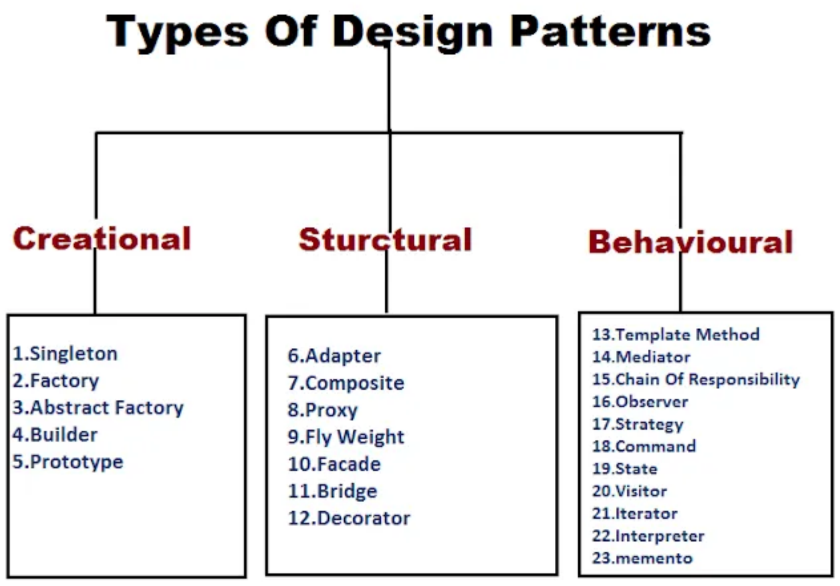
Design Patterns

Lets keep the word design patterns aside for a while and talk about a real life scenario. Suppose you are a car driver 🚕, and you are instructed to go from point A to B. Now more than 5 roads lead from A to B, hence you need to analyse now which of the 5 paths will be the best in terms of distance, which is a task 😐. But wait, you have google maps 📍 😀. Hence now you have a exact path which is already tested by other people.

Similarly, in the software development industry we have concepts that provide reusable solutions to common problem like e-commerce website, inventory management system, etc. These are tried and tested plus they promote best practices in coding and known as Design Patterns.

Code Link: https://github.com/iabhayysharma/Design-Patterns/tree/main/Creational



Note: Design patterns will be called DPs in the document.

Creational Design Patterns

These design patterns focus on creation of object

1. Singleton DP

This DP ensures that a class has only 1 instance and a global access point is provided.Example database connection details, details of ministries but with a single government class as government can not be more than once, etc. This single instance i.e. single object can be accessed directly without instantiation of the class

Steps:-
- Come into the class which requires singleton behaviour and make a private static variable with the new() keyword.
- Create a private constructor (so that this can not instantiated by other objects)
- Make a getter and **no** setter 🚫.

Two types of Singleton Creation

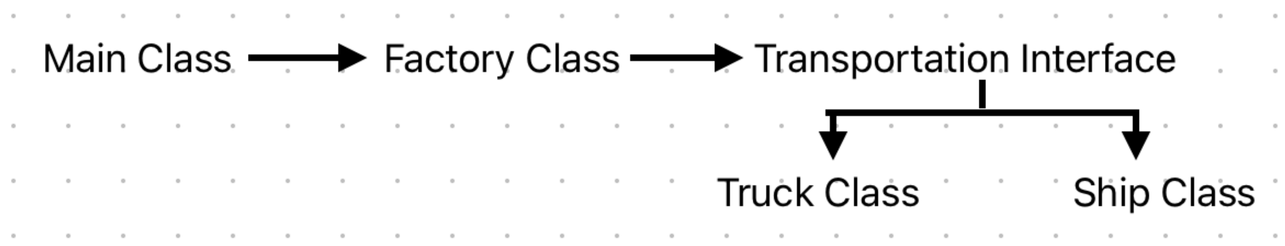
A. Early Loading: Instance creation done at load time. This is a thread-safe method to implement singleton patter.
B. Lazy Loading: Instance creation done when required. This is not thread-safe and required keyword synchronised.

Note: please refer to the code for implementation with comments.

Before jumping to the next DP, please know that a single ton pattern can be broken due to Cloning, Serialisation, Java reflection. These can be handled though via appropriate exception handling but the best way is to use **Enum** classes for implementing singleton DP.

2. Factory DP

This DP focus on object creation but with instantiation login hidden from the client(Main class). Example a transportation agency does delivery via trucks 🚚, but now they have requirement to add Ships 🚢 for fulfilling international orders. Hence now rather than making 2 objects directly from the client, we just make Transportation Factory object with the appropriate parameter and then accordingly make object of Truck or Ship Class.



Steps:-

- Main Class calls factory class
- Factory Class returns required instance

Note: please refer to the code for implementation with comments.

3. Abstract Factory DP

This DP is also known as Factories of factories. This DP is used when the Products have variations. Example in the previous DPs Transportation Factory, what if the company has Two type of shipping I.e. Fast 💨 and Standard 🐢? Hence here now firstly new Interface will be made i.e. ShippingSpeed.java along the concrete methods i.e. Fast.java and Standard.java, an finally an Abstract Factory Class be made with Abstract methods of type Transportation and ShippingSpeed.

Steps:-

- Main class calls the Factory of Factory
- According to 'variation' parameter, the Factory of Factory will create instance of factory class required
- The final factory class returns the required class instance

Note: please refer to the code for implementation with comments.

4. Builder DP

This DP focuses on a Step by Step process for building complex requirements. Real life example is a House 🏠 where we first build floor, then build walls and finally the terrace. Here we can make multiple Builder also according to the terrain

Steps:-
- Create POJO for the home
- Make Abstract Builder (Or interface)
- Create multiple builder like EarthquakeBuilder/FloodBuilder
- Finally make a director class to manage the construction which will be calling the appropriate builder. This director is called from the Main class

Note: please refer to the code for implementation with comments.

5. Prototype DP

This DP is based on concept of CLONING objects instead of creating new objects which are costly. Please note that java allows Copying of objects also but sometime variables are private hence exception comes

Two type of cloning
1.Shallow copying: primitive variables are copied while nested objects are shared by reference
2.Deep copying: Both primitive variables and nested objects are copied

Steps:-
- Make Interface/Abstract class and declare method for cloning
- Implement in a concrete class
- Override clone() and return the copy of the object according to deep copy or shallow copy

Structural Design Patterns

These DPs focus on ways to assemble different pieces together in a flexible and extensible fashion, so that when one of the part changes, entire structure doesn't change. In this document, we will focus on one of the most used Structural Dp i.e. Facade DP.

1. Facade DP

This DP provides a simplified interface to a complex system of classes/libraries and frameworks. This hides the complexity of the underlying system and hence provides a single interface to interact, hence decoupling client code from implementation details.

Example: We want to send rs 100 via HDFC Bank 🏦. In the main class we will call HDFCFacade and then call the deposit method. Now when this deposit is called via the facade, we can do other important tasks like Account Validation, Money sending, Notification service etc

Behavioural Design Patterns

These DPs focus on interaction between different objects of a system. We will focus on most used Strategy pattern in this document.

1. Strategy DP

This DP enables selecting algorithm at runtime hence replacing large conditional statements

Example: Payment 🧾 can be done via domestic and international methods. System should decide on runtime the preferred algorithm/function/class.

Steps:
- Make a Paymentprocessor class, which has a map in the constructor
- Make concrete strategy classes from a single interface
- Make Paymentprocessor object so that the constructor will populate the mop<String typeOfPayment, New {typeOfPayment}Class()>
- Call the concrete class from main with key, so that code can go to the correct value i.e. Correct class Domestic/International