Design Patterns in Object Oriented Analysis and Design

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

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

Abstract—**Design pattern is a general reusable solution to a commonly occurring problem in software development. Good OO designs are reusable, extensible and maintainable. Patterns only give you a general rule not code. Patterns show you how to build systems with good OO design qualities. Most of them address issues of change in software and allow some part of a system to vary independently of all other parts by trying to take what varies in a system and encapsulate it. Patterns also provide a shared language that can maximize the value of your communication with other developers. In this paper, we briefly introduce the concept of software design pattern and give a research on some design patterns including Strategy Pattern, Iterator Pattern, Adapter Pattern and Façade Pattern.**

Keywords- Design Pattern;OOAD;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.

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.

II. STRATEGY PATTERN

A. Definition

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

B. Description

Strategy Pattern is an encapsulation of each of a family of algorithms. It separates the use and the algorithm itself and assigns them to different objects. Generally Strategy Pattern encapsulates a family of algorithms into a family of Strategy classes as the subclass of an abstract Strategy super class.

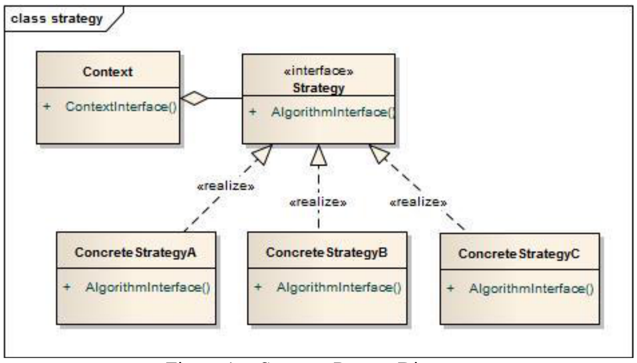

Figure 1. Strategy Pattern Diagram

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 Strategy Pattern provides an object design where context and strategy are loosely coupled

In some situations, behaviors of methods change frequently between models. A common approach is to implement these behaviors in subclasses. This approach has significant drawbacks: behaviors must be declared in each new subclass. The work of managing these behaviors increases greatly as the number of models increases, and requires code to be duplicated across models. Additionally, it is not easy to determine the exact nature of the behavior for each model without investigating the code in each.

The strategy pattern uses composition instead of inheritance. In the strategy pattern behaviors are defined as separate interfaces and specific classes that implement these interfaces. Specific classes encapsulate these interfaces. This allows better decoupling between the behavior and the class that uses the behavior. The behavior can be changed without breaking the classes that use it, and the classes can switch between behaviors by changing the specific implementation used without requiring any significant code changes. Behaviors can also be changed at run-time as well as at design-time.

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 strategy interfaces.

III. ITERATOR PATTERN

A. Definition

The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation

B. Description

The pattern gives you a way to step through the elements of an aggregate without having to know how things are represented under the covers. The effect of using iterators in your design is very important: once you have a uniform way of accessing the elements of all your aggregate objects, you can write polymorphic code that works with any of these aggregates, without caring if the elements are held in an Array or ArrayList (or anything else that can create an Iterator), as long as it can get hold of an Iterator.

The other important impact on your design is that the Iterator Pattern takes the responsibility of traversing elements and gives that responsibility to the iterator object, not the aggregate object. This not only keeps the aggregate interface and implementation simpler, it removes the responsibility for iteration from the aggregate and keeps the aggregate focused on the things it should be focused on (managing a collection of objects), not on iteration. Figure 2 shows the architecture of iterator pattern.

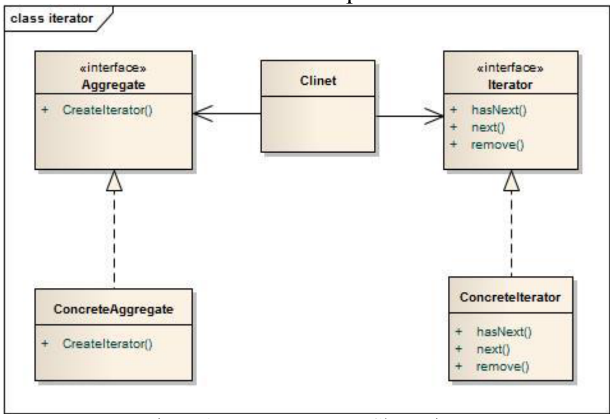

Figure 2. Iterator Pattern Class Diagram

The remove() method is considered optional. You don't have to provide remove functionality. But, obviously you do need to provide the method because it's part of the Iterator interface. If you're not going to allow remove() in your iterator you'll want to throw the runtime exception java.lang.UnsupportedOperationException. The Iterator API documentation specifies that this exception may be thrown from remove() and any client that is a good citizen will check for this exception when calling the remove() method.

Iterators imply no ordering. The underlying collections may be unordered as in a hash table or in a bag; they may even contain duplicates. So ordering is related to both the properties of the underlying collection and to the implementation. In general, you should make no assumptions about ordering unless the Collection documentation indicates otherwise.

In general the primary purpose of an iterator is to allow a user to process every element of a container while isolating the user from the internal structure of the container. This allows the container to store elements in any manner it wishes while allowing the user to treat it as if it were a simple sequence or list.

IV. ADAPTER PATTERN AND FACADE PATTERN

i. Adapter Pattern

A. Definition

The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

B. Decription

Adapter pattern allows us to use a client with an incompatible interface by creating an Adapter that does the conversion. This acts to decouple the client from the implemented interface, and if we expect the interface to change over time, the adapter encapsulates that change so that the client doesn't have to be modified each time it needs to operate against a different interface

The Adapter Pattern is full of good OO design principles: check out the use of object composition to wrap the adaptee with an altered interface. This approach has the added advantage that we can use an adapter with any subclass of the adaptee.

Also check out how the pattern binds the client to an interface, not an implementation; we could use several adapters, each converting a different backend set of classes. Or, we could add new implementations after the fact, as long as they adhere to the Target interface.

This is employing the principle of Loosely Coupled. where two objects can interact, but have very little knowledge of each 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 agreed on interfaces.

The job of implementing an adapter is really proportional to the size of the interface you need to support as your target interface. Think about your options, however. You could rework all your client-side calls to the interface, which would result in a lot of investigative work and code changes. Or, you can cleanly provide one class that encapsulates all the changes in one class.

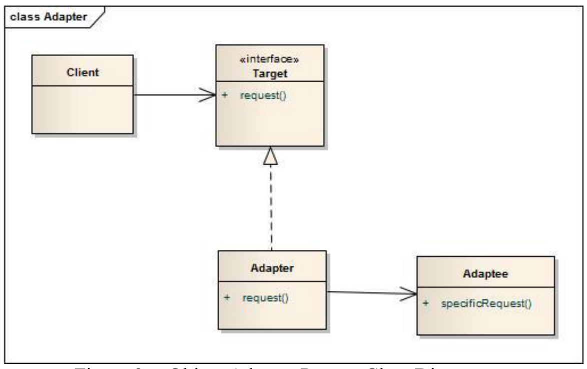

Figure 3. Object Adapter Pattern Class Diagram

There are two kinds of adapters: object adapters and class adapters: object adapters and class adapters. You need multiple-inheritance to implement it, which isn't possible in Java. But, that doesn't mean you might not encounter a need for class adapters down the road when using your favorite multiple inheritance language! Let's look at the class diagram for multiple-inheritance.

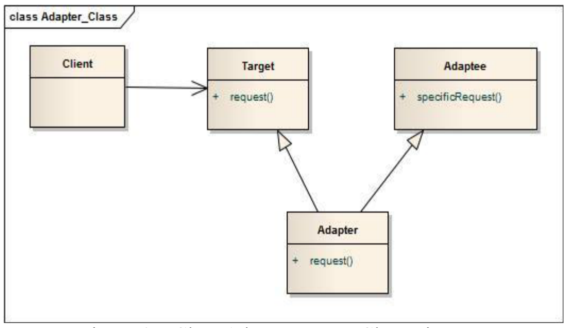

Figure 4. Class Adapter Pattern Class Diagram

The two diagrams look similar. The only difference is that with class adapter we subclass the Target and the Adaptee, while with object adapter we use composition to pass requests to an Adaptee.

There are some issues to be mentioned while considering object adapter and class adapter pattern.

Object adapter pattern use composition, it can not only adapt an adaptee class, but any of its subclasses. Class adapter patter do have trouble with that because It's committed to one specific adaptee class, but it have a huge advantage because it doesn't have to reimplement its entire adaptee. Class adapter pattern can also override the behavior of my adaptee if it need to because it's just subclassing. In object adapter pattern, we like to use composition over inheritance; class adapter may be saving a few lines of code, but all class adapter is doing is writing a little code to delegate to the adaptee. Object adapater can keep things flexible. Using a class adapter there is just one of that, not an adapter and an adaptee, which means efficiency. But one little object can't have much impact on efficient of the system. Class adapter might be able to quickly override a method, but any behavior object adapter add to its adapter code works with it's adaptee class and all its subclasses.

In general the adapter acts as the middleman by receiving requests from the client and converting them into requests that make sense on the vendor classes, without changing your existing code (and you can't change the vendor's code)

The Adapter Pattern's role is to convert one interface into another. While most examples of the adapter pattern show an adapter wrapping one adaptee, we both know the world is often a bit more messy. So, you may well have situations where an adapter holds two or more adaptees that are needed to implement the target interface. This relates to another pattern called the Facade Pattern; people often confuse the two. Next this paper will give a brief introduction to Façade Pattern

ii. Façade Pattern

A. Definition

The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher level interface that makes the subsystem easier to use.

B. Discription

Unlike a lot of patterns, Facade is fairly straightforward; there are no mind bending abstractions to get your head around. But that doesn't make it any less powerful: the Facade Pattern allows us to avoid tight coupling between clients and subsystems, and also helps us adhere to a new object oriented principle: the principle of least knowledge.

Facades don't "encapsulate" the subsystem classes; they merely provide a simplified interface to their functionality. The subsystem classes still remain available for direct use by clients that need to use more specific interfaces. This is a nice property of the Facade Pattern: it provides a simplified interface while still exposing the full functionality of the system to those who may need it.

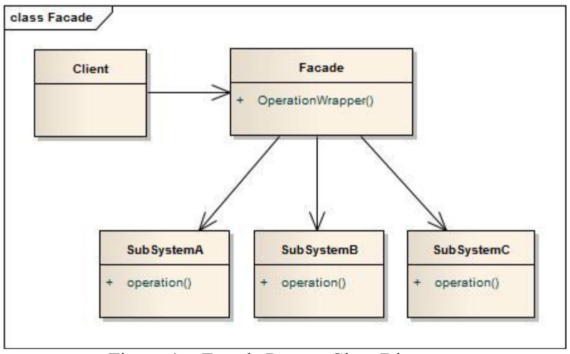

Figure 4. Facade Pattern Class Diagram

A facade is free to add its own "smarts" in addition to making use of the subsystem. For instance, while our home theater facade doesn't implement any new behavior, it is smart enough to know that the popcorn popper has to be turned on before it can pop (as well as the details of how to turn on and stage a movie showing).

The Facade Pattern also allows you to decouple your client implementation from any one subsystem. Let's say for instance that you get a change in your business and decide to upgrade your subsystem to all new components that have different interfaces. Well, if you coded your client to the façade rather than the subsystem, your client code doesn't need to change, just the façade (and hopefully the vender is supplying that!).

iii. Conclusion

When you need to use an existing class and its interface is not the one you need, use an adapter. When you need to simplify and unify a large interface or complex set of interfaces, use a facade.

An adapter changes an interface into one a client expects. A facade decouples a client from a complex subsystem. Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.

Implementing a facade requires that we compose the facade with its subsystem and use delegation to perform the work of the façade. There are two forms of the Adapter Pattern: object and class adapters. Class adapters require multiple-inheritance. You can implement more than one facade for a subsystem.

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, John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented software" 1995: Addison-Wesley

[2] Bieman, J.M., Jain, D., Yang, H.J., "OO design patterns, design structure, and program changes: an industrial case study" Software Maintenance, Proceedings. IEEE International Conference, Nov. 2001, pp:580, doi: 10.1109/ICSM.2001.972775

[3] McNatt, W.B., Bieman, J.M., "Coupling of design patterns: common practices and their benefits" Computer Softwasre and Applications Conference, Oct. 2001. COMPSAC 2001. 25th Annual International, pp:574, doi: 10.1109/CMPSAC.2001.960670