**DEFINITION**

Design Patterns are general reusable solution to commonly occurring problems. These are the best practices, used by the experienced developers. Patterns are not complete code, but it can use as a template which can be applied to a problem. Patterns Java Design Patterns 2 / 173 are re-usable; they can be applied to similar kind of design problem regardless to any domain.

**CATEGORIES**

**Creational Patterns->**

* Abstract Factory
* Builder
* Factory Method
* Prototype
* Singleton

**Structural Patterns->**

* Adapter
* Bridge
* Composite
* Decorator
* FaÃ§ade
* Flyweight
* Proxy

**Behavioral Patterns**

* Chain of Responsibility
* Command
* Interpreter
* Iterator
* Mediator
* Memento
* Observer
* State
* Strategy
* Template Method
* Visitor

**ADAPTER PATTERN**

The adapter pattern convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.Adapters are used when we have a class (Client) expecting some type of object and we have an object (Adaptee) offering the same features but exposing a different interface.

To use an adapter:

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates that request on the adaptee using the adaptee interface.
3. Client receive the results of the call and is unaware of adapter’s presence.



**Java implementation**

interface Bird

{

// birds implement Bird interface that allows

// them to fly and make sounds adaptee interface

public void fly();

public void makeSound();

}

class Sparrow implements Bird

{

// a concrete implementation of bird

public void fly()

{

System.out.println("Flying");

}

public void makeSound()

{

System.out.println("Chirp Chirp");

}

}

interface ToyDuck

{

// target interface

// toyducks dont fly they just make

// squeaking sound

public void squeak();

}

class PlasticToyDuck implements ToyDuck

{

public void squeak()

{

System.out.println("Squeak");

}

}

class BirdAdapter implements ToyDuck

{

// You need to implement the interface your

// client expects to use.

Bird bird;

public BirdAdapter(Bird bird)

{

// we need reference to the object we

// are adapting

this.bird = bird;

}

public void squeak()

{

// translate the methods appropriately

bird.makeSound();

}

}

class Main

{

public static void main(String args[])

{

Sparrow sparrow = new Sparrow();

ToyDuck toyDuck = new PlasticToyDuck();

// Wrap a bird in a birdAdapter so that it

// behaves like toy duck

ToyDuck birdAdapter = new BirdAdapter(sparrow);

System.out.println("Sparrow...");

sparrow.fly();

sparrow.makeSound();

System.out.println("ToyDuck...");

toyDuck.squeak();

// toy duck behaving like a bird

System.out.println("BirdAdapter...");

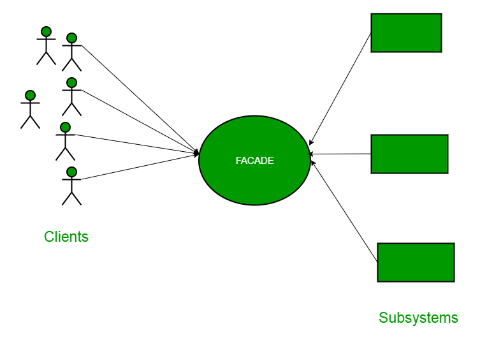
birdAdapter.squeak();

}

}

**Facade Design Pattern**

Facade is a part of the Gang of Four design patterns and it is categorized under Structural design patterns.  the interface JDBC can be called a facade because we as users or clients create connections using the “java.sql.Connection” interface, the implementation of which we are not concerned about. The implementation is left to the vendor of the driver.It hides the complexities of the system and provides an interface to the client from where the client can access the system.

[](https://media.geeksforgeeks.org/wp-content/uploads/facadeA.png)

**JAVA implemntation**

package structural.facade;

public interface Hotel {

public Menus getMenus();

}

package structural.facade;

public class NonVegRestaurant implements Hotel {

public Menus getMenus()

{

NonVegMenu nv = new NonVegMenu();

return nv;

}

}

package structural.facade;

public class VegRestaurant implements Hotel {

public Menus getMenus()

{

VegMenu v = new VegMenu();

return v;

}

}

package structural.facade;

public class VegNonBothRestaurant implements Hotel {

public Menus getMenus()

{

Both b = new Both();

return b;

}

}

/\*package whatever //do not write package name here \*/

package structural.facade;

public interface HotelKeeper {

public VegMenu getVegMenu();

public NonVegMenu getNonVegMenu();

public Both getVegNonMenu();

}

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;

}

public Both getVegNonMenu()

{

VegNonBothRestaurant v = new VegNonBothRestaurant();

Both bothMenu = (Both)v.getMenus();

return bothMenu;

}

}

package structural.facade;

public class Client

{

public static void main (String[] args)

{

HotelKeeper keeper = new HotelKeeperImplementation();

VegMenu v = keeper.getVegMenu();

NonVegMenu nv = keeper.getNonVegMenu();

Both = keeper.getVegNonMenu();

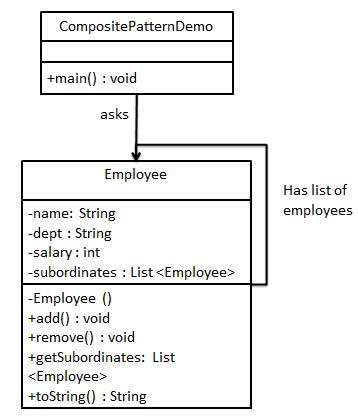
}

}

**Composite pattern**

The composite pattern is meant to allow treating individual objects and compositions of objects, or “composites” in the same way.

It can be viewed as a tree structure made up of types that inherit a base type, and it can represent a single part or a whole hierarchy of objects.



**Java Implementation**

import java.util.ArrayList;

import java.util.List;

public class Employee {

private String name;

private String dept;

private int salary;

private List<Employee> subordinates;

// constructor

public Employee(String name,String dept, int sal) {

this.name = name;

this.dept = dept;

this.salary = sal;

subordinates = new ArrayList<Employee>();

}

public void add(Employee e) {

subordinates.add(e);

}

public void remove(Employee e) {

subordinates.remove(e);

}

public List<Employee> getSubordinates(){

return subordinates;

}

public String toString(){

return ("Employee :[ Name : " + name + ", dept : " + dept + ", salary :" + salary+" ]");

}

}

public class CompositePatternDemo {

public static void main(String[] args) {

Employee CEO = new Employee("John","CEO", 30000);

Employee headSales = new Employee("Robert","Head Sales", 20000);

Employee headMarketing = new Employee("Michel","Head Marketing", 20000);

Employee clerk1 = new Employee("Laura","Marketing", 10000);

Employee clerk2 = new Employee("Bob","Marketing", 10000);

Employee salesExecutive1 = new Employee("Richard","Sales", 10000);

Employee salesExecutive2 = new Employee("Rob","Sales", 10000);

CEO.add(headSales);

CEO.add(headMarketing);

headSales.add(salesExecutive1);

headSales.add(salesExecutive2);

headMarketing.add(clerk1);

headMarketing.add(clerk2);

//print all employees of the organization

System.out.println(CEO);

for (Employee headEmployee : CEO.getSubordinates()) {

System.out.println(headEmployee);

for (Employee employee : headEmployee.getSubordinates()) {

System.out.println(employee);

}

}

}

}

**Bridge pattern**

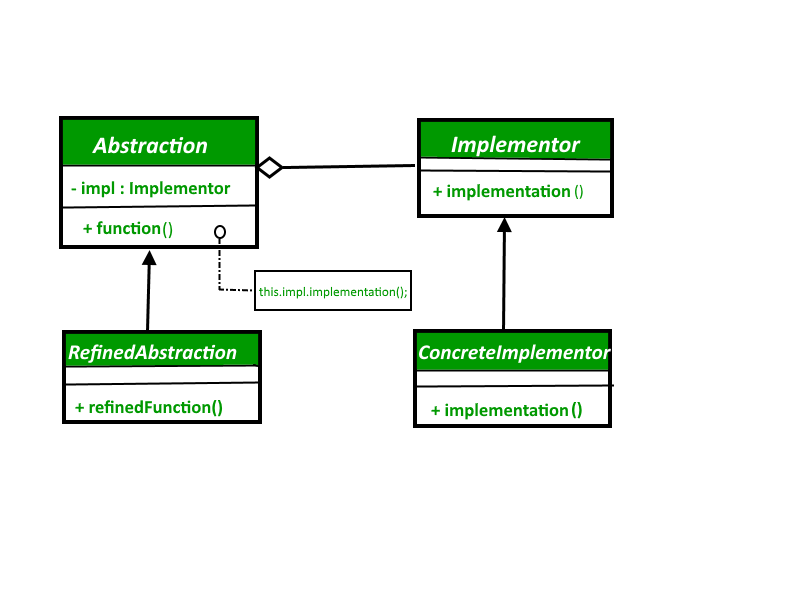
The Bridge design pattern allows you to separate the abstraction from the implementation. It is a structural design pattern.

**There are 2 parts in Bridge design pattern :**

1. Abstraction
2. Implementation

The bridge pattern allows the Abstraction and the Implementation to be developed independently and the client code can access only the Abstraction part without being concerned about the Implementation part.

The abstraction is an interface or abstract class and the implementer is also an interface or abstract class.



**Java Implementation**

// abstraction in bridge pattern

abstract class Vehicle {

protected Workshop workShop1;

protected Workshop workShop2;

protected Vehicle(Workshop workShop1, Workshop workShop2)

{

this.workShop1 = workShop1;

this.workShop2 = workShop2;

}

abstract public void manufacture();

}

// Refine abstraction 1 in bridge pattern

class Car extends Vehicle {

public Car(Workshop workShop1, Workshop workShop2)

{

super(workShop1, workShop2);

}

@Override

public void manufacture()

{

System.out.print("Car ");

workShop1.work();

workShop2.work();

}

}

// Refine abstraction 2 in bridge pattern

class Bike extends Vehicle {

public Bike(Workshop workShop1, Workshop workShop2)

{

super(workShop1, workShop2);

}

@Override

public void manufacture()

{

System.out.print("Bike ");

workShop1.work();

workShop2.work();

}

}

// Implementer for bridge pattern

interface Workshop

{

abstract public void work();

}

// Concrete implementation 1 for bridge pattern

class Produce implements Workshop {

@Override

public void work()

{

System.out.print("Produced");

}

}

// Concrete implementation 2 for bridge pattern

class Assemble implements Workshop {

@Override

public void work()

{

System.out.print(" And");

System.out.println(" Assembled.");

}

}

// Demonstration of bridge design pattern

class BridgePattern {

public static void main(String[] args)

{

Vehicle vehicle1 = new Car(new Produce(), new Assemble());

vehicle1.manufacture();

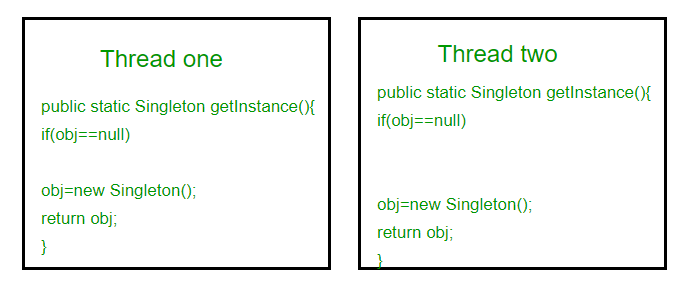
Vehicle vehicle2 = new Bike(new Produce(), new Assemble());

vehicle2.manufacture();

}

}

**Singleton Design Pattern**   
The singleton pattern is one of the simplest design patterns. Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.   
*The singleton pattern is a design pattern that restricts the instantiation of a class to one object.*   
Let’s see various design options for implementing such a class. If you have a good handle on static class variables and access modifiers this should not be a difficult task.



**Java Implementation**

// Classical Java implementation of singleton

// design pattern

class Singleton

{

private static Singleton obj;

// private constructor to force use of

// getInstance() to create Singleton object

private Singleton() {}

public static Singleton getInstance()

{

if (obj==null)

obj = new Singleton();

return obj;

}

}

**Observer design pattern**

Observer is a behavioral design pattern. It specifies communication between objects: observable and observers. **An**observable**is an object which notifies**observers**about the changes in its state.**

## Benefits:

* It describes the coupling between the objects and the observer.
* It provides the support for broadcast-type communication.

## Usage:

* When the change of a state in one object must be reflected in another object without keeping the objects tight coupled.
* When the framework we writes and needs to be enhanced in future with new observers with minimal chamges.

Classical Java implementation

// design pattern

class Singleton

{

private static Singleton obj;

// private constructor to force use of

// getInstance() to create Singleton object

private Singleton() {}

public static Singleton getInstance()

{

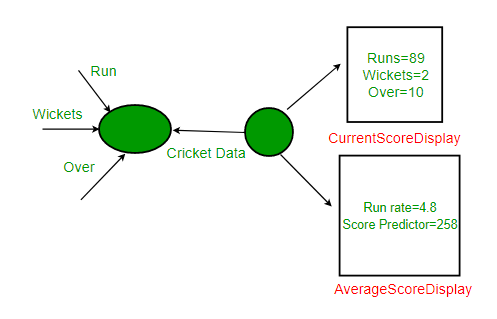
if (obj==null)

obj = new Singleton();

return obj;

}

}



**Java implementation**

// Java implementation of above design for Cricket App. The

// problems with this design are discussed below.

// A class that gets information from stadium and notifies

// display elements, CurrentScoreDisplay & AverageScoreDisplay

class CricketData

{

int runs, wickets;

float overs;

CurrentScoreDisplay currentScoreDisplay;

AverageScoreDisplay averageScoreDisplay;

// Constructor

public CricketData(CurrentScoreDisplay currentScoreDisplay,

AverageScoreDisplay averageScoreDisplay)

{

this.currentScoreDisplay = currentScoreDisplay;

this.averageScoreDisplay = averageScoreDisplay;

}

// Get latest runs from stadium

private int getLatestRuns()

{

// return 90 for simplicity

return 90;

}

// Get latest wickets from stadium

private int getLatestWickets()

{

// return 2 for simplicity

return 2;

}

// Get latest overs from stadium

private float getLatestOvers()

{

// return 10.2 for simplicity

return (float)10.2;

}

// This method is used update displays when data changes

public void dataChanged()

{

// get latest data

runs = getLatestRuns();

wickets = getLatestWickets();

overs = getLatestOvers();

currentScoreDisplay.update(runs,wickets,overs);

averageScoreDisplay.update(runs,wickets,overs);

}

}

// A class to display average score. Data of this class is

// updated by CricketData

class AverageScoreDisplay

{

private float runRate;

private int predictedScore;

public void update(int runs, int wickets, float overs)

{

this.runRate = (float)runs/overs;

this.predictedScore = (int) (this.runRate \* 50);

display();

}

public void display()

{

System.out.println("\nAverage Score Display:\n" +

"Run Rate: " + runRate +

"\nPredictedScore: " + predictedScore);

}

}

// A class to display score. Data of this class is

// updated by CricketData

class CurrentScoreDisplay

{

private int runs, wickets;

private float overs;

public void update(int runs,int wickets,float overs)

{

this.runs = runs;

this.wickets = wickets;

this.overs = overs;

display();

}

public void display()

{

System.out.println("\nCurrent Score Display: \n" +"Runs: " + runs +"\nWickets:"+ wickets + "\nOvers: " + overs );

}

}

// Driver class

class Main

{

public static void main(String args[])

{

// Create objects for testing

AverageScoreDisplay averageScoreDisplay= new AverageScoreDisplay();

CurrentScoreDisplay currentScoreDisplay =new CurrentScoreDisplay();

// Pass the displays to Cricket data

CricketData cricketData = new CricketData(currentScoreDisplay,averageScoreDisplay);

// In real app you would have some logic to call this

// function when data changes

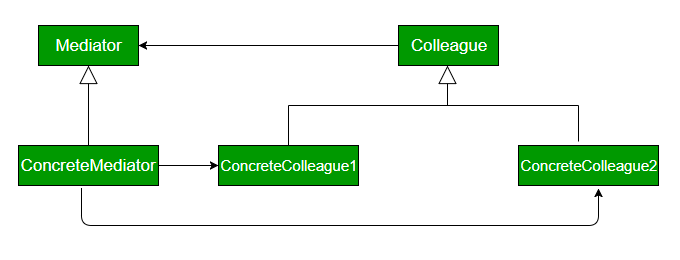
cricketData.dataChanged();

}

}

**Meditor Design patern**

Mediator design pattern is one of the important and widely used behavioral design pattern. Mediator enables decoupling of objects by introducing a layer in between so that the interaction between objects happen via the layer. If the objects interact with each other directly, the system components are tightly-coupled with each other that makes higher maintainability cost and not hard to extend. Mediator pattern focuses on providing a mediator between objects for communication and help in implementing loose-coupling between objects.



**Java Implementation**

class ATCMediator implements IATCMediator

{

private Flight flight;

private Runway runway;

public boolean land;

public void registerRunway(Runway runway)

{

this.runway = runway;

}

public void registerFlight(Flight flight)

{

this.flight = flight;

}

public boolean isLandingOk()

{

return land;

}

@Override

public void setLandingStatus(boolean status)

{

land = status;

}

}

interface Command

{

void land();

}

interface IATCMediator

{

public void registerRunway(Runway runway);

public void registerFlight(Flight flight);

public boolean isLandingOk();

public void setLandingStatus(boolean status);

}

class Flight implements Command

{

private IATCMediator atcMediator;

public Flight(IATCMediator atcMediator)

{

this.atcMediator = atcMediator;

}

public void land()

{

if (atcMediator.isLandingOk())

{

System.out.println("Successfully Landed.");

atcMediator.setLandingStatus(true);

}

else

System.out.println("Waiting for landing.");

}

public void getReady()

{

System.out.println("Ready for landing.");

}

}

class Runway implements Command

{

private IATCMediator atcMediator;

public Runway(IATCMediator atcMediator)

{

this.atcMediator = atcMediator;

atcMediator.setLandingStatus(true);

}

@Override

public void land()

{

System.out.println("Landing permission granted.");

atcMediator.setLandingStatus(true);

}

}

class MediatorDesignPattern

{

public static void main(String args[])

{

IATCMediator atcMediator = new ATCMediator();

Flight sparrow101 = new Flight(atcMediator);

Runway mainRunway = new Runway(atcMediator);

atcMediator.registerFlight(sparrow101);

atcMediator.registerRunway(mainRunway);

sparrow101.getReady();

mainRunway.land();

sparrow101.land();

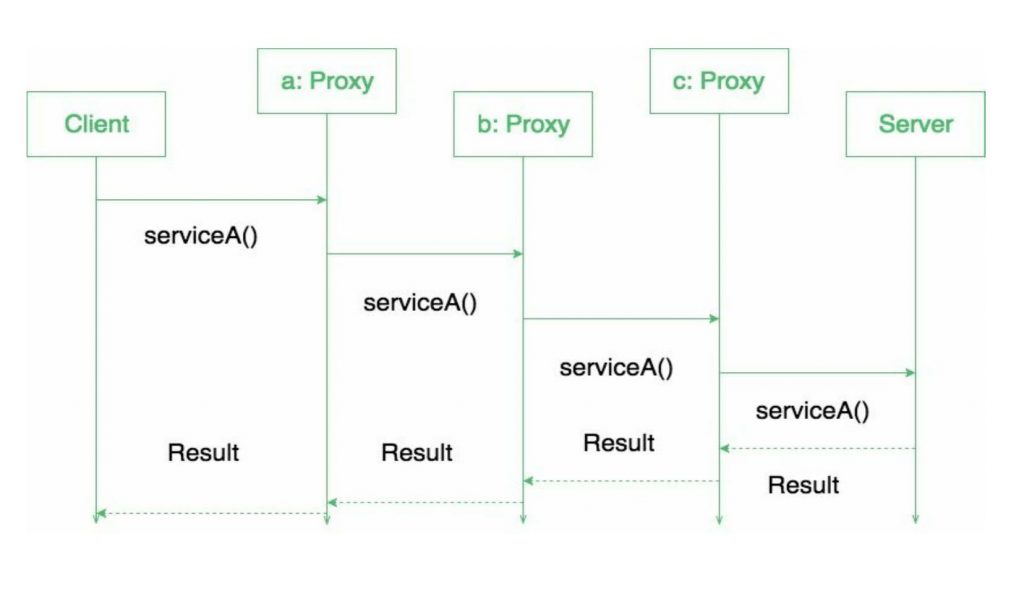
}

}

**Proxy Design Pattern**

Proxy means ‘in place of’, representing’ or ‘in place of’ or ‘on behalf of’ are literal meanings of proxy and that directly explains **Proxy Design Pattern**.  
Proxies are also called surrogates, handles, and wrappers. They are closely related in structure, but not purpose, to [Adapters](https://www.geeksforgeeks.org/adapter-pattern/) and [Decorators](https://www.geeksforgeeks.org/the-decorator-pattern-set-2-introduction-and-design/).

A real world example can be a cheque or credit card is a proxy for what is in our bank account. It can be used in place of cash, and provides a means of accessing that cash when required. And that’s exactly what the Proxy pattern does – “**Controls and manage access to the object they are protecting**“.



**JAVA Implementation**

package com.saket.demo.proxy;

public interface Internet

{

public void connectTo(String serverhost) throws Exception;

}

package com.saket.demo.proxy;

public class RealInternet implements Internet

{

@Override

public void connectTo(String serverhost)

{

System.out.println("Connecting to "+ serverhost);

}

}

package com.saket.demo.proxy;

import java.util.ArrayList;

import java.util.List;

public class ProxyInternet implements Internet

{

private Internet internet = new RealInternet();

private static List<String> bannedSites;

static

{

bannedSites = new ArrayList<String>();

bannedSites.add("abc.com");

bannedSites.add("def.com");

bannedSites.add("ijk.com");

bannedSites.add("lnm.com");

}

@Override

public void connectTo(String serverhost) throws Exception

{

if(bannedSites.contains(serverhost.toLowerCase()))

{

throw new Exception("Access Denied");

}

internet.connectTo(serverhost);

}

}

package com.saket.demo.proxy;

public class Client

{

public static void main (String[] args)

{

Internet internet = new ProxyInternet();

try

{

internet.connectTo("geeksforgeeks.org");

internet.connectTo("abc.com");

}

catch (Exception e)

{

System.out.println(e.getMessage());

}

}

}

**Chain of Responsibility Design Pattern**

Chain of responsibility pattern is used to achieve loose coupling in software design where a request from the client is passed to a chain of objects to process them. Later, the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

**Where and When Chain of Responsibility pattern is applicable :**

* When you want to decouple a request’s sender and receiver
* Multiple objects, determined at runtime, are candidates to handle a request
* When you don’t want to specify handlers explicitly in your code
* When you want to issue a request to one of several objects without specifying the receiver explicitly.

https://media.geeksforgeeks.org/wp-content/uploads/desigmpatternuml1.png

**Java Implementation**

public class Chain

{

Processor chain;

public Chain(){

buildChain();

}

private void buildChain(){

chain = new NegativeProcessor(new ZeroProcessor(new PositiveProcessor(null)));

}

public void process(Number request) {

chain.process(request);

}

}

abstract class Processor

{

private Processor nextProcessor;

public Processor(Processor nextProcessor){

this.nextProcessor = nextProcessor;

};

public void process(Number request){

if(nextProcessor != null)

nextProcessor.process(request);

};

}

class Number

{

private int number;

public Number(int number)

{

this.number = number;

}

public int getNumber()

{

return number;

}

}

class NegativeProcessor extends Processor

{

public NegativeProcessor(Processor nextProcessor){

super(nextProcessor);

}

public void process(Number request)

{

if (request.getNumber() < 0)

{

System.out.println("NegativeProcessor : " + request.getNumber());

}

else

{

super.process(request);

}

}

}

class ZeroProcessor extends Processor

{

public ZeroProcessor(Processor nextProcessor){

super(nextProcessor);

}

public void process(Number request)

{

if (request.getNumber() == 0)

{

System.out.println("ZeroProcessor : " + request.getNumber());

}

else

{

super.process(request);

}

}

}

class PositiveProcessor extends Processor

{

public PositiveProcessor(Processor nextProcessor){

super(nextProcessor);

}

public void process(Number request)

{

if (request.getNumber() > 0)

{

System.out.println("PositiveProcessor : " + request.getNumber());

}

else

{

super.process(request);

}

}

}

class TestChain

{

public static void main(String[] args) {

Chain chain = new Chain();

//Calling chain of responsibility

chain.process(new Number(90));

chain.process(new Number(-50));

chain.process(new Number(0));

chain.process(new Number(91));

}

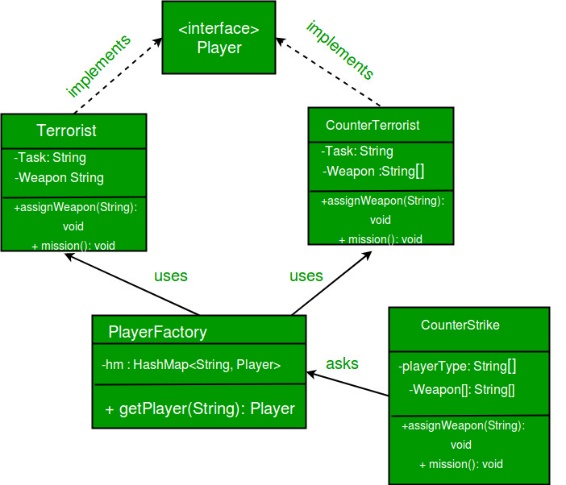
}

**Flyweight Design Pattern**

Flyweight pattern is one of the [structural design patterns](https://www.geeksforgeeks.org/design-patterns-set-1-introduction/) as this pattern provides ways to decrease object count thus improving application required objects structure. Flyweight pattern is used when we need to create a large number of similar objects (say 105). One important feature of flyweight objects is that they are **immutable**. This means that they cannot be modified once they have been constructed.

**Why do we care for number of objects in our program?**

* Less number of objects reduces the memory usage, and it manages to keep us away from errors related to memory like [java.lang.OutOfMemoryError.](https://docs.oracle.com/javase/7/docs/api/java/lang/OutOfMemoryError.html)
* Although creating an object in Java is really fast, we can still reduce the execution time of our program by sharing objects.



**Java implementation**

// A Java program to demonstrate working of

// FlyWeight Pattern with example of Counter

// Strike Game

import java.util.Random;

import java.util.HashMap;

// A common interface for all players

interface Player

{

public void assignWeapon(String weapon);

public void mission();

}

// Terrorist must have weapon and mission

class Terrorist implements Player

{

// Intrinsic Attribute

private final String TASK;

// Extrinsic Attribute

private String weapon;

public Terrorist()

{

TASK = "PLANT A BOMB";

}

public void assignWeapon(String weapon)

{

// Assign a weapon

this.weapon = weapon;

}

public void mission()

{

//Work on the Mission

System.out.println("Terrorist with weapon "

+ weapon + "|" + " Task is " + TASK);

}

}

// CounterTerrorist must have weapon and mission

class CounterTerrorist implements Player

{

// Intrinsic Attribute

private final String TASK;

// Extrinsic Attribute

private String weapon;

public CounterTerrorist()

{

TASK = "DIFFUSE BOMB";

}

public void assignWeapon(String weapon)

{

this.weapon = weapon;

}

public void mission()

{

System.out.println("Counter Terrorist with weapon "

+ weapon + "|" + " Task is " + TASK);

}

}

class PlayerFactory

{

/\* HashMap stores the reference to the object

of Terrorist(TS) or CounterTerrorist(CT). \*/

private static HashMap <String, Player> hm =

new HashMap<String, Player>();

// Method to get a player

public static Player getPlayer(String type)

{

Player p = null;

/\* If an object for TS or CT has already been

created simply return its reference \*/

if (hm.containsKey(type))

p = hm.get(type);

else

{

/\* create an object of TS/CT \*/

switch(type)

{

case "Terrorist":

System.out.println("Terrorist Created");

p = new Terrorist();

break;

case "CounterTerrorist":

System.out.println("Counter Terrorist Created");

p = new CounterTerrorist();

break;

default :

System.out.println("Unreachable code!");

}

// Once created insert it into the HashMap

hm.put(type, p);

}

return p;

}

}

// Driver class

public class CounterStrike

{

// All player types and weapon (used by getRandPlayerType()

// and getRandWeapon()

private static String[] playerType =

{"Terrorist", "CounterTerrorist"};

private static String[] weapons =

{"AK-47", "Maverick", "Gut Knife", "Desert Eagle"};

// Driver code

public static void main(String args[])

{

/\* Assume that we have a total of 10 players

in the game. \*/

for (int i = 0; i < 10; i++)

{

/\* getPlayer() is called simply using the class

name since the method is a static one \*/

Player p = PlayerFactory.getPlayer(getRandPlayerType());

/\* Assign a weapon chosen randomly uniformly

from the weapon array \*/

p.assignWeapon(getRandWeapon());

// Send this player on a mission

p.mission();

}

}

// Utility methods to get a random player type and

// weapon

public static String getRandPlayerType()

{

Random r = new Random();

// Will return an integer between [0,2)

int randInt = r.nextInt(playerType.length);

// return the player stored at index 'randInt'

return playerType[randInt];

}

public static String getRandWeapon()

{

Random r = new Random();

// Will return an integer between [0,5)

int randInt = r.nextInt(weapons.length);

// Return the weapon stored at index 'randInt'

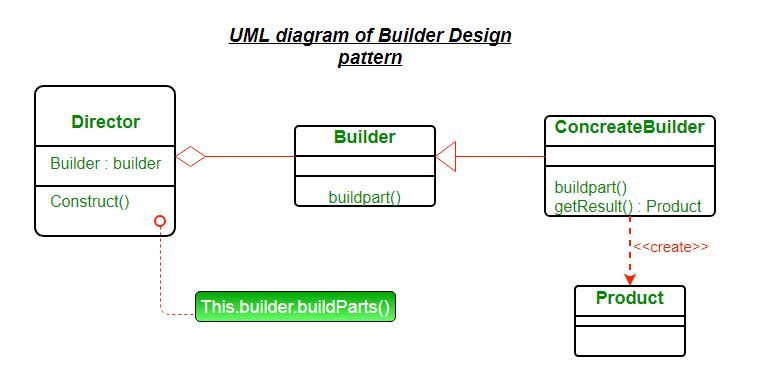
return weapons[randInt];

}

}

**Builder Design Pattern**

Builder pattern aims to “Separate the construction of a complex object from its representation so that the same construction process can create different representations.” It is used to construct a complex object step by step and the final step will return the object. The process of constructing an object should be generic so that it can be used to create different representations of the same object.



**Java Implemention**

interface HousePlan

{

public void setBasement(String basement);

public void setStructure(String structure);

public void setRoof(String roof);

public void setInterior(String interior);

}

class House implements HousePlan

{

private String basement;

private String structure;

private String roof;

private String interior;

public void setBasement(String basement)

{

this.basement = basement;

}

public void setStructure(String structure)

{

this.structure = structure;

}

public void setRoof(String roof)

{

this.roof = roof;

}

public void setInterior(String interior)

{

this.interior = interior;

}

}

interface HouseBuilder

{

public void buildBasement();

public void buildStructure();

public void buildRoof();

public void buildInterior();

public House getHouse();

}

class IglooHouseBuilder implements HouseBuilder

{

private House house;

public IglooHouseBuilder()

{

this.house = new House();

}

public void buildBasement()

{

house.setBasement("Ice Bars");

}

public void buildStructure()

{

house.setStructure("Ice Blocks");

}

public void buildInterior()

{

house.setInterior("Ice Carvings");

}

public void buildRoof()

{

house.setRoof("Ice Dome");

}

public House getHouse()

{

return this.house;

}

}

class TipiHouseBuilder implements HouseBuilder

{

private House house;

public TipiHouseBuilder()

{

this.house = new House();

}

public void buildBasement()

{

house.setBasement("Wooden Poles");

}

public void buildStructure()

{

house.setStructure("Wood and Ice");

}

public void buildInterior()

{

house.setInterior("Fire Wood");

}

public void buildRoof()

{

house.setRoof("Wood, caribou and seal skins");

}

public House getHouse()

{

return this.house;

}

}

class CivilEngineer

{

private HouseBuilder houseBuilder;

public CivilEngineer(HouseBuilder houseBuilder)

{

this.houseBuilder = houseBuilder;

}

public House getHouse()

{

return this.houseBuilder.getHouse();

}

public void constructHouse()

{

this.houseBuilder.buildBasement();

this.houseBuilder.buildStructure();

this.houseBuilder.buildRoof();

this.houseBuilder.buildInterior();

}

}

class Builder

{

public static void main(String[] args)

{

HouseBuilder iglooBuilder = new IglooHouseBuilder();

CivilEngineer engineer = new CivilEngineer(iglooBuilder);

engineer.constructHouse();

House house = engineer.getHouse();

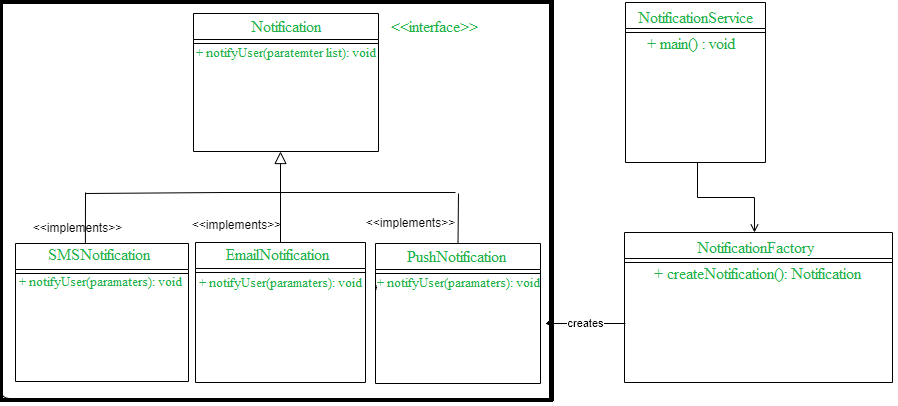
System.out.println("Builder constructed: "+ house);

}

}

**Factory Method Design Pattern**

It is a creational design pattern that talks about the creation of an object. The factory design pattern says that define an interface ( A java interface or an abstract class) for creating object and let the subclasses decide which class to instantiate. The factory method in the interface lets a class defers the instantiation to one or more concrete subclasses. Since these design patterns talk about the instantiation of an object and so it comes under the category of creational design pattern. If we notice the name **Factory method**, that means there is a method which is a factory, and in general, factories are involved with creational stuff and here with this, an object is being created. It is one of the best ways to create an object where object creation logic is hidden from the client. Now Let’s look at the implementation.



**JAVA Implementation**

public interface Notification {

void notifyUser();

}

public class SMSNotification implements Notification {

@Override

public void notifyUser()

{

// TODO Auto-generated method stub

System.out.println("Sending an SMS notification");

}

}

public class EmailNotification implements Notification {

@Override

public void notifyUser()

{

// TODO Auto-generated method stub

System.out.println("Sending an e-mail notification");

}

}

public class PushNotification implements Notification {

@Override

public void notifyUser()

{

// TODO Auto-generated method stub

System.out.println("Sending a push notification");

}

}

public class NotificationFactory {

public Notification createNotification(String channel)

{

if (channel == null || channel.isEmpty())

return null;

switch (channel) {

case "SMS":

return new SMSNotification();

case "EMAIL":

return new EmailNotification();

case "PUSH":

return new PushNotification();

default:

throw new IllegalArgumentException("Unknown channel "+channel);

}

}

}

public class NotificationService {

public static void main(String[] args)

{

NotificationFactory notificationFactory = new NotificationFactory();

Notification notification = notificationFactory.createNotification("SMS");

notification.notifyUser();

}

}

**Abstract Factory Method**

Abstract Factory design pattern is one of the Creational pattern. Abstract Factory pattern is almost similar to [Factory Pattern](https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/) and is considered as another layer of abstraction over factory pattern. Abstract Factory patterns work around a super-factory which creates other factories.  
Abstract factory pattern implementation provides us with a framework that allows us to create objects that follow a general pattern. So at runtime, the abstract factory is coupled with any desired concrete factory which can create objects of the desired type.

Lightbox

**Java Implementation**

// working of Abstract Factory Pattern

enum CarType

{

MICRO, MINI, LUXURY

}

abstract class Car

{

Car(CarType model, Location location)

{

this.model = model;

this.location = location;

}

abstract void construct();

CarType model = null;

Location location = null;

CarType getModel()

{

return model;

}

void setModel(CarType model)

{

this.model = model;

}

Location getLocation()

{

return location;

}

void setLocation(Location location)

{

this.location = location;

}

@Override

public String toString()

{

return "CarModel - "+model + " located in "+location;

}

}

class LuxuryCar extends Car

{

LuxuryCar(Location location)

{

super(CarType.LUXURY, location);

construct();

}

@Override

protected void construct()

{

System.out.println("Connecting to luxury car");

}

}

class MicroCar extends Car

{

MicroCar(Location location)

{

super(CarType.MICRO, location);

construct();

}

@Override

protected void construct()

{

System.out.println("Connecting to Micro Car ");

}

}

class MiniCar extends Car

{

MiniCar(Location location)

{

super(CarType.MINI,location );

construct();

}

@Override

void construct()

{

System.out.println("Connecting to Mini car");

}

}

enum Location

{

DEFAULT, USA, INDIA

}

class INDIACarFactory

{

static Car buildCar(CarType model)

{

Car car = null;

switch (model)

{

case MICRO:

car = new MicroCar(Location.INDIA);

break;

case MINI:

car = new MiniCar(Location.INDIA);

break;

case LUXURY:

car = new LuxuryCar(Location.INDIA);

break;

default:

break;

}

return car;

}

}

class DefaultCarFactory

{

public static Car buildCar(CarType model)

{

Car car = null;

switch (model)

{

case MICRO:

car = new MicroCar(Location.DEFAULT);

break;

case MINI:

car = new MiniCar(Location.DEFAULT);

break;

case LUXURY:

car = new LuxuryCar(Location.DEFAULT);

break;

default:

break;

}

return car;

}

}

class USACarFactory

{

public static Car buildCar(CarType model)

{

Car car = null;

switch (model)

{

case MICRO:

car = new MicroCar(Location.USA);

break;

case MINI:

car = new MiniCar(Location.USA);

break;

case LUXURY:

car = new LuxuryCar(Location.USA);

break;

default:

break;

}

return car;

}

}

class CarFactory

{

private CarFactory()

{

}

public static Car buildCar(CarType type)

{

Car car = null;

// We can add any GPS Function here which

// read location property somewhere from configuration

// and use location specific car factory

// Currently I'm just using INDIA as Location

Location location = Location.INDIA;

switch(location)

{

case USA:

car = USACarFactory.buildCar(type);

break;

case INDIA:

car = INDIACarFactory.buildCar(type);

break;

default:

car = DefaultCarFactory.buildCar(type);

}

return car;

}

}

class AbstractDesign

{

public static void main(String[] args)

{

System.out.println(CarFactory.buildCar(CarType.MICRO));

System.out.println(CarFactory.buildCar(CarType.MINI));

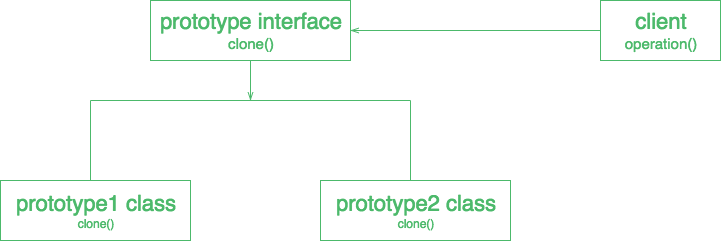
System.out.println(CarFactory.buildCar(CarType.LUXURY));

}

}

**Prototype Design Pattern-**

Prototype allows us to hide the complexity of making new instances from the client. The concept is to copy an existing object rather than creating a new instance from scratch, something that may include costly operations. The existing object acts as a prototype and contains the state of the object. The newly copied object may change same properties only if required. This approach saves costly resources and time, especially when object creation is a heavy process.



**JAVA Implementation**

import java.util.HashMap;

import java.util.Map;

abstract class Color implements Cloneable

{

protected String colorName;

abstract void addColor();

public Object clone()

{

Object clone = null;

try

{

clone = super.clone();

}

catch (CloneNotSupportedException e)

{

e.printStackTrace();

}

return clone;

}

}

class blueColor extends Color

{

public blueColor()

{

this.colorName = "blue";

}

@Override

void addColor()

{

System.out.println("Blue color added");

}

}

class blackColor extends Color{

public blackColor()

{

this.colorName = "black";

}

@Override

void addColor()

{

System.out.println("Black color added");

}

}

class ColorStore {

private static Map<String, Color> colorMap = new HashMap<String, Color>();

static

{

colorMap.put("blue", new blueColor());

colorMap.put("black", new blackColor());

}

public static Color getColor(String colorName)

{

return (Color) colorMap.get(colorName).clone();

}

}

// Driver class

class Prototype

{

public static void main (String[] args)

{

ColorStore.getColor("blue").addColor();

ColorStore.getColor("black").addColor();

ColorStore.getColor("black").addColor();

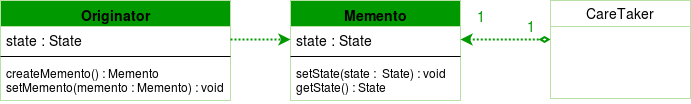
ColorStore.getColor("blue").addColor();

}

}

**Memento Design Pattern**

Memento pattern is a behavioral design pattern. Memento pattern is used to restore state of an object to a previous state. As your application is progressing, you may want to save checkpoints in your application and restore back to those checkpoints later. Intent of Memento Design pattern is without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later.



**Java Implementation**

import java.util.List;

import java.util.ArrayList;

class Life

{

private String time;

public void set(String time)

{

System.out.println("Setting time to " + time);

this.time = time;

}

public Memento saveToMemento()

{

System.out.println("Saving time to Memento");

return new Memento(time);

}

public void restoreFromMemento(Memento memento)

{

time = memento.getSavedTime();

System.out.println("Time restored from Memento: " + time);

}

public static class Memento

{

private final String time;

public Memento(String timeToSave)

{

time = timeToSave;

}

public String getSavedTime()

{

return time;

}

}

}

class Design

{

public static void main(String[] args)

{

List<Life.Memento> savedTimes = new ArrayList<Life.Memento>();

Life life = new Life();

//time travel and record the eras

life.set("1000 B.C.");

savedTimes.add(life.saveToMemento());

life.set("1000 A.D.");

savedTimes.add(life.saveToMemento());

life.set("2000 A.D.");

savedTimes.add(life.saveToMemento());

life.set("4000 A.D.");

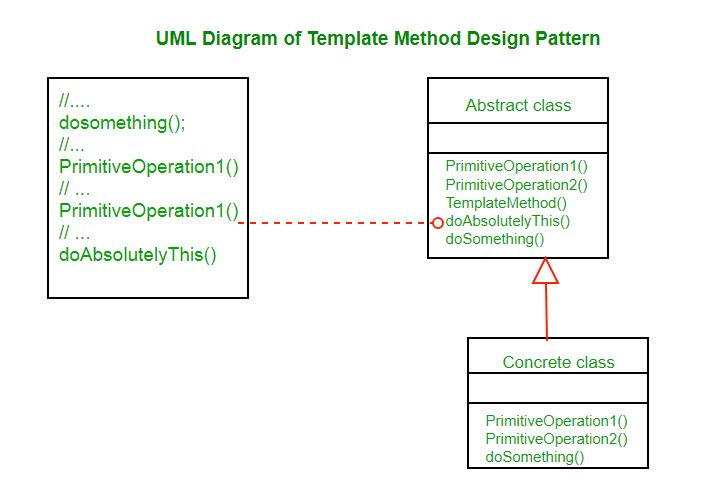
life.restoreFromMemento(savedTimes.get(0));

}

}

**Template Design Pattern**

Template method design pattern is to define an algorithm as a skeleton of operations and leave the details to be implemented by the child classes. The overall structure and sequence of the algorithm are preserved by the parent class.   
Template means Preset format like HTML templates which has a fixed preset format. Similarly in the template method pattern, we have a preset structure method called template method which consists of steps. These steps can be an abstract method that will be implemented by its subclasses.  
This behavioral design pattern is one of the easiest to understand and implement. This design pattern is used popularly in framework development.



**JAVA Implementation**

abstract class OrderProcessTemplate

{

public boolean isGift;

public abstract void doSelect();

public abstract void doPayment();

public final void giftWrap()

{

try

{

System.out.println("Gift wrap successful");

}

catch (Exception e)

{

System.out.println("Gift wrap unsuccessful");

}

}

public abstract void doDelivery();

public final void processOrder(boolean isGift)

{

doSelect();

doPayment();

if (isGift) {

giftWrap();

}

doDelivery();

}

}

class NetOrder extends OrderProcessTemplate

{

@Override

public void doSelect()

{

System.out.println("Item added to online shopping cart");

System.out.println("Get gift wrap preference");

System.out.println("Get delivery address.");

}

@Override

public void doPayment()

{

System.out.println

("Online Payment through Netbanking, card or Paytm");

}

@Override

public void doDelivery()

{

System.out.println

("Ship the item through post to delivery address");

}

}

class StoreOrder extends OrderProcessTemplate

{

@Override

public void doSelect()

{

System.out.println("Customer chooses the item from shelf.");

}

@Override

public void doPayment()

{

System.out.println("Pays at counter through cash/POS");

}

@Override

public void doDelivery()

{

System.out.println("Item delivered to in delivery counter.");

}

}

class TemplateMethodPatternClient

{

public static void main(String[] args)

{

OrderProcessTemplate netOrder = new NetOrder();

netOrder.processOrder(true);

System.out.println();

OrderProcessTemplate storeOrder = new StoreOrder();

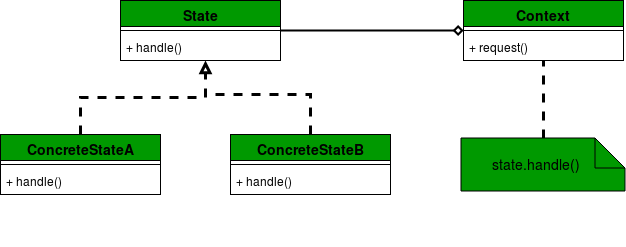
storeOrder.processOrder(true);

}

}

**State Design Pattern**

The state pattern is one of [the behavioral design patterns](https://www.geeksforgeeks.org/design-patterns-set-1-introduction/). A state design pattern is used when an Object changes its behavior based on its internal state. If we have to change the behavior of an object based on its state, we can have a state variable in the Object and use the if-else condition block to perform different actions based on the state. The state pattern is used to provide a systematic and lose-coupled way to achieve this through Context and State implementations.



**JAVA Implementation**

// Java program to demonstrate working of

// State Design Pattern

interface MobileAlertState {

public void alert(AlertStateContext ctx);

}

class AlertStateContext {

private MobileAlertState currentState;

public AlertStateContext()

{

currentState = new Vibration();

}

public void setState(MobileAlertState state)

{

currentState = state;

}

public void alert() { currentState.alert(this); }

}

class Vibration implements MobileAlertState {

@Override public void alert(AlertStateContext ctx)

{

System.out.println(" vibration... & quot;);

}

}

class Silent implements MobileAlertState {

@Override public void alert(AlertStateContext ctx)

{

System.out.println(" silent... & quot;);

}

}

class StatePattern {

public static void main(String[] args)

{

AlertStateContext stateContext

= new AlertStateContext();

stateContext.alert();

stateContext.alert();

stateContext.setState(new Silent());

stateContext.alert();

stateContext.alert();

stateContext.alert();

}

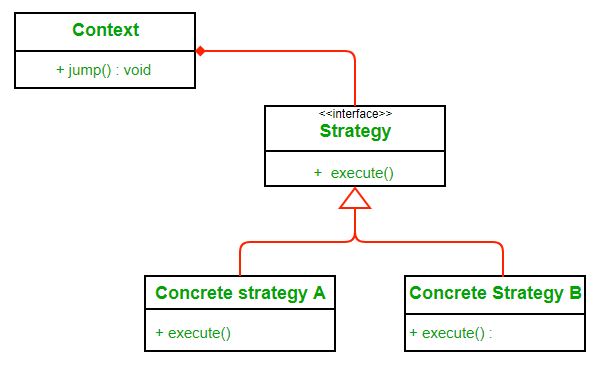
}

**Strategy pattern**

**Strategy pattern** is a behavioral design pattern that allows the behavior of an object to be selected at runtime. It is one of the Gang of Four (GoF) design patterns, which are widely used in object-oriented programming.

The Strategy pattern is based on the idea of encapsulating a family of algorithms into separate classes that implement a common interface. The pattern consists of three main components: the Context, the Strategy, and the Concrete Strategy.

The Context is the class that contains the object whose behavior needs to be changed dynamically. The Strategy is the interface or abstract class that defines the common methods for all the algorithms that can be used by the Context object. The Concrete Strategy is the class that implements the Strategy interface and provides the actual implementation of the algorithm.



**JAVA Implementation**

public interface Discounter {

BigDecimal applyDiscount(BigDecimal amount);

}

public static class EasterDiscounter implements Discounter {

@Override

public BigDecimal applyDiscount(final BigDecimal amount) {

return amount.multiply(BigDecimal.valueOf(0.5));

}

}

public static class ChristmasDiscounter implements Discounter {

@Override

public BigDecimal applyDiscount(final BigDecimal amount) {

return amount.multiply(BigDecimal.valueOf(0.9));

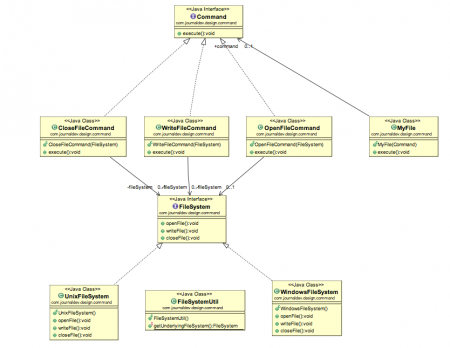
}

}

**Command Pattern**

The **command pattern**encapsulates a request as an object, thereby letting us parameterize other objects with different requests, queue or log requests, and support undoable operations.

The definition is a bit confusing at first but let’s step through it. In analogy to our problem above remote control is the client and stereo, lights etc. are the receivers. In command pattern there is a Command object that *encapsulates a request*by binding together a set of actions on a specific receiver. It does so by exposing just one method execute() that causes some actions to be invoked on the receiver.



**JAVA Implementation**

// A simple Java program to demonstrate

// implementation of Command Pattern using

// a remote control example.

// An interface for command

interface Command

{

public void execute();

}

// Light class and its corresponding command

// classes

class Light

{

public void on()

{

System.out.println("Light is on");

}

public void off()

{

System.out.println("Light is off");

}

}

class LightOnCommand implements Command

{

Light light;

// The constructor is passed the light it

// is going to control.

public LightOnCommand(Light light)

{

this.light = light;

}

public void execute()

{

light.on();

}

}

class LightOffCommand implements Command

{

Light light;

public LightOffCommand(Light light)

{

this.light = light;

}

public void execute()

{

light.off();

}

}

// Stereo and its command classes

class Stereo

{

public void on()

{

System.out.println("Stereo is on");

}

public void off()

{

System.out.println("Stereo is off");

}

public void setCD()

{

System.out.println("Stereo is set " +

"for CD input");

}

public void setDVD()

{

System.out.println("Stereo is set"+

" for DVD input");

}

public void setRadio()

{

System.out.println("Stereo is set" +

" for Radio");

}

public void setVolume(int volume)

{

// code to set the volume

System.out.println("Stereo volume set"

+ " to " + volume);

}

}

class StereoOffCommand implements Command

{

Stereo stereo;

public StereoOffCommand(Stereo stereo)

{

this.stereo = stereo;

}

public void execute()

{

stereo.off();

}

}

class StereoOnWithCDCommand implements Command

{

Stereo stereo;

public StereoOnWithCDCommand(Stereo stereo)

{

this.stereo = stereo;

}

public void execute()

{

stereo.on();

stereo.setCD();

stereo.setVolume(11);

}

}

// A Simple remote control with one button

class SimpleRemoteControl

{

Command slot; // only one button

public SimpleRemoteControl()

{

}

public void setCommand(Command command)

{

// set the command the remote will

// execute

slot = command;

}

public void buttonWasPressed()

{

slot.execute();

}

}

// Driver class

class RemoteControlTest

{

public static void main(String[] args)

{

SimpleRemoteControl remote =

new SimpleRemoteControl();

Light light = new Light();

Stereo stereo = new Stereo();

// we can change command dynamically

remote.setCommand(new

LightOnCommand(light));

remote.buttonWasPressed();

remote.setCommand(new

StereoOnWithCDCommand(stereo));

remote.buttonWasPressed();

remote.setCommand(new

StereoOffCommand(stereo));

remote.buttonWasPressed();

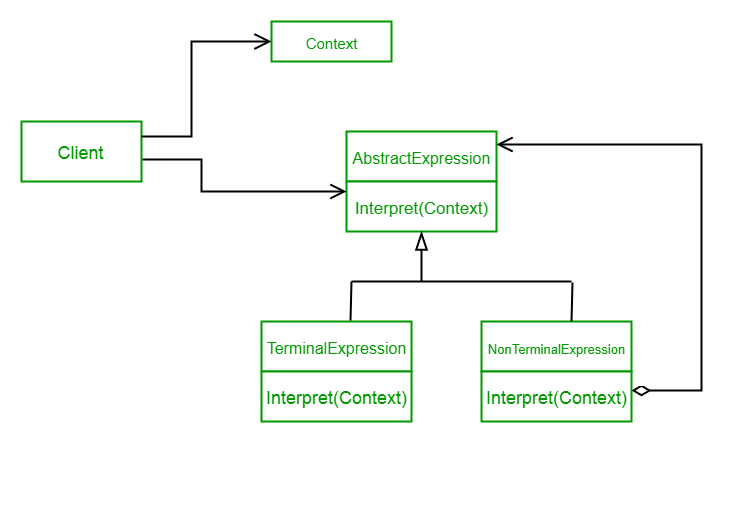
}

}

**Interpreter design pattern**

Interpreter design pattern is one of the **behavioral** design pattern. Interpreter pattern is used to defines a grammatical representation for a language and provides an interpreter to deal with this grammar.

* This pattern involves implementing an expression interface which tells to interpret a particular context. This pattern is used in SQL parsing, symbol processing engine etc.
* This pattern performs upon a hierarchy of expressions. Each expression here is a terminal or non-terminal.
* The tree structure of Interpreter design pattern is somewhat similar to that defined by the composite design pattern with terminal expressions being leaf objects and non-terminal expressions being composites.
* The tree contains the expressions to be evaluated and is usually generated by a parser. The parser itself is not a part of the interpreter pattern.



**JAVA Implementation**

// Expression interface used to

// check the interpreter.

interface Expression

{

boolean interpreter(String con);

}

// TerminalExpression class implementing

// the above interface. This interpreter

// just check if the data is same as the

// interpreter data.

class TerminalExpression implements Expression

{

String data;

public TerminalExpression(String data)

{

this.data = data;

}

public boolean interpreter(String con)

{

if(con.contains(data))

{

return true;

}

else

{

return false;

}

}

}

// OrExpression class implementing

// the above interface. This interpreter

// just returns the or condition of the

// data is same as the interpreter data.

class OrExpression implements Expression

{

Expression expr1;

Expression expr2;

public OrExpression(Expression expr1, Expression expr2)

{

this.expr1 = expr1;

this.expr2 = expr2;

}

public boolean interpreter(String con)

{

return expr1.interpreter(con) || expr2.interpreter(con);

}

}

// AndExpression class implementing

// the above interface. This interpreter

// just returns the And condition of the

// data is same as the interpreter data.

class AndExpression implements Expression

{

Expression expr1;

Expression expr2;

public AndExpression(Expression expr1, Expression expr2)

{

this.expr1 = expr1;

this.expr2 = expr2;

}

public boolean interpreter(String con)

{

return expr1.interpreter(con) && expr2.interpreter(con);

}

}

// Driver class

class InterpreterPattern

{

public static void main(String[] args)

{

Expression person1 = new TerminalExpression("Kushagra");

Expression person2 = new TerminalExpression("Lokesh");

Expression isSingle = new OrExpression(person1, person2);

Expression vikram = new TerminalExpression("Vikram");

Expression committed = new TerminalExpression("Committed");

Expression isCommitted = new AndExpression(vikram, committed);

System.out.println(isSingle.interpreter("Kushagra"));

System.out.println(isSingle.interpreter("Lokesh"));

System.out.println(isSingle.interpreter("Achint"));

System.out.println(isCommitted.interpreter("Committed, Vikram"));

System.out.println(isCommitted.interpreter("Single, Vikram"));

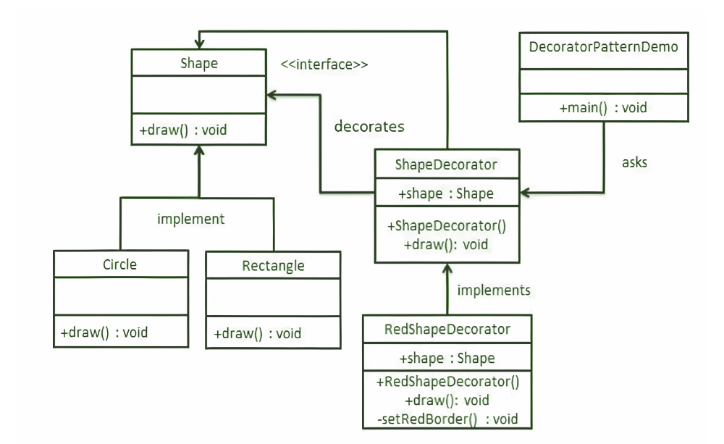
}

}

**Decorator Design Pattern**

**Decorator design pattern** allows us to dynamically add functionality and behavior to an object without affecting the behavior of other existing objects within the same class. We use inheritance to extend the behavior of the class. This takes place at compile-time, and all the instances of that class get the extended behavior.

* Decorator patterns allow a user to add new functionality to an existing object without altering its structure. So, there is no change to the original class.
* The decorator design pattern is a structural pattern, which provides a wrapper to the existing class.
* The decorator design pattern uses abstract classes or interfaces with the composition to implement the wrapper.
* Decorator design patterns create decorator classes, which wrap the original class and supply additional functionality by keeping the class methods’ signature unchanged.
* Decorator design patterns are most frequently used for applying single responsibility principles since we divide the functionality into classes with unique areas of concern.
* The decorator design pattern is structurally almost like the chain of responsibility pattern.



**Java Implementation**

// Class 1

// Class 1 will be implementing the Shape interface

// Rectangle.java

public class Rectangle implements Shape {

// Overriding the method

@Override public void draw()

{

// /Print statement to execute when

// draw() method of this class is called

// later on in the main() method

System.out.println("Shape: Rectangle");

}

}

// Circle.java

public class Circle implements Shape {

@Override

public void draw()

{

System.out.println("Shape: Circle");

}

}

// Class 2

// Abstract class

// ShapeDecorator.java

public abstract class ShapeDecorator implements Shape {

// Protected variable

protected Shape decoratedShape;

// Method 1

// Abstract class method

public ShapeDecorator(Shape decoratedShape)

{

// This keywordd refers to current object itself

this.decoratedShape = decoratedShape;

}

// Method 2 - draw()

// Outside abstract class

public void draw() { decoratedShape.draw(); }

}

// Class 3

// Concrete class extending the abstract class

// RedShapeDecorator.java

public class RedShapeDecorator extends ShapeDecorator {

public RedShapeDecorator(Shape decoratedShape)

{

super(decoratedShape);

}

@Override public void draw()

{

decoratedShape.draw();

setRedBorder(decoratedShape);

}

private void setRedBorder(Shape decoratedShape)

{

// Display message whenever function is called

System.out.println("Border Color: Red");

}

}

// DecoratorPatternDemo.java

// Class

// Main class

public class DecoratorPatternDemo {

// Main driver method

public static void main(String[] args)

{

// Creating an object of Shape interface

// inside the main() method

Shape circle = new Circle();

Shape redCircle

= new RedShapeDecorator(new Circle());

Shape redRectangle

= new RedShapeDecorator(new Rectangle());

// Display message

System.out.println("Circle with normal border");

// Calling the draw method over the

// object calls as created in

// above classes

// Call 1

circle.draw();

// Display message

System.out.println("\nCircle of red border");

// Call 2

redCircle.draw();

// Display message

System.out.println("\nRectangle of red border");

// Call 3

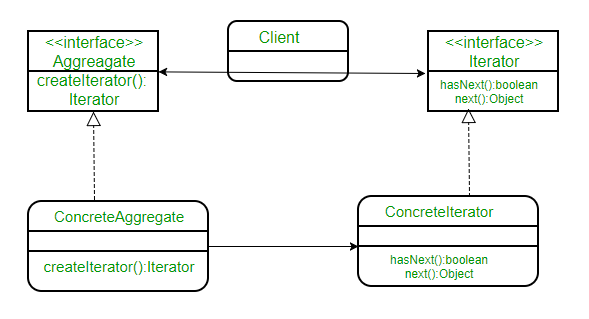
redRectangle.draw();

}

}

**Iterator Design Pattern**

Iterator design pattern in one of the behavioral pattern. Iterator pattern is used to provide a standard way to traverse through a group of Objects. Iterator pattern is widely used in [Java Collection Framework](https://www.digitalocean.com/community/tutorials/collections-in-java-tutorial). Iterator interface provides methods for traversing through a collection.



**Java Implementation**

// A Java program to demonstrate implementation

// of iterator pattern with the example of

// notifications

// A simple Notification class

class Notification

{

// To store notification message

String notification;

public Notification(String notification)

{

this.notification = notification;

}

public String getNotification()

{

return notification;

}

}

// Collection interface

interface Collection

{

public Iterator createIterator();

}

// Collection of notifications

class NotificationCollection implements Collection

{

static final int MAX\_ITEMS = 6;

int numberOfItems = 0;

Notification[] notificationList;

public NotificationCollection()

{

notificationList = new Notification[MAX\_ITEMS];

// Let us add some dummy notifications

addItem("Notification 1");

addItem("Notification 2");

addItem("Notification 3");

}

public void addItem(String str)

{

Notification notification = new Notification(str);

if (numberOfItems >= MAX\_ITEMS)

System.err.println("Full");

else

{

notificationList[numberOfItems] = notification;

numberOfItems = numberOfItems + 1;

}

}

public Iterator createIterator()

{

return new NotificationIterator(notificationList);

}

}

// We could also use Java.Util.Iterator

interface Iterator

{

// indicates whether there are more elements to

// iterate over

boolean hasNext();

// returns the next element

Object next();

}

// Notification iterator

class NotificationIterator implements Iterator

{

Notification[] notificationList;

// maintains curr pos of iterator over the array

int pos = 0;

// Constructor takes the array of notificationList are

// going to iterate over.

public NotificationIterator (Notification[] notificationList)

{

this.notificationList = notificationList;

}

public Object next()

{

// return next element in the array and increment pos

Notification notification = notificationList[pos];

pos += 1;

return notification;

}

public boolean hasNext()

{

if (pos >= notificationList.length ||

notificationList[pos] == null)

return false;

else

return true;

}

}

// Contains collection of notifications as an object of

// NotificationCollection

class NotificationBar

{

NotificationCollection notifications;

public NotificationBar(NotificationCollection notifications)

{

this.notifications = notifications;

}

public void printNotifications()

{

Iterator iterator = notifications.createIterator();

System.out.println("-------NOTIFICATION BAR------------");

while (iterator.hasNext())

{

Notification n = (Notification)iterator.next();

System.out.println(n.getNotification());

}

}

}

// Driver class

class Main

{

public static void main(String args[])

{

NotificationCollection nc = new NotificationCollection();

NotificationBar nb = new NotificationBar(nc);

nb.printNotifications();

}

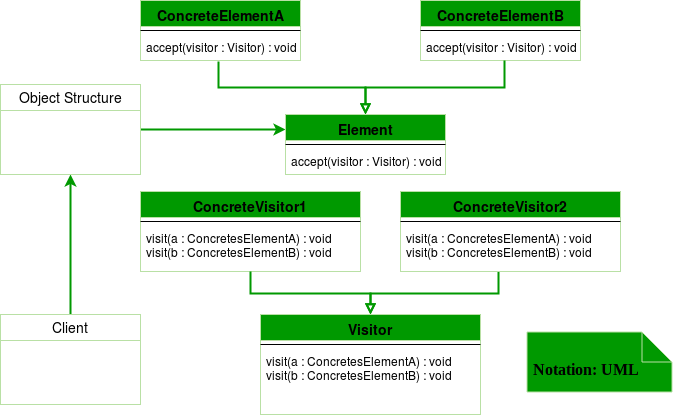
}

**Visitor Design Pattern**

Visitor design pattern is one of the behavioral design patterns. It is used when we have to perform an operation on a group of similar kind of Objects. With the help of visitor pattern, we can move the operational logic from the objects to another class.

The visitor pattern consists of two parts:

* a method called **Visit()** which is implemented by the visitor and is called for every element in the data structure
* visitable classes providing **Accept()** methods that accept a visitor



**Java Implementation**

interface ItemElement

{

public int accept(ShoppingCartVisitor visitor);

}

class Book implements ItemElement

{

private int price;

private String isbnNumber;

public Book(int cost, String isbn)

{

this.price=cost;

this.isbnNumber=isbn;

}

public int getPrice()

{

return price;

}

public String getIsbnNumber()

{

return isbnNumber;

}

@Override

public int accept(ShoppingCartVisitor visitor)

{

return visitor.visit(this);

}

}

class Fruit implements ItemElement

{

private int pricePerKg;

private int weight;

private String name;

public Fruit(int priceKg, int wt, String nm)

{

this.pricePerKg=priceKg;

this.weight=wt;

this.name = nm;

}

public int getPricePerKg()

{

return pricePerKg;

}

public int getWeight()

{

return weight;

}

public String getName()

{

return this.name;

}

@Override

public int accept(ShoppingCartVisitor visitor)

{

return visitor.visit(this);

}

}

interface ShoppingCartVisitor

{

int visit(Book book);

int visit(Fruit fruit);

}

class ShoppingCartVisitorImpl implements ShoppingCartVisitor

{

@Override

public int visit(Book book)

{

int cost=0;

//apply 5$ discount if book price is greater than 50

if(book.getPrice() > 50)

{

cost = book.getPrice()-5;

}

else

cost = book.getPrice();

System.out.println("Book ISBN::"+book.getIsbnNumber() + " cost ="+cost);

return cost;

}

@Override

public int visit(Fruit fruit)

{

int cost = fruit.getPricePerKg()\*fruit.getWeight();

System.out.println(fruit.getName() + " cost = "+cost);

return cost;

}

}

class ShoppingCartClient

{

public static void main(String[] args)

{

ItemElement[] items = new ItemElement[]{new Book(20, "1234"),

new Book(100, "5678"), new Fruit(10, 2, "Banana"),

new Fruit(5, 5, "Apple")};

int total = calculatePrice(items);

System.out.println("Total Cost = "+total);

}

private static int calculatePrice(ItemElement[] items)

{

ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();

int sum=0;

for(ItemElement item : items)

{

sum = sum + item.accept(visitor);

}

return sum;

}

}