**Java design patterns**

**Contents**

Structural design patterns ………………………………………. 2- 15

Creational pattern ………………………………………………… 16 – 24

Behavioral pattern ………………………………………………… 25 – 33

**Structural Design Patterns**

**Adapter pattern**

**Inbuilt Example :** Arrays.asList(arr). It converts legacy system array to a arraylist.

**Design :**

Legacy1 ---------------------🡪 Adpater ---------------------------🡪 Client

Legacy2 ---------------------🡪 Adpater ---------------------------🡪 Client

Client only accepts the new code, but if we need to adapt the legacy code in our system, we can use adapter to adapt the legacy1 and legacy2 code into our current code

**public** **interface** Employee {

**int** getEmployeeId();

String getEmail();

String getLastname();

}

Class EmployeeDB implemensts Employee {

String lastname; --------------------- New code

String email;

**int** employeeId;

}

Class EmployeeLdap{

String name;

String mail; ------------------- Legacy1

**int** id;

}

Class EmployeeCSV{

String firstName; --------------------------Legacy 2

String emailId;

**int** empId;

}

To adapt theses two legacy code we can write two adaper

**public** **class** EmployeeAdapteCSv **implements** Employee {

**private** EmployeeCSV instance;

**public** EmployeeAdapterLdap(EmployeeLdap instance) {

**this**.instance = instance;

}

@Override

**public** **int** getEmployeeId() {

**return** instance.getEmpId();

}

@Override

**public** String getEmail() {

**return** instance.getemailId();

}

@Override

**public** String getLastname() {

**return** instance.getfirstName();

}

}

And

**public** **class** EmployeeAdapterLdap **implements** Employee {

**private** EmployeeLdap instance;

**public** EmployeeAdapterLdap(EmployeeLdap instance) {

**this**.instance = instance;

}

@Override

**public** **int** getEmployeeId() {

**return** instance.getId();

}

@Override

**public** String getEmail() {

**return** instance.getMail();

}

@Override

**public** String getLastname() {

**return** instance.getName();

}

}

From client we can call

Employee employee = new EmployeeAdapterLdap(employeeDbObject);

Employee.getId() will call adapter method getId() but return id from EmployeeLdap

**Pitfalls :**

* Try to use multiple adapter
* Don’t add functionality in legacy code.

**Bridge Pattern**

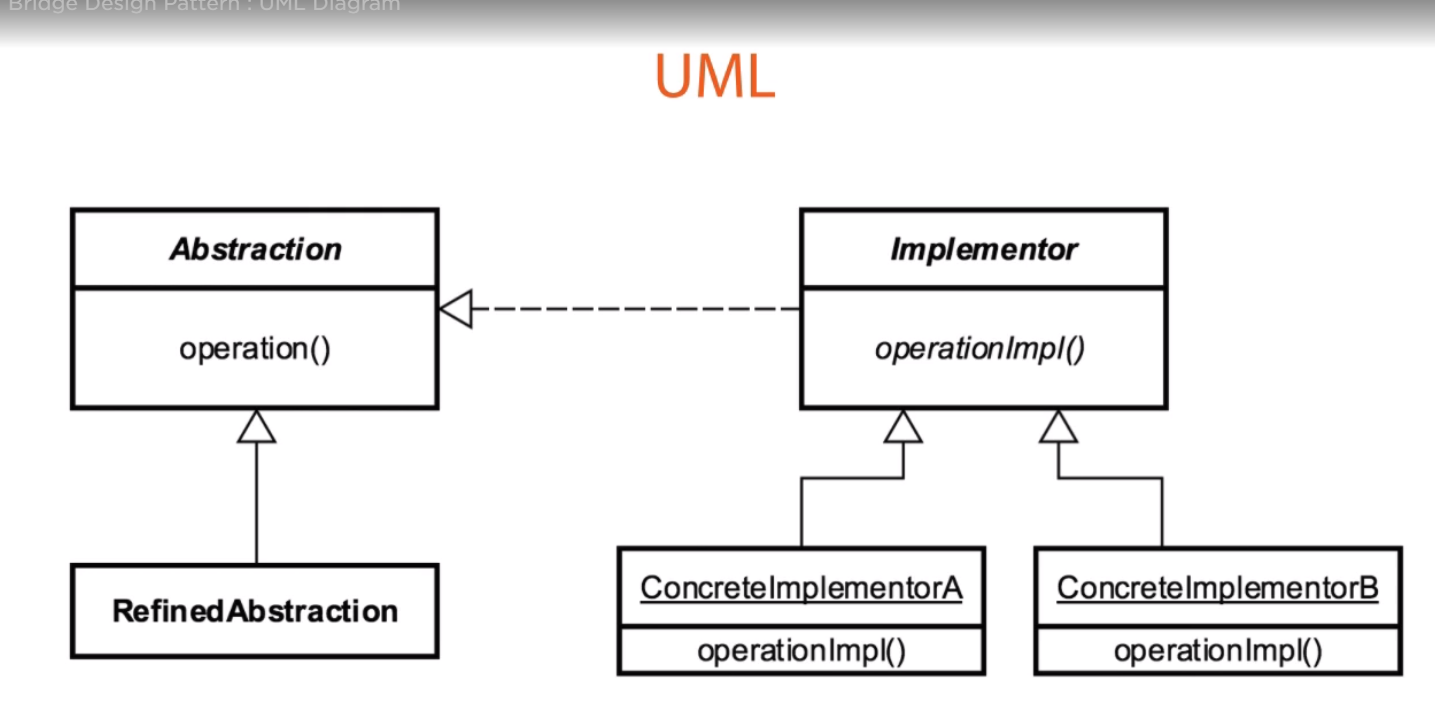
**Inbuilt example :**  Jdbc driver.

It is used to make a bridge between two logics

Client is unknown of any changes in abstract level.

Has a relationship is used here.

**UML diagram**



**Design :**

We have a movie with some properties, we need to print the movie details using some printer in some particular format. Suppose a movie has

Comedy

Name : Pink panther

Runtime : 2 hour

In future if we want to print in a different format like in html

<h1> Pink Panther <h1>

<h2> Comedy : 2 HR </h2>

In future if we want to change the format again, or if we want to print any other event rather than movie like a concert, then we need to change our code . Here Movie is a model, printer is an action and the format is another action. We need to make a bridge between printer and formatter to access the model.

**public** **class** Movie {

**private** String classification;

**private** String title;

**private** **int** runtime;

}

Printer has a formatter (Aabstraction)

**public** **abstract** **class** Printer {

String print(Formatter formatter) {

**return** formatter.format(getHeader(), getDetails());

}

**protected** **abstract** List<Detail> getDetails();

**protected** **abstract** String getHeader();

}

Concreate abstraction

MoviePrinter has a movie and it will override getDetails() and getHeader()

**public** **class** MoviePrinter **extends** Printer {

**private** Movie movie;

**public** MoviePrinter(Movie movie) {

**super**();

**this**.movie = movie;

}

@Override

**protected** List<Detail> getDetails() {

List<Detail> details = **new** ArrayList<Detail>();

details.add(**new** Detail("Title", movie.getTitle()));

details.add(**new** Detail("Runtime", String.*valueOf*(movie.getRuntime())));

**return** details;

}

@Override

**protected** String getHeader() {

**return** movie.getClassification();

}

}

Implementor

**public** **interface** Formatter {

String format(String header, List<Detail> details);

}

PrinterFormatter implements Formatter which will format in different format, not depending on Movie or Printer object.

Concreate implementor

**public** **class** PrintFormatter **implements** Formatter {

@Override

**public** String format(String header, List<Detail> details) {

StringBuilder strBuilder = **new** StringBuilder();

strBuilder.append(header);

strBuilder.append("\n");

**if**(details != **null** && !details.isEmpty()) {

**for**(Detail detail : details) {

strBuilder.append(detail.getLabel()+" : "+detail.getValue());

}

}

**return** strBuilder.toString();

}

}

Client class

Movie movie = **new** Movie("Comedy", "Pink panther", 2);

Formatter printFormatter = **new** PrintFormatter();

Printer moviPrinter = **new** MoviePrinter(movie);

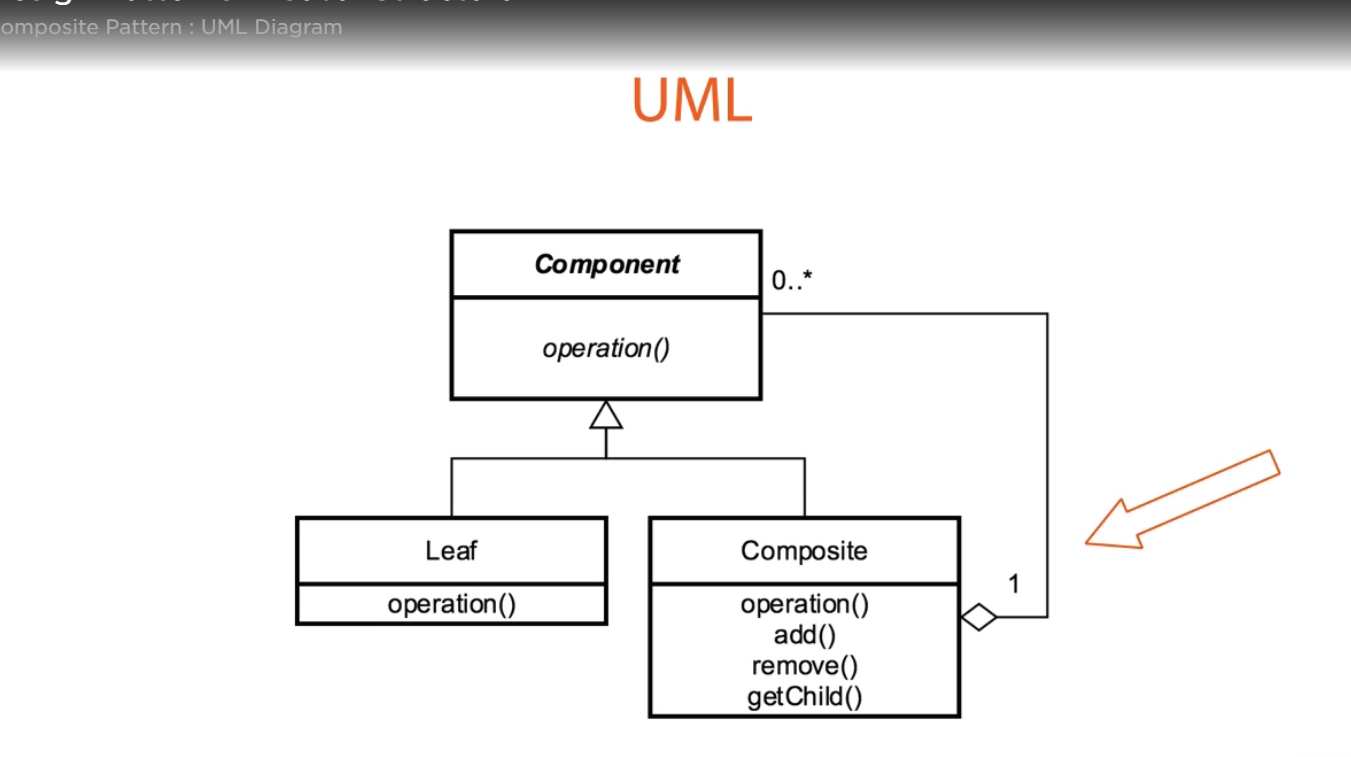
System.***out***.println(moviPrinter.print(printFormatter));

**Pitfalls :**

Complex

**Composite Pattern :**

**UML diagram :**



Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently.

**Example :**

* putAll() or addAll() methods in a collection framework re adding all composite objects, whether each collection can add leaf element using add() or put() method
* Menus that contain menu items, each of which could be a menu.
* Directories that contain files, each of which could be a directory.

**Design :**

Main

HR

Meal card

Claim

Medical

Travel

Component

**public** **abstract** **class** MenuComponent {

**protected** List<MenuComponent> menucomponents;

**protected** String name;

**protected** String url;

**public** MenuComponent() {

**this**.menucomponents = **new** ArrayList<MenuComponent>();

}

**public** List<MenuComponent> getMenucomponents() {

**return** menucomponents;

}

**public** String getName() {

**return** name;

}

**public** String getUrl() {

**return** url;

}

**public** String print(MenuComponent menuComponent) {

StringBuilder strBuilder = **new** StringBuilder();

strBuilder.append(menuComponent.getName()).append(" : ").append(menuComponent.getUrl());

strBuilder.append("\n");

**return** strBuilder.toString();

}

**public** **abstract** String toString();

}

Leaf

**public** **class** MenuItem **extends** MenuComponent {

**public** MenuItem(String name, String url) {

**this**.name = name;

**this**.url = url;

}

@Override

**public** String toString() {

**return** print(**this**);

}

}

Composite

**public** **class** Menu **extends** MenuComponent {

**public** Menu(String name, String url) {

**this**.name = name;

**this**.url = url;

}

@Override

**public** String toString() {

System.***out***.println(print(**this**));

Iterator<MenuComponent> itr = menucomponents.iterator();

StringBuilder strBuilder = **new** StringBuilder();

**while**(itr.hasNext()) {

MenuComponent menuComponent = itr.next();

strBuilder.append(print(menuComponent));

}

**return** strBuilder.toString();

}

**protected** MenuComponent add(MenuComponent menucomponenet) {

menucomponents.add(menucomponenet);

**return** menucomponenet;

}

**protected** MenuComponent remove(MenuComponent menuComponent) {

menucomponents.remove(menuComponent);

**return** menuComponent;

}

}

Client

Menu mainMenu = **new** Menu("main", "/main");

MenuItem mainMenuItem1 = **new** MenuItem("HR", "/myhr");

MenuItem mainMenuItem2 = **new** MenuItem("Meal", "/mealcard");

mainMenu.add(mainMenuItem1);

mainMenu.add(mainMenuItem2);

Menu subMenu = **new** Menu("claim", "/cliam");

MenuItem subMenuItem = **new** MenuItem("medical", "/medical");

subMenu.add(subMenuItem);

mainMenu.add(subMenu);

System.***out***.println(mainMenu.toString());

**Decorator Pattern :**

Also called wrapper

Add behavior without affecting others.

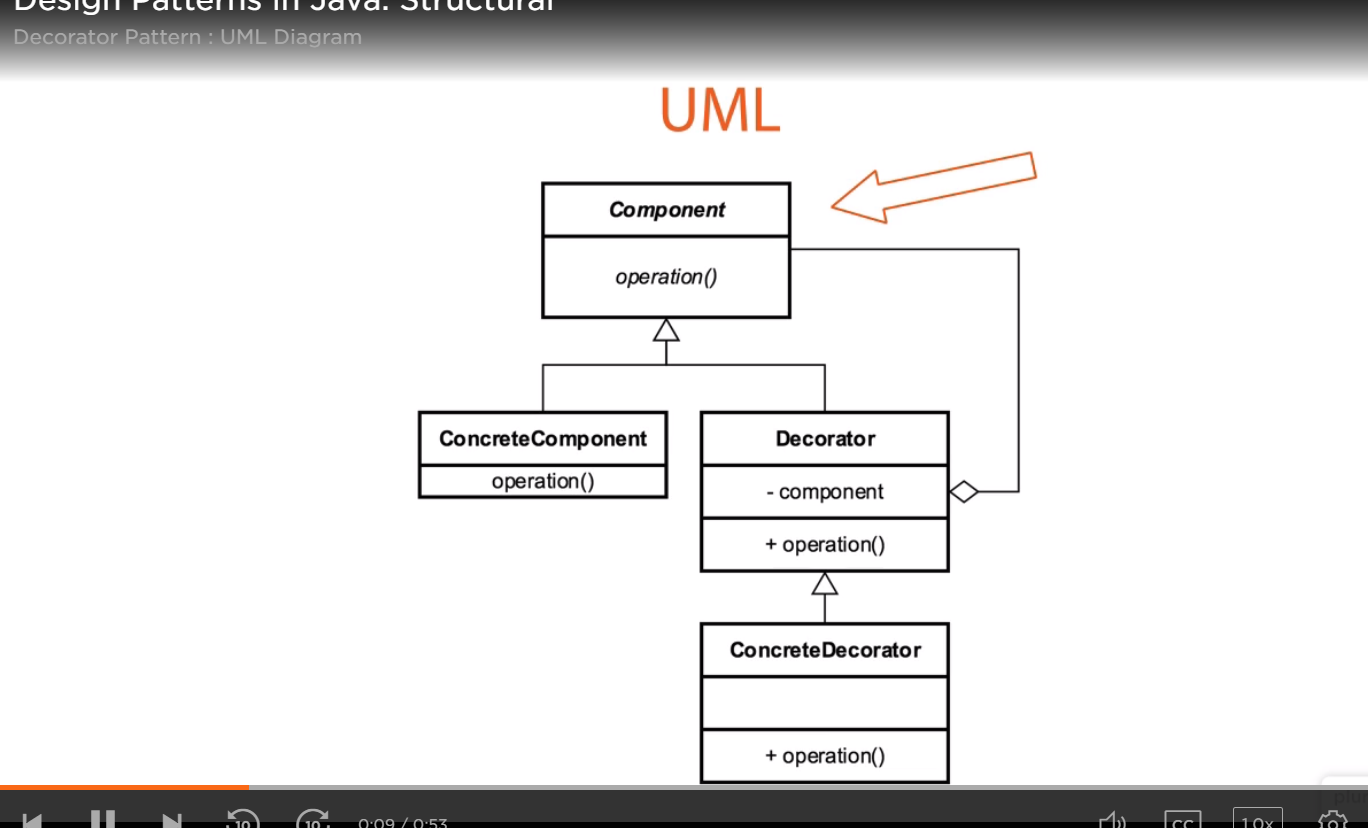
Single responsibility principal.

**Example :**

Java.io.InputStream

Java awt components

**UML :**



**Problem statement :**

We need to make a sandwich. We can make it brown or white, veg or non veg type, the decoration will be according to that only. The design should be loosly coupled and easily expandable , like in future if we want to add some sauce or mayonnaise then we can add .

Component

**public** **interface** Sandwich {

**public** String make();

}

Concreate component

**public** **class** Simplesandwich **implements** Sandwich {

@Override

**public** String make() {

**return** "Bread";

}

}

Decorator abstract class which extends component

**public** **abstract** **class** SandwichDecorator **implements** Sandwich{

**protected** Sandwich customsandwich;

**public** SandwichDecorator(Sandwich customsandwich) {

**super**();

**this**.customsandwich = customsandwich;

}

**public** SandwichDecorator() {

**super**();

}

**public** String make() {

**return** **this**.customsandwich.make();

}

}

Concreate decorator 1

**public** **class** MeatDecorator **extends** SandwichDecorator{

**private** String meat;

**public** MeatDecorator() {

**super**();

}

**public** MeatDecorator(Sandwich customsandwich, String meat) {

**super**(customsandwich);

**this**.meat = meat;

}

**private** String addMeat() {

**return** **this**.meat;

}

**public** String make() {

**return** **super**.make()+" + "+addMeat();

}

}

Concreate decorator 2

**public** **class** FillingDecorator **extends** SandwichDecorator {

**private** String fillingType;

**public** FillingDecorator(Sandwich customsandwich, String fillingType) {

**super**(customsandwich);

**this**.fillingType = fillingType;

};

**private** String addFilling() {

**return** **this**.fillingType;

}

**public** String make() {

**return** **super**.make()+" + "+addFilling();

}

}

Client

Sandwich simpleSandwich = **new** Simplesandwich();

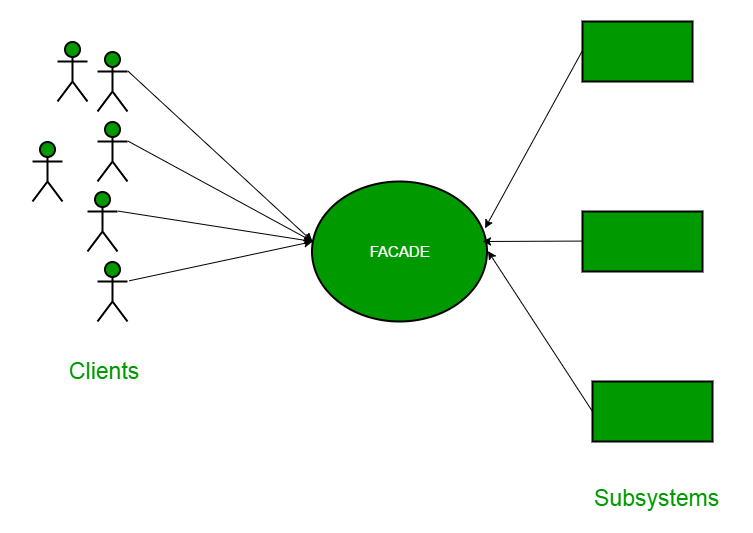
Sandwich breadTypeDecorator = **new** BreadTypeDecorator(simpleSandwich, "brown");

Sandwich meatDecorator = **new** MeatDecorator(breadTypeDecorator, "Chicken");

Sandwich fillingDecorator = **new** FillingDecorator(meatDecorator, "Mayonize");

System.***out***.println(fillingDecorator.make());

**Façade Pattern**



As the name suggests, it means the face of the building. The people walking past the road can only see this glass face of the building. They do not know anything about it, the wiring, the pipes and other complexities. It hides all the complexities of the building and displays a friendly face.

It is used when we need to hide the existing complex logic behind an interface, and client will call the interface and get the required information.

Example : URL is the example of façade pattern.

Problem statement : If we order a hotel keeper for some foods or drinks, it will go to the particular restaurant and get the menus for you according to your need. So Hotelkeeper is used here as a façade for all the restaurants which are different subsystems.

Restaurant interface

**public** **interface** Restaurant {

**public** List<String> getMenu();

}

**Menu enum**

**public** **enum** Menu {

***VEG***, ***NONVEG***, ***LIQUOR***;

}

Subsystem 1

**public** **class** VegRestaurant **implements** Restaurant {

@Override

**public** List<String> getMenu() {

List<String> menus = **new** ArrayList<String>();

menus.add("Paneer");

menus.add("Gobi");

**return** menus;

}

}

Subsystem 2

**public** **class** NonVegRestaurant **implements** Restaurant {

@Override

**public** List<String> getMenu() {

List<String> menus = **new** ArrayList<String>();

menus.add("Chicken");

menus.add("Fish");

**return** menus;

}

}

Façade class

**public** **class** HotelKeeper {

**public** List<String> getMenus(Menu menu){

List<String> menus = **null**;

**switch**(menu) {

**case** ***VEG*** :

Restaurant vegResTaurant = **new** VegRestaurant();

menus = vegResTaurant.getMenu();

**break**;

**case** ***NONVEG*** :

Restaurant nonVegRestaurant = **new** NonVegRestaurant();

menus = nonVegRestaurant.getMenu();

**break**;

**case** ***LIQUOR*** :

Restaurant liquorShop = **new** LiquorShop();

menus = liquorShop.getMenu();

**break**;

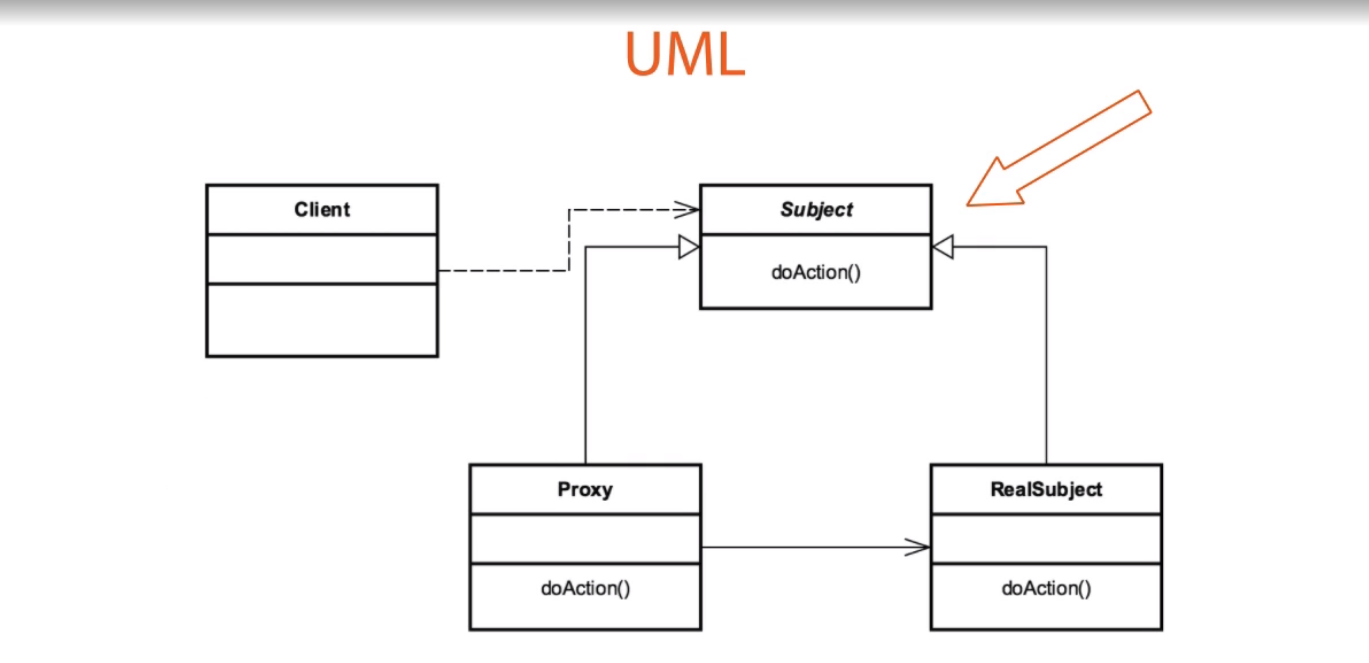
}

**return** menus;

}

}

**Proxy pattern**



If we want to add some extra feature to a object from outside the system, we need to apply proxy patter.

Here proxy and the real object has a “has a” or “is a ” relationship. This pattern is often used to apply different cross cutting feature

Logging, security, mailing

Subject

**public** **interface** TwitterService {

**public** **void** post(String param1, String param2);

**public** **void** comment(String param1);

}

Real Subject

**public** **class** TwitterServiceImpl **implements** TwitterService {

@Override

**public** **void** post(String param1, String param2) {

System.***out***.println("first post");

}

@Override

**public** **void** comment(String param1) {

System.***out***.println(" First comment ");

}

}

Proxy

**public** **class** SecurityProxy **implements** InvocationHandler{

**private** TwitterService twitterService;

**public** SecurityProxy(TwitterService twitterService) {

**this**.twitterService = twitterService;

}

@Override

**public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {

System.***out***.println("before method call"+" method name "+method.getName()+" "+method.getParameterCount());

method.invoke(twitterService, args);

System.***out***.println(" after method call"+args[0].toString());

**return** **null**;

}

}

Client

TwitterService original = **new** TwitterServiceImpl();

InvocationHandler securityPHandler = **new** SecurityProxy(original);

TwitterService twitterService = (TwitterService) Proxy.*newProxyInstance*(TwitterService.**class**.getClassLoader(),**new** Class[] { TwitterService.**class** },securityPHandler);

twitterService.post(" Hello ", "Hi" );

twitterService.comment("OK");

Output : before method call method name post 2

first post

after method call Hello

before method call method name comment 1

First comment

after method callOK

**Pitfall**

Only one proxy for one real object

**Creational pattern**

**Builder Pattern**

**Inbuilt example :**  StringBuilder is a great example of builder patern.

Main idea is to pick small small part of the application/ object and build a big application/object with the builder pattern. We often confuse with builder and decorator pattern. Let me create one pizza example with builder

Base class

**public** **class** Pizza {

**private** String bread;

**private** String nonvegTopping;

**private** String vegTopping;

**private** **double** price;

// getter setter

}

Builder class

**public** **class** PizzaBuilder {

**private** Pizza pizza;

**public** PizzaBuilder() {

pizza = **new** Pizza();

}

**public** **void** bread(String bread, **double** price) {

pizza.setBread(bread);

pizza.setPrice(pizza.getPrice()+price);

}

**public** **void** nonvegTopping(String topping, **double** price) {

pizza.setNonvegTopping(topping);

pizza.setPrice(pizza.getPrice()+price);

}

**public** **void** vegTopping(String topping, **double** price) {

pizza.setVegTopping(topping);

pizza.setPrice(pizza.getPrice()+price);

}

**public** Pizza build() {

**return** pizza;

}

}

Client method

PizzaBuilder pb1 = **new** PizzaBuilder();

pb1.bread("Thin crust", 150);

pb1.nonvegTopping("Chicken", 200);

Pizza pizza1 = pb1.build();

PizzaBuilder pb2 = **new** PizzaBuilder();

pb2.bread("Normal ", 100);

pb2.nonvegTopping("Mutton ", 300);

pb2.vegTopping("Paneer ", 250);

Pizza pizza2 = pb2.build();

System.***out***.println(" Pizza 1 "+ pizza1.getBread()+" "+pizza1.getNonvegTopping()+" "+pizza1.getVegTopping()+" "+pizza1.getPrice());

System.***out***.println(" Pizza 2 "+ pizza2.getBread()+" "+pizza2.getNonvegTopping()+" "+pizza2.getVegTopping()+" "+pizza2.getPrice());

**Difference between Builder and Decorator**

Decorator Pattern is used to add or remove additional functionalities or responsibilities from the object dynamically without impacting the original object. The use case would be that some base pizza is first prepared and then different specifications are added to it. On other hand builder will build the pizza in a single go and deliver it to client. Let’s decorate the same pizza now in decorator pattern

Component

**public** **interface** Pizza {

**public** String bake();

**public** **double** getPrice();

}

ConcreateCompnent

**public** **class** BasePizza **implements** Pizza {

@Override

**public** String bake() {

**return** "Base";

}

@Override

**public** **double** getPrice() {

**return** 100;

}

}

Abstract decorator

**abstract** **public** **class** PizzaDecorator **implements** Pizza{

**protected** Pizza pizza;

**public** PizzaDecorator(Pizza pizza) {

**this**.pizza = pizza;

}

@Override

**public** String bake() {

**return** pizza.bake();

}

**public** **double** getPrice() {

**return** pizza.getPrice();

}

}

Concreate Decorator 1

**public** **class** ChickenPizzaDecortor **extends** PizzaDecorator {

**private** String topping;

**private** **double** price;

**public** ChickenPizzaDecortor(Pizza pizza, String topping, **double** price) {

**super**(pizza);

**this**.topping = topping;

**this**.price = price;

}

**public** String getTopping() {

**return** topping;

}

**public** String bake() {

**return** **this**.topping + " "+**super**.bake();

}

**public** **double** getPrice() {

**return** **this**.price + **super**.getPrice();

}

}

Concreate Decorator 2

**public** **class** PaneerPizzaDecorator **extends** PizzaDecorator{

**private** String topping;

**private** **double** price;

**public** PaneerPizzaDecorator(Pizza pizza, String topping, **double** price) {

**super**(pizza);

**this**.topping = topping;

**this**.price = price;

}

**public** String getTopping() {

**return** topping;

}

**public** String bake() {

**return** **this**.topping + " "+**super**.bake();

}

**public** **double** getPrice() {

**return** **this**.price + **super**.getPrice();

}

}

Concreate decorator 3

**public** **class** CheeseDecorator **extends** PizzaDecorator {

**private** String topping;

**private** **double** price;

**public** CheeseDecorator(Pizza pizza, String topping, **double** price) {

**super**(pizza);

**this**.topping = topping;

**this**.price = price;

}

**public** String getTopping() {

**return** topping;

}

**public** String bake() {

**return** **this**.topping + " "+**super**.bake();

}

**public** **double** getPrice() {

**return** **this**.price + **super**.getPrice();

}

}

Client class

BasePizza basePizza = **new** BasePizza();

ChickenPizzaDecortor chickenPizzaDecortor = **new** ChickenPizzaDecortor(basePizza, "Chicken", 200);

PaneerPizzaDecorator paneerPizzaDecorator = **new** PaneerPizzaDecorator (basePizza,"Paneer",150);

CheeseDecorator finalChickenPizza = **new** CheeseDecorator(chickenPizzaDecortor, "Cheese", 50);

CheeseDecorator finalPaneerPizza = **new** CheeseDecorator(paneerPizzaDecorator, "Cheese", 50);

System.***out***.println("Chicken pizza "+finalChickenPizza.bake()+" "+finalChickenPizza.getPrice());

System.***out***.println(" Paneer pizza "+finalPaneerPizza.bake()+" "+finalPaneerPizza.getPrice());

**Singleton Pattern**

* When we don’t need to create multiple objects for a single class singleton pattern is used
* A singleton class has a private constructor, so the class cannot be initialized from outside
* No matters how many time we call Singleton class instance it will return the same object
* Example of Singleton class Runtime, utility classes

Go through this link

<https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples>

**Prototype Pattern**

* This pattern avoids costly creation
* avoids subclassing because like builder pattern it follows immutablity
* Usually implemented with registry
* Example {@link Object#clone()}
* Unlike builder pattern it is difficult implement in legacy code.
* Prototype design pattern is used when the Object creation is a costly affair and requires a lot of time

and resources and you have a similar object already existing.

* We can achieve this by using clone() method but it will create just another instance of the same object.

Abstract class which implements Clonnable

**abstract** **class** Item **implements** Cloneable{

**private** String name;

**private** Long price;

// getter setter

@Override

**public** Object clone() **throws** CloneNotSupportedException {

**return** **super**.clone();

}

}

Concreate class 1

**class** Book **extends** Item{

**private** **int** numberofPages;

**public** **int** getNumberofPages() {

**return** numberofPages;

}

**public** **void** setNumberofPages(**int** numberofPages) {

**this**.numberofPages = numberofPages;

}

}

Concreate class 2

**class** Dvd **extends** Item{

**private** **double** runTime;

**public** **double** getRunTime() {

**return** runTime;

}

**public** **void** setRunTime(**double** runTime) {

**this**.runTime = runTime;

}

}

Registry class

**class** ItemRegistry{

**private** Map<Type,Item> items = **new** HashMap<Type,Item>();

**public** ItemRegistry() {

loadItems();

}

**public** Item createItem(Type type) **throws** CloneNotSupportedException {

**return** (Item)items.get(type).clone();

}

**private** **void** loadItems() {

Book book = **new** Book();

book.setName("gangs of four");

book.setPrice(335L);

book.setNumberofPages(1000);

items.put(Type.***BOOK***, book);

Dvd dvd = **new** Dvd();

dvd.setName("Gangs of wassepur");

dvd.setPrice(100L);

dvd.setRunTime(2.54);

items.put(Type.***DVD***, dvd);

}

**public** **static** **enum** Type{

***BOOK***,***DVD***;

}

}

Client class

ItemRegistry itemRegistry = **new** ItemRegistry();

Book book = (Book) itemRegistry.createItem(ItemRegistry.Type.***BOOK***);

Dvd dvd = (Dvd)itemRegistry.createItem(ItemRegistry.Type.***DVD***);

System.***out***.println("Books data"+book.getName()+" "+book.getPrice()+" "+book.getNumberofPages());

System.***out***.println("DVD data"+dvd.getName()+" "+dvd.getPrice()+" "+dvd.getRunTime());

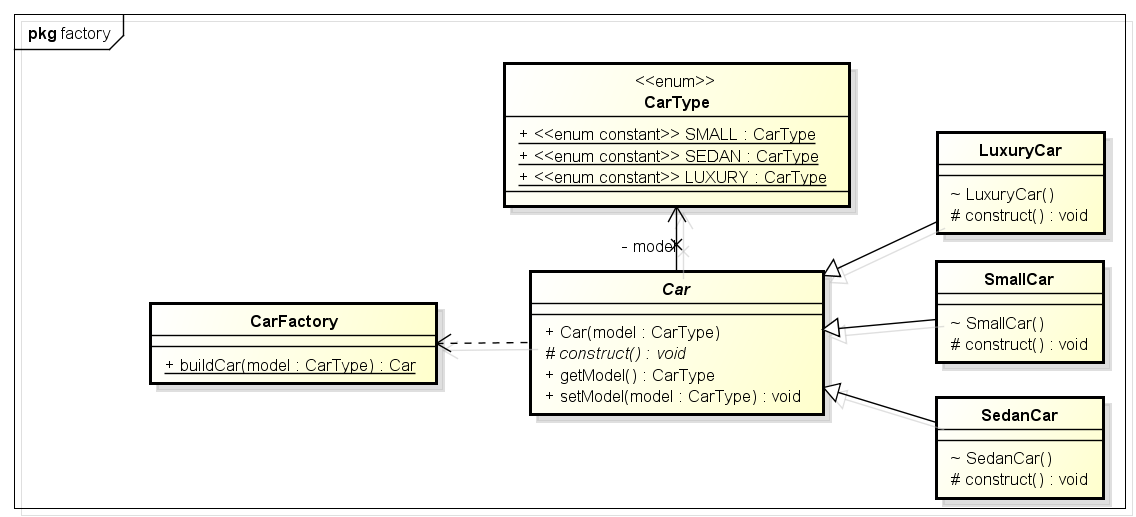
When we create a book using the method createItem(), it is actually returning the same object of Book. Because in registry we usually don’ change anything,

**Factory pattern**

* It does not expose instantiation logic
* Client only knows about a common interface that factoy exposes.
* The lifetime management of the generated objects must be centralized to ensure a consistent behavior within the application.

**Inbuilt example:** Calendar, NumberFormat

Lets crete a Car based on the type. CarFactory will create the cars.



Type

**public** **enum** CarType {

***SMALL***, ***SEDAN***, ***LUXURY***

}

Abstract class

**public** **abstract** **class** Car {

**protected** CarType model;

**protected** String name;

**protected** **long** price;

**protected** **abstract** **void** construct();

**public** Car(CarType model) {

**this**.model = model;

construct();

}

// getter setter

}

Concreate class 1

**public** **class** LuxaryCar **extends** Car{

**public** LuxaryCar() {

**super**(CarType.***LUXURY***);

}

@Override

**protected** **void** construct() {

**this**.name = "Audi";

**this**.price = 30000000;

}

}

Concreate class 2

**public** **class** SmallCar **extends** Car{

**public** SmallCar(CarType model) {

**super**(model);

}

@Override

**protected** **void** construct() {

**this**.name = "Alto 800";

**this**.price = 200000;

}

}

Concreate class 3

**public** **class** SedanCar **extends** Car{

**public** SedanCar() {

**super**(CarType.***SEDAN***);

}

@Override

**protected** **void** construct() {

**this**.name = "Honda city";

**this**.price = 1000000;

}

}

Factory class

**public** **class** CarFactory {

**public** Car buildCar(CarType carType) {

Car car = **null**;

**switch**(carType) {

**case** ***LUXURY*** :

car = **new** LuxaryCar();

**break**;

**case** ***SEDAN*** :

car = **new** SedanCar();

**break**;

**case** ***SMALL*** :

car = **new** SmallCar(carType);

**break**;

}

**return** car;

}

}

Client class

CarFactory carFactory = **new** CarFactory();

Car luxary = carFactory.buildCar(CarType.***LUXURY***);

Car sedan = carFactory.buildCar(CarType.***SEDAN***);

Car small = carFactory.buildCar(CarType.***SMALL***);

System.***out***.println(" Small "+small.getName()+" "+small.getPrice());

System.***out***.println(" Sedan "+sedan.getName()+" "+sedan.getPrice());

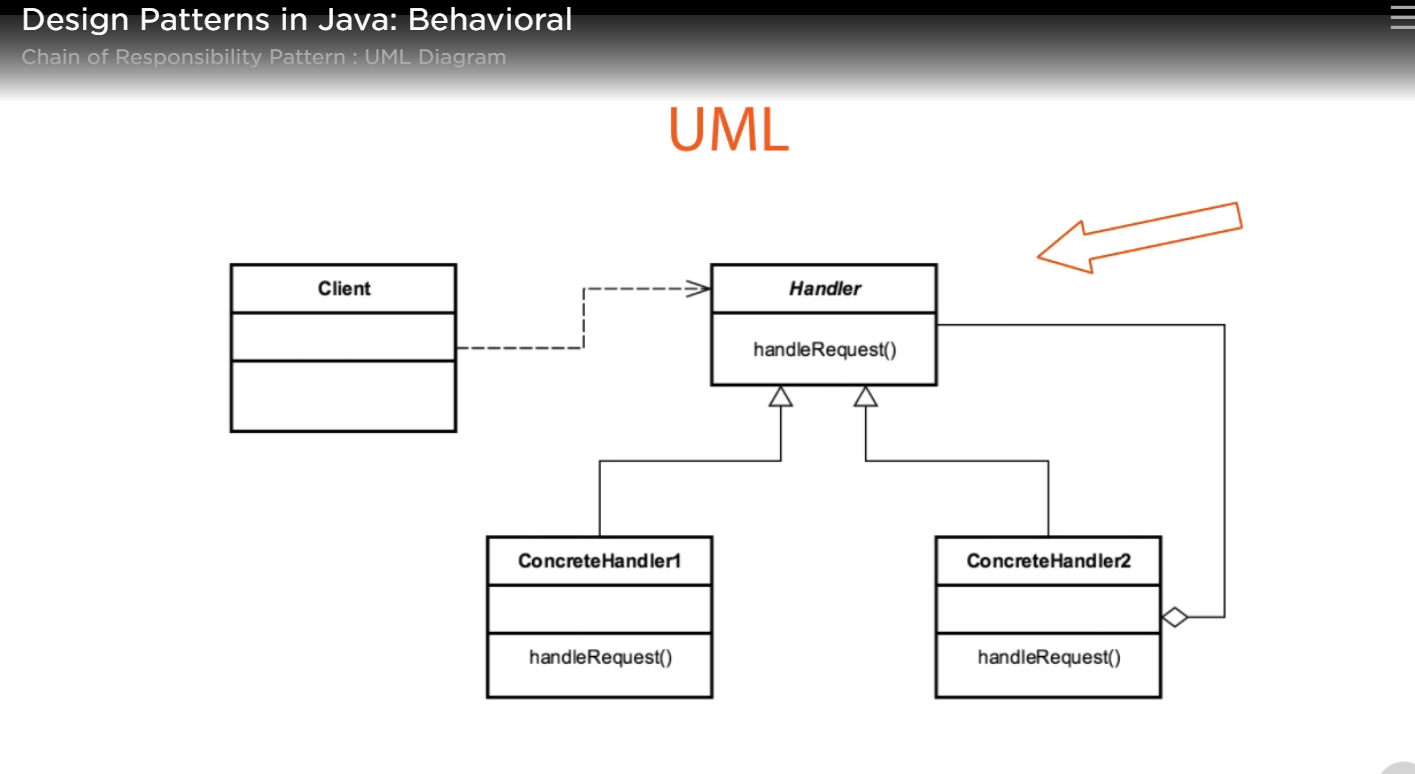
System.***out***.println(" Luxary "+luxary.getName()+" "+luxary.getPrice());

**Behavioral pattern**

**Chain of responsibility pattern :**

**Inbuilt example :**  Spring security filter, javax.servlet.Filter#doFilter()

* Sender and receiver within a chain, where a sender knows who is the next.
* Decouple between objects, we can add or remove classes or objects from the chain anytime



Lets build a system which approves request of a type and if a hadler does not handle the request it moves escalate it to his senior, but cannot defer it to its junior.

Request

**public** **class** Request {

**private** RequestType reqType;

**private** **double** amount;

// getter setter

}

RequestType

**public** **enum** RequestType {

***PURCHASE***, ***CONFERENCE***;

}

Abstract handler

**public** **abstract** **class** Handler {

**protected** Handler successor;

**public** **void** setSuccessor(Handler successor) {

**this**.successor = successor;

}

**public** **abstract** **void** handleRequest(Request request);

}

Concreate handler 3

**public** **class** Director **extends** Handler{

@Override

**public** **void** handleRequest(Request request) {

**if**(request.getReqType() == RequestType.***CONFERENCE***) {

System.***out***.println(" Director approves conference ");

}**else** {

successor.handleRequest(request);

}

}

}

Concreate handler 2

**public** **class** Vp **extends** Handler{

@Override

**public** **void** handleRequest(Request request) {

**if**(request.getReqType() == RequestType.***PURCHASE***) {

**if**(request.getAmount() < 10000) {

System.***out***.println(" Vp approves the order of "+request.getReqType()+" amount "+request.getAmount());

}**else** {

successor.handleRequest(request);

}

}

}

}

Concreate handler 3

**public** **class** CEO **extends** Handler{

@Override

**public** **void** handleRequest(Request request) {

System.***out***.println(" CEO approves "+request.getReqType()+" amount "+request.getAmount());

}

}

Client class

Request confReq = **new** Request(RequestType.***CONFERENCE***, 0);

Request purchaseSmall = **new** Request(RequestType.***PURCHASE***, 9000);

Request purchaseBig = **new** Request(RequestType.***PURCHASE***, 20000);

Director director = **new** Director();

Vp vp = **new** Vp();

CEO ceo = **new** CEO();

director.setSuccessor(vp);

vp.setSuccessor(ceo);

// fire the responsiblity

System.***out***.println(" passing to director ");

director.handleRequest(confReq);

director.handleRequest(purchaseSmall);

director.handleRequest(purchaseBig);

System.***out***.println(" passing to VP ");

vp.handleRequest(confReq);

vp.handleRequest(purchaseSmall);

vp.handleRequest(purchaseBig);

System.***out***.println(" passing to CEO ");

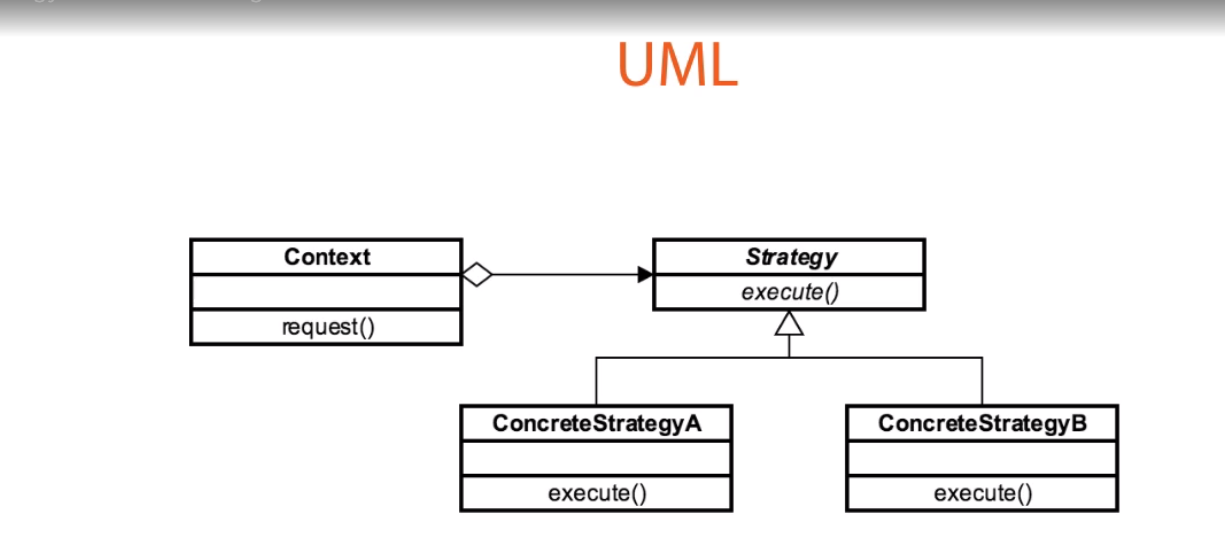
ceo.handleRequest(confReq);

ceo.handleRequest(purchaseSmall);

ceo.handleRequest(purchaseBig);

vp.handleRequest(confReq) cannot escalate to its junior, so this request cannot be processed, to process this request we need to hndle the Vp handler in different way

**Strategy Pattern :**



Context class

**public** **class** CreditCard {

String number;

**int** cvv;

Date expire;

Strategy strategy;

}

Abstract strategy

**public** **abstract** **class** Strategy {

**public** **abstract** **boolean** isValid(CreditCard creditCard);

**protected** **final** **boolean** luhnAlgo(String cardNo) {

// card validation algo

}

**}**

Concreate strategy 1

**public** **class** AmexStrategy **extends** Strategy{

@Override

**public** **boolean** isValid(CreditCard creditCard) {

**boolean** isValid = **false**;

isValid = creditCard.getNumber().length() == 14;

isValid = isValid && luhnAlgo(creditCard.getNumber());

Date today = **new** Date();

isValid = isValid && today.before(creditCard.getExpire());

**return** isValid;

}

}

Concreate strategy 2

**public** **class** VisaStrategy **extends** Strategy{

@Override

**public** **boolean** isValid(CreditCard creditCard) {

**boolean** isValid = **false**;

isValid = creditCard.getNumber().length() == 16;

//isValid = isValid && luhnAlgo(creditCard.getNumber());

Date today = **new** Date();

isValid = isValid && today.before(creditCard.getExpire());

**return** isValid;

}

}

Client class

CreditCard card1 = **new** CreditCard("4111111111111111", 123, **new** ~~Date~~(2023-1900, 3, 12));

card1.setStrategy(**new** VisaStrategy());

CreditCard card2 = **new** CreditCard("34852721055858", 321, **new** ~~Date~~(2022-1900, 3, 12));

card2.setStrategy(**new** AmexStrategy());

CreditCard card3 = **new** CreditCard("32345678909876", 123, **new** ~~Date~~(2018-1900, 3, 12));

card3.setStrategy(**new** AmexStrategy());

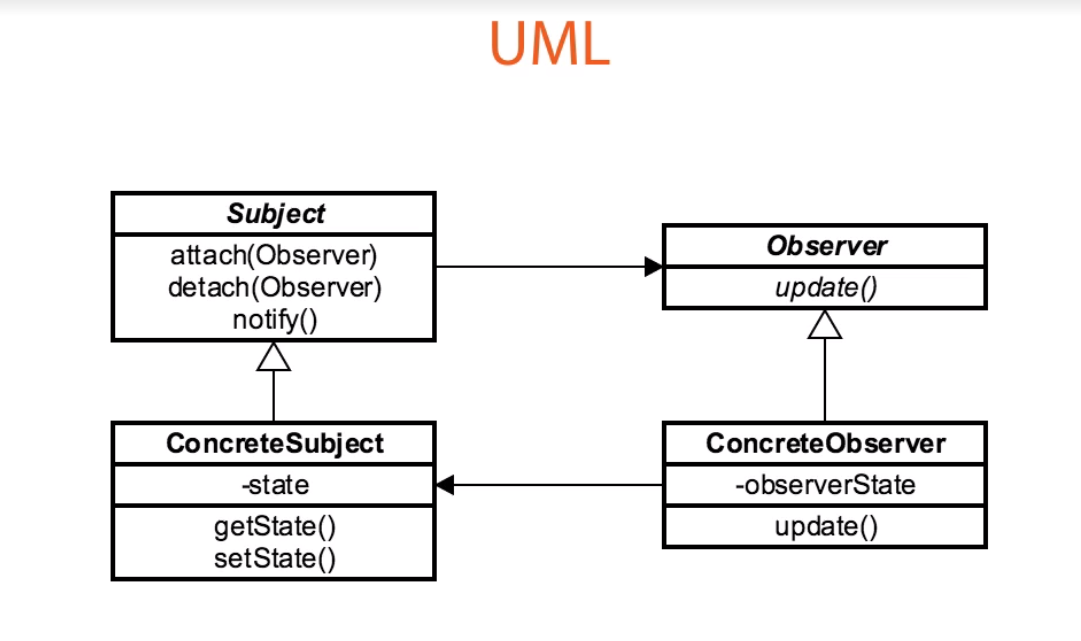
System.***out***.println(" card1 is visa card "+card1.isValid());

System.***out***.println(" card2 is amex card "+card2.isValid());

System.***out***.println(" card3 is amex card "+card3.isValid());

**Observer Pattern**

* A subjet which needs to be observed.
* It can be compared with publisher subscriber pattern, but publisher subscriber is a asynchronous but this is synchronous.
* Used in mvc architecture where views are the observers.



Subject

**abstract** **class** MyObservable{

**private** Set<MyObserver> observers = **new** HashSet<MyObserver>();

**public** **void** notifyObservers() {

**if**(!observers.isEmpty()) {

**for**(MyObserver observer : observers) {

observer.update(**this**);

}

}

}

**public** **boolean** addObserver(MyObserver observer) {

**return** observers.add(observer);

}

}

Concreate subject

**class** ConcreatObservable **extends** MyObservable{

**public** **void** method1() {

System.***out***.println(" method1 called");

notifyObservers();

}

**public** **void** method2() {

System.***out***.println(" method2 called");

}

**public** **void** method3() {

System.***out***.println(" method3 called");

}

}

Observer

**interface** MyObserver{

**public** **void** update(Object ob);

}

Concreate observer 1

**class** ConcreatObserver1 **implements** MyObserver{

@Override

**public** **void** update(Object ob) {

System.***out***.println(" first observer called");

**if**(ob **instanceof** ConcreatObservable) {

((ConcreatObservable) ob).method2();

}

}

}

Concreate observer 2

**class** ConcreatObserver2 **implements** MyObserver{

@Override

**public** **void** update(Object ob) {

System.***out***.println(" second observer called");

**if**(ob **instanceof** ConcreatObservable) {

((ConcreatObservable) ob).method3();

}

}

}

Client class

ConcreatObservable concreatObservable = **new** ConcreatObservable();

ConcreatObserver1 observer1 = **new** ConcreatObserver1();

ConcreatObserver2 observer2 = **new** ConcreatObserver2();

concreatObservable.addObserver(observer1);

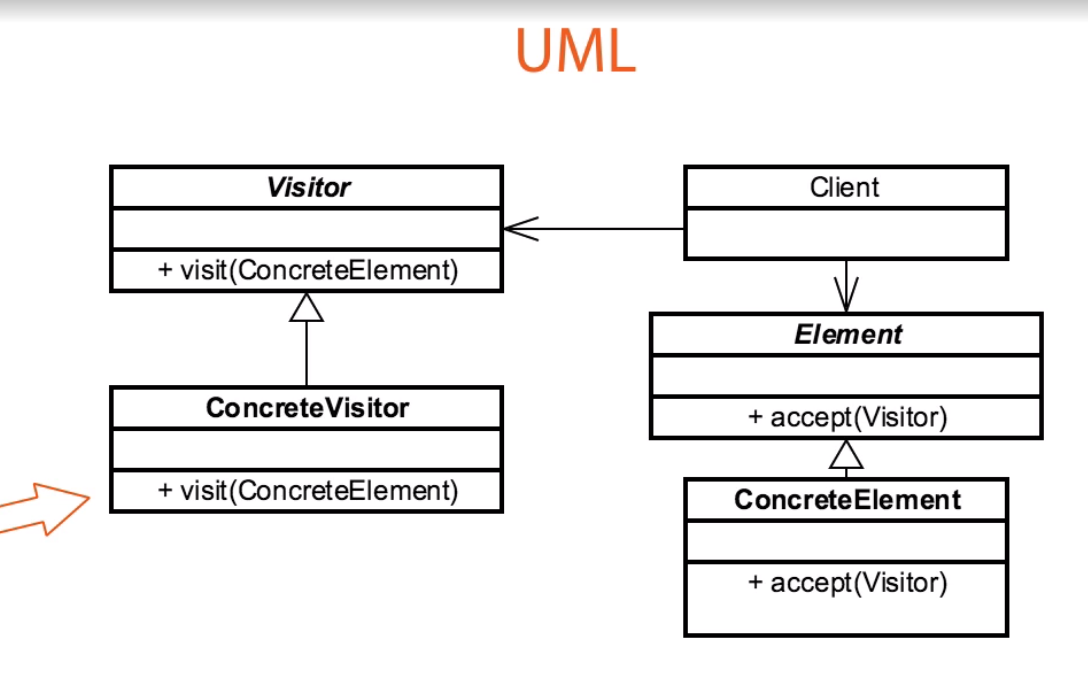
concreatObservable.addObserver(observer2);

concreatObservable.method1();

We called method 1 but observer1 and 2 called method3 when method1 is triggered.

**Visitor pattern**

* When we want to segregate the main business logic with object structure we used this pattern
* This pattern is not going to hamper the existing object structure, if there is any logic change.
* This is an interface based pattern, annotation can also be used.
* Elements has accept method
* Visitors know every element.



Element interface

**public** **interface** Item {

**int** accept(ShoppingVisitor visitor);

}

Concreate element 1

**public** **class** Book **implements** Item {

**private** String name;

**private** **int** price;

@Override

**public** **int** accept(ShoppingVisitor visitor) {

**return** visitor.visit(**this**);

}

// getter setter

}

Concreate element 2

**public** **class** Fruit **implements** Item{

**private** String name;

**private** **int** price;

**private** **int** weight;

@Override

**public** **int** accept(ShoppingVisitor visitor) {

**return** visitor.visit(**this**);

}

}

Visitor

**public** **interface** ShoppingVisitor {

**int** visit(Book book);

**int** visit(Fruit fruit);

}

Concreate visitor 1

**public** **class** ShoppingVisitorImpl **implements** ShoppingVisitor{

**public** **int** visit(Book book) {

System.***out***.println(" Visited the book "+book.getName());

**return** book.getPrice();

}

@Override

**public** **int** visit(Fruit fruit) {

System.***out***.println(" Visited the fruit "+fruit.getName()+" of weight "+fruit.getWeight());

**return** fruit.getWeight()\*fruit.getPrice();

}

}

Client class

List<Item> items = **new** ArrayList<Item>();

items.add(**new** Book("Physics ", 500));

items.add(**new** Book("Datastructure", 400));

items.add(**new** Fruit("mango", 50, 2));

items.add(**new** Fruit("Watermelon", 40, 4));

ShoppingVisitor shoppingVisitor = **new** ShoppingVisitorImpl();

**int** totatCost =0;

**for**(Item item : items) {

totatCost += item.accept(shoppingVisitor);

}

System.***out***.println(" Total shopping cost "+totatCost);

* We can add more than one visitor concreate class, then it will be very tough to handle.
* We can add one more element without affecting the existing logic.
* This is good for open close principal.