pp 158-161, Vienna, Austria, 2001.

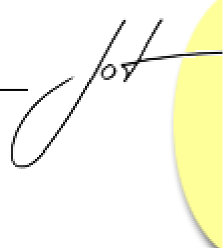
[11] Hyper/J, http://www.alphaworks.ibm.com/tech/hyperj

[12] Model Driven Architecture. http://www.omg.org/mda/

[13] W3C, Extensible Markup Language (XML), http://www.w3.org/

[14] W3C, XSL Transformations (XSLT), http://www.w3.org/

[15] Rational Rose website. http://www.rational.com/



[16] IBM, http://www-128.ibm.com/developerworks/rational/ library/05/503_sebas/

About the authors

Jing Dong is an assistant professor in the Computer Science Department at the University of Texas at Dallas. He received a Ph.D. in Computer Science from the University of Waterloo. He also holds a B.S. degree in Computer Science from Peking University. His research interests include design patterns, UML, model-driven architecture, software evolution and analysis, and formal methods. He can be reached at jdong@utdallas.edu and http://www.utdallas.edu/~jdong.

Sheng Yang received the BE degree from Tsinghua University in 1994, and the MS and PhD degrees in computer science from the University of Texas at Dallas in 2001 and 2006, respectively. His research interests include automated software engineering methods, model-driven architecture, software evolution and analysis, and software engineering tools and environment.

Yongtao Sun is a senior software architect at the Information Technology Service department of American Airlines. His research interests are design pattern, machine learning, data mining and pattern recognition.