Módulo 1

**Padrões de projeto**

**Resumo: Padrões de Projeto**

Os **padrões de projeto** são soluções reutilizáveis para problemas comuns no desenvolvimento de software. Eles melhoram a **qualidade do código**, facilitam a **manutenção** e criam um **vocabulário comum** entre os desenvolvedores, o que melhora a comunicação entre equipes.

**Por que são importantes?**

* Evitam que cada equipe "reinvente a roda".
* Ajudam a **organizar melhor o código**.
* Tornam mais fácil trabalhar em sistemas já existentes.
* Reduzem o esforço e o tempo no desenvolvimento.
* Servem como **boas práticas testadas** ao longo do tempo.

**Analogia com o xadrez**

Assim como enxadristas experientes estudam jogadas conhecidas (como “aberturas”), **desenvolvedores experientes usam padrões** para resolver problemas já enfrentados por outros.  
Isso acelera decisões e melhora os resultados.

**Gang of Four (GoF)**

Em 1994, quatro engenheiros de software — **Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides** — lançaram o livro:

**"Design Patterns: Elements of Reusable Object-Oriented Software"**

Nele, descreveram **23 padrões de projeto** que ficaram conhecidos como **padrões GoF**, amplamente utilizados até hoje.

**Exemplos de padrões famosos**

* **Singleton** – Garante que uma classe tenha apenas uma instância.
* **Observer** – Um objeto notifica outros quando seu estado muda.
* **Factory Method** – Cria objetos sem precisar especificar a classe exata.

**Conclusão**

Conhecer e aplicar padrões de projeto é essencial para se tornar um desenvolvedor melhor. Assim como no xadrez, estudar boas jogadas permite tomar **decisões mais rápidas, mais inteligentes** e com menos erros.

Interface gráfica do usuário, Texto, Aplicativo, Email

O conteúdo gerado por IA pode estar incorreto.

**Conhecendo o padrão de projeto**

**Resumo: Conhecendo o Padrão de Projeto**

Um **padrão de projeto (design pattern)** é uma **solução nomeada e reutilizável** para um problema recorrente no desenvolvimento de software. Ele descreve **como estruturar a solução**, geralmente por meio de módulos, classes, interfaces e interações.

**Características principais dos padrões**

Cada padrão é composto por quatro elementos fundamentais:

1. **Nome** – Permite que desenvolvedores se comuniquem de forma rápida e objetiva. Ex: "Use o padrão *Observer*".
2. **Problema** – Explica o contexto, quando usar, e o tipo de desafio que o padrão resolve.
3. **Solução** – Define a estrutura conceitual e como os elementos interagem. A implementação é feita pelo desenvolvedor.
4. **Consequências** – Avalia os impactos do padrão, como **flexibilidade**, **complexidade** e **performance**.

**Vantagens do uso de padrões**

* **Produtividade**: economizam tempo com soluções já validadas.
* **Reutilização**: facilitam alterações sem grandes quebras, evitando herança rígida.
* **Comunicação**: criam um **vocabulário comum** entre desenvolvedores.
* **Facilidade**: permitem reutilizar **boas práticas** em problemas semelhantes.

**Pontos de atenção / Desvantagens**

* Há **muitos padrões** disponíveis, o que exige estudo e prática.
* **Escolher o padrão certo** para um problema pode ser difícil.
* Padrões **devem ser adaptados** conforme o contexto, e isso exige experiência.
* Iniciantes podem **usar padrões em excesso** ou de forma inadequada.
* Nem todos os padrões são unânimes (exemplo: o **Singleton** é polêmico).

**Conclusão**

Padrões de projeto não são receitas prontas, mas **formas comprovadas de pensar e estruturar soluções**. Aprender e aplicar esses padrões com sabedoria pode elevar a qualidade dos sistemas desenvolvidos e facilitar a colaboração entre profissionais.

**Principais padrões em linhas gerais**

**Resumo: Principais Padrões de Projeto GoF**

Os padrões de projeto da chamada **“Gang of Four” (GoF)** são amplamente utilizados no desenvolvimento de software orientado a objetos. Eles ajudam a lidar com **criação de objetos, estruturação de sistemas e definição de comportamentos**, promovendo reutilização, flexibilidade e organização no código.

**🔹 Categorias dos Padrões GoF**

**1. Padrões Criacionais**

Focam **na criação de objetos** de forma flexível e independente da implementação concreta.

* **Abstract Factory** – Cria famílias de objetos relacionados.
* **Builder** – Separa a construção de um objeto complexo da sua representação.
* **Factory Method** – Define uma interface para criação, mas deixa as subclasses decidirem qual classe instanciar.
* **Prototype** – Cria objetos a partir da clonagem de um protótipo existente.
* **Singleton** – Garante que uma classe tenha **uma única instância** e fornece um ponto de acesso global.

**2. Padrões Estruturais**

Lidam com a **composição de classes e objetos** para formar estruturas mais complexas.

* **Adapter** – Converte a interface de uma classe em outra esperada.
* **Bridge** – Separa a abstração da sua implementação.
* **Composite** – Cria estruturas hierárquicas de objetos (ex: árvore de elementos).
* **Decorator** – Adiciona funcionalidades a objetos dinamicamente.
* **Facade** – Fornece uma interface simplificada para um subsistema.
* **Flyweight** – Compartilha objetos para reduzir o uso de memória.
* **Proxy** – Controla o acesso a um objeto, podendo adicionar lógica extra.

**3. Padrões Comportamentais**

Tratam da **comunicação entre objetos** e da **distribuição de responsabilidades**.

* **Chain of Responsibility** – Passa uma solicitação por uma cadeia de objetos.
* **Command** – Encapsula uma solicitação como um objeto.
* **Interpreter** – Avalia expressões de uma linguagem.
* **Iterator** – Permite percorrer os elementos de uma coleção sem expor sua estrutura.
* **Mediator** – Controla a comunicação entre objetos para reduzir o acoplamento.
* **Memento** – Permite restaurar um estado anterior de um objeto.
* **Observer** – Permite que objetos sejam notificados sobre mudanças em outro objeto.
* **State** – Altera o comportamento de um objeto conforme seu estado interno.
* **Strategy** – Define uma família de algoritmos intercambiáveis.
* **Template Method** – Define o esqueleto de um algoritmo em uma classe base.
* **Visitor** – Permite adicionar operações a objetos sem modificar suas classes.

**✅ Conclusão**

Você **não precisa memorizar todos os padrões**, mas deve saber **quando e onde encontrá-los**, além de entender **quais problemas eles resolvem**. Assim, quando enfrentar desafios semelhantes, poderá aproveitar **soluções testadas e aprovadas por outros desenvolvedores**.

Interface gráfica do usuário, Texto, Aplicativo, Email

O conteúdo gerado por IA pode estar incorreto.

Módulo 2

**Padrão GRASP - Especialista**

🔹 O que é GRASP?

GRASP (General Responsibility Assignment Software Patterns) é um conjunto de princípios de design orientado a objetos, proposto por Craig Larman, que auxilia na atribuição de responsabilidades entre classes de forma coesa, compreensível e reutilizável.

📌 Padrões GoF vs GRASP

Aspecto GoF: Foco Solução para problemas específicos de projeto Princípios gerais para design orientado a objetos

Aspecto GRASP: Abordagem Técnicas reutilizáveis Diretrizes para atribuição de responsabilidades

🔧 Padrão GRASP: Especialista

✅ Problema

Como distribuir responsabilidades entre objetos de forma organizada, coesa e fácil de manter?

🎯 Solução

Atribuir a responsabilidade ao especialista: ou seja, ao objeto que detém a informação necessária para realizá-la.

💡 Exemplo Prático

Cenário: Um site de vendas precisa exibir o valor total de um pedido.

-A classe Pedido conhece todos os seus itens → Ela calcula o valor total.

-A classe ItemPedido conhece a quantidade e o produto → Ela calcula o valor do item.

-A classe Produto fornece o preço unitário → Apenas retorna essa informação.

✅ Cada classe realiza somente o que sabe fazer → responsabilidade distribuída corretamente.

📉 Se o padrão for ignorado…

Ocorre o antipadrão God Class:

-Toda lógica centralizada em uma única classe.

-Classes de domínio apenas com getters/setters (chamadas de "classes idiotas").

-Solução de baixa coesão e difícil manutenção.

⚠️ Limitação do Padrão Especialista

Às vezes, seguir o especialista pode gerar alto acoplamento.

Exemplo: salvar um Pedido no banco de dados.

-A classe Pedido tem os dados → seria o especialista.

-Mas, isso acopla a lógica de negócio com a de persistência.

-Solução correta: separar responsabilidade com padrões como DAO, Repository, etc.

✅ Conclusão

O padrão Especialista ajuda a distribuir responsabilidades com lógica e clareza, mas deve ser aplicado com equilíbrio, considerando cohesão e acoplamento. Ele é a base para classes inteligentes, reutilizáveis e manuteníveis.

Interface gráfica do usuário, Texto, Aplicativo, Email

O conteúdo gerado por IA pode estar incorreto.

**Criador e baixo acoplamento**

**Padrão GRASP: Criador (Creator)**

Problema  
Como decidir qual classe deve ser responsável por criar instâncias de outra classe?

Solução  
Atribuir a responsabilidade de criar um objeto da classe Y à classe X se uma das seguintes condições for verdadeira:

* X contém instâncias de Y (é um agregado de Y);
* X utiliza intensamente instâncias de Y;
* X possui os dados necessários para inicializar Y.

Exemplo  
No sistema de vendas online:

* A classe Pedido deve criar instâncias de ItemPedido.
* Isso porque o Pedido é o agregado (comanda o ciclo de vida) dos seus itens.
* Portanto, a operação adicionarItem pertence à classe Pedido, que internamente instancia o ItemPedido.

Consequência

* Promove encapsulamento e coesão.
* Evita criação espalhada de objetos em serviços genéricos (como “service” ou “controller”).
* Reduz o acoplamento com instanciadores externos.

Cuidados

* Contraindicado quando houver criação condicional de objetos dentro de famílias de classes similares (ex.: quando há herança ou uso de fábrica).

**Padrão GRASP: Baixo Acoplamento**

Problema  
Como distribuir responsabilidades entre módulos para que fiquem pouco dependentes entre si, facilitando manutenção e reuso?

Solução  
Atribuir responsabilidades de forma que minimize dependências diretas entre os módulos. Isso significa:

* Evitar que uma classe dependa de implementações concretas.
* Priorizar o uso de interfaces (abstrações).
* Delegar responsabilidades de forma que cada módulo saiba o mínimo necessário sobre os demais.

Exemplo de problema  
No cálculo do valor total de um Pedido, uma má prática seria:

* A classe Pedido acessar diretamente Produto (através dos seus Itens) e calcular o preço final multiplicando quantidade × preço unitário.

Essa abordagem gera:

* Forte acoplamento entre Pedido, ItemPedido e Produto.
* Dificuldade de alteração/reuso isolado de qualquer uma dessas classes.

Solução alternativa

* A responsabilidade de calcular o valor do item deve estar em ItemPedido (princípio do Especialista).
* A classe Pedido apenas soma os valores retornados por cada ItemPedido, sem saber como cada um foi calculado.

Consequência

* Redução do impacto de mudanças (uma alteração em Produto, por exemplo, não exige alteração em Pedido).
* Melhora a legibilidade e reutilização dos módulos.

Cuidados

* Evitar criar soluções excessivamente abstratas ou genéricas sem necessidade real.
* A recomendação geral é manter as classes de domínio independentes de tecnologias específicas (como banco de dados, GUI, etc.).

1. Considerações Finais

* O padrão Criador organiza a responsabilidade de instanciação com base na relação natural entre objetos.
* O padrão Baixo Acoplamento favorece a flexibilidade e a manutenção do sistema, orientando o desenvolvedor a separar bem as dependências.
* Ambos se complementam e são essenciais para a construção de sistemas modulares, coesos e com boa arquitetura.

**Interface gráfica do usuário, Texto, Aplicativo, Email

O conteúdo gerado por IA pode estar incorreto.**

**Padrão GRASP – Alta Coesão**

1. **O que é Coesão?**

Coesão é a medida do grau de relacionamento entre as responsabilidades de um módulo (classe, método, pacote, subsistema).

* Alta coesão: responsabilidades fortemente relacionadas, com foco claro e propósito único.
* Baixa coesão: responsabilidades dispersas, com pouca relação entre si.

Módulos com alta coesão são mais compreensíveis, reutilizáveis, fáceis de manter e menos propensos a erros quando há mudanças.

1. Padrão GRASP: Alta Coesão

Problema  
Como organizar as responsabilidades dentro de um módulo para que a complexidade geral do sistema seja gerenciável?

Solução  
Agrupar responsabilidades que possuem um propósito comum e bem definido, promovendo:

* Clareza semântica (o que o módulo faz está evidente);
* Redução da complexidade interna;
* Especialização do módulo para uma tarefa específica.

Tipos de Coesão por nível de estrutura:

* Método: deve conter instruções que realizem uma tarefa específica (evite métodos que fazem “um pouco de tudo”).
* Classe: deve agrupar atributos e métodos relacionados ao mesmo conceito ou entidade.
* Pacote: deve reunir classes e interfaces que compartilham a mesma finalidade.
* Subsistema: deve conter pacotes com objetivos alinhados a uma área funcional do sistema.

Exemplo prático  
No pacote java.io da linguagem Java:

* Todas as classes são responsáveis por entrada e saída de dados.
* Cada classe tem uma função bem definida:
  + FileOutputStream: grava arquivos binários;
  + FileInputStream: lê arquivos binários;
  + FileReader: lê arquivos texto;
  + FileWriter: grava arquivos texto.

Esse agrupamento é um exemplo clássico de alta coesão: um pacote com propósito claro e classes bem especializadas.

1. Consequências

* Módulos com alta coesão facilitam o entendimento do sistema, promovem reutilização e manutenção mais simples.
* Alta coesão anda lado a lado com baixo acoplamento: juntos, são a base da boa modularidade de software.
* Em sistemas distribuídos, é preciso equilibrar a coesão com o princípio de minimizar chamadas remotas entre processos (para não gerar gargalos).

1. Considerações Finais

* O padrão Alta Coesão nos guia a manter cada módulo “focado” em um único propósito.
* Evite “módulos utilitários” genéricos demais ou classes com múltiplas responsabilidades.
* Estruture pacotes com nomes representativos e responsabilidades bem delimitadas.
* Use a coesão funcional como critério para agrupar elementos: se colaboram para o mesmo fim, devem estar juntos.

**Interface gráfica do usuário, Texto, Aplicativo, Email

O conteúdo gerado por IA pode estar incorreto.**

**Padrão GRASP – Controlador**

1. O que é o Padrão Controlador?

O padrão GRASP Controlador trata da atribuição de responsabilidade para tratar eventos recebidos pelo sistema. Ele define uma classe responsável por iniciar operações lógicas quando ocorre um evento externo (como o clique de um botão ou o envio de um formulário).

Analogias:

* Um técnico orientando os jogadores de um time.
* Um maestro coordenando os músicos em uma orquestra.

1. Problema

Como lidar com eventos que chegam ao sistema (por usuários ou sistemas externos)?  
A quem atribuir a responsabilidade de coordenar a resposta a esses eventos?

Exemplo:  
Em uma loja on-line, quando o cliente clica em “Finalizar Pedido”, o sistema precisa capturar esse evento e processar a compra.

1. Solução

Atribuir a responsabilidade do processamento inicial a uma classe Controladora.  
Ela atua como intermediária entre a interface do usuário e a lógica de negócio do sistema.

Formas de aplicar:

🟢 Opção 1 – Controlador como fachada:

* Uma classe representa o sistema ou um subsistema.
* Útil quando há poucos eventos a tratar.

🟢 Opção 2 – Controlador por caso de uso:

* Uma classe representa um caso de uso específico (ex: TransferenciaControlador).
* Evita concentração de responsabilidades em uma única classe e melhora a coesão.

Observações importantes:

* O controlador não deve tratar a interface gráfica (isso é responsabilidade da view).
* O controlador apenas orquestra os objetos que realmente realizam as ações.
* Pode receber requisições de diferentes interfaces (web, mobile, terminal etc.).

1. Exemplo prático: MVC (Model-View-Controller)

* View: Componente da interface com o usuário.
* Controller: Recebe eventos da view e inicia ações apropriadas.
* Model: Responsável pela lógica de negócio e persistência dos dados.

Exemplo de frameworks que seguem esse padrão: Spring (@Controller), Angular, Django.

1. Consequências

✅ Benefícios:

* Organização clara da responsabilidade de iniciar processos no sistema.
* Reutilização: o mesmo controlador pode atender diferentes interfaces.
* Coesão: cada controlador trata um conjunto específico de eventos relacionados.

⚠️ Riscos:

* Evite dar ao controlador responsabilidades extras (ex: regras de negócio, acesso direto a banco de dados).  
  Isso gera classes inchadas, difíceis de manter (maestro que tenta tocar todos os instrumentos!).

—

Resumo final:

O Controlador é o "maestro" que coordena os responsáveis por responder a um evento. Ele inicia, mas não executa tudo — cada objeto faz sua parte.

**Interface gráfica do usuário, Texto, Aplicativo

O conteúdo gerado por IA pode estar incorreto.**

**Resumo: Padrão GRASP – Polimorfismo**

1. O que é Polimorfismo?

Polimorfismo é a capacidade de objetos diferentes responderem de forma distinta a uma mesma chamada de método, desde que compartilhem uma interface ou classe base comum.

No contexto de GRASP, o padrão Polimorfismo propõe o uso dessa característica para substituir estruturas condicionais (como if-else ou switch-case) por chamadas genéricas a métodos polimórficos.

1. Problema

Como evitar estruturas condicionais complexas e rígidas no código, especialmente quando há muitas variações de um mesmo comportamento?

Exemplo:  
Na integração de uma loja virtual com diferentes brokers de pagamento, uma abordagem baseada em if-else ou switch-case tornaria o código rígido, difícil de manter e altamente acoplado a cada broker.

1. Solução

Aplicar o polimorfismo para criar uma estrutura genérica e flexível:

* Criar uma interface genérica (ex: IBrokerPagamento).
* Implementar essa interface em diferentes classes, uma para cada broker.
* O código cliente interage apenas com a interface, sem conhecer a implementação específica.

Assim, novas variações (como novos brokers) podem ser adicionadas sem alterar o código cliente, apenas criando novas classes que implementem a interface.

Benefício central: O chamador não precisa mais saber qual implementação está sendo utilizada.

1. Vantagens

* Redução do acoplamento entre módulos.
* Código mais limpo e legível.
* Facilidade para extensão e manutenção.
* Flexibilidade: novos comportamentos são adicionados sem alterar o código existente.

1. Relação com outros padrões

O princípio do polimorfismo é a base de muitos padrões do catálogo GoF, como:

* Adapter
* Strategy
* Command
* State
* Composite
* Proxy

1. Atenção

Evite aplicar polimorfismo de forma desnecessária. Só use uma estrutura genérica quando há variações reais de comportamento. Criar abstrações para casos fixos pode gerar complexidade desnecessária.

—

Resumo final:

O padrão Polimorfismo propõe que, em vez de depender de estruturas condicionais, diferentes comportamentos sejam encapsulados em classes que implementam uma interface comum. Isso torna o código mais limpo, flexível e preparado para mudanças futuras.

**Interface gráfica do usuário, Texto, Aplicativo, Email

O conteúdo gerado por IA pode estar incorreto.**

**Módulo 3**

**Conhecendo o princípio da responsabilidade única (SRP)**

O que é SRP (Single Responsibility Principle)?

O princípio da responsabilidade única afirma que um módulo deve ter apenas uma razão para mudar, ou seja, ele deve ser responsável por uma única parte do funcionamento do sistema, associada a um único ator.

Parte do conjunto SOLID

| Letra | Princípio | Significado em português |
| --- | --- | --- |
| S | Single Responsibility Principle | Princípio da Responsabilidade Única |
| O | Open-Closed Principle | Princípio Aberto-Fechado |
| L | Liskov Substitution Principle | Princípio da Substituição de Liskov |
| I | Interface Segregation Principle | Princípio da Segregação de Interface |
| D | Dependency Inversion Principle | Princípio da Inversão de Dependência |

Objetivo do SRP

Evitar que um mesmo módulo atenda a múltiplas responsabilidades ou atores diferentes, promovendo:

* Melhor organização do código
* Maior facilidade de manutenção
* Maior coerência e clareza
* Maior facilidade de escalabilidade

Exemplo prático

Cenário: Loja virtual com as seguintes funções no módulo ProdutoRepository:

1. Buscar produtos por nome e departamento  
   → Responsabilidade do cliente
2. Incluir novo produto  
   → Responsabilidade da área de vendas
3. Exportar dados para CSV  
   → Responsabilidade da área de integração

Problema:  
Essas três funções atendem a atores diferentes. Se estiverem juntas em um mesmo módulo, ele terá várias razões para mudar. Isso viola o SRP.

Solução com SRP

Separar as responsabilidades em módulos distintos, por exemplo:

* ProdutoConsultaService → para a busca de produtos
* CadastroProdutoService → para inclusão de produtos
* ProdutoExportacaoService → para geração de CSV

Resultado:

* Cada módulo tem uma única responsabilidade
* Facilidade para escalar partes específicas do sistema
* Menos impacto de mudanças locais

Conclusão

O SRP incentiva a divisão lógica do sistema com base nas responsabilidades e atores envolvidos, resultando em módulos:

* Mais simples
* Mais fáceis de manter
* Mais flexíveis para evoluir

Interface gráfica do usuário, Texto, Aplicativo, chat ou mensagem de texto, Email

O conteúdo gerado por IA pode estar incorreto.

**O princípio aberto fechado (OCP) e suas características**

O que é OCP (Open-Closed Principle)?

O princípio aberto-fechado estabelece que um módulo deve estar:

* Aberto para extensão → Deve ser possível adicionar novos comportamentos.
* Fechado para modificação → Não deve ser necessário alterar o código existente para incorporar novos comportamentos.

Objetivo do OCP

Evitar que modificações em um módulo existente impactem outros módulos que dependem dele, o que melhora:

* Manutenção
* Escalabilidade
* Testabilidade
* Estabilidade do sistema

Como aplicar?

A chave é a abstração. Ao programar para interfaces ou classes abstratas:

* O comportamento pode ser modificado ou estendido por meio de novas classes que implementam ou estendem essas abstrações.
* O código existente permanece inalterado.

Exemplo prático: problema com violação do OCP

Cenário: Uma calculadora geométrica que calcula áreas de diferentes figuras.

java

CopiarEditar

public class CalculadoraGeometrica {

public double obterArea(FiguraGeometrica figura) {

if (figura instanceof Triangulo) {

// cálculo para triângulo

} else if (figura instanceof Quadrado) {

// cálculo para quadrado

}

// Se quiser incluir Círculo: terá que modificar o código!

}

}

Problema:  
A cada novo tipo de figura, é necessário modificar a classe CalculadoraGeometrica, violando o OCP.

Solução com OCP, Especialista e Polimorfismo

1. Criar uma interface abstrata FiguraGeometrica com o método obterArea().
2. Cada figura concreta (como Triangulo, Quadrado, Circulo, etc.) implementa sua própria lógica de cálculo.
3. A classe CalculadoraGeometrica apenas invoca figura.obterArea().

java

CopiarEditar

public interface FiguraGeometrica {

double obterArea();

}

public class Triangulo implements FiguraGeometrica {

public double obterArea() {

// cálculo da área do triângulo

}

}

public class CalculadoraGeometrica {

public double calcular(FiguraGeometrica figura) {

return figura.obterArea(); // sem "ifs", sem alterações futuras

}

}

Vantagens da aplicação correta do OCP

* Evita cascatas de alterações
* Código mais limpo e coeso
* Facilidade para adicionar novos comportamentos
* Alta aderência a boas práticas de design orientado a objetos

Conclusão

O OCP permite que novos comportamentos sejam adicionados ao sistema sem alterar módulos já existentes, promovendo:

* Extensibilidade
* Estabilidade
* Baixo acoplamento com alta coesão

Interface gráfica do usuário, Texto, Aplicativo

O conteúdo gerado por IA pode estar incorreto.

**Princípio da Substituição de Liskov (LSP)**

**O que é o LSP?**

**O Liskov Substitution Principle afirma que:**

**"Objetos de uma classe derivada devem poder substituir objetos da classe base sem alterar o comportamento esperado do sistema."**

**Esse princípio foi definido por Barbara Liskov em 1988, sendo um dos pilares da orientação a objetos.**

**Objetivo do LSP**

**Evitar que heranças criem subtipos que violem a lógica esperada da superclasse.  
Ao seguir o LSP:**

* **Polimorfismo funciona corretamente**
* **O comportamento é previsível**
* **Evita-se código frágil ou com efeitos colaterais inesperados**

**Exemplo clássico de violação do LSP: Quadrado e Retângulo**

**Situação**

**Na geometria, um quadrado é um tipo especial de retângulo (com lados iguais). Porém, essa relação não deve ser reproduzida diretamente em código.**

**Java**

**class Retangulo {**

**protected int largura;**

**protected int comprimento;**

**public void setLargura(int l) { largura = l; }**

**public void setComprimento(int c) { comprimento = c; }**

**public int getArea() { return largura \* comprimento; }**

**}**

**Agora, imaginamos:**

**Java**

**class Quadrado extends Retangulo {**

**public void setLargura(int l) {**

**largura = l;**

**comprimento = l;**

**}**

**public void setComprimento(int c) {**

**largura = c;**

**comprimento = c;**

**}**

**}**

**Problema**

**Suponha o seguinte código de cliente:**

**java**

**CopiarEditar**

**public class ClienteRetangulo {**

**public void verificarArea(Retangulo r) {**

**r.setComprimento(10);**

**r.setLargura(8);**

**System.out.println(r.getArea()); // Esperado: 80**

**}**

**}**

**Se r for uma instância de Quadrado, o resultado será:**

* **setComprimento(10) → largura = 10, comprimento = 10**
* **setLargura(8) → largura = 8, comprimento = 8**
* **getArea() retorna 64, não 80 → comportamento incorreto.**

**Motivo da Violação**

* **O Quadrado impõe regras extras (largura == comprimento) que quebram o contrato da superclasse Retângulo.**
* **O cliente espera um comportamento genérico (lógica do retângulo), mas recebe um comportamento alterado (regra do quadrado).**

**Conclusão do LSP**

**Subtipos não devem restringir, alterar ou quebrar o comportamento previsto pela superclasse.**

**Se uma subclasse não pode se comportar como a superclasse sem gerar efeitos colaterais inesperados, a herança está errada.**

**Como resolver?**

* **Evite herança onde não há uma substituição segura.**
* **Use composição: em vez de Quadrado extends Retangulo, crie uma interface comum Figura e implemente comportamentos específicos para cada figura.**
* **Respeite contratos das classes base.**

**Resumo rápido**

| **Conceito** | **Correto** | **Incorreto** |
| --- | --- | --- |
| **Herança** | **Subtipo se comporta como o supertipo** | **Subtipo altera comportamento do supertipo** |
| **Liskov Substitution** | **O código que usa supertipo funciona com subtipo** | **O código falha com o subtipo** |
| **Quadrado vs Retângulo** | **Usar composição ou interface comum** | **Estender Retângulo em Quadrado** |

Interface gráfica do usuário, Texto, Aplicativo

O conteúdo gerado por IA pode estar incorreto.

**Fundamentos do princípio da segregação de interfaces (ISP)**

O que é o ISP?

O Interface Segregation Principle afirma que:

"Nenhum cliente deve ser forçado a depender de métodos que não usa."

Ou seja, interfaces devem ser específicas ao papel de quem as consome, e não genéricas e inchadas com métodos que servem a múltiplos propósitos.

Problema que o ISP resolve

* Interfaces muito grandes, com métodos variados e não relacionados, acabam obrigando as classes a implementar funções desnecessárias.
* Isso viola o encapsulamento e aumenta o acoplamento.
* Também torna o código menos flexível e mais difícil de manter.

Exemplo de Violação do ISP

Imagine uma interface com responsabilidades diversas:

Java

public interface IUsuario {

void login(String usuario, String senha);

void registrar(String usuario, String senha);

void enviarEmail(String mensagem);

void logErro(String mensagem);

}

Essa interface mistura:

* Autenticação
* Envio de mensagens
* Registro de logs

Violação do ISP: classes que só precisam fazer login são obrigadas a implementar métodos de envio de e-mail ou de log, mesmo que não os usem.

Aplicando o ISP: Segregação das Interfaces

A solução é separar a interface original em interfaces menores e coesas.

Java

public interface IAutorizacao {

void login(String usuario, String senha);

void registrar(String usuario, String senha);

}

java

CopiarEditar

public interface IMensagem {

void logErro(String mensagem);

void logInfo(String mensagem);

}

java

CopiarEditar

public interface IEmail {

void enviarEmail(String mensagem);

}

Benefícios

* Cada classe implementa apenas o que realmente utiliza.
* Interfaces ficam mais coesas e fáceis de manter.
* O sistema fica mais modular, testável e extensível.

Resumo rápido

| Problema | Solução com ISP |
| --- | --- |
| Interface faz “tudo” | Quebrar em interfaces menores e coesas |
| Classe implementa o que não usa | Implementa só o necessário |
| Código acoplado | Sistema modular e flexível |

Conclusão

Segregue interfaces para manter alta coesão e baixo acoplamento.

Se uma interface obriga uma classe a implementar métodos que não fazem sentido para ela, essa interface precisa ser dividida.

Interface gráfica do usuário, Texto, Aplicativo, chat ou mensagem de texto

O conteúdo gerado por IA pode estar incorreto.

**O que é o DIP?**

O **Dependency Inversion Principle** estabelece que:

**"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."**

**Definições**

* **Módulo de alto nível**: Contém a lógica de negócio da aplicação (ex: ServicoVenda, ServicoConsultaProduto, CarrinhoCompras).
* **Módulo de baixo nível**: Contém detalhes técnicos e infraestrutura (ex: banco de dados, arquivos, interfaces gráficas, APIs externas).

**Problema que o DIP resolve**

Quando um módulo de alto nível depende diretamente de uma implementação concreta de baixo nível, ele se torna **rigidamente acoplado à tecnologia**, dificultando a manutenção, os testes e a evolução do sistema.

**Exemplo de Violação do DIP**

java

CopiarEditar

public class ServicoConsultaProduto {

public double obterPrecoProduto(String codigo) {

ProdutoRepositoryOracle repo = new ProdutoRepositoryOracle();

Produto produto = repo.buscarPorCodigo(codigo);

return produto.getPreco();

}

}

**Problemas:**

* ServicoConsultaProduto está diretamente acoplado a ProdutoRepositoryOracle.
* Qualquer mudança na tecnologia de persistência **exige alterar o código do serviço**, quebrando o princípio da responsabilidade única e prejudicando a testabilidade.

**Aplicando o DIP com Abstrações**

**Passo 1:** Criar uma **interface abstrata**:

java

CopiarEditar

public interface ProdutoRepository {

Produto buscarPorCodigo(String codigo);

}

**Passo 2:** Fazer a classe de baixo nível implementá-la:

java

CopiarEditar

public class ProdutoRepositoryOracle implements ProdutoRepository {

public Produto buscarPorCodigo(String codigo) {

// Lógica de acesso ao banco Oracle

}

}

**Passo 3:** Injetar a dependência no serviço (via construtor, por exemplo):

java

CopiarEditar

public class ServicoConsultaProduto {

private ProdutoRepository repository;

public ServicoConsultaProduto(ProdutoRepository repository) {

this.repository = repository;

}

public double obterPrecoProduto(String codigo) {

Produto produto = repository.buscarPorCodigo(codigo);

return produto.getPreco();

}

}

**Benefícios do DIP**

| **Sem DIP** | **Com DIP** |
| --- | --- |
| Serviço depende de classe concreta | Serviço depende de abstração (interface) |
| Pouco flexível e difícil de testar | Fácil de trocar implementação (ex: mock) |
| Alta chance de modificar o serviço | Baixa chance de alteração no código de negócio |
| Forte acoplamento entre camadas | Camadas desacopladas por meio de interfaces |

**Conclusão**

**"Ao aplicar o DIP, seu código fica mais flexível, desacoplado e preparado para evoluir com mudanças tecnológicas."**

Essa abordagem favorece a **injeção de dependência**, **testes unitários com mocks**, e mantém a lógica de negócio isolada de decisões técnicas.

Interface gráfica do usuário, Texto, Aplicativo, chat ou mensagem de texto

O conteúdo gerado por IA pode estar incorreto.

Módulo 4