Factory Method e Chain of Responsibility

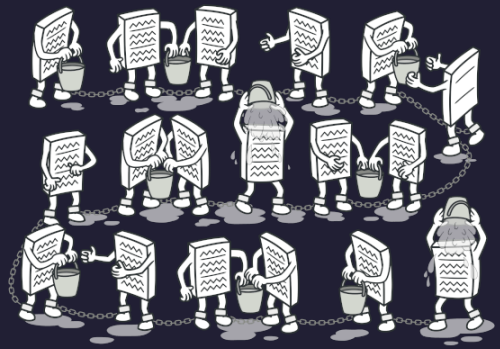
Alunos: Matheus Baron Lauritzen, Gustavo Baron Lauritzen, Gabriel Bosio e Eduardo da Rocha Weber

01

Chain of Responsibility

Introdução

O padrão de projeto Chain of Responsibility é um padrão comportamental que permite tratar solicitações de forma flexível, desacoplada e escalável. Ele é baseado em alguns princípios subjacentes e possui características específicas que o tornam útil em diversas situações.



Princípios Subjacentes

- Responsabilidade Única;
- Encapsulamento.

Características

- Cada objeto recebe apenas as características necessárias para a aplicação do seu tratamento;
- A solução é percorrida de forma sequencial até que um objeto consiga realizar o tratamento ou a corrente termine.
- Os objetos podem ser colocados, retirados ou trocados de posição na corrente com base na necessidade da aplicação.

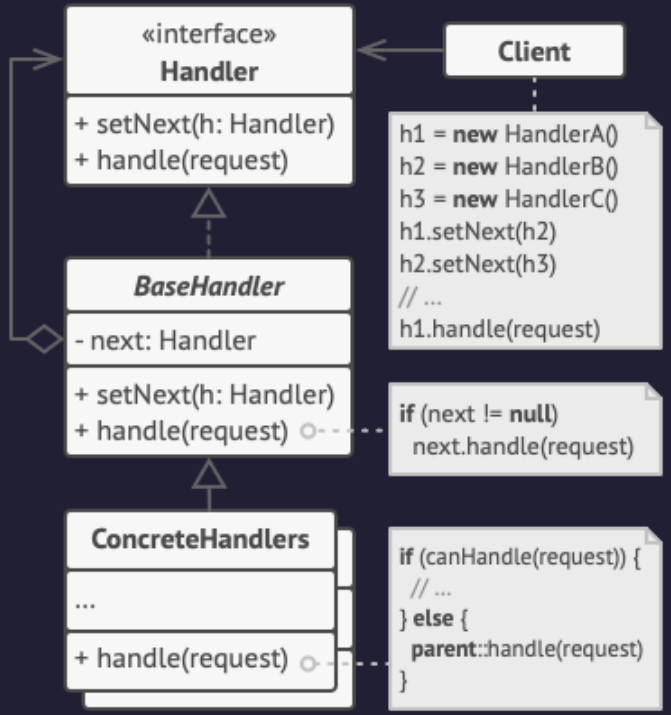
Quando aplicar

- O remetente de uma solicitação não precisa saber quais objetos irão tratá-la.
- Diferentes objetos podem tratar a solicitação, mas o objeto correto não é conhecido antecipadamente.
- É necessário adicionar ou alterar dinamicamente a lógica de tratamento de solicitações.
- A ordem de tratamento ou o conjunto de objetos que tratam a solicitação podem variar.

Benefícios

- Flexibilidade;
- Escalabilidade;
- Desacoplamento;
- Manutenibilidade;
- Reutilização.

Estrutura Genérica em UML



02

Factory Method

Introdução

O padrão Factory Method é amplamente utilizado na programação orientada a objetos para resolver o problema de criação de objetos em um sistema. Ele é útil quando um sistema precisa criar objetos de diferentes tipos, mas o código do cliente não deve depender das classes concretas desses objetos.

Princípios Subjacentes

- Abstração;
- Polimorfismo;
- Herança;
- Divisão por responsabilidade.

Características

- Definição de uma interface ou classe abstrata;
- As classes concretas que implementam essa interface ou herdam dessa classe abstrata são as fábricas reais que fornecem a implementação do método de fábrica;
- Cada fábrica concreta pode criar um tipo específico de objeto, mas todos os objetos criados devem seguir a mesma interface comum;
- O código cliente pode trabalhar com qualquer objeto criado pela fábrica, sem se preocupar com sua classe concreta.

Quando aplicar

- O código do cliente precisa trabalhar com múltiplas implementações de uma mesma interface, sem conhecer antecipadamente as classes concretas;
- Não souber de antemão os tipos e dependências exatas dos objetos com os quais seu código deve funcionar;
- É necessário adicionar novos tipos de objetos no futuro, sem modificar o código existente;

Benefícios

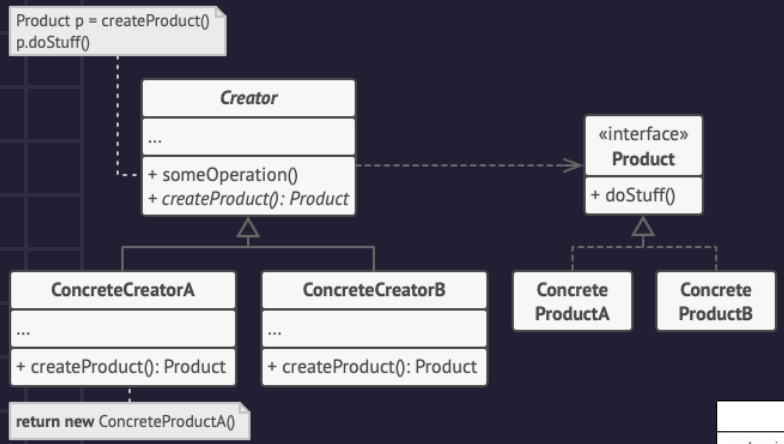
- Separação de responsabilidades;
- Flexibilidade;
- Escalabilidade;
- Reutilização de código.

Implementação

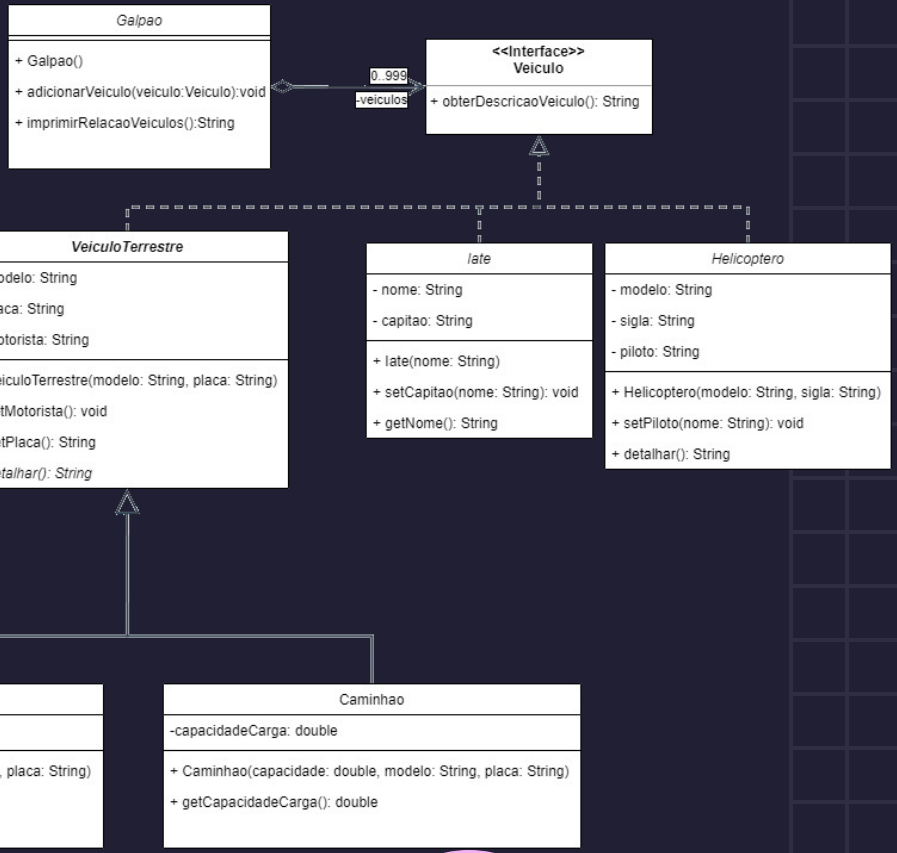
Factory Method...

Diagrama UML

Genérico



Exemplificado



Código da Implementação

```java
package dominio;

public interface Veiculo {
    String obterDescricaoVeiculo();
}
```

```java
package dominio;

public abstract class VeiculoTerrestre implements Veiculo {
    protected String modelo;
    protected String placa;
    protected String motorista;

    public VeiculoTerrestre(String modelo, String placa){
        this.modelo = modelo;
        this.placa = placa;
    }

    public void setMotorista(String nome){
        this.motorista = nome;
    }

    public String getPlaca(){
        return this.placa;
    }

    abstract public String detalhar();

    @Override
    public String obterDescricaoVeiculo(){
        return this.detalhar();
    }
}
```

Código da Implementação

```java
package dominio;

public class Carro extends VeiculoTerrestre {
    private double potencia;

    public Carro(double potencia, String modelo, String placa){
        super(modelo, placa);
        this.potencia = potencia;
    }

    public String detalhar(){
        return modelo + " " + placa + " " + potencia + " " + motorista;
    }
}
```

```java
package dominio;

public class Caminhao extends VeiculoTerrestre{
    private double capacidadeCarga;

    public Caminhao(double capacidade, String modelo, String placa){
        super(modelo, placa);
        this.capacidadeCarga = capacidade;
    }

    @Override
    public String detalhar(){
        return modelo + " " + placa + " " + capacidadeCarga + " " + motorista;
    }
}
```

Código da Implementação

```java
package dominio;

public class late implements Veiculo {
  private String nome;
  private String capitao;

   public late(String nome){
     this.nome = nome;
   }

   public void setCapitao(String capitao){
     this.capitao = capitao;
   }

  @Override
   public String obterDescricaoVeiculo(){
     return nome + " " + capitao;
   }
}
```

```java
package dominio;

public class Helicoptero implements Veiculo{
   private String modelo;
   private String sigla;
   private String piloto;

   public Helicoptero(String modelo, String sigla){
     this.modelo = modelo;
     this.sigla = sigla;
   }

   public void setPiloto(String piloto){
     this.piloto = piloto;
   }
   public String detalhar(){
     return modelo + " " + sigla + " " + piloto;
   }

   @Override
   public String obterDescricaoVeiculo(){
     return this.detalhar();
   }
}
```

Código da Implementação

```
package dominio;

public class Galpao {
    private Veiculo[ ] veiculos;
    private int contadorVeiculo = 0;

    public Galpao(){
        this.veiculos = new Veiculo[999];
    }
    public void adicionarVeiculo(Veiculo veiculo){
        this.veiculos[contadorVeiculo++] = veiculo;
    }
    public String imprimirRelacaoVeiculos(){
        String aux = "";
        for (int i = 0; i < contadorVeiculo; i++)
            aux += veiculos[i].obterDescricaoVeiculo() + "\n";
        return aux;
    }
}
```

Fontes

- https://refactoring.guru/pt-br/design-patterns/factory-method;
- https://www.tutorialspoint.com/design_pattern/factory_pattern.htm;
- https://refactoring.guru/pt-br/design-patterns/chain-of-responsibility.