Princípios de Projeto

O capítulo 5 do livro "*Engenharia de Software Moderna"* introduz e explora os **princípios fundamentais de projeto de software**, abordando a importância de decompor problemas complexos em partes menores e gerenciáveis. O autor inicia o capítulo citando John Ousterhout, que afirma que o maior desafio da Ciência da Computação é justamente essa decomposição de problemas para que possam ser resolvidos de forma independente. A complexidade do software moderno exige técnicas eficazes para estruturar e organizar sistemas de maneira eficiente. Entre as estratégias discutidas, destaca-se o conceito de **abstração**, que permite simplificar sistemas sem perder funcionalidades essenciais. O autor enfatiza que um bom projeto de software deve não apenas dividir problemas, mas também garantir que essa divisão resulte em módulos coesos, de fácil manutenção e expansão.

5.1 Exemplo Prático: Projeto de um Compilador

Para ilustrar a importância de um bom projeto, o autor apresenta o **exemplo do projeto de um compilador**. O problema inicial – converter um código escrito em uma linguagem X para uma linguagem Y – é extremamente complexo. Para torná-lo gerenciável, a solução encontrada ao longo dos anos foi dividir o compilador em módulos especializados:

1. **Analisador Léxico** – Responsável por dividir o código-fonte em *tokens* (ex: palavras-chave if, for, while).
2. **Analisador Sintático** – Constrói a **Árvore de Sintaxe Abstrata (AST)**, verificando se os tokens respeitam a gramática da linguagem.
3. **Analisador Semântico** – Verifica erros como uso de variáveis sem declaração ou operações inválidas.
4. **Gerador de Código** – Converte o programa para uma linguagem de nível mais baixo, como Assembly ou bytecode.

Essa decomposição torna o problema mais gerenciável e exemplifica **a importância da modularização e abstração**.

5.2 Integridade Conceitual

O conceito de **integridade conceitual** foi introduzido por Frederick Brooks e enfatiza que **um sistema deve parecer ter sido projetado por uma única mente, mesmo que várias pessoas trabalhem nele**. Isso significa que o sistema deve manter uma consistência interna em termos de nomenclatura, organização, estilo e estrutura.

Exemplo: Imagine um sistema onde algumas classes manipulam arquivos com o método lerArquivo(), enquanto outras usam readFile(). Isso reduz a integridade conceitual, tornando o código mais difícil de entender.

Diretrizes para alcançar a integridade conceitual:

- Definir e seguir **padrões claros** de nomenclatura e design.
- Minimizar a variedade de abordagens para resolver problemas semelhantes.

- **Centralizar decisões arquiteturais** em um pequeno grupo de desenvolvedores experientes.

5.3 Ocultamento de Informação (*Information Hiding*)

O ocultamento de informação é um princípio fundamental introduzido por David Parnas. Ele propõe que os módulos de um sistema **devem esconder detalhes internos de implementação**, expondo apenas o necessário para interação com outros módulos.

Benefícios:

- Redução do impacto de mudanças internas (baixo acoplamento).
- Aumento da manutenibilidade do código.
- Melhor modularização e organização do sistema.

Exemplo: Um sistema de banco de dados pode encapsular a forma como os dados são armazenados, permitindo que mudanças na estrutura do banco não afetem outras partes do software.

Boas práticas para ocultamento de informação:

- Utilizar modificadores de acesso adequados (private, protected, public).
- Definir interfaces claras e bem documentadas.
- Evitar exposição desnecessária de detalhes internos.

5.4 Coesão

A **coesão** mede **o quão fortemente relacionadas e focadas são as responsabilidades de um módulo ou classe**.

- **Alta coesão**: Um módulo realiza **uma única responsabilidade bem definida**.
- **Baixa coesão**: Um módulo executa muitas tarefas diferentes e não relacionadas.

Exemplo de baixa coesão: Uma classe RelatorioFinanceiro que também gerencia usuários e autenticação.

Como melhorar a coesão?

- Seguir o **Princípio da Responsabilidade Única (SRP)** – cada classe ou módulo deve ter um único propósito.
- Refatorar classes grandes em classes menores e especializadas.

5.5 Acoplamento

O **acoplamento** mede o **grau de dependência entre módulos**. O objetivo é reduzir ao máximo o acoplamento, tornando os sistemas mais flexíveis e fáceis de manter.

Tipos de acoplamento:

1. **Acoplamento forte** (*tight coupling*) – Módulos dependem fortemente uns dos outros.
2. **Acoplamento fraco** (*loose coupling*) – Módulos interagem de forma independente, facilitando modificações.

Estratégias para reduzir acoplamento:

- Utilizar **interfaces** para definir contratos de comunicação.
- Aplicar **injeção de dependência** (Dependency Injection).
- Seguir o princípio **Aberto/Fechado** (Open/Closed Principle).

5.6 Princípios Gerais de Projeto

O autor apresenta sete princípios fundamentais para um bom design de software:

1. **Princípio da Responsabilidade Única (SRP)** – Cada módulo deve ter uma única responsabilidade bem definida.
2. **Princípio da Segregação de Interfaces** – Interfaces devem ser específicas e não forçar módulos a implementar métodos desnecessários.
3. **Princípio da Inversão de Dependências (DIP)** – Depender de abstrações, não de implementações concretas.
4. **Prefira Composição a Herança** – Utilizar composição (objetos internos) ao invés de herança excessiva para melhorar flexibilidade.
5. **Lei de Demeter** – Uma classe deve interagir diretamente apenas com seus próprios atributos e métodos, evitando cadeias longas de chamadas (`obj.getA().getB().getC()`).
6. **Princípio Aberto/Fechado** – Classes devem ser abertas para extensão, mas fechadas para modificação.
7. **Princípio da Substituição de Liskov (LSP)** – Objetos derivados devem poder substituir objetos da classe base sem quebrar o sistema.

5.7 Métricas para Avaliação de Qualidade

O capítulo encerra com uma discussão sobre **métricas objetivas para avaliar projetos de software**. Algumas das principais métricas abordadas incluem:

- **Número médio de métodos por classe** (avalia coesão).
- **Acoplamento entre classes** (mede a dependência entre módulos).
- **Complexidade ciclomática** (mede a quantidade de caminhos lógicos dentro do código).

Essas métricas ajudam a identificar **pontos de melhoria** no design de sistemas.

O capítulo 5 de "*Engenharia de Software Moderna"* fornece uma abordagem abrangente sobre princípios de projeto, destacando a importância da modularização, ocultamento de informação, coesão e acoplamento na construção de software robusto e de fácil manutenção. Os princípios apresentados servem como diretrizes essenciais para desenvolvedores que desejam criar sistemas escaláveis e sustentáveis, enfatizando que **um bom design de software não acontece por acaso, mas é resultado da aplicação consciente de boas práticas e princípios bem estabelecidos**.