**Design Patterns**

Design Patterns represent the best practices used by object oriented software developers. DPs are solutions to general problems that software developers faced during software development.

DPs are in general classified into 3:

1. Creational
2. Behavioral
3. Structural

**Creational:**

Creational Design Patterns deals with creation of Objects.

How efficiently you can create objects? What are the different patterns, which are related to that?

1. **Prototype**

**A fully initialized instance to be copied or cloned.**

Ex: Chess game – In chess game we have initial setup. There is a particular place where bishops have to be, there is particular place where king, queen etc.., There is specific initial setup that is there for chess game. Creating this setup in a class, an object would involve some effort. I am implementing a program for chess game. In that kind of game, I can store initial setup for chess game as a prototype. Whenever somebody asks this game, I can clone this and give it back. This is prototype Pattern.

Prototype pattern is used to create a copy of fully initialized instance. Instead of doing initial setup again and again, we reuse the initial setup, there might be other places into while we are doing software where the initial creation of an object might be really time consuming. I might talk to interface and get something. In these kinds of situations, I would go for Prototype Pattern. I would do the initialization of the object once and later I can use the object for cloning.

1. **Builder**

Used when there is a complex object structure. In OOP

Ex: Multi-course dinner- It contains drink, starter, Main course and Dessert. These are the different parts for the class Multi-course dinner. The creation of this particular class would be really complicated. In these kinds of situations, we will be go with builder pattern.

**We would try a separate object construction from its representation.** The representation of dinner is that it has a Drink, Starters (multiple options), main course and Dessert. However, to construct this we would really provide sa simple interface, so somebody wants to take a multi-course dinner would just say, I want this, this and this. We would take care constructing the entire object for that. We are separating the creation from internal representation. Even though internal representation is complex, we are providing the easy interface to the outside world.

The Builder Pattern comes in when we want to hide the complex internal structure from somebody who is using the interface to create an object. **Separate object construction from its representation.**

**Problem:** 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 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.

**Solution:** 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.

1. **Singleton**

A class of which only a single instance can exist.

Ex: President of a Country.

Look at Java; the java.lang.System is a good example of Singleton. JVM will run on only one system, so system should be Singleton, you cannot have multiple systems.

When you have multiple JVM’s running the same program, then you would have one instance per JVM. Even though we call it a singleton, a program runs on multiple JVM’s it can still have multiple instances in these different JVM’s.

**Things to remember:**

* Private Constructor
* Effective Java- Using Enum
* In JEE7, we have an annotation @Singleton, out on a class. You can customize @Startup, @PostConstruct, @DependsOn(“Other beans”)
* Difficult to Unit Test.

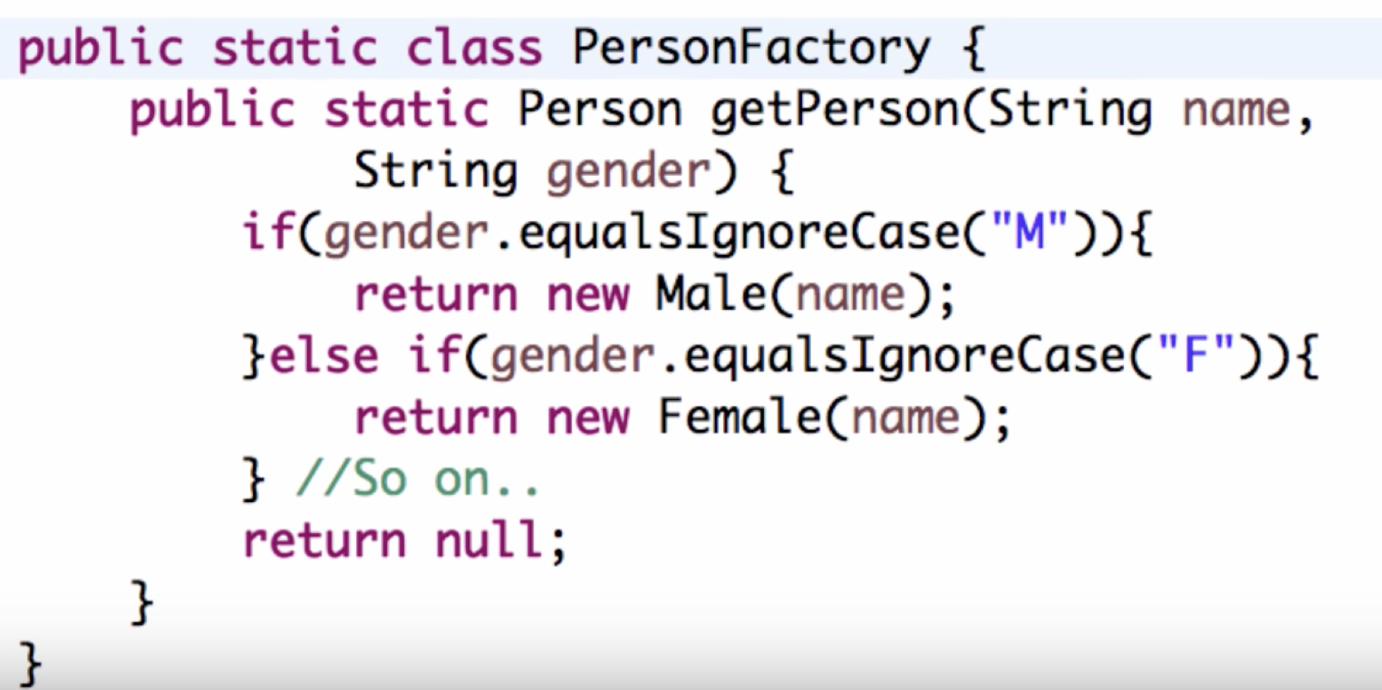
**Note:** In Spring all the beans that you create are singleton by default.

1. **Factory**

The Factory method pattern helps is to create an instance of single family of objects, and it can separate the instantiation and also separate the existence of this object. So that tomorrow, If, I want to change the Male Object to something else I want to create new class and I can do that without affecting the other method, which I am using.

Factory is nothing but manufactures an Object.

If you have any change in your application, there won’t be any impact on Client.



You have a PersonFactory where you have getPerson() method with accepting gender. Based on the value of gender, it’s creating an object. All the instantiation is present inside this particular factory. Once I have factory, I can use to create different persons.

User of getPerson() method doesn’t know Male class or Female class present. He knows only about Person class. So you can the existence of certain classes.

**Behavioral:**

The Behavioral Patterns discuss the different objects and how they interact with each other.

1. **Chain of Responsibility:** The way of passing of request between a chain of objects.

**Ex:** Loan Approval process.

**Technical Ex:** Exceptional Handling in Java

1. **Iterator:** Sequentially access the elements of a Collection.

**Ex:** Next/Previous buttons on TV Remote.

1. **Observer:** A way of notifying change to a number of classes. There is a change in one object, and it needs to be notified to multiple different objects. In those kind of situations, we go with Observer DP.

**Ex:** Online Bidding: If I want to listen to Online Bidding, I can place a bid and register to it. Whenever, somebody new places a bid, which is higher than, what I place I would get a notification, and everybody gets notification too. Everybody was bidding is observing bid. Whenever there a new bid, all the things, which are listening are notified.

**Note:** Observer Pattern is implemented by default in Java. Whichever subject you want to listen; you can Extends class called Observable. And whichever classes want to observe can implements Observer.

**Structural:**

The Structural Patterns are how the objects are composed.

Is there any inheritance relationship between them? One object present in another object? What is the Structure if the object?

1. **Proxy:**

An object representing another object.

A proxy is something authorized to do on behalf of the original object.

**Why do we need to it?**

* Do something more which is not in scope of original object
* To handle certain functionality

**Common Applications:**

* Pre Processing
* Post Processing
* Override functionality
* Caching mechanisms
* Security mechanisms
* Provide a class which will limit access to another class
* You may do this for security reasons, because an Object is intensive to create, or is accessed from a remote location

Ex: Lets assume I want to buy something at the shop. What I do, I will take my debit/credit card and swipe it. But actually where does the real amount go from? It’s from my back account.

So, the credit/debit, which I am using, is a Proxy of my bank account. I am not really using bank account, but I am using the proxy of bank account.

This is exactly the pattern of Proxy.

In EJB world, there is something called Remote object and Home object. Remote object represents the class, which is, exists on some where on different application server or different container. You use Home object to talk to it. Home objects as a proxy.

1. **Decorative**

Add responsibilities to objects dynamically.

* The Decorator allows you to modify an object dynamically
* You would use when you want the capabilities of inheritance with subclasses, but you need to add functionality at run time
* It is more flexible than inheritance
* Simplifies code because you add functionality using many simple classes
* Rather than write new code you can extend with new code

Ex: There are 10 different types of Pizza. So, we have 10 different classes. Now, I want to add 3 different toppings for each one of these Pizzas. Suppose if we go through inheritance, we have to create 30 classes. Suppose if you have different size of pizzas, again using inheritance we have to go on creating classes, which is very complex and tedious.

Toppings as dynamic things, what we do are, we decorate the pizza. We put topping around the pizza.

We would create a new topping and pass Pizza as parameter to it. That’s the decorative pattern. So, we don’t need to create a new type of pizza for toppings, we are decorating on the existing pizza and put topping on top of it.

Another Ex: Adding discount to an order: There is a black Friday sale. There is an order, I want to add discount on top of it. The discount is added as a decoration on the Order.

**Good Example in Java I/O is:**

LineNumberInputStream(BufferedInputStream(FileInputStream))

Here BufferedInputStream can work on any InputStream, doesn’t have to be only FileInputStream.

**Note:** Complexity of creating objects.

In Pizza Ex: Instead of saying new PineAppleToppingVegetablePizza. You would say new PineAppleTopping and Pass the instance of Vegetable Pizza class to it.

Small drop back (Complexity of creating objects), But adds the functionality in dynamically.

1. **Façade:**

A single class that represents an entire subsystem.

**Ex**: Event Managers are good Façade.

**Explanation:** Let’s have one to organize an event. I have to take care of lot of things. I have to book a place, Decoration, Drinks, Food, Invitations, and Music Troop etc.

Instead of doing all this myself, I can go through Even Manager.

The Event Manager would acts as Façade. He would take care of decoration, Food, Drinks etc. All that I need to do is, tell Event Manager to do A, B, C, D and E. He takes care of arranging that.

I just need to call the Façade and it takes care of all these steps.

**Technical Ex:** I want to buy a book online. There are different steps. Check whether there is stock available, if available then reserve the book, Make the payment, Update the stock and generate Invoice.

All these different steps are needed.

When I look at UI perspective all this is on single click (OnlineBookOrder).When I click on UI, it goes to the web layer and the web layer invokes a method on the business layer. Business Layer where the Façade would be. Business Layer offers façade, which would do all these steps.

**Advantages:**

* Reduced network calls: The Business Layer is deployed in different Application server, then if there is no façade then from UI we have to make multiple calls.
* Reduced coupling: Web Layer needs to know only about Façade, no need to know about the subsystems. So, we can change the subsystems without worrying how it affects Web Layer.
* Helps in establishing the transaction boundary: Any of the steps fails, the previous steps rollback.

1. **Adaptor:**

Match interfaces of different classes.

**Ex: Power Adaptors:** I am from India, I bought a mobile from India and get the charger which works only in India. What should I do, if I go to USA? What I do is, I will buy an adaptor, which can take the input as plug points for the power adaptor in India has and on other side, it can connect to power plug in USA. That is Adaptor Pattern.

I have 2 different classes, which need to interact, but using 2 different Interfaces. So, what I do is, I will create an Adaptor class between them and help talk to each other.

1. **Flyweight:**

A fine-grained instance used for efficient sharing.

**Ex:** In certain times, not needed to create separate instances of objects. If there are 1000 instances of objects, that’s needs to be created. Then that is creating lots of objects. What we can do is, instead if creating a new instance of an object each time, we would set the right values into the object and then do the functionality that is needed. And then again set the values and do the functionality without creating a new object. We would try and reuse the objects as much as possible. We would create fixed instance of objects, we would try and use them directly.

**Singleton Pattern:**

4 ways to create singleton pattern:

* Eager initialization or initialize before use
* Lazy initialization or initialize as and when we needed
* Using inner class
* Singleton with serialization and deserialization