Capítulo 5 -
5. Princípios de Projeto

Entendendo a Importância do Projeto de Software

Projetar software é uma atividade que vai muito além de simplesmente escrever código. Como John Ousterhout aponta, a chave para lidar com problemas complexos na computação é a decomposição eficiente. Isso significa dividir um sistema em partes menores e mais gerenciáveis, o que não só facilita a implementação, mas também melhora a manutenção e a evolução do software ao longo do tempo.

No dia a dia do desenvolvimento de software, essa ideia de decomposição aparece constantemente. Por exemplo, imagine que uma empresa está construindo um sistema de e-commerce. Em vez de tentar desenvolver tudo de uma vez, a equipe pode dividir o projeto em módulos como gerenciamento de usuários, catálogo de produtos, carrinho de compras e sistema de pagamento. Cada um desses módulos pode ser tratado separadamente, permitindo que equipes diferentes trabalhem simultaneamente sem grandes interferências.

Exemplo de Aplicação:

Um aplicativo de banco digital pode ser dividido em funcionalidades como abertura de conta, transferências, investimentos e atendimento ao cliente. Ao desenvolver cada funcionalidade de forma independente, as equipes podem garantir que cada parte do sistema seja implementada e testada corretamente antes da integração final. Isso reduz a complexidade e facilita a identificação e correção de erros.

Abstração: O Segredo para Combater a Complexidade

A complexidade dos sistemas modernos exige estratégias que tornem seu desenvolvimento mais eficiente. Uma dessas estratégias é a criação de abstrações. Uma abstração é uma maneira de esconder os detalhes internos de um sistema e fornecer uma interface simples para interação. É como dirigir um carro: você só precisa saber acelerar e frear, sem se preocupar com o funcionamento interno do motor.

No desenvolvimento de software, isso pode ser visto no uso de funções, classes e bibliotecas. Quando um desenvolvedor utiliza uma biblioteca de autenticação para login, por exemplo, ele não precisa entender todos os detalhes internos de como os tokens de autenticação são gerados e validados. Ele simplesmente chama uma função e obtém o resultado esperado. Isso economiza tempo e evita erros.

Exemplo de Aplicação:

Muitas empresas integram sistemas de pagamento, como Stripe ou PayPal, em seus aplicativos. Em vez de desenvolver um sistema de pagamento do zero, os desenvolvedores usam a API da empresa, que já abstrai toda a complexidade envolvida na transação financeira. Isso permite que a equipe se concentre em outras funcionalidades do sistema, como a experiência do usuário ou a segurança dos dados.

Integridade Conceitual

A integridade conceitual é um princípio de projeto introduzido por Frederick Brooks em 1975, no livro The Mythical Man-Month. Esse princípio afirma que um sistema deve manter coerência e coesão entre suas funcionalidades, evitando se tornar um amontoado desconexo de elementos. A integridade conceitual melhora a experiência do usuário, pois garante consistência na interface e nas funcionalidades do sistema, permitindo uma curva de aprendizado mais suave.

Exemplo de Falta de Integridade Conceitual:

Imagine um sistema que apresenta informações em tabelas, mas sem padronização:

- Cada tela possui um layout diferente para as tabelas (variações em tamanho de fonte, espaçamento, uso de negrito etc.).
- Algumas tabelas permitem ordenação por coluna, enquanto outras não.
- Os valores apresentados utilizam moedas distintas sem uma conversão padronizada (algumas em reais, outras em dólares).

Essa inconsistência causa confusão e dificuldade para os usuários, tornando o sistema mais complexo do que o necessário.

Exemplo de Aplicação:

Um sistema de blogs que permite que usuários adicionem um sinal de interrogação no título de um post, o que ativa uma janela perguntando se eles desejam receber respostas. No entanto, o sistema já possui a funcionalidade de comentar posts, criando confusão entre os usuários, que não entendem a diferença entre "respostas" e "comentários". Essa duplicação de funcionalidades é um exemplo clássico de como a falta de integridade conceitual pode afetar negativamente a experiência do usuário.

Coesão e Acoplamento

- Coesão: Uma classe deve ter uma única responsabilidade. Por exemplo, uma classe Stack<T> que gerencia apenas operações de pilha é altamente coesa.
- Acoplamento: Mede a dependência entre classes. Um acoplamento aceitável ocorre quando uma classe usa métodos públicos de outra com uma interface

estável.

Exemplo de Coesão:

Uma classe **`Estacionamento`** que gerencia operações de estacionamento, mas também armazena dados do gerente, viola a coesão. A solução é separar as responsabilidades em uma classe **`Funcionario`.**

Exemplo de Acoplamento Ruim:

Duas classes que compartilham um arquivo para armazenar dados. Se uma alterar o formato do arquivo, a outra será impactada. A solução é tornar a dependência explícita, garantindo um acoplamento aceitável.

Exemplo de Aplicação:

Em um sistema de gerenciamento de estoque, métricas de coesão podem identificar classes com responsabilidades mistas, como uma classe que gerencia estoque e também calcula impostos. A refatoração para separar essas responsabilidades melhora a manutenção.

Princípios SOLID

1. Responsabilidade Única: Uma classe deve ter apenas uma razão para mudar.
   Exemplo: Separar lógica de cálculo e exibição em classes distintas (Disciplina e Console).
2. Segregação de Interfaces: Interfaces devem ser específicas para cada cliente.
   Exemplo: Criar interfaces separadas para funcionários CLT e públicos, evitando métodos desnecessários.
3. Inversão de Dependências: Depender de abstrações, não de implementações concretas.
   Exemplo: Usar interfaces para permitir flexibilidade na escolha de implementações.
4. Prefira Composição a Herança: Composição oferece maior flexibilidade e evita acoplamento rígido.
   Exemplo: Usar composição para permitir que um cachorro emita sons sem herdar de Animal.
5. Aberto/Fechado: Classes devem ser fechadas para modificação, mas abertas para extensão.
   Exemplo: O método sort da classe Collections em Java permite ordenação personalizada sem modificar o código original.
6. Substituição de Liskov: Subclasses devem substituir a classe base sem alterar o comportamento esperado.
   Exemplo: Uma subclasse que redefine soma para concatenar strings viola o princípio, pois altera a semântica do método.

Exemplo de Aplicação:

Em um sistema de gerenciamento de bolsas de estudo, o Princípio Aberto/Fechado pode ser aplicado para permitir a adição de novos tipos de bolsas (como doutorado) sem modificar o código existente. Isso é feito através do uso de polimorfismo e métodos abstratos.

Métricas de Código-Fonte

Métricas como coesão, acoplamento e complexidade ajudam a avaliar a qualidade do código, mas devem ser interpretadas no contexto do projeto. Por exemplo, um sistema pode tolerar um acoplamento maior em troca de maior flexibilidade.

Exemplo de Aplicação:

Em um sistema de gerenciamento de estoque, métricas de coesão podem identificar classes com responsabilidades mistas, como uma classe que gerencia estoque e também calcula impostos. A refatoração para separar essas responsabilidades melhora a manutenção.

Conclusão

O projeto de software eficiente requer decomposição, abstração, integridade conceitual e a aplicação de princípios como SOLID. Essas práticas garantem sistemas robustos, fáceis de manter e escaláveis. Exemplos reais, como a integração de APIs e a divisão de funcionalidades em módulos, demonstram a importância desses conceitos no mercado. Ao aplicar esses princípios, as empresas podem criar soluções que atendem às necessidades dos usuários e se adaptam às mudanças ao longo do tempo.

Capítulo 6 -
6. Padrões de Projeto

Introdução aos Padrões de Projeto

Os padrões de projeto surgiram com o arquiteto Christopher Alexander, que documentou soluções para problemas recorrentes na construção de cidades e edifícios. Em 1995, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides adaptaram essa ideia para o desenvolvimento de software, criando um catálogo de soluções para problemas comuns de projeto de software. Esses padrões são divididos em três categorias principais:

Criacionais: Soluções para a criação flexível de objetos. Exemplos: Fábrica, Singleton, Builder.

Estruturais: Soluções para composição de classes e objetos. Exemplos: Proxy, Adaptador, Fachada, Decorador.

Comportamentais: Soluções para interação e divisão de responsabilidades. Exemplos: Strategy, Observador, Template Method, Visitor.

O uso de padrões de projeto promove a reutilização de código, a flexibilidade e a manutenção do software, permitindo que sistemas sejam projetados para mudanças futuras sem grandes reestruturações.

Fábrica

Contexto:

Em um sistema distribuído baseado em TCP/IP, várias funções criam objetos do tipo TCPChannel para comunicação remota. No entanto, o sistema precisa suportar também comunicação via UDP, o que exige flexibilidade na criação de objetos.

Problema:

O operador new exige um nome de classe fixo, o que dificulta a extensão do sistema para suportar novos tipos de canais de comunicação.

Solução:

O padrão Fábrica encapsula a criação de objetos em um método estático, ocultando o tipo específico por trás de uma interface.

Implementação:

Criamos uma classe ChannelFactory com um método estático create() que retorna um objeto do tipo Channel.

O método create() pode ser modificado para retornar diferentes tipos de canais (TCP, UDP) sem alterar o código do cliente.

```
class ChannelFactory {
  public static Channel create() {
    return new TCPChannel(); // Pode ser alterado para UDPChannel
  }
}

void f() {
  Channel c = ChannelFactory.create(); // Uso do método fábrica
  ...
}
```

Aplicações da Fábrica:

Criação de objetos em sistemas que precisam suportar múltiplas implementações.

Encapsulamento da lógica de instanciação para facilitar a manutenção.

Exemplo de Aplicação:

Em um sistema de comunicação, a fábrica pode ser usada para criar diferentes tipos de conexões (TCP, UDP, WebSocket) sem modificar o código do cliente.

Singleton

Contexto:

Alguns sistemas necessitam de uma única instância de uma classe, como um gerenciador de conexões a um banco de dados.

Problema:

Como garantir que apenas uma instância seja criada e fornecida globalmente?

Solução:

O padrão Singleton usa um método estático para gerenciar a instância única da classe.

Implementação:

O construtor da classe é privado para evitar instanciação direta.

```
class Singleton {
  private static Singleton instance;

  private Singleton() {} // Construtor privado

  public static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton(); // Cria a instância única
    }
    return instance;
  }
}
```

Um método estático getInstance() controla a criação e o acesso à instância única.

Aplicações do Singleton:

Gerenciadores de recursos compartilhados (banco de dados, cache, log).

Configurações globais do sistema.

Exemplo de Aplicação:

Um gerenciador de conexões de banco de dados pode ser implementado como um Singleton para garantir que todas as partes do sistema usem a mesma conexão.

Proxy

Contexto:

Uma classe BookSearch busca livros por ISBN. Com o aumento de usuários, surge a necessidade de melhorar o desempenho com um sistema de cache.

Problema:

Como adicionar funcionalidades (como cache) sem modificar a classe original e sem violar o Princípio da Responsabilidade Única?

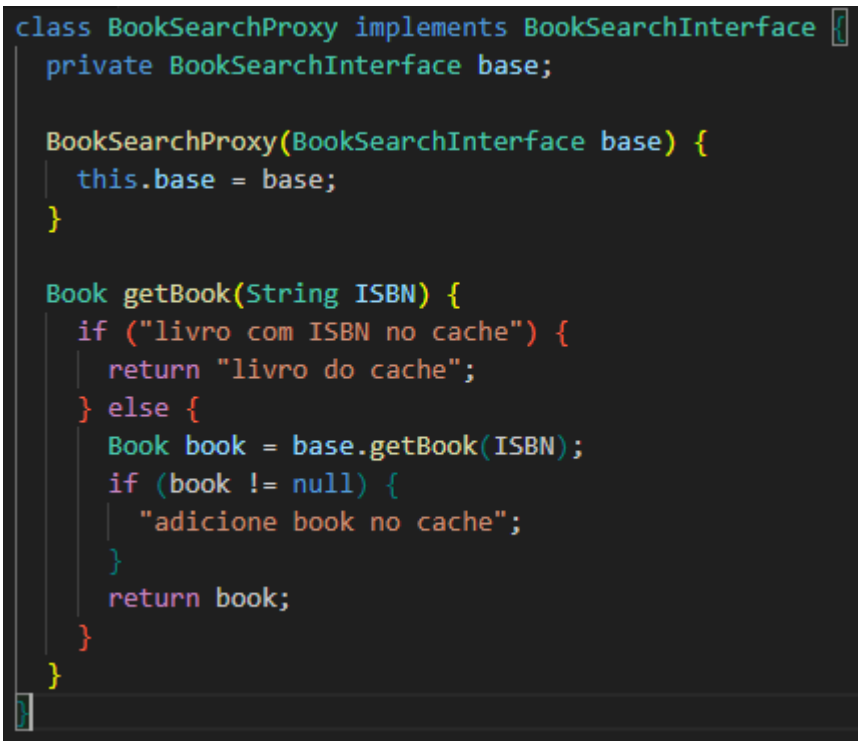
Solução:

O padrão Proxy insere um objeto intermediário entre o cliente e o objeto real, implementando a mesma interface e agregando funcionalidades adicionais.

Implementação:

Criamos uma interface BookSearchInterface para garantir que tanto BookSearch quanto BookSearchProxy possam ser usados de forma intercambiável.

O BookSearchProxy verifica se o livro está no cache antes de chamar o método real de busca.



```
class BookSearchProxy implements BookSearchInterface {
  private BookSearchInterface base;

  BookSearchProxy(BookSearchInterface base) {
    this.base = base;
  }

  Book getBook(String ISBN) {
    if ("livro com ISBN no cache") {
      return "livro do cache";
    } else {
      Book book = base.getBook(ISBN);
      if (book != null) {
        "adicione book no cache";
      }
      return book;
    }
  }
}
```

Aplicações do Proxy:

Cache: Melhora o desempenho ao evitar operações repetidas.

Comunicação remota: Encapsula protocolos de comunicação com servidores remotos.

Controle de acesso: Valida permissões antes de permitir operações.

Exemplo de Aplicação:

Em um sistema de e-commerce, um ProductSearchProxy pode ser usado para armazenar em cache os detalhes de produtos frequentemente acessados.

Adaptador

Contexto:

Um sistema precisa controlar projetores de diferentes fabricantes, como Samsung e LG, mas cada um tem uma interface diferente para ligar o dispositivo.

Problema:

Como unificar a interface de controle para projetores de diferentes marcas sem modificar suas implementações originais?

Solução:

O padrão Adaptador converte a interface de uma classe em outra interface esperada pelo cliente.

Implementação:

Criamos uma interface comum Projetor com o método liga().

Implementamos adaptadores para cada tipo de projetor, como AdaptadorProjetorSamsung.

```
class AdaptadorProjetorSamsung implements Projetor {
  private ProjetorSamsung projetor;

  AdaptadorProjetorSamsung(ProjetorSamsung projetor) {
    this.projetor = projetor;
  }

  public void liga() {
    projetor.turnOn(); // Chama o método específico do projetor Samsung
  }
}
```

Aplicações do Adaptador:

Integração de sistemas legados com novas interfaces.

Compatibilidade entre bibliotecas de terceiros.

Exemplo de Aplicação:

Em um sistema de automação residencial, um adaptador pode ser usado para controlar lâmpadas de diferentes marcas (como Philips e Xiaomi) através de uma única interface.

Fachada

Contexto:

Um interpretador de uma linguagem de programação X requer múltiplas etapas para executar um programa: análise léxica, análise sintática, geração de código e execução.

Problema:

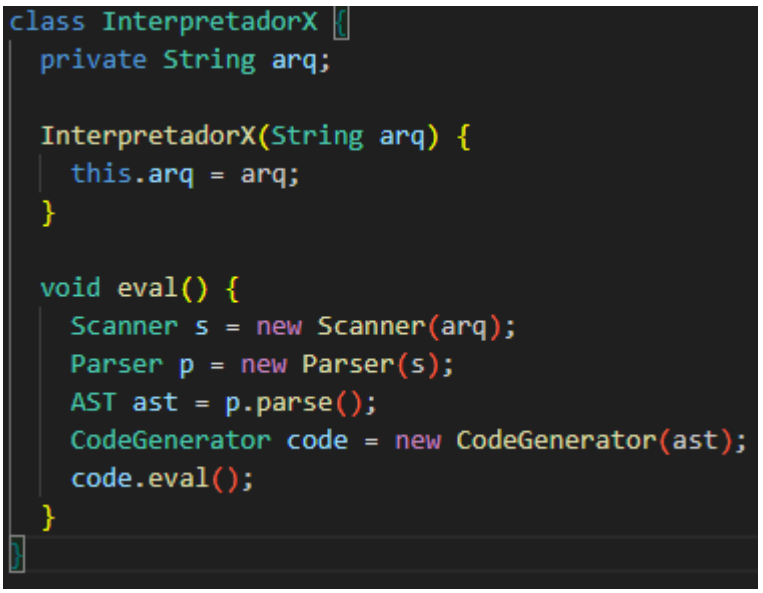
Os usuários querem uma interface mais simples para executar programas, sem precisar conhecer os detalhes internos.

Solução:

O padrão Fachada encapsula a complexidade interna, fornecendo uma interface simplificada.

Implementação:

A classe InterpretadorX esconde os detalhes de execução, como a criação do scanner, parser e gerador de código.



```
class InterpretadorX {
  private String arq;

  InterpretadorX(String arq) {
    this.arq = arq;
  }

  void eval() {
    Scanner s = new Scanner(arq);
    Parser p = new Parser(s);
    AST ast = p.parse();
    CodeGenerator code = new CodeGenerator(ast);
    code.eval();
  }
}
```

Aplicações da Fachada:

Simplificação de APIs complexas.

Encapsulamento de subsistemas.

Exemplo de Aplicação:

Em um sistema de gerenciamento de pedidos, uma fachada pode ser usada para encapsular a lógica de criação de pedidos, cálculo de impostos e envio de notificações.

Flyweight

Contexto:

Um editor de texto precisa exibir milhões de caracteres, mas armazenar cada caractere individualmente consome muita memória.

Problema:

Como reduzir o consumo de memória ao lidar com objetos que possuem muitos atributos repetidos?

Solução:

O padrão Flyweight compartilha instâncias de objetos imutáveis para reduzir o uso de memória.

Implementação:

Criamos uma classe FlyweightCaracter para armazenar o símbolo do caractere.

Usamos uma fábrica (FlyweightFactory) para gerenciar as instâncias compartilhadas.

```
class FlyweightCaracter {
  private char simbolo;

  public FlyweightCaracter(char simbolo) {
    this.simbolo = simbolo;
  }
}

class FlyweightFactory {
  private Map<Character, FlyweightCaracter> pool = new HashMap<>();

  public FlyweightCaracter getCaracter(char simbolo) {
    pool.putIfAbsent(simbolo, new FlyweightCaracter(simbolo));
    return pool.get(simbolo);
  }
}
```

Aplicações do Flyweight:

Editores de texto e processadores de documentos.

Jogos com muitos objetos repetidos.

Exemplo de Aplicação:

Em um jogo de RPG, o padrão Flyweight pode ser usado para compartilhar texturas e modelos 3D de personagens e objetos.

Conclusão

Os padrões de projeto, como Fábrica, Singleton, Proxy, Adaptador, Fachada e Flyweight, são ferramentas essenciais para resolver problemas comuns de design de software. Eles promovem a reutilização de código, a flexibilidade e a manutenção, permitindo que sistemas sejam projetados para mudanças futuras sem grandes reestruturações. A aplicação desses padrões em cenários reais demonstra sua importância no desenvolvimento de software moderno.