Design Patterns

Need of Design Pattern

- The design pattern is created to fix known problems, they can be predicted before they become visible during the implementation process.
- The design pattern speeds up the development process.
- Standardization related to the design pattern is also very useful to facilitate code readability.
- The design pattern is useful when moving from an analysis model to a development model.

Types of Design Patterns

Creational: The design patterns that deal with the creation of an object.

Structural: The design patterns in this category deals with the class structure such as Inheritance and Composition.

Behavioral: This type of design patterns provide solution for the better interaction between objects, how to provide lose coupling, and flexibility to extend easily in future.

- **Creational Design patterns:**
  1. Factory
  2. Abstract Factory
  3. Builder
- **Structural Design Patterns:**
  1. Adapter
  2. Decorator
  3. Proxy
- **Behavioral Design Patterns:**
  1. Chain of Responsibility
  2. Strategy
- **Miscellaneous Design Patterns:**
  1. DAO Design pattern
  2. Dependancy Injection Pattern
  3. MVC Pattern

Factory Design Pattern

- The factory design pattern is used when we have a superclass with multiple sub-classes and based on input, we need to return one of the sub-class.
- Super class in factory design pattern can be an interface, abstract class or a normal java class.
- This pattern takes out the responsibility of the instantiation of a class from the client program to the factory class.

Some important points about Factory Design Pattern method are:

- We can keep Factory class Singleton or we can keep the method that returns the subclass as static.
- Notice that based on the input parameter, different subclass is created and returned.

Factory Design Pattern Advantages

- Factory design pattern provides approach to code for interface rather than implementation.
- Factory pattern removes the instantiation of actual implementation classes from client code. Factory pattern makes our code more robust, less coupled and easy to extend. For example, we can easily change PC class implementation because client program is unaware of this.
- Factory pattern provides abstraction between implementation and client classes through inheritance.

Factory Design Pattern Examples in JDK

- java.util.Calendar, ResourceBundle and NumberFormat getInstance() methods uses Factory pattern.
- valueOf() method in wrapper classes like Boolean, Integer etc.

Abstract Factory pattern

- Abstract Factory pattern is almost similar to Factory Pattern except the fact that its more like factory of factories.

Abstract Factory Pattern Benefits

- Abstract Factory pattern is “factory of factories” and can be easily extended to accommodate more products.
- Abstract Factory pattern is robust and avoid conditional logic of Factory pattern.

Builder Pattern

- Builder pattern was introduced to solve some of the problems with Factory and Abstract Factory design patterns when the Object contains a lot of attributes. There are three major issues with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.

- Too Many arguments to pass from client program to the Factory class that can be error prone because most of the time, the type of arguments are same and from client side its hard to maintain the order of the argument.
- Some of the parameters might be optional but in Factory pattern, we are forced to send all the parameters and optional parameters need to send as NULL.
- If the object is heavy and its creation is complex, then all that complexity will be part of Factory classes that is confusing.
- We can solve the issues with large number of parameters by providing a constructor with required parameters and then different setter methods to set the optional parameters. The problem with this approach is that the Object state will be inconsistent until unless all the attributes are set explicitly. 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.

Let’s see how we can implement builder design pattern in java.

- First of all you need to create a static nested class and then copy all the arguments from the outer class to the Builder class which are optional. We should follow the naming convention and if the class name is Computer then builder class should be named as ComputerBuilder.
- Java Builder class should have a public constructor with all the required attributes as parameters.
- Java Builder class should have methods to set the optional parameters and it should return the same Builder object after setting the optional attribute.
- The final step is to provide a build() method in the builder class that will return the Object needed by client program. For this we need to have a private constructor in the Class with Builder class as argument.

Some of the builder pattern example in Java classes are

- java.lang.StringBuilder#append() (unsynchronized)
- java.lang.StringBuffer#append() (synchronized)

Adapter Design Pattern

- The adapter design pattern is a structural design pattern that allows two unrelated/uncommon interfaces to work together. In other words, the adapter pattern makes two incompatible interfaces compatible without changing their existing code.
- Interfaces may be incompatible, but the inner functionality should match the requirement.
- The adapter pattern is often used to make existing classes work with others without modifying their source code.
- Adapter patterns use a single class (the adapter class) to join functionalities of independent or incompatible interfaces/classes.
- The adapter pattern also is known as the wrapper, an alternative naming shared with the decorator design pattern.
- This pattern converts the (incompatible) interface of a class (the adaptee) into another interface (the target) that clients require.
- The adapter pattern also lets classes work together, which, otherwise, couldn't have worked, because of the incompatible interfaces.

Types Of Adapter Pattern

Class adapters:-

Class adapters use inheritance and can wrap a class only. We can't wrap an interface since, by definition, it must be derived from some base class.

Object adapters:-

Object adapters use the composition and can wrap classes as well as interfaces. It contains a reference to the class or interfaces object instance. The object adapter is the easier one and can be applied in most of the scenarios.

Two Ways Adapter

- We can also create the adapter by implementing the target and the adaptee. That approach is known as the two ways adapter.

- The two-ways adapters are adapters that implement both interfaces of the target and adaptee. The adapted object can be used as the target in new systems dealing with target classes or as the adaptee in other systems dealing with the adaptee classes. The use of the two ways adapter is bit rare.

Decorator Design 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.

We need to have following types to implement decorator design pattern

Component Interface - The interface or abstract class defining the methods that will be implemented.

Component Implementation - The basic implementation of the component interface.

Decorator - Decorator class implements the component interface and it has a HAS-A relationship with the component interface. The component variable should be accessible to the child decorator classes, so we will make this variable protected.

Concrete Decorators - Extending the base decorator functionality and modifying the component behavior accordingly.

Decorator Design Pattern - Important Points

- Decorator design pattern is helpful in providing runtime modification abilities and hence more flexible. Its easy to maintain and extend when the number of choices are more.
- The disadvantage of decorator design pattern is that it uses a lot of similar kind of objects (decorators).
- Decorator pattern is used a lot in Java IO classes, such as FileReader, BufferedReader etc.

Proxy Design Pattern

- Proxy design pattern intent is: Provide a surrogate or placeholder for another object to control access to it.
- proxy design 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. Here a proxy class can be created to provide controlled access of the program.

Chain of Responsibility Design Pattern

- Chain of responsibility pattern is used to achieve loose 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.
- Chain of Responsibility Pattern Example in JDK:
- Let’s see the example of chain of responsibility pattern in JDK and then we will proceed to implement a real life example of this pattern. 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.

Chain of Responsibility Design Pattern Important Points

- Client doesn’t know which part of the chain will be processing the request and it will send the request to the first object in the chain.
- Each object in the chain will have it’s own implementation to process the request, either full or partial or to send it to the next object in the chain.
- Every object in the chain should have reference to the next object in chain to forward the request to, its achieved by java composition.
- Creating the chain carefully is very important otherwise there might be a case that the request will never be forwarded to a particular processor or there are no objects in the chain who are able to handle the request.
- Chain of Responsibility design pattern is good to achieve lose coupling but it comes with the trade-off of having a lot of implementation classes and maintenance problems if most of the code is common in all the implementations.

Strategy Pattern

- Strategy pattern is also known as Policy Pattern.
- We define multiple algorithms and let client application pass the algorithm to be used as a parameter.
- One of the best example of strategy 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.

Dependency Injection Design Pattern

- Java Dependency Injection design pattern allows us to remove the hard-coded dependencies and make our application loosely coupled, extendable and maintainable.
- We can implement dependency injection in java to move the dependency resolution from compile-time to runtime.

Dependency Injection in java requires at least the following:

- Service components should be designed with base class or interface.
- It's better to prefer interfaces or abstract classes that would define contract for the services.
- Consumer classes should be written in terms of service interface.
- Injector classes that will initialize the services and then the consumer classes.

```java
package com.journaldev.java.legacy;

public class EmailService {

	public void sendEmail(String message, String receiver){
		//logic to send email
		System.out.println("Email sent to "+receiver+ " with Message="+message);
	}
}
```

`EmailService` class holds the logic to send an email message to the recipient email address. Our application code will be like below.

```java
package com.journaldev.java.legacy;

public class MyApplication {

	private EmailService email = new EmailService();

	public void processMessages(String msg, String rec){
		//do some msg validation, manipulation logic etc
		this.email.sendEmail(msg, rec);
	}
}
```

Our client code that will use `MyApplication` class to send email messages will be like below.

```
package com.journaldev.java.legacy;

public class MyLegacyTest {

        public static void main(String[] args) {
                MyApplication app = new MyApplication();
                app.processMessages("Hi Pankaj", "pankaj@abc.com");
        }

}
```

At first look, there seems nothing wrong with the above implementation. But above code logic has certain limitations.

- `MyApplication` class is responsible to initialize the email service and then use it. This leads to hard-coded dependency. If we want to switch to some other advanced email service in the future, it will require code changes in MyApplication class. This makes our application hard to extend and if email service is used in multiple classes then that would be even harder.
- If we want to extend our application to provide an additional messaging feature, such as SMS or Facebook message then we would need to write another application for that. This will involve code changes in application classes and in client classes too.
- Testing the application will be very difficult since our application is directly creating the email service instance. There is no way we can mock these objects in our test classes.

Benefits of Java Dependency Injection

- Separation of Concerns
- Boilerplate Code reduction in application classes because all work to initialize dependencies is handled by the injector component
- Configurable components makes application easily extendable
- Unit testing is easy with mock objects

One of the best example of setter dependency injection is Struts2 Servlet API Aware interfaces. Whether to use Constructor based dependency injection or setter based is a design decision and depends on your requirements. For example, if my application can't work at all without the service class then I would prefer constructor based DI or else I would go for setter method based DI to use it only when it's really needed. **Dependency Injection in Java** is a way to achieve **Inversion of control** (**IoC**) in our application by moving objects binding from compile time to runtime. We can achieve IoC through Factory Pattern, Template Method Design Pattern, Strategy Pattern and Service Locator pattern too. **Spring Dependency Injection**, **Google Guice** and **Java EE CDI** frameworks facilitate the process of dependency injection through use of Java Reflection API and java annotations. All we need is to annotate the field, constructor or setter method and configure them in configuration xml files or classes.

Disadvantages of Java Dependency Injection

- If overused, it can lead to maintenance issues because the effect of changes are known at runtime.
- Dependency injection in java hides the service class dependencies that can lead to runtime errors that would have been caught at compile time.

DAO Design Pattern

- DAO stands for Data Access Object.
- DAO Design Pattern is used to separate the data persistence logic in a separate layer.
- This way, the service remains completely in dark about how the low-level operations to access the database is done.
- This is known as the principle of Separation of Logic.

With DAO design pattern, we have following components on which our design depends:

- The model which is transferred from one layer to the other.
- The interfaces which provides a flexible design.
- The interface implementation which is a concrete implementation of the persistence logic

Advantages of DAO pattern

There are many advantages for using DAO pattern. Let’s state some of them here:

- While changing a persistence mechanism, service layer doesn’t even have to know where the data comes from. For example, if you’re thinking of shifting from using MySQL to MongoDB, all changes are needed to be done in the DAO layer only.
- DAO pattern emphasis on the low coupling between different components of an application. So, the View layer have no dependency on DAO layer and only Service layer depends on it, even that with the interfaces and not from concrete implementation.
- As the persistence logic is completely separate, it is much easier to write Unit tests for individual components. For example, if you’re using JUnit and Mockito for testing frameworks, it will be easy to mock the individual components of your application.
- As we work with interfaces in DAO pattern, it also emphasizes the style of “work with interfaces instead of implementation” which is an excellent OOPs style of programming.

MVC Design Pattern

- The **Model View Controller** (MVC) design pattern specifies that an application consist of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects.
- MVC is more of an architectural pattern, but not for complete application. MVC mostly relates to the UI / interaction layer of an application. You’re still going to need business logic layer, maybe some service layer and data access layer.

MVC Components

- The **Model** contains only the pure application data, it contains no logic describing how to present the data to a user.
- The **View** presents the model's data to the user. The view knows how to access the model's data, but it does not know what this data means or what the user can do to manipulate it.
- The **Controller** exists between the view and the model. It listens to events triggered by the view (or another external source) and executes the appropriate reaction to these events. In most cases, the reaction is to call a method on the model. Since the view and the model are connected through a notification mechanism, the result of this action is then automatically reflected in the view.

Advantages

- Multiple developers can work simultaneously on the model, controller and views.
- MVC enables logical grouping of related actions on a controller together. The views for a specific model are also grouped together.
- Models can have multiple views.

Disadvantages

- The framework navigation can be complex because it introduces new layers of abstraction and requires users to adapt to the decomposition criteria of MVC.
- Knowledge on multiple technologies becomes the norm. Developers using MVC need to be skilled in multiple technologies.

Thank You