TÓPICO 03: PRINCÍPIOS SOLID

Design Patterns - Professor Ramon Venson - SATC 2026.1

Roteiro

- O que é SOLID
- SRP – Single Responsibility
- OCP – Open/Closed
- LSP – Liskov Substitution
- ISP – Interface Segregation
- DIP – Dependency Inversion

SOLID

Conjunto de **5 princípios** para OOP:

- **S –** Single Responsibility
- **O –** Open/Closed
- **L –** Liskov Substitution
- **I –** Interface Segregation
- **D –** Dependency Inversion

Objetivo: código **flexível**, **modular** e **fácil de manter**.

S – Single Responsibility

> Uma classe deve ter **apenas uma razão para mudar**.

- Uma responsabilidade principal
- Aumenta coesão
- Facilita testes e manutenção

SRP – Exemplo ruim

```
public class Pedido {
  private Database db;
  private EmailService email;

  public double calcularTotal() { ... }
  public void salvar() { db.save(this); }
  public void enviarEmailConfirmacao() {
    email.send("Pedido criado", "Seu pedido foi criado.");
  }
}
```

- Calcula, salva e envia email
- Múltiplos motivos de mudança

SRP – Exemplo melhor

```
public class Pedido {
  public double calcularTotal() { ... }
}

public class PedidoRepository {
  public void salvar(Pedido pedido) { ... }
}

public class PedidoNotificador {
  public void enviarConfirmacao(Pedido pedido) { ... }
}
```

- Cada classe com um foco

O – Open/Closed

> Entidades devem ser **abertas para extensão**,
> mas **fechadas para modificação**.

- Adicionar novos comportamentos sem mexer em código já testado
- Usa muito **abstrações** e **polimorfismo**

OCP – Exemplo ruim

```
public class FolhaDePagamento {
  public double calcularSalario(Funcionario f) {
    if (f instanceof FuncionarioHorista) { ... }
    else if (f instanceof FuncionarioMensalista) { ... }
    // novos tipos exigem alterar este método
  }
}
```

- Cada novo tipo de funcionário mexe nessa classe

OCP – Exemplo melhor

```
public abstract class Funcionario {
  public abstract double calcularSalario();
}

public class FuncionarioHorista extends Funcionario {
  public double calcularSalario() { ... }
}

public class FolhaDePagamento {
  public double calcularSalario(Funcionario f) {
    return f.calcularSalario();
  }
}
```

- Folha depende da **abstração** `Funcionario`

L – Liskov Substitution

> Subclasses devem poder substituir a superclasse
> **sem quebrar o comportamento esperado**.

Se `B` herda de `A`, objetos de `B` devem funcionar onde se espera `A`.

LSP – Exemplo clássico ruim

```
public class Retangulo {
  public void setLargura(int l) { ... }
  public void setAltura(int a) { ... }
}

public class Quadrado extends Retangulo {
  @Override
  public void setLargura(int l) {
    super.setLargura(l);
    super.setAltura(l);
  }
}
```

- Cliente espera largura e altura independentes
- `Quadrado` quebra essa expectativa

LSP – Melhor abordagem

```
public abstract class Forma {
  public abstract int calcularArea();
}

public class Retangulo extends Forma { ... }
public class Quadrado extends Forma { ... }
```

- `Quadrado` não é forçado a se comportar como `Retangulo`

I – Interface Segregation

> Clientes não devem ser forçados a depender de **métodos que não usam**.

- Prefira interfaces **pequenas e específicas**

ISP – Exemplo ruim

```
public interface Veiculo {
  void acelerar();
  void frear();
  void voar();
}

public class Carro implements Veiculo {
  public void voar() {
    throw new UnsupportedOperationException();
  }
}
```

- `Carro` implementa algo que não faz sentido para ele

ISP – Exemplo melhor

```
public interface VeiculoTerrestre {
  void acelerar();
  void frear();
}

public interface VeiculoAereo {
  void voar();
}

public class Carro implements VeiculoTerrestre { ... }
public class Aviao implements VeiculoAereo { ... }
```

- Cada cliente implementa só o que precisa

D – Dependency Inversion

> Módulos de alto nível não devem depender de módulos de baixo nível.
> Ambos devem depender de **abstrações**.

- Abstrações não devem depender de detalhes
- Detalhes devem depender de abstrações

DIP – Exemplo ruim

```
public class Gmail {
  public void enviarEmail(String dest, String msg) { ... }
}

public class ServicoEmail {
  private Gmail gmail = new Gmail();

  public void enviar(String dest, String msg) {
    gmail.enviarEmail(dest, msg);
  }
}
```

- `ServicoEmail` depende diretamente de `Gmail`

DIP – Exemplo melhor

```
public interface EmailService {
  void enviarEmail(String dest, String msg);
}

public class Gmail implements EmailService {
  public void enviarEmail(String dest, String msg) { ... }
}

public class ServicoEmail {
  private EmailService email;

  public ServicoEmail(EmailService email) {
    this.email = email;
  }

  public void enviar(String dest, String msg) {
    email.enviarEmail(dest, msg);
  }
}
```

SOLID e Design Patterns

- Muitos padrões de projeto ajudam a aplicar SOLID:
  - Strategy, Observer OCP, DIP
  - Factory Method, Abstract Factory DIP
  - Decorator OCP
- SOLID não é obrigatório, mas é um **guia** útil

Resumo

- **SRP**: uma classe, uma responsabilidade
- **OCP**: estender sem modificar
- **LSP**: subclasses substituíveis
- **ISP**: interfaces específicas
- **DIP**: depender de abstrações

> Objetivo final: código mais simples de **entender**, **testar** e **evoluir**.