**Design Patterns**-

Design patterns are solutions to the common problems in OOP development. They are reusable solutions.

Design patterns are mainly categorized into three types-

1. Creational Design Patterns
2. Behavioral Design Patterns
3. Structural Design Patterns

**Creational Design Patterns**-

These patterns provide solutions to problem associated with object creation.

**Singleton Design Pattern-**

When we need to create only one object of a class, we need to use Singleton design pattern.

Intent of singleton design pattern is to create only one object of a class and to provide global access point for that object.

Usage:

1. When we want to share a common resource across whole application .
2. Centralized logging
3. Printer pool which is common then we create a single object of printer pool which is shared.
4. Factories are implemented as singletons.
5. Properties object which contains information about remote connection or username is implemented by using singleton and that object is kept into cache after initialization.

Singleton class-

class Singleton {

private static Singleton instance;

private Singleton() {

...

}

public static synchronized Singleton getInstance(){

if (instance == null)

instance = new Singleton();

return instance; }

}

Above class is naïve implementation of singleton class. By using lazy instantiation we can reduce the time complexity . as synchronized keyword is costly in terms of time we simply put object creation code into synchronized block and remove synchronized keyword from method signature. Time can be reduced by using early instantiation.

Drawback-

1. If singleton class loaded by two class loaders then there are two classes of singleton class in application memory then there will be two instances.

**Factory Design Pattern-**

Factory design patterns hides the instantiation logic from the client and return the required object.

Lets see example of a car –

interface Car{

String company;

String model;

int manufacturingYear;

}

class Lamborginhi implements Car {

……………

}

Class Ferrari implements Car {

………….

}

In general if we want to create a object of Ferrari we will simply do

Car ferrari1 = new Ferrari();

But every time using new keyword is not good because it exposes instantiation logic to client and down the line during maintenance process we need to change a lot of code it violates DRY and open closed principle.

That’s the situation where factory design pattern comes into picture.

class CarFactory{

public Car getCar(String carType){

if(carType == “Ferrari”){\

return new Ferrari();

} else if (carType == “Lamborginhi”){

return new Lamborginhi();

}

}

}

Now we can get the specific car instance by using getCar() method of CarFactory class.

If new car comes in future then we can only edit CarFactory class and this will not violet single responsibility and open closed principle as factory class has only responsibility of instantiation of car objects.

**Abstract Factory –**

Abstract factory provides interface to combine two similar type of factories. Lets consider a example of pizza factory.

Suppose we have two pizza factories one is Indian and second one is Russian lets name them as

IndianPizzaFactory and RussianPizzaFactory both have same algorithm for delivery of pizza but they differ in how different versions of pizza are created. Ultimately both the factories are pizza factories and they are kind of similar so we can group both by using a abstract factory named AbstractPizzaFactory.

abstract class AbstractPizzaFactory {

public void deliver(){

……………..

}

public void getPizza();

}

class IndianPizzaFactory implements AbstractPizzaFactory{

public void getPizza(){

if(type == “cheese”){

return new IndianCheesePizza();

}

}

}

class RussianPizzaFactory implements AbstractPizzaFactory{

public void getPizza(String type){

if(type == “cheese”){

return new RussianCheesePizza();

}

}

}

There could be multiple versions of pizza factories according to countries that’s why abstract factory design pattern is helpful for us to group all the pizza factories.

**Builder Pattern-**

Builder design pattern helps in creation of complex objects. Sometimes we need complex object creation and complex objects contain many dependencies to be included. If we do all the stuff in single class it will violate the single responsibility principle. What we can do here is delegate the object creation task to builder object so that builder object can encapsulate create and add dependencies and will provide a object to us.

Lets consider example of creating and sending a email. Here creating email object is complex task as we need to add recipient , add formatted body text, greeting etcetra. What we can do here is to create a

class Email{

private String title;

…………………….

Private Email(){

}

public void send(){

…………….

}

public static class EmailBuilder{

//contains procedure to build an email

public Email build(){

return new Email(params);

}

}

}

Here we can create email using Email email = new Email.EmailBuilder().setXXX;

Here we completely make Email and EmailBuilder separate so that both of them should follow single responsibility principle.

**Behavioral Patterns-**

Behavioral patterns provides solution to problems associated with object communication.

**Chain of responsibility-**

Intention behind chain of responsibility is to avoid coupling between sender of a request and request handler.

In software development of a system most of the times we need to handle events associated with a object . to solve this problem we can make a chain of handlers, when an event needs to be handled it flows through the recursive list of handlers. A handler can either handle the request or pass it on to next handler.

Usasge:

Chain of responsibility is useful during designing of support ticket system.

lets see the example

public class Request{

}

public abstract class Handler{

protected Handler successorObject;

public void setSuccessor(Handler successor){

successorObject = successor;

}

public abstract void handleRequest(Request request);

}

public class ConcreteHandler1 implements Handler {

public void handleRequest(Request request){

// it will either handle the request or pass it on to next concrete handler by calling

// super.handleRequest();

}

}

public class ConcreteHandler2 implements Handler {

// it will either handle the request or pass it on to next concrete handler by calling

// super.handleRequest();

}

At first we need to make the chain . chain should not be broken . broken chain might fail to serve some requests.

**Command design pattern**

Command design encapsulate requests into object. A command can be made up of several smaller commands. In command design pattern smaller commands will be encapsulated into macro one.

Let’s consider stock example-

class StockMarket{

public void buy(){

}

public void sell(){

}

}

Interface command {

Public void execute();

}

class BuyCommand implements command {

public void execute(){

stock.buy();

}

}

class SellCommand implements command {

public void execute(){

stock.sell();

}

}

Here we encapsulate the command under SellCommand and BuyCommand so that the agent which is the invoker of these command don’t know about the actual command.It decouples the object that perform action from the requester.

**Strategy Design pattern**

Strategy design pattern help us choose algorithm dynamically. That is algorithm selection is deferred to runtime.

This pattern comes into picture when we have classes that are different in behavior only.

Lets take example of Robot class-

interface IBehavior {

public void move();

}

class AggresiveMove implements IBehavior{

public void move(){

…………..

}

}

class DefensiveMove implements IBehavior {

public void move(){

………..

}

}

class Robot{

IBehavior behavior;

}

In above example we can dynamically choose the algorithm for move() ;

**Structural Patterns**

Structural design patterns ease the design of a software and helps in providing relation between entities.

**Adapter Design Pattern-**

When we have a vendor class having interface different than client interface then in order to make client class workable with vendor class we need to have one adapter class which plays the role of intermediate.

Lets consider a general example-

We have one client ,vendor and adapter class

Class client {

Public void connectToServer(){

VendorAdapter v=new VendorAdapter();

v.connectToServer();

}

}

Class VendorAdapter{

Public void connectToServer(){

VendorAdapter v=new Vendor();

v.connect();

}

}

Class vendor implements VendorAdapter{

Public void connect(){

…………..

}

}

In above example connection between vendor and client has been done using adapter class.

**Decorator Design Pattern-**

Decorator pattern attaches additional responsibility to a object dynamically. Decorator provides alternative to doing subclasses to extend functionality. Decorator pattern encapsulates a object inside another one.

Let’s take example of Weapon class.

interface Weapon {

public void aim\_and\_fire();

public void cost();

public void getDescription();

}

Class Baseweapon implements Weapon{

Public void getDescription(){

………..

}

Public float cost(){

…………..

}

Public void aim\_and\_fire(){

……….

}

}

Abstract class WeaponAccessary implements Weapon{

Public void aim\_and\_fire(){

……………

}

}

Class AccessaryOne extends WeaponAccessary{

Weapon weapon;

Float cost;

Public AccessaryOne(Weapon weapon){

this.weapon=weapon;

}

Public float cost(){

Return weapon.getCost()+this.cost;

}

Public void getDescription(){

}

}

In above example we can easily add any number of baseWeapons by simply implementing Weapon interface.

If someone wants to purchase weapon with baseWeapon and accessories

It would be like

Weapon weapon=new BaseWeapon();

weapon=new AccessaryOne(weapon);

in this way at final we will get whole amount of base + accessary amount.

Decorator design pattern can be effectively used in shopping when we purchase multiple items as complimentary to something like if we are purchasing laptops then headphones , mouse ,external keyboard are kind of related thing which come under Laptop accessories. Which will help us prevent creating unnecessary subclasses for each accessary under each laptop like DellLaptop ,HpLaptop etc.