Design Pattern Course

Code: JAV-403
Length: 4 days
URL: View Online

The Java Design Patterns Course is authored by Heinz Kabutz, the publisher of The Java Specialists' Newsletter, with contributions from one of the authors of the original "Gang-of-Four" Design Patterns book. Design Patterns have become the standard way in which we think about object oriented software development. During this 3-day course, we study the most useful Gang-of-Four design patterns: Singleton, Factory Method, Abstract Factory, Template Method, Strategy, Iterator, Observer, Adapter, Decorator, Composite, Visitor, Command, Memento, Chain of Responsibility, State, Facade, Flyweight, Bridge and Proxy. Each design pattern is followed by practical hands-on exercises to apply what you have learned, using both UML and Java code.

Skills Gained

Please refer to Design Pattern Course Content Details

Prerequisites

Prior Knowledge: Participants should have a good understanding of object orientation concepts such as inheritance, encapsulation and polymorphism. Knowledge of Java is an advantage. Preparation: Read the book Head First Design Patterns. Equipment: Computers with the latest version of Sun JDK preinstalled, together with either Eclipse or IntelliJ IDEA.

Course Details

Schedule

Our Design Patterns Course takes a total of 21 lectures. Each pattern is followed by a short group discussion and then some hands-on exercises to apply what we have learned. The course can be delivered either as classroom or as live virtual class. The classroom training would usually be done in a three day block. Here is the typical schedule:

- Day 1: Introduction, Proxy, Adapter, Facade, Composite, Template Method, Strategy
- Day 2: Iterator, Observer, Singleton, Factory Method, Abstract Factory, Visitor, Command
- Day 3: Memento, Chain of Responsibility, State, Flyweight, Bridge, Decorator, Conclusion
- The live virtual classes are presented via webinar and can be done over a longer period. For example, three lectures a week over a 7 week period. This gives students a longer time to assimilate the new information and to complete all of the exercises.
- Exercises:
- It is very important that students try to apply the pattern in code. We spend approximately 46% of our time doing active hands-on labs.
- During classroom courses, we check that everyone is indeed completing all the required tasks.
- With live virtual classes, we need to use another approach. Some of our clients have policies that their staff may not

send out source code. This makes it hard for them to submit their solutions.

- Instead, at the start of each session, students have to confirm whether they have completed the previous exercise. Late arrival or abstentions are counted as a "no". We then spend ten to fifteen minutes going over our model solution to show what we expected from them.
- Timing of classroom course:
- Each days training starts at 9:00am and ends at 5:00pm, with one hour of lunch and 30 minutes of tea / coffee / nicotene breaks.
- Timing of live virtual sessions:
- Each session takes approximately an hour, including the group discussion and our look at the previous exercise solutions. Exercises for each pattern will take anything from fifteen minutes to an hour, depending on the skill and coding experience of the student.

1. Introduction to Patterns

In the first section, we lay the foundation for the rest of the course. We talk about why patterns are important, where they come from, their general structure and give a UML refresher.

- Importance of patterns: Experienced object oriented programmers use patterns to build their software. This helps to make their components reusable. The reduction in copy and pasted code makes maintenance easier. In this section we look at why patterns are important to help your team speak the same language.
- Origin: We look at some great resources that explain where object oriented design patterns come from. We look at some of the great books on patterns in which you can find more detailed information. Examples are: Head First Design Patterns, Pattern Hatching and of course the famous Gang-of-Four book.
- Names: The name is useful to document which pattern we are referring to. We talk about the dangers of having a weak name and a strong pattern, as well as having a strong name and a weak pattern. Or just having the wrong name to start with.
- Diagrams: Throughout our course we use UML to describe the patterns. We explain why being able to draw a class diagram of the pattern is important. A pattern has three main elements that identify it: its name, its structure and its intent. We look at what the structure typically looks like to help you recognise typical patterns. However, the structure cannot give you a conclusive answer as to what the pattern is. This can only be discovered once you know what the intent of the author was.
- UML refresher: We use UML class diagrams to describe the designs. In this section we review the basic UML components used to show our patterns. This helps us to understand the subtleties of association, aggregation and composition. We also show how to draw the UML components. We show why reverse engineering UML from code is of little use.
- jpatterns.org annotations: We give a short tutorial on how to use the jpatterns.org annotations to describe the design patterns in our code. These will be used extensively in our coding to show what patterns we were thinking of using.

2. Structural Patterns (I)

Our first stop is structural patterns, since most of us can relate to these easily.

Proxy

The Proxy pattern is used in many systems to provide placeholder objects. Most programmers have seen or used this

pattern. This makes it a good first pattern to study as we quickly see the benefit of using design patterns. We study various ways of creating proxies. We show how to write them by hand, how to generate them using the new Java 6 built-in compiler and how to create them as dynamic proxies. We also study different uses of proxy types. Examples are virtual, remote and protection proxies. We show how these can be created with dynamic proxies or automatically generated code.

- Virtual Proxy: The virtual proxy creates objects on demand. It helps us simplify our clients when we want to lazily create objects. It does this by moving the lazy creation functionality into the virtual proxy class. Students get to build their own virtual proxy using dynamic proxies.
- Remote Proxy: Remote proxies are used a lot in Java to communicate between different machines. We look at the challenges with using this type of proxy. Examples are the checked exceptions and the much slower remote methoc calls.
- Protection Proxy: Our last variation is the protection proxy, where we guard the real object. As part of the exercises, students have to build a protection proxy that only allows us to view salary details if we have the right permissions.

Adapter

The adapter is next. In a perfect world, this pattern would not exist. However, often interfaces do not match correctly between different parts of our system and this pattern is the glue that connects these mismatched parts. We look at the naming of this pattern, which has caused confusion in the past. Lastly we compare the structure of this pattern against the proxy.

- Object Adapter: The most common form of this pattern is the object adapter. We explain why this is more flexible and typically the better choice, even though it is more coding. We also explain how we can create dynamic object adapters using the dynamic proxy mechanism.
- Class Adapter: Class adapter is possible in Java, when the subject is an interface, rather than a class. We talk about cases where the class adapter is not possible to use. It is less code, so it is a good first step, but if a class hierarchy evolves with the adapted object, then it is better to use the object adapter.

Facade

The Facade pattern is often confused with the session facade. In fact, it is not really a design pattern. Rather, it is a necessary part of any complex design thanks to the added flexibility that patterns bring.

- Facade vs Session Facade: The Session Facade usually prevents direct access to the single pieces of the subsystem. The GoF Facade allows direct access to the subsystem. We also look at the motivation of using the Session Facade.
- Is it a Design Pattern?: We look at the question: Is the Facade really a design pattern?

Composite

When you have objects arranged in a tree structure, this pattern will help you make clients simple. Clients do not need to worry whether they are speaking to a single object or a large group of objects. We explain why the interface at the top of the hierarchy seems too general, but is in fact exactly the way that it should be.

- Recursively visiting: One of the posers of this pattern is how to find all leaves of the tree without building up a temporary list. In our exercise we discover how to do this in code.

3. Behavioral Patterns (I)

Our next set of patterns are used for algorithms and behavior. They help to reduce duplicate code and get rid of if-else and switch statements. Code becomes less complex with less duplication, leading to lower costs.

Template Method

The template method is used when we have some classes which do the same thing, but a bit differently. We can replace the duplicate code with a single template method and can push the differences into primitive methods, which are contained in subclasses. We compare the template method to the strategy and show how they might be combined to produce a truly flexible design.

Strategy

We use this pattern when we need to have several ways of doing the same thing. For example, we might want to sort a list using as little temporary memory as possible. Or our constraint might be CPU, in which case we would want to replace our sorting algorithm with one that performs better. We also see some of the drawbacks of using the strategy pattern by studying the AWT layout managers.

- Converting switch statements: When writing code, we should avoid using if-else or switch. Every time we have a condition, we double the state space. This makes it hard to test our code. Much better is to encapsulate the functionality into strategy objects. In this part we learn how to convert a switch statement into the strategy pattern thus reducing cyclomatic complexity and making it easier to test our code.
- Intrinsic vs Extrinsic State: We look at ways to reduce the intrinsic state of strategy objects, so that we can share them between different contexts. We do this by converting extrinsic state to intrinsic.

Iterator

Most Java people think that they know the iterator, after all, that is how we walk through groups of objects. However, the GoF pattern has some interesting variations and options that are not available in the Java version. For example, why does it have to be the client that controls the iteration? How could we write a robust iterator in Java? This pattern has a lot more facets than most people realize.

- Robustness: A robust iterator will preserve the order even if the objects are changed whilst we are walking through. This should be achieved without copying the collection. We look at various approaches to building this.
- Iterating Collections: One of the subtleties of this pattern is the question: who controls the iteration? In this pattern, we show how we can do this from within the collection, thus taking care of synchronization in a much better way than if we lock from outside. The WalkingCollection is an example of how this can be done.

Observer

The observer pattern is one of the most prolific ideas in object oriented designs. It helps us to decouple the components. We show how the Java observer was implemented and use this to build a simple stock price viewer, where observers can get notified when a share changes.

4. Creational Patterns

Creational patterns are the most tricky to get right. They are frequently abused for building systems that are non object oriented. In this section we look at the three most common creational patterns and show how they should be used.

Singleton

Ask a group of novice programmers if they know any patterns and the "singleton" is the most common answer you will get, next to "factory" and "MVC". In this section we discover the real purpose of this pattern. The purpose of this pattern is to be able to substitute one singleton object for another without the clients having to know. Sadly most of the time it is used to put global variables into systems. We learn how to correctly use this pattern.

- Polymorphic: One of the characteristics of the singleton is that the constructor is private. This makes it impossible to create several instances. We have a look at how to allow for polymorphism.
- Managing synchronization: A lot has been written about how to actually initialize the instance. The most tricky part

is getting synchronization correct. We look at solutions and reasons why the most simple option is usually the best.

- When to use & avoid: There are several reasons why we should avoid using the singleton. It makes our system very static. It is difficult to reset instances. We cause global access to shared values. In this section we look at the reasons why singletons cause so many problems in designs.

Factory Method

The GoF factory method pattern is used to allow subclasses to create their own instances of their own specific types. Ar example would be the iterator() method of the Collection interface. We give examples of how and when this should be used.

- Simple factory: Martin Fowler uses the name "factory method" to describe something quite different to the GoF pattern. His is a static method that creates objects. This is a useful pattern in its own right, but is not the GoF pattern. In Java we can often use his pattern with the reflection API.
- Tough questions:This pattern is used to produce decoupled systems. We combine this with strategy and facade to produce a flexible design. We also ask some hard questions to make sure that we all understand the differences between the interpretations of this pattern.

Abstract Factory

We use this pattern to manage families of objects that need to be created. An example is when we need to support GUI components for several operating systems. We show how we can use this pattern to create software that can easily be migrated to new hardware.

5. Behavioral Patterns (II)

In the second set of behavior patterns, we learn how to decouple our designs even further with the Visitor and Command. We implement a multi-level undo and redo with the memento. We show how to build chains of implementers with the Chain of Responsibility. We end with the State Pattern that should be used whenever we have more than a couple of states in our classes.

Visitor

The Visitor integrates well with the composite patterns. It provides a way to modify the methods invoked on all of the objects in the structure. Instead of hard-coding the composite method, we provide a mechanism to "visit" each class and apply class specific methods. It is a great pattern for managing collections of disparate classes.

Command

The command pattern helps to clearly assign responsibilities, which leads to a decoupled design. We show why the typical AWT action listener is not the best way of using the pattern. We then demonstrate how we get a cleaner design when we follow the pattern structure more closely. The exercise is challenging as the first instinct is to use an object adapter, but that would lead to an inflexible design.

- Swing: Swing and AWT use the command pattern to bind actions to receivers. We learn the advantages of an event based, decoupled design.
- Thread Pools: In Java 5, thread pools were introduced into the standard JDK. These solve the question of how to deal with results that you might want to return from commands. We look at how Runnable and Callable is used within the ExecutorService.

Memento

We learn how to use the memento pattern to store state. This allows us to do multi-level undo and redo, combined with the Command pattern. The memento is also compared to the serialization mechanism in Java.

Chain of Responsibility

This pattern allows us to define flexible systems where we do not need to specify at coding time which object will process a message. This is set up by configuring the chain. We look at some of the issues that happen when a message drops off the end of the chain.

State

The last behavioral pattern is the State pattern. Use this whenever you have more than one conditional statement in your class that changes the branch depending on some internal field. It is one of the most useful patterns to produce reliable software.

- Implementation Choices: The state pattern can be implemented in more than five different ways. We look at the advantage and disadvantage of each one.

6. Structural Patterns (II)

We end our course with three structural patterns that are again fairly easy to grasp. Flyweight, similarly to the Facade, is mainly used to deal with the effects of patterns - lots of little objects. We see how Bridge is used to decouple the abstraction from the implementation. Lastly we show how we can change the interface of an object dynamically with the decorator.

Flyweight

The flyweight is used to reduce the memory footprint of objects. It does this by changing intrinsic state to extrinsic and then sharing instances. Well factored code often causes lots of objects to be constructed. The flyweight is one of the solutions to manage this problem.

- Performance: We look at the impact on performance of using this pattern. Even though there will be less object instances, the flyweight mechanism can add contention to the system. This would cause performance drag under heavily threaded systems.
- Measuring memory: To know the impact of large numbers of small objects, we look at the memory structure of objects in Java. This will give us a metric to calculate the potential savings of using the flyweight pattern.

Bridge

The bridge pattern helps us to define separate hierarchies for the abstractions and implementations. We learn how this relates to strategy and adapter. Bridge helps us design systems that can work with different external components, without changing the main abstraction.

Decorator

The decorator is used for two main purposes. Firstly we use it to add functionality to existing objects. We call this decorating. Secondly we can take it away from objects. We call this filtering.

- Collections: We learn why the special collections (unmodifiable, synchronized, checked, etc.) are in fact not decorators, but rather another pattern. We also demonstrate how we would design the collection classes using the decorator or filter pattern.

7. Conclusion

The course ends with a conclusion about what we have learned and where to study them further. We also look at how the patterns are all connected.

References

We give a number of great references where you can learn new patterns and study the subtleties of patterns further.

Discovering Patterns

Lastly, we give some pointers on how to write your own patterns. This process is called discovering or inventing a pattern.

Schedule (as of 3)

Date	Location