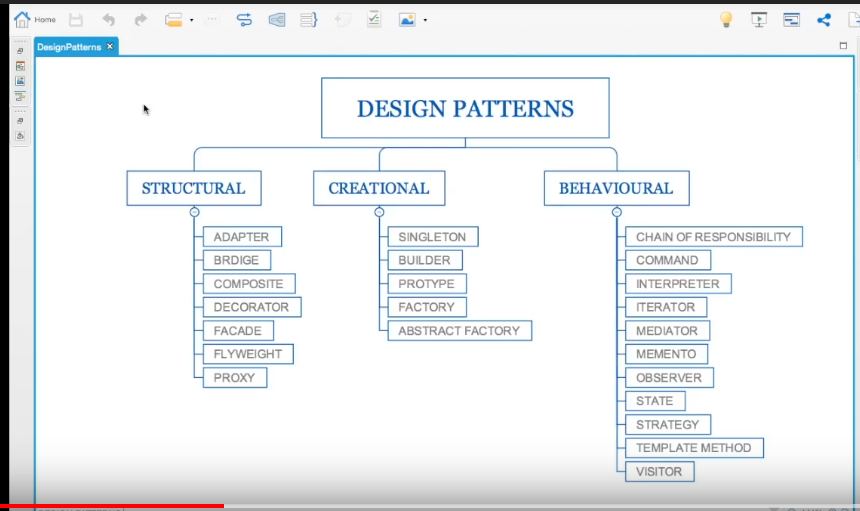
**Design Pattern** :



**1.Singleton D.P :**

# is Static in nature

# is Thread safe

# having private Constructor

# having private instance of class

# having No parameter to the constructor

Ex: **public class** Singleton {  
 **private static** Singleton *singleton* = **null**;  
 **private** Singleton(){  
  
 }  
 **public static** Singleton getInstance(){  
 **if**(*singleton*== **null**){  
 **synchronized** (Singleton.**class**) {  
 **if** (*singleton*==**null**) {  
 *singleton* = **new** Singleton();  
 }  
 }  
 }  
 **return** *singleton*;  
 }  
}

**public class** SingletonExample {  
  
 **public static void** main(String[] args) {  
 Singleton firstInstance = Singleton.*getInstance*();  
 System.***out***.println(firstInstance.hashCode());  
 Singleton secondInstance = Singleton.*getInstance*();  
 System.***out***.println(secondInstance.hashCode());  
 }  
}  
*/\*General Example of Singleton  
1. RunTime.getRuntime();  
2.Spring Beans*

*3. Loggers  
\**

O/P : 1163157884

1163157884

Disadvantages of Singleton D.P :

Overuse of Singleton Slows down our application

Note : **Calendar.getInstance()** is not Singleton desingn pattern rather it’s a Factory D.P because every time it returns a different calendar time so don’t confuse here.

**2.Factory D.P :**

**# It doesn’t expose the creation or instantiation logic**

**# Instead subclass creates the object**

**# There is a common abstract class or interface which needs to be done in the factory**

**# Generally Factory D.P is used to create a framework and the client /the implementation uses this factory**

**Builder D.P:**

**# It solves the multiple constructor problem (Telescoping constructors)**

**# It has Static inner class (i.e. Builder class)**

**# Internally calls the required constructor**

**# Removes the need for setter method rather using builder**

**# Provides immutable**

It is an **alternative way to construct complex objects**. This should be used only when you want to build different immutable objects using same object building process.

Let’s discuss a common problem in our application. In any user management module, primary entity is User, let’s say. let’s assume, our User object has following 5 attributes i.e. firstName, lastName, age, phone and address.

In normal practice, if you want to make a immutable User class, then you must pass all five information as parameters to constructor. It will look like this:

**public** User (String firstName, String lastName, **int** age, String phone, String address){  
 **this**.firstName = firstName;  
 **this**.lastName = lastName;  
 **this**.age = age;  
 **this**.phone = phone;  
 **this**.address = address;  
}

Very good. Now what if only firstName and lastName are **mandatory** and rest 3 fields are optional. Problem !! We need more constructors.

**public** User (String firstName, String lastName, **int** age, String phone){ ... }  
**public** User (String firstName, String lastName, String phone, String address){ ... }  
**public** User (String firstName, String lastName, **int** age){ ... }  
**public** User (String firstName, String lastName){ ... }

We will need some more like above. Still can manage? Now let’s introduce our sixth attribute i.e. salary. Now it is problem.

One way it to create more constructors, and another is to loose the immutability and introduce setter methods. You choose any of both options, you loose something, right?

Here, builder pattern will help you to consume additional attributes while retaining the immutability of Use class.

**A sample implementation using Builder Pattern**

Below is the coded solution of problem we discussed above. This uses a additional class **UserBuilder** which helps us in building desired User object with all mandatory attributes and combination of optional attributes, without loosing the immutability.

**public class** User  
{  
 *//All final attributes* **private final** String **firstName**; *// required* **private final** String **lastName**; *// required* **private final int age**; *// optional* **private final** String **phone**; *// optional* **private final** String **address**; *// optional* **private** User(UserBuilder builder) {  
 **this**.**firstName** = builder.**firstName**;  
 **this**.**lastName** = builder.**lastName**;  
 **this**.**age** = builder.**age**;  
 **this**.**phone** = builder.**phone**;  
 **this**.**address** = builder.**address**;  
 }  
  
 *//All getter, and NO setter to provde immutability* **public** String getFirstName() {  
 **return firstName**;  
 }  
 **public** String getLastName() {  
 **return lastName**;  
 }  
 **public int** getAge() {  
 **return age**;  
 }  
 **public** String getPhone() {  
 **return phone**;  
 }  
 **public** String getAddress() {  
 **return address**;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "User: "**+**this**.**firstName**+**", "**+**this**.**lastName**+**", "**+**this**.**age**+**", "**+**this**.**phone**+**", "**+**this**.**address**;  
 }  
  
 **public static class** UserBuilder  
 {  
 **private final** String **firstName**;  
 **private final** String **lastName**;  
 **private int age**;  
 **private** String **phone**;  
 **private** String **address**;  
  
 **public** UserBuilder(String firstName, String lastName) {  
 **this**.**firstName** = firstName;  
 **this**.**lastName** = lastName;  
 }  
 **public** UserBuilder age(**int** age) {  
 **this**.**age** = age;  
 **return this**;  
 }  
 **public** UserBuilder phone(String phone) {  
 **this**.**phone** = phone;  
 **return this**;  
 }  
 **public** UserBuilder address(String address) {  
 **this**.**address** = address;  
 **return this**;  
 }  
 *//Return the finally consrcuted User object* **public** User build() {  
 User user = **new** User(**this**);  
 **return** user;  
 }  
 }  
}

And below is the way, we will use the UserBuilder in our code:

**Main.java**

**public class** Main {  
 **public static void** main(String args[]){  
  
 User user1 = **new** User.UserBuilder(**"Lokesh"**, **"Gupta"**)  
 .age(30)  
 .phone(**"1234567"**)  
 .address(**"Fake address 1234"**)  
 .build();  
  
 System.***out***.println(user1);  
  
 User user2 = **new** User.UserBuilder(**"Jack"**, **"Reacher"**)  
 .age(40)  
 .phone(**"5655"**)  
 *//no address* .build();  
  
 System.***out***.println(user2);  
  
 User user3 = **new** User.UserBuilder(**"Super"**, **"Man"**)  
 *//No age  
 //No phone  
 //no address* .build();  
  
 System.***out***.println(user3);  
 }  
}

Output:  
  
 User: Lokesh, Gupta, 30, 1234567, Fake address 1234  
 User: Jack, Reacher, 40, 5655, **null** User: Super, Man, 0, **null**, **null**

E.g : StringBuilder,StringBuffer,DocumentBuilder,Locale.builder,Spring reactive etc

Advantages : 1. I can build **immutable objects** without much complex logic

2. Builder pattern also helps minimizing the number of parameters in constructor and thus there is **no need to pass in null for optional parameters** to the constructor.

**3. number of lines of code increase** at least to double in builder pattern, but the effort pays off in terms of **design flexibility** and much more **readable code**.

Disavantages :  **number of lines of code increase**