

DESIGN PATTERN

112-1 Programming
Language I TA CLASS

CONTENTS



Introduction



Why should you try to use patterns



Types of Design Patterns



Factory Method



Conclusion

Introduction

Design patterns are solutions to recurring problems; **guidelines on how to tackle certain problems.**

They are not classes, packages or libraries that you can plug into your application and wait for the magic to happen. These are, rather, guidelines on how to tackle certain problems in certain situations.

Why should you try to use patterns



Accelerate the development process

The design pattern can accelerate the development process. It provides proven development paradigms, which helps save time without having to reinvent patterns every time a problem arises.

Facilitate Code Readability

Standardization related to the design pattern is also very useful to facilitate code readability.

Types of Design Patterns



Creational:

The design patterns that deal with the creation of an object.
Abstract Factory, Builder, Factory Method, Prototype, Singleton



Structural:

The design patterns in this category deals with the class structure such as Inheritance and Composition.
Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy



Behavioral:

This type of design patterns provide solution for the better interaction between objects, how to provide loose coupling, and flexibility, to extend easily in future.

Chain of Responsibility, Iterator, Memento, State, Template Method, Command, Mediator, Observer, Strategy, Visitor

Factory Method : Introduction

Aka. Virtual Constructor, Simple Factory, Static Factory Method.

Factory Method 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.



Factory Method : When to use Factory Method



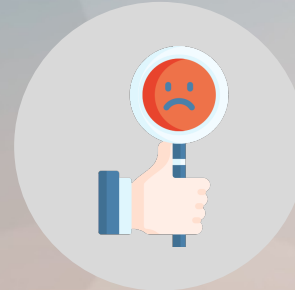
Useful when there is some generic processing in a class but the required sub-class is dynamically decided at runtime. Or putting it in other words, when the client does not know what exact sub-class it might need.

Factory Method : Pros & Cons



Pros

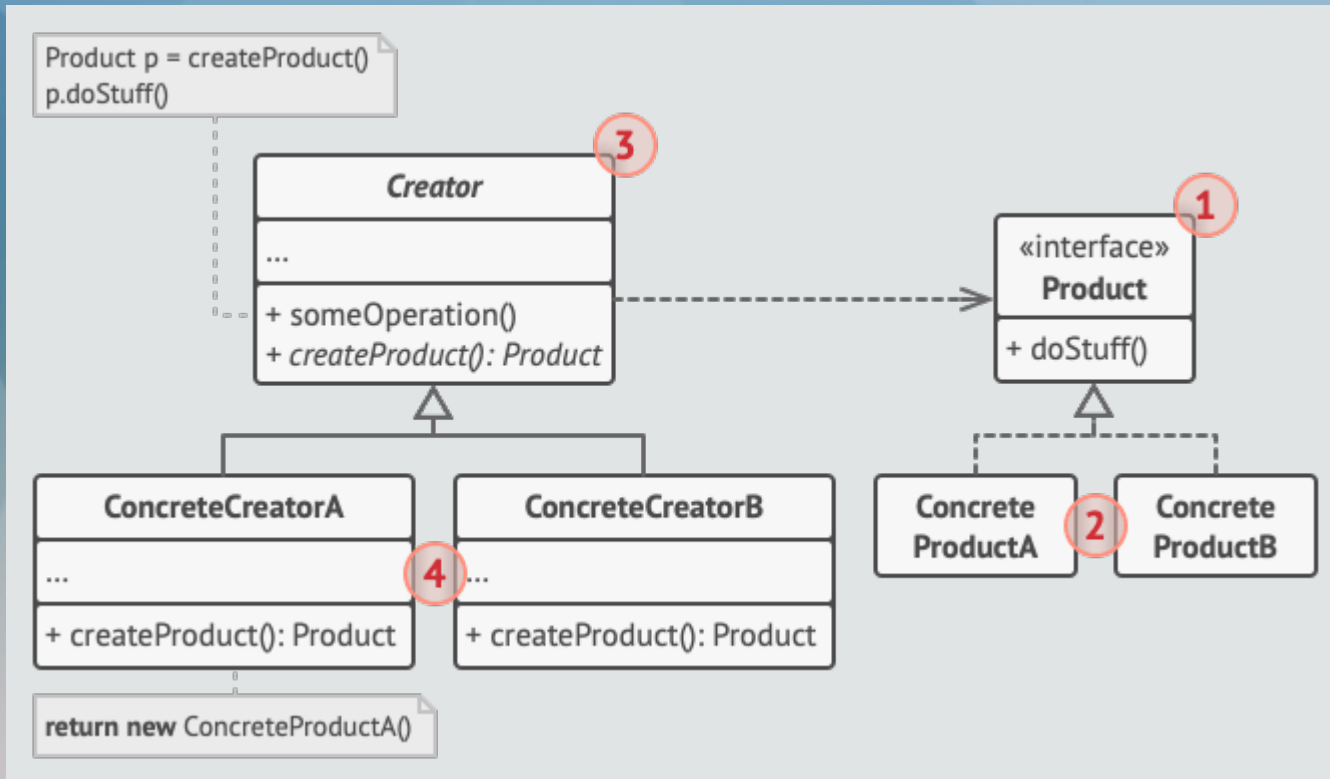
- Loose coupling
- Encapsulation
- Code reuse
- Extensibility



Cons

- Complexity
- Overhead

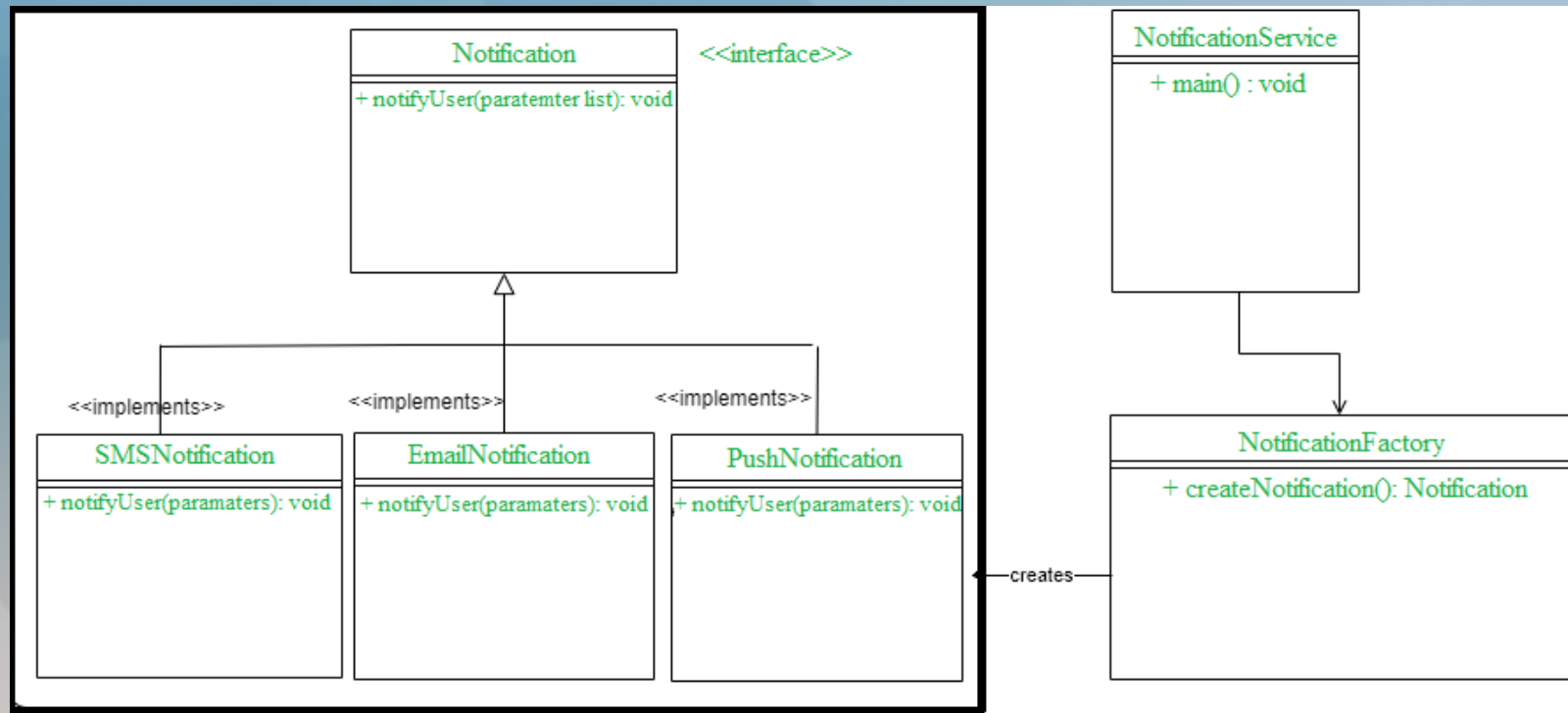
Factory Method : Structure



1. The Product declares the interface, which is common to all objects that can be produced by the creator and its subclasses.
2. Concrete Products are different implementations of the product interface.
3. The Creator class declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface.
4. Concrete Creators override the base factory method so it returns a different type of product.

Case : Problem Description

- Consider we want to implement a notification service through email, SMS, and push notifications. Let's try to implement this with the help of the factory method design pattern.
- First, we will design a UML class diagram for this.



- In the above class diagram, we have an interface called Notification, and three concrete classes are implementing the Notification interface. A factory class NotificationFactory is created to get a Notification object.

Case : Java Code Implementation

Create Notification interface

```
1 public interface Notification {  
2     void notifyUser();  
3 }
```

Create all implementation classes

```
public class PushNotification implements Notification {  
  
    @Override  
    public void notifyUser()  
    {  
        // TODO Auto-generated method stub  
        System.out.println("Sending a push notification");  
    }  
}
```

Case : Java Code Implementation

Create all implementation classes (cont.)

```
public class SMSNotification implements Notification {  
    @Override  
    public void notifyUser()  
    {  
        // TODO Auto-generated method stub  
        System.out.println("Sending an SMS notification");  
    }  
}
```

```
public class EmailNotification implements Notification {  
    @Override  
    public void notifyUser()  
    {  
        // TODO Auto-generated method stub  
        System.out.println("Sending an e-mail notification");  
    }  
}
```

Case : Java Code Implementation

Create a factory class NotificationFactory.java to instantiate concrete class

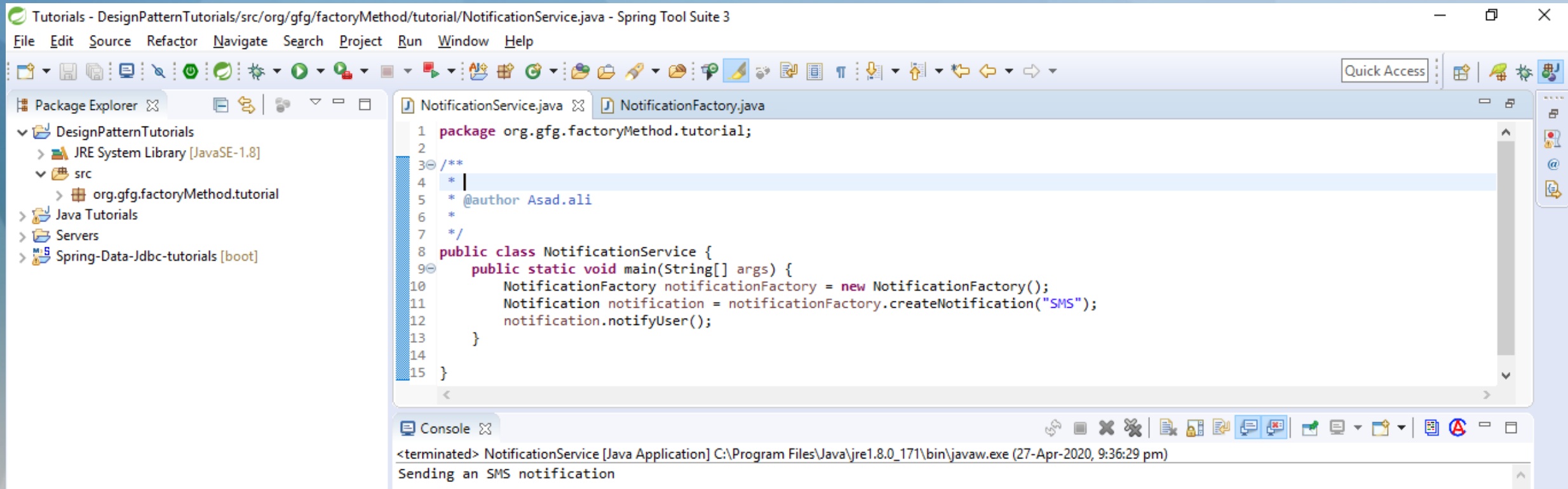
```
public class NotificationFactory {  
    public Notification createNotification(String channel)  
    {  
        if (channel == null || channel.isEmpty())  
            return null;  
        switch (channel) {  
            case "SMS":  
                return new SMSNotification();  
            case "EMAIL":  
                return new EmailNotification();  
            case "PUSH":  
                return new PushNotification();  
            default:  
                throw new IllegalArgumentException("Unknown channel "+channel);  
        }  
    }  
}
```


Case : Java Code Implementation

Now let's use factory class to create and get an object of concrete class by passing some information.

```
public class NotificationService {  
    public static void main(String[] args)  
    {  
        NotificationFactory notificationFactory = new NotificationFactory();  
        Notification notification = notificationFactory.createNotification("SMS");  
        notification.notifyUser();  
    }  
}
```

Case : Java Code Implementation



The screenshot displays the Spring Tool Suite 3 IDE. The Package Explorer on the left shows the project structure: DesignPatternTutorials > JRE System Library [JavaSE-1.8] > src > org.gfg.factoryMethod.tutorial. The main editor window shows the code for NotificationService.java. The code includes a package declaration, a Javadoc comment, and a public class NotificationService with a main method. The main method creates a NotificationFactory, creates an SMS notification, and calls notifyUser(). The Console at the bottom shows the output: <terminated> NotificationService [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (27-Apr-2020, 9:36:29 pm) Sending an SMS notification.

```
1 package org.gfg.factoryMethod.tutorial;
2
3 /**
4  *
5  * @author Asad.ali
6  *
7  */
8 public class NotificationService {
9     public static void main(String[] args) {
10         NotificationFactory notificationFactory = new NotificationFactory();
11         Notification notification = notificationFactory.createNotification("SMS");
12         notification.notifyUser();
13     }
14 }
15 }
```

<terminated> NotificationService [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (27-Apr-2020, 9:36:29 pm)
Sending an SMS notification

Conclusion



I

Design patterns are not a silver bullet to all your problems.



II

Do not try to force them; bad things are supposed to happen, if done so.



III

Keep in mind that design patterns are solutions to problems, not solutions finding problems; so don't overthink.



IV

If used in a correct place in a correct manner, they can prove to be a savior; or else they can result in a horrible mess of a code.

Reference

- [design-patterns-for-humans](#)
- [java-design-patterns](#)
- [awesome-design-patterns](#)
- [design-patterns](#)
- [design-patterns-java](#)
- [Design Patterns](#)
- [Design Patterns](#)

