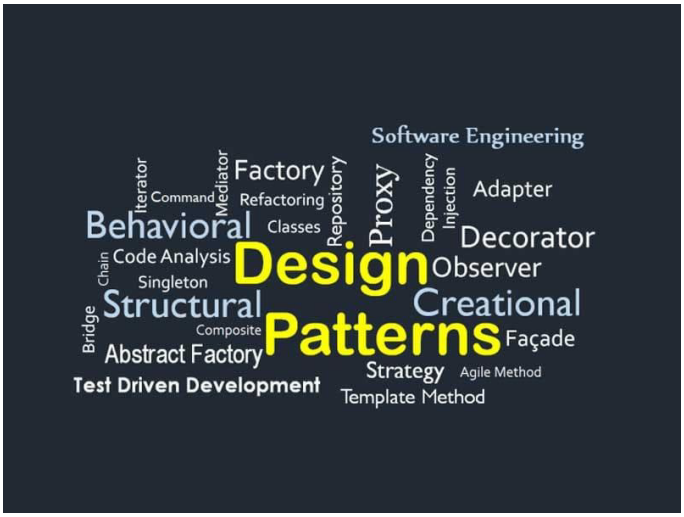
Java Design Patterns



By

Praveen Oruganti

Blog: https://praveenoruganti.blogspot.com

Facebook Group: https://www.facebook.com/groups/2340886582907696/

Github repo: https://github.com/praveenoruganti

Why Design Patterns?

1. Design patterns are used for solving recursive problems in a software application design.
2. A design pattern is a description for how to solve a recursive problem.
3. Design patterns are not a technology or a tool or a language or a platform or a framework.
4. Design patterns are effective proven solutions for recursive problems.

How many Design Patterns?

SUN Microsystems constituted a group with four professional with the name of Gang of Four (GOF) to find effective solutions for the recursive problems.

According to GOF, they found 23 design patterns as effective solutions for re-occurring problems.

Java design patterns are divided into 3 i.e. Creational, Structural and Behavioral.

Creational patterns

Creational patterns are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case. Ex. Singleton pattern – used to restrict the creation of an object to one instance.

Here with the Creational Design Patterns
1. Factory Pattern - subclasses are responsible to create the instance of the class.
2. Abstract Factory Pattern - A Factory of Factories
3. Singleton Pattern
4. Prototype Pattern
5. Builder Pattern.
6. Object Pool Pattern.

Structural patterns

Eases the design by identifying a way to realize relationship between entities.
Ex. Flyweight pattern – high quantity objects to share common properties object to save space.

Here with the Structural Design Patterns.

1. Adapter
2. Bridge

3. Filter
4. Composite
5. Decorator
6. Facade
7. Flyweight
8. Proxy

Behavioral patterns

A solution for the ideal interaction between objects and how to provide loose coupling and flexibility
Ex. Observer pattern – get notified of the change of the state of an object being observed (observer/subject)

Here with the behavioral patterns

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Momento
7. Observer
8. State
9. Strategy
10. Template method
11. Visitor

Singleton Design Pattern

It is a type of Creational Design Pattern.

This is a pattern which restricts the instantiation of a class to one object.

We can make singleton class using below 5 approaches and my favourite is static inner class and Enum.

1. Using Eager Initialization

```
package com.praveen.designpatterns.creational.singleton;

public class EagerSingletonExample {

private static EagerSingletonExample instance = new EagerSingletonExample();

private EagerSingletonExample() {

}

public static EagerSingletonExample getInstance() {
return instance;
}

public static void main(String args[]) {
EagerSingletonExample ese1 = EagerSingletonExample.getInstance();
EagerSingletonExample ese2 = EagerSingletonExample.getInstance();
if (ese1.equals(ese2)) {
System.out.println("Single Object");
} else {
System.out.println("Multiple Objects");
}
}

}
```

2. Using Lazy Initialization

```
package com.praveen.designpatterns.creational.singleton;

public class LazySingletonExample {
private static LazySingletonExample instance =null;

private LazySingletonExample() {

}
```

```
public static LazySingletonExample getInstance() {
if(instance==null) {
instance= new LazySingletonExample();
}
return instance;
}
public static void main(String args[]) {
LazySingletonExample lse1 = LazySingletonExample.getInstance();
LazySingletonExample lse2 = LazySingletonExample.getInstance();
if (lse1.equals(lse2)) {
System.out.println("Single Object");
} else {
System.out.println("Multiple Objects");
}
}
}
```

3. Using Double checked Locking

```
package com.praveen.designpatterns.creational.singleton;
public class DoubleCheckedLockSingletonExample {
private static DoubleCheckedLockSingletonExample instance =null;
private DoubleCheckedLockSingletonExample() {
}
public static DoubleCheckedLockSingletonExample getInstance() {
if(instance==null) {
synchronized(DoubleCheckedLockSingletonExample.class) {
if(instance==null) {
instance= new DoubleCheckedLockSingletonExample();
}
}
}
return instance;
}
public static void main(String args[]) {
DoubleCheckedLockSingletonExample dclse1 =
DoubleCheckedLockSingletonExample.getInstance();
DoubleCheckedLockSingletonExample dclse2 =
DoubleCheckedLockSingletonExample.getInstance();
if (dclse1.equals(dclse2)) {
System.out.println("Single Object");
} else {
System.out.println("Multiple Objects");
```

```
}
}
}
```

4. Using static inner static class

```
package com.praveen.designpatterns.creational.singleton;
public class InnerClassSingletonExample {
private InnerClassSingletonExample() {
}
public static class SingletonHolder {
private static InnerClassSingletonExample instance = new
InnerClassSingletonExample();
}
public static InnerClassSingletonExample getInstance() {
return SingletonHolder.instance;
}
public static void main(String[] args) {
InnerClassSingletonExample icse1= InnerClassSingletonExample.getInstance();
InnerClassSingletonExample icse2= InnerClassSingletonExample.getInstance();
if(icse1.equals(icse2)) {
System.out.println("Single Object");
}else {
System.out.println("Multiple Objects");
}
}
}
```

5. Using Enum

```
package com.praveen.designpatterns.creational.singleton;
public enum EnumSingletonExample {
INSTANCE;
public static void main(String args[]) {
EnumSingletonExample ese1 = EnumSingletonExample.INSTANCE;
EnumSingletonExample ese2 = EnumSingletonExample.INSTANCE;
if (ese1.equals(ese2)) {
System.out.println("Single Object");
} else {
System.out.println("Multiple Objects");
}
}
}
```

You can checkout code in my git repository(https://github.com/…/pr…/designpatterns/creational/singleton)

Benefits

1. Limits the objects creations
2. Reduces the memory required
3. Reduces garbage collection
4. Helps in Lazy initialization. That means, an object is not created until we need it and call the getInstance() method
5. Saves memory as the instances are reused

Limitations

1. Same object is shared hence the object must be immutable
2. We can not have class level variables,If we use they will be overridden by other client
3. Extra code need to write for making it singleton this increases complexity
4. The getInstance() method is not thread safe, but can be overcome by using synchronized keyword

Singleton Pattern is generally used for logging,driver objects,caching and threadpool.

Singleton Design Pattern Example in JDK

Some of the singleton pattern example in Java classes are;
1. java.lang.Runtime.getRuntime(): This method gives Runtime class that has only one instance in a JVM.
2. java.lang.System.getSecurityManager(): This method returns a SecurityManager for the current platform.
3. java.awt.Desktop.getDesktop()

Builder Design Pattern

It is a type of Creational Design 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.

Static factories and constructors share a limitation, they dont scale well to large number of optional paramaters.

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.

Lets start coding

```
package com.praveen.designpatterns.creational.builder;

public class Employee {
//All final attributes
private final int empId;
private final String empName;
```

```
private final double empSalary;
private final String empAddress;

private Employee(EmployeeBuilder builder) {
this.empId = builder.empId;
this.empName = builder.empName;
this.empSalary = builder.empSalary;
this.empAddress = builder.empAddress;
}

//All getter, and NO setter to provde immutability
public int getEmpId() {
return empId;
}

public String getEmpName() {
return empName;
}

public double getEmpSalary() {
return empSalary;
}

public String getEmpAddress() {
return empAddress;
}

public static class EmployeeBuilder {
private int empId;
private String empName;
private double empSalary;
private String empAddress;

EmployeeBuilder() {

}

public EmployeeBuilder empId(int empId) {
this.empId = empId;
return this;
}

public EmployeeBuilder empName(String empName) {
this.empName = empName;
return this;
}

public EmployeeBuilder empSalary(double empSalary) {
this.empSalary = empSalary;
return this;
}
```

```
public EmployeeBuilder empAddress(String empAddress) {
this.empAddress = empAddress;
return this;
}
//Return the finally constructed Employee object
public Employee build() {
Employee emp = new Employee(this);
return emp;
}
}
@Override
public String toString() {
return "Employee [empId=" + empId + ", empName=" + empName + ", empSalary=" +
empSalary + ", empAddress="
+ empAddress + "]";
}
public static void main(String args[]) {
Employee emp = new
Employee.EmployeeBuilder().empId(149903).empName("Praveen").empSalary(100000
0d)
.empAddress("Hyderabad").build();
System.out.println(emp);
}
}
```

Output

Employee [empId=149903, empName=Praveen, empSalary=1000000.0, empAddress=Hyderabad]

You can also check out code from my got repository(https://github.com/…/prav…/designpatterns/creational/builder)

Benefits

1. Solves the multiple constructor
2. problem(telescoping constructor)
3. Static inner class(builder class)
4. Internally required constructor
5. removes the need for setters

Limitations

1. Immutable
2. Inner static class
3. Design first
4.Complex

Builder Design Pattern Example in JDK

Some of the builder pattern example in Java classes are;
1. java.lang.StringBuilder#append() (unsynchronized)
2. java.lang.StringBuffer#append() (synchronized)
3. DocumentBuilder
4. Locale.Builder

Factory Design Pattern

It is a type of Creational Design Pattern.

It doesn't exposes instantiation or creation logic instead subclass create the object.

Lets start coding

NotificationExecutor

```
package com.praveen.designpatterns.creational.factory;

public interface NotificationExecutor {
public void executeNotification();
}
```

EmailNotificationExecutor

```
package com.praveen.designpatterns.creational.factory;

public class EmailNotificationExecutor implements NotificationExecutor {

@Override
public void executeNotification() {
System.out.println("Email notification sent");
}
}
```

SMSNotificationExecutor

```
package com.praveen.designpatterns.creational.factory;

public class SMSNotificationExecutor implements NotificationExecutor {

@Override
public void executeNotification() {
System.out.println("SMS notification sent.");
}
}
```

NoNotificationExecutor

```
package com.praveen.designpatterns.creational.factory;

public class NoNotificationExecutor implements NotificationExecutor {
private String notificationType;
```

```
NoNotificationExecutor(String notificationType) {
this.notificationType = notificationType;
}
@Override
public void executeNotification() {
System.out.println("Notification Implementation not defined for "+ notificationType);
}
}
```

NotificationExecutorFactory

```
package com.praveen.designpatterns.creational.factory;
public class NotificationExecutorFactory {
public static NotificationExecutor getNotificationExecutor(String executorType) {
switch (executorType) {
case "Email":
return new EmailNotificationExecutor();
case "SMS":
return new SMSNotificationExecutor();
default:
return new NoNotificationExecutor(executorType);
}
}
}
```

NotificationSender

```
package com.praveen.designpatterns.creational.factory;
public class NotificationSender {
public static void main(String[] args) {
NotificationExecutorFactory.getNotificationExecutor("Email").executeNotification();
NotificationExecutorFactory.getNotificationExecutor("SMS").executeNotification();
NotificationExecutorFactory.getNotificationExecutor("FTP").executeNotification();
}
}
```

Output

```
Email notification sent
SMS notification sent.
Notification Implementation not defined for FTP
```

You can also check the code in my git repository(https://github.com/…/prav…/designpatterns/creational/factory)

Benefits
1. Creation of different types of objects is possible at run time
2. It separates the object creation logic from the object usage logic
3.Removes duplicate code
Thus, makes changing or addition to object creation easier

Limitations
1. The different types of objects created must have the same parent class
2. The addition of new classes and interfaces could increase the complexity of the code

Factory Design Pattern Example in JDK
Some of the factory pattern example in Java classes are;
1. Calender.getInstance()
2. NumberFormat.getInstance()

Abstract Factory Design Pattern

It is a type of Creational Design Pattern.

Abstract factory design pattern is used to mange different object types of same family. All the object should belong to same family but they are of different categories.

You can check the code for this pattern in my git repository(https://github.com/…/designpatte…/creational/abstractfactory)

Benefits
1.It helps to group related objects or functions
2.Also, reduces errors of mixing of objects or functions from different groups
3.Helps to abstract code so that user don't need to worry about object creations

Limitations
1.Only useful when we have to group processes or objects
2.Before getting object or calling the function we need to get the factory which adds one more processes
3.Adds more classes and abstraction hence code could become complex

Abstract Factory Design Pattern Example in JDK
Some of the Abstract factory pattern example in Java classes are;
1.javax.xml.parsers.DocumentBuilderFactory#newInstance()
2.javax.xml.transform.TransformerFactory#newInstance()
3.javax.xml.xpath.XPathFactory#newInstance()

Bridge Design Pattern

It is a type of Structural Design Pattern.

It allows you to separate the abstraction from implementation.

A classic example of bridge is Drivers.

You can check the code for this pattern in my repository(https://github.com/…/prave…/designpatterns/structural/bridge)

Benefits

1.It enables the separation of implementation from the interface.
2.It improves the extensibility.
3.It allows the hiding of implementation details from the client.

Adapter Design Pattern

It is a type of structural design pattern.

It converts the interface of a class into another interface that a client expects.

Lets start coding

ITarget

```
package com.praveen.designpatterns.structural.adapter;

public interface ITarget {
void request();

}
```

Adaptee

```
package com.praveen.designpatterns.structural.adapter;

public class Adaptee {

public void specificRequest() {
System.out.println("In Adaptee");
}
}
```

Adapter

```
package com.praveen.designpatterns.structural.adapter;

public class Adapter implements ITarget {

private Adaptee adaptee;

Adapter(Adaptee adaptee) {
this.adaptee = adaptee;
}
@Override
public void request() {
System.out.println("Using Adapter");
this.adaptee.specificRequest();
}
}
```

Main

```
package com.praveen.designpatterns.structural.adapter;

public class Main {

public static void main(String args[]) {
ITarget target= new Adapter(new Adaptee());
target.request();
}

}
```

You can also check the code in my repository(https://github.com/…/prav…/designpatterns/structural/adapter)

Benefits
1.It allows two or more previously incompatible objects to interact.
2.It allows reusability of existing functionality.
Limitations
3.No new functionalities can be added
4.Multiple Adapters difficult to maintain

Adapter Design Pattern in JDK

1.java.util.Arrays#asList()
2.java.util.Collections#list()
3.java.util.Collections#enumeration()
4.java.io.InputStreamReader(InputStream) (returns a Reader)
5.java.io.OutputStreamWriter(OutputStream) (returns a Writer)
6.javax.xml.bind.annotation.adapters.XmlAdapter#marshal() and #unmarshal()

Adapter vs Bridge

Adapter makes thing work after they are designed whereas Bridge makes them work before they are.

Decorator Pattern

It is a type of Structural Design Pattern.

A Decorator Pattern says that just "attach a flexible additional responsibilities to an object dynamically".

In other words, The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime.

The Decorator Pattern is also known as Wrapper.

You can check the code for this pattern in my repository(https://github.com/…/pr…/designpatterns/structural/decorator)

Limitations

1.New class for every feature
2.Number of Objects are more hence more complexity
3.More complex for client

Decorator pattern examples in JDK

1.All subclasses of java.io.InputStream, OutputStream, Reader and Writer have a constructor taking an instance of same type.
2.java.util.Collections, the checkedXXX(), synchronizedXXX() and unmodifiableXXX() methods.
3.javax.servlet.http.HttpServletRequestWrapper and HttpServletResponseWrapper javax.swing.JScrollPane

Facade Design Pattern

It is a type of Structural Design Pattern.

It just provides a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client.

Practically, every Abstract Factory is a type of Facade.

Benefits

1.It shields the clients from the complexities of the sub-system components.
2.It promotes loose coupling between subsystems and its clients.

Flyweight Design Pattern

It is a type of Structural Design Pattern.

To reuse already existing similar kind of objects by storing them and create new object when no matching object is found

This is primarily used to reduce the number of objects created and to decrease the memory foot print there by increase in performance.

It is same as Factory pattern but provides you with a basic caching mechanism for created immutable instances.

When you ask getInstance(Property p) it checks if property exists
a) For factory , if property doesnt exist it throws exception
b) For Flyweight, Creates the object and returns.

Modern web browsers uses this pattern like loading similar pages.

String intern() follows Flyweight Pattern

Benefits

1.It reduces the number of objects.
2.It reduces the amount of memory and storage devices required if the objects are persisted.

Flyweight Pattern example in JDK

java.lang.Integer#valueOf(int) (also on Boolean, Byte, Character, Short, Long and BigDecimal)

Composite Pattern

It is a type of Structural Design Pattern.

This is used when tree structure is present.

Compose objects into tree structure to represent part- whole hierarchies.

Composite lets client treat individual objects and compositions of object uniformly.

Proxy Pattern

It is a type of Structural Design Pattern.

Provides the control for accessing the original object.

It provides the protection to the original object from outside world for example RMI Stubs/Skeleton.

Filter Pattern

It is a type of Structural design pattern

Filter design pattern is used for building a criteria to filter items or objects dynamically. You can choose your own criteria and apply it on your objects to filter out the desired objects.

Strategy Pattern

It is a type of Behavioral Design Pattern.

It is very useful for implementing a family of Algorithms.

With Strategy Pattern we can select the algorithm at runtime. We can use to select sorting strategy based on input it selects for execution, saving files etc.

State Pattern

It is a type of Behavioral Design Pattern.

It provides a mechanism to change the behavior of an object based on the object's state.

Strategy vs State Design Pattern

Strategy Pattern decides on how to perform some action whereas State Pattern decides on when to perform them.

Chain of responsibility Pattern

It is a type of Behavioral Design Pattern.

Give more than one object an opportunity to handle a request by linking receiving objects together.

For example, customer service helpdesk

Momento Pattern

It is a type of Behavioral Design Pattern.

It is used to restore state of an object to its previous state.

For example, Text Editor

Observer Pattern

It is a type of Behavioral Design Pattern.

Observer is a behavioral design pattern. It specifies communication between objects: observable and observers. An observable is an object which notifies observers about the changes in its state.

For example, a news agency can notify channels when it receives news. Receiving news is what changes the state of the news agency, and it causes the channels to be notified.

Visitor Pattern

It is a type of Behavioral Design Pattern.

The purpose of a Visitor pattern is to define a new operation without introducing the modifications to an existing object structure.

Consequently, we'll make good use of the Open/Closed principle as we won't modify the code, but we'll still be able to extend the functionality by providing a new Visitor implementation.

Iterator Pattern

It is a type of Behavioral Design Pattern.

Provides a way to access the elements of an aggregate object without exposing its underlying representation.

Template Method Pattern

It is a type of Behavioral Design Pattern.

Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithms structure.

Mediator Pattern

It is a type of Behavioral Design Pattern.

Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other. Allows for the actions of each object set to vary independently of one another.

Command Pattern

It is a type of Behavioral Design Pattern.

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

Interpreter Pattern

It is a type of Behavioral Design Pattern.

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.