Departamento de Computação – DComp - IFMA
Padrões de Software – 7º Período
Entrega: 20-08-2021

Atividade Etapa 02 – Parte 01

Princípios de Design Orientando a Objetos

Dica: nesse link, existem alguns exemplos de outros padrões que são chamados de Factory. Veja nesse link uma excelente explicação:

https://refactoring.guru/design-patterns/factory-comparison

Preâmbulo

O Factory é empregado quando várias atividades estão envolvidas na criação da instância desejada como a definição de valores iniciais, por exemplo. Nestes casos, em vez de exigir que a decisão seja do conhecimento de todo e qualquer cliente, assim como detalhes da montagem de um objeto válido, tais operações são encapsuladas em um Factory (Method). Este padrão retorna uma instância de uma classe, dentre um conjunto de classes possíveis. A escolha da classe é baseada em informações fornecidas ao padrão. Em geral, todas as classes das quais uma instância pode ser gerada possuem uma classe ancestral comum ou implementam uma interface comum.

1. **Defina o padrão *Factory Method*.**

2. **Como se dá a implementação deste padrão? Descreva a terminologia e estrutura (UML).**

3. **Exercícios de Fixação para consolidar o aprendizado**

   OBS: É permitida a adaptação do código para outra linguagem

   1. *Crie um projeto Java chamado FactoryMethod em sua IDE favorita*

2. *Defina uma interface Emissor*

```
public interface Emissor {
  void envia(String mensagem);
}
```

3. *Defina as classes EmissorSMS, EmissorEmail e EmissorJMS que irão implementar a interface Emissor.*

```
public class EmissorSMS implements Emissor {
  public void envia(String message) {
    System.out.println("Enviando por SMS a mensagem: ");
    System.out.println(message);
  }
}
```

```
public class EmissorEmail implements Emissor {
  public void envia(String message) {
    System.out.println("Enviando por email a mensagem: ");
    System.out.println(message);
  }
}
```

```
public class EmissorJMS implements Emissor {
  public void envia(String message) {
    System.out.println("Enviando por JMS a mensagem: ");
    System.out.println(message);
  }
}
```

4. *Crie uma classe para testar os emissores.*

```
public class TestaEmissores {
  public static void main(String[] args) {
    Emissor emissor1 = new EmissorSMS();
    emissor1.envia("K19 Treinamentos");

    Emissor emissor2 = new EmissorEmail();
    emissor2.envia("K19 Treinamentos");

    Emissor emissor3 = new EmissorJMS();
    emissor3.envia("K19 Treinamentos");
  }
}
```

5. *Para tornar a classe TestaEmissores menos dependente das classes que implementam os mecanismos de envio, podemos definir uma classe intermediária que será responsável pela criação dos emissores.*

```
public class EmissorCreator {
  public static final int SMS = 0;
  public static final int EMAIL = 1;
  public static final int JMS = 2;

  public Emissor create(int tipoDeEmissor) {
    if(tipoDeEmissor == EmissorCreator.SMS) {
      return new EmissorSMS();
    } else if (tipoDeEmissor == EmissorCreator.EMAIL) {
      return new EmissorEmail();
    } else if (tipoDeEmissor == EmissorCreator.JMS) {
      return new EmissorJMS();
    } else {
      throw new IllegalArgumentException("Tipo de emissor não suportado");
    }
  }
}
```

6. *Altere a classe TestaEmissores para utilizar a classe EmissorCreator.*

```
public class TestaEmissores {
  public static void main(String[] args) {
    EmissorCreator creator = new EmissorCreator();

    //SMS
    Emissor emissor1 = creator.create(EmissorCreator.SMS);
    emissor1.envia("K19 Treinamentos");

    //Email
    Emissor emissor2 = creator.create(EmissorCreator.EMAIL);
    emissor2.envia("K19 Treinamentos");

    //JMS
    Emissor emissor3 = creator.create(EmissorCreator.JMS);
    emissor3.envia("K19 Treinamentos");
  }
}
```

Comente o que você achou dessa solução.

4. A classe ImpostoDeRenda possui o método calcula() que utiliza um de dois métodos para cálculo do imposto de renda: ou o simplificado ou completo, implementados pelas classes Simplificado e Completo, respectivamente. Estas duas classes são derivadas da classe MetodoCalculo, que é abstrata e possui um único método que, à semelhança da classe ImpostoDeRenda, também é calcula(). O método calcula() da classe ImpostoDeRenda deve fazer uso de uma implementação de MetodoCalculo, sem exatamente conhecer quem implementa o método, ou seja, se a classe Simplificado ou a classe Completo. A decisão de qual método empregar é definida em tempo de execução, quando uma instância da classe ImpostoSimplificado ou da classe ImpostoCompleto, ambas derivadas de ImpostoDeRenda, é criada. Código cliente da classe ImpostoDeRenda deverá empregar uma instância de uma das classes que implementa ImpostoDeRenda sem conhecer realmente quem implementa esta classe abstrata. A decisão de qual classe utilizar é decidida no método

newImpostoDeRenda() da classe ImpostoDeRendaFactory. Este método (pode ser estático) retorna uma instância de ImpostoDeRenda. Estabeleça a decisão de qual classe empregar da forma que considerar apropriado. Observe que a instância retornada deverá estar apta a receber a mensagem calcula() cuja operação deverá ser consistente com a classe da instância criada.