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

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

spring®

EraaSoft

🔒 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

spring®

🍕 BEFORE Applying Factory Pattern

EraaSoft

```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"); }
}
```

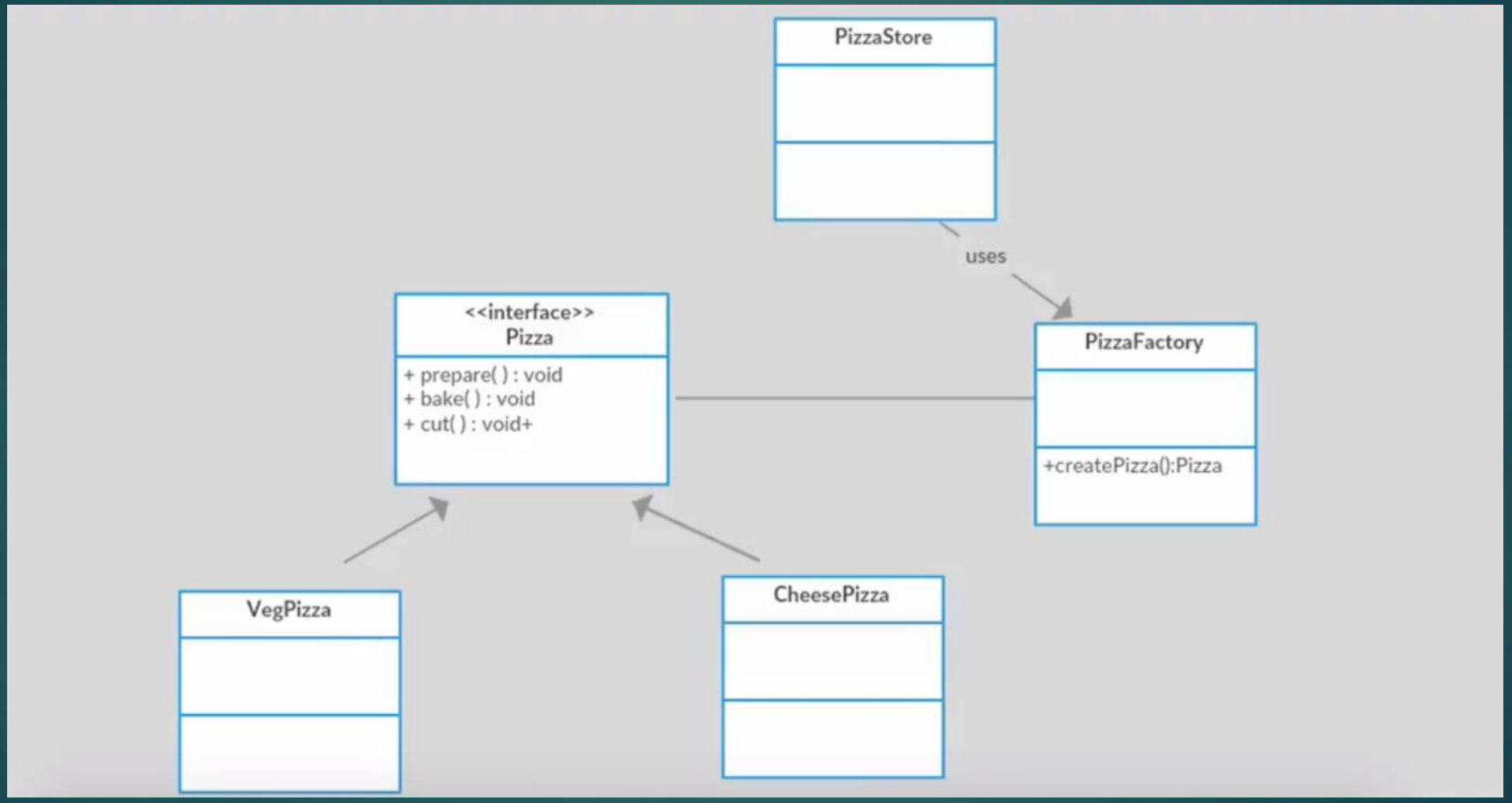
```java
// PizzaStore directly creates pizza types
public class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza = null;

        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

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)

EraaSoft

```
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)

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)

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

- **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"

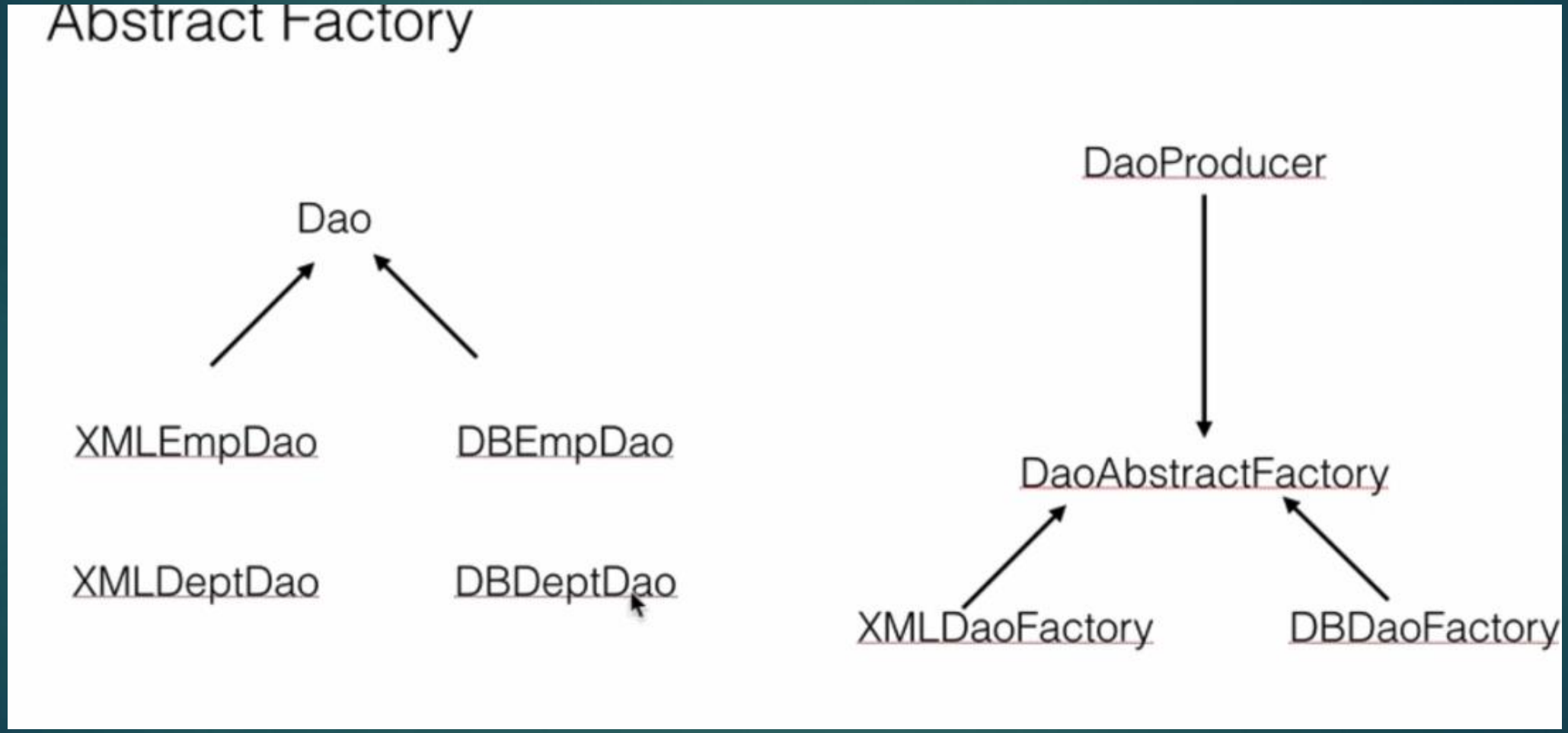
Once a class is written and tested, you **shouldn't modify it directly** to add new features—especially in ways that might break existing code.

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

✅ 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

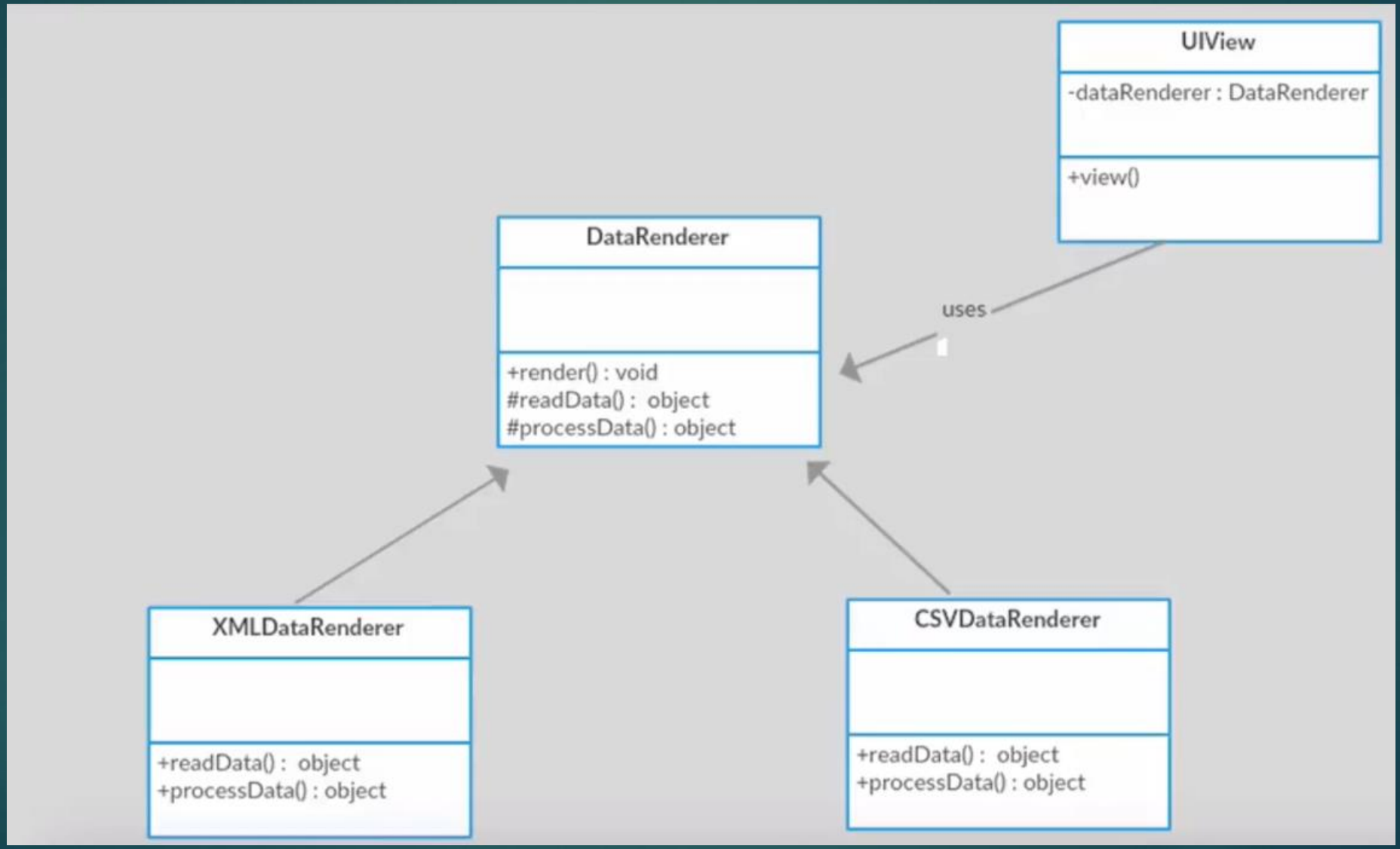
- **Problem:** Duplicate logic in `render()` for XML and CSV.

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).

By Eslam Khder

By Eslam Khder

By Eslam Khder

By Eslam Khder