# Design pattern

In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages. Some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

1. Creational,
2. Structural
3. Behavioral patterns

**Creational patterns:**

1. Abstract factory
2. Factory method
3. Builder
4. Prototype
5. Singleton
6. Object Pool

**Structural:**

1. adapter,
2. Bridge
3. Composite
4. Decorator
5. Facade
6. Flyweight
7. Proxy
8. Private Class Data

**Behavioral patterns**

1. Blackboard
2. Chain of responsibility
3. Command
4. Iterator
5. Interpreter
6. Mediator
7. Memento
8. Observer or Publish/subscribe
9. Strategy
10. State
11. Template method
12. Visitor
13. Null Object

# Design pattern in Java

**Java core design pattern:** there are many design pattern. But in core java we learn something important that we need more.

1. **Creational patterns:**
2. Singleton
3. Abstract factory
4. Builder
5. Factory method
6. Prototype
7. **Structural:**
8. adapter,
9. Bridge Pattern
10. Composite
11. Decorator
12. Facade
13. Flyweight
14. Proxy
15. **Behavioral patterns**
16. Chain of Responsibility Pattern
17. Command Pattern
18. Interpreter Pattern
19. Iterator Pattern
20. Mediator Pattern
21. Memento Pattern
22. Observer Pattern
23. State Pattern
24. Strategy Pattern
25. Template Pattern
26. Visitor Pattern

**Advance java design pattern:**

1. DOA
2. DTO
3. MVC
4. ORM
5. AOP & IOC
6. DI

# Singleton design pattern in Java:

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. Example- for DB connection we create one object of database connectivity then use this object when we need it.

Create singleton class:

1. Create a private constructor of the class to restrict object creation outside of the class.
2. Create a private attribute of the class type that refers to the single object.
3. Create a public static method that allows us to create and access the object we created. Inside the method, we will create a condition that restricts us from creating more than one object.

Code:

class Singleton {

private static Singleton obj;

private Singleton () {};

public static Singleton getInstance(){

         if (obj==null) {

             obj = new Singleton ();

}

         return obj;

     }

}

# Facade Design Pattern

Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

Now Let’s try and understand the facade pattern better using a simple example. Let’s consider a hotel. This hotel has a hotel keeper. There are a lot of restaurants inside hotel e.g. Veg restaurants, Non-Veg restaurants and Veg/Non Both restaurants. You, as client want access to different menus of different restaurants. You do not know what are the different menus they have. You just have access to hotel keeper who knows his hotel well. Whichever menu you want, you tell the hotel keeper and he takes it out of from the respective restaurants and hands it over to you. Here, the hotel keeper acts as the facade, as he hides the complexities of the system hotel.

**Code:**

1. **Interface of Hotel:**

package structural.facade;

public interface Hotel {

     public Menus getMenus();

}

1. **NonVegRestaurant.java:**

package structural.facade;

 public class NonVegRestaurant implements Hotel {

     public Menus getMenus() {

        NonVegMenu nv = new NonVegMenu();

        return nv;

    }

}

1. **VegRestaurant.java**

package structural.facade;

 public class VegRestaurant implements Hotel {

     public Menus getMenus()  {

        VegMenu v = new VegMenu();

        return v;

    }

}

1. **HotelKeeper.java**

package structural.facade;

 public interface HotelKeeper {

     public VegMenu getVegMenu();

   public NonVegMenu getNonVegMenu();

   public Both getVegNonMenu();

}

1. **HotelKeeperImplementation.java**

package structural.facade;

public class HotelKeeperImplementation implements HotelKeeper {

public VegMenu getVegMenu(){

        VegRestaurant v = new VegRestaurant();

        VegMenu vegMenu = (VegMenu)v.getMenus();

        return vegMenu;

 }

public NonVegMenu getNonVegMenu()    {

        NonVegRestaurant v = new NonVegRestaurant();

        NonVegMenu NonvegMenu = (NonVegMenu)v.getMenus();

        return NonvegMenu;

}

}

1. **Client use façade class:**

package structural.facade;

public class Client{

    public static void main (String[] args) {

        HotelKeeper keeper = new HotelKeeper();

        VegMenu v = keeper.getVegMenu();

        NonVegMenu nv = keeper.getNonVegMenu();

    }

}

# Dependency injection

**Dependency:**

Before know dependency injection we need to know what is dependency. Dependency or dependent means relying on something for support. As example, if we want to go somewhere we depend on car.

In programming we say When class A uses some functionality of class B, then it’s said that class A has a dependency of class B. if we want to use other class method, we need to create the object of that class. Without create object we cannot use method of other class. Here class A create the object of class B and then use class B method.

Code:

Class engine{

Function show(){

Count<<”this is goo”;

}

}

Class Car {

Car yahamaEngine = new Engine();

YahamaEngine.show();

}

In above, Car use show method by create object of Engine class. So, we called Car is dependence of engine class.

Here is some problem of class dependency:

1. Class is not testable
2. Code is not extensible
3. Single responsibility
4. Lifetime of Object:

To solve this type of problem we use dependency injection.

**Dependency injection:**

In dependency injection dependence class object are create by someone else and dependable class use this object. because dependencies can be injected at runtime rather than at compile time

The 3 Types of Dependency Injection---

1. constructor injection,
2. method injection,
3. property injection.

**Constructor Injection:**

Constructor injection is the process of using the constructor to pass in the dependencies of a class. You should use constructor injection when your class has a dependency that the class requires in order to work properly.

If your class cannot work without a dependency, then inject it via the constructor.

you should use constructor injection when the dependency in question has a lifetime longer than a single method. Dependencies passed into the constructor should be useful to the class in a general way, with its use spanning multiple methods in the class. If a dependency is used in only one spot, method injection

Checking for null is necessary and is boilerplate code. Protecting against null being passed as a parameter is called the guard pattern

public class CustomerBusinessLogic{

ICustomerDataAccess \_dataAccess;

public CustomerBusinessLogic(ICustomerDataAccess custDataAccess){

\_dataAccess = custDataAccess;

}

public CustomerBusinessLogic(){

\_dataAccess = new CustomerDataAccess();

}

public string ProcessCustomerData(int id){

return \_dataAccess.GetCustomerName(id);

}

}

public interface ICustomerDataAccess{

string GetCustomerName(int id);

}

public class CustomerDataAccess: ICustomerDataAccess{

public CustomerDataAccess(){

}

public string GetCustomerName(int id){

return "Dummy Customer Name";

}

}

**Property Injection (setter injection):**

You should use property injection in case the dependency is truly optional

Property Injection however causes Temporal Coupling and when writing Line of Business applications, your dependencies should never be optional: you should instead apply the Null Object pattern.

property injection is considered bad in 98% of all scenarios because it hides dependencies and there is no guarantee that the object will be injected when the class is created.

The built-in IoC container does not support property injection. You will have to use a third-party IoC container.

Code:

public class CustomerBusinessLogic{

public CustomerBusinessLogic(){

}

public string GetCustomerName(int id){

return DataAccess.GetCustomerName(id);

}

public ICustomerDataAccess DataAccess { get; set; }

}

public class CustomerService{

CustomerBusinessLogic \_customerBL;

public CustomerService(){

\_customerBL = new CustomerBusinessLogic();

\_customerBL.DataAccess = new CustomerDataAccess();

}

public string GetCustomerName(int id) {

return \_customerBL.GetCustomerName(id);

}

}

**Method Injection:**

Thus, method injection is useful in two scenarios: when the implementation of dependency will vary, and when the dependency needs to be renewed after each use. In both cases, it’s up to the caller to decide what implementation to pass to the method.

interface IDataAccessDependency{

void SetDependency(ICustomerDataAccess customerDataAccess);

}

public class CustomerBusinessLogic : IDataAccessDependency{

ICustomerDataAccess \_dataAccess;

public CustomerBusinessLogic() { }

public string GetCustomerName(int id){

return \_dataAccess.GetCustomerName(id);

}

public void SetDependency(ICustomerDataAccess customerDataAccess){

\_dataAccess = customerDataAccess;

}

}

public class CustomerService{

CustomerBusinessLogic \_customerBL;

public CustomerService(){

\_customerBL = new CustomerBusinessLogic();

((IDataAccessDependency)\_customerBL).SetDependency(new CustomerDataAccess());

}

public string GetCustomerName(int id) {

return \_customerBL.GetCustomerName(id);

}

}

# Inversion of Control (IoC)