13. Design Patterns - day 1 of 2 (1)

Design Patterns

Design Patterns are reusable solutions to commonly occuring problems(in the context of software design). Design patterns were started as best practices that were applied again and again to similar problems encountered in different contexts. They become popular after they were collected, in a formalized form, in the Gang Of Four book.

Real-World Example

Do you love food? (Everyone loves that)… Which one is your favorite restauran? You might have tried different cuisines and you might have experimented with different places as well. If there is a favorite restaurant in your list where most of the time you love to eat your favorite dish then what's the reason behind that?. Of course, the **experienced chef** of that restaurant might be using a **specific technique** to prepare the dish. What if you also want to prepare the same food with the same test at home? what do you need to do now? You need to follow the same approach or technique used by the experienced chef. They might have **tried many recipes** and they might have **changed their approach** to prepare that dish. Finally, at one point they stopped when they learned a particular technique to prepare that specific dish and it tastes good.

Well if you also want to prepare the same dish that tastes good like in your favorite restaurant then you need to follow the same approach and techniques given by the experienced chefs or you need to approach your friend who **cook good** and prepares dishes using some specific technique.

Enough talking about food, let’s come to the design patterns. Notice the words highlighted above, **tried many recipes**, **changed their approach**, **experienced chef** and **cook really good**. The same is the case with design patterns. DPs are like some of the best practices used by chefs (**Gang of Four (GoF)**) to prepare a dish so that it tastes the best.

- Design patterns are some design practices used by experienced object-oriented software developers (experienced chef or friends in our context). They are general solutions to problems faced during software development.
- They are the solutions obtained by trial and error (trying many recipes) by numerous software developers over a substantial period.

Factory Pattern

Think about design pattens in terms of classes being 'people,' and the patterns are the ways that the people talk to each other.

So, the factory pattern is like a hiring agency. You've got someone that will need a variable number of workers. This person may know some info they need in the people they hire, but that's it.

So, when they need a new employee, they call the hiring agency and tell them what they need. Now, to actually *hire* someone, you need to know a lot of stuff - benefits, eligibility verification, etc. But the person hiring doesn't need to know any of this - the hiring agency handles all of that.

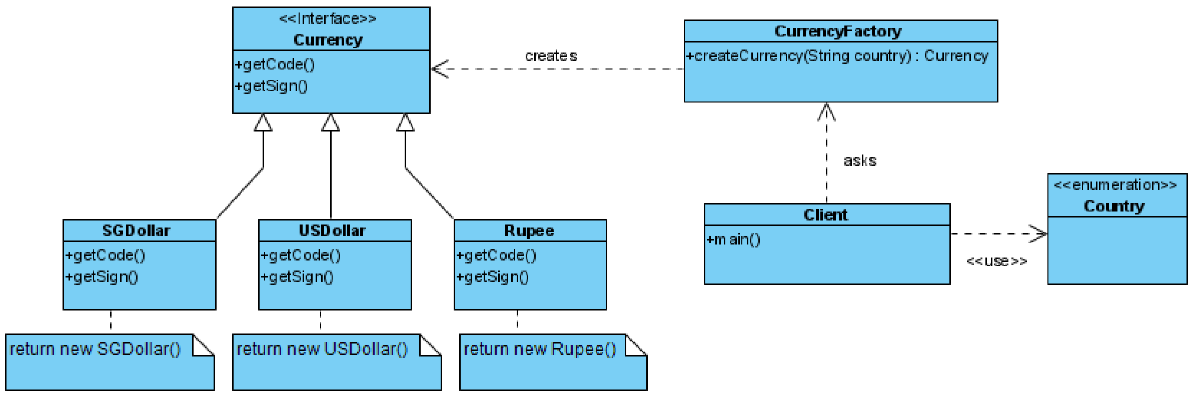
In the same way, using a Factory allows the consumer to create new objects without having to know the details of how they're created, or what their dependencies are - they only have to give the information they actually want.

```
public interface IThingFactory {
    Thing GetThing(string theString);
}
public class ThingFactory implements IThingFactory {
    public Thing GetThing(string theString) {
    return new Thing(theString, firstDependency, secondDependency);
    }
}
```

So, now the consumer of the ThingFactory can get a Thing, without having to know about the dependencies of the Thing, except for the string data that comes from the consumer.

Code Example of Factory Pattern in Java:

Let's see an example of how factory pattern is implemented in Code.We have requirement to create multiple currency e.g. INR, SGD, USD and code should be extensible to accommodate new Currency as well. Here we have made Currency as interface and all currency would be concrete implementation of Currency interface. Factory Class will create Currency based upon country and return concrete implementation which will be stored in interface type. This makes code dynamic and extensible.



Factory method comes into creational design pattern category. The main objective of the creational pattern is to instantiate an object and in Factory Pattern an interface is responsible for creating the object but the sub classes decides which class to instantiate.

Here is complete code example of Factory pattern in Java:

```
import java.util.Scanner;

// Interface Currency
interface Currency {
    String getSymbol();
}
    //Concrete Rupee Class code
```

```
class Rupee implements Currency {
    @Override
    public String getSymbol() {
        return "Rs";
    }
}
   // Concrete US Dollar code
class USDollar implements Currency {
    @Override
    public String getSymbol() {
        return "USD";
    }
}
   // Factory Class code
class CurrencyFactory {
    public static Currency createCurrency (String country) {
        if (country.equalsIgnoreCase ("India")){
            return new Rupee();

        }else if(country. equalsIgnoreCase ("US")){
            return new USDollar();
        }else{
            throw new IllegalArgumentException("No such currency");
        }
    }
}
   // Factory client code
   public class FactoryClient {
    public static void main(String args[]) {
        Scanner scanner=new Scanner(System.in);
        System.out.println("enter country name");

        String country = scanner.next();
        Currency rupee = CurrencyFactory.createCurrency(country);
        System.out.println(rupee.getSymbol());
    }
}
```

Advantages of Factory Method Pattern

1. Factory method design pattern decouples the calling class from the target class, which result in less coupled and highly cohesive code.E.g.: JDBC is a good example for this pattern; application code doesn’t need to know what database it will be used with, so it doesn’t know what database-specific driver classes it should use. Instead, it uses factory methods to get Connections, Statements, and other objects to work with. Which gives you flexibility to change your back-end database without changing your DAO layer in case you are using ANSI SQL features and not coded on DBMS specific feature?

2. Factory pattern in Java enables the subclasses to provide extended version of an object, because creating an object inside factory is more flexible than creating an object directly in the client. Since client is working on interface level any time you can enhance the implementation and return from Factory.

3. Another benefit of using Factory design pattern in Java is that it encourages consistency in Code since every time object is created using Factory rather than using different constructor at different client side.

4. Code written using Factory design pattern in Java is also easy to debug and troubleshoot because you have a centralized method for object creation and every client is getting object from same place.

Some more advantages of factory method design pattern is:

- ● Static factory method used in factory design pattern enforces use of Interface than implementation which itself is a good practice.
- ● Since factory method have return type as Interface, it allows you to replace implementation with better performance version in newer release.
- ● Another advantage of static factory method pattern is that they can cache frequently used object and eliminate duplicate object creation. Boolean.valueOf() method is good example which caches true and false boolean value.
- ● Factory method pattern offers alternative way of creating object.
- ● Factory pattern can also be used to hide information related to creation of object.

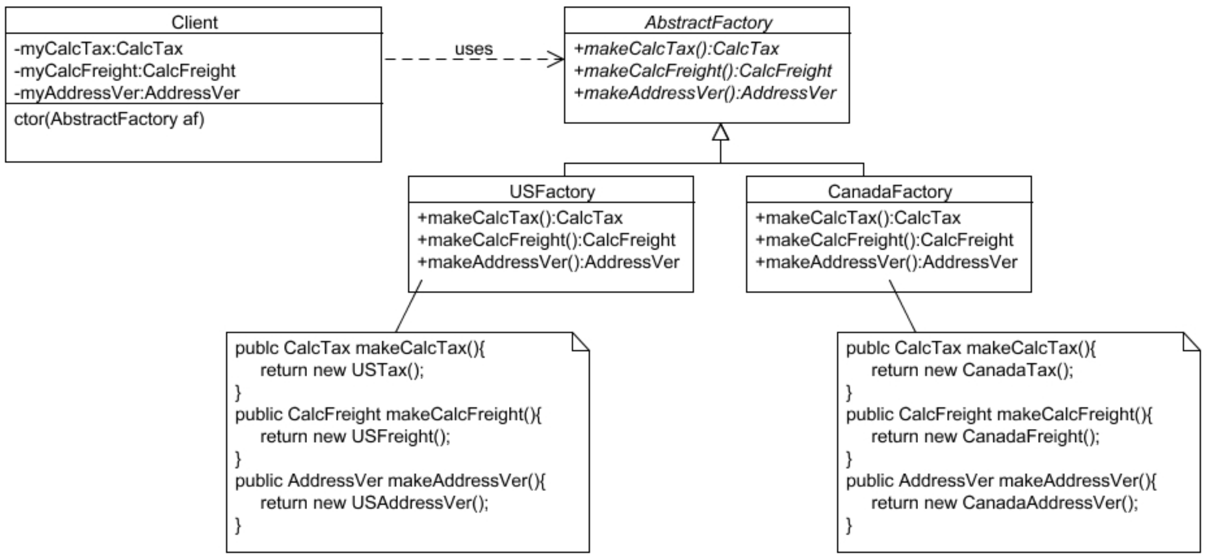
[Further Reading] Abstract Factory Pattern:

This pattern is one level of abstraction higher than factory pattern. This means that the abstract factory returns the factory of classes. Like Factory pattern returned one of the several sub-classes, this returns such factory which later will return one of the sub-classes.

Let's assume we are doing business in two countries, the United States and Canada. Because of this our application must be able to calculate taxes for both countries, as we as calculate freight charges based on the services we use in each case. Also, we must determine that an address is properly formatted given the country we're dealing with any

any point in time. To support this, we have the abstractions in place CalcTax, CalcFreight, and AddressVer, which have implementing classes for each country in each case.

The Client would be designed to take an implementation of AbstractFactory in its constructor, then use it to make all the helper objects that it needs. Note that since the client is designed to work only with a single factory, and since there is no factory version that produces, say, a USTax object and a CanadaFreight object, it is impossible for the client to obtain this illegitimate combination of helper objects.


Client
-myCalcTax:CalcTax
-myCalcFreight:CalcFreight
-myAddressVer:AddressVer
ctor(AbstractFactory af)
uses
AbstractFactory
+makeCalcTax():CalcTax
+makeCalcFreight():CalcFreight
+makeAddressVer():AddressVer
USFactory
+makeCalcTax():CalcTax
+makeCalcFreight():CalcFreight
+makeAddressVer():AddressVer
CanadaFactory
+makeCalcTax():CalcTax
+makeCalcFreight():CalcFreight
+makeAddressVer():AddressVer

```
publc CalcTax makeCalcTax(){
    return new USTax();
}
public CalcFreight makeCalcFreight(){
    return new USFreight();
}
public AddressVer makeAddressVer(){
    return new USAddressVer();
}
```

```
publc CalcTax makeCalcTax(){
    return new CanadaTax();
}
public CalcFreight makeCalcFreight(){
    return new CanadaFreight();
}
public AddressVer makeAddressVer(){
    return new CanadaAddressVer();
}
```

```
public class Client{
     private CalcTax myCalcTax;
     private CalcFreight myCalcFreight;
     private AddressVer myAddressVer;

     public Client(AbsractFactory af){
          myCalcTax = af.makeCalcTax();
          myCalcFreight = af.makeCalcFreight();
          myAddressVer = af.makeAddressVer();
     }

     // The rest of the code uses the helper objects generically
}
```

```
public abstract class AbstractFactory{
     abstract CalcTax makeCalcTax();
     abstract CalcFreight makeCalcFreight();
     abstract AddressVer makeAddressVer();
}

public class USFactory : AbtractFactory{
     public CalcTax makeCalcTax(){
          return new USCalcTax();
     }
     public CalcFreight makeCalcFreight(){
          return new USCalcFreight();
     }
     public AddressVer makeAddressVer(){
          return new USAddressVer();
}

public class CanadaFactory : AbtractFactory{
     public CalcTax makeCalcTax(){
          return new CanadaCalcTax();
     }
     public CalcFreight makeCalcFreight(){
          return new CanadaCalcFreight();
     }
     public AddressVer makeAddressVer(){
          return new CanadaAddressVer();
}
```

Singleton Design Pattern

The Singleton design pattern comes under the creational pattern type.It is one of the most simple design pattern in terms of the modelling but on the other hand this is one of the most controversial pattern in terms of complexity of usage.

Sometimes it is necessary that one and only one instance of a particular class exists in the entire Java Virtual Machine. This is achieved through the Singleton design pattern .

The Singleton pattern is widely used and has proved its usability in designing software. Although the pattern is not specific to Java, it has become a classic in Java programming.

Implementation of Singleton Design Pattern

There are many ways this can be done in Java. All these ways differs in their implementation of the pattern, but it in the end, they all achieve the same end result of a single instance.

```
public class Client {
    public static void main(String[] args) {
        //2 ways of creating object
        //1. create new object using new operator + constructor e.g. new User()
        //2. Using class name and decalring it static
        MySingleton one = MySingleton.getUniqueInstance();
        MySingleton two = MySingleton.getUniqueInstance();
        User u1=new User();
        User u2 = new User();

        System.out.println(one==two);
        System.out.println(u1==u2);
        System.out.println(MySingletonWithEnum.INSTANCE);

    }
}

class User{}
//class loading...

//eager loading/initialization

class MySingleton1{
    private static MySingleton1 uniqueInstance = new MySingleton1();

    public static MySingleton1 getUniqueInstance() {
        return uniqueInstance;
    }
}

//Lazy initialisation
class MySingleton{
    private static volatile MySingleton uniqueInstance;
    public static synchronized MySingleton getUniqueInstance() {
        if(uniqueInstance==null)
            uniqueInstance=new MySingleton();
        return uniqueInstance;
    }
}

//using enum
enum COLOR {R}
 enum MySingletonWithEnum {
    INSTANCE;
    //private MySingletonWithEnum() { System.out.println("Here"); }
}
```

0**. Eager initialization**

If the program will always need an instance, or if the cost of creating the instance is not too large in terms of time/resources, the programmer can switch to eager initialization, which always creates an instance when the class is loaded into the JVM.
problems with Eager Loading:
1. Slows down application at startup

2. Wastes a lot of memory

```
public class SingletonExample {
// Static member holds only one instance of the
// SingletonExample class
private static SingletonExample singletonInstance =
new SingletonExample();
// SingletonExample prevents any other class from instantiating


private SingletonExample() {}
// Providing Global point of accesspublic static SingletonExample getSingletonInstance() {
return singletonInstance; }
}
```

1. Lazy initialization

The Singleton pattern is implemented by creating a class with a method that creates a new instance of the object if one does not exist. If an instance already exists, it simply returns a reference to that object. To make sure that the object cannot be instantiated any other way, the constructor is made private. Although a Singleton can be implemented as

a static instance, it can also be lazily constructed, requiring no memory or resources until needed.

2. Lazy initialization with Double check locking

The above code works absolutely fine in a single threaded environment and processes the result faster because of lazy initialization. However the above code might create some abrupt behavior in the results in a multithreaded environment as in this situation multiple threads can possibly create multiple instance of the same SingletonExample class if they try to access the *getSingletonInstance()* method at the same time.

In the multithreading environment to prevent each thread to create another instance of singleton object and thus creating concurrency issue we will need to use locking mechanism. This can be achieved by synchronized keyword. By using this synchronized keyword we prevent

Thread2 or Thread3 to access the singleton instance while Thread1 is inside the method *getSingletonInstance()*.

From code perspective it means:

```
public static synchronized SingletonExample getSingletonInstance() { if (null ==
singletonInstance) {
```

```
singletonInstance = new SingletonExample(); }
```

```
return singletonInstance; }
```

So this means that every time the *getSingletonInstance()* is called it gives us an additional overhead . To prevent this expensive operation we will use **double checked locking** so that the synchronization happens only during the first call and we limit this expensive operation to happen only once.

```
public static volatile SingletonExample getSingletonInstance() { if (null ==
singletonInstance) {
```

```
synchronized (SingletonExample.class){ if (null == singletonInstance) {
```

```
singletonInstance = new SingletonExample();
```

```
} }
```

```
}
```

```
return singletonInstance; }
```

Volatile Keyword in Java

Volatile keyword is used to modify the value of a variable by different threads. It is also used to make classes thread safe. It means that multiple threads can use a method and instance of the classes at the same time without any problem. The volatile keyword can be used either with primitive type or objects.

The volatile keyword does not cache the value of the variable and always read the variable from the main memory. The volatile keyword cannot be used with classes or methods. However, it is used with variables. It also guarantees visibility and ordering. It prevents the compiler from the reordering of code.

The contents of the particular device register could change at any time, so you need the volatile keyword to ensure that such accesses are not optimized away by the compiler.

3**. Using Enum**

From Java 5 onwards there is a new way to implement Singleton design pattern, by using Enum. Enum has some distinct benefits in terms of thread-safety during instance creation, serialization guarantee by JVM and amazingly reduce amount of code which makes it perfect choice of using as Singleton class.

`/*** Singleton pattern example using Java Enum */public enum EnumSingleton{`

`INSTANCE; }`

`This,`

// `public enum MySingleton { INSTANCE; }`

has an implicit empty constructor. Making it explicit instead,

`public enum MySingleton { INSTANCE; private MySingleton() { System.out.println("Here"); } }`

If you then added another class with a main() method like

`public static void main(String[] args) { System.out.println(MySingleton.INSTANCE); }`

Disadvantages of using Enum as a singleton:

1. Coding Constraints

In regular classes, there are things that can be achieved but prohibited in enum classes. Accessing a static field in the constructor, for example. Since he's working at a special level, the programmer needs to be more careful.

2. Serializability

For singletons, it is very common to be stateful. In general, those singletons should not be serializable. There is no real example where transporting a stateful singleton from one VM to another VM makes sense; a singleton means "unique within a VM" not "unique in the universe"

If serialization really makes sense for a stateful singleton, the singleton should specify explicitly and accurately what it means in another VM to deserialize a singleton where there may already be a singleton of the same type.

> We save some lines of code on enum, but the price is too high, we have to carry all the baggage and enum limitations, we inadvertently inherit enum “features” that have unintended consequences. A disadvantage turns out to be the only alleged benefit – automatic serializability.

References:
https://javarevealed.wordpress.com/2013/07/05/factory-meth od-design-pattern/

https://www.geeksforgeeks.org/