**SOLID**

* **SOLID** - serie de principii ce pot fi aplicate asupra codului ca el sa fie cat mai bun
* **S - Single-resposnability** - se poate aplica la metode,clase,module etc. si inseamna ca fiecare dintre ele trebuie sa fac un singur lucru concret, adica sa aiba o singura responsabilitate. De ex, fie o clasa Book, ce are fielduri, getters, setters, toString. Asa e bine, caci clasa e doar pentru a stoca date. Insa, daca mai adaugam metode, ca printTextUpperCase(), replaceText() etc. incalcam principiul de single responsability. Aceste metode trebuie separate intr-o noua clasa.
* **O - open/closed - adica open for extension, closed for modification**. De exemplu, cream o clasa Guitar:

public class Guitar {

private String make;

private String model;

private int volume;

//Constructors, getters & setters

}

Dupa un timp, decidem ca vrem sa adaugam niste chestii in plus la guitara, ca flame, adica daca are culoare in flacari sau nu. Nu e o idee buna sa modificam clasa Guitar, caci am putea sa influentam modul in care ea functioneaza acum, de aceea, cel ami bine extindem clasa in una noua:

public class SuperCoolGuitarWithFlames extends Guitar {

private String flameColor;

//constructor, getters + setters

}

* **L** - Liskov substitution - Daca clasa A este un subtip a clasei B, atunci trebuie sa fim capabili sa inclouim B cu A fara a influenta programul. De exemplu: fie ca avem clasa Bird si metoda fly():

public class Bird{

public void fly(){}

}

Acum, clasa Duck extinde Bird si duck poate zbura, nu e nimic anormla ca acceseaza fly()

public class Duck extends Bird{}

Dar, si strutul(Ostrich) e pasare, dar el nu poate zbura

public class Ostrich extends Bird{}

clasa Ostrich incalca principiul Liskov Substitution, caci putem inlocui Bird cu Ostrich, dar stricam logica

Acum, sa zicem ca vrem sa cream o aplicatie ce se contecteaza la o baza de date

Cream:

interface Connection{

void connect();

}

Si cream clasa

class DataBaseConnection implements Connection{

@Override

void connect() { }

}

Si, cand vom folosi @Autowired la vreo Connection, si daca se va injecta un DataBaseConnection object, totul e bine.

Dar, daca cream sa zicem apoi un alt tip de conexiune, gen

class ServerConnection implements Connection{

@Override

void connect() { }

}

Si daca vom da @Autowired si asta va fi unica clasa, programul se va strica, caci nu se face conexiune la BD, ci la Server. De asta, ca solutie, trebuie interfete inca separate pentru ServerConnection si DataBaseConnection

* **I** - Interface segregation - interfetele mari trebuie descompuse in interfete mici. Asa ne asiguram ca clasele care implementeaza anumite metode sunt bazate strict anume pe acele metode. Daca o clasa implementeaza o inerfata ce are o metoda de care ea nu are nevoie, problema e ca clasa va trebui oricum sa suprascrie metoda.
* **D** - dependecy inversion - se bazeaza pe decuplarea la software modules. Asa, in loc de a folosi propriu zis obiecte, folosim interfete si deci high-level modules nu depind de low-leves modules.

De ex:

Fie clasa Windows98Machine:

public class Windows98Machine {

private final StandardKeyboard keyboard;

private final Monitor monitor;

public Windows98Machine() {

monitor = new Monitor();

keyboard = new StandardKeyboard();

}

}

Vedem ca noi oferim un obiect de tip Monitor si StandardKeyboard, care sunt bune pentru Windows 98

Totusi, facand asta, am cuplat clasele Windows98Machine, Monitor si StandardKeyboard impreuna. Acum, apare un windows nou, si Monitor si StandardKeyboard vor fi imbunatatite, dar problema e ca ele nu sunt cele care au fost pentru Windows98. Solutia este de a folosi interfete pentru Monitor si Keyboard in loc de clase concrete:

public class Windows98Machine{

private final Keyboard keyboard;

private final Monitor monitor;

public Windows98Machine(Keyboard keyboard, Monitor monitor) {

this.keyboard = keyboard;

this.monitor = monitor;

}

}

interfate Keyboard{ }

class StandardKeyboard implements Kwyboard

interface Monitor { }

class StandardMonitor implements Monitor {

Acum clasele sunt decuplate si comunica prin abstractizare. Putem usor schimba tipul de Keyboars sau de Monitor

**Decorator**

* Permite sa adaugam noi functionalitati la un obiect existent fara a-i altera structura la runtime
* Acest pattern creaza o clasa decorator care impacheteaza clasa originala si ofera functionalitati aditionale pastrand metodele claei intacte.
* De exemplu, fie ca avem un Car interface. Putem cream o clasa ce o implementeaza, gen BasicCar, ce creaza o car general, si apoi altele 2 mai concrete, gen SportsCar si LuxuryCar



* Dar, daca vrem sa cream un car ce este si sport si luxury, aici e problema cam. Mostenirea nu ne poate ajuta, de aceea folosim decorator

1. Avem nevoie de o interfata:

interface Car {  
  
 void assemble();  
}

2. Avem nevoie de o implementre concreta:

class BasicCar implements Car {  
  
 @Override  
 public void assemble() {  
 System.*out*.print("Basic Car.");  
 }  
  
}

3. Acum, ne trebuie o clasa care implementeaza interfata si are o relatie HAS-A cu interfata. Ideea este ca clasele copii, create cu decorator, trebuie sa aiba acces la metoda assemble() a obiectului trimis. Obiectul trimis mereu va trimis in constructor cu un BasicCar. Rolul la la CarDecorator e pur si simplu de a rula metoda obiectului de tip Car trimis cu aceeasi metoda ca la interfata si gata

class CarDecorator implements Car {  
  
 private Car car;  
  
 public CarDecorator(Car c){  
 this.car=c;  
 }  
  
 @Override  
 public void assemble() {  
 this.car.assemble();  
 }  
  
}

4. Cream decoratori concreti, care vor apela metoda la CarDeecorator numita assemble(), ce apeleaza si ea metoda la Car(), si plus SportsCar va mai face ceva specific la sports car

class SportsCar extends CarDecorator {  
  
 public SportsCar(Car c) {  
 super(c);  
 }  
  
 @Override  
 public void assemble(){  
 super.assemble();  
 System.*out*.print(" Adding features of Sports Car.");  
 }  
}  
class LuxuryCar extends CarDecorator {  
  
 public LuxuryCar(Car c) {  
 super(c);  
 }  
  
 @Override  
 public void assemble(){  
 super.assemble();  
 System.*out*.print(" Adding features of Luxury Car.");  
 }  
}

Si avem main:

{  
 public static void main( String[] args ) {  
 Car sportsCar = new SportsCar(new BasicCar());  
 sportsCar.assemble();  
 System.*out*.println("\n\*\*\*\*\*");  
  
 Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new BasicCar()));  
 sportsLuxuryCar.assemble();  
 }

Vedem ca am putut crea un sportsCar si apoi un car care este deja si luxury si sports

E important mereu sa oferim un obiect care implementeaza interfata, aici BasicCar, cand cream obiecte de tip copiii ai decorator, ca el sa creeze functiunile de baza la oricare Car, adica BasicCar, si deja apoi orice setam noi la ale tippuri de cars. Putem face macar si u car ce cuprinde specificatile la 1000 de cars, cu new(new(new(new....)))), la oate se vor executa metoda assemble()



**Proxy**

* Prevede sa cream o clasa proxy, care implementam interfata unui obiect, si sa avem un obiect de acel tip in clasa proxy.

public interface ExpensiveObject {

void process();

}

public class ExpensiveObjectImpl implements ExpensiveObject {

public ExpensiveObjectImpl() {

heavyInitialConfiguration();

}

@Override

public void process() {

LOG.info("processing complete.");

}

private void heavyInitialConfiguration() {

LOG.info("Loading initial configuration...");

}

public class ExpensiveObjectProxy implements ExpensiveObject {

private static ExpensiveObject object;

@Override

public void process() {

if (object == null) {

object = new ExpensiveObjectImpl();

}

object.process();

}

}

}

**Factory**

Permite crearea obiectelor fara a expune logica . Incalca single responsability.

public interface Shape {

void draw();

}

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Inside Rectangle::draw() method.");

}

}

public class Square implements Shape {

@Override

public void draw() {

System.out.println("Inside Square::draw() method.");

}

}

public class Circle implements Shape {

@Override

public void draw() {

System.out.println("Inside Circle::draw() method.");

}

}

public class ShapeFactory {

//use getShape method to get object of type shape

public Shape getShape(String shapeType){

if(shapeType == null){

return null;

}

if(shapeType.equalsIgnoreCase("CIRCLE")){

return new Circle();

} else if(shapeType.equalsIgnoreCase("RECTANGLE")){

return new Rectangle();

} else if(shapeType.equalsIgnoreCase("SQUARE")){

return new Square();

}

return null;

}

}

**AbstractFactory**

* Factory pattern are o singura clasa Factory, care returneaza diferite obiecte de la alte clase bazate pe if-uri ce decid ce obiect sa fie creat in dependenta de ce parametru oferim noi.
* In Abstract factory pattern, scapam de if-uri si avem o factory class pentru fiecare sub clasa. Apoi, vom avea o clasa Abstract Factory care va returna obiecte in dependeta de input folosind clasele factory.

public abstract class Computer {

public abstract String getRAM();

public abstract String getHDD();

public abstract String getCPU();

@Override

public String toString(){

return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+", CPU="+this.getCPU();

}

}

public class PC extends Computer {

private String ram;

private String hdd;

private String cpu;

public PC(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public String getRAM() {

return this.ram;

}

@Override

public String getHDD() {

return this.hdd;

}

@Override

public String getCPU() {

return this.cpu;

}

}

public class Server extends Computer {

private String ram;

private String hdd;

private String cpu;

public Server(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public String getRAM() {

return this.ram;

}

@Override

public String getHDD() {

return this.hdd;

}

@Override

public String getCPU() {

return this.cpu;

}

}

public interface ComputerAbstractFactory {

public Computer createComputer();

}

public class PCFactory implements ComputerAbstractFactory {

private String ram;

private String hdd;

private String cpu;

public PCFactory(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public Computer createComputer() {

return new PC(ram,hdd,cpu);

}

}

public class ServerFactory implements ComputerAbstractFactory {

private String ram;

private String hdd;

private String cpu;

public ServerFactory(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public Computer createComputer() {

return new Server(ram,hdd,cpu);

}

}

public class ComputerFactory {

public static Computer getComputer(ComputerAbstractFactory factory){

return factory.createComputer();

}

}

**Adapter**

* Este folosit pentru a permite interfetelor ce nu ua legtatura sa poata lucra impreuna. Deci, adapter converteste interfata unei clase in interfata pe care clientul o asteapta. Asa, clasele pot lucra impreuna chiar daca au interfete incompatibile
* Este folosit pentru a conecta ca si cum 2 interfete
* De ex, bateria telefonului are nevoie de 3V, dar priza are 120V, deci folosim un incarcator, care este acest adapter ce conecteaza bateria cu priza
* De ex, avem o clasa Bird cu metoda fly() si makeSound(). Avem si o clasa ToyDuck cu metoda squeak(). Vrem sa folosim clasa Bird, dar asteptam comportamentul la ToyDuck. Dar, ele implementeaza interfete difertie, deci nu le putem folosi.

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();

}

}

Deci, am folosi un Bird care sa se comporte ca un ToyDuckAnume conexiunea a fost facuta de BirdAdapter, care implementeaza ToyDuck, dar foloseste in interior Bird, insa in metoda squeak(), ce apartine lui ToyDuck

**Strategy**

* Se bazeaza pe aceea ca cream mai multi algoritmi si lasam clientul sa il aleaga pe cel de care are nevoie si sa il trimita ca parametru
* Un exemplu practic este Collections.sort(Comparator) care ia un comparator ca argument, pe baza la care se va face sortarea.

package com.journaldev.design.strategy;

public interface PaymentStrategy {

public void pay(int amount);

}

public class CreditCardStrategy implements PaymentStrategy {

private String name;

private String cardNumber;

private String cvv;

private String dateOfExpiry;

public CreditCardStrategy(String nm, String ccNum, String cvv, String expiryDate){

this.name=nm;

this.cardNumber=ccNum;

this.cvv=cvv;

this.dateOfExpiry=expiryDate;

}

@Override

public void pay(int amount) {

System.out.println(amount +" paid with credit/debit card");

}

}

public class PaypalStrategy implements PaymentStrategy {

private String emailId;

private String password;

public PaypalStrategy(String email, String pwd){

this.emailId=email;

this.password=pwd;

}

@Override

public void pay(int amount) {

System.out.println(amount + " paid using Paypal.");

}

}

public class Item {

private String upcCode;

private int price;

public Item(String upc, int cost){

this.upcCode=upc;

this.price=cost;

}

public String getUpcCode() {

return upcCode;

}

public int getPrice() {

return price;

}

}

public class ShoppingCart {

//List of items

List<Item> items;

public ShoppingCart(){

this.items=new ArrayList<Item>();

}

public void addItem(Item item){

this.items.add(item);

}

public void removeItem(Item item){

this.items.remove(item);

}

public int calculateTotal(){

int sum = 0;

for(Item item : items){

sum += item.getPrice();

}

return sum;

}

public void pay(PaymentStrategy paymentMethod){

int amount = calculateTotal();

paymentMethod.pay(amount);

}

}

public static void main(String[] args) {

ShoppingCart cart = new ShoppingCart();

Item item1 = new Item("1234",10);

Item item2 = new Item("5678",40);

cart.addItem(item1);

cart.addItem(item2);

//pay by paypal

cart.pay(new PaypalStrategy("myemail@example.com", "mypwd"));

//pay by credit card

cart.pay(new CreditCardStrategy("Pankaj Kumar", "1234567890123456", "786", "12/15"));

}

**Template Method**

* **Template method** defineste pasii de a executa un algoritm si ofera metode deja implementate ce vor fi folosite e orice sublasa si metode ce vor trebui suprascrise inca
* De ex, pentru a crea o casa trebuie urmat algoritmul:

1. cream fundamentul

2. cream pilonii

3. cream peretii

4. punem ferestrele

* Acum, avem nevoie de o metoda ce sa asigure ca aceste operatii se executa anume in ordinea potrivita. **Anume metoda asta se numeste template method**
* Template method, deobicei, e final, ca sa nu poata fi suprascrisa de subclase

public abstract class HouseTemplate {

//template method, final so subclasses can't override

public final void buildHouse(){

buildFoundation();

buildPillars();

buildWalls();

buildWindows();

System.out.println("House is built.");

}

//default implementation

private void buildWindows() {

System.out.println("Building Glass Windows");

}

//methods to be implemented by subclasses

public abstract void buildWalls();

public abstract void buildPillars();

private void buildFoundation() {

System.out.println("Building foundation with cement,iron rods and sand");

}

}

public class WoodenHouse extends HouseTemplate {

@Override

public void buildWalls() {

System.out.println("Building Wooden Walls");

}

@Override

public void buildPillars() {

System.out.println("Building Pillars with Wood coating");

}

}

public class GlassHouse extends HouseTemplate {

@Override

public void buildWalls() {

System.out.println("Building Glass Walls");

}

@Override

public void buildPillars() {

System.out.println("Building Pillars with glass coating");

}

}

public static void main(String[] args) {

HouseTemplate houseType = new WoodenHouse();

//using template method

houseType.buildHouse();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*");

houseType = new GlassHouse();

houseType.buildHouse();

}

**Builder**

* Builder e bun pentru a facilita crearea obiectelor ce au multe atribute
* Factory si abstract FACTORY PATTERN au probleme, si anume ca cand obiectele de creat au multi parametri, trebuie sa trimitem multe argumente, si nu e deloc comod.
* Builder ofera posibilitatea de a crea obiectul folosind metode, unde setam parametrii, si ofera o metoda finala ce returneaza obiectul final
* Cerinte:

1. Cream o nested static class si apoi copiem toate argumentele de la outer class la Builder class. Builder class are numele Nume+Builder

2. Builder class trebuie sa aiba un constructor default cu toate atributele necesare ce sunt obligatorii, caci pot fi si atribute nu neaparat de setat

3. Builder class trebuie sa aiba metoda de a seta parametrii si sa returneze acelasi Builder object

4. Oferim o metoda build() in builder class care sa returneaza Object creat. Trebuie sa avem un constructor privat ce ia Builder ca argument

public class Computer {

//required parameters

private String HDD;

private String RAM;

//optional parameters

private boolean isGraphicsCardEnabled;

private boolean isBluetoothEnabled;

public String getHDD() {

return HDD;

}

public String getRAM() {

return RAM;

}

public boolean isGraphicsCardEnabled() {

return isGraphicsCardEnabled;

}

public boolean isBluetoothEnabled() {

return isBluetoothEnabled;

}

private Computer(ComputerBuilder builder) {

this.HDD=builder.HDD;

this.RAM=builder.RAM;

this.isGraphicsCardEnabled=builder.isGraphicsCardEnabled;

this.isBluetoothEnabled=builder.isBluetoothEnabled;

}

//Builder Class

public static class ComputerBuilder{

// required parameters

private String HDD;

private String RAM;

// optional parameters

private boolean isGraphicsCardEnabled;

private boolean isBluetoothEnabled;

public ComputerBuilder(String hdd, String ram){

this.HDD=hdd;

this.RAM=ram;

}

public ComputerBuilder setGraphicsCardEnabled(boolean isGraphicsCardEnabled) {

this.isGraphicsCardEnabled = isGraphicsCardEnabled;

return this;

}

public ComputerBuilder setBluetoothEnabled(boolean isBluetoothEnabled) {

this.isBluetoothEnabled = isBluetoothEnabled;

return this;

}

public Computer build(){

return new Computer(this);

}

}

}