Design Pattern

✅ What is a **Design Pattern?**

A **Design Pattern** is a **general reusable solution** to a common problem that occurs within a given **context** in **software design**.

It's **not code**, but rather a **template** or a **best practice** that can be used to solve a problem in many different situations.

Think of it like a **blueprint** that you can customize to solve a specific design issue in your application.

By Eslam Khder

spring®

EraaSoft

Categories of Design Patterns:

1. **Creational Patterns**: Deal with object creation.
    - Examples: Singleton, Factory, Abstract Factory, Builder, Prototype
2. **Structural Patterns**: Deal with object composition and relationships.
    - Examples: Adapter, Composite, Proxy, Decorator, Facade, Bridge
3. **Behavioral Patterns**: Deal with object interaction and communication.
    - Examples: Observer, Strategy, Command, Iterator, State, Template Method

✅ What are the **Benefits** of Using Design Patterns?

Improved code maintainability: Patterns make the code easier to manage and extend.

Better communication: Patterns give developers a shared vocabulary (e.g., "Let's use a Factory pattern here").

Code reusability: Promotes DRY (Don't Repeat Yourself) principles.

Faster development: Avoid reinventing the wheel for common problems.

Loosely coupled code: Easier to test, modify, and scale.

By Eslam Khder

🔒 What is the **Singleton Design Pattern?**

The **Singleton Pattern** ensures that **a class has only one instance** and provides a **global point of access** to it.

✅ When to Use It:

- When exactly **one object** is needed to **coordinate actions** across the system.
- When you need a **centralized configuration**, **logging**, **database connection**, or **cache manager**.

By Eslam Khder


```java
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        // Private constructor to prevent instantiation
    }

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

By Eslam Khder

🏭 What is the **Factory Design Pattern?**

The **Factory Pattern** is a **creational design pattern** that provides an **interface** for creating objects in a **superclass**, but allows **subclasses to alter the type of objects** that will be created.

In simple terms:

> It **hides the object creation logic** from the client and allows you to create objects **without exposing the instantiation logic**.

By Eslam Khder

🍕 BEFORE Applying Factory Pattern

```java
// Pizza Interface
public interface Pizza {
    void prepare();
    void bake();
    void cut();
}
```

```java
// VegPizza class
public class VegPizza implements Pizza {
    public void prepare() { System.out.println("Preparing Veg Pizza"); }
    public void bake() { System.out.println("Baking Veg Pizza"); }
    public void cut() { System.out.println("Cutting Veg Pizza"); }
}
```

```java
// CheesePizza class
public class CheesePizza implements Pizza {
    public void prepare() { System.out.println("Preparing Cheese Pizza"); }
    public void bake() { System.out.println("Baking Cheese Pizza"); }
    public void cut() { System.out.println("Cutting Cheese Pizza"); }
}
```

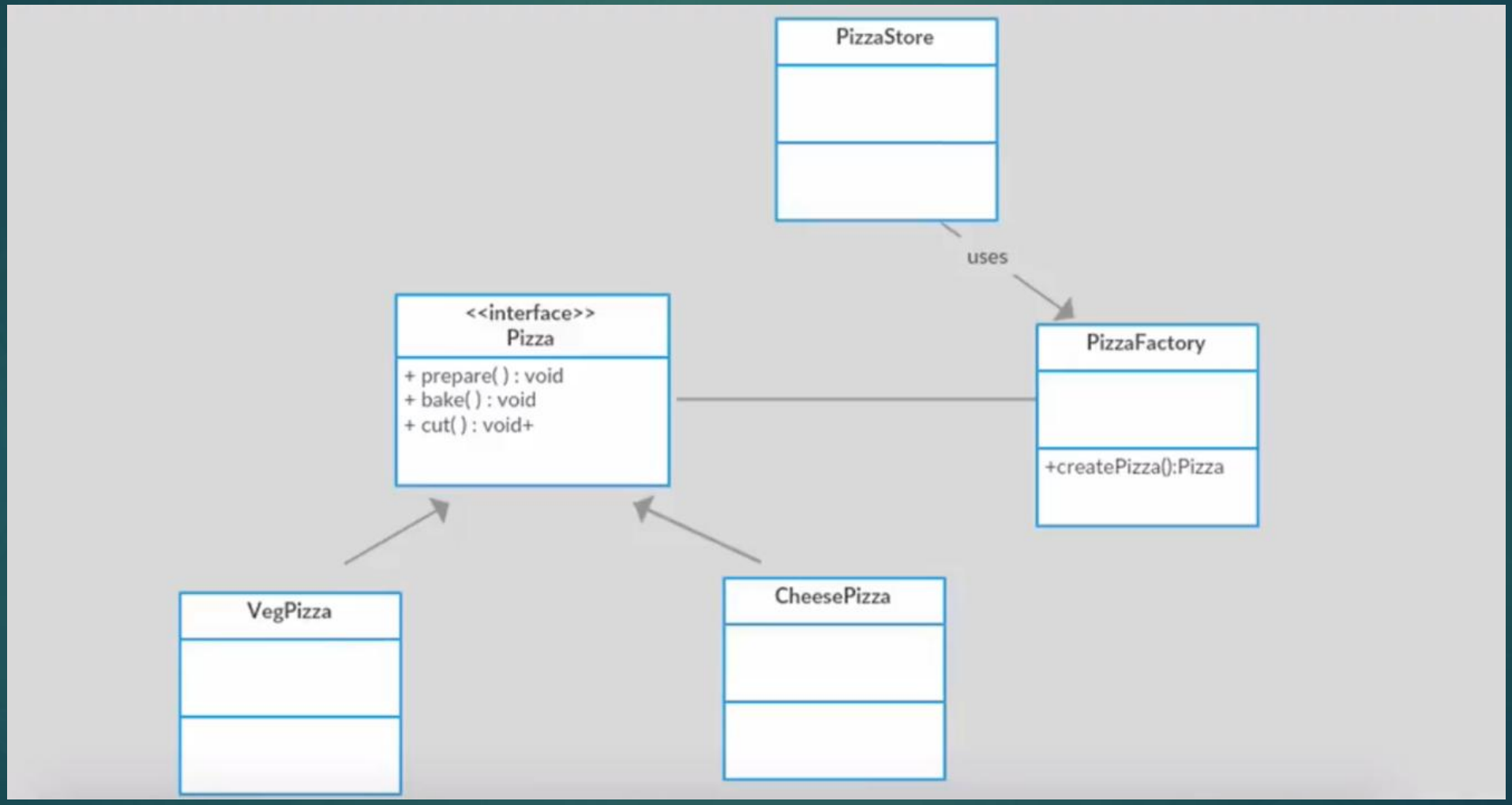
```java
// PizzaStore directly creates pizza types
public class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza = null;

        if (type.equalsIgnoreCase("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equalsIgnoreCase("veg")) {
            pizza = new VegPizza();
        }

        pizza.prepare();
        pizza.bake();
        pizza.cut();

        return pizza;
    }
}
```

By Eslam Khder

🍕 BEFORE Applying Factory Pattern

The client (PizzaStore) is directly creating objects like `VegPizza` or `CheesePizza`, which makes it tightly coupled to specific pizza types.

By Eslam Khder



By Eslam Khder

After Applying Factory Pattern

Pizza.java

```
package com.bharath.patterns.factory;

public interface Pizza {

    void prepare();

    void bake();

    void cut();

}
```

Pizza.java | CheesePizza.java

```
package com.bharath.patterns.factory;

public class CheesePizza implements Pizza {

    @Override
    public void prepare() {
        System.out.println("Preparing Cheese Pizza");
    }

    @Override
    public void bake() {
        System.out.println("Baking Cheese Pizza");

    }

    @Override
    public void cut() {
        System.out.println("Cutting Cheese Pizza");

    }

}
```

Pizza.java | CheesePizza.java | ChickenPizza.java

```
package com.bharath.patterns.factory;

public class ChickenPizza implements Pizza {

    @Override
    public void prepare() {
        System.out.println("Preparing Chicken Pizza");
    }

    @Override
    public void bake() {
        System.out.println("Baking Chicken Pizza");

    }

    @Override
    public void cut() {
        System.out.println("Cutting Chicken Pizza");

    }

}
```

By Eslam Khder

🍕 After Applying Factory Pattern

Pizza.java | CheesePizza.java | ChickenPizza.java | VeggiePizza.java | PizzaStore.java | Test.java | PizzaFactory.java

```java
package com.bharath.patterns.factory;

public class PizzaFactory {

    public static Pizza createPizza(String type) {
        Pizza p = null;

        if (type.equals("cheese")) {
            p = new CheesePizza();
        } else if (type.equals("chicken")) {
            p = new ChickenPizza();
        } else if (type.equals("veggie")) {
            p = new VeggiePizza();
        }

        return p;
    }

}
```

```java
public class Test {

    public static void main(String[] args) {

        PizzaStore ps = new PizzaStore();
        ps.orderPizza("chicken");

    }

}
```

Pizza.java | CheesePizza.java | ChickenPizza.java | VeggiePizza.java | *PizzaStore.java

```java
package com.bharath.patterns.factory;

public class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza p = PizzaFactory.createPizza(type);
        p.prepare();
        p.bake();
        p.cut();

        return p;
    }

}
```

🍕 AFTER Applying Factory Pattern

Now `PizzaStore` uses `PizzaFactory` to create pizza objects, which improves modularity and decouples the object creation logic.

By Eslam Khder

🎯 Benefit You Get After Refactor:

- `PizzaStore` no longer depends on concrete classes like `VegPizza` or `CheesePizza`.
- You can easily add new pizza types without changing the store logic—only update the factory.

By Eslam Khder

🧠 أولًا: تطبيقات حقيقية فعليًا بتستخدم Factory Pattern

1. JDBC (في Java Database Connectivity)

لما بتكتب كود زي كده:

Copy code

java

```
Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/db", "root", "1234");
```

🔹 هل إنت هنا عملت `new MySQLConnection()`؟

❌ لأ

🔹 اللي حصل فعلاً:

`DriverManager` بيستخدم **Factory Pattern** علشان يرجع لك كائن من نوع `Connection`

(ممكن يكون MySQL أو Oracle أو PostgreSQL على حسب الـ URL).

📦 يعني:

- `DriverManager` → Factory
- `Connection` → Product
- `Main` أو الـ DAO → Client

✅ وده مثال حي فعلاً كلنا بنستخدمه يوميًا مع قواعد البيانات

spring

EraaSoft

Hibernate – SessionFactory .2

زي ما شرحنا قبل شوية،

`SessionFactory` هو Factory بيولد لك Sessions للتعامل مع الـ DB.

Copy code java

```java
SessionFactory sf = new Configuration().configure().buildSessionFactory();
Session session = sf.openSession();
```

By Eslam Khder

🧩 ثانيًا: تطبيق عملي بسيط بالكود (من حياتنا اليومية)

تخيل إنك بتبني تطبيق إرسال إشعارات 🔔

ممكن تبعت:

- Email
- SMS
- Push Notification

كل نوع ليه منطق خاص في الإرسال.

spring®

EraaSoft

🧱 الكود بدون Factory (الفوضى 💀)

Copy code

java

```java
public class NotificationService {
    public void sendNotification(String type, String message) {
        if (type.equals("email")) {
            System.out.println("Sending EMAIL: " + message);
        } else if (type.equals("sms")) {
            System.out.println("Sending SMS: " + message);
        } else if (type.equals("push")) {
            System.out.println("Sending PUSH: " + message);
        } else {
            System.out.println("Unknown type");
        }
    }
}
```

لو بكرة أضفت `WhatsAppNotification` → لازم تعدّل هنا 👇

وده ضد مبدأ **Open/Closed Principle**.

By Eslam Khder

🏭 بالكود مع Factory Pattern ✅

1️⃣ نعمل Interface عام

Copy code

java

```java
public interface Notification {
    void notifyUser(String message);
}
```

By Eslam Khder

2 نعمل Implementations لكل نوع

Copy code java

```java
public class EmailNotification implements Notification {
    public void notifyUser(String message) {
        System.out.println("Sending EMAIL: " + message);
    }
}

public class SmsNotification implements Notification {
    public void notifyUser(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

public class PushNotification implements Notification {
    public void notifyUser(String message) {
        System.out.println("Sending PUSH: " + message);
    }
}
```

3 نعمل Factory مسؤول عن الإنشاء

Copy code java

```java
public class NotificationFactory {
    public static Notification createNotification(String type) {
        switch (type.toLowerCase()) {
            case "email":
                return new EmailNotification();
            case "sms":
                return new SmsNotification();
            case "push":
                return new PushNotification();
            default:
                throw new IllegalArgumentException("Unknown notification type: " + type);
        }
    }
}
```

By Eslam Khder

4 نستخدمه في الكود الرئيسي

Copy code java

```java
public class Main {
    public static void main(String[] args) {
        Notification n1 = NotificationFactory.createNotification("email");
        n1.notifyUser("Welcome to our app!");

        Notification n2 = NotificationFactory.createNotification("sms");
        n2.notifyUser("Your OTP is 1234");
    }
}
```

By Eslam Khder

🧠 مثال حقيقي من الواقع:

تخيل في شركتك فيه موظف واحد اسمه "أحمد" بيعمل كل حاجة:

- يطبخ 🍔
- يقدّم الأكل 🍽️
- يحاسب الزبون 💰

لو عايز تضيف خدمة جديدة (مثلاً "طلبات أونلاين")
لازم تعلّم أحمد كمان البرمجة 😅
ده هو الكود اللي جوه كل حاجة.

لكن لما تعمل نظام منظم:

- الشيف مسؤول عن الطبخ
- الكاشير مسؤول عن الفواتير
- والدليفري مسؤول عن التوصيل

لو عايز تضيف خدمة جديدة (مثلاً "Delivery")
بتضيف شخص جديد فقط،
مش بتغيّر في كل الأقسام.
وده بالضبط اللي بيعمله الـ **Factory Pattern**.

✅ الخلاصة:

> الفرق مش إنك "مش هتعدل"
> الفرق إنك "لما تعدل، التعديل هيكون في مكان واحد مخصص وواضح".

By Eslam Khder

spring®

EraaSoft

🏭 What is **Abstract Factory Design Pattern?**

The **Abstract Factory Pattern** is a **creational pattern** that provides an **interface for creating families of related or dependent objects** without specifying their concrete classes.

✅ When to Use:

- When your system needs to be **independent of how its objects are created.**
- When you need to create **multiple objects** that belong to the same **family.**

By Eslam Khder

❌ BEFORE Abstract Factory (Tightly Coupled Code)

```java
// Dao Interface
interface Dao {
    void save();
}
```

```java
// Concrete Implementations
class XMLEmpDao implements Dao {
    @Override
    public void save() {
        System.out.println("Saving Employee to XML");
    }
}

class XMLDeptDao implements Dao {
    @Override
    public void save() {
        System.out.println("Saving Department to XML");
    }
}
```

```java
class DBEmpDao implements Dao {
    @Override
    public void save() {
        System.out.println("Saving Employee to DB");
    }
}

class DBDeptDao implements Dao {
    @Override
    public void save() {
        System.out.println("Saving Department to DB");
    }
}
```

By Eslam Khder

❌ BEFORE Abstract Factory (Tightly Coupled Code)

```java
public static void main(String[] args) {
    // Tight coupling: Client directly creates concrete DAOs
    Dao empDao = new XMLEmpDao();  // Hardcoded XML dependency
    Dao deptDao = new XMLDeptDao();

    // If we switch to DB, we must modify the Client:
    // Dao empDao = new DBEmpDao();
    // Dao deptDao = new DBDeptDao();

    empDao.save();
    deptDao.save();
}
```

By Eslam Khder

❌ BEFORE Abstract Factory (Tightly Coupled Code)

EraaSoft

Problems with This Approach

1. Tight Coupling (Hard to Change Storage)

- **Issue:** The `Client` directly creates `XMLEmpDao` or `DBEmpDao`.
- **Consequence:** If you want to switch from **XML to DB**, you must **modify the `Client` code** (uncomment/comment lines).

java Copy Download

```
// Switching storage requires code changes:
Dao empDao = new DBEmpDao();  // Manual change needed
Dao deptDao = new DBDeptDao();
```

By Eslam Khder

❌ BEFORE Abstract Factory (Tightly Coupled Code)

EraaSoft

2. Risk of Inconsistency

- **Issue:** Nothing stops you from mixing **XML** and **DB** DAOs by mistake:

java Copy Download

```java
Dao empDao = new XMLEmpDao();      // XML
Dao deptDao = new DBDeptDao();     // DB (Oops! Inconsistent!)
```

- **Consequence:** Employee data goes to **XML**, but Department data goes to **DB** → **Data inconsistency!**

By Eslam Khder

❌ BEFORE Abstract Factory (Tightly Coupled Code)

3. Violates Open/Closed Principle

.Open for extension, Closed for modification

- **Issue:** If you add a new storage type (e.g., **JSON**), you must modify the `Client` to support it:

java Copy Download

```java
// New JSON DAOs (requires Client changes)
Dao empDao = new JSONEmpDao();  // Breaks Open/Closed Principle
Dao deptDao = new JSONDeptDao();
```

By Eslam Khder

❌ BEFORE Abstract Factory (Tightly Coupled Code)

🧱 Open/Closed Principle (OCP) – One of the SOLID Principles

Definition:

> *Software entities (classes, modules, functions, etc.) should be **open for extension**, but **closed for modification**.*

🔓 "Open for Extension"

You can add **new behavior or functionality** to a class **without changing its existing source code**.

🔒 "Closed for Modification"

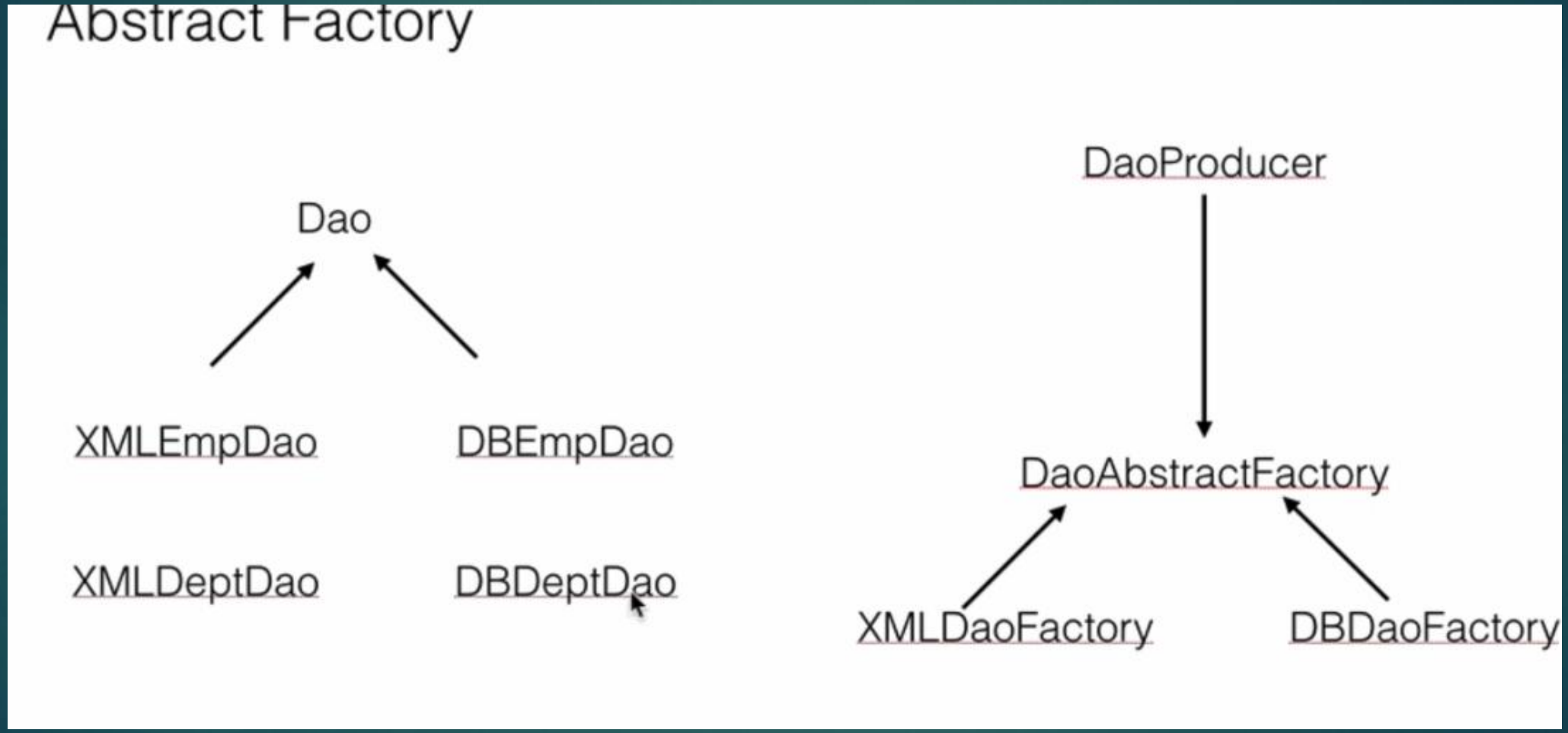
Once a class is written and tested, you **shouldn't modify it directly** to add new features—especially in ways that might break existing code.

By Eslam Khder

spring

✅ AFTER Abstract Factory (Loosely Coupled and Scalable)

EraaSoft



By Eslam Khder

```java
public interface Dao {

    void save();
}
```

Dao.java | XMLEmpDao.java

```java
package com.bharath.patterns.abstractfactory;

public class XMLEmpDao implements Dao {

    @Override
    public void save() {
        System.out.println("Saving Employee to XML");
    }

}
```

Dao.java | XMLEmpDao.java | XMLDeptDao.java | DBDeptDao.java

```java
package com.bharath.patterns.abstractfactory;

public class DBDeptDao implements Dao {

    @Override
    public void save() {
        System.out.println("Saving Department to DB");
    }

}
```

Dao.java | XMLEmpDao.java | XMLDeptDao.java

```java
package com.bharath.patterns.abstractfactory;

public class XMLDeptDao implements Dao {

    @Override
    public void save() {
        System.out.println("Saving Department to XML");
    }

}
```

XMLEmpDao.java | XMLDeptDao.java | DBDeptDao.java | DBEmpDao.java

```java
package com.bharath.patterns.abstractfactory;

public class DBEmpDao implements Dao {

    @Override
    public void save() {
        System.out.println("Saving Employee to DB");
    }

}
```

By Eslam Khder

✅ AFTER Abstract Factory (Loosely Coupled and Scalable)

Dao.java | XMLEmpDao.java | XMLDeptDao.java | DBDeptDao.java | DBEmpDao.java | *DaoAbstractFactory.java

```
package com.bharath.patterns.abstractfactory;

public abstract class DaoAbstractFactory {

    public abstract Dao createDao(String type);
}
```

By Eslam Khder

✅ AFTER Abstract Factory (Loosely Coupled and Scalable)

Dao.java | XMLEmpDao.java | XMLDeptDao.java | DBDeptDao.java | DBEmpDao.java | DaoAbstractFactory.java | XMLDaoFactory.java

```java
package com.bharath.patterns.abstractfactory;

public class XMLDaoFactory extends DaoAbstractFactory {

    @Override
    public Dao createDao(String type) {
        Dao dao = null;
        if (type.equals("emp")) {
            dao = new XMLEmpDao();
        } else if (type.equals("dept")) {
            dao = new XMLDeptDao();
        }
        return dao;
    }

}
```

Dao.java | XMLEmpDao.java | XMLDeptDao.java | DBDeptDao.java | DBEmpDao.java | DaoAbstractFactory.java | XMLDaoFactory.java | *DBDaoFactory.java

```java
package com.bharath.patterns.abstractfactory;

public class DBDaoFactory extends DaoAbstractFactory {

    @Override
    public Dao createDao(String type) {
        Dao dao = null;
        if (type.equals("emp")) {
            dao = new DBEmpDao();
        } else if (type.equals("dept")) {
            dao = new DBDeptDao();
        }
        return dao;
    }

}
```

By Eslam Khder

✅ AFTER Abstract Factory (Loosely Coupled and Scalable)

XMLEmpDao.java | XMLDeptDao.java | DBDeptDao.java | DBEmpDao.java | DaoAbstractFact | XMLDaoFactory.j | DBDaoFactory.ja | DaoFactoryProdu

```java
package com.bharath.patterns.abstractfactory;

public class DaoFactoryProducer {

    public static DaoAbstractFactory produce(String factoryType) {
        DaoAbstractFactory daf = null;

        if (factoryType.equals("xml")) {
            daf = new XMLDaoFactory();
        } else if (factoryType.equals("db")) {
            daf = new DBDaoFactory();
        }

        return daf;

    }
}
```

By Eslam Khder

✅ AFTER Abstract Factory (Loosely Coupled and Scalable)

```java
public class Test {

    public static void main(String[] args) {

        DaoAbstractFactory daf = DaoFactoryProducer.produce("xml");
        Dao dao = daf.createDao("emp");
        dao.save();

    }

}
```

```java
public static void main(String[] args) {

    DaoAbstractFactory daf = DaoFactoryProducer.produce("db");
    Dao dao = daf.createDao("emp");
    dao.save();

}
```

By Eslam Khder

spring®

EraaSoft

✅ Template Design Pattern – Overview

The **Template Design Pattern** defines the **skeleton of an algorithm** in a **base class**, but **lets subclasses override specific steps** of the algorithm **without changing its structure**.

By Eslam Khder


```java
public class XMLRenderer {
    public void render() {
        Object data = readData();
        Object processed = processData(data);
        System.out.println("Rendering: " + processed);
    }

    private Object readData() {
        // Read XML
        return "XML Data";
    }

    private Object processData(Object data) {
        // Process XML
        return "Processed " + data;
    }
}
```

By Eslam Khder


```java
public class CSVRenderer {
    public void render() {
        Object data = readData();
        Object processed = processData(data);
        System.out.println("Rendering: " + processed);
    }

    private Object readData() {
        // Read CSV
        return "CSV Data";
    }

    private Object processData(Object data) {
        // Process CSV
        return "Processed " + data;
    }
}
```

By Eslam Khder

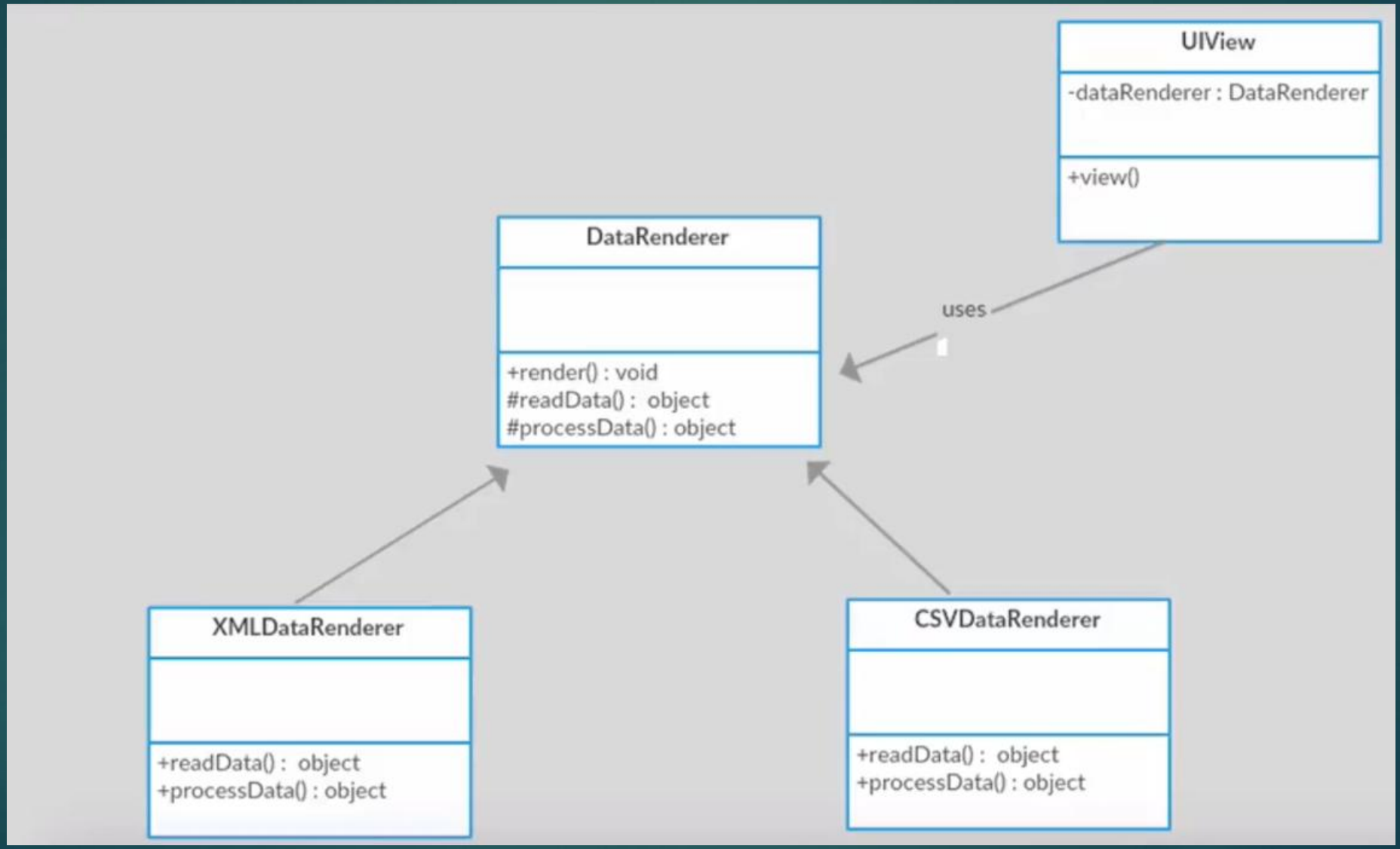
- **Problem**: Duplicate logic in `render()` for XML and CSV.

By Eslam Khder

spring®

✅ Code After Applying Template Design Pattern

EraaSoft



By Eslam Khder


```java
// Subclass for XML
public class XMLDataRenderer extends DataRenderer {
    @Override
    protected Object readData() {
        return "XML Data";
    }

    @Override
    protected Object processData(Object data) {
        return "Processed " + data;
    }
}
```

```java
// Subclass for CSV
public class CSVDataRenderer extends DataRenderer {
    @Override
    protected Object readData() {
        return "CSV Data";
    }

    @Override
    protected Object processData(Object data) {
        return "Processed " + data;
    }
}
```

By Eslam Khder


```java
// Abstract class - defines template method
public abstract class DataRenderer {
    public void render() {
        Object data = readData();
        Object processed = processData(data);
        System.out.println("Rendering: " + processed);
    }

    protected abstract Object readData();
    protected abstract Object processData(Object data);
}
```

By Eslam Khder

✅ Benefits You Gain:

- **Avoids duplication** of `render()` logic.
- Subclasses focus **only** on differences (read/process logic).
- **Open/Closed Principle**: Easy to add new data types (e.g., JSON).

Adapter pattern

The **Adapter Design Pattern** is a **structural design pattern** that allows objects with incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces.

✅ Use Case (Why Use It?)

Suppose you have a class that expects input in one format, but you have another class or library that provides output in a different format. Instead of changing existing code, you **create an adapter** to convert the format.

By Eslam Khder

الـ **Adapter Design Pattern** من أهم وأنظف الأنماط في التصميم (Structural Patterns)،
وبيتحل بيه مشكلة بتحصل في كل المشاريع تقريبًا، وهي:

> عندي كود جاهز أو مكتبة خارجية، بس مش "ماشية" مع الكود بتاعي — أعمل إيه؟ 🤔

By Eslam Khder

🎯 الفكرة العامة في جملة واحدة:

Adapter Pattern هو "محوّل" (زي الفيشة اللي بتحوّل من نوع كهربا لنوع تاني 🔌).

وظيفته إنه **يوصل بين كودين غير متوافقين**،
بحيث يقدروا يشتغلوا مع بعض بدون ما تعدّل الكود الأصلي.

By Eslam Khder

spring

EraaSoft

🧠 مثال من الحياة الواقعية

تخيل إن عندك **شاحن موبايل** مصري (فيشة ٢ دبوس)،
وسافرت لأوروبا، هناك الفيشة ٣ دبوس.

هل هتغيّر الشاحن؟ 😅
أكيد لأ — هتشتري **Adapter** (محول فيشة).

المحول ده **ما غيّرش لا الشاحن ولا المقبس**،
هو بس ترجم بينهم بحيث كل واحد يفهم التاني ⚡

By Eslam Khder

🧩 الجزء الأول: قبل استخدام الـ Adapter Pattern (المشكلة)

تخيل إن عندك كودك بيستخدم `PaymentProcessor` بالشكل ده:

Copy code

java

```java
// ✅ الكود الأساسي بتاعك
interface PaymentProcessor {
    void pay(String account, double amount);
}

// كلاس بيستخدم الـ PaymentProcessor
class ShoppingCart {
    private PaymentProcessor paymentProcessor;

    public ShoppingCart(PaymentProcessor paymentProcessor) {
        this.paymentProcessor = paymentProcessor;
    }

    public void checkout(String account, double amount) {
        paymentProcessor.pay(account, amount);
    }
}
```


```java
class CreditCardPayment implements PaymentProcessor {
    @Override
    public void pay(String account, double amount) {
        System.out.println("Processing Credit Card payment for " + account + " amount " + amount);
    }
}
```

```java
class BankTransferPayment implements PaymentProcessor {
    @Override
    public void pay(String account, double amount) {
        System.out.println("Processing Bank Transfer for account " + account + " amount " + amount
    }
}
```

Copy code java

```java
class PayPalPayment implements PaymentProcessor {
    @Override
    public void pay(String account, double amount) {
        System.out.println("Processing PayPal payment for user " + account + " with amount " + amc
    }
}
```

By Eslam Khder


```java
public class Main {
    public static void main(String[] args) {
        // 1 الدفع بالكريدت كارد
        PaymentProcessor credit = new CreditCardPayment();
        ShoppingCart cart1 = new ShoppingCart(credit);
        cart1.checkout("EslamCard", 150.0);

        // 2 الدفع عن طريق البنك
        PaymentProcessor bank = new BankTransferPayment();
        ShoppingCart cart2 = new ShoppingCart(bank);
        cart2.checkout("EslamBank", 300.0);

        // 3 الدفع بالبايبال
        PaymentProcessor paypal = new PayPalPayment();
        ShoppingCart cart3 = new ShoppingCart(paypal);
        cart3.checkout("EslamPayPal", 250.0);
    }
}
```

Copy code

rust

```
Processing Credit Card payment for EslamCard amount 150.0
Processing Bank Transfer for account EslamBank amount 300.0
Processing PayPal payment for user EslamPayPal with amount 250.0
```

By Eslam Khder

🧠 بعدين — جاتلك مكتبة قديمة مش ماشية مع الكود بتاعك

Copy code java

```java
class OldPaymentSystem {
    public void makePayment(String accNo, double money) {
        System.out.println("Payment done for " + accNo + " of amount " + money);
    }
}
```

By Eslam Khder

💡 نعمل Adapter علشان نخليها تشتغل مع كودنا الحالي

Copy code java

```java
class PaymentAdapter implements PaymentProcessor {
    private OldPaymentSystem oldSystem;

    public PaymentAdapter(OldPaymentSystem oldSystem) {
        this.oldSystem = oldSystem;
    }

    @Override
    public void pay(String account, double amount) {
        oldSystem.makePayment(account, amount);
    }
}
```

By Eslam Khder


```java
public class Main {
    public static void main(String[] args) {
        // 1 الدفع بالكريدت كارد
        PaymentProcessor credit = new CreditCardPayment();
        ShoppingCart cart1 = new ShoppingCart(credit);
        cart1.checkout("EslamCard", 150.0);

        // 2 الدفع عن طريق التحويل البنكي
        PaymentProcessor bank = new BankTransferPayment();
        ShoppingCart cart2 = new ShoppingCart(bank);
        cart2.checkout("EslamBank", 300.0);

        // 3 الدفع بالـ PayPal
        PaymentProcessor paypal = new PayPalPayment();
        ShoppingCart cart3 = new ShoppingCart(paypal);
        cart3.checkout("EslamPayPal", 250.0);

        // 4 استخدام النظام القديم لكن من خلال الـ Adapter
        OldPaymentSystem oldSystem = new OldPaymentSystem();
        PaymentProcessor adapter = new PaymentAdapter(oldSystem);
        ShoppingCart cart4 = new ShoppingCart(adapter);
        cart4.checkout("EslamOld", 500.0);
```

By Eslam Khder

📜 الناتج في الكونسول:

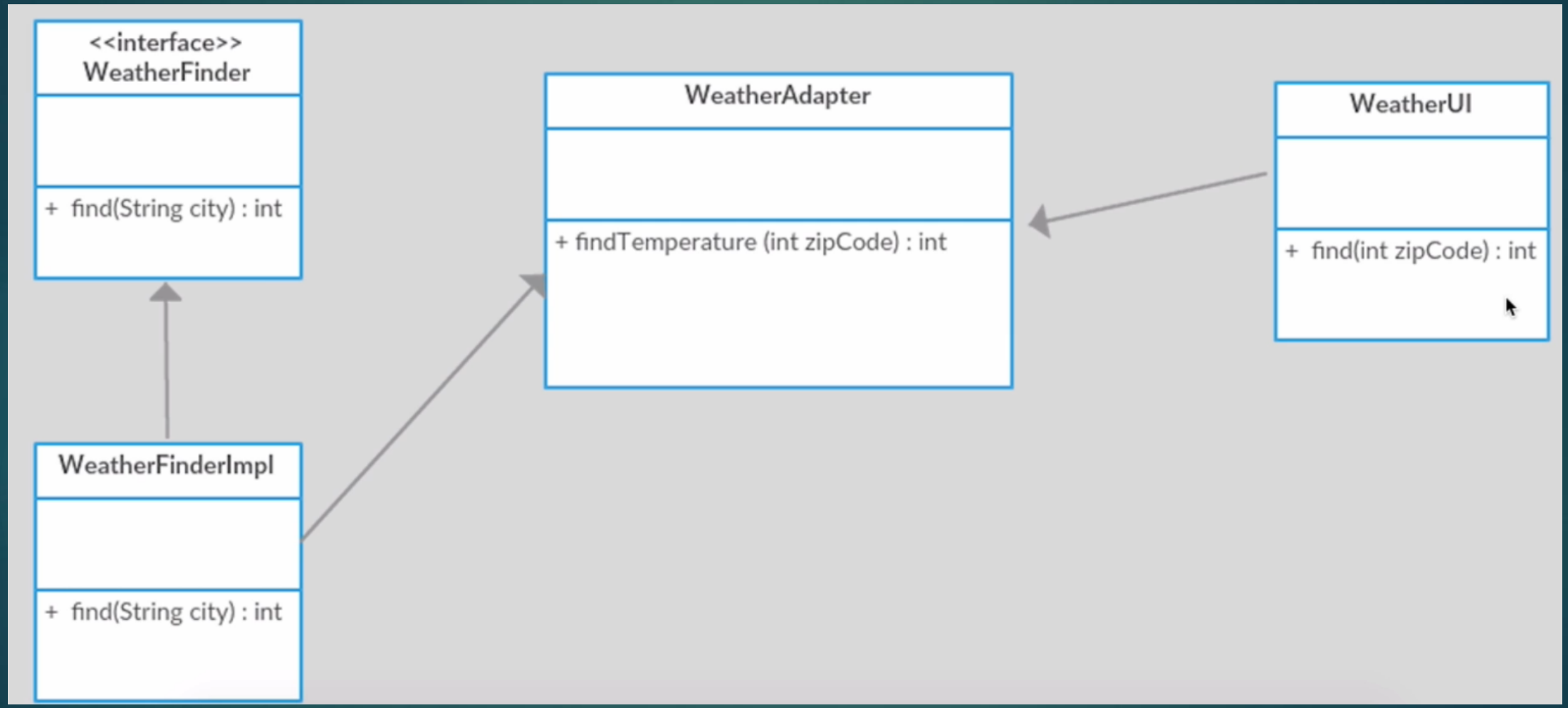
Copy code rust

```
Processing Credit Card payment for EslamCard amount 150.0
Processing Bank Transfer for account EslamBank amount 300.0
Processing PayPal payment for user EslamPayPal with amount 250.0
✅ [OLD SYSTEM] Payment done for EslamOld of amount 500.0
```

🎯 الفكرة كلها في سطرين:

- كل وسائل الدفع الجديدة بتنفّذ الـ `PaymentProcessor` interface بشكل مباشر.
- النظام القديم مش متوافق، فعملنا له **Adapter** علشان يشتغل بنفس الطريقة من غير ما نعدّل عليه.

By Eslam Khder



By Eslam Khder

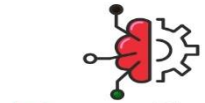
```java
public interface WeatherFinder {

    public int find(String city);

}
```

```java
public class WeatherFinderImpl implements WeatherFinder {

    @Override
    public int find(String city) {
        return 33;
    }

}
```

```java
public class WeatherUI {

    public void showTemperature(int zipcode) {
        WeatherFinder finder = new WeatherFinderImpl();
        finder.find

    }

}
```

Problem

By Eslam Khder




```java
public class WeatherAdapter {

    public int findTemperature(int zipCode) {
        String city = null;

        if (zipCode == 19406) {
            city = "King Of Prussia";
        }

        WeatherFinder finder = new WeatherFinderImpl();
        int temperature = finder.find(city);

        return temperature;

    }

}
```

```java
public class WeatherUI {

    public void showTemperature(int zipcode) {
        WeatherAdapter adapter = new WeatherAdapter();
        System.out.println(adapter.findTemperature(zipcode));

    }

    public static void main(String[] args) {
        WeatherUI ui = new WeatherUI();
        ui.showTemperature(19406);
    }

}
```

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder