#شعبوليات

Design Pattern

يلا نشعبولي العملية . . .

Created By

SE/ Ahmed Mabrouk

Reach me via Linkedin

يعني ايه Design Pattern ؟

هي عبارة عن طرق لكتابة الكود. والطرق دي بتحللك مشاكل شائعه بتتكرر مع الـ programmers. وبتخلي الكود بتاعك منظم اكثر ومفهوم اكثر وبيشتغل بكفاءة اكبر.

ليه مهم تبقا عارف Design pattern ؟

1_ عشان الكود بتاعك يبقا قابل للتعديل والتوسع بسهولة

2_ لو واجهتك مشكلة معينه تبقا عارف تستخدملها انهي Design Pattern.

3_ واجهتك مشكلة معينه وخدت رأي الـ leader بتاعك وقالك استخدم الـ Design pattern الفلاني مينفعش تبقا مش عارف ساعتها الـ Design pattern الفلاني دا.

ايه هي أنواع الـ Design pattern بقا دي ؟

1_ الـ Creational Patterns ودا متعلق بإنشاء الـ Objects.

2_ الـ Structural Patterns ودا بيركز على ترتيب الـ Objects وترتيب العلاقات بينهم بشكل يجعل النظام أكثر مرونة و قابلاً للتوسعة.

3_ الـ Behavioral Patterns ودا بيركز على طريقة تفاعل الـ Objects مع بعضها وكيفية توزيع المسؤوليات بينهم.

يلا بقا نتقاشهم واحد واحد.

Creational Design Pattern

Singleton

اول حاجه عندي ف الـ Creational Design Pattern وهو

الـ Singleton ودا اتعمل عشان يضمنلي ان الـ Class بتاعي هيتاخد منه Object واحد فقط لا غير.

طب ليه يبقا عندي Class هيتكريت منه Object واحد فقط ؟

لو انت مثلا الـ Application بتاعك بيتعامل مع الـ database هل انت محتاج كذا Object من الـ Connection class ؟ ولا هو Object واحد يتعامل مع الـ database كفايه ؟ اظن كده وصلت.

وع سبيل الامثله بقا

- عندك تطبيق بيتصل بشبكة معينه وعايز تضمن انه مش هيفتح اكتر من اتصال واحد.
- اعدادات الـ app بتاعك كفايه object واحد يتعامل مع الاعدادات عندك عشان تضمن ان كل الكود هيستخدم نفس الاعدادات.

Creational Design Pattern

Singleton

ازاي اطبق الـ Singleton كـ code ؟

معايا مثلا Database connection class وزي ما اتكلمنا فوق انا الـ class دا مش محتاج اكريت منه غير object واحد فقط لا غير هو اللي هيتعامل مع الـ connections.

١- أولا هكريت انا object من الـ class جوا نفسي الـ class.

٢- ثاني حاجه لازم الـ Constractor بتاعي يبقا private عشان امنع أي حد من برا الـ class انه يعمل منه object.

٣- ثالث حاجه هعمل public method هتشوف لو اول مره تتنده اذا هتكريت instance من الـ class انما لو الـ object موجود مش هكريت حاجه وهرجلعه الـ obeject الموجود.

```java
public class DatabaseConnection {

    private static DatabaseConnection instance;

    private DatabaseConnection() {
        System.out.println("Connecting to database...");
    }

    public static DatabaseConnection getInstance() {
        if (instance == null) {
            instance = new DatabaseConnection();
        }
        return instance;
    }
}
```

Creational Design Pattern

Singleton

وطا اروح الـ main مهما كريت Objects من الـ database connection class هيرجعلي نفس الـ obejct ومش هيكريت أي objects جديده وبالتالي ضمنت ان الـ class بتاعي مش هيتكريت منه الا object واحد فقط.

```java
public class Main {
    public static void main(String[] args) {
        DatabaseConnection connection1 = DatabaseConnection.getInstance();
        DatabaseConnection connection2 = DatabaseConnection.getInstance();

        System.out.println(connection1 == connection2);  // هيرجع true
    }
}

```

وهي دي كل حكايه الـ Singleton

Creational Design Pattern

Builder

ثاني Creational Design Pattern معانا وهو الـBuilder Design Pattern

أي هو بقا الـ Builder Design Pateern ؟

لو انت عايز تبني بيت والبيت دي فيه حاجات كتير المفروض تتحدد زي عدد الغرف + اللون + في حديقة ولا لا + عدد الطوابق + المساحه الخ.

ف انت بتروح لشركة البناء وتقولهم عاوز ابني بيت وبعد كدا تقوله المواصفات واحده واحده زي عدد الغرف كدا وبعدين تقوله عدد الطوابق كدا وبعدين اللون كدا وهكذا.

هي دي بقا فكره الـ Builder design pattern.

- بدل ما تـ Create الـ object وترص كل حاجه ف الـ constractor مره واحده، لا انت بالـ Builder بتباصي الخصائص واحده واحده.
- ودي بتبقا مفيده معايا ف الـ Objects المعقده اللي ليها attributes كتيره.

ف الـ Builder عنده مزايا حلوه وهما

١- بيخليني استغني عن الـ constractor overload، مش محتاج اعمل كدا constractor كل واحد بياخد attributes تختلفه، لا انا بالـ builder هقدر اباصي الـ attributes اللي انا عايزها.

٢- وبالتالي واكود بتاعي واضح ومنظم اكتر.

Creational Design Pattern

Builder

ازاي بقا انفذ ال Builder ك Code ؟

هنفترض مثلا ان معاك Class car عادي

```
class Car {
    private String engine;
    private String color;
    private int wheels;
    private boolean sunroof;

    // Constructor بيكون private عشان ما نقدرش نعمل object من غير builder.
    private Car(CarBuilder builder) {
        this.engine = builder.engine;
        this.color = builder.color;
        this.wheels = builder.wheels;
        this.sunroof = builder.sunroof;
    }

    // لو عايزين نستخدم البيانات في أماكن تانية getters هنا ممكن نضيف
    public String getEngine() {

    public String getColor() {

    public int getWheels() {

    public boolean hasSunroof() {
```

لازم تكون خدت بالك ان الـ Constractor هنا private عشان محدش يقدر يكريت object من الـ class الا باستخدام الـ Builder .

طب هو فين ال Builder ؟

جايلك ف الكلام اهووو

Creational Design Pattern

Builder

وادي ال Builder class وبعمله جوا الـ Class بناع الـ Car.

```
32      // 2. نعمل class builder:
33      public static class CarBuilder {
34          private String engine;
35          private String color;
36          private int wheels = 4;  // Default value
37          private boolean sunroof = false;
38
39          // Builder method عشان نحدد كل خصيصة للسيارة
40          public CarBuilder setEngine(String engine) {
41              this.engine = engine;
42              return this;
43          }
44
45          public CarBuilder setColor(String color) {
46              this.color = color;
47              return this;
48          }
49
50          public CarBuilder setWheels(int wheels) {
51              this.wheels = wheels;
52              return this;
53          }
54
55          public CarBuilder setSunroof(boolean sunroof) {
56              this.sunroof = sunroof;
57              return this;
58          }
59
60          // طريقة عشان نبني السيارة بالنهاية
61          public Car build() {
62              return new Car(this);
63          }
64      }
65  }
```

وهو دا الـ Class اللي هستخدمه وانا ب Build الـ Object بتاعي وبكريته.

لاحظ إنه Static عشان اقدر اناديه باستخدام اسم الـ class علطول.

Creational Design Pattern

Builder

وهنا الـ Main بناعنا

```java
public class Main {
    public static void main(String[] args) {
        // بنبني السيارة باستخدام builder:
        Car myCar = new Car.CarBuilder()
                        .setEngine("V8")
                        .setColor("Red")
                        .setSunroof(true)
                        .build();

        System.out.println("Engine: " + myCar.getEngine());
        System.out.println("Color: " + myCar.getColor());
        System.out.println("Wheels: " + myCar.getWheels());
        System.out.println("Sunroof: " + myCar.hasSunroof());
    }
}
```

واضح اوي قدامي الفرق بين الطريقة دي وبين طريقه اني ابعت الـ paramaters كلها ف الـ constarctor.

ف الـ Builder عنده مزايا حلوه وهما

١- بيخليني استغني عن الـ constractor overload، مش محتاج اعمل كذا constractor كل واحد بياخد attributes تختلفه، لا انا بالـ builder هقدر اباصي الـ attributes اللي انا عايزها.

٢- وبالتالي واكود بناعي واضح ومنظم اكتر.

ودي بتبقا مفيده معايا ف الـ Objects المعقده اللي ليها attributes كتيره.

فبدل م ارص ف الـ constractor واتوه وابدل اتنين attribute ومش عارف ايه، لا انا بالـ Builder عارف انا بـ set value ل ايه وبباصي للـ values الـ attributes اللي انا عايزها بالترتيب اللي انا عايزه.

Creational Design Pattern

Prototype

اكيد ف مره وانت بتنسخ Object في Object تاني استخدمت method اسمها Clone

الف مبروك انت كدا استخدمت Prototype Design Pattern

لان الـ Clone يعتبر Prototype Design Pattern

طب ليه عملنا الـ clone method ؟ مكنا عملنا Object جديد ومشينا ع الـ attributes اللي ف الـ Object الاصلي ونسخناها ف الـ Object الجديد وريحنا دماغنا.

طب ولو كان عندنا private attributes ؟ مليش access عليها فمش هقدر انسخها ولذلك عملنا الـ clone.

بغض النظر انه بيحللي مشكله الـ private attributes هو

١- بدل ما اعمل Object من اول وجديد، انا بنسخ Object موجود معايا واعدل فيه وبالتالي وفرت وقت ومجهود.

٢- بدل ما اقعد الف ع الـ attributes وانسخ فيهم انا بستخدم clone وخلاص وبالتالي الكود ابسط وقللت التركارت.

بيستخدم بكثرة فالـ Game development ما مثلا بناخد character كوبي بيست عشره عشرين تلاتين مره ع حسب عايز تعمل كام character وبعدين تعدل ع كل واحد الصفات الخاصه بيه بدل ما تكريت كل شخصيه from scratch

Creational Design Pattern

Prototype

عامل ازاي بقا هو كـ Code ؟

أولا عندي Interface جواه الـ clone method

```java
package design_pattern;

public interface Prototype {
    Prototype clone();
}
```

ثاني حاجه Character class عادي جدا بيـ implement prototype وبيـ override الـ clone method وبنكتب الـbody بناعها بالشكل دا

```java
package design_pattern;

public class Character implements Prototype{
    private String name;
    private int health;
    private int power;

    // Constractor
    public Character(String name, int health, int power) {
        this.name = name;
        this.health = health;
        this.power = power;
    }

    // copy method
    @Override
    public Prototype clone() {
        return new Character(this.name, this.health, this.power);
    }

    @Override
    public String toString() {
        return "Character: " + name + ", Health: " + health + ", Power: " + power;
    }
}
```

Creational Design Pattern

Prototype

وادي الـ Main اهي بتوضح ازاي بنسخ الـ Object باستخدام الـ clone method
وازاي بعدل ف الـ Copied object.

```java
package design_pattern;

public class Main {
    public static void main(String[] args) {
        // Original character
        Character originalCharacter = new Character("Warrior", 100, 50);
        System.out.println("Original Character: " + originalCharacter);

        // Copied character
        Character clonedCharacter = (Character) originalCharacter.clone();
        System.out.println("Cloned Character: " + clonedCharacter);

        // Modify copied character
        clonedCharacter = new Character("Mage", 80, 60);
        System.out.println("Modified Cloned Character: " + clonedCharacter);
    }
}
```

ودي كل حكايه الـ Protoype

Creational Design Pattern

Factory Method

افترض انك بتعمل App لإدارة مركبات النقل زي الـ cars و الـ trucks والـ motorcycles. و كل نوع منهم ليه الخصائص والسلوكيات بتاعته.

و الـ App بتاعك بقا بيعتمد ف إنشاء الـ Objects علي Input من الـ User.

If input car create car object

if input truck create truck object

if input motors create motors object

اي المشكلة ؟

كل مرة عايز تضيف نوع جديد من المركبات، هتحتاج تعدل في الكود في كل مكان بيتم فيه إنشاء الكائنات، وده هيؤدي لمشاكل وهي

١- كررت الكود بتاعي وعمال اكتب ف سطور creation.

٢- عشان اضيف مركبة جديده هحتاج اروح لكل مكان بـ create objects فيه واضيف سطر creation للمركبة الجديده.

٣- احتمال انسي اعدل ف حته وبالتالي احتماليه الخطأ كبرت.

وهنا بقا جت الـ Factory method عشان تنقذنا.

وهي بتخليك تفصل حته إنشاء الـ Objects ف class لوحدها وكل اللي انت بتعمله انك بدل ما بتكتب كل اللي فوق بتكتب

createObjectOf(input) وشكرا ع كدا

وع حسب م الـ input جالي عايز اي هي بتنشأ الـ Object.

Creational Design Pattern

Factory Method

إزاي بقا هنفذ دا كـ Code ؟

اول حاجه معايا Vehicle interface عادي جدا.

```java
public interface Vehicle {
    void start();
    void stop();
}
```

ومعايا Classes car and truck عادي جدا بردو.

```java
public class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car is starting.");
    }

    @Override
    public void stop() {
        System.out.println("Car is stopping.");
    }
}
```

```java
public class Truck implements Vehicle {
    @Override
    public void start() {
        System.out.println("Truck is starting.");
    }

    @Override
    public void stop() {
        System.out.println("Truck is stopping.");
    }
}
```

Creational Design Pattern

Factory Method

بعد كدا عندي Class VehicleFactory جواه Static method getVehicle بتاخد مني String input وعلي حسب الـ input هترجعلي الـ Class اللي انا عايزه.

```java
public class VehicleFactory {

    public static Vehicle getVehicle(String vehicleType) {
        if (vehicleType == null) {
            return null;
        }
        if (vehicleType.equalsIgnoreCase("CAR")) {
            return new Car();
        } else if (vehicleType.equalsIgnoreCase("TRUCK")) {
            return new Truck();
        }
        return null;
    }
}
```

بعد كدا الـ Main عندي بالشكل دا

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        String typeOfCar = sc.nextLine(); // take the type of car from user
        Vehicle specificVehicle = VehicleFactory.getVehicle(typeOfCar);
        specificVehicle.start();
        specificVehicle.stop();

        /*
        if (typeOfCar == "car") {
            Car car = new Car();
            specificVehicle.start();
            specificVehicle.stop();
        }
        else if (typeOfCar == "truck") {
            Truck truck = new Truck();
            specificVehicle.start();
            specificVehicle.stop();
        }
        */
    }
}
```

بدل ما اكتب كل اللي انا عامله كومنت دا. لأ انا عملت Factory هيريحني من كل دا ولو بعد كدا ضفت Class جديد مش هحتاج اجي اعدل هنا أي حاجه.

Creational Design Pattern
Abstract Factory Interface

لو عندي مصنع بيصنّع انواع تختلفه من السيارات وليكن مثلا سيارات رياضيه و سيارات عائلية. و client جاي يشتري من عندي سيارة.

الطبيعي انا كمصنع اسأل الـ client عن ايه ؟

صح، عايز سياره عائلية ولا رياضية.

انما هل طبيعي اسأله عايز محرك رياضي ولا عائلي؟ عايز إطارات رياضية ولا عائلية؟ عايز كراسي رياضية ولا عائلية؟

اكيد لا الـ client ملهوش علاقه بكل دا وانا كمصنع المفروض مسألهوش عن التفاصيل دي.

وهي دي بقا مهمة الـ Abstract method interface.

طب ازاي بقا ننفذه كـ Code ؟

أولا عندنا Interface Engine and Tires

```
1
2 public interface Engine {
3     void start();
4 }
```

```
1
2 public interface Tires {
3     void roll();
4 }
5
```

Creational Design Pattern

Abstract Factory Interface

ثاني خاجه عندي بيعملوا Class SportsEngine and SportsTires
implementation لـ Interfaces اللي فوق.

```java

public class SportsEngine implements Engine{

    @Override
    public void start() {
        System.out.println("Starting sports engine with high speed!");
    }
}
```

```java
public class SportsTires implements Tires {

    @Override
    public void roll() {
        System.out.println("Sports tires are rolling fast!");
    }
}
```

وعندي زيهم للـ Family Cars

```java
public class FamilyEngine implements Engine{

    @Override
    public void start() {
        System.out.println("Starting family engine with efficiency!");
    }
}
```

```java

public class FamilyTires implements Tires {

    @Override
    public void roll() {
        System.out.println("Family tires are rolling smoothly!");
    }
}
```

Creational Design Pattern

Abstract Factory Interface

بعد كدا عندي Car Factory Interface الـ Sports والـ Family هيعملولهم Impl.

```java
//Abstract Factory Interface
interface CarFactory {
    Engine createEngine();
    Tires createTires();
}
```

```java

public class SportsCarFactory implements CarFactory {

    @Override
    public Engine createEngine() {
        return new SportsEngine();
    }

    @Override
    public Tires createTires() {
        return new SportsTires();
    }
}
```

```java
public class FamilyCarFactory implements CarFactory{

    @Override
    public Engine createEngine() {
        return new FamilyEngine();
    }

    @Override
    public Tires createTires() {
        return new FamilyTires();
    }
}
```

كل Class هيرجع الـ Objects بنوع الـ Engine والـ Tires الخاصين بيه.

Creational Design Pattern

Abstract Factory Interface

نيجي بقا للـ Main اللي هيبان فيها مهمة الـ Abstract Factory

```java
public class Main {
    public static void main(String[] args) {
        CarFactory sportsFactory = new SportsCarFactory();
        Engine sportsEngine = sportsFactory.createEngine();
        Tires sportsTires = sportsFactory.createTires();

        sportsEngine.start(); // output -> Starting sports engine with high speed!
        sportsTires.roll(); // output -> Sports tires are rolling fast!

        System.out.println();

        CarFactory familyFactory = new FamilyCarFactory();
        Engine familyEngine = familyFactory.createEngine();
        Tires familyTires = familyFactory.createTires();

        familyEngine.start(); // output -> Starting family engine with efficiency!
        familyTires.roll(); // output -> Family tires are rolling smoothly!
    }
}
```

زي محنا شايفين انا كريت Object من الـ SportsCarFactory

وبعدين زي ما قولنا فوق خالص انا مش مضطر أقوله اعملي Sport Engine ولا مضطر أقوله اعملي Sports Tiers.

انا كل اللي بقولهوله CreateEngine وهو بقا يبروح يعمل الـ Sport Engine لوحده بقوله CraeteTiers وهو يبروح يعمل الـ Sports Tiers لوحده.

ونفس الكلام هعمله تحت لما اكريت Object من الـ FamilyCarFactory.

ودي كل حكاية الـ Abstract Method Interface

وكدا يبقا خلصنا الـ Creational Design Pattern

Structural Design Pattern

Adapter

كنا اتكلمنا عن الـ Structural Design Pattern وقولنا انه مهتم بتنظيم الـ Objects وتكوين العلاقات بينهم وأول Design Pattern معانا فيه هو الـ Adapter.

ايه هو بقا الـ Adapter Design Pattern؟

شاحن اللاب بتاعك ثلاثي المخارج واحنا عندنا اي كوبس بيبقا ثنائي المدخل ولذلك بنستخدم ايه ؟

بنستخدم Adapter يحولك الثلاثي لثنائي عشان يعيشوا ويتعايشوا مع بعض من غير ما نغير جزيا ف الكوبس ولا ف الفيشه.

بالظظبط دا اللي بيعمله الـ Adapter design pattern.

لو الـ app بتاعك عنده خدمه ارسال messages والـ method اسمها sendSMS، بعدين قررت تستخدم خدمه جديده لإرسال الـ messages بـ method اسمها SendTextMessages.

- طبعا اول ما تقرر تشتغل بالخدمه الجديده وتلغي الخدمه القديمه الكود بتاعك كله هيضرب ويقولك الـ method القديمه مبقتش اعرفها ومش موجوده عندي.

الحل هنا ايه ؟

هتلف ع الكود كله وتقعد تعدل وتغير وتشيل وتحط ؟! اكيد لا.

الحل هنا بقا هو الـ Adapter design pattern انه هيحولك طريقه تشغيل الكود الجديد للطريقه اللي كان شغال بيها الكود القديم وكدا انت ضربت عصفورين بحجر.

✓ فعلت الخدمه الجديده.

✓ مروحتش عدلت ف الكود.

Structural Design Pattern

Adapter

إزاي الـAdapter بينفذ كـ Code ؟

أولا معايا Class OldSMSService ودا الطريقه القديمه اللي كنت ببعت بيها الرسايل.

```
1 class OldSMSService {
2     public void sendSMS(String phoneNumber, String message) {
3         System.out.println("Sending SMS via Old SMS Service to: "
4                 + phoneNumber + " Message: " + message);
5     }
6 }
```

ومعايا Class NewSMSService ودا الطريقه الجديده اللي هبعت بيها الرسايل.

```
1 class NewSMSService {
2     public void sendTextMessage(String mobile, String text) {
3         System.out.println("Sending Text via New SMS Service to: "
4                 + mobile + " Message: " + text);
5     }
6 }
```

لازم ناخد بالنا ان الـ Method اسمها أختلف في الكلاسين والكود القديم بتاعنا شغال ع اسم الـ Method اللي في الكود القديم

وبالتالي لو جيت اشغل الكود الجديد والغي الكود القديم هيضرب معايا ويقولي ان الـ Method اللي اسمها SendSMS مبقتش متعرفه عندي ومعرفهاش.

إيه الحل هنا ؟

بدل ما امشي ع الكود كله واعدل فيه. لأ انا هستخدم Adapter.

Structural Design Pattern

Adapter

ودا الـ Adapter class اللي معايا

```java
class SMSAdapter {
    private NewSMSService newSMSService;

    public SMSAdapter(NewSMSService newSMSService) {
        this.newSMSService = newSMSService;
    }

    // هنا بنحول الطريقة القديمة service الجديدة للطريقة
    public void sendSMS(String phoneNumber, String message) {
        newSMSService.sendTextMessage(phoneNumber, message); // بنستخدم الطريقة الجديدة هنا
    }
}
```

وودا الـ Main بتاعي

```java
public class Main {
    public static void main(String[] args) {
        // بنشئ كائن من الخدمة الجديدة
        NewSMSService newSMSService = new NewSMSService();

        // لو حاولنا استخدام الخدمة الجديدة بشكل مباشر في الكود القديم ده هيعمل مشكلة
        // newSMSService.sendSMS("0123456789", "Hello, this is an SMS!");
        // هنا هنواجه مشكله NewSMSService في sendSMS لأنه مفيش طريقة

        // اللي هيربط الخدمة القديمة بالجديدة Adapter دلوقتي بننشئ الـ
        SMSAdapter smsAdapter = new SMSAdapter(newSMSService);

        // هنستخدم الـ Adapter علشان نبعت رسالة بالطريقة القديمة
        smsAdapter.sendSMS("0123456789", "Hello, this is an SMS!");

        // دلوقتي بنقدر نبعت رسالة بنفس الكود القديم ولكن بنستخدم الخدمة الجديدة!
    }
}
```

وزي ماهو موضح كدا انا لو كنت استخدمت اسم الـ Method القديمة كان هيضرب معايا لان الكود الجديد اسم الـ Method فيه متغير وبالتالي

١- لإما امشي ع الكود كله واقعد اعدل فيه ودا هياخد مني وقت ومجهود كبير واحتمالية الأخطاء هتبقا كبيرة.

٢- اعمل Adapter Class واريح دماغي وهيبقا الكود قابل للتوسع وكفء اكتر.

ودي كل حكاية الـ Adapter.

Structural Design Pattern

Bridge

لو عايزين نبني برنامج لإدارة شكل الـ أشكال هندسية زي الدائرة و المستطيل، وكل واحد منهم ممكن يكون له لون تختلف زي الأحمر والازرق. هنعمل ايه ؟

هنقولي هروح اعمل CircleRed Class و CircleBlue Class و RectangleRed Class و RectangleBlue Class.

طب لو حبيت اضيف لون جديد ؟ هتروح تعمل 2 Classes لكل شكل باللون الجديد.

طب لو حبيت بعدها اضيف شكل جديد ؟ هتروح تعمل 3 Classes لكل لون من الألوان اللي عندي.

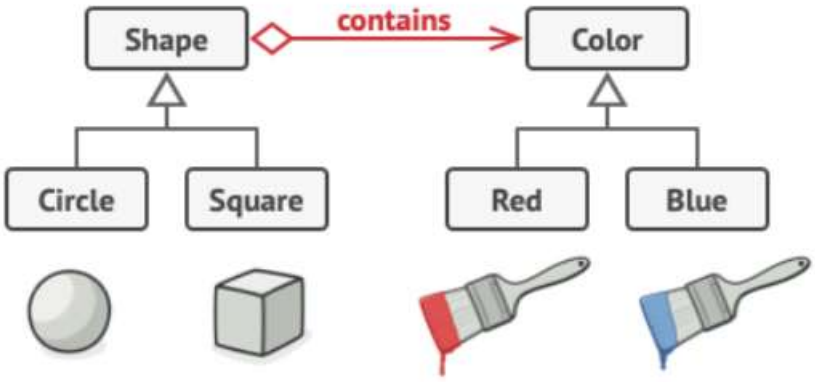
طب يزمتك ي اخي انت شايف دا كلام ؟! عشان اضيف لون ولا شكل اروح اعمل تلت اربع كلاسات وكل شويه بيزيدوا معايا اكتر من اطره اللي قبلهم.

هنا بقا يجي دور الـ Bridge Design Pattern

وهو بيساعدك في فصل الكود عن بعضه بحيث نقسمه لكونين منفصلين، بحيث كل واحد منهم يقدر يتطور بشكل مستقل عن التاني.

يعني نقدر نضيف شكل جديد بدون ما نأثر علي الألوان ولا نفكر فيها

ونقدر نضيف شكل جديد بدون ما نأثر ولا نفكر في الاشكال.



Structural Design Pattern

Bridge

إزاي بيشتغل كـ Code ؟

أول حاجة هنعرّف Interfaces لتمثيل الـ Shape و الـ Color.

```java
interface Shape {
    void draw();  // العمليات المشتركة زي رسم الشكل
}
```

```java
interface Color {
    void applyColor();  // العمليات الخاصة باللون
}
```

بعدين بنعمل Classes للـ Shape زي الـ Circle و Rectangle وهعرف الـ Class الـ Color جواهم إن كل شكل منهم ليه لون زي ما قولنا فوق.

```java
class Circle implements Shape {
    private Color color;

    // Constructor بياخد اللون
    public Circle(Color color) {
        this.color = color;
    }

    @Override
    public void draw() {
        System.out.print("Drawing Circle with color: ");
        color.applyColor();  // هنطبق اللون الخاص بالشكل
    }
}
```

```java
class Rectangle implements Shape {
    private Color color;

    // Constructor بياخد اللون
    public Rectangle(Color color) {
        this.color = color;
    }

    @Override
    public void draw() {
        System.out.print("Drawing Rectangle with color: ");
        color.applyColor();  // هنطبق اللون الخاص بالشكل
    }
}
```

Structural Design Pattern

Bridge

بعد كده بنعمل Classes للـ Color زي الـ Red و Blue

```java
class Red implements Color {
    @Override
    public void applyColor() {
        System.out.println("Red");
    }
}
```

```java
class Blue implements Color {
    @Override
    public void applyColor() {
        System.out.println("Blue");
    }
}
```

وأخيرا الـ Main.

```java
public class Main {
    public static void main(String[] args) {
        Shape redCircle = new Circle(new Red());  // دائرة حمراء
        Shape blueRectangle = new Rectangle(new Blue());  // مستطيل أزرق

        redCircle.draw();  // هتطبع "Drawing Circle with color: Red"
        blueRectangle.draw();  // هتطبع "Drawing Rectangle with color: Blue"
    }
}
```

وزي ماهو واضح قدامنا الـ Color منفصل عن الـ Shapes وبدل ماكنت رايح اعمل اربع كلاسات اللي هما عباره عن لونين * شكلين وكل ما اجي اضيف لون او شكل هضطر اضيف اكثر من كلاس والموضوع هيكبر مني اوي.

لا انا عملت كلاسين لكل شكل وكلاسين لكل لون وبعد كدا مهما ضيفت الوان او اشكال مش هبقا مضطر غير اني اضيف Class واحد فقط.

ودي كل حكايه الـ Bridge Design Pattern

Structural Design Pattern

Facade

يعني ايه كلمة Facade دي أصلا ؟! يعني واجهة يا واجهة.

إيه هو بقا الـ Faade Design Pattern ؟

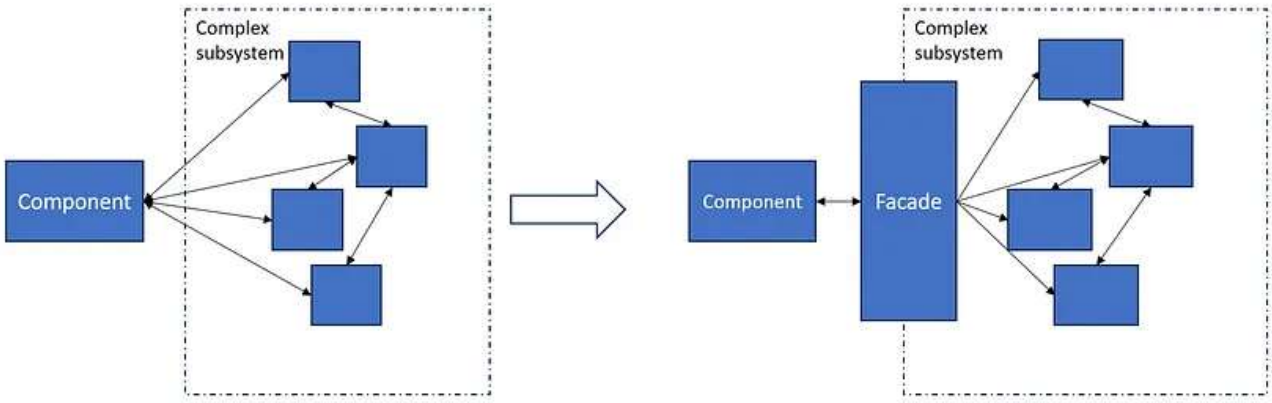
تصور معايا مثلاً عندك مجموعة من الأنظمة المعقدة في مشروع كبير، زي مثلاً نظام للحجز، نظام للدفع، ونظام للتوصيل.

وكل نظام جواه شويه تفاصيل وحكايات خاصه بيه عشان يشتغل.

هل الـ User لما يجي يطلب حاجه هيروح يتعامل مع نظام الحجز وبعدين يروح يتعامل مع نظام الدفع بعدين يروح يتعامل مع نظام التوصيل ؟ انت شايف دا منطقي ؟!

يجي هنا بقا دور الـ Facade وهو بيستخدم علشان يبسط استخدام الأنظمة المعقدة من خلال واجهة بسيطة واحدة.

يعني بدلاً ما تتعامل مع كل جزء في النظام بشكل منفصل، الـ Facade بيقدملك واجهة واحدة بسيطة تقدر من خلالها تتحكم في كل حاجة.



Structural Design Pattern

Facade

بينفذ ازاي بقا كـ Code الـ Facade دا ؟

عندنا عدة أنظمة زي PaymentSystem, OrderSystem، و DeliverySystem.

```java
class OrderSystem {
    public void placeOrder(String food) {
        System.out.println("Placing order for: " + food);
    }
}
```

```java
class PaymentSystem {
    public void makePayment(double amount) {
        System.out.println("Processing payment of: " + amount + " dollars.");
    }
}
```

```java
class DeliverySystem {
    public void deliverOrder(String address) {
        System.out.println("Delivering order to: " + address);
    }
}
```

بعدين عندي الـ Facade Interface.

```java
// واجهة الـ Facade
class RestaurantFacade {
    private OrderSystem orderSystem;
    private PaymentSystem paymentSystem;
    private DeliverySystem deliverySystem;

    public RestaurantFacade() {
        orderSystem = new OrderSystem();
        paymentSystem = new PaymentSystem();
        deliverySystem = new DeliverySystem();
    }

    public void placeOrderAndPay(String food, double amount, String address) {
        orderSystem.placeOrder(food);
        paymentSystem.makePayment(amount);
        deliverySystem.deliverOrder(address);
        System.out.println("Your order is placed successfully!");
    }
}
```

Structural Design Pattern

Facade

وبعدين عندي الـ Main.

```java
// استخدام الـ Facade
public class Main {
    public static void main(String[] args) {
        RestaurantFacade restaurant = new RestaurantFacade();
        restaurant.placeOrderAndPay("Pizza", 15.0, "123 Main St");
    }
}
```

وزي ما احنا شايفين كدا الـ Facade Class خلاني متعاملش مع كل نظام علي حدا لوحده.

لكن انا اتعاملت مع الـ Facade وهو راح خلصلي الموضوع كله.

إيه هي فوايده ؟

أولا تبسيط الاستخدام يعني بتخلي المستخدم يتعامل مع واجهة بسيطة بدل ما يتعامل مع عدة أنظمة معقدة.

ثانيا لو عندك أنظمة معقدة جداً، الـ Facade بيقلل الحاجة للتعامل مع كل نظام على حدة وبالتالي مفيش أي تعقيد في الـ Application بتاعي.

ودي كل حكايه الـFacade Design Pattern.

Structural Design Pattern

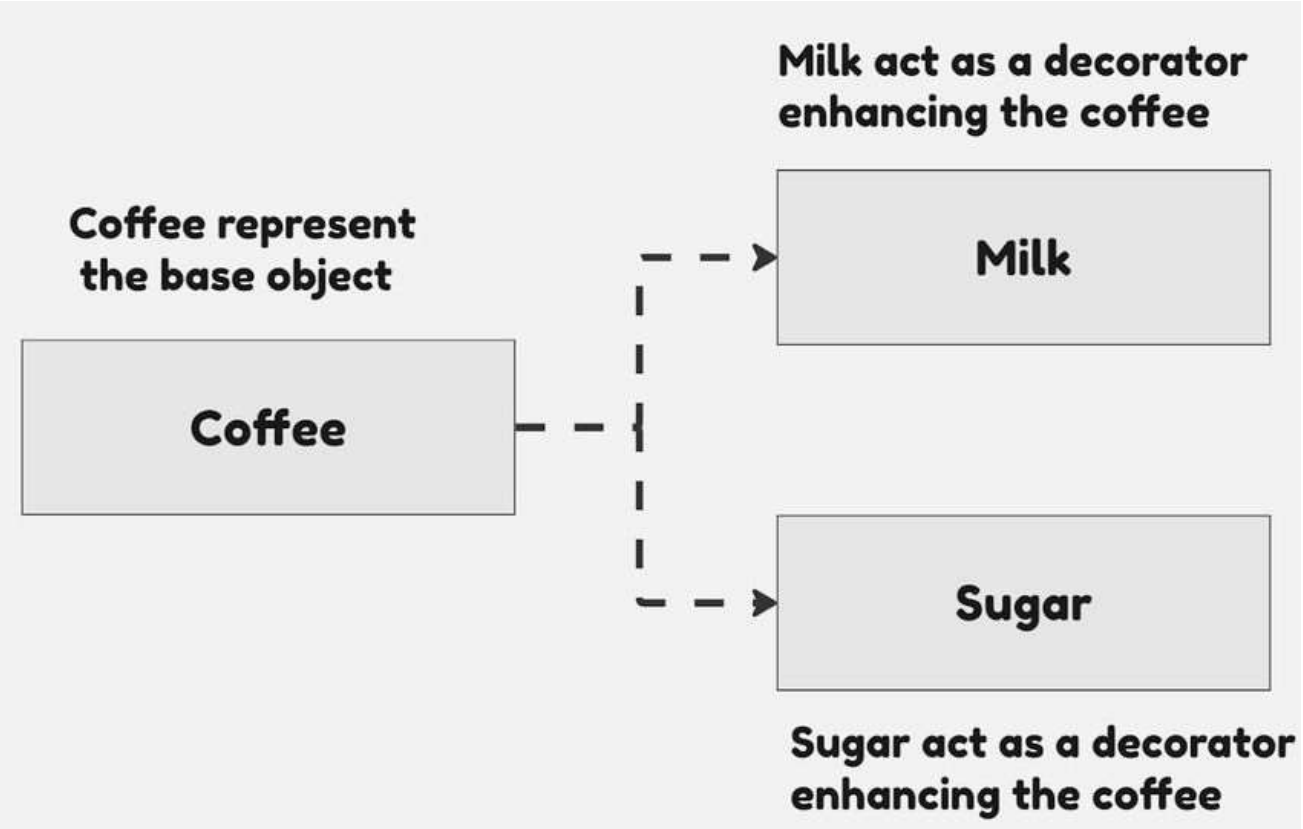
Decoractor

لو عندنا محل قهوة، وعايزين نضيف مكونات مختلفة للقهوة زي (حليب، سكر، إلخ) بدون ما نعدل في الكود الأصلي لصناعة القهوة. كل مكون جديد هنضيفه هيكون Decorator.

يعني الـ Decorator دا هيسمحلي اضيف مكونات مختلفة للقهوة (الحليب، السكر) بطريقة مرنة.

كل مكون إضافي بنضيفه هيكون ديكور (Decorator) للقهوة الأساسية، وده يعني أننا هنضيف خصائص جديدة بدون ما نعدل على الكود الأصلي للقهوة.

وهي دي مهمة الـ Decorator.



Structural Design Pattern

Decoractor

تعالا بقا نشوف ازاي هيتنفذ كـ Code ؟

أولا عندي Interface عادي.

```
interface Coffee {
    double cost();  // دالة لحساب تكلفة القهوة
}

```

بعدين القهوة الأساسية (SimpleCoffee): دي هي القهوة الأصلية أو البسيطة بدون أي إضافات. كل اللي هتعمله إنها هترجع التكلفة الأساسية للقهوة.

```
class SimpleCoffee implements Coffee {
    @Override
    public double cost() {
        return 5.0; // تكلفة القهوة العادية
    }
}
```

بعدين الديكور (CoffeeDecorator): وهو عبارة عن الـ(abstract class) اللي هتسمح لنا بإضافة مكونات إضافية للقهوة، زي الحليب أو السكر.

وخد بالك ان الـ CoffeeDecorator نفسه مش هيضيف أي مكونات، لكن كل Decorator آخر هيضيف مكون جديد زي الحليب أو السكر.

```
abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;  // الكائن اللي هنضيف ليه المكونات

    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;  // نحفظ الكائن اللي هنضيف ليه المكونات
    }

    @Override
    public double cost() {
        return decoratedCoffee.cost();  // هترجع تكلفة القهوة الأصلية، لكن هيتعدل عليها في الديكورات
    }
}
```

Structural Design Pattern

Decoractor

بعدين هنعرف الـ Classes التي تضيف المكونات الفعلية للقهوة. كل Class هيكون ديكور جديد يضيف سلوك تخصص. مثلا:

MilkDecorator: يضيف الحليب.

```
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);  // نفذ البناء الأساسي لاضافة القهوة الأساسية
    }

    @Override
    public double cost() {
        return decoratedCoffee.cost() + 1.5;  // نضيف تكلفة الحليب للقهوة
    }
}
```

SugarDecorator: يضيف السكر.

```
class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);  // نفذ البناء الأساسي لاضافة القهوة الأساسية
    }

    @Override
    public double cost() {
        return decoratedCoffee.cost() + 0.5;  // نضيف تكلفة السكر للقهوة
    }
}
```

Structural Design Pattern

Decoractor

بعد كدا عندي الـ Main.

```java
public class Main {
    public static void main(String[] args) {
        Coffee coffee = new SimpleCoffee();  // القهوة الأساسية

        // طباعة تكلفة القهوة الأساسية
        System.out.println("Basic Coffee cost: " + coffee.cost());
        // output-> Basic Coffee cost: 5.0

        // إضافة الحليب
        coffee = new MilkDecorator(coffee);
        System.out.println("Coffee with Milk cost: " + coffee.cost());
        // output-> Coffee with Milk cost: 6.5

        // إضافة السكر
        coffee = new SugarDecorator(coffee);
        System.out.println("Coffee with Milk and Sugar cost: " + coffee.cost());
        // output-> Coffee with Milk and Sugar cost: 7.0

    }
}
```

ودي الطريقة اللي بستخدم بيها الـ Decorator
بيبقا معايا Object من الـ Class الأساسي وباستخدام الـ Decorator اقدر اضيف أي خصائص انا عايزها للـ Object الأساسي اللي معايا.

ودي حكاية الـ Decorator Design Pattern.

✓ وها هنا تنتهي سهرتنا مع الـ Structural Design Pattern

Behavioral Design Pattern

Method templete

اتكلمنا عن الـ Behavioral وقولنا انه مهتم بطريقة تفاعل الـ Objects بينهم وبين بعض وازاي بيوزعوا المسؤوليات بينهم.

وأول نوع معانا هو الـ Method templete.

تخيل نفسك في مطبخ بتحضر أكلة معينة. الخطة العامة للعملية هي:

١. تحضير المكونات.

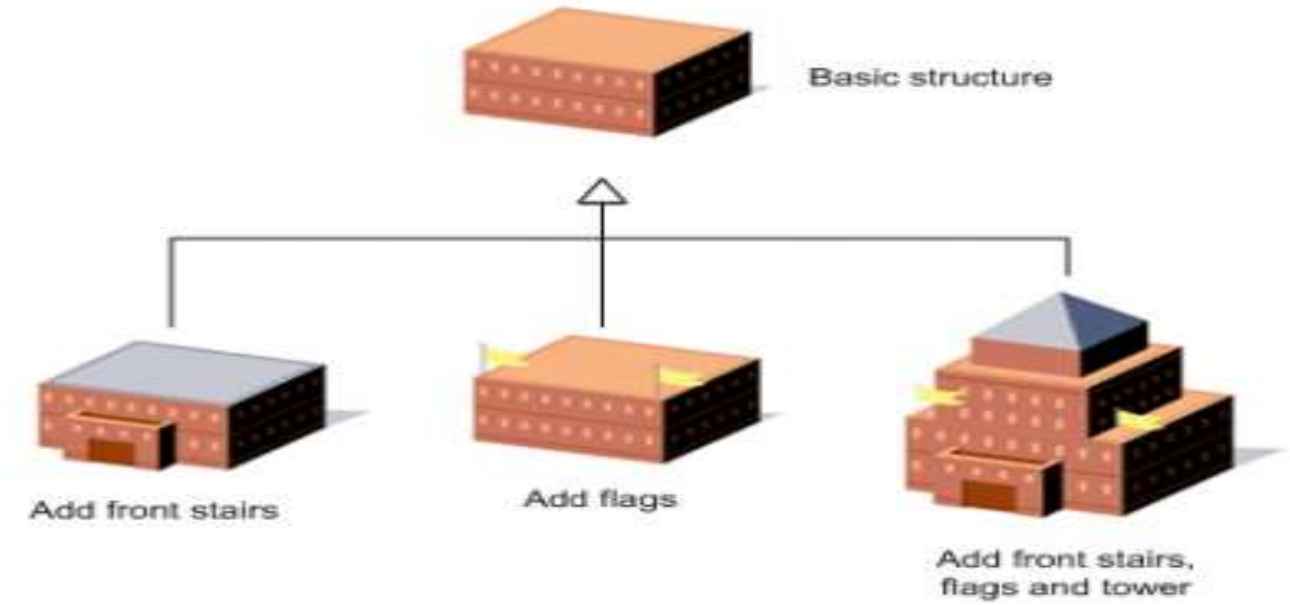
٢. طهي الطعام.

٣. تقديم الطعام.

طبعا تحضير المكونات وتقديم الطعام مبيتغيروش معايا من اكلة للثانية ، لكن التفاصيل زي نوع المكونات أو طريقة الطهي بتختلف من أكلة لأخرى.

فبدلاً من كتابة نفس الخطوات في كل مرة، بتستخدم طريقة ثابتة (template) لكتابة الأساس، وكل أكلة بتعدل التفاصيل الخاصة بيها.

والصوره دي هتوضحلنا الفكره اكتر



Behavioral Design Pattern

Method templete

أهلا بيك في فقرة إزاي الـ Method templete هينفذ كـ Code ؟

اول حاجه معايا Abstract class اللي هيبقا فيه الطريقه العامه اللي أي SubClass هيستخدمها.

```java
// الفئة الأساسية Template Class
abstract class CookingRecipe {

    // طريقة الـTemplate اللي بتحكم الخوارزمية العامة
    public final void prepareRecipe() {
        gatherIngredients();
        cook();
        serve();
    }

    // الخطوات اللي بيتم تنفيذها في جميع الوصفات
    private void gatherIngredients() {
        System.out.println("Gathering basic ingredients.");
    }

    // خطوة طبخ الطعام اللي الفئات الفرعية هتخصصها
    protected abstract void cook();

    // الخطوة الأخيرة، تقديم الطعام
    private void serve() {
        System.out.println("Serving the dish.");
    }
}
```

ليه عملناه Abstract ؟ عشان مينفعش يتاخد منه أي Object هو بيحتوي ع الطريقه العامه بتاخد طهي أي اكله انما الاكله نفسها ملهوش دعوه بيها.

ليه عملنا PrepareRecipe Final ؟ عشان مفيش أي SubClass يعدلي في الطريقه العامه. هي طريقه عامه هتمشي ع الكل من غير تعديل.

وزي محنا شايفين عندنا الـ Cook method هي Abstract عشان دي الطريقه اللي هتختلف من SubClass للتاني وبالتالي كل SubClass يعملها بالطريقه اللي تخصه.

Behavioral Design Pattern

Method templete

ثاني حاجه معانا الـ SubClasses.

```java
// فئة فرعية وصفة أكلة محددة
class PastaRecipe extends CookingRecipe {

    @Override
    protected void cook() {
        System.out.println("Boiling pasta and adding sauce.");
    }
}
```

```java
// فئة فرعية وصفة أكلة مختلفة
class SoupRecipe extends CookingRecipe {

    @Override
    protected void cook() {
        System.out.println("Boiling vegetables and adding broth.");
    }
}
```

وبعدين الـ Main.

```java
public class Main {
    public static void main(String[] args) {
        // طهي الباستا
        CookingRecipe pasta = new PastaRecipe();
        pasta.prepareRecipe();

        System.out.println();

        // طهي الحساء
        CookingRecipe soup = new SoupRecipe();
        soup.prepareRecipe();
    }
}
```

ودي حكاية الـ Method Templete.

Behavioral Design Pattern

Observer

افترض إن عندك Store و Customer: الـCustomer مهتم جداً بمنتج معين (مثلاً، موديل جديد من الآيفون) والمنتج ده هيتوفر في المخزن قريباً.

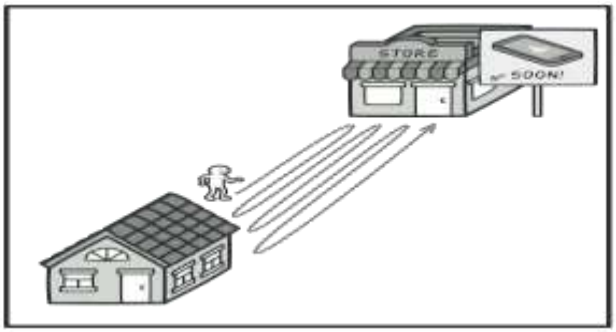
المشكلة بتيجي لو الـCustomer قرر يروح المخزن كل يوم عشان يتأكد إن المنتج بقى موجود. في الفترة دي، الزيارة دي مش هتفيد الـ Customer لإن المنتج لسه مش متاح، وكمان الـCustomer هيضيع وقته على الفاضي.

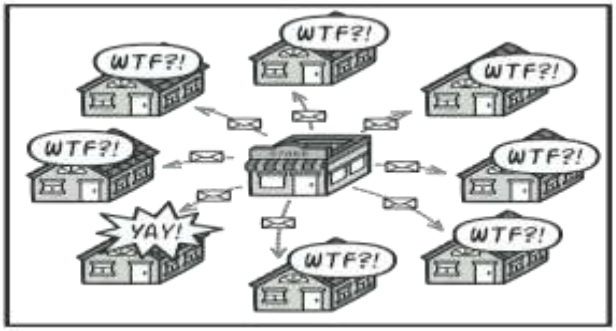
طب نخلي الـ Store تبعت Emails لكل الـ Customers لما يبقا المنتج متاح؟

دي مشكلة تانية، لأن الـCustomer اللي مش مهتم بالمنتجات الجديدة هيزعجه الإيميل ده.

طب الحل إيه ؟

الحل إنك تستخدم الـ Observer Pattern، بحيث الـStore هيبقا هو الـ publisher، وكل الـ Customers المهتمين بالمنتج ده يبقوا subscribers. الـCustomer يقدر يشترك في متابعة الـ Store علشان يعرف أول ما المنتج يتوفر، وكده المخزن يبقى عنده آلية لإرسال إشعار فقط للـCustomers المهتمين بالمنتج ده.





Behavioral Design Pattern

Observer

إزاي بيشتغل كـ Code ؟

اول حاجه عندي Interface و Customer class عادي جدا.

```java
// الكلاس الخاص بالـ Observer (العميل)
public interface Customer {
    void update(boolean isProductAvailable); // طريقة لتحديث حالة المنتج
}
```

```java
// عميل مهتم بالآيفون
public class IphoneCustomer implements Customer {
    @Override
    public void update(boolean isProductAvailable) {
        if (isProductAvailable) {
            System.out.println("IphoneCustomer: Product is available now");
        } else {
            System.out.println("IphoneCustomer: Product is not available");
        }
    }
}
```

وبعدين عندي Store Class.

```java
import java.util.ArrayList;
import java.util.List;

// الكلاس الخاص بالـ Publisher (المخزن)
public class Store {
    private List<Customer> customers = new ArrayList<>(); // لستة العملاء المشتركين
    private boolean isProductAvailable = false; // الحالة الأولية للمنتج

    // طريقة للاشتراك في المتابعة
    public void addCustomer(Customer customer) {
        customers.add(customer);
    }

    // طريقة لإلغاء الاشتراك
    public void removeCustomer(Customer customer) {
        customers.remove(customer);
    }

    // طريقة لتغيير حالة المنتج
    public void setProductAvailability(boolean availability) {
        this.isProductAvailable = availability;
        notifyCustomers(); // لو المنتج متاح، نبعت إشعار لكل العملاء
    }

    // إشعار العملاء بأي تحديث في حالة المنتج
    private void notifyCustomers() {
        for (Customer customer : customers) {
            customer.update(isProductAvailable); // بنرسل لهم التحديث
        }
    }
}
```

Behavioral Design Pattern

Observer

هنلاحظ ان عندنا List شايله الـ Customers اللي مهتمين بجيلهم ايميل عشان مبعتش لكل الـ Customers.

وبعدين عندنا الـ Main.

```java
public class Main {
    public static void main(String[] args) {
        // انشاء المخزن (Publisher)
        Store store = new Store();

        // انشاء العميل المهتم بالآيفون (Observer)
        IphoneCustomer iphoneCustomer = new IphoneCustomer();

        // العميل بيسجل اشتراكه في المتابعة
        store.addCustomer(iphoneCustomer);

        // تغير حالة المنتج (المنتج بقى متاح)
        System.out.println("Product is not available");
        store.setProductAvailability(false); // المنتج مش متاح

        System.out.println("Product is available");
        store.setProductAvailability(true); // المنتج متاح
    }
}
```

بضيف الـ Customer اللي مهتم بالمنتج بتاع الـ Store وبعدين بيروحلهم Update بحاله المنتج طا بعمل انه مش متاح او متاح.

ودي حكاية الـ Observer.

Behavioral Design Pattern

Strategy

فكر في نفسك لما تكون محتاج تروح مكان معين. في مواقف مختلفة، ممكن تختار وسيلة نقل مختلفة حسب الحالة أو الظروف، زي:

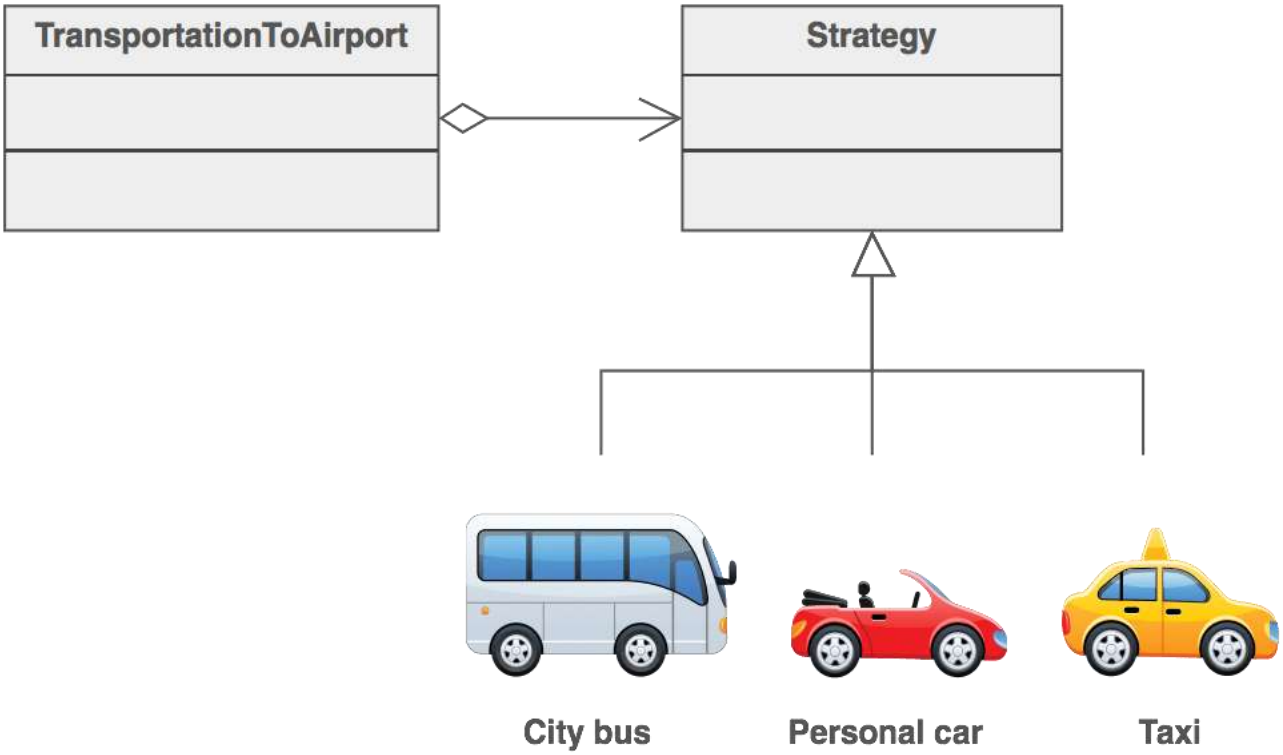
لو الطريق زحمة، تختار المترو عشان تتجنب الزحام.

لو كنت مستعجل وعايز توصل بسرعة، تختار التاكسي.

لو الجو حلو ومافيش زحمة، ممكن تختار المشي أو الدراجة.

في المثال ده، وسيلة النقل هي الاستراتيجية، والهدف هو الوصول للمكان.

و بتغير الاستراتيجية (وسيلة النقل) بناءً على الوضع.



Behavioral Design Pattern

Strategy

إزاي بيشتغل كـ Code ؟

أولا عندي الـ Interface بتاع كل الـ Strategies.

```
1
2 public interface TransportStrategy {
3     void travel();
4 }
5
```

بعدين عندي الـ Strategies المختلفة اللي كلهم بيعملوا نفس الحاجه (Travel).

```
1 public class CarStrategy implements TransportStrategy {
2     @Override
3     public void travel() {
4         System.out.println("Travel By Car");
5     }
6 }
```

```
1 public class BikeStrategy implements TransportStrategy {
2     @Override
3     public void travel() {
4         System.out.println("Travel By Bike");
5     }
6 }
```

```
1 public class MetroStrategy implements TransportStrategy {
2     @Override
3     public void travel() {
4         System.out.println("Travel By Metro");
5     }
6 }
```

كل استراتيجية من دول بتحقق نفس الهدف وهو السفر ولكن كل واحده بتعملها بطريقتها.

هعرف ازاي بقا انا هستخدم انهي استراتيجيه ؟ تابع معايا

Behavioral Design Pattern

Strategy

بعدين عندي Travelar Class ودي الـ Class اللي انا بتعامل معاه ومن خلاله بحدد انا عايز انهي استارتيجية. ودا بيخليني متعاملش مباشرة مع الـ Strategy Classes.

```
1 public class Traveler {
2     private TransportStrategy strategy;
3
4     public void setStrategy(TransportStrategy strategy) {
5         this.strategy = strategy;
6     }
7
8     public void travel() {
9         strategy.travel();
10    }
11 }
```

وبعدين عندي الـ Main.

```
1 public class Main {
2     public static void main(String[] args) {
3         Traveler traveler = new Traveler();
4
5         // اختر وسيلة النقل حسب الحاجة
6         traveler.setStrategy(new MetroStrategy());  // لو الطريق زحمة
7         traveler.travel();  // OutPut -> Travel By Metro
8
9         traveler.setStrategy(new CarStrategy());  // لو كنت مستعجل
10        traveler.travel();  // OutPut -> Travel By Car
11
12        traveler.setStrategy(new BikeStrategy());  // لو الجو حلو ومافيش زحمة
13        traveler.travel();  // OutPut -> Travel By Bike
14    }
15 }
```

هتقولي طب ليه مستخدمتش الـ Strategy Classes مباشرة ؟

يعني ليه مكتبتش كدا `TransportStrategy car = new CarStrategy();`

هقولك لان استخدامك لواجهه بينك وبين الـ Classes بتخلي الكود بتاعك قابل للتوسع بشكل احسن وف الـ Apps الضخمه هتخليك تقدر تضيف استراتيجيات تانيه بسهوله وتعدل براحتك بشكل احسن بكتير.

ودي حكايه الـ Strategy.

Behavioral Design Pattern

State

تخيل معايا عندك "آلة بيع تذاكر" في مكان عام. الآلة دي عندها حالات مختلفة زي حالة "انتظار" و حالة "بيع تذاكر" و حالة "إيقاف".

طا تكتب الكود علشان تدير الآلة دي، هتحتاج تكتب جمل if-else أو switch طول الوقت علشان تحدد السلوك بناءً على الحالة الحالية للآلة.

وبالتالي هيظهر عندي شوية مشاكل وهي ان:

١. الكود هيبقى معقد جدًا طا يكون في حالات كتير.

٢. لو عاوز تضيف حالة جديدة، هتضطر تروح تعدل الكود في أكتر من مكان، وهتبقى عرضة للأخطاء.

٣. صعوبة صيانة الكود على المدى الطويل.

الـ State Pattern بيحل المشكلة دي عن طريق تقسيم كل حالة في Class منفصل، وكل Class هيكون مسؤول عن سلوك معين. و طا الآلة تغير حالتها، بتغير الـClass اللي بتنفذ السلوك بدلاً من كتابة if-else أو switch كبيرة.

وبالتالي:

١. فصلت بين الـ States عندي.

٢. كل State بتحددلي الـ State اللي هتكون بعدها بشكل منفصل بدل ما الكود بتاعي يبقا بينفذ سلوك الـ State وف نفس الوقت بيحدد الـ States.

٣. سهولة إضافة State جديدة.

٤. سهولة الصيانة فيما بعد.

Behavioral Design Pattern

State

إزاي بينفذ كـ Code ؟

اول حاجه عندي State interface.

```java
interface State {
    void handleRequest(TicketMachine machine);
}
```

بعدين عندي Classes كل واحد بيمثل State معينة.

```java
// الحالة الأولى: في حالة انتظار
class WaitingState implements State {
    public void handleRequest(TicketMachine machine) {
        System.out.println("Machine is pending. Sale has been activated.");
        machine.setState(new SellingState());  // تغير الحالة لبيع التذاكر
    }
}
```

```java
// الحالة الثانية: في حالة بيع تذكرة
class SellingState implements State {
    public void handleRequest(TicketMachine machine) {
        System.out.println("The machine is in ticket selling state.");
        machine.setState(new StoppedState());  // تغير الحالة لإيقاف الآلة
    }
}
```

```java
// الحالة الثالثة: في حالة إيقاف
class StoppedState implements State {
    public void handleRequest(TicketMachine machine) {
        System.out.println("The machine is stopped.");
        machine.setState(new WaitingState());  // تغير الحالة للانتظار
    }
}
```

Behavioral Design Pattern

State

ومن ثم عندي Class بيمثل الـ Ticket Machine.

```java
// فئة الآلة نفسها اللي بتدير الحالات
class TicketMachine {
    private State currentState;  // الحالة الحالية

    public TicketMachine() {
        // البداية تكون في حالة انتظار
        currentState = new WaitingState();
    }

    public void setState(State state) {
        currentState = state;  // تغيير الحالة
    }

    public void request() {
        currentState.handleRequest(this);  // تنفيذ السلوك على حسب الحالة الحالية
    }
}
```

وزي ماهو واضح قدامنا الـ Machine بيبقا ليها State معينه و الـ State الابتدائية اللي معايا هي الـ Waiting state.

وبعد كدا عندي الـ Main.

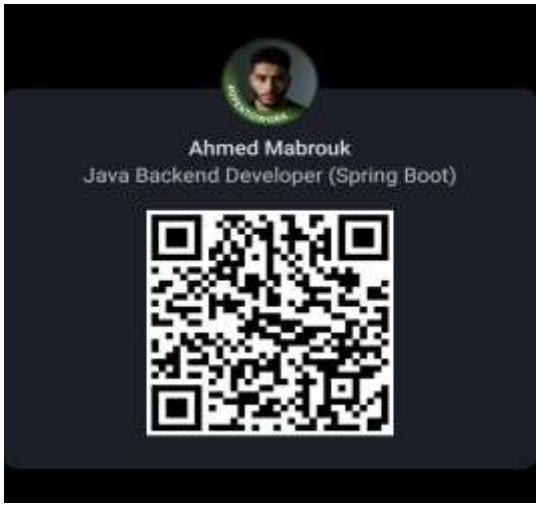
```java
public class Main {
    public static void main(String[] args) {
        TicketMachine machine = new TicketMachine(); // إنشاء آلة بيع التذاكر

        // المحاكاة: الآلة بتبدأ في حالة انتظار
        machine.request();  // Machine is pending. Sale has been activated.

        // الآلة بتتحول لحالة بيع التذاكر
        machine.request();  // The machine is in ticket selling state.

        // الآلة بتتحول لحالة إيقاف
        machine.request();  // The machine is stopped.

        // الآلة بتتحول تاني لحالة انتظار
        machine.request();  // Machine is pending. Sale has been activated.
    }
}
```

ودي حكاية الـ State Design Pattern

وها هنا تنتهي سهرتنا مع الـ Design Pattern.

الاكواد بتاعة كل الـ Patterns هتلاقوها ع الـ GitHub

If you have any feedback, I welcome the opportunity to hear from you. Please feel free to contact me.



وأخيرًا