UNIVERSIDADE POSITIVO

DESIGN PATTERNS

Prof Escobar

FACTORY METHOD

O **padrão Factory Method** é recomendado quando queremos delegar a responsabilidade de criação de objetos a subclasses, permitindo a elas definir qual classe concreta deve ser instanciada. Esse padrão é útil em cenários onde a criação de objetos precisa ser flexível ou onde queremos evitar o acoplamento direto entre classes.

Cenário Prático: Sistema de Notificações Multicanal

Imagine que você está desenvolvendo um sistema de notificações para uma empresa que envia alertas para seus clientes por diferentes canais: e-mail, SMS e push notifications. O formato da notificação e o comportamento específico de cada canal de comunicação podem variar, mas a interface básica para enviar uma notificação deve ser a mesma.

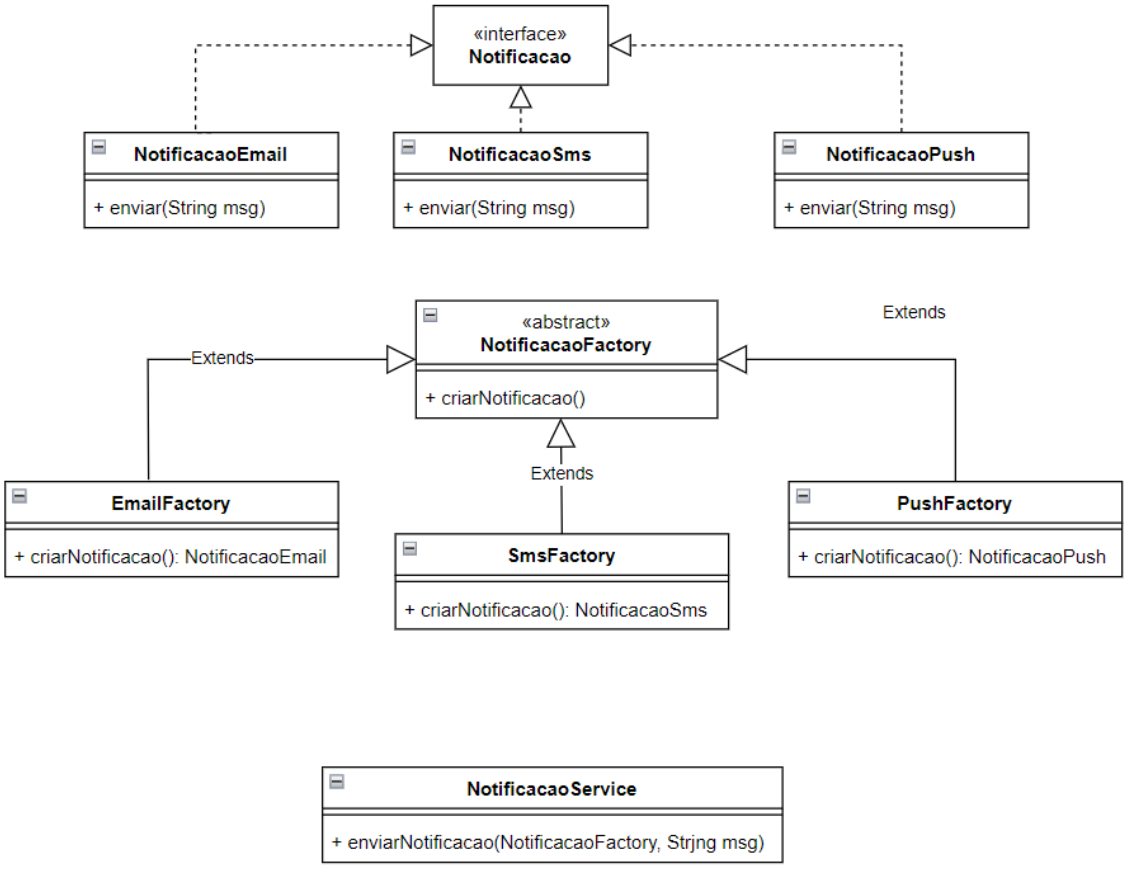
Problema

Se você criasse todas as instâncias de notificação diretamente no código, teria que lidar com um código altamente acoplado e difícil de manter, já que, para cada novo canal, precisaria modificar o código existente para suportá-lo.

Solução com Factory Method

Usar o **Factory Method** permite criar uma arquitetura flexível onde novos canais de notificação podem ser adicionados sem modificar o código existente.

Diagrama de Classes



Código em Java

Interface das notificações

```java
// Interface comum para todos os tipos de notificações
public interface Notificacao {
    void enviar(String mensagem);
}
```

Classes concretas para as notificações

```java
// Implementação de notificação por e-mail
public class NotificacaoEmail implements Notificacao {
    @Override
    public void enviar(String mensagem) {
        System.out.println("Enviando email: " + mensagem);
    }
}

// Implementação de notificação por SMS
public class NotificacaoSMS implements Notificacao {
    @Override
    public void enviar(String mensagem) {
        System.out.println("Enviando SMS: " + mensagem);
    }
}

// Implementação de notificação por Push Notification
public class NotificacaoPush implements Notificacao {
    @Override
    public void enviar(String mensagem) {
        System.out.println("Enviando Push Notification: " + mensagem);
    }
}
```

Classe abstrata para a factory

```java
// Criador (Factory Method)
public abstract class NotificacaoFactory {
    public abstract Notificacao criarNotificacao();
}
```

Implementação das *factories* concretas

```java
// Implementação concreta para notificação por e-mail
public class EmailFactory extends NotificacaoFactory {
    @Override
    public Notificacao criarNotificacao() {
        return new NotificacaoEmail();
    }
}

// Implementação concreta para notificação por SMS
public class SMSFactory extends NotificacaoFactory {
    @Override
    public Notificacao criarNotificacao() {
        return new NotificacaoSMS();
    }
}

// Implementação concreta para notificação por Push
public class PushFactory extends NotificacaoFactory {
    @Override
    public Notificacao criarNotificacao() {
        return new NotificacaoPush();
    }
}
```

Classe agente para envio das notificações

```java
// Cliente usa a factory para criar a notificação desejada
public class NotificacaoService {
    public void enviarNotificacao(NotificacaoFactory factory, String mensagem) {
        Notificacao notificacao = factory.criarNotificacao();
        notificacao.enviar(mensagem);
    }
}
```

Exemplo de uso/consumo

```java
public class Main {
    public static void main(String[] args) {
        NotificacaoService service = new NotificacaoService();

        // Enviar email
        service.enviarNotificacao(new EmailFactory(), "Bem-vindo ao nosso sistema!");

        // Enviar SMS
        service.enviarNotificacao(new SMSFactory(), "Seu código de verificação é 1234");

        // Enviar Push Notification
        service.enviarNotificacao(new PushFactory(), "Você tem uma nova mensagem!");
    }
}
```

EXERCÍCIO

Sistema de Pagamentos Multiplataforma

Você foi contratado para desenvolver um Sistema de Pagamento para uma plataforma de e-commerce que oferece diferentes métodos de pagamento (cartão de crédito, PayPal, criptomoedas). A empresa precisa de uma arquitetura que permita fácil extensão para novos métodos de pagamento no futuro, além de uma validação específica para cada tipo de transação. A implementação deve garantir a flexibilidade necessária para novos métodos de pagamento sem alterar a lógica do sistema.

Requisitos:

1. O sistema deve permitir o registro de novos métodos de pagamento sem alterar a lógica existente.

2. Cada método de pagamento tem uma lógica de validação específica:

- **Cartão de Crédito**: Verificar se o número do cartão é válido (baseado em um algoritmo fictício).

- **PayPal**: Verificar se a conta PayPal está vinculada ao e-mail do usuário.

- **Criptomoedas**: Verificar se o saldo disponível na carteira digital é suficiente.

3. O sistema deve permitir que novas formas de pagamento sejam facilmente adicionadas no futuro.

4. O sistema deve ser projetado de forma que o cliente (código que utiliza os métodos de pagamento) nunca precise se preocupar com o tipo específico de pagamento.

5. O método de pagamento deve ser escolhido dinamicamente com base na escolha do usuário.

Regras de Negócio:

- Todos os métodos de pagamento devem implementar uma interface comum *Pagamento* que define o método *processarPagamento*.

- O método *processarPagamento* recebe o valor da transação e retorna uma confirmação (ou erro) ao cliente.

- Cada método de pagamento tem seu próprio processo de validação antes de confirmar o pagamento.

- O sistema deve conter um **Factory Method** para criar as instâncias dos métodos de pagamento.

Estrutura de Classes:

- Interface *Pagamento*: Interface que define o método processarPagamento.

- Classes concretas: *PagamentoCartaoCredito*, *PagamentoPayPal*, *PagamentoCriptomoeda*.

- Factory: Classes concretas de factory para cada método de pagamento: *FactoryCartaoCredito*, *FactoryPayPal*, *FactoryCriptomoeda*.

- Cliente (PagamentoService): Serviço responsável por processar o pagamento utilizando o Factory Method.

Descrição da Tarefa:

1. Implemente a interface Pagamento:

- O método processarPagamento recebe o valor da transação e deve retornar uma mensagem de confirmação ou erro.

2. Crie as classes concretas:

- **PagamentoCartaoCredito**:

  - Deve validar se o número do cartão tem 16 dígitos.

- **PagamentoPayPal**:

  - Deve validar se o e-mail está vinculado a uma conta PayPal (use um método fictício para simular essa validação).

- **PagamentoCriptomoeda**:

  - Deve verificar se a carteira digital tem saldo suficiente para cobrir o valor da transação.

3. Implemente o Factory Method:

- Crie uma classe abstrata *PagamentoFactory* que tenha o método abstrato *criarPagamento*.

- Crie classes concretas de factory (*FactoryCartaoCredito*, *FactoryPayPal*, *FactoryCriptomoeda*) que sobrescrevem o método criarPagamento e retornam a instância correspondente.

4. Crie a classe *PagamentoService*:

- A classe *PagamentoService* deve usar o factory apropriado para processar o pagamento sem saber qual implementação específica está sendo usada.

- Ela deve escolher o método de pagamento com base na escolha do usuário (dinamicamente).

Requisitos de design:

- O sistema deve ser flexível e extensível para novos métodos de pagamento no futuro.

- O cliente nunca deve instanciar diretamente os objetos de pagamento.

- Use o Factory Method para encapsular a lógica de criação dos métodos de pagamento.

Exemplo de Uso:

Requisitos Técnicos:

1. Implemente validações realistas para cada método de pagamento.

2. Crie uma arquitetura robusta que permita fácil adição de novos métodos.

3. Certifique-se de que o cliente (PagamentoService) não depende das implementações concretas de pagamento.

Extensão do Exercício:

- Após a implementação, adicione um novo método de pagamento, por exemplo, ***Apple Pay***, sem modificar a lógica de negócio existente. Crie a classe *PagamentoApplePay* e uma nova factory *FactoryApplePay*.