Programação Orientada a Objetos

Encontro 2

Herança e polimorfismo

Objetivos do Encontro

- Apresentar conceitos centrais sobre **Herança e polimorfismo**.
- Discutir exemplos práticos com UML e código.
- Fixar conteúdos com exercícios práticos.

Introdução

Nesta aula abordaremos:

- Fundamentos de **Herança e polimorfismo**.
- Recapitulação de conceitos de **UML**.
- Herança Múltipla e suas implicações.
- Inicio de padrões de projeto que usam herança e polimorfismo.

UML – Visão Geral

Conteúdo teórico detalhado sobre **UML – Visão Geral**.

- **Definições:** UML (Unified Modeling Language) é uma linguagem de modelagem padrão para visualizar, especificar, construir e documentar sistemas orientados a objetos através de diagramas que representam diferentes aspectos do sistema.

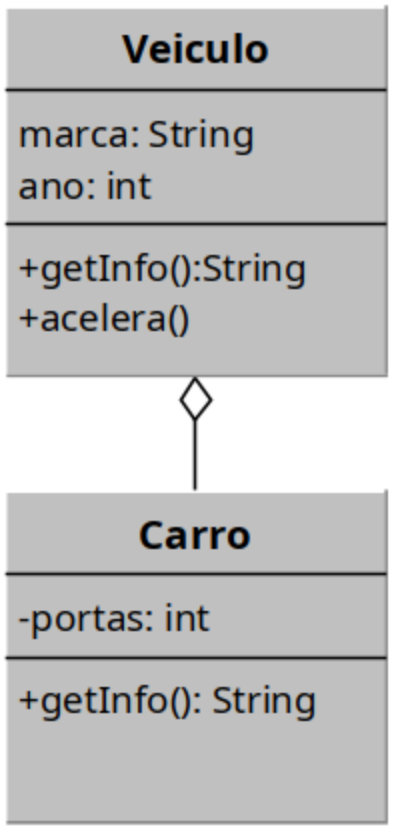
Herança simples

Definições:

- **Herança** é um mecanismo que permite criar uma nova classe baseada em uma classe existente
- A classe **base** (superclasse) define atributos e métodos comuns
- A classe **derivada** (subclasse) herda características da superclasse e pode adicionar novas ou modificar existentes
- Promove **reutilização de código** e estabelece relacionamentos hierárquicos
- Em Java, usa-se a palavra-chave `extends` para implementar herança
- Suporta o princípio **"é um"** - subclasse "é um" tipo da superclasse

Exemplo prático – Herança simples

```java
// Classe base (superclasse)
public class Veiculo {
    protected String marca;
    protected int ano;

    public Veiculo(String marca, int ano) {
        this.marca = marca;
        this.ano = ano;
    }

    public void acelerar() {
        System.out.println("Veículo acelerando...");
    }

    public String getInfo() {
        return marca + " (" + ano + ")";
    }
}

// Classe derivada (subclasse)
public class Carro extends Veiculo {
    private int portas;

    public Carro(String marca, int ano, int portas) {
        super(marca, ano);
        this.portas = portas;
    }

    @Override
    public String getInfo() {
        return super.getInfo() + " - " + portas + " portas";
    }
}
```

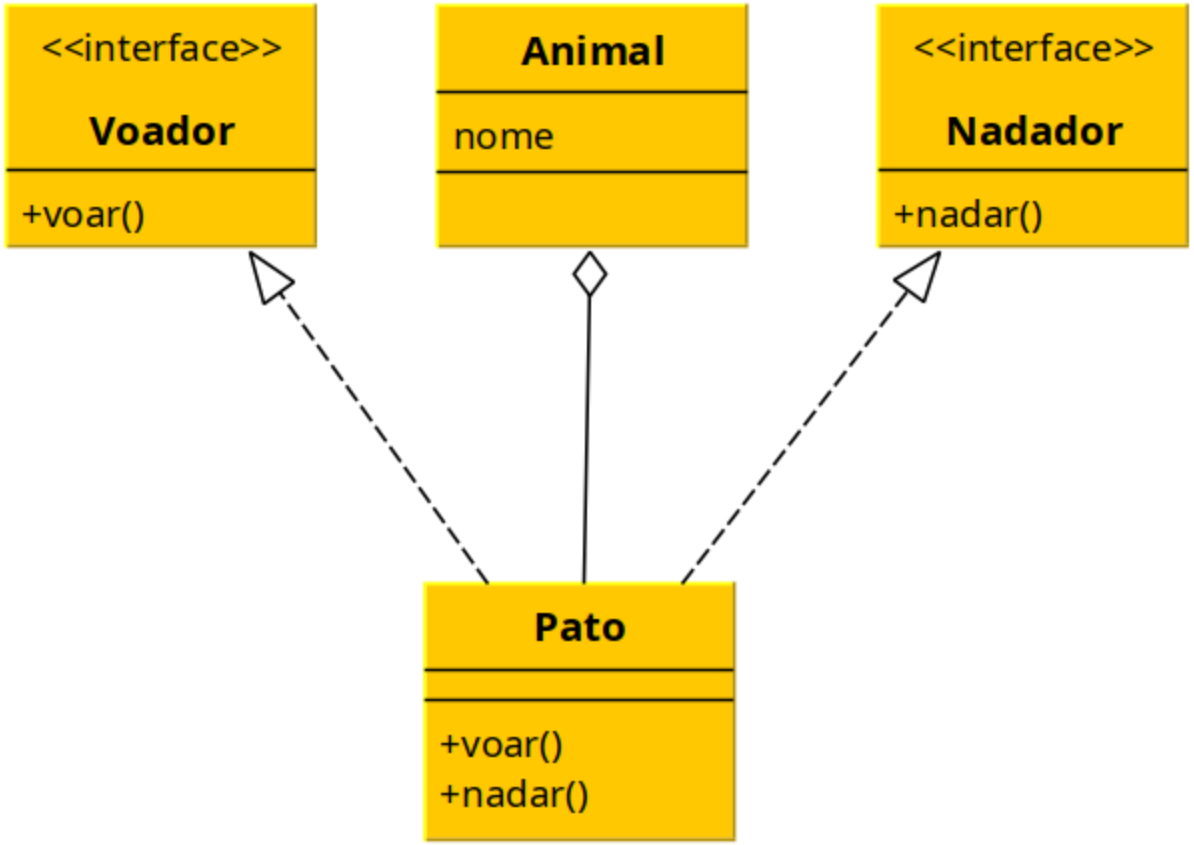


Herança múltipla

Definições:

- **Herança múltipla** permite que uma classe herde de mais de uma superclasse
- Java **não suporta** herança múltipla de classes para evitar ambiguidades
- Em Java, simula-se através de **interfaces** - uma classe pode implementar múltiplas interfaces
- **Interface** define um contrato de métodos que devem ser implementados
- Resolve o problema do **"diamante"** presente em linguagens com herança múltipla real
- Permite que uma classe tenha múltiplos **"tipos"** ou comportamentos

UML – Herança múltipla



Exemplo prático – Herança múltipla

```java
// Interfaces (simulam herança múltipla em Java)
interface Voador {
    void voar();
}

interface Nadador {
    void nadar();
}

// Classe base
public class Animal {
    protected String nome;

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

// Classe que implementa múltiplas interfaces
public class Pato extends Animal implements Voador, Nadador {

    public Pato(String nome) {
        super(nome);
    }

    @Override
    public void voar() {
        System.out.println(nome + " está voando");
    }

    @Override
    public void nadar() {
        System.out.println(nome + " está nadando");
    }
}
```

Sobrescrita de métodos

Definições:

- **Sobrescrita** (Override) permite que uma subclasse redefina um método herdado da superclasse
- O método sobrescrito deve ter a **mesma assinatura** (nome, parâmetros, tipo de retorno)
- Usa-se a anotação `@Override` em Java para indicar sobrescrita explícita
- Implementa **ligação dinâmica** - o método executado depende do tipo real do objeto
- Permite **especialização** do comportamento nas subclasses
- Fundamental para implementar **polimorfismo** efetivo

Exemplo prático – Sobrescrita de métodos

```java
// Classe base
public class Funcionario {
    protected String nome;
    protected double salario;

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

    public double calcularSalario() {
        return salario;
    }

    public void trabalhar() {
        System.out.println(nome + " trabalhando");
    }
}

// Classe derivada que sobrescreve métodos
public class Gerente extends Funcionario {
    private double bonus;

    public Gerente(String nome, double salario, double bonus) {
        super(nome, salario);
        this.bonus = bonus;
    }

    @Override
    public double calcularSalario() {
        return salario + bonus; // Sobrescrita
    }

    @Override
    public void trabalhar() {
        System.out.println(nome + " gerenciando equipe");
    }
}
```

Polimorfismo de variáveis

Definições:

- **Polimorfismo** significa "muitas formas" - capacidade de um objeto assumir múltiplas formas
- **Polimorfismo de variáveis** permite que uma variável da superclasse referencie objetos de subclasses
- O **tipo declarado** da variável pode ser diferente do **tipo real** do objeto
- Permite tratar objetos de diferentes classes de forma uniforme
- Facilita **extensibilidade** - novos tipos podem ser adicionados sem alterar código existente
- Base para **programação genérica** e **padrões de design**

Exemplo prático – Polimorfismo de variáveis

```
public class ExemploPolimorfismo {

    public static void main(String[] args) {
        // Polimorfismo: variáveis da superclasse referenciam subclasses
        Funcionario func1 = new Funcionario("João", 3000);
        Funcionario func2 = new Gerente("Maria", 5000, 2000);

        // Mesma variável, comportamentos diferentes
        System.out.println("Salário João: " + func1.calcularSalario());
        System.out.println("Salário Maria: " + func2.calcularSalario());

        func1.trabalhar(); // "João trabalhando"
        func2.trabalhar(); // "Maria gerenciando equipe"

        // Array polimórfico
        Funcionario[] equipe = {
            new Funcionario("Ana", 2800),
            new Gerente("Carlos", 6000, 1500)
        };

        double total = 0;
        for (Funcionario f : equipe) {
            total += f.calcularSalario(); // Polimorfismo em ação
        }
        System.out.println("Folha total: R$ " + total);
    }
}
```

Polimorfismo em mensagens

Definições:

- **Polimorfismo em mensagens** refere-se à capacidade de enviar a mesma mensagem para objetos diferentes
- Cada objeto responde à mensagem de acordo com sua **implementação específica**
- A **mesma interface** produz **comportamentos diferentes** dependendo do tipo do objeto
- Implementado através de **métodos virtuais** e **ligação dinâmica**
- Permite **código mais flexível** e **desacoplado**
- Essencial para **inversão de dependência** e **programação orientada a interfaces**

Exemplo prático – Polimorfismo em mensagens

```
// Interface comum
interface Forma {
    double calcularArea();
    void desenhar();
}

// Implementações específicas
class Retangulo implements Forma {
    private double largura, altura;

    public Retangulo(double largura, double altura) {
        this.largura = largura;
        this.altura = altura;
    }

    @Override
    public double calcularArea() {
        return largura * altura;
    }

    @Override
    public void desenhar() {
        System.out.println("Desenhando retângulo " + largura + "x" + altura);
    }
}
```

Exemplo prático – Polimorfismo em mensagens

```
class Circulo implements Forma {
    private double raio;

    public Circulo(double raio) {
        this.raio = raio;
    }

    @Override
    public double calcularArea() {
        return Math.PI * raio * raio;
    }

    @Override
    public void desenhar() {
        System.out.println("Desenhando círculo raio " + raio);
    }
}
```

Exemplo prático – Polimorfismo em mensagens

```
public class TesteFomas {
    public static void main(String[] args) {
        Forma[] formas = {new Retangulo(5, 3), new Circulo(4)};

        for (Forma f : formas) {
            f.desenhar(); // Mesma mensagem, comportamentos diferentes
            System.out.println("Área: " + f.calcularArea());
        }
    }
}
```

Estudo de Caso — Retífica Automotiva

Contexto: Uma oficina de retífica presta diferentes serviços em motores e cabeçotes: retífica completa, plaina, teste de trinca, troca de sede/guia, regulagem, etc. Cada serviço tem regras de precificação e prazos distintos. A oficina atende vários tipos de veículos (carro, moto, caminhão), e precisa:

- registrar veículos e seus dados técnicos;
- registrar serviços realizados;
- calcular orçamento e prazo total combinando vários serviços;
- gerar relatórios padronizados (laudo) por serviço;
- aplicar políticas tributárias (quando houver).

Estudo de Caso — Retífica Automotiva

Objetivos:

- Modelar uma hierarquia simples e justificar quando herdar e quando compor.
- Aplicar polimorfismo por sobrescrita para cálculos de custo/prazo e geração de laudo.
- Usar classes abstratas e interfaces para separar contratos de implementação.
- Discutir LSP (Liskov Substitution Principle) e armadilhas comuns.

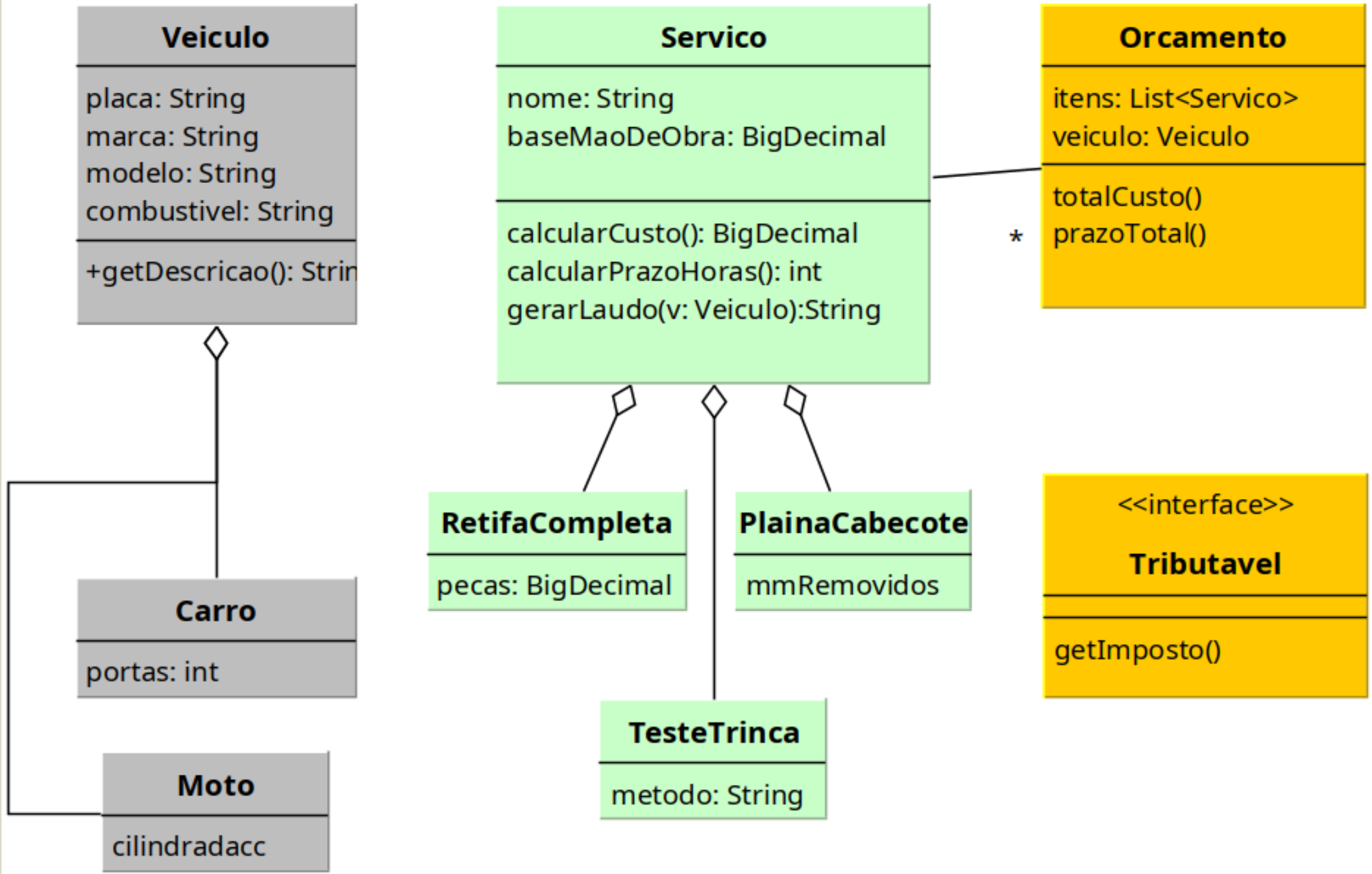
Estudo de Caso — Retífica Automotiva

Requisitos funcionais (recorte):

- Cadastrar veículos: carro, moto, caminhão (dados: placa, marca, modelo, cilindrada/cilindros, combustível).
- Cadastrar serviços realizados: cada serviço informa custo e prazo estimados.
- Combinar múltiplos serviços em um Orçamento, somando custo e prazo.
- Gerar Laudo textual por serviço (conteúdo varia).
- Aplicar impostos quando o cliente é pessoa jurídica.

Estudo de Caso — Retífica Automotiva

Diagrama de classes UML


Veiculo
placa: String
marca: String
modelo: String
combustivel: String
+getDescricao(): Strin
Servico
nome: String
baseMaoDeObra: BigDecimal
calcularCusto(): BigDecimal
calcularPrazoHoras(): int
gerarLaudo(v: Veiculo):String
Orcamento
itens: List<Servico>
veiculo: Veiculo
totalCusto()
prazoTotal()
*
Carro
portas: int
Moto
cilindradacc
RetifaCompleta
pecas: BigDecimal
PlainaCabecote
mmRemovidos
TesteTrinca
metodo: String
<<interface>>
Tributavel
getImposto()

Estudo de Caso — Retífica Automotiva

Notas do projeto:

- Veiculo e Servico são abstratas: não faz sentido instanciar “um veículo” genérico no sistema final, e cada serviço concreto tem regra própria.
- Polimorfismo acontece nas chamadas calcularCusto(), calcularPrazoHoras() e gerarLaudo().
- Tributavel é interface (não herança) para separar o “comportamento opcional” (nem todo serviço é tributável).

Implementação do estudo de caso

```java
// Veiculo.java
public abstract class Veiculo {
    private final String placa, marca, modelo, combustivel;

    protected Veiculo(String placa, String marca, String modelo, String combustivel) {
        this.placa = placa; this.marca = marca; this.modelo = modelo; this.combustivel = combustivel;
    }

    public String getDescricao() {
        return "%s %s (%s) - %s".formatted(marca, modelo, placa, combustivel);
    }
}
```

Implementação do estudo de caso

```java
// Servico.java
import java.math.BigDecimal;

public abstract class Servico {
    protected final String nome;
    protected final BigDecimal baseMaoDeObra;

    protected Servico(String nome, BigDecimal baseMaoDeObra) {
        this.nome = nome;
        this.baseMaoDeObra = baseMaoDeObra;
    }

    public abstract BigDecimal calcularCusto();      // polimórfico
    public abstract int calcularPrazoHoras();        // polimórfico

    public String gerarLaudo(Veiculo v) {            // pode ser hook method
        return "Laudo do serviço '%s' para %s".formatted(nome, v.getDescricao());
    }

    public String getNome() { return nome; }
}
```

Implementação do estudo de caso

```java
// Tributavel.java
import java.math.BigDecimal;

public interface Tributavel {
    BigDecimal getImposto(); // por exemplo, 8.5% do valor
}
```

Implementação do estudo de caso

```java
// Carro.java
public class Carro extends Veiculo {
    private final int portas;
    public Carro(String placa, String marca, String modelo, String combustivel, int portas) {
        super(placa, marca, modelo, combustivel);
        this.portas = portas;
    }
}

// Moto.java
public class Moto extends Veiculo {
    private final int cilindradaCc;
    public Moto(String placa, String marca, String modelo, String combustivel, int cilindradaCc) {
        super(placa, marca, modelo, combustivel);
        this.cilindradaCc = cilindradaCc;
    }
}
```

Sobre as classes Veiculo e Servico

- **Veiculo**: classe abstrata que define os dados comuns a todos os veículos. Não pode ser instanciada diretamente, apenas através de subclasses como Carro e Moto.
- **Servico**: classe abstrata que define o contrato para serviços. Contém atributos comuns e métodos abstratos que devem ser implementados pelas subclasses. Também pode ter métodos concretos (como gerarLaudo) que podem ser sobrescritos.

Implementação do estudo de caso

```java
// RetificaCompleta.java
import java.math.BigDecimal;

public class RetificaCompleta extends Servico implements Tributavel {
    private final BigDecimal pecas;

    public RetificaCompleta(BigDecimal maoDeObra, BigDecimal pecas) {
        super("Retífica completa", maoDeObra);
        this.pecas = pecas;
    }

    @Override
    public BigDecimal calcularCusto() {
        return baseMaoDeObra.add(pecas);
    }

    @Override
    public int calcularPrazoHoras() {
        return 24; // estimativa
    }

    @Override
    public BigDecimal getImposto() {
        // imposto somente sobre mão de obra neste exemplo (8.5%)
        return baseMaoDeObra.multiply(new BigDecimal("0.085"));
    }

    @Override
    public String gerarLaudo(Veiculo v) {
        return super.gerarLaudo(v) + "\n- Processo: desmontagem, medição, usinagem total, montagem e testes.";
    }
}
```

Sobre a classe RetificaCompleta

- **RetificaCompleta**: representa um serviço de retífica completa. Implementa a interface Tributavel, pois tem imposto sobre a mão de obra. Sobrescreve os métodos calcularCusto(), calcularPrazoHoras() e gerarLaudo() para fornecer regras específicas.
  Possui herança multipla: herda de Servico e implementa Tributavel. O método calcularCusto() soma o custo da mão de obra com o custo das peças.

Implementação do estudo de caso

```java
// PlainaCabecote.java
import java.math.BigDecimal;

public class PlainaCabecote extends Servico {
    private final int mmRemovidos;

    public PlainaCabecote(BigDecimal maoDeObra, int mmRemovidos) {
        super("Plaina de cabeçote", maoDeObra);
        this.mmRemovidos = mmRemovidos;
    }

    @Override
    public BigDecimal calcularCusto() {
        // custo adicional por mm removido
        BigDecimal adicional = new BigDecimal(mmRemovidos).multiply(new BigDecimal("25.00"));
        return baseMaoDeObra.add(adicional);
    }

    @Override
    public int calcularPrazoHoras() {
        return 6;
    }

    @Override
    public String gerarLaudo(Veiculo v) {
        return super.gerarLaudo(v) + "\n- Remoção: " + mmRemovidos + " mm; dentro de tolerâncias.";
    }
}
```

Implementação do estudo de caso

```java
// TesteTrinca.java
import java.math.BigDecimal;

public class TesteTrinca extends Servico {
    private final String metodo; // ex.: "líquido penetrante" ou "partículas magnéticas"
    public TesteTrinca(BigDecimal maoDeObra, String metodo) {
        super("Teste de trinca", maoDeObra);
        this.metodo = metodo;
    }

    @Override
    public BigDecimal calcularCusto() {
        return baseMaoDeObra;
    }

    @Override
    public int calcularPrazoHoras() {
        return 2;
    }

    @Override
    public String gerarLaudo(Veiculo v) {
        return super.gerarLaudo(v) + "\n- Método: " + metodo + "; nenhuma trinca detectada.";
    }
}
```

Implementação do estudo de caso

```java
// Orçamento.java
// Orcamento.java
import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.List;

public class Orcamento {
    private final Veiculo veiculo;
    private final List<Servico> itens = new ArrayList<>();

    public Orcamento(Veiculo veiculo) { this.veiculo = veiculo; }

    public void adicionar(Servico s) { itens.add(s); }

    public BigDecimal totalCusto() {
        return itens.stream()
                .map(Servico::calcularCusto)       // polimórfico!
                .reduce(BigDecimal.ZERO, BigDecimal::add);
    }

    public int prazoTotalHoras() {
        return itens.stream()
                .mapToInt(Servico::calcularPrazoHoras) // polimórfico!
                .sum();
    }

    public BigDecimal totalImpostos() {
        return itens.stream()
                .filter(s -> s instanceof Tributavel)
                .map(s -> ((Tributavel) s).getImposto())
                .reduce(BigDecimal.ZERO, BigDecimal::add);
    }

    public String laudos() {
        StringBuilder sb = new StringBuilder("LAUDOS PARA: " + veiculo.getDescricao() + "\n");
        for (Servico s : itens) {
            sb.append("\n--- ").append(s.getNome()).append(" ---\n");
            sb.append(s.gerarLaudo(veiculo)).append("\n");
        }
        return sb.toString();
    }
}
```

Implementação do estudo de caso

```java
// TesteOrcamento.java
import java.math.BigDecimal;
public class TesteOrcamento {
    public static void main(String[] args) {
        var carro = new Carro("ABC-1234", "GM", "Onix 1.0", "Flex", 4);

        var orc = new Orcamento(carro);
        orc.adicionar(new RetificaCompleta(new BigDecimal("1500.00"), new BigDecimal("820.00")));
        orc.adicionar(new PlainaCabecote(new BigDecimal("300.00"), 2));
        orc.adicionar(new TesteTrinca(new BigDecimal("180.00"), "Líquido penetrante"));

        System.out.println("Custo total: R$ " + orc.totalCusto());
        System.out.println("Prazo total (h): " + orc.prazoTotalHoras());
        System.out.println("Impostos: R$ " + orc.totalImpostos());
        System.out.println(orc.laudos());
    }
}
```

Padrões de Projeto que usam Herança e Polimorfismo

- **Strategy**: define uma família de algoritmos, encapsula cada um e os torna intercambiáveis. Permite que o algoritmo varie independentemente dos clientes que o utilizam.
- **Template Method**: define o esqueleto de um algoritmo em uma operação, deixando alguns passos para que subclasses implementem. Permite que subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura.
- **Factory Method**: define uma interface para criar um objeto, mas permite que subclasses decidam qual classe instanciar. O Factory Method permite que uma classe delegue a responsabilidade de instanciar objetos a subclasses, promovendo o uso de polimorfismo.

Padrão de Projeto - Strategy

Definições:

- **Strategy** é um padrão de design comportamental que define uma família de algoritmos, encapsula cada um e os torna intercambiáveis.
- Permite que o algoritmo varie independentemente dos clientes que o utilizam.
- Promove **abstração** e **encapsulamento** de comportamentos.
- Facilita a **extensibilidade** - novos algoritmos podem ser adicionados sem alterar o código existente.
- Baseado no princípio **"programar para interfaces, não para implementações"**.

Padrão de Projeto - Strategy

Exemplo prático:

```
// Interface Strategy
public interface CalculaValorPagamento {
    BigDecimal calcularValor(int horaInicial, int horaFinal, BigDecimal valorHora);
}

// Implementação concreta do Strategy
public class PagamentoPorHoraNormal implements CalculaValorPagamento {
    @Override
    public BigDecimal calcularValor(int horaInicial, int horaFinal, BigDecimal valorHora) {
        //Considerar hora inicial 8 o início do expediente, e hora final 18
        int horaInicialAux = Math.max(8, horaInicial);
        int horaFinalAux = Math.min(18, horaFinal);
        if (horaFinal <= horaInicial) {
            return BigDecimal.ZERO; // Não há horas válidas
        }
        return valorHora.multiply(new BigDecimal(horaFinalAux - horaInicialAux));
    }
}
```

Padrão de Projeto - Strategy Teste da Strategy

```
// Classe Context que usa o Strategy
public class Pagamento {

    private CalculaValorPagamento estrategia;

    public Pagamento(CalculaValorPagamento estrategia) {
        this.estrategia = estrategia;
    }

    public void setEstrategia(CalculaValorPagamento estrategia) {
        this.estrategia = estrategia;
    }

    public BigDecimal calcularPagamento(int horaInicial, int horaFinal, BigDecimal valorHora) {
        return estrategia.calcularValor(horaInicial, horaFinal, valorHora);
    }
}
```

```java
// Teste da Strategy
public class TestePagamento {
    public static void main(String[] args) {
        Pagamento pagamento = new Pagamento(new PagamentoPorHoraNormal());
        BigDecimal valorHora = new BigDecimal("50.00");
        BigDecimal valor = pagamento.calcularPagamento(9, 17, valorHora);
        System.out.println("Valor do pagamento: R$ " + valor);
    }
}
```

Padrão de Projeto - Strategy

Vantagens:

- **Flexibilidade**: permite trocar algoritmos em tempo de execução.
- **Reutilização**: encapsula comportamentos comuns em classes separadas.
- **Manutenção**: facilita a manutenção e evolução do código.
- **Desacoplamento**: separa a lógica de negócio da implementação dos algoritmos.
- **Testabilidade**: cada estratégia pode ser testada isoladamente.

Exercício prático

- Implemente uma nova estratégia de pagamento que calcule o valor considerando horas extras (acima de 8 horas diárias).
- A estratégia deve aplicar um fator de multiplicação (ex: 1.5) para horas extras.
- Crie uma classe de teste que utilize a nova estratégia e verifique o cálculo correto.

Referências

- Page-Jones, M. *Fundamentos do Desenho Orientado a Objeto com UML*. Makron Books.
- Larman, C. *Utilizando UML e Padrões*. Saraiva.
- Ementa da disciplina.