Alunos: Caio Buard;Leonardo;Gabriel Cegantini;Reinaldo Vieira;

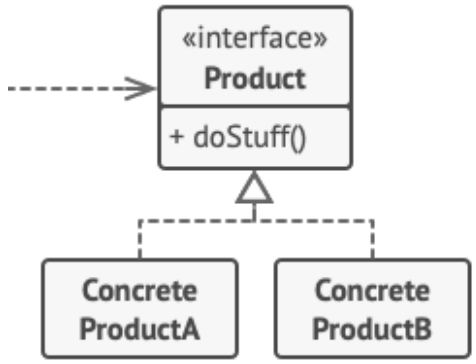
ATIVIDADE - Engenharia de software moderna

Criação

1- Factory Method

O Factory Method é um interface que fornece a criação de objetos em uma superclasse e a alteração em uma subclasse. Ele também substitui chamadas diretas para chamadas de método fábrica especial, os objetos ainda são criados pelo operador **new.**

No método fábrica o código utilizado não tem diferença nos produtos reais retornado para as subclasses, todos os produtos são tratados como abstratos pelo cliente.



Após passar pelo Creator o produto declara a interface, isso é como sobre os objetos que podem ser criados pela subclasse. Os produtos concretos são implementações diferentes da interface.

Dentro do código tem a classe Creator, ele declara o método fábrica que faz o retorno de novos objetos do produto, o tipo de retorno é referente ao tipo de interface do produto. Os criadores concretos subscrevem o método fábrica para retornar um tipo de produto diferente.

2- Singleton

O método de Singleton é um padrão de projeto criacional que permite a classe ter apenas uma instância provendo pontos de acesso global a ela. Ele resolve esses dois problemas, garante que a classe tenha apenas uma instância e permite acesso global a esta instância.

O singleton possui um construtor privado, o que significa que a instância única da classe só pode ser criada dentro da própria classe. A instância única é geralmente fornecida como uma variável estática dentro da própria classe, e o acesso a essa instância é feito por meio de um método estático.

```java
class Logger {
  private Logger() {} // proíbe clientes de chamar new Logger()

  private static Logger instance; // instância única

  public static Logger getInstance() {
    if (instance == null) // 1a vez que chama-se getInstance
      instance = new Logger();
    return instance;
  }

  public void println(String msg) {
    // registra msg na console, mas poderia ser em um arquivo
    System.out.println(msg);
  }

}
```

Nessa parte do código do singleton o privete logger é declarado privado , isso faz com que ele não possa ser acessado fora da classe logger impedindo que outras classes criem instâncias. A instância única do logger é armazenada como uma variável estática privada. Ela é declarada como static para que seja compartilhada por todas as instâncias da classe. Esse método static é responsável por fornecer a instância única.

```java
class Main {

  void teste () {
    // Logger = new Logger(); => daria um erro de compilação
  }

  void f() {
    Logger log = Logger.getInstance();
    log.println("Executando f " + log);
  }

  void g() {
    Logger log = Logger.getInstance();
    log.println("Executando g " + log);
  }

  void h() {
    Logger log = Logger.getInstance();
    log.println("Executando h " + log);
  }
```

Nessa parte do código de Singleton tem a função de mostrar como usar a classe logger de maneira consistente, garantindo que você obtenha a mesma instância única em todo o código.

Estrutura

1 Class Adapter

O padrão de *adapter* tem como objetivo fazer com que uma classe ou tipos de dados diferentes consigam trabalhar e comunicar entre si. Imagine que todo o fluxo do seu sistema funciona através de arquivos XML, mas ao adicionar uma nova biblioteca que as interações dela são feitas

somente em JSON, nesses casos podemos usar um adapter, para converter XML em JSON toda vez que usarmos essa biblioteca.
Veja esse exemplo de Adapter para um código que precisa trabalhar com vários tipos de projetores(como se fosse um controle universal)

```java
/**
 * Classe concreta, representando um projetor da Samsung
 */
class ProjetorSamsung {

  public void turnOn() {
    System.out.println("Ligando projetor da Samsung");
  }

}


/**
 * Classe concreta, representando um projetor da LG
 */
class ProjetorLG {

  public void enable(int timer) {
    System.out.println("Ligando projetor da LG em " + timer + " minutos");
  }

}
```

Pense que ProjetorLG e ProjetorSamsung são API's que não podem ser modificadas(pois são feitas de maneira privada), veja que nas classes desses projetores existem métodos que em tese fazem a mesma coisa, mas possuem nomes diferentes.

Para facilitar nosso trabalho criamos uma interface Projetor, que vai abstrair o conceito de projetor para criarmos uma função ligar() que vai conseguir ligar qualquer tipo de projetor.

```java
/**
 * Interface para "abstrair" o tipo de projetor (Samsung ou LG)
 */
interface Projetor {
  void liga();
}
```

Criamos classes de adaptador para de fato realizar as traduções das funções de ligar,

```java
/**
* Adaptador de ProjetorSamsung para Projetor
* Um objeto da classe a seguir é um Projetor (pois implementa essa interface),
* mas internamente repassa toda chamada de método para o objeto adaptado
* (no caso, um ProjetorSamssung)
*/
class AdaptadorProjetorSamsung implements Projetor {

  private ProjetorSamsung projetor;

  AdaptadorProjetorSamsung (ProjetorSamsung projetor) {
    this.projetor = projetor;
  }

  public void liga() {
    projetor.turnOn(); // chama método do objeto adaptado (ProjetorSamsung)
  }

}
```

```java
/**
* Idem classe anterior, mas agora adaptando ProjetoLG para Projetor
*/
class AdaptadorProjetorLG implements Projetor {

  private ProjetorLG projetor;

  AdaptadorProjetorLG (ProjetorLG projetor) {
    this.projetor = projetor;
  }

  public void liga() {
    projetor.enable(0); // chama método de objeto adaptado (ProjetorLG)
  }

}
```

2 Decorador

O design pattern de decoradores tem como objetivo “decorar” algum algoritmo, ou seja, adicionar mais funcionalidades antes ou depois dele.

Esse efeito pode ser atingido criando uma interface que abstrai os métodos do objetivo final.

```java
/*
 * Decoradores implementam sempre a interface Channel
 */
interface Channel {
  void send(String msg);
  String receive();
}
```

Uma classe final, que vai realizar a operação, nessa classe final é necessário que ela seja implementada da classe abstrata.

```java
/**
 * TCPChannel é uma canal de comunicação concreto (que usa protocolo TCP)
 * E, por isso, é a classe final de uma cadeia de decoradores
 */
class TCPChannel implements Channel {

  public void send(String m) {
    System.out.println("Canal concreto (TCP) enviando > " + m);
  }

  public String receive() {
    System.out.println("Canal concreto (TCP) recebendo...");
    return "José";
  }

}
```

É preciso também criar uma classe que estende a interface, que vai definir todos os decoradores.

```java
/**
 * Todos os decoradores são subclasse de ChannelDecorator
 */
class ChannelDecorator implements Channel {
  protected Channel channel;

  public ChannelDecorator(Channel channel) {
    this.channel = channel;
  }

  public void send(String m) {
    channel.send(m);
  }

  public String receive() {
    return channel.receive();
  }

}
```

E por final, você pode criar os decoradores de fato que vão realizar os comportamentos alterados na classe final(neste exemplo a TCPChannel).

```java
/**
 * Decorador que:
 *  - compacta mensagens antes de enviar (send)
 *  - descompacta mensagens depois de receber (receiver)
 */
class ZipChannel extends ChannelDecorator {

    public ZipChannel(Channel c) {
      super(c);
    }

    public void send(String m)  {
      System.out.println("Decorador compactando > " + m);
      super.send(m);
    }

    public String receive()  {
      String m = super.receive();
      System.out.println("Decorador descompactando < " + m);
      return m;
    }

}
```

Nesse exemplo passado o fluxo de código ficaria da seguinte maneira:
ZipChannel(compacta a requisição neste exemplo) -> TCPChannel(envia a requisição).
Esse comportamento acontece porque a classe ChannelDecorator é um intermediário entre a classe decoradora e a classe principal que faz a operação “decoradora” e a operação final.

```java
public class Main {

  public static void main(String args[]) {
    Channel c = new ZipChannel(new TCPChannel());
    c.send("Qual seu nome?");
    String r = c.receive();
    System.out.println(r);
  }
```

Se quisermos podemos criar uma outra classe decoradora por exemplo: Md5Channel extends ChannelDecorator

Ao chamar a função send() usando essa nova classe decoradora, poderíamos configurar para enviar com hashes md5 por exemplo dentro de TCPChannel.

Em conclusão, o design pattern de decorador é ótimo para criar diferentes fluxos de alteração e de adição de comportamento, ajudando a deixar o fluxo expansível se possível. Sua desvantagem pode

ser a questão do código criar uma complexidade maior, então não se faz necessário usar este design pattern para todas as funcionalidades de um sistema, por exemplo.

Comportamento

1- Template Method

O método de template é um dos mais fáceis de compreender, pois não gera uma base de código muito complexa, e possui uma hierarquia e possibilidade de nomenclatura simples. Uma das maneiras de realizar este *pattern* é através de uma classe abstrata que cria uma fundação para outras classes:

```java
/**
 * Classe que implementa um Template Method (calcSalarioLiquido)
 * Veja que essa classe é abstrata
 */
abstract class Funcionario {

    protected double salario;

    public Funcionario(double salario) {
      this.salario = salario;
    }

    abstract double calcDescontosPrevidencia();
    abstract double calcDescontosPlanoSaude();
    abstract double calcOutrosDescontos();

    /**
    * Template Method: define o esqueleto de um algoritmo
    * Ele ainda é um "template" porque os métodos chamados são abstratos
    */
    public double calcSalarioLiquido() {
      double prev = calcDescontosPrevidencia();
      double saude = calcDescontosPlanoSaude();
      double outros = calcOutrosDescontos();
      return salario - prev - saude - outros;
    }
}
```

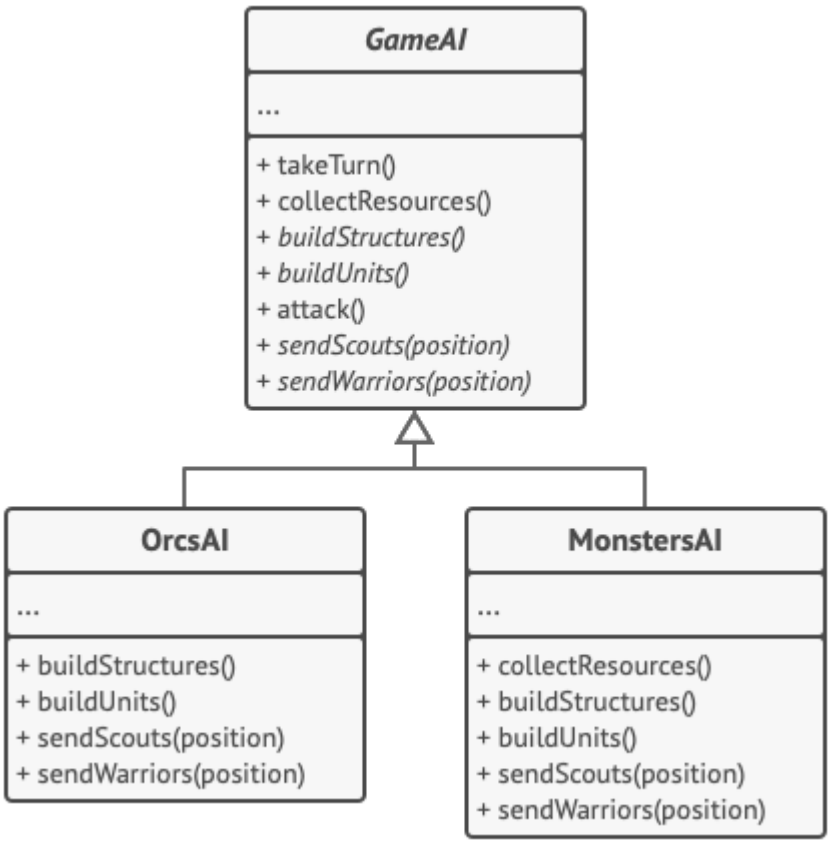
Nesse caso é criado uma classe abstrata Funcionario, e nela já existem alguns métodos e propriedades, importante ressaltar que como Funcionario é uma classe abstrata **ela não pode ser instanciada**, é preciso uma outra classe estender ela para utiliza-la. Observe também que são declaradas 3 classes abstratas que são utilizadas para o cálculo do salário líquido, isso se dá porque ao estender Funcionario pode ser que o programador queira especificar diferentes valores para estes

tipos de funcionários.

```java
/**
 * Subclasse que implementa os métodos abstratos chamados pelo Template Method
 * Ela vai herdar o template method (calcSalarioLiquido)
 * Mas vai ter que implementar os métodos abstrados chamados por ele
 */
class FuncionarioCLT extends Funcionario {

    public FuncionarioCLT(double salario) {
        super(salario);
    }

    // implementa método abstrato
    double calcDescontosPrevidencia() {
        return salario * 0.1;    // somente um exemplo
    }

    // implementa método abstrato
    double calcDescontosPlanoSaude() {
        return 100.0;
    }

    // implementa método abstrato
    double calcOutrosDescontos() {
      return 20.0;
    }

}
```

Veja que agora a classe FuncionarioCLT toma vantagem da classe Funcionario, além disso ela utiliza das funções calcDescontosPrevidencia(), calcDescontosPlanoSaude(), calcOutrosDescontos() para definir seus próprios descontos em cima do cálculo de salário.

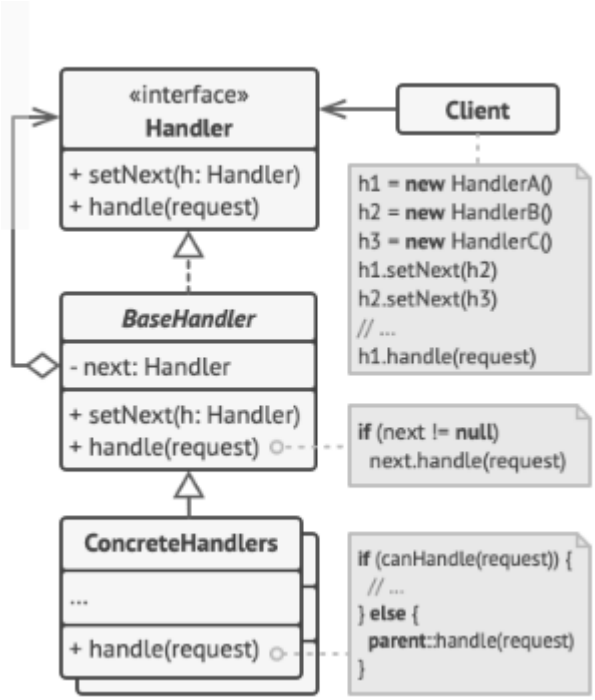
Em suma, o *Template Method* possui uma estrutura de hierarquia clara, onde existe um modelo e classes que estendem esse modelo, veja exemplo na imagem abaixo.



2 - Chain of Responsibility

O Chain of responsibility é um padrão comportamental que cria uma cadeia de objetos que podem lidar com solicitações de maneira sequencial. Cada objeto na cadeia decide se processa a solicitação ou a passa para o próximo objeto na cadeia. Isso permite que vários objetos manipulem a solicitação sem a necessidade de um acoplamento explícito entre o remetente da solicitação e seus manipuladores.

Esse padrão se baseia em transformar certos comportamentos em objetos solitários chamados handlers. Ele liga esses handlers em uma corrente. Cada handler ligado tem um campo para armazenar uma referência ao próximo handler da corrente.



A interface é declarada pelos handlers. Ele geralmente contém apenas um único método para lidar com pedidos. A classe opcional de handlers base é onde você pode colocar o código padrão, essa classe define um campo para armazenar uma referência para o próximo handler.

O código real fica nos handlers concretos, eles recebem os pedidos e cada handlers decide processar o pedido. O Cliente pode compor correntes apenas uma vez ou compô-las dinamicamente, dependendo da lógica da aplicação.