# Design Patterns in Software Development

MU Huaxin
International School
Beijing University of Posts and Telecommunications
Beijing, PRC
E-mail: ee08b434@bupt.edu.cn

JIANG Shuai
School of Software Engineering
Beijing University of Posts and Telecommunications
Beijing, PRC
E-mail: cheetach@126.com

***Abstract*—Design pattern describes a repeatedly presenting issue during software designing, as well the solution to it. Applying design pattern enables developers to reuse it to solve a specified designing issue. Design patterns help designers communicate architectural knowledge, help people learn a new design paradigm, and help new developers avoid traps and pitfalls that have traditionally been learned only by costly experiences. In this paper, we briefly introduce the concept of software design pattern and give a research on some design patterns, including Observer Pattern, Decorator Pattern, Factory Method Pattern and Abstract Factory Pattern.**

***Keywords- Design Pattern;Object-Oriented;Software Development***

## I. INTRODUCTION

In software development, no matter where we work, what we are building, or what language we are programming in, "change" is the one true constant that will be with us always. A design pattern is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Many patterns imply object-orientation or more generally mutable state, and so may not be as applicable in functional programming languages, in which data is immutable or treated as such.

Good OO designs are reusable, extensible and maintainable. Patterns show us how to build systems with good OO design qualities. Reference [1] lists 23 design patterns. They are proven object-oriented experience. They don't give us codes, they give us general solutions to design problems. We apply them to our specific application.

## II. OBSERVER PATTERN

### A. Definition

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

### B. Description

The subject and observers define the one-to-many relationship. The observers are dependent on the subject such that when the subject's state changes, the observers get notified. Depending on the style of notification, the observer may also be updated with new values.

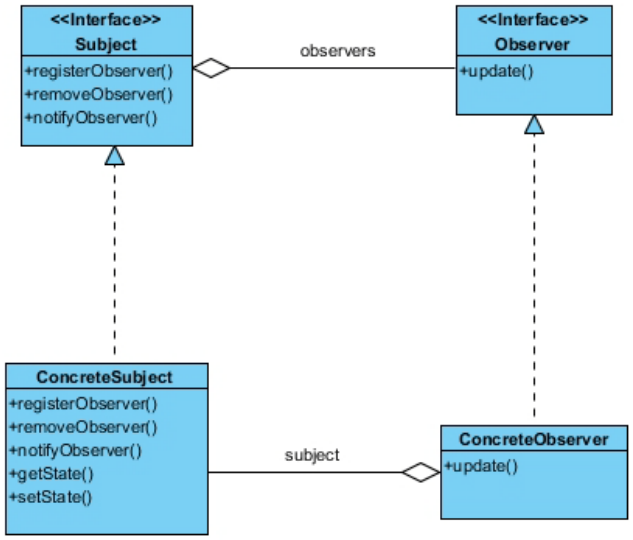


Figure 1. Observer Pattern Diagram

Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

When two objects are loosely coupled, they can interact, but have very little knowledge of each other. The Observer Pattern provides an object design where subjects and observers are loosely coupled.

The only thing the subject knows about an observer is that it implements a certain interface, the Observer interface. It doesn't need to know the concrete class of the observer, what it does, or anything else about it.

We can add new observers at any time. Because the only thing the subject depends on is a list of objects that implement the Observer interface, we can add new observers whenever we want. In fact, we can replace any

**978-1-4244-9698-3/11/$26.00 ©2011 IEEE**

observer at runtime with another observer and the subject will keep purring along. Likewise, we can remove observers at any time.

We never need to modify the subject to add new types of observers. Let's say we have a new concrete class come along that needs to be an observer. We don't need to make any changes to the subject to accommodate the new class type, all we have to do is implement the Observer interface in the new class and register as an observer. The subject doesn't care, it will deliver notifications to any object that implements the Observer interface.

We can reuse subjects or observers independently of each other. If we have another use for a subject or an observer, we can easily reuse them because the two aren't tightly coupled.

Changes to either the subject or an observer will not affect the other.

Because the two are loosely coupled, we are free to make changes to either, as long as the objects still meet their obligations to implement the subject or observer interfaces.

## III. DECORATOR PATTERN

### A. *Definition*

The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality. Decorators have the same super type as the objects they decorate.

### B. *Description*

We can use one or more decorators to wrap an object.

Given that the decorator has the same super type as the object it decorates, we can pass around a decorated object in place of the original object.

The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.

Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

Figure 2 shows the architecture of decorator pattern.

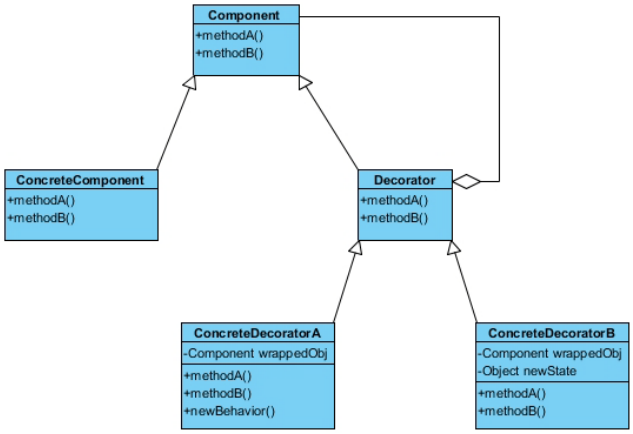


Figure 2. Decorator Pattern Class Diagram

The Decorator Pattern provides an alternative to sub-classing for extending behavior.

The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.

Decorator classes mirror the type of the components they decorate. (In fact, they are the same type as the components they decorate, either through inheritance or interface implementation.)

Decorators change the behavior of their components by adding new functionality before and/or after (or even in place of) method calls to the component.

We can wrap a component with any number of decorators.

Decorators are typically transparent to the client of the component; that is, unless the client is relying on the component's concrete type.

## IV. FACTORY METHOD PATTERN AND ABSTRACT FACTORY PATTERN

### 1. Factory Method Pattern

### A. *Definition*

The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

### B. *Decription*

As with every factory, the Factory Method Pattern gives us a way to encapsulate the instantiations of concrete types. Looking at the class diagram below, we can see that the abstract Creator gives us an interface with a method for creating objects, also known as the "factory method." Any other methods implemented in the abstract Creator are written to operate on products produced by the factory

method. Only subclasses actually implement the factory method and create products.

We often hear developers say that the Factory Method lets subclasses decide which class to instantiate. They say "decides" not because the pattern allows subclasses themselves to decide at runtime, but because the creator class is written without knowledge of the actual products that will be created, which is decided purely by the choice of the subclass that is used.

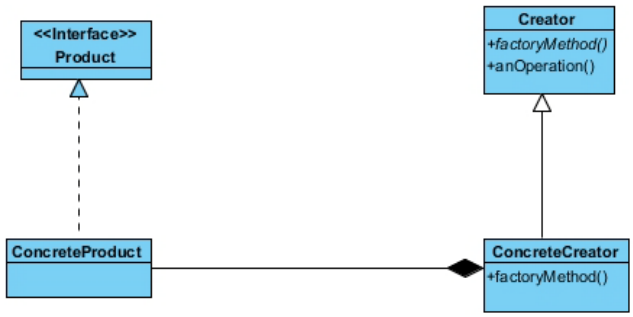


Figure 3. Factory Method Pattern Class Diagram

### C. *Explain Class Diagram*

- All concrete products must implement the same interface *Product* so that the classes which use the products can refer to the interface, not the concrete class.
- The *Creator* is an abstract class that contains the implementations for all of the methods to manipulate products, except for the factory method.
- The abstract "factoryMethod()" is what all Creator subclasses must implement.
- The ConcreteCreator implements the factoryMethod(), which is the method that actually produces products.
- The ConcreteCreator is responsible for creating one or more ConcreteProduct. It is the only class that has the knowledge of how to create these products.

## 2. Abstract Factory Pattern

### A. *Definition*

The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

### B. *Discription*

Abstract Factory allows a client to use an abstract interface to create a set of related products without knowing about the concrete products that are actually produced. In this way, the client is decoupled from any of the specifics of the concrete products. Figure 4 shows how this all holds together:

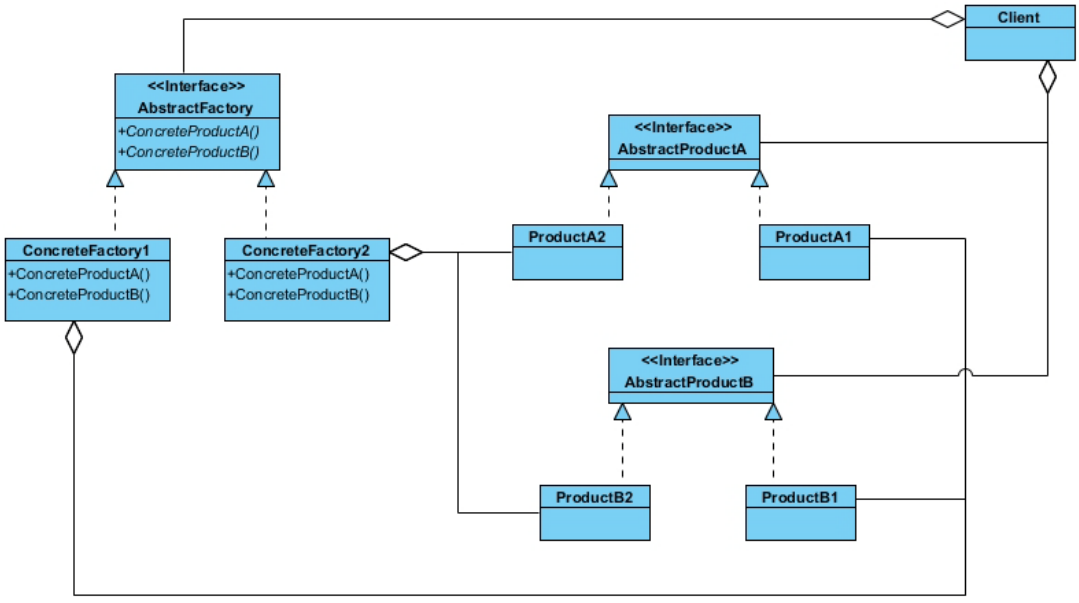


Figure 4. Abstract Factory Pattern Class Diagram

### C. *Explain Class Diagram*

- The *AbstractFactory* defines the interface that all Concrete factories must implement, which consists of a set of methods for producing products.
- The concrete factories implement the different product families. To create a product, the client uses one of these factories, so it never has to instantiate a product object.

- *AbstractProducts* are the product family. Each concrete factory can produce an entire set of products.
- The Client is written against the abstract factory and then composed at runtime with an actual factory.

3. Conclusion

Factory Method relies on inheritance: object creation is delegated to subclasses which implement the factory method to create objects. Abstract Factory relies on object composition: object creation is implemented in methods exposed in the factory interface. All factory patterns promote loose coupling by reducing the dependency of your application on concrete classes. The intent of Factory Method is to allow a class to defer instantiation to its subclasses. The intent of Abstract Factory is to create families of related objects without having to depend on their concrete classes.

## V. CONCLUSION

After learning the design patterns above, we get some design principles:

- Identify the aspects of your application that vary and separate them from what stays the same.

  Take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don't.

  It forms the basis for almost every design pattern.

- Program to an interface, not an implementation.

  "Program to an interface" really means "Program to a super type."

  The point is to exploit polymorphism by programming to a super type so that the actual runtime object isn't locked into the code. And we could rephrase "program to a super type" as "the declared type of the variables should be a super type, usually an abstract class or interface, so that the objects assigned to those variables can be of any concrete implementation of the super type, which means the class declaring them doesn't have to know about the actual object types!"

- Classes should be open for extension, but closed for modification, which is the famous Open-Close-Principle.

- Creating systems using composition gives you a lot more flexibility.

  Not only does it let you encapsulate a family of algorithms into their own set of classes, but it also lets you change behavior at runtime as long as the object you're composing with implements the correct behavior interface.

## REFERENCES


[1] Gamma, E., Richard Helm, Ralph Johnson and John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented software," Addison-Wesley, 1995

[2] Eric Freeman, Elisabeth Freeman, Kathy Sierra and Bert Bates, "Head First Design Patterns," O'Reilly, 2004

[3] Jing Dong, Sheng Yang and Kang Zhang. "A Model Transformation Approach for Design Pattern Evolutions," in IEEE International Conference and Workshop on the Engineering of Computer Based System, 2006, Germany, pp. 10

[4] Wang Xuebin, Wu Quanyuan, Wang Huaimin and Shi Dianxi, "Research and Implementation of Design Pattern-Oriented Model Transformation," Computing in the Global Information Technology, IEEE Press, Mar. 2007, pp. 24