CAPÍTULO 5 - PRINCÍPIOS DE PROJETO

John Ousterhout explica que um bom design de software envolve dividir o problema em partes menores e usar abstrações para tornar o desenvolvimento mais fácil e organizado. Isso permite que cada parte seja tratada separadamente, simplificando a complexidade do sistema.

A integridade conceitual é fundamental, pois um sistema deve ser coerente e padronizado. Quando partes do sistema funcionam de maneira inconsistente, tanto os desenvolvedores quanto os usuários têm dificuldades. A falta de padronização no código, como em nomenclatura ou estrutura, pode tornar a manutenção difícil e confusa.

O ocultamento de informação (information hiding), proposto por David Parnas, sugere que detalhes internos das classes ou módulos devem ser escondidos, o que melhora a flexibilidade, organização e compreensão do sistema. Isso torna o sistema mais fácil de manter e modificar, uma vez que as mudanças internas não afetam o restante do código.

Os principais benefícios desse princípio são:

- **Desenvolvimento paralelo**: diferentes desenvolvedores podem trabalhar em classes separadas sem interferência.
- **Facilidade para mudanças**: alterações em uma classe não impactam o restante do sistema.
- **Facilidade de entendimento**: novos desenvolvedores podem focar apenas nas partes do sistema que precisam trabalhar.

Para obter essas vantagens, as classes devem esconder detalhes de implementação sujeitos a mudanças, tornando públicos apenas os métodos essenciais para a interação externa. O conjunto desses métodos define a **interface da classe**, que deve ser estável para evitar a necessidade de alterações no código que a utiliza. Isso garante um software mais modular, seguro e fácil de manter.

Os métodos get e set (getters e setters) são usados em linguagens orientadas a objetos para acessar atributos privados de uma classe. Getters permitem leitura e setters permitem escrita desses atributos.

Apesar de serem amplamente utilizados, getters e setters não garantem ocultamento de informação, pois ainda expõem detalhes internos da classe. Portanto, só devem ser usados quando for realmente necessário liberar acesso aos dados.

A coesão em design de software significa que uma classe deve ter uma única responsabilidade ou funcionalidade. Todos os seus métodos e atributos devem estar focados em implementar esse único serviço, o que facilita sua implementação, compreensão e manutenção. Além disso, classes coesas são mais fáceis de alocar para manutenção, reuso e teste.

A separação de interesses é um conceito relacionado que também defende que uma classe deve tratar apenas de um único interesse ou responsabilidade. Portanto, a coesão e a separação de interesses são essenciais para criar sistemas mais organizados, reutilizáveis e fáceis de manter.

Os Princípios SOLID, definidos por Robert Martin e Michael Feathers, são diretrizes para tornar sistemas de software mais fáceis de manter e evoluir. Eles ajudam a evitar a degradação da qualidade interna do código ao longo do tempo.

1. **Princípio da Responsabilidade Única (SRP)** – Cada classe deve ter uma única responsabilidade, ou seja, um único motivo para ser modificada. Isso separa a apresentação de regras de negócio.
2. **Princípio da Segregação de Interfaces (ISP)** – Interfaces devem ser pequenas e específicas para cada cliente, evitando métodos desnecessários.
3. **Princípio da Inversão de Dependências (DIP)** – Classes devem depender de abstrações (interfaces) e não de implementações concretas, garantindo maior flexibilidade.
4. **Princípio Aberto/Fechado (OCP)** – Classes devem ser abertas para extensão, mas fechadas para modificação, permitindo customizações sem alterar o código original.
5. **Princípio de Substituição de Liskov (LSP)** – Subclasses devem substituir a classe base sem alterar seu comportamento esperado.

Métricas de software são usadas para quantificar propriedades de um projeto de software, como tamanho, coesão, acoplamento e complexidade. Elas analisam características do código fonte para fornecer uma avaliação objetiva da qualidade do projeto.

Métricas de Tamanho:

A métrica mais comum é Lines of Code (LOC), que mede a quantidade de linhas de código de um sistema, função, classe ou pacote. Porém, seu uso tem limitações:

- LOC não mede produtividade com precisão, pois quantidade de código não reflete necessariamente qualidade ou eficiência.
- Dependendo do contexto, menos código pode significar um sistema melhor e mais otimizado.

- Outras métricas incluem número de métodos, atributos, classes e pacotes, que ajudam a entender a estrutura do software.

Embora ferramentas automatizadas possam calcular essas métricas, a interpretação dos valores exige análise de contexto, pois o que é aceitável em um sistema pode não ser em outro.

Métrica LCOM (Lack of Cohesion Between Methods)

LCOM mede a falta de coesão de uma classe, ou seja, quanto maior o LCOM, pior a organização do código.

Interpretação do LCOM:

- **LCOM alto** → Baixa coesão, indicando que os métodos da classe não trabalham juntos de forma eficiente.
- **LCOM baixo** → Alta coesão, o que é desejável, pois indica que os métodos compartilham atributos e trabalham com um propósito comum.

Métrica CBO (Coupling Between Objects)

A métrica CBO (Coupling Between Objects) mede o acoplamento estrutural de uma classe, ou seja, quantas classes diferentes ela depende diretamente.

A métrica conta o número de classes das quais A depende estruturalmente. A depende de uma classe B se:

1. Chama um método da classe B
2. Acessa um atributo público de B
3. Herde de B (herança)
4. Declara variáveis locais, parâmetros ou tipos de retorno do tipo B
5. Captura uma exceção do tipo B
6. Levanta uma exceção do tipo B
7. Cria objetos do tipo B

Interpretação do CBO:

- CBO alto → Alto acoplamento, o que pode dificultar a manutenção, reuso e testes.
- CBO baixo → Melhor modularidade e independência entre classes.

Complexidade Ciclomática (CC)

A Complexidade Ciclomática (CC) é uma métrica que mede a complexidade de um método ou função com base no número de caminhos de execução possíveis. O cálculo é simples:

CC = Número de comandos de decisão + 1

Comandos de decisão incluem if, while, for, case, entre outros.

Interpretação:

- 1 a 10: Baixa complexidade (código fácil de testar e manter).
- 11 a 20: Complexidade moderada (pode precisar de refatoração).
- 21 a 50: Alta complexidade (dificuldade em testar e manter).
- Acima de 50: Muito complexo (refatoração necessária).

Quanto maior a CC, mais difícil o código se torna de entender e testar.