TÓPICO 01: REVISÃO DE CONCEITOS

Design Patterns - Professor Ramon Venson - SATC 2026.1

Roteiro

- Classe vs Objeto
- Interfaces
- Herança vs Composição
- Acoplamento vs Coesão

Classe

- Define um tipo de objeto
- Descreve atributos (estado)
- Descreve métodos (comportamento)
- Funciona como um “molde” / “planta”

```
public class Pessoa {
  private String nome;
  private int idade;
}
```

Objeto

- Instância de uma classe
- Possui valores próprios de atributos
- Usa métodos definidos na classe

```
Pessoa pessoa1 = new Pessoa("Alice", 30);
pessoa1.apresentar();
```

Classe x Objeto (Resumo)

- Classe
  - Define o que o objeto **pode ter** e **pode fazer**
- Objeto
  - É um **exemplo concreto** da classe

Interfaces

- Definem um **contrato** (métodos)
- Não dizem **como** implementar
- Uma classe **implementa** uma interface
- Permitem múltiplas implementações

```
public interface Notificador {
  void notificar(String mensagem);
}
```

Implementando Interfaces

```
public class EmailNotificador implements Notificador {
  @Override
  public void notificar(String mensagem) {
    System.out.println("Enviando e-mail: " + mensagem);
  }
}

public class SmsNotificador implements Notificador {
  @Override
  public void notificar(String mensagem) {
    System.out.println("Enviando SMS: " + mensagem);
  }
}
```

Uso de Interface (Baixo Acoplamento)

```
public class Pedido {
  private Notificador notificador;

  public Pedido(Notificador notificador) {
    this.notificador = notificador;
  }

  public void confirmar() {
    System.out.println("Pedido confirmado.");
    notificador.notificar("Seu pedido foi confirmado!");
  }
}
```

- `Pedido` depende da **interface**, não da implementação

Interfaces em Design Patterns

- Muito usadas em:
  - Strategy
  - Observer
  - Factory Method
  - Adapter
- Ajudam a:
  - Reduzir acoplamento
  - Trocar implementações em tempo de execução
  - Facilitar testes (mocks)

Herança

- Relação **“é um”**
  - `Cachorro` **é um** `Animal`
- Subclasse herda:
  - Atributos
  - Métodos
- Pode sobrescrever comportamentos

```
public class Animal {
  protected String nome;
  public void emitirSom() { ... }
}

public class Cachorro extends Animal {
  @Override
  public void emitirSom() { ... }
}
```

Composição

- Relação **“tem um”**
  - `Carro` **tem um** `Motor`
- Classe é formada por outras classes
- Mais flexível que herança

```
public class Motor {
  public void ligar() { ... }
}

public class Carro {
  private Motor motor = new Motor();
  public void ligarCarro() {
    motor.ligar();
  }
}
```

Herança vs Composição

- Herança
  - Reutiliza código
  - Acoplamento mais forte
  - Usar quando for claramente “é um”
- Composição
  - Baixo acoplamento
  - Maior flexibilidade
  - Preferida na maioria dos casos

Acoplamento

- Grau de **dependência** entre classes/módulos
- Queremos **baixo acoplamento**

Baixo acoplamento:

```
public interface Notificador { ... }

public class Pedido {
  private Notificador notificador;
}
```

- `Pedido` não conhece detalhes de `EmailNotificador` ou `SmsNotificador`

Coesão

- Quão bem relacionadas são as responsabilidades de uma classe
- Queremos **alta coesão**

Alta coesão:

```
public class Calculadora {
  public int somar(int a, int b) { ... }
  public int subtrair(int a, int b) { ... }
}
```

- Uma classe, um foco: operações matemáticas

Acoplamento x Coesão

- **Acoplamento**
  - Relação **entre** classes
  - Ideal: **baixo**
- **Coesão**
  - Organização **dentro** da classe
  - Ideal: **alta**

Conclusões

- Classe define, objeto instancia
- Interfaces definem contratos flexíveis
- Prefira composição a herança quando possível
- Busque:
  - **Baixo acoplamento**
  - **Alta coesão**