Programação Orientada a Objetos

Prof.ª Ma. Jessica Oliveira

Católica
Universidade Católica de Brasília

Aula 09 – 05/05/2025

Acoplamento, coesão e boas práticas no desenvolvimento.

Católica
Universidade Católica de Brasília

Acoplamento (*Coupling*).

Conceito.

- No contexto da POO, acoplamento refere-se ao **nível de dependência entre classes**, **módulos** ou **componentes** de um sistema.
- Classes que compartilham muitas informações internas ou que se apoiam fortemente umas nas outras apresentam **alto acoplamento**.
- Por outro lado, quando as classes conhecem apenas o necessário umas sobre as outras e interagem por meio de contratos bem definidos, há **baixo acoplamento**, o que é considerado uma boa prática de desenvolvimento.
- Manter o acoplamento baixo é essencial para garantir que o código seja:
  - **mais fácil de modificar**, sem causar efeitos colaterais;
  - **mais simples de testar**, pois as dependências são menores;
  - **mais reutilizável**, já que não depende de implementações específicas.

Classificações.

Tipo	Características principais
Acoplamento de dados	A classe “A” passa somente os dados necessários para a classe “B” (ideal).
Acoplamento de controle	A classe “A” influencia o comportamento da classe “B” passando instruções de controle.
Acoplamento comum	Duas classes acessam uma estrutura de dados global (ex.: variáveis estáticas compartilhadas).
Acoplamento de conteúdo	Uma classe acessa diretamente os atributos/métodos privados de outra classe (gravíssimo).

Exemplo de baixo acoplamento.

- No projeto da biblioteca, observa-se um bom nível de encapsulamento nas classes **`Livro`**, **`Revista`** e **`Ebook`**, que são manipuladas indiretamente pela classe **`Principal`**, por meio de seus métodos públicos.
- A classe **`Publicacao`** define atributos como **`titulo`**, **`autor`** e **`editora`** com modificadores **`protected`**, mas todos os acessos externos utilizam os métodos **`get`** e **`set`**.

Exemplo de baixo acoplamento.

- Dessa forma, mesmo que a forma de armazenamento interna desses atributos mude no futuro (ex.: usar um **`Map`** em vez de campos diretos), o código que os consome não precisará ser alterado.
- Além disso, o uso do polimorfismo na lista **`List<Publicacao>`** contribui diretamente para o baixo acoplamento, permitindo o uso de diferentes tipos concretos (**`Livro`**, **`Revista`**, **`Ebook`**) sem que a classe **`Principal`** dependa diretamente das suas implementações.

Exemplo hipotético de alto acoplamento.

- Considere o seguinte cenário: ao invés de acessar as informações por métodos públicos, o código acessasse diretamente os atributos protegidos ou privados de outra classe.
- Esse tipo de acesso cria dependência direta da implementação, quebrando o encapsulamento e tornando o sistema frágil a alterações internas.

Estratégias para reduzir o Acoplamento.

- **Encapsular atributos internos** utilizando **`private`** ou **`protected`**, e controlar o acesso por meio de métodos públicos (**`getters`** e **`setters`**);
- **Utilizar interfaces**, como a interface **`Digitalizavel`**, que permite interações genéricas e polimórficas;
- **Aplicar o princípio da inversão de dependência (DIP)**, em que classes de alto nível não dependem diretamente de classes de baixo nível, mas sim de abstrações (interfaces);
- **Evitar a criação de objetos diretamente dentro de métodos**. Prefira a injeção de dependência via construtor ou método específico;
- **Promover a coesão interna**, limitando as responsabilidades de cada classe.

Coesão (*Cohesion*).

Conceito.

- Diz respeito ao **grau de proximidade entre as responsabilidades de uma classe ou método**.
- Em outras palavras, uma classe coesa é aquela que **realiza uma única tarefa ou um conjunto de tarefas altamente relacionadas entre si**.
- Quando há muitas responsabilidades desconexas agrupadas em uma única classe, dizemos que ela possui **baixa coesão**, o que prejudica a organização e a manutenção do código.
- A coesão pode ser entendida como a “unidade de propósito” de um componente de *software*: quanto mais focado, **mais coeso** ele é.

Alta Coesão *vs.* Baixa Coesão.

Aspecto	Alta Coesão	Baixa Coesão
Clareza de função	Tem uma responsabilidade principal bem definida.	Mistura diversas responsabilidades.
Manutenção	Facilitada, pois a lógica está bem separada.	Difícil, pois mudanças afetam comportamentos diversos.
Reutilização	Alta, pois os métodos podem ser utilizados de forma modular	Baixa, pois os métodos não fazem sentido isoladamente.
Testabilidade	Alta: fácil isolar e testar comportamentos.	Baixa: difícil garantir que o todo esteja funcionando.

Exemplo de alta coesão: classe `Livro`.

- Todos os atributos e métodos dizem respeito exclusivamente ao objeto “livro”;
- A exibição da ficha literária inclui apenas informações do livro;
- A responsabilidade de geração de versão digital é coerente com a natureza da classe e foi corretamente implementada via interface;
- Não há mistura de responsabilidades, como lógica de persistência, controle de interface ou validação de entrada do usuário.

Exemplo de baixa coesão: classe genérica `Util`.

- É comum, em projetos mal estruturados, encontrarmos classes chamadas **`Util`**, **`Helper`** ou **`Funcoes`**, que agrupam métodos variados e sem relação entre si. Por exemplo:

```
public class Util {
    public void conectarBanco() {...}
    public void lerArquivo() {...}
    public void calcularMedia() {...}
    public void imprimirRelatorio() {...}
}
```

Exemplo de baixa coesão: classe genérica `Util`.

- Problemas com essa abordagem:
  - os métodos não compartilham contexto nem propósito;
  - dificulta a manutenção e reaproveitamento individual;
  - quebra o princípio da responsabilidade única.

Estratégias para aumentar a coesão.

- Aplicar o Princípio da Responsabilidade Única (SRP), onde cada classe deve ter uma e apenas uma razão para mudar, ou seja, deve ser responsável por apenas uma parte da lógica do sistema.
- Refatorar métodos longos ou multifuncionais, separando comportamentos distintos em novos métodos ou classes.
- Agrupar funcionalidades por contexto, criando novas classes especializadas conforme o domínio (ex.: **`GeradorDeRelatorio`**, **`ConversorDeFormato`**, **`CalculadoraDeEstatisticas`**).
- Evitar métodos utilitários genéricos, a menos que sejam abstrações matemáticas, de data/hora ou tratamento de *strings* com foco único.

Princípios SOLID.

Católica
Universidade Católica de Brasília

Introdução.

- O acrônimo SOLID representa um conjunto de cinco princípios fundamentais da programação orientada a objetos.
- Eles foram formalizados por Robert C. Martin (também conhecido como Uncle Bob) e são amplamente utilizados como diretrizes para criar códigos mais robustos, flexíveis e fáceis de manter.
- Aplicar esses princípios corretamente melhora a modularidade, reutilização, manutenção e testabilidade dos sistemas.
- Cada letra de SOLID representa um princípio:

Introdução.

Letra	Princípio	Tradução
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 Interfaces
D	*Dependency Inversion Principle*	Princípio da Inversão de Dependência

Single Responsibility Principle (SRP).

- **Uma classe deve ter apenas uma razão para mudar.** Esse princípio afirma que cada **classe deve possuir apenas uma responsabilidade**, ou seja, deve se concentrar em um único aspecto do sistema.
- No projeto da Biblioteca, a classe **`Livro`** respeita esse princípio. Ela é responsável apenas por representar um livro, com seus atributos (**`isbn`**, **`volume`**, **`gênero`**, etc.) e comportamentos (**`exibeFichaLiteraria()`**, **`geraVersaoDigital()`**), e **não se envolve com outras preocupações**, como impressão, persistência ou envio de e-mails.
- Se incluíssemos um método como **`enviarEmailParaEditora()`** na classe **`Livro`**, estaríamos infringindo o SRP, pois esse comportamento pertence a uma possível classe **`ServicoDeEmail`**.

Open/Closed Principle (OCP).

- **Entidades de software devem estar abertas para extensão, mas fechadas para modificação**. Este princípio recomenda que o comportamento de uma classe possa ser estendido **sem alterar seu código-fonte original**. Isso é possível usando herança, polimorfismo e interfaces.
- A hierarquia entre **`Publicacao`**, **`PublicacaoDigital`** e **`Ebook`** é um bom exemplo. A classe **`Ebook`** estende **`PublicacaoDigital`** e adiciona comportamentos específicos, sem modificar a superclasse.
- Essa estrutura permite que novos tipos de publicações digitais (como **`Audiobook`**) sejam criados sem a necessidade de alterar a superclasse **`PublicacaoDigital`**.

Liskov Substitution Principle (LSP).

- **Objetos de uma subclasse devem poder substituir objetos da superclasse sem causar erros.** Esse princípio assegura que **uma subclasse deve ser completamente compatível com a sua superclasse**, ou seja, deve manter os comportamentos esperados, sem violar a lógica do sistema.
- A classe **`Revista`** estende **`Publicacao`** e substitui (**`override`**) o método **`exibeFichaLiteraria()`** de forma compatível com o comportamento da superclasse.
- Quando **`Revista`** é usada como **`Publicacao`**, o sistema funciona corretamente (o polimorfismo está garantido).

Interface Segregation Principle (ISP).

- **Uma classe não deve ser obrigada a implementar métodos que não utiliza.** Esse princípio sugere que devemos **criar várias interfaces específicas**, em vez de uma única interface grande e genérica.
- Isso evita que classes sejam forçadas a implementar métodos irrelevantes.
- A interface **`Digitalizavel`** define um único método. Tanto **`Livro`** quanto **`Revista`** implementam essa interface de maneira coerente com seu propósito.
- Se a interface incluísse outros métodos não relacionados (**`como baixar()`** ou **`converterParaAudio()`**), seria uma violação do ISP. Nesse caso, seria mais adequado criar interfaces como **`Baixavel`**, **`ConversivelParaAudio`**, etc.

Dependency Inversion Principle (DIP).

- **Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.**
- Este princípio incentiva **a programação voltada a interfaces ou classes abstratas, e não a implementações concretas**. Com isso, o código fica mais desacoplado, reutilizável e testável.
- Atualmente, a classe Principal cria e manipula diretamente os objetos Livro, Revista e Ebook. Para aplicar o DIP, seria possível:
  - Criar uma interface **`RepositorioDePublicacoes`** com métodos como **`salvar()`**, **`buscarPorCodigo()`**, etc.
  - Fazer a classe **`Principal`** interagir apenas com a interface, e não com a implementação.

Refatoração e boas práticas no desenvolvimento orientado a objetos.

Católica
Universidade Católica de Brasília

O que é Refatorar?

- É o processo de reestruturar o código-fonte sem alterar seu comportamento externo.
- O objetivo é melhorar a legibilidade, modularidade, coesão, desempenho e manutenibilidade do software.
- Segundo Fowler (2019), “Refatoração é uma mudança no projeto interno do software para torná-lo mais fácil de entender e mais barato de modificar, sem alterar seu comportamento observável.”

Quando Refatorar?

- Refatoramos quando percebemos que o código:
  - Está muito longo ou difícil de entender;
  - Possui repetições desnecessárias;
  - Realiza várias tarefas ao mesmo tempo;
  - Cresceu demais e perdeu o foco (baixa coesão);
  - Está fortemente dependente de outras classes (alto acoplamento);
  - Precisa se adequar a novos requisitos, sem comprometer a base existente.

Técnicas de refatoração mais comuns.

Técnica	Descrição
Extração de método	Quebra de métodos longos em métodos menores e mais claros.
Extração de classe	Separação de responsabilidades distintas em novas classes.
Substituição condicional por polimorfismo	Uso de herança/interfaces para evitar estruturas complexas de controle.
Introdução de interface	Redução de acoplamento, permitindo múltiplas implementações.
Renomeação semântica	Alteração de nomes para melhorar legibilidade e significado.

No projeto da Biblioteca...

- A classe **`Principal`** tem um papel importante: orquestrar a criação e manipulação dos objetos. Contudo, ela acumula diversas responsabilidades, como:
  - Criação de objetos (**`Livro`**, **`Revista`**, **`Ebook`**);
  - Avaliação de publicações;
  - Exibição de fichas literárias;
  - Exibição da média de avaliações;
  - Impressão de listas e conjuntos;
  - Busca por código;
  - Tratamento de exceções.
- Essa estrutura viola diretamente o Princípio da Responsabilidade Única , o que compromete sua coesão.

Vamos para a prática?

Proposta de refatoração.

Nova Classe	Responsabilidade
`CatalogoDePublicacoes`	Gerenciar lista de publicações, adicionar, buscar e listar publicações.
`ServicoDeAvaliacao`	Registrar e calcular avaliações das publicações.
`InterfaceDeUsuario`	Exibir informações na tela, lidar com mensagens e entrada de dados.
`ServicoDeDigitalizacao`	Gerar versões digitais, se aplicável (trabalha com a interface `Digitalizavel`).

Microprojeto 09

🎯 **Objetivo:** melhorar a estrutura do código para reduzir acoplamento e aumentar coesão.

Para o pós-aula...

- Criar interfaces para reduzir dependências diretas entre classes.
- Aplicar princípios SOLID.
- Refatorar código para torná-lo mais modular.
- Relatório técnico explicando as melhorias feitas no design do código.

Dúvidas?

jessica.oliveira@p.ucb.br

Católica
Universidade Católica de Brasília