Java Design Patterns – Example Tutorial

Design Patterns are very popular among software developers. A design pattern is a well described solution to a common software problem.

Some of the benefits of using design patterns are:

1. Design Patterns are already defined and provides **industry standard approach** to solve a recurring problem, so it saves time if we sensibly use the design pattern.
2. Using design patterns promotes **reusability** that leads to more **robust** and highly maintainable code. It helps in reducing total cost of ownership (TCO) of the software product.
3. Since design patterns are already defined, it makes our code easy to understand and debug. It leads to faster development and new members of team understand it easily.

Java Design Patterns are divided into three categories – **creational**, **structural**, and **behavioral** design patterns. This post serves as an index for all the java design patterns articles I have written so far.

- Creational Design Patterns
    1. Singleton Pattern
    2. Factory Pattern
    3. Abstract Factory Pattern
    4. Builder Pattern
    5. Prototype Pattern

- Structural Design Patterns
    1. Adapter Pattern
    2. Composite Pattern

Creational Design Patterns

Creational design patterns provide solution to instantiate a object in the best possible way for specific situations.

1. Singleton Pattern

Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine. It seems to be a very simple design pattern but when it comes to implementation, it comes with a lot of implementation concerns. The implementation of Singleton pattern has always been a controversial topic among developers. Check out **Singleton Design Pattern** to learn about different ways to implement Singleton pattern and pros and cons of each of the method.

2. Factory Pattern

Factory design pattern is used when we have a super class with multiple sub-classes and based on input, we need to return one of the sub-class. This pattern take out the responsibility of instantiation of a class from client program to the factory class. We can apply Singleton pattern on Factory class or make the factory method static. Check out **Factory Design Pattern** for example program and factory pattern benefits.

3. ### Abstract Factory Pattern

   Abstract Factory pattern is similar to Factory pattern and it's factory of factories. If you are familiar with factory design pattern in java, you will notice that we have a single Factory class that returns the different sub-classes based on the input provided and factory class uses if-else or switch statement to achieve this.

   In Abstract Factory pattern, we get rid of if-else block and have a factory class for each sub-class and then an Abstract Factory class that will return the sub-class based on the input factory class. Check out **Abstract Factory Pattern** to know how to implement this pattern with example program.

4. ### Builder Pattern

   This pattern was introduced to solve some of the problems with Factory and Abstract Factory design patterns when the Object contains a lot of attributes. Builder pattern solves the issue with large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object. Check out **Builder Pattern** for example program and classes used in JDK.

5. ### Prototype Pattern

   Prototype pattern is used when the Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing. So this pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. This pattern uses java cloning to copy the object.

   Prototype design pattern mandates that the Object which you are copying should provide the copying feature. It should not be done by any other class. However whether to use shallow or deep copy of the Object properties depends on the requirements and its a design decision. Check out **Prototype Pattern** for sample program.

Structural Design Patterns

Structural patterns provide different ways to create a class structure, for example using inheritance and composition to create a large object from small objects.

1. ### Adapter Pattern

Adapter design pattern is one of the structural design pattern and its used so that two unrelated interfaces can work together. The object that joins these unrelated interface is called an Adapter. As a real life example, we can think of a mobile charger as an adapter because mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India). So the mobile charger works as an adapter between mobile charging socket and the wall socket. Check out **Adapter Pattern** for example program and it's usage in Java.

2. Composite Pattern

Composite pattern is one of the Structural design pattern and is used when we have to represent a part-whole hierarchy. When we need to create a structure in a way that the objects in the structure has to be treated the same way, we can apply composite design pattern.

Lets understand it with a real life example – A diagram is a structure that consists of Objects such as Circle, Lines, Triangle etc and when we fill the drawing with color (say Red), the same color also gets applied to the Objects in the drawing. Here drawing is made up of different parts and they all have same operations. Check out **Composite Pattern** article for different component of composite pattern and example program.

3. Proxy Pattern

Proxy pattern intent is to "Provide a surrogate or placeholder for another object to control access to it". The definition itself is very clear and proxy pattern is used when we want to provide controlled access of a functionality.

Let's say we have a class that can run some command on the system. Now if we are using it, its fine but if we want to give this program to a client application, it can have severe issues because client program can issue command to delete some system files or change some settings that you don't want. Check out **Proxy Pattern** post for the example program with implementation details.

4. Flyweight Pattern

Flyweight design pattern is used when we need to create a lot of Objects of a class. Since every object consumes memory space that can be crucial for low memory devices, such as mobile devices or embedded systems, flyweight design pattern can be applied to reduce the load on memory by sharing objects. String Pool implementation in java is one of the best example of Flyweight pattern implementation. Check out **Flyweight Pattern** article for sample program and implementation process.

5. Facade Pattern

Facade Pattern is used to help client applications to easily interact with the system. Suppose we have an application with set of interfaces to use MySql/Oracle database and to generate different types of reports, such as HTML report, PDF report etc. So we will have different set of interfaces to work with different types of database. Now a client application can use these interfaces to get the required database connection and generate reports. But when the complexity increases or the interface behavior names are confusing, client application will find it difficult to manage it. So we can apply Facade pattern here and provide a wrapper interface on top of the existing interface to help client application. Check out **Facade Pattern** post for implementation details and sample program.

6. Bridge Pattern

When we have interface hierarchies in both interfaces as well as implementations, then builder design pattern is used to decouple the interfaces from implementation and hiding the implementation details from the client programs. Like Adapter pattern, its one of the Structural design pattern.

The implementation of bridge design pattern follows the notion to prefer Composition over inheritance. Check out Bridge Pattern post for implementation details and sample program.

7. Decorator Pattern

Decorator design pattern is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behavior. Decorator design pattern is one of the structural design pattern (such as Adapter Pattern, Bridge Pattern, Composite Pattern) and uses abstract classes or interface with composition to implement.

We use inheritance or composition to extend the behavior of an object but this is done at compile time and its applicable to all the instances of the class. We can't add any new functionality of remove any existing behavior at runtime – this is when Decorator pattern comes into picture. Check out **Decorator Pattern** post for sample program and implementation details.

Behavioral Design Patterns

Behavioral patterns provide solution for the better interaction between objects and how to provide lose coupling and flexibility to extend easily.

1. Template Method Pattern

Template Method is a behavioral design pattern and it's used to create a method stub and deferring some of the steps of implementation to the subclasses. Template method defines the steps to execute an algorithm and it can provide default implementation that might be common for all or some of the subclasses.

Suppose we want to provide an algorithm to build a house. The steps need to be performed to build a house are – building foundation, building pillars, building walls and windows. The important point is that the we can't change the order of execution because we can't build windows before building the foundation. So in this case we can create a template method that will use different methods to build the house. Check out **Template Method Pattern** post for implementation details with example program.

2. Mediator Pattern

Mediator design pattern is used to provide a centralized communication medium between different objects in a system. Mediator design pattern is very helpful in an enterprise application where multiple objects are interacting with each other. If the objects interact with each other directly, the system components are tightly-coupled with each other that makes maintainability cost higher and not flexible to extend easily. Mediator pattern focuses on provide a mediator between objects for communication and help in implementing lose-coupling between objects.

Air traffic controller is a great example of mediator pattern where the airport control room works as a mediator for communication between different flights. Mediator works as a router between objects and it can have it's own logic to provide way of communication. Check out **Mediator Pattern** post for implementation details with example program.

3. Chain of Responsibility Pattern

Chain of responsibility pattern is used to achieve lose coupling in software design where a request from client is passed to a chain of objects to process them. Then the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

We know that we can have multiple catch blocks in a try-catch block code. Here every catch block is kind of a processor to process that particular exception. So when any exception occurs in the try block, its send to the first catch block to process. If the catch block is not able to process it, it forwards the request to next object in chain i.e next catch block. If even the last catch block is not able to process it, the exception is thrown outside of the chain to the calling program.

ATM dispense machine logic can be implemented using **Chain of Responsibility Pattern**, check out the linked post.

4. ## Observer Pattern

Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change. In observer pattern, the object that watch on the state of another object are called **Observer** and the object that is being watched is called **Subject**.

Java provides inbuilt platform for implementing Observer pattern through java.util.Observable class and java.util.Observer interface. However it's not widely used because the implementation is really simple and most of the times we don't want to end up extending a class just for implementing Observer pattern as java doesn't provide multiple inheritance in classes.

Java Message Service (JMS) uses Observer pattern along with Mediator pattern to allow applications to subscribe and publish data to other applications. Check out **Observer Pattern** post for implementation details and example program.

5. ## Strategy Pattern

Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime.

Strategy pattern is also known as Policy Pattern. We defines multiple algorithms and let client application pass the algorithm to be used as a parameter. One of the best example of this pattern is Collections.sort() method that takes Comparator parameter. Based on the different implementations of Comparator interfaces, the Objects are getting sorted in different ways.

Check out **Strategy Pattern** post for implementation details and example program.

6. ## Command Pattern

Command Pattern is used to implement lose coupling in a request-response model. In command pattern, the request is send to the invoker and *invoker* pass it to the encapsulated *command* object. Command object passes the request to the appropriate method of *Receiver* to perform the specific action.

Let's say we want to provide a File System utility with methods to open, write and close file and it should support multiple operating systems such as Windows and Unix.

To implement our File System utility, first of all we need to create the receiver classes that will actually do all the work. Since we code in terms of java interfaces, we can have FileSystemReceiver interface and it's implementation classes for different operating system flavors such as Windows, Unix, Solaris etc. Check out **Command Pattern** post for the implementation details with example program.

7. State Pattern

State design pattern is used when an Object change it's behavior based on it's internal state.

If we have to change the behavior of an object based on it's state, we can have a state variable in the Object and use if-else condition block to perform different actions based on the state. State pattern is used to provide a systematic and lose-coupled way to achieve this through Context and State implementations.

Check out **State Pattern** post for implementation details with example program.

8. Visitor Pattern

Visitor pattern is used when we have to perform an operation on a group of similar kind of Objects. With the help of visitor pattern, we can move the operational logic from the objects to another class.

For example, think of a Shopping cart where we can add different type of items (Elements), when we click on checkout button, it calculates the total amount to be paid. Now we can have the calculation logic in item classes or we can move out this logic to another class using visitor pattern. Let's implement this in our example of visitor pattern. Check out **Visitor Pattern** post for implementation details.

9. Interpreter Pattern

is used to defines a grammatical representation for a language and provides an interpreter to deal with this grammar.

The best example of this pattern is java compiler that interprets the java source code into byte code that is understandable by JVM. Google Translator is also an example of interpreter pattern where the input can be in any language and we can get the output interpreted in another language.

Check out **Interpreter Pattern** post for example program.

10. Iterator Pattern

Iterator pattern in one of the behavioral pattern and it's used to provide a standard way to traverse through a group of Objects. Iterator pattern is widely used in Java Collection

Framework where Iterator interface provides methods for traversing through a collection.

Iterator pattern is not only about traversing through a collection, we can provide different kind of iterators based on our requirements. Iterator pattern hides the actual implementation of traversal through the collection and client programs just use iterator methods. Check out **Iterator Pattern** post for example program and implementation details.

11. Memento Pattern

Memento design pattern is used when we want to save the state of an object so that we can restore later on. Memento pattern is used to implement this in such a way that the saved state data of the object is not accessible outside of the object, this protects the integrity of saved state data.

Memento pattern is implemented with two objects – Originator and Caretaker. Originator is the object whose state needs to be saved and restored and it uses an inner class to save the state of Object. The inner class is called Memento and its private, so that it can't be accessed from other objects.

Check out **Memento Pattern** for sample program and implementation details.

That's all for different design patterns in java, this post intent is to provide an index to browse all of them easily.