Engenharia de Software

Capítulo 5 - Princípios de Projeto

Livro-texto

Slides baseados no conteúdo do livro **Engenharia de Software Moderna** de Marco Tulio Valente

ISBN: 978-65-00-01950-6

Site: https://engsoftmoderna.info

Introdução

- **Projeto de software** é a solução de um problema que consiste na implementação de um sistema que atenda aos requisitos funcionais e não-funcionais definidos por um cliente (ou PO)
- Para conseguir chegar nessa solução, devemos:
  - Decompor o problema me partes menores
  - Implementar estas partes de forma independente
  - As partes devem ser abstratas
    - **Abstração** é uma representação simplificada de uma entidade
    - A **complexidade** é combatida por meio de **abstrações**

Introdução

- Exemplo: compilador
- Dado um programa em uma linguagem X, deve-se convertê-lo em uma linguagem Y (de máquina)
- Deve-se implementar:
  - Analisador léxico: lê o arquivo e o divide em tokens
  - Analisador sinático: analisa os tokens e monta uma AST
  - Analisador semântico: detecta erros de tipo e etc.
  - Gerador de código: converte o programa da linguagem X para Y

Propriedades de Projeto

- Projeto de software possui as seguintes propriedades:
  - Integridade conceitual
  - Ocultamento de informação
  - Coesão
  - Acoplamento
- Os **princícipos de projeto** visam garantir que um determinado projeto atende a essas propriedades
- Utilizar **métricas** para quantificar estas propriedades

Integridade Conceitual

- Um sistema não pode ser um amontoado de funcionalidades, sem coerência e coesão entre elas
- Falta de padronização revela falta de integridade conceitual:
  - Variáveis com padrões de nomes diferentes (`notaTotal` e `nota_total`)
  - Dois frameworks web em partes diferentes do sistema
  - Dois problemas parecidos utilizando diferentes est. de dados
  - Inconsistência no acesso à informação (arquivo vs. parâmetro)

Ocultamento de Informação

- Módulos devem esconder detalhes de implementação que estão sujeitos a mudanças
  - A modularização é um mecanismo capaz de tornar sistemas de software mais flexíveis e fáceis de entender, reduzindo assim o tempo de desenvolvimento
- Módulos podem ser *classes*, métodos, funções, pacotes ou componentes

Vantagens do Ocultamento de Informação

- **Desenvolvimento em paralelo**: quando classes ocultam suas principais informações, fica mais fácil implementar em paralelo por DEVs diferentes
- **Flexibilidade a mudanças**: quando classes ocultam detalhes de implementação do resto do sistema torna mais fácil trocar sua implementação
- **Facilidade de entendimento**: não é preciso entender toda a complexidade do sistema, mas apenas as classes pelas quais ficou responsável

Ocultamento de Informação

- Uma classe para ser útil deve tornar alguns de seus métodos públicos
- Código externo que chama métodos de uma classe é dito ser **cliente** da classe
- O conjunto de métodos públicos de uma classe define a sua **interface**
- Interfaces devem ser estáveis, pois mudanças na interface de uma classe podem demandar atualizações em seus clientes

Ocultamento de Informação

```
public class Estacionamento {
    public Hashtable<String, String> veiculos;
    public Estacionamento() {
        veiculos = new Hashtable<>();
    }
    public static void main(String[] args) {
        Estacionamento e = new Estacionamento();
        e.veiculos.put("TCP-7030", "Uno");
        e.veiculos.put("BNF-4501", "Gol");
        e.veiculos.put("JKL-3481", "Corsa");
    }
}
```

Ocultamento de Informação

```
public class Estacionamento {
    private Hashtable<String, String> veiculos;
    public Estacionamento() {
        veiculos = new Hashtable<>();
    }
    public void estaciona(String placa, String veiculo) {
        veiculos.put(placa, veiculo);
    }
    public static void main(String[] args) {
        Estacionamento e = new Estacionamento();
        e.estaciona("TCP-7030", "Uno");
        e.estaciona("BNF-4501", "Gol");
        e.estaciona("JKL-3481", "Corsa");
    }
}
```

Getters e Setters

- São os métodos de acesso de leitura (getters) e acesso de escrita (setters) usados na maioria das linguagens orientadas a objetos
- Não são uma garantia de estar ocultando dados da classe (acabam sendo um instrumento de liberação de informação)
- São uma alternativa melhor que dar acesso direto a um atributo da classe, pois é possível mudar a implementação sem impactar em clientes da classe

Coesão

- Toda a classe deve implementar uma única funcionalidade ou serviço (ou toda a classe deve ter uma única responsabilidade no sistema)
- Vantagens:
  - Facilita a implementação, entendimento e manutenção da classe
  - Facilita a alocação de um único responsável por manter uma classe
  - Facilita o reúso e teste de uma classe (é mais fácil manter uma classe coesa do que uma com várias responsabilidades)

Coesão

```
float sin_or_cos(double x, int op) {
    if (op == 1) {
        "calcula e retorna o seno de x"
    } else {
        "calcula e retorna o cosseno de x"
    }
}
```

Coesão

```
class Stack<T> {
    boolean empty() { ... }
    T pop() { ... }
    void push(T e) { ... }
    int size() { ... }
}
```

```
class Estacionamento {
    private String nome_gerente;
    private String fone_gerente;
    private String cpf_gerente;
    private String endereco_gerente;
}
```

Acoplamento

- É a força da conexão entre duas classes
- Existem dois tipos de acoplamento: **acoplamento aceitável** e **acoplamento ruim**
- Acoplamento aceitável
  - A classe `A` usa apenas métodos públcos da classe `B`
  - A interface provida por `B` é estável (não muda com frequência)

Acoplamento

- Acoplamento ruim
  - Quando a classe `A` realiza um acesso direto a um arquivo ou banco de dados da classe `B`
  - Quando as classe `A` e `B` compartilham uma variável ou estrutura de dados global
  - Quando a interface da classe `B` não é estável

Acoplamento

- *Maximize a coesão das classes e minimze o acoplamento entre elas*
- Não se deve eliminar completamente o acoplamento de uma classe com outras classes, pois é natural que uma classe precise de outras

Acoplamento

```
class A {
    private void f() {
        int total;
        File arq = File.open("arq1.db");
        total = arq.readInt();
        ...
    }
}
```

Acoplamento

```
class B {
    private void g() {
        int total;
        // computa o valor de total
        File arq = File.open("arq1.db");
        arq.writeInt(total);
        ...
        arq.close();
    }
}
```

Acoplamento

```
class A {
    private void f(B b) {
        int total;
        total = b.getTotal();
        ...
    }
}
```

Acoplamento

```
class B {
    int total;

    public int getTotal() {
        return total;
    }

    private void g() {
        // computa o valor de total
        File arq = File.open("arq1.db");
        arq.writeInt(total);
    }
}
```

Acoplamento

- Acoplamento estrutural
  - Ocorre quando uma classe `A` possui uma referência explícita em seu código para uma classe `B`
  - Exemplo: acoplamento entre `Estacionamento` e `Hashtable`
- Acoplamento evolutivo (ou lógico)
  - Ocorre quando mudanças na classe `B` tendem a se propagar para a classe `A`
  - Exemplo: mudanças no formato do arquivo criado por `B` terão impacto em `A`

SOLID e Outros Princípios de Projeto

- O objetivo não é apenas entregar um projeto capaz de resolver um problema, mas também que facilite as manutenções futuras
  - Princípio da Responsabilidade Única (**S**)
  - Princípio da Segregação de Interfaces (**I**)
  - Princípio da Inversão de Dependências (**D**)
  - Prefira Composição a Herança
  - Princípio de Demeter
  - Princípio Aberto/Fechado (**O**)
  - Princípio de Substituição de Liskov (**L**)

Princípio da Responsabilidade Única (S)

- Deve existir um único motivo para modificar qualquer classe em um sistema
- Recomenda separar **apresentação** de **regras de negócio**
- Exemplo

```
class Disciplina {
  void calculaIndiceDesistencia() {
      indice = "calcula índice de desistência";
      System.out.println(indice);
  }
}
```

Princípio da Responsabilidade Única (S)

```
class Console {
    void imprimeIndiceDesistencia(Disciplina disciplina) {
        double indice = disciplina.calculaIndiceDesistencia();
        System.out.println(indice);
    }
}
```

```
class Disciplina {
    void calculaIndiceDesistencia() {
        double indice = "calcula índice de desistência";
        return indice;
    }
}
```

Princípio da Segregação de Interfaces (I)

- É um caso particular de Responsabilidade Única com foco em interfaces
- Interfaces têm que ser pequenas, coesas e específicas para cada tipo de cliente
- O objetivo é evitar que clientes dependam de interfaces com métodos que eles não vão usar

Princípio da Segregação de Interfaces (I)

```
interface Funcionario {
    double getSalario();
    double getFGTS(); // Apenas funcionário CLT
    int getSIAPE();   // Apenas funcionários públicos
    ...
}
```

Princípio da Segregação de Interfaces (I)

```
interface Funcionario {
    double getSalario();
    ...
}

interface FuncionarioCLT extends Funcionario {
    double getFGTS();
    ...
}

interface FuncionarioPublico extends Funcionario {
    int getSIAPE();
    ...
}
```

Princípio da Inversão de Dependências (D)

- Clientes devem depender de interfaces ao invés de classes concretas
  - O nome mais intuitivo seria **Prefira Interfaces a Classes**

Princípio da Inversão de Dependências (D)

```
void f() {
    ...
    ProjetorLG projetor = new ProjetorLG();
    ...
    g(projetor);
}
```

```
void g(Projetor projetor) {
...
}
```

Prefira Composição a Herança

- Existem dois tipos de herança:
  - Herança de classes (`class A extends B`)
    - Envolve reúso de código
  - Herança de interfaces (`interface I extends J`)
    - Não envolve reúso de código
- Herança expõe para subclasses detalhes de implementação das classes pai, logo diz-se que herança viola o encapsulamento da classe pai
- Entre herança e composição, composição é a melhor solução

Prefira Composição a Herança

- Solução via Herança

```
class Stack extends ArrayList {
    ...
}
```

- Solução via composição

```
class Stack {
    private ArraList elementos
    ...
}
```

Prefira Composição a Herança

- A solução por composição é melhor pois:
  - (1) Um `Stack` em termos conceituais não é um `ArrayList`, mas sim uma estrutura que pode usar um `ArrayList` na sua implementação interna
  - (2) Quando se usa herança, a classe `Stack` irá herdar métodos como `get` e `set`, que não fazem parte da especificação de pilhas

Princípio de Demeter

- A implementação de um método deve ivocar apenas os seguintes outros métodos:
  - De sua própria classe (caso 1)
  - De objetos passados como parâmetros (caso 2)
  - De objetos criados pelo próprio método (caso 3)
  - De atributos da classe do método (caso 4)

Princípio de Demeter

```
class PrincipioDemeter {
    T1 attr;
    void f1() {
        ...
    }
    void m1(T2 p) {      // Método que segue Demeter
        f1();            // Caso 1: própra  classe
        p.f2();          // Caso 2: parâmetro
        new T3().f3();   // Caso 3: criado pelo método
        attr.f4();       // Caso 4: atributo da classe
    }
    void m2(T4 p) {      // Método que viola Demeter
        p.getX().getY().getZ().doSomething();
    }
}
```

Princípio Aberto/Fechado (O)

- Uma classe deve estar fechada para modificações e aberta para extensões
- Tem como objetivo a construção de classes flexíveis e extesíveis, capazes de se adaptarem a diversos cenários de uso, sem modificações no seu código fonte
- Exemplo:
  - A classe `Collections` de Java

Princípio Aberto/Fechado (O)

```
List<String> nomes = Arrays.asList("joao", "maria", "alexandre", "ze");
Collections.sort(nomes);
System.out.println(nomes); // resultado: ["alexandre", "joao", "maria", "ze"]
```

```
Comparator<String> comparator = new Comparator<String>() {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
Collections.sort(nomes, comparator);
System.out.println(nomes); // resultado: ["ze", "joao", "maria", "alexandre"]
```

Princípio de Substituição de Liskov (L)

- Se `S` é um subtipo de `T`, então objetos do tipo `T` em um programa podem ser substituídos por objetos do tipo `S` sem alterar nenhuma doas propriedades desse programa (referência)

Princípio de Substituição de Liskov (L)

```
void f(A a) {
    ...
    a.g();
    ...
}
```

Sendo `B1`, `B2`, e `B3` subclasses de `A`, podemos ter:

```
f(new B1()); // f pode receber objetos da subclasse B1
...
f(new B2()); // e de qualquer subclasse de A, como B2
...
f(new B3()); // e B3
```

Métricas de Código Fonte

- Visam quantificar propriedades de um projeto de software
- Permite a avaliação da qualidade de um projeto de forma mais objetiva
- Propriedades avaliadas:
  - Tamanho (LOC)
  - Coesão (LCOM)
  - Acoplamento (CBO)
  - Complexidade (CC)

Tamanho (LOC)

- Chamada de LOC (*Lines of Code*)
- Baseia-se em contar a quantidade de linhas de código
- Usada para medir o tamanho de uma função, classe, pacote ou sistema inteiro
- Não deve ser usada para medir a produtividade dos DEVs
- Outras metas de tamanho:
  - Número de métodos, número de atributos, número de classes e número de pacotes

Coesão (LCOM)

- Chamada de LCOM (*Lack of Cohesion Between Methods*)
- Quanto maior o valor LCOM, maior a falta de coesão de uma classe e portanto, pior o seu projeto
- LCOM(C) é o número de pares de métodos - dentre todos os possíveis pares de métodos de C - que não usam atributos em comum, isto é, a interseção deles é vazia

Coesão (LCOM)

```
class A {
    int a1;
    int a2;
    int a3;
    void m1() {
        a1 = 10;
        a2 = 20;
    }
    void m2() {
        System.out.println(a1);
        a3 = 30;
    }
    void m3() {
        System.out.println(a3);
    }
}
```

Coesão (LCOM)

Pares de métodos (M)	Conjunto A	Interseção dos Conjuntos A
(m1, m2)	A(m1) = {**a1**,a2} A(m2) = {**a1**,a3}	{a1}
(m1, m3)	A(m1) = {a1,a2} A(m3) = {a3}	ø
(m2, m3)	A(m2) = {a1,**a3**} A(m3) = {**a3**}	{a3}

Coesão (LCOM)

- LCOM(C) = 1

Acoplamento (CBO)

- Chamada CBO (*Coupling Between Objects*)
- É utilizada para medir acoplamento estrutural entre duas classes
- Diz que `A` depende de uma classe `B` quando:
  - `A` chama um método de `B`
  - `A` acessa um atributo público de `B`
  - `A` herda de `B`
  - `A` declara uma variável local, um parâmetro ou tipo de retorno do tipo `B`

Acoplamento (CBO)

- Diz que `A` depende de uma classe `B` quando:
  - `A` captura uma exceção do tipo `B`
  - `A` lança uma exceção do tipo `B`
  - `A` cria um objeto do tipo `B`

Acoplamento (CBO)

```
class A extends T1 implements T2 {
    T3 a;
    T4 metodo1(T5 p) throws T6 {
        T7 v;
        ...
    }
    void metodo2() {
        T8 = new T8();
        try {
            ...
        } catch (T9 e) { ... }
    }
}
```

Acoplamento (CBO)

- CBO(A) = 9

Complexidade (CC)

- Chamada de CC (Complexidade Ciclomática)
- Medir a complexidade do código de uma função ou método
- *CC = "número de comandos de decisão em uma função" + 1*
- Em que comando de decisão podem ser `if`, `while`, `case`, `for`, etc.
- O menor valor de CC = 1 (não há comandos de decisão)
- Limite superior aceitável, mas não mágico, é CC = 10