

Universidade Tecnológica Federal do Paraná

Engenharia de Software - Curso de Arquitetura de Software (AS27S)

INSTRUTOR: Prof. Dr. Gustavo Santos

Ibanez Fernandes da Silva Junior, 2033500

CCH - Design Patterns Bridge

Problema

Imagine que você está desenvolvendo um sistema de gerenciamento de entregas, onde diferentes tipos de entregas podem estar em diferentes estados (pendente, em separação, enviada, finalizada) e podem ser processadas por diferentes transportadoras (Correios, Transportadora SuperSul, Transportadora SuperNorte). No entanto, você percebe que o código atual está fortemente acoplado, pois cada classe de entrega está diretamente dependente de uma transportadora específica. Isso torna difícil adicionar novos tipos de entregas ou transportadoras, bem como alterar o comportamento das entregas existentes sem modificar as classes existentes.

Descrição da Solução

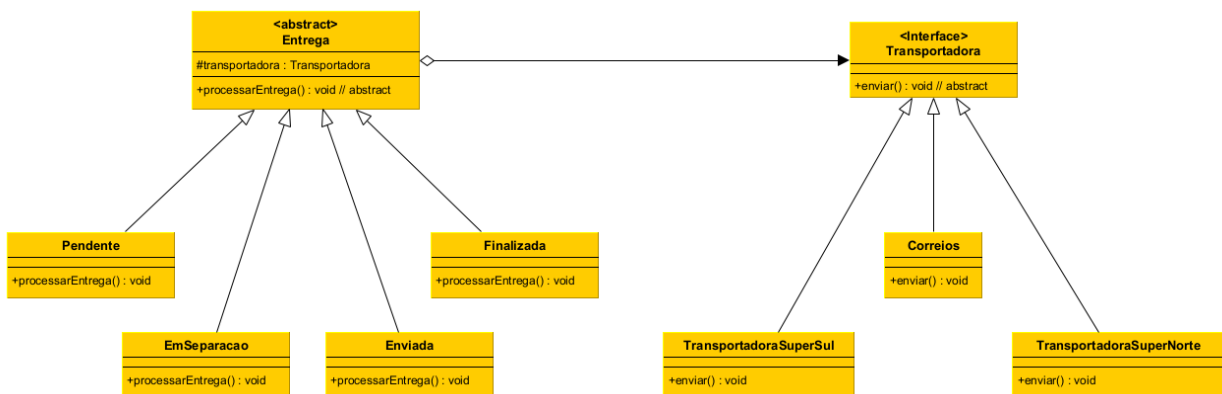
Para resolver esse problema, podemos aplicar o padrão de projeto Bridge. Vamos separar a abstração (classe abstrata **Entrega**) da implementação (classe abstrata **Transportadora**). Agora, a classe **Entrega** terá uma referência para a classe abstrata **Transportadora**, permitindo que possamos variar independentemente as subclasses de **Entrega** e de **Transportadora**.

Dessa forma, cada subclasse de **Entrega** representa um estado de entrega específico, enquanto cada subclasse de **Transportadora** representa uma transportadora específica. A classe **Entrega** invocará o método **enviar()** da classe **Transportadora**, independentemente da transportadora específica.

Com essa separação, podemos adicionar novos estados de entrega ou transportadoras sem a necessidade de modificar as classes existentes, tornando o sistema mais flexível e extensível.

A aplicação do padrão Bridge nos permite desacoplar a abstração da implementação, facilitando a manutenção, extensão e evolução do sistema de gerenciamento de entregas.

Visão Geral



Exemplo de Código

```
Java
// Classe abstrata Entrega
public abstract class Entrega {
    protected Transportadora transportadora;
    public Entrega(Transportadora transportadora) {
        this.transportadora = transportadora;
    }
    public abstract void processarEntrega();
}

// Subclasses de Entrega: Pendente
public class Pendente extends Entrega {
    public Pendente(Transportadora transportadora) {
        super(transportadora);
    }
    @Override
    public void processarEntrega() {
        System.out.println("Entrega pendente");
        transportadora.enviar();
    }
}

// Subclasses de Entrega: EmSeparacao
```

```

public class EmSeparacao extends Entrega {
    public EmSeparacao(Transportadora transportadora) {
        super(transportadora);
    }
    @Override
    public void processarEntrega() {
        System.out.println("Entrega em separação");
        transportadora.enviar();
    }
}

// Subclasses de Entrega: Enviada
public class Enviada extends Entrega {
    public Enviada(Transportadora transportadora) {
        super(transportadora);
    }
    @Override
    public void processarEntrega() {
        System.out.println("Entrega enviada");
        transportadora.enviar();
    }
}

// Subclasses de Entrega: Finalizada
public class Finalizada extends Entrega {
    public Finalizada(Transportadora transportadora) {
        super(transportadora);
    }
    @Override
    public void processarEntrega() {
        System.out.println("Entrega concluída");
        transportadora.enviar();
    }
}

// Interface Transportadora
public interface Transportadora {
    public abstract void enviar();
}

// Implementação de Transportadora: Correios
public class Correios implements Transportadora {
    @Override
    public void enviar() {
        System.out.println("Enviando via Correios");
    }
}

// Implementação de Transportadora: TransportadoraSuperSul
public class TransportadoraSuperSul implements Transportadora {
    @Override
    public void enviar() {
        System.out.println("Enviando via Transportadora SuperSul");
    }
}

// Implementação de Transportadora: TransportadoraSuperNorte
public class TransportadoraSuperNorte implements Transportadora {
    @Override
    public void enviar() {
        System.out.println("Enviando via Transportadora SuperNorte");
    }
}

```

```

}

// Exemplo: Bridge
public class ExemploBridge {
    public static void main(String[] args) {
        // Criando instâncias das transportadoras
        Transportadora correios = new Correios();
        Transportadora transportadoraSuperSul = new
        TransportadoraSuperSul();
        Transportadora transportadoraSuperNorte = new
        TransportadoraSuperNorte();
        // Realizando algumas entregas
        Entrega entrega1 = new Pendente(correios);
        entrega1.processarEntrega();
        Entrega entrega2 = new EmSeparacao(transportadoraSuperSul);
        entrega2.processarEntrega();
        Entrega entrega3 = new Enviada(transportadoraSuperNorte);
        entrega3.processarEntrega();
        Entrega entrega4 = new Finalizada(correios);
        entrega4.processarEntrega();
    }
}

```

A classe **Entrega** é uma classe abstrata que representa uma entrega e possui um atributo **transportadora** que identifica qual transportadora será utilizada. Ela possui um método abstrato **processarEntrega()**, que será implementado pelas subclasses concretas.

As subclasses de **Entrega** são: **Pendente**, **EmSeparacao**, **Enviada** e **Finalizada**, que representam diferentes estados de uma entrega.

A classe abstrata **Transportadora** define a interface para as implementações específicas de transportadora. As subclasses concretas **Correios**, **TransportadoraSuperSul** e **TransportadoraSuperNorte** são as implementações concretas das transportadoras.

Cada uma das subclasses de **Entrega** recebe uma instância de **Transportadora** em seu construtor e, ao processar a entrega, chama o método **enviar()** da transportadora correspondente.

Esse exemplo demonstra como o padrão Bridge permite que as classes de entrega e as classes de transportadora possam variar independentemente, permitindo uma maior flexibilidade na extensão e manutenção do sistema.

Na classe `ExemploBridge` (main), criamos instâncias das transportadoras `Correios`, `TransportadoraSuperSul` e `TransportadoraSuperNorte`. Em seguida, realizamos algumas entregas utilizando diferentes estados de entrega (`Pendente`, `EmSeparacao`, `Enviada`, `Finalizada`) em combinação com as transportadoras.

Cada entrega é processada chamando o método `processarEntrega()`, que internamente chama o método `enviar()` da transportadora correspondente. Assim, cada entrega utiliza a implementação específica da transportadora definida na criação da instância de entrega.

O exemplo demonstra como a abstração (`Entrega`) é desacoplada da implementação (`Transportadora`), permitindo que as classes possam variar independentemente e possibilitando diferentes combinações de entrega e transportadora.

Consequências

- Vantagens
 - Você pode criar classes e aplicações independentes de plataforma.
 - O código cliente trabalha com abstrações em alto nível. Ele não fica exposto a detalhes da plataforma.
 - *Princípio aberto/fechado*. Você pode introduzir novas abstrações e implementações independentemente uma das outras.
 - *Princípio de responsabilidade única*. Você pode focar na lógica de alto nível na abstração e em detalhes de plataforma na implementação.
- Desvantagens
 - Você pode tornar o código mais complicado ao aplicar o padrão em uma classe altamente coesa.

Repositório: <https://github.com/ibanez-junior-utfpr/as27s-2023-1>

Pasta: CCH-2