Design Patterns

Design Pattern

It is a well defined solution to a common problem.

- It is industry standard approach.
- Captures the experience of experts
- Language independent
- Template, but not a solution

Design patterns are like recipes – generic solutions to expected situations

Why do we need Design Patterns?

- To solve common coding problems
- Here is some common problems:
    1. How to properly create a Class
    2. How to instantiate an Object
    3. How to interact between Objects
    4. How to write loosely coupled code
    5. How to write reusable code
- Using design patterns we can try to solve problem in better ways.

Benefits of Design Pattern

- Robust Code
- Code Reusability
- Highly Maintainability
- Loosely coupled application

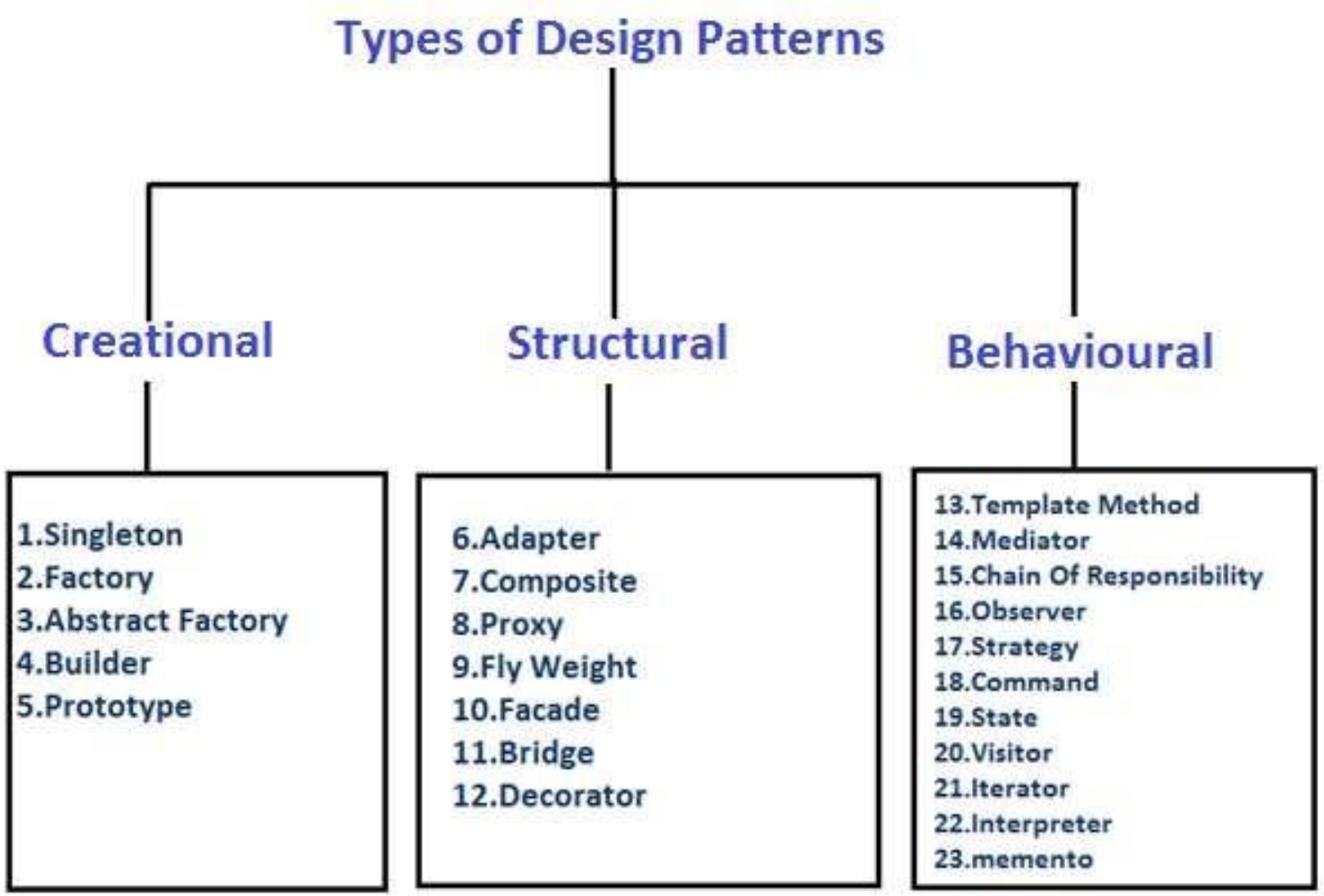
Types of Design Patterns

a) **Structural:** Deals with the structure of a class
   i. Adapter design pattern
   ii. Façade design pattern
   iii. Bridge design pattern etc.

b) **Creational:** Deals with the best way to create/instansiate objects
   i. Singletone design pattern
   ii. Factory design pattern
   iii. Bridge design pattern etc.

c) **Behavioral:** Ways for class/objects to communicate & interact with another
   i. Strategy design pattern
   ii. Observer design pattern
   iii. Iterator design pattern etc.

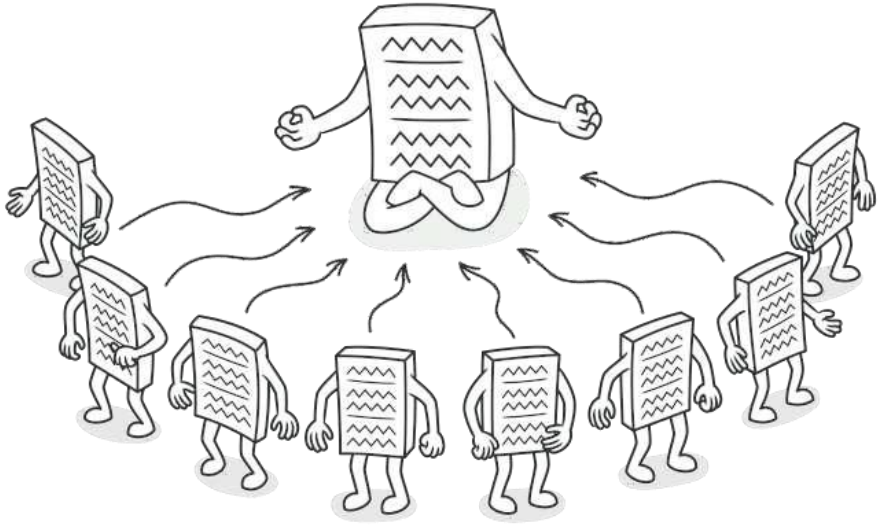
Types of Design Patterns

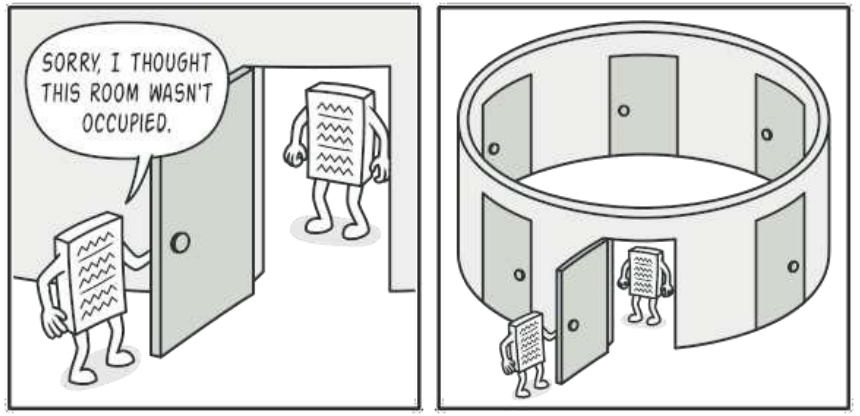


Singleton Pattern

- Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

- Where to use & benefits
  - Use the Singleton pattern when a class in your program should have just a single instance available to all clients;
  - for example, a single database object shared by different parts of the program.





How Singleton Pattern can be implemented?

- A private static variable of the same class which is the only reference of the class.
- A private constructor to instantiate the object of the class.
- A public static method to get the instance of the class from anywhere in the application.

Singleton Pattern

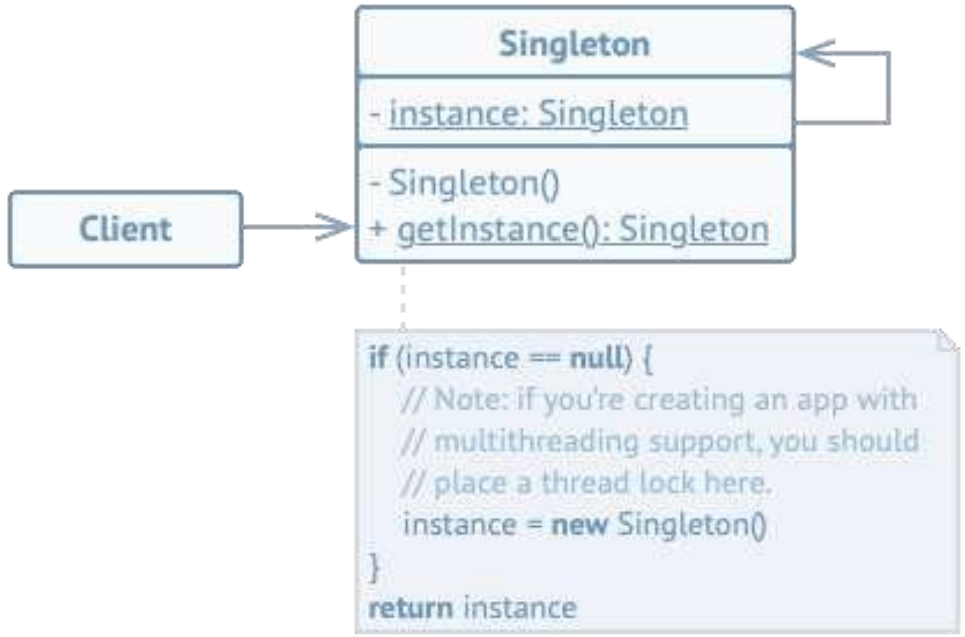
```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```




```
//Get the only object available
SingleObject object = SingleObject.getInstance();
```

Factory Pattern

- **Definition**
  - Provides an abstraction for deciding which class should be instantiated based on parameters given

- **Where to use & benefits**
  - A class cannot anticipate which subclasses must be created
  - Separate a family of objects using shared interface
  - Hide concrete classes from the client

How Factory Pattern can be implemented?

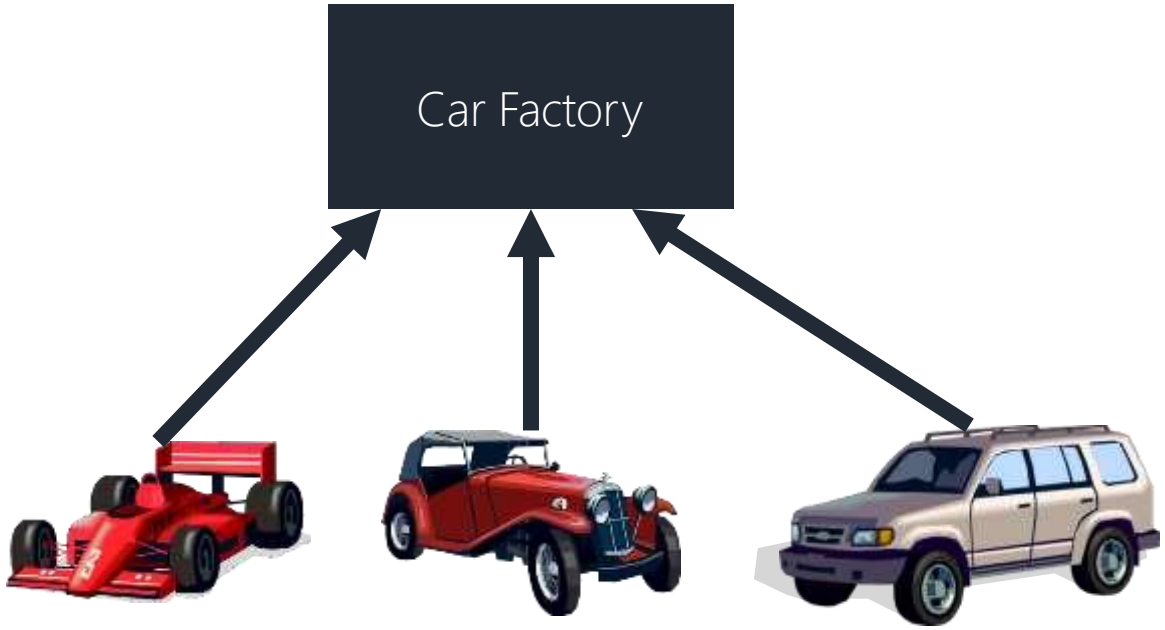
- Create an interface or abstract class that defines the method common to all classes.
- Implement the interface (or extend the abstract class) with specific classes for each type of object.
- Create the Factory Class that includes a method to decide which object to instantiate based on input.
- Then, call the factory method from client code.

Factory Pattern

- **Example**

  - Car Factory produces different Car objects

  - Original
    - Different classes implement Car interface
    - Directly instantiate car objects
    - Need to modify client to change cars
  - Using pattern
    - Use car factory class to produce car objects
    - Can change cars by changing car factory



Factory Pattern example

```
// step-1
interface Car {
	public void drive();
}
// step-2
class Ferrari implements Car {
	public void drive() {
		System.out.println("Driving a fast Ferrari.");
	}
}
class Bentley implements Car {
	public void drive() {
		System.out.println("Driving an antique Bentley.");
	}
}
class Explorer  implements Car {
	public void drive() {
		System.out.println("Driving a versatile Explorer.");
	}
}
```

Factory Pattern example

```
// step-3
public class carFactory {

        public static Car create(String type) {
                if (type.equals("fast"))
                        return new Ferrari();
                else if (type.equals(“antique"))
                        return new Bentley();
                else
                       return new Explorer();
        }
}
// step-4
Car fast = carFactory.create("fast");       // returns fast car
```