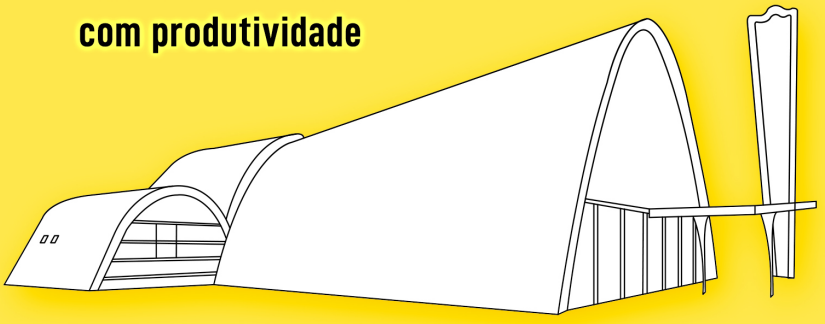
ENGENHARIA *DE* SOFTWARE MODERNA

Princípios e práticas para desenvolvimento de software com produtividade



MARCO TULIO VALENTE

Engenharia de Software Moderna

Princípios e Práticas para Desenvolvimento de Software com Produtividade

Marco Tulio Valente

Versão 2020.1.0 - LeanPub

Para Cynthia, Daniel e Mariana.

Conteúdo

Capítulo 1

Princípios de Projeto

> *The most fundamental problem in computer science is problem decomposition: how to take a complex problem and divide it up into pieces that can be solved independently.* – John Ousterhout

Este capítulo inicia com uma introdução ao projeto de software, na qual procuramos definir e motivar a importância desse tipo de atividade (Seção 5.1). Em seguida, discutimos diversas considerações relevantes em projetos de software. Especificamente, tratamos de Integridade Conceitual (Seção 5.2), Ocultamento de Informação (Seção 5.3), Coesão (Seção 5.4) e Acoplamento (Seção 5.5). Na Seção 5.6 discutimos um conjunto de princípios de projeto, incluindo: Responsabilidade Única, Segregação de Interfaces, Inversão de Dependências, Prefira Composição a Herança, Demeter, Aberto/Fechado e Substituição de Liskov. Por fim, tratamos de métricas para avaliar a qualidade de projetos de software (Seção 5.7).

1.1 Introdução

A afirmação de John Ousterhout que abre este capítulo é uma excelente definição para **projeto de software**. Apesar de não afirmar explicitamente, a citação assume que quando falamos de projeto estamos procurando uma solução para um determinado problema. No contexto de Engenharia de Software, esse problema consiste na implementação de um sistema que atenda aos requisitos funcionais e não-funcionais definidos por um cliente — ou Dono do Produto, para usar um termo mais moderno. Prosseguindo, Ousterhout sugere como devemos proceder para chegar a essa solução: devemos decompor, isto é, quebrar o problema inicial,

que pode ser bastante complexo, em partes menores. Por fim, a frase impõe uma restrição a essa decomposição: ela deve permitir que cada uma das partes do projeto possa ser resolvida (ou implementada) de forma independente.

Essa explicação pode passar a impressão de que projeto é uma atividade simples. No entanto, no projeto de software temos que combater um grande inimigo: a **complexidade** que caracteriza sistemas modernos de software. Talvez, por isso, Ousterhout mencione que a decomposição de um problema em partes independentes é uma questão fundamental, não apenas em Engenharia de Software, mas em toda Ciência da Computação!

Uma estratégia importante para combater a complexidade de sistemas de software passa pela criação de **abstrações**. Uma abstração — pelo menos em Computação — é uma representação simplificada de uma entidade. Apesar de simplificada, ela nos permite interagir e tirar proveito da entidade abstraída, sem que tenhamos que dominar todos os detalhes envolvidos na sua implementação. Funções, classes, interfaces, pacotes, bibliotecas, etc são os instrumentos clássicos oferecidos por linguagens de programação para criação de abstrações.

Em resumo, o primeiro objetivo de projeto de software é decompor um problema em partes menores. Além disso, deve ser possível implementar tais partes de forma independente. Por fim, mas não menos importante, essas partes devem ser abstratas. Em outras palavras, a implementação delas pode ser desafiadora e complexa, mas apenas para os desenvolvedores envolvidos em tal tarefa. Para os demais desenvolvedores, deve ser simples usar a abstração que foi criada.

1.1.1 Exemplo

Para ilustrar essa introdução a projetos de software, vamos usar o exemplo de um compilador. Os requisitos no caso são claros: dado um programa em uma linguagem X devemos convertê-lo em um programa em uma linguagem Y, que costuma ser a linguagem de uma máquina. No entanto, o projeto de um compilador não é trivial. Então, após anos de pesquisa, descobriu-se uma solução — ou projeto — para esse tipo de sistema, a qual é ilustrada na figura da próxima página.

O problema inicial — projetar um compilador — foi decomposto em quatro problemas menores, que vamos descrever brevemente neste parágrafo. Primeiro, temos que implementar um analisador léxico, que vai ler o arquivo de entrada e dividi-lo em tokens (como `if`, `for`, `while`, `x`, `+`, etc).

Depois, temos que implementar um analisador sintático, que vai analisar as tokens e verificar se elas respeitam a gramática da linguagem fonte. Feito isso, ele deve "hierarquizar" essas tokens, isto é, transformá-las em uma estrutura conhecida

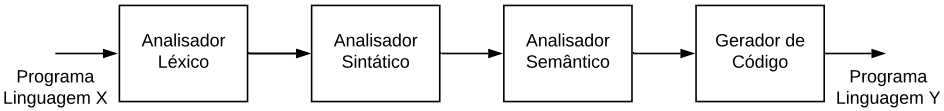


Figura 1.1: Principais módulos de um compilador

como Árvore de Sintaxe Abstrata (AST). Por fim, temos o analisador semântico, que detecta, por exemplo, erros de tipo; e o gerador de código, que vai converter a representação do programa para uma linguagem de mais baixo nível, que possa ser executada por um determinado hardware.

Essa descrição do projeto de um compilador é bastante simples e resumida. Mesmo assim, ela deixa claro o primeiro objetivo do projeto de um software: decompor um problema em partes menores. No nosso exemplo, o problema inicial tornou-se mais concreto, pois agora temos quatro problemas menores para resolver. Isto é, temos que (1) projetar e implementar um analisador léxico, (2) um analisador sintático, (3) um analisador semântico e (4) um gerador de código. Ainda existem desafios importantes em cada uma dessas tarefas, mas estamos mais perto de uma solução para o problema proposto inicialmente.

Continuando com o exemplo, vamos agora focar na implementação de um analisador léxico, a qual envolve certos desafios. No entanto, eles devem ser uma preocupação apenas dos desenvolvedores que ficaram responsáveis por essa parte do sistema. Para os demais desenvolvedores, deve ser possível usar o analisador léxico da forma mais simples possível. Por exemplo, apenas chamando uma função que retorna a próxima token do arquivo de entrada, como no seguinte código:

```
String token = Scanner.next_token();
```

Em resumo, a complexidade envolvida na implementação de um analisador léxico está abstraída (ou, se preferir, encapsulada) na função `next_token()`, cujo uso é bem simples.

1.1.2 O Que Vamos Estudar?

É verdade que o projeto de sistemas de software depende de experiência e, em alguma medida, também de talento e criatividade. No entanto, existem algumas propriedades importantes no projeto de sistemas. Por isso, estudar e conhecer essas **propriedades de projeto** pode ajudar na concepção de sistemas com maior

qualidade. No restante deste capítulo, iremos estudar as seguintes propriedades de projetos de software: integridade conceitual, ocultamento de informação, coesão e acoplamento. Para tornar o estudo mais prático, iremos, em seguida, enunciar alguns **princípios de projeto**, os quais representam diretrizes para se garantir que um projeto atende a determinadas propriedades. Para concluir, vamos descrever métricas para quantificar propriedades como coesão, acoplamento e complexidade.

Aviso: Os assuntos discutidos neste capítulo aplicam-se a **projeto orientado a objetos**. Ou seja, a suposição é que o sistema será implementado em linguagens como Java, C++, C#, Python, Go, Ruby, etc. Certamente, alguns dos temas discutidos valem para projetos que serão implementados em linguagens estruturadas (como C) ou em linguagens funcionais (como Haskell, Clojure ou Erlang). Mas não podemos garantir que oferecemos uma cobertura completa dos aspectos de projeto mais importantes em tais casos.

1.2 Integridade Conceitual

Integridade conceitual é uma propriedade de projeto proposta por Frederick Brooks — o mesmo da Lei de Brooks mencionada no Capítulo 1. O princípio foi enunciado em 1975, na primeira edição do livro *The Mythical Man-Month* (link). Brooks defende que um sistema não pode ser um amontoado de funcionalidades, sem coerência e coesão entre elas. Integridade conceitual é importante porque facilita o uso e entendimento de um sistema por parte de seus usuários. Por exemplo, com integridade conceitual, o usuário acostumado a usar uma parte de um sistema se sente confortável a usar uma outra parte, pois as funcionalidades e a interface implementadas ao longo do produto são consistentes.

Para citar um contra-exemplo, isto é, um caso de ausência de integridade conceitual, vamos assumir um sistema que usa tabelas para apresentar seus resultados. Dependendo da tela do sistema na qual são usadas, essas tabelas possuem leiautes diferentes, em termos de tamanho de fontes, uso de negrito, espaçamento entre linhas, etc. Além disso, em algumas tabelas pode-se ordenar os dados clicando-se no título das colunas, mas em outras tabelas essa funcionalidade não está disponível. Por fim, os valores são mostrados em moedas distintas. Em algumas tabelas, os valores referem-se a reais; em outras tabelas, eles referem-se a dólares. Essa falta de padronização é um sinal de falta de integridade conceitual e, como afirmamos, ela adiciona complexidade acidental no uso e entendimento do sistema.

Na primeira edição do seu livro, Brooks faz uma defesa enfática do princípio, afirmando que:

> "Integridade conceitual é a consideração mais importante no projeto de sistemas. É melhor um sistema omitir algumas funcionalidades e melhorias anômalas, de forma a oferecer um conjunto coerente de ideias, do que oferecer diversas ideias interessantes, mas independentes e descoordenadas."

Em 1995, em uma edição comemorativa dos 20 anos do lançamento do livro (link), Brooks voltou a defender o princípio, ainda com mais ênfase:

> "Hoje, eu estou mais convencido do que antes. Integridade conceitual é fundamental para qualidade de produtos de software."

Sempre que falamos de integridade conceitual, surge uma discussão sobre se o princípio requer que uma autoridade central — um único arquiteto ou gerente de produto, por exemplo — seja responsável por decidir quais funcionalidades serão incluídas no sistema. Sobre essa questão, temos que ressaltar que essa pré-condição — o projeto ser liderado por uma pessoa apenas — não faz parte da definição de integridade conceitual. No entanto, existe um certo consenso de que decisões importantes de projeto não devem ficar nas mãos de um grande comitê, onde cada membro tem direito a um voto. Quando isso ocorre, a tendência é a produção de sistemas com mais funcionalidades do que o necessário, isto é, sistemas sobrecarregados (*bloated systems*). Por exemplo, um grupo pode defender uma funcionalidade A e outro grupo defender uma funcionalidade B. Talvez, as duas não sejam necessárias; porém, para obter consenso, o comitê acaba decidindo que ambas devem ser implementadas. Assim, os dois grupos vão ficar satisfeitos, embora a integridade conceitual do sistema ficará comprometida. Existe uma frase que resume o que acabamos de discutir; ela afirma que "um camelo é um cavalo que foi projetado por um comitê".

Nos parágrafos anteriores, enfatizamos o impacto da falta de integridade conceitual nos usuários finais de um sistema. No entanto, o princípio se aplica também ao design e código de um sistema. Nesse caso, os afetados são os desenvolvedores, que terão mais dificuldade para entender, manter e evoluir o sistema. A seguir, mencionamos alguns exemplos de falta de integridade conceitual em nível de código ou design:

- Quando uma parte do sistema usa um padrão de nomes para variáveis (por exemplo, *camel case*, como em `notaTotal`), enquanto em outra parte usa-se um outro padrão (por exemplo, *snake case*, como em `nota_total`).
- Quando uma parte do sistema usa um determinado framework para manipulação de páginas Web, enquanto em outra parte usa-se um segundo framework ou então uma versão diferente do primeiro framework.
- Quando em uma parte do sistema resolve-se um problema usando-se uma estrutura de dados X, enquanto que, em outra parte, um problema parecido é resolvido por meio de uma estrutura Y.
- Quando funções de uma parte do sistema que precisam de uma determinada informação — por exemplo, o endereço de um servidor — a obtém diretamente de um arquivo de configuração. Porém, em outras funções, de outras partes do sistema, a mesma informação deve ser passada como parâmetro.

Esses exemplos revelam uma falta de padronização e, logo, de integridade conceitual. Eles são um problema porque tornam mais difícil um desenvolvedor acostumado a manter uma parte do sistema ser alocado para manter uma outra parte.

Mundo Real: Samuel Roso e Daniel Jackson, pesquisadores do MIT, nos EUA, dão um exemplo real de sistema que implementa duas funcionalidades com propósitos semelhantes — o que também revela uma falta de integridade conceitual (link). Segundo eles, em um conhecido sistema de blogs, quando um usuário incluía um sinal de interrogação no título de um post, uma janela era aberta, solicitando que ele informasse se desejava receber respostas para esse post. No entanto, os pesquisadores argumentam que essa possibilidade deixava os usuários confusos, pois já existia no sistema a possibilidade de comentar posts. Logo, a confusão acontecia devido a duas funcionalidades parecidas: comentários (em posts normais) e respostas (em posts cujos títulos terminavam com um ponto de interrogação).

1.3 Ocultamento de Informação

Essa propriedade, uma tradução da expressão *information hiding*, foi discutida pela primeira vez em 1972, por David Parnas, em um dos artigos mais importantes e influentes da área de Engenharia de Software, de todos os tempos, cujo título é *On the criteria to be used in decomposing systems into modules* (link). O resumo do artigo começa da seguinte forma:

> "Este artigo discute modularização como sendo um mecanismo capaz de tornar sistemas de software mais flexíveis e fáceis de entender e, ao mesmo tempo, reduzir o tempo de desenvolvimento deles. A efetividade de uma determinada modularização depende do critério usado para dividir um sistema em módulos."

Aviso: Parnas usa o termo *módulo* no seu artigo, mas isso em uma época em que orientação a objetos ainda não havia surgido, pelo menos como conhecemos hoje. Já neste capítulo, escrito quase 50 anos após o trabalho de Parnas, optamos pelo termo **classe**, em vez de módulo. O motivo é que classes são a principal unidade de modularização de linguagens de programação modernas, como Java, C++, Ruby, etc. No entanto, o conteúdo do capítulo aplica-se a outras unidades de modularização, incluindo aquelas menores do que classes, como métodos e funções; e também a unidades maiores, como pacotes.

Ocultamento de informação traz as seguintes vantagens para um sistema:

- **Desenvolvimento em paralelo.** Suponha que um sistema X foi implementado por meio de classes C1, C2, ..., Cn. Quando essas classes ocultam suas principais informações, fica mais fácil implementá-las em paralelo, por desenvolvedores diferentes. Consequentemente, teremos uma redução no tempo total de implementação do sistema.
- **Flexibilidade a mudanças.** Por exemplo, suponha que descobrimos que a classe Ci é responsável pelos problemas de desempenho do sistema. Quando detalhes de implementação de Ci são ocultados do resto do sistema, fica mais fácil trocar sua implementação por uma classe Ci', que use estruturas de dados e algoritmos mais eficientes. Essa troca também é mais segura, pois como as classes são independentes, diminui-se o risco de a mudança introduzir bugs em outras classes.
- **Facilidade de entendimento.** Por exemplo, um novo desenvolvedor contratado pela empresa pode ser alocado para trabalhar em algumas classes apenas. Portanto, ele não precisará entender toda a complexidade do sistema, mas apenas a implementação das classes pelas quais ficou responsável.

No entanto, para se atingir os benefícios acima, classes devem satisfazer à seguinte condição (ou critério): elas devem esconder decisões de projeto que são sujeitas a mudanças. Devemos entender decisão de projeto como qualquer aspecto de projeto da classe, como os requisitos que ela implementa ou os algoritmos e estruturas de dados que serão usados no seu código. Portanto, ocultamento de informação recomenda que classes devem esconder detalhes de implementação que

estão sujeitos a mudanças. Modernamente, os atributos e métodos que uma classe pretende encapsular são declarados com o modificador de visibilidade **privado**, disponível em linguagens como Java, C++, C# e Ruby.

Porém, se uma classe encapsular toda a sua implementação ela não será útil. Dito de outra forma, uma classe para ser útil deve tornar alguns de seus métodos públicos, isto é, permitir que eles possam ser chamados por código externo. Código externo que chama métodos de uma classe é dito ser **cliente** da classe. Dizemos também que o conjunto de métodos públicos de uma classe define a sua **interface**. A definição da interface de uma classe é muito importante, pois ela constitui a sua parte visível.

Interfaces devem ser estáveis, pois mudanças na interface de uma classe podem implicar em atualizações em seus clientes. Para ser mais claro, suponha uma classe Math, com métodos que realizam operações matemáticas. Suponha um método sqrt, que calcula a raiz quadrada de seu parâmetro. Suponha ainda que a assinatura desse método seja alterada — para, por exemplo, retornar uma exceção caso o valor do parâmetro seja negativo. Essa alteração terá impacto em todo código cliente do método sqrt, que deverá ser alterado para tratar a nova exceção.

1.3.1 Exemplo

Suponha um sistema para controle de estacionamentos. Suponha ainda que, em uma primeira versão, a classe principal desse sistema seja a seguinte:

```
import java.util.Hashtable;
public class Estacionamento {
public Hashtable<String, String> veiculos;
public Estacionamento() {
veiculos = new Hashtable<String, String>();
}
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");
}
}
```

Essa classe tem um problema de exposição excessiva de informação ou, em outras palavras, ela não oculta estruturas que podem mudar no futuro. Especificamente, a tabela hash que armazena os veículos estacionados no estacionamento é pública. Com isso, clientes — como o método `main` — têm acesso direto a ela para, por exemplo, adicionar veículos no estacionamento. Se, futuramente, decidirmos usar uma outra estrutura de dados para armazenar os veículos, todos os clientes deverão ser modificados.

Suponha que o sistema de estacionamento fosse manual, com o nome dos veículos anotados em uma folha de papel. Fazendo uma comparação, essa primeira versão da classe `Estacionamento` corresponderia — no caso desse sistema manual — ao cliente do estacionamento, após estacionar seu carro, entrar na cabine de controle e escrever ele mesmo a placa e o modelo do seu carro na folha de controle.

Já a próxima versão da classe é melhor, pois ela encapsula a estrutura de dados responsável por armazenar os veículos. Para estacionar um veículo, existe agora o método estaciona. Com isso, os desenvolvedores da classe têm liberdade para trocar de estrutura de dados, sem causar impacto nos seus clientes. A única restrição é que a assinatura do método `estaciona` deve ser preservada.

```
import java.util.Hashtable;
public class Estacionamento {
private Hashtable<String,String> veiculos;
public Estacionamento() {
veiculos = new Hashtable<String, String>();
}
public void estaciona(String veiculo, String placa) {
veiculos.put(veiculo, placa);
}
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");
}
}
```

Em resumo, essa nova versão oculta uma estrutura de dados — sujeita a alterações durante a evolução do sistema — e disponibiliza uma interface estável para os

clientes da classe — representada pelo método `estaciona`, que só requer dois parâmetros do tipo `String`.

Mundo Real: Em 2002, consta que Jeff Bezos, dono da Amazon, enviou um mail para todos os desenvolvedores da empresa, com um conjunto de diretrizes para projeto de software que eles deveriam obrigatoriamente seguir a partir de então. Reproduzimos a mensagem na tabela a seguir (apenas fizemos adaptações cosméticas para ela ficar mais clara em Português; essa mesma mensagem é mencionada no livro de Fox e Patterson (link, Cap. 1, Seção 1.4):

1. Todos os times devem, daqui em diante, garantir que os sistemas exponham seus dados e funcionalidades por meio de interfaces.
2. Os sistemas devem se comunicar apenas por meio de interfaces.
3. Não deve haver outra forma de comunicação: sem links diretos, sem leituras diretas em bases de dados de outros sistemas, sem memória compartilha ou variáveis globais ou qualquer tipo de back-doors. A única forma de comunicação permitida é por meio de interfaces.
4. Não importa qual tecnologia vocês vão usar: HTTP, CORBA, Pubsub, protocolos específicos — isso não interessa. Bezos não liga para isso.
5. Todas as interfaces, sem exceção, devem ser projetadas para uso externo. Ou seja, os times devem planejar e projetar interfaces pensando em usuários externos. Sem nenhuma exceção à regra.
6. Quem não seguir essas recomendações está demitido.
7. Obrigado; tenham um excelente dia!

Essas recomendações podem ser resumidas da seguinte forma: todos os desenvolvedores da Amazon deveriam, após receber a mensagem, seguir o princípio de ocultamento de informação, enunciado por David Parnas em 1972.

1.3.2 Getters e Setters

Métodos `get` e `set` — muitas vezes chamados apenas de *getters* e *setters* — são muito usados em linguagens orientadas a objetos, como Java e C++. A recomendação para uso desses métodos é a seguinte: todos os dados de uma classe devem ser privados e o acesso a eles — se necessário — deve ocorrer por meio de getters (acesso de leitura) e setters (acesso de escrita).

Veja um exemplo a seguir, onde métodos `get` e `set` são usados para acessar o atributo `matricula` de uma classe `Aluno`.

```
class Aluno {
private int matricula;
...
public int getMatricula() {
return matricula;
}
public setMatricula(int matricula) {
this.matricula = matricula;
}
...
}
```

No entanto, getters e setters não são uma garantia de que estamos ocultando dados da classe, como mencionado em alguns livros e discussões pela Internet. Pelo contrário, eles são um instrumento de liberação de informação (*information leakage*). Veja o que John Ousterhout diz sobre esses métodos (link, Seção 19.6):

> "Embora possa fazer sentido usar getters e setters para expor dados privados de uma classe, é melhor evitar essa exposição logo de início. Ela torna parte da implementação da classe visível externamente, o que viola a ideia de ocultamento de informação e aumenta a complexidade da interface da classe."

Em resumo: certifique-se de que é imprescindível liberar informação privativa de uma classe. Se isso for, de fato, importante, considere a ideia de implementar essa liberação por meio de getters e setters — e não tornando o atributo público.

No nosso exemplo, vamos então assumir que é imprescindível que os clientes possam ler e alterar a matrícula de alunos. Assim, é melhor que o acesso a esse atributo seja feito por meio de métodos `get` e `set`, pois eles constituem uma interface mais estável para tal acesso, pelos seguintes motivos:

- No futuro, podemos precisar de recuperar a matrícula de um banco de dados, ou seja, ela não estará mais em memória. Essa nova lógica poderá, então, ser implementada no método get, sem impactar nenhum cliente da classe.
- No futuro, podemos precisar de adicionar um dígito verificador nas matrículas. Essa lógica — cálculo e incorporação do dígito verificador — poderá ser implementada no método set, sem impactar os seus clientes.

Além disso, getters e setters são requeridos por algumas bibliotecas, tais como bibliotecas de depuração, serialização e mocks (iremos estudar mais sobre mocks no capítulo de Testes).

1.4 Coesão

A implementação de qualquer classe deve ser coesa, isto é, toda classe deve implementar uma única funcionalidade ou serviço. Especificamente, todos os métodos e atributos de uma classe devem estar voltados para a implementação do mesmo serviço. Uma outra forma de explicar coesão é afirmando que toda classe deve ter uma única responsabilidade no sistema. Ou, ainda, afirmando que deve existir um único motivo para modificar uma classe.

Coesão tem as seguintes vantagens:

- Facilita a implementação de uma classe, bem como o seu entendimento e manutenção.
- Facilita a alocação de um único responsável por manter uma classe.
- Facilita o reúso e teste de uma classe, pois é mais simples reusar e testar uma classe coesa do que uma classe com várias responsabilidades.

Separação de interesses (*separation of concerns*) é uma outra propriedade desejável em projeto de software, a qual é muito semelhante ao conceito de coesão. Ela defende que uma classe deve implementar apenas um **interesse** (*concern*). Nesse contexto, o termo interesse se refere a qualquer funcionalidade, requisito ou responsabilidade da classe. Portanto, as seguintes recomendações são equivalentes: (1) uma classe deve ter uma única responsabilidade; (2) uma classe deve implementar um único interesse; (3) uma classe deve ser coesa.

1.4.1 Exemplos

Exemplo 1: A discussão anterior foi voltada para coesão de classes. No entanto, o conceito se adapta também a métodos ou funções. Por exemplo, suponha uma função como a seguinte:

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

Essa função — que consiste em um exemplo extremo e, queremos acreditar, pouco comum na prática — apresenta um problema sério de coesão, pois ela faz duas coisas: calcula o seno ou o cosseno de seu argumento. O recomendável, em casos como esse, é criar funções separadas para cada uma dessas tarefas.

Exemplo 2: Suponha agora a seguinte classe:

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

Trata-se de uma classe coesa, pois todos os seus métodos implementam operações importantes em uma estrutura de dados do tipo Pilha.

Exemplo 3: Para concluir a lista de exemplos, vamos voltar à classe `Estacionamento`, na qual foram adicionados agora quatro atributos com informações sobre o gerente do estacionamento:

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

A principal responsabilidade dessa classe é gerenciar a operação do estacionamento, incluindo métodos como `estaciona()`, `calcula_preco()`, `libera_veiculo()`, etc. Portanto, ela não deveria assumir responsabilidades relacionadas com o gerenciamento dos funcionários do estacionamento. Para isso, poderia ser criada uma segunda classe, chamada, por exemplo, `Funcionario`.

1.5 Acoplamento

Acoplamento é a força (*strength*) da conexão entre duas classes. Apesar de parecer simples, o conceito possui algumas nuances, as quais derivam da existência de dois tipos de acoplamento entre classes: acoplamento aceitável e acoplamento ruim.

Dizemos que existe um **acoplamento aceitável** de uma classe A para uma classe B quando:

- A classe A usa apenas métodos públicos da classe B.
- A interface provida por B é estável do ponto de vista sintático e semântico. Isto é, as assinaturas dos métodos públicos de B não mudam com frequência; e o mesmo acontece como o comportamento externo de tais métodos. Por isso, são raras as mudanças em B que terão impacto na classe A.

Por outro lado, existe um **acoplamento ruim** de uma classe A para uma classe B quando mudanças em B podem facilmente impactar A. Isso ocorre principalmente nas seguintes situações:

- Quando a classe A realiza um acesso direto a um arquivo ou banco de dados da classe B.
- Quando as classes A e B compartilham uma variável ou estrutura de dados global. Por exemplo, a classe B altera o valor de uma variável global que a classe A usa no seu código.
- Quando a interface da classe B não é estável. Por exemplo, os métodos públicos de B são renomeados com freqüência.

Em essência, o que caracteriza o acoplamento ruim é o fato de que a dependência entre as classes não é mediada por uma interface estável. Por exemplo, quando uma classe altera o valor de uma variável global, ela não tem consciência do impacto dessa mudança em outras partes do sistema. Por outro lado, quando uma classe altera sua interface, ela está ciente de que isso vai ter impacto nos clientes, pois a função de uma interface é exatamente anunciar os serviços que uma classe oferece para o resto do sistema.

Resumindo: acoplamento pode ser de grande utilidade, principalmente quando ocorre com a interface de uma classe estável que presta um serviço relevante para a classe de origem. Já o acoplamento ruim deve ser evitado, pois é um acoplamento não mediado por interfaces. Mudanças na classe de destino do acoplamento podem facilmente se propagar para a classe de origem.

Frequentemente, as recomendações sobre acoplamento e coesão são reunidas em uma única recomendação:

> Maxime a coesão das classes e minimize o acoplamento entre elas.

De fato, se uma classe depende de muitas outras classes, por exemplo, de dezenas de classes, ela pode estar assumindo responsabilidades demais, na forma de funcionalidades não coesas. Lembre-se que uma classe deve ter uma única responsabilidade (ou um único motivo para ser modificada). Por outro lado, devemos tomar cuidado com o significado do verbo minimizar. O objetivo não deve ser eliminar completamente o acoplamento de uma classe com outras classes, pois é natural que uma classe precise de outras classes, principalmente daquelas que implementam serviços básicos, como estruturas de dados, entrada/saída, etc.

1.5.1 Exemplos

Exemplo 1: Suponha a classe `Estacionamento`, usada na Seção 5.2, a qual possui um atributo que é uma `Hashtable`. Logo, dizemos que `Estacionamento` está acoplada a `Hashtable`. No entanto, na nossa classificação, trata-se de um acoplamento aceitável, isto é, ele não deve ser motivo de preocupação, pelos seguintes motivos:

- `Estacionamento` só usa métodos públicos de `Hashtable`.
- A interface de `Hashtable` é estável, já que ela faz parte do pacote oficial de estruturas de dados de Java (estamos supondo que o sistema será implementado nessa linguagem). Assim, uma alteração na assinatura dos métodos públicos de `Hashtable` quebraria não apenas nossa classe `Estacionamento`, mas talvez milhões de outras classes de diversos sistemas Java ao redor do mundo.

Exemplo 2: Suponha o seguinte trecho de código, no qual existe um arquivo compartilhado por duas classes, `A` e `B`, mantidas por desenvolvedores distintos. O método `B.g()` grava um inteiro no arquivo, que é lido por `A.f()`. Essa forma de comunicação origina um acoplamento ruim entre as classes. Por exemplo, o desenvolvedor que implementa `B` pode não saber que o arquivo é lido por `A`. Assim, ele pode decidir mudar o formato do arquivo por conta própria, sem comunicar o desenvolvedor da classe `A`.

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

Antes de avançar, um pequeno comentário: no exemplo, existe também um acoplamento entre `B` e `File`. Porém, ele é um acoplamento aceitável, pois `B` realmente precisa persistir seus dados. Então, para conseguir isso, nada melhor do que usar uma classe da biblioteca de entrada e saída da linguagem.

Exemplo 3: Uma solução melhor para o acoplamento entre as classes `A` e `B` do exemplo anterior é mostrada no código a seguir.

```
class A {                          class B {
private void f(B b) {              int total;
int total;                         public int getTotal(){
total = b.getTotal();              return total;
...                                }
}                                  private void g() {
}                                  // computa valor de total
                                   File f = File.open("arq1");
                                   f.writeInt(total);
                                   ...
                                   }
                                   }
```

Nessa nova versão, a dependência entre `A` e `B` é tornada explícita. Agora, `B` possui um método público que retorna o valor `total`. E a classe `A` possui uma dependência para a classe `B`, por meio de um parâmetro do método `f`. Esse parâmetro é usado para requisitar explicitamente o valor de `total`, chamando-se o método `getTotal()`. Como esse método foi declarado público em B, espera-se que o desenvolvedor

dessa classe se esforce para não alterar a sua assinatura. Por isso, nessa nova versão, dizemos que, apesar de existir uma dependência de `A` para `B`, o acoplamento criado por ela é aceitável. Em outras palavras, não é um acoplamento que gera preocupações.

Ainda sobre o exemplo anterior, é interessante mencionar que, na primeira versão, o código de `A` não declara nenhuma variável ou atributo do tipo `B`. E, mesmo assim, temos um acoplamento ruim entre as classes. Na segunda versão, ocorre o contrário, pois o método `A.f()` declara um parâmetro do tipo `B`. Mesmo assim, o acoplamento entre as classes é de melhor qualidade, pois é mais fácil estudar e manter o código de `A` sem conhecer detalhes de `B`.

Alguns autores usam ainda os termos acoplamento estrutural e acoplamento evolutivo (ou lógico), com o seguinte significado:

- **Acoplamento estrutural** entre A e B ocorre quando uma classe A possui uma referência explícita em seu código para uma classe B. Por exemplo, o acoplamento entre `Estacionamento` e `Hashtable` é estrutural.
- **Acoplamento evolutivo (ou lógico)** entre A e B ocorre quando mudanças na classe B tendem a se propagar para a classe A. No exemplo mencionado, no qual a classe A depende de um inteiro armazenado em um arquivo interno de B, não existe acoplamento estrutural entre A e B, pois A não declara nenhuma variável do tipo B, mas existe acoplamento evolutivo. Por exemplo, mudanças no formato do arquivo criado por B terão impacto em A.

Acoplamento estrutural pode ser aceitável ou ruim, dependendo da estabilidade da interface da classe de destino. Acoplamento evolutivo, principalmente quando qualquer mudança em B se propaga para a classe de origem A, representa um acoplamento ruim.

Kent Beck — na época em que trabalhou no Facebook — criou um glossário de termos relacionados com projeto de software. Nesse glossário, acoplamento é definido da seguinte forma (link):

> "Dois elementos estão acoplados quando mudanças em um elemento implicam em mudanças em um outro elemento ... Acoplamento pode dar origem a uma relação bem sutil entre classes, como frequentemente observamos no Facebook. Certos eventos que interrompem o funcionamento de uma parte do sistema normalmente são causados por pequenos bits de acoplamento que não são esperados — por exemplo, mudanças na configuração do sistema A causam um time-out no sistema B, que causa uma sobrecarga no sistema C."

A definição de acoplamento proposta por Beck — "quando mudanças em um elemento implicam em mudanças em um outro elemento" — corresponde à definição de acoplamento evolutivo. Ou seja, parece que Beck não se preocupa com o acoplamento aceitável (isto é, estrutural e estável) entre duas classes; pois ele, de fato, não deve ser motivo de preocupação.

O comentário também deixa claro que acoplamento pode ser indireto. Isto é, mudanças em A podem ser propagar para B, e então alcançar C. Nesse caso, C está acoplado a A, mas de forma indireta.

Mundo Real: Um exemplo de problema real causado por acoplamento indireto ficou conhecido como **episódio do left-pad**. Em 2016, uma disputa de direitos autorais motivou um desenvolvedor a remover uma de suas bibliotecas do diretório npm, muito usado para armazenar e distribuir bibliotecas node.js/JavaScript. A biblioteca removida — chamada leftPad — tinha uma única função JavaScript, de nome leftPad, com apenas 11 linhas de código. Ela preenchia uma string com brancos à esquerda. Por exemplo, `leftPad('foo', 5)` iria retornar `' foo'`, ou seja, `'foo'` com dois brancos à esquerda.
Milhares de sistemas Web dependiam dessa função trivial, porém a dependência ocorria de modo indireto. Os sistemas usavam o npm para baixar dinamicamente o código JavaScript de uma biblioteca B1, que por sua vez dependia de uma biblioteca B2 cujo código também estava no npm e, assim por diante, até alcançar uma biblioteca Bn que dependia do left-pad. Como resultado, todos os sistemas que dependiam do left-pad — de forma direta ou indireta — ficaram fora do ar por algumas horas, até que a biblioteca fosse inserida de novo no npm. Em resumo, os sistemas foram afetados por um problema em uma biblioteca trivial; e eles não tinham a menor ideia de que estavam acoplados a ela.

1.6 Princípios de Projeto

Princípios de projeto são recomendações mais concretas que desenvolvedores de software devem seguir para atender às propriedades de projeto que estudamos na seção anterior. Assim, propriedades de projeto podem ser vistas como recomendações ainda genéricas (ou táticas), enquanto que os princípios que estudaremos agora estão em um nível operacional.

Nesta seção, iremos estudar os sete princípios de projeto listados na próxima tabela. A tabela mostra ainda as propriedades de projeto que são contempladas ao seguir cada um desses princípios.

Princípio de Projeto	**Propriedade de Projeto**
Responsabilidade Única	Coesão
Segregação de Interfaces	Coesão
Inversão de Dependências	Acoplamento
Prefira Composição a Herança	Acoplamento
Demeter	Ocultamento de Informação
Aberto/Fechado	Extensibilidade
Substituição de Liskov	Extensibilidade

Cinco dos princípios que vamos estudar são conhecidos como **Princípios SOLID**, que é uma sigla cunhada por Robert Martin e Michael Feathers (link). Ela deriva da letra inicial de cada princípio, em inglês:

- **S**ingle Responsibility Principle
- **O**pen Closed/Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Os princípios de projeto que vamos estudar têm um ponto em comum: eles não visam apenas "resolver" um problema, mas também assegurar que a solução encontrada possa ser mantida e evoluída com sucesso, no futuro. Os maiores problemas com projetos de software costumam ocorrer após a implementação, quando o sistema precisa ser mantido. Normalmente, existe uma tendência de que essa manutenção fique gradativamente mais lenta, custosa e arriscada. Portanto, os princípios de projeto que estudaremos tentam reduzir ou postergar essa contínua degradação da qualidade interna de sistemas de software. Em resumo, o objetivo não é apenas entregar um projeto capaz de resolver um problema, mas também

que facilite manutenções futuras. Lembre-se que a principal regra sobre requisitos de software é que eles mudam com freqüência. O mesmo acontece com tecnologias de implementação, como bibliotecas e frameworks.

1.6.1 Princípio da Responsabilidade Única

Esse princípio é uma aplicação direta da ideia de coesão. Ele propõe o seguinte: toda classe deve ter uma única responsabilidade. Mais ainda, responsabilidade, no contexto do princípio, significa "motivo para modificar uma classe". Ou seja, deve existir um único motivo para modificar qualquer classe em um sistema.

Um corolário desse princípio recomenda separar **apresentação** de **regras de negócio**. Portanto, um sistema deve possuir classes de apresentação, que vão tratar de aspectos de sua interface com os usuários, formato das mensagens, meio onde as mensagens serão exibidas, etc. E classes responsáveis por regras de negócio, isto é, que vão realizar as computações, processamento, análises, etc. São interesses e responsabilidades diferentes. E que podem evoluir e sofrer modificações por razões distintas. Portanto, elas devem ser implementadas em classes diferentes. Por esse motivo, não é surpresa que existam desenvolvedores que tratam apenas de requisitos de *front-end* (isto é, de classes de apresentação) e desenvolvedores que tratam de requisitos de *backend* (isto é, de classes com regras de negócio).

Exemplo: A próxima classe ilustra uma violação do Princípio da Responsabilidade Única. O método `calculaIndiceDesistencia` da classe `Disciplina` possui duas responsabilidades: calcular o índice de desistência de uma disciplina e imprimi-lo na console do sistema.

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

Uma solução consiste em dividir essas responsabilidades entre duas classes: uma classe de interface com o usuário (`Console`) e uma classe de "regra de negócio" (`Disciplina`). Dentre outros benefícios, essa solução permite reusar a classe de negócio com outras classes de interface, como classes de interface gráfica, interface web, interface para celular, etc.

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

1.6.2 Princípio da Segregação de Interfaces

Assim como o princípio anterior, esse princípio é uma aplicação da ideia de coesão. Melhor dizendo, ele é um caso particular de Responsabilidade Única com foco em interfaces. O princípio define que interfaces tem que ser pequenas, coesas e, mais importante ainda, específicas para cada tipo de cliente. O objetivo é evitar que clientes dependam de interfaces com métodos que eles não vão usar. Para evitar isso, duas ou mais interfaces específicas podem, por exemplo, substituir uma interface de propósito geral.

Uma violação do princípio ocorre, por exemplo, quando uma interface possui dois conjuntos de métodos M_x e M_y. O primeiro conjunto é usado por clientes C_x (que então não usam os métodos M_y). De forma inversa, os métodos M_y são usados apenas por clientes C_y (que não usam os métodos M_x). Consequentemente, essa interface deveria ser quebrada em duas interfaces menores e específicas: uma interface contendo apenas os métodos M_x e a segunda interface contendo apenas os métodos M_y.

Exemplo: Suponha uma interface `Funcionario` com os seguintes métodos: (1) retornar salário, (2) retornar contribuição mensal para o FGTS (Fundo de Garantia por Tempo de Serviço) e (3) retornar SIAPE (isto é, o "número de matrícula" de todo funcionário público federal). Essa interface viola o Princípio de Segregação de Interfaces, pois apenas funcionários de empresas privadas, contratados em regime de CLT, possuem uma conta no FGTS. Por outro lado, apenas funcionários públicos possuem uma matrícula no SIAPE.

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

Uma alternativa — que atende ao Princípio de Segregação de Interfaces — consiste em criar interfaces específicas (`FuncionarioCLT` e `FuncionarioPublico`) que estendem a interface genérica (`Funcionario`).

```
interface Funcionario {
double getSalario();
...
}
interface FuncionarioCLT extends Funcionario {
double getFGTS();
...
}
interface FuncionarioPublico extends Funcionario {
int getSIAPE();
...
}
```

1.6.3 Princípio de Inversão de Dependências

Esse princípio recomenda que uma classe cliente deve estabelecer dependências prioritariamente com abstrações e não com implementações concretas, pois abstrações (isto é, interfaces) são mais estáveis do que implementações concretas (isto é, classes). A ideia é então trocar (ou "inverter") as dependências: em vez de depender de classes concretas, clientes devem depender de interfaces. Portanto, um nome mais intuitivo para o princípio seria **Prefira Interfaces a Classes**.

Para detalhar a ideia do princípio, suponha que exista uma interface I e uma classe C1 que a implementa. Se puder escolher, um cliente deve se acoplar a I e não a C1. O motivo é que quando um cliente se acopla a uma interface I ele fica imune a mudanças na implementação dessa interface. Por exemplo, em vez de C1, pode-se mudar a implementação para C2, que isso não terá impacto no cliente em questão.

Exemplo 1: O código a seguir ilustra o cenário que acabamos de descrever. Nesse código, o mesmo `Cliente` pode "trabalhar" com objetos concretos das classes `C1` e `C2`. Ele não precisa conhecer a classe concreta que está por trás — ou que implementa — a interface `I` que ele referencia em seu código.

```
interface I { ... }
class C1 implements I {
...
}
class C2 implements I {
...
}
```

```
class Cliente {
I i;
Cliente (I i) {
this.i = i;
...
}
...
}
```

```
class Main {
void main () {
C1 c1 = new C1();
new Cliente(c1);
...
C2 c2 = new C2();
new Cliente(c2);
...
}
}
```

Exemplo 2: Agora, mostramos um exemplo de código que segue o Princípio de Inversão de Dependências. Esse princípio justifica a escolha de `Projetor` como tipo do parâmetro do método `g`. Amanhã, o tipo da variável local `projetor` no método `f` pode mudar para, por exemplo, `ProjetorSamsung`. Se isso vier a acontecer, a implementação de g permanecerá válida, pois ao usarmos um tipo interface estamos nos preparando para receber parâmetros de vários tipos concretos que implementam essa interface.

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

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

Exemplo 3: Como um exemplo final, suponha um pacote de estruturas de dados que oferece uma interface `List` e algumas implementações concretas (classes) para ela, como `ArrayList`, `LinkedList` e `Vector`. Sempre que possível, em código cliente desse pacote, declare variáveis, parâmetros ou atributos usando o tipo `List`, pois assim você estará criando código compatível com as diversas implementações concretas dessa interface.

1.6.4 Prefira Composição a Herança

Antes de explicar o princípio, vamos esclarecer que existem dois tipos de herança:

- **Herança de classes** (exemplo: `class A extends B`), que é aquela que envolve reúso de código. Não apenas neste capítulo, mas em todo o livro, quando mencionarmos apenas o termo herança estaremos nos referindo a herança de classes.
- **Herança de interfaces** (exemplo: `interface I implements J`), que não envolve reúso de código. Essa forma de herança é mais simples e não suscita preocupações. Quando precisarmos de nos referir a ela, iremos usar o nome completo: herança de interfaces.

Voltando ao princípio, quando orientação a objetos se tornou comum, na década de 80, houve um incentivo ao uso de herança. Acreditava-se que o conceito seria talvez uma "bala de prata" capaz de resolver os problemas de reúso de software. Argumentava-se que hierarquias de classes profundas, com vários níveis, seriam um indicativo de um bom projeto, no qual foi possível atingir elevados índices de reúso. No entanto, com o tempo, percebeu-se que herança não era a tal "bala de prata". Pelo contrário, herança tende a introduzir problemas na manutenção e evolução das classes de um sistema. Esses problemas têm sua origem no forte acoplamento que existe entre subclasses e superclasses, conforme descrito por Gamma e colegas no livro sobre padrões de projeto (link):

> "Herança expõe para subclasses detalhes de implementação das classes pai. Logo, frequentemente diz-se que herança viola o encapsulamento das classes pai. A implementação das subclasses se torna tão acoplada à implementação da classe pai que qualquer mudança nessas últimas pode forçar modificações nas subclasses."

O princípio, porém, não proíbe o uso de herança. Mas ele recomenda: se existirem duas soluções de projeto, uma baseada em herança e outra em composição, a solução por meio de composição, normalmente, é a melhor. Só para deixar claro, existe uma relação de **composição** entre duas classes A e B quando a classe A possui um atributo do tipo B.

Exemplo: Suponha que temos que implementar uma classe `Stack`. Existem pelo menos duas soluções — por meio de herança ou por meio de composição — conforme mostra o seguinte código:

Solução via Herança	**Solução via Composição**
class Stack **extends** ArrayList {	**class** Stack {
...	**private** ArrayList elementos;
}	...
	}

A solução por meio de herança não é recomendada por vários motivos, sendo que os principais são os seguintes: (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 força uma solução via herança, a class `Stack` irá herdar métodos como `get` e `set`, que não fazem parte da especificação de pilhas. Portanto, nesse caso, devemos preferir a solução baseada em composição.

Uma segunda vantagem de composição é que a relação entre as classes não é estática, como no caso de herança. No exemplo, se optássemos por herança, a classe `Stack` estaria acoplada estaticamente a `ArrayList`; e não seria possível mudar essa decisão em tempo de execução. Por outro lado, quando adota-se uma solução baseada em composição, isso fica mais fácil, como mostra o exemplo a seguir:

```
class Stack {
private List elementos;
Stack(List elementos) {
this.elementos = elementos;
}
...
}
```

No exemplo, a estrutura de dados que armazena os elementos da pilha passou a ser um parâmetro do construtor da classe `Stack`. Com isso, torna-se possível instanciar objetos `Stack` com estruturas de dados distintas. Por exemplo, um objeto no qual os elementos da pilha são armazenados em um `ArrayList` e outro objeto onde eles são armazenado em um Vector. Como uma observação final, veja que o tipo do atributo elementos de `Stack` passou a ser um List; ou seja, fizemos uso também do Princípio de Inversão de Dependências (ou Prefira Interfaces a Classes).

Antes de concluir, gostaríamos de mencionar três pontos suplementares ao que discutimos sobre “Prefira Composição a Herança”:

- Herança é classificada como um mecanismo de **reuso caixa-branca**, pois as subclasses costumam ter acesso a detalhes de implementação da classe base. Por outro lado, composição é um mecanismo de **reuso caixa-preta**.
- Um padrão de projeto que ajuda a substituir uma solução baseada em herança por uma solução baseada em composição é o Padrão Decorador, que vamos estudar no próximo capítulo.
- Por conta dos problemas discutidos nesta seção, linguagens de programação mais recentes — como Go e Rust — não incluem suporte a herança.

1.6.5 Princípio de Demeter

O nome desse princípio faz referência a um grupo de pesquisa da Northeastern University, em Boston, EUA. Esse grupo, chamado Demeter, desenvolvia pesquisas na área de modularização de software. No final da década de 80, em uma de suas pesquisas, o grupo enunciou um conjunto de regras para evitar problemas de encapsulamento em projeto de sistemas orientados a objetos, as quais ficaram conhecidas como Princípio ou Lei de Demeter.

O Princípio de Demeter — também chamado de **Princípio do Menor Privilégio** (*Principle of Least Privilege*) — defende que a implementação de um método deve invocar 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)

Exemplo: O seguinte código mostra um método, `m1`, com quatro chamadas que respeitam o Princípio de Demeter. E, em seguida, temos um método `m2`, com uma chamada que não obedece ao princípio.

```
class PrincipioDemeter {
T1 attr;
void f1() {
...
}
void m1(T2 p) { // método que segue Demeter
f1(); // caso 1: própria 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();
}
}
```

O método `m2`, ao chamar três métodos `get` em sequência, viola o Princípio de Demeter. O motivo é que os objetos intermediários — retornados pelos métodos `get` — são usados apenas como "passagem" para se chegar ao objeto final, que é aquele que de fato nos interessa e sobre o qual vamos executar uma operação "útil"— no exemplo, `doSomething()`. No entanto, esses objetos intermediários podem existir apenas para liberar informação interna sobre o estado de suas classes. Além de tornar a chamada mais complexa, a informação liberada pode estar sujeita a mudanças. Se isso ocorrer, um dos "elos" da sequência de chamadas será quebrado e o cliente — o método `m2`, no exemplo — terá que descobrir um outro modo de atingir o método final. Em resumo, chamadas que violam o Princípio de Demeter têm grande chance de quebrar o encapsulamento dos objetos de passagem.

Costuma-se dizer que o Princípio de Demeter recomenda que os métodos de uma classe devem falar apenas com seus "amigos", isto é, com métodos da própria classe ou então com métodos de objetos que eles recebem como parâmetro ou que eles criam. Por outro lado, não é recomendável falar com os "amigos dos amigos".

Um exemplo — formulado por David Bock (link) — ilustra com clareza os benefícios do Princípio de Demeter. O exemplo baseia-se em um cenário com três "objetos": um entregador de jornais, um cliente e sua carteira. Uma violação do Princípio de Demeter ocorre se, para receber o valor de um jornal, o entregador tiver que executar o seguinte código:

```
preco = 6.00;
Carteira carteira = cliente.getCarteira();
if (carteira.getValorTotal() >= preco) { // viola Demeter
carteira.debita(preco); // viola Demeter
} else {
// volto amanhã, para cobrar o valor do jornal
}
```

Veja que o jornaleiro têm acesso à carteira do seu cliente — via método `getCarteira()` — e então ele mesmo retira o valor do jornal dela. Provavelmente, nenhum cliente aceitaria que um jornaleiro tivesse tamanha liberdade ... Portanto, uma solução mais realista é a seguinte:

```
preco = 6.00;
try {
cliente.pagar(preco);
}
catch (ExcecaoValorInsuficiente e) {
// volto amanhã, para cobrar o valor do jornal
}
```

No novo código, o cliente não libera o acesso à sua carteira. Pelo contrário, o jornaleiro nem fica ciente de que o cliente possui uma carteira. Essa informação está encapsulada na classe `Cliente`. Em vez disso, o cliente oferece um método pagar, que deve ser chamado pelo jornaleiro. Finalmente, uma exceção sinaliza quando o `Cliente` não possui recursos suficientes para pagar pelo jornal.

1.6.6 Princípio Aberto/Fechado

Esse princípio, originalmente proposto por Bertrand Meyer ainda na década de 80 (link), defende algo que pode parecer paradoxal: uma classe deve estar fechada para modificações e aberta para extensões.

No entanto, o aparente paradoxo se esclarece quando o projeto da classe prevê a possibilidade de extensões e customizações. Para isso, o projetista pode se valer de recursos como herança, funções de mais alta ordem (ou funções lambda) e padrões de projeto, como Abstract Factory, Template Method e Strategy. Especificamente,

no próximo capítulo, iremos tratar de padrões de projeto que permitem customizar e estender uma classe sem modificar o seu código.

Em resumo, o Princípio Aberto/Fechado tem como objetivo a construção de classes flexíveis e extensíveis, capazes de se adaptarem a diversos cenários de uso, sem modificações no seu código fonte.

Exemplo 1: Um exemplo de classe que segue o Princípio Aberto/Fechado é a classe `Collections` de Java. Ela possui um método estático para ordenar uma lista em ordem crescente de seus elementos. Um exemplo de uso desse método é mostrado a seguir:

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

No entanto, futuramente, podemos precisar de usar o método `sort` para ordenar as strings de acordo com seu tamanho em caracteres. Felizmente, a classe `Collections` está preparada para esse novo cenário de uso. Mas para isso precisamos implementar um objeto `Comparator`, que irá comparar as strings pelo seu tamanho, como no seguinte código:

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

Ou seja, a classe `Collections` se mostrou "aberta" a lidar com esse novo requisito, mas mantendo o seu código "fechado", isto é, o código fonte da classe não teve que ser modificado.

Exemplo 2: Mostramos agora um exemplo de função que não segue o Princípio Aberto/Fechado.

```
double calcTotalBolsas(Aluno[] lista) {
double total = 0.0;
foreach (Aluno aluno in lista) {
if (aluno instanceof AlunoGrad) {
AlunoGrad grad = (AlunoGrad) aluno;
total += "código que calcula bolsa de grad";
}
else if (aluno instanceof AlunoMestrado) {
AlunoMestrado mestrando = (AlunoMestrado) aluno;
total += "código que calcula bolsa de mestrando";
}
}
return total;
}
```

Se amanhã tivermos que criar mais uma subclasse de `Aluno`, por exemplo, `AlunoDoutorado`, o código de `calcTotalBolsas` terá que ser adaptado. Ou seja, a função não está preparada para acomodar extensões (isto é, ela não está aberta), nem imune a alterações no seu código (isto é, ela também não está fechada).

O Princípio Aberto/Fechado requer que o projetista de uma classe antecipe os seus pontos de extensão. Por isso, não é possível a uma classe acomodar todos os possíveis tipos de extensões que podem aparecer. Mas apenas aqueles para os quais são oferecidos pontos de extensão, seja via herança, funções de mais alta ordem ou padrões de projeto. Por exemplo, a implementação da classe `Collections` (no exemplo 1) usa um algoritmo de ordenação que é uma versão do MergeSort. Porém, os clientes da classe não podem alterar e customizar esse algoritmo, tendo que se contentar com a implementação default que é oferecida. Logo, sob o critério de customização do algoritmo de ordenação, o método sort não atende ao Princípio Aberto/Fechado.

1.6.7 Princípio de Substituição de Liskov

Conforme já discutimos ao falar do princípio "Prefira Composição a Herança", herança não é mais um conceito popular como foi na década de 80. Hoje, o emprego de herança é mais restrito e raro. No entanto, alguns casos de uso ainda são justificados. Herança define uma relação "é-um" entre objetos de uma classe base e objetos de subclasses. A vantagem é que comportamentos (isto é, métodos)

comuns a essas classes podem ser implementados uma única vez, na classe base. Feito isso, eles são herdados em todas as subclasses.

O Princípio de Substituição de Liskov explicita regras para redefinição de métodos de classes base em classes filhas. O nome do princípio é uma referência a Barbara Liskov, professora do MIT e ganhadora da edição de 2008 do Prêmio Turing. Dentre outros trabalhos, Liskov desenvolveu pesquisas sobre sistemas de tipos para linguagens orientadas a objetos. Foi em um desses trabalhos que ela enunciou o princípio que depois ganhou seu nome.

Para explicar o Princípio de Substituição de Liskov vamos nos basear no seguinte exemplo:

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

O método `f` pode ser chamado passando-se como parâmetros objetos de subclasses `B1`, `B2`, ..., `Bn` da classe base `A`, como mostrado a seguir:

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

O Princípio de Substituição de Liskov determina as condições — semânticas e não sintáticas — que as subclasses devem atender para que um programa como o anterior funcione.

Suponha que as subclasses `B1`, `B2`, ..., `Bn` redefinam o método `g()` de `A`, que é um método chamado no corpo de `f`. O Princípio de Substituição de Liskov prescreve que essas redefinições não podem violar o contrato da implementação original de `g` em A.

Exemplo 1: Suponha uma classe base que calcula números primos. Suponha ainda algumas subclasses que implementam outros algoritmos com o mesmo propósito. Especificamente, o método `getPrimo(n)` é um método que retorna o

n-ésimo número primo. Esse método existe na classe base e é redefinido em todas as subclasses.

Suponha ainda que o contrato do método `getPrimo(n)` especifique o seguinte: 1 <= n <= 1 milhão. Ou seja, o método deve ser capaz de retornar qualquer número primo, para n variando de 1 até 1 milhão. Nesse exemplo, uma violação do contrato de `getPrimo(n)` ocorre, por exemplo, se, em uma das classes, o algoritmo implementado calcule apenas números primos até 900 mil.

De forma mais concreta, o Princípio de Substituição de Liskov define o seguinte: suponha que um cliente chame um método `getPrimo(n)` de um objeto `p` da classe `NumeroPrimo`. Suponha agora que o objeto `p` seja "substituído" por um objeto de uma subclasse de `NumeroPrimo`. Nesse caso, o cliente vai passar a executar o método `getPrimo(n)` dessa subclasse. Porém, essa "substituição" de métodos não deve ter impacto no comportamento do cliente. Para tanto, todos os métodos `getPrimo(n)` das subclasses de `NumeroPrimo` devem realizar as mesmas tarefas que o método original, possivelmente de modo mais eficiente.

Exemplo 2: Vamos mostrar um segundo exemplo de violação, dessa vez bem forte, exatamente para reforçar o sentido do Princípio de Substituição de Liskov.

```
class A {
int soma(int a, int b) {
return a+b;
}
}
class B extends A {
int soma(int a, int b) {
String r = String.valueOf(a) + String.valueOf(b);
return Integer.parseInt(r);
}
}
class Cliente {
void f(A a) {
...
a.soma(1,2); // pode retornar 3 ou 12
...
}
}
class Main {
void main() {
A a = new A();
B b = new B();
Cliente cliente = new Cliente();
cliente.f(a);
cliente.f(b);
}
}
```

Nesse exemplo, o método que soma dois inteiros foi redefinido na subclasse com uma semântica de concatenação dos respectivos valores convertidos para strings. Logo, para um desenvolvedor encarregado de manter a classe `Cliente` a situação fica bastante confusa. Em uma execução, a chamada `soma(1,2)` retorna 3 (isto é, 1+2); na execução seguinte, a mesma chamada irá retornar 12 (isto é, "1"+ "2" = "12" ou 12, como inteiro).

1.7 Métricas

Ao longo dos anos, diversas métricas foram propostas para quantificar propriedades de um projeto de software. Normalmente, essas métricas precisam do código fonte

de um sistema, isto é, o projeto já deve ter sido implementado. Por meio da análise de características do código fonte, elas expressam de forma quantitativa — por meio de valores numéricos — propriedades como tamanho, coesão, acoplamento e complexidade do código. O objetivo é permitir a avaliação da qualidade de um projeto de forma mais objetiva.

No entanto, a monitoração do projeto de um sistema por meio de métricas de código fonte não é uma prática tão comum nos dias de hoje. Um dos motivos é que diversas propriedades de um sistema de software — como coesão e acoplamento, por exemplo — possuem um grau de subjetividade, o que dificulta a sua mensuração. Além disso, a interpretação dos resultados de métricas de software depende de informações de contexto. Uma determinada faixa de valores de uma métrica pode ser admissível em um sistema, mas não ser em outro sistema, de um domínio diferente. Mesmo entre as classes de um sistema, a interpretação dos valores de uma determinada métrica pode ser bem distinta.

Nesta seção, vamos estudar métricas para mensurar as seguintes propriedades de um projeto de software: tamanho, coesão, acoplamento e complexidade. Iremos detalhar os procedimentos de cálculo dessas métricas e dar alguns exemplos. Existem ainda ferramentas e plugins para IDEs que calculam essas métricas de forma automática.

1.7.1 Tamanho

A métrica de código fonte mais popular é **linhas de código** (LOC, *lines of code*). Ela pode ser usada para medir o tamanho de uma função, classe, pacote ou de um sistema inteiro. Quando se reporta os resultados de LOC, deve-se tomar o cuidado de deixar claro quais linhas foram de fato contadas. Por exemplo, se comentários ou linhas em branco foram considerados ou não.

Embora LOC possa dar uma ideia do tamanho de um sistema, ela não deve ser usada para medir a produtividade de programadores. Por exemplo, se um desenvolvedor implementou 1 KLOC em um mês e outro implementou 5 KLOC, não podemos afirmar que o segundo foi 5 vezes mais produtivo. Dentre outros motivos, os requisitos implementados por cada um deles podem ter complexidade diferente. Ken Thompson — um dos desenvolvedores do sistema operacional Unix — tem uma frase a esse respeito:

> "Um dos dias mais produtivos da minha vida foi quando eu deletei 1.000 linhas de código de um sistema."

Essa frase é atribuída a Thompson no seguinte livro, de Eric Raymond, página 24. Portanto, métricas de software, quaisquer que sejam, não devem ser vistas como uma meta. No caso de LOC, isso poderia, por exemplo, incentivar os desenvolvedores a gerar código duplicado apenas para cumprir a meta estabelecida.

Outras metas de tamanho de um sistema incluem: número de métodos, número de atributos, número de classes e número de pacotes.

1.7.2 Coesão

Uma das métricas mais conhecidas para se calcular coesão é chamada de **LCOM** (*Lack of Cohesion Between Methods*). Na verdade, como seu nome indica, LCOM mede a "falta de coesão" de uma classe. Em geral, métricas de software são interpretadas da seguinte forma: quanto maior o valor da métrica, pior a qualidade do código ou do projeto. No entanto, coesão é uma exceção a essa regra, pois quanto maior a coesão de uma classe, melhor o ser seu projeto. Por isso, LCOM foi planejada para medir a falta de coesão de classes. Quanto maior o valor de LCOM, maior a falta de coesão de uma classe e, portanto, pior o seu projeto.

Para calcular o valor de LCOM de uma classe C deve-se, primeiro, criar o seguinte conjunto:

M(C) = { (f_1, f_2) | f_1 e f_2 são métodos de C }

Ele é formado por todos os pares não-ordenados de métodos da classe C. Seja ainda o seguinte conjunto:

A(f) = conjunto de atributos da classe que são acessados por um método f

O valor de LCOM de C é assim definido:

P = | { (f_1, f_2) in M(C) | A(f_1) inter A(f_2) = empty } |

Isto é, LCOM(C) é o número de pares de métodos — dentre todos os possíveis pares de métodos de C — que não usam atributos em comum.

Exemplo: Para deixar a explicação mais clara, suponha a seguinte classe:

```
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);
}
}
```

A próxima tabela mostra os elementos dos conjuntos M e A; e o resultado da interseção que define o valor de LCOM.

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

Logo, nesse exemplo, LCOM(C) = 1, pois a classe C tem três possíveis pares de métodos, mas dois deles acessam pelo menos um atributo em comum (veja terceira coluna da tabela). Resta um único par de métodos que não tem atributos em comum.

Portanto, LCOM parte do pressuposto que, em uma classe coesa, qualquer par de métodos deve acessar pelo menos um atributo em comum. Ou seja, o que dá coesão a uma classe é o fato de seus métodos trabalharem com os mesmos

atributos. Por isso, a coesão de uma classe é prejudicada — isto é, seu LCOM aumenta em uma unidade — sempre que achamos um par de métodos (f_1,f_2), onde f_1 manipula alguns atributos e f_2 manipula atributos diferentes.

Para cálculo de LCOM normalmente não são considerados métodos construtores e getters/setters. Construtores tendem a ter atributos em comum com a maioria dos outros métodos. E o contrário tende a acontecer com getters e setters.

Por fim, é importante mencionar que existem propostas alternativas para o cálculo de LCOM. A versão que apresentamos é chamada de LCOM1 e foi proposta por Shyam Chidamber e Chris Kemerer, em 1991 (link). As versões alternativas ganham os nomes de LCOM2, LCOM3, etc. Por isso, ao reportar valores de LCOM, é importante deixar claro qual versão da métrica está sendo adotada.

1.7.3 Acoplamento

CBO (*Coupling Between Objects*) é uma métrica para medir **acoplamento estrutural** entre duas classes. Ela também foi proposta por Chidamber e Kemerer (link1 e link2).

Dada uma classe A, CBO conta o número de classes das quais A depende de forma sintática (ou estrutural). Diz-se 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 um tipo de retorno do tipo B
- A captura uma exceção do tipo B
- A levanta uma exceção do tipo B
- A cria um objeto do tipo B.

Seja uma classe A com dois métodos (`metodo1` e `metodo2`):

```
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) { ... }
}
}
```

Conforme indicamos numerando os tipos dos quais A depende, temos que CBO(A) = 9.

A definição de CBO não distingue as classes das quais uma classe depende. Por exemplo, tanto faz se a dependência é para uma classe da biblioteca de Java (por exemplo, String) ou uma classe mais instável da própria aplicação que está sendo desenvolvida.

1.7.4 Complexidade

Complexidade Ciclomática (CC) é uma métrica proposta por Thomas McCabe em 1976 para medir a complexidade do código de uma função ou método (link). Às vezes, ela é chamada também de Complexidade de McCabe. No contexto dessa métrica, o conceito de complexidade relaciona-se com a dificuldade de manter e testar uma função. A definição de CC baseia-se no conceito de grafos de fluxo de controle. Em tais grafos, os nodos representam os comandos de uma função ou método; e as arestas representam os possíveis fluxos de controle. Portanto, comandos como if geram fluxos de controle alternativos. O nome da métrica deriva do fato de ser calculada usando um conceito de Teoria dos Grafos chamado de número ciclomático (*cyclomatic number*).

Porém, existe uma alternativa simples para calcular o CC de uma função, a qual dispensa a construção de grafos de fluxo de controle. Essa alternativa define CC da seguinte forma:

CC = "número de comandos de decisão em uma função" + 1

Onde comandos de decisão podem ser if, while, case, for, etc. A intuição subjacente a essa fórmula é que comandos de decisão tornam o código mais difícil de entender e testar e, portanto, mais complexo.

Portanto, o cálculo de CC é bastante simples: dado o código fonte de uma função, conte o número dos comandos listados acima e some 1. Consequentemente, o menor valor de CC é 1, que ocorre em um código que não possui nenhum comando de decisão. No artigo onde definiu a métrica, McCabe sugere que um "limite superior razoável, mas não mágico" para CC é 10.

Bibliografia

Robert C. Martin. Clean Architecture: A Craftsman's Guide to Software Structure and Design, Prentice Hall, 2017.

John Ousterhout. A Philosophy of Software Design, Yaknyam Press, 2018.

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

Frederick P. Brooks. O Mítico Homem-Mês. Ensaios Sobre Engenharia de Software. Alta Books, 1a edição, 2018.

Diomidis Spinellis. Code Quality. Addison-Wesley, 2006.

Andrew Hunt, David Thomas. The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley, 1999.

Mauricio Aniche. Orientação a Objetos e SOLID para Ninjas. Projetando classes flexíveis. Casa do Código, 2015.

Thomas J. McCabe. A Complexity Measure. IEEE Transactions on Software Engineering, 1976.

Shyam Chidamber and Chris Kemerer. A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 1994.

Shyam Chidamber and Chris Kemerer. Towards a metrics suite for object oriented design. Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA), 1991.

Exercícios de Fixação

1. Descreva três benefícios da propriedade de projeto chamada ocultamento de informação (*information hiding*)?

2. Suponha que um programador adote a seguinte estratégia: ao implementar qualquer nova funcionalidade ou corrigir um bug que implique na modificação de duas classes A e B localizadas em arquivos diferentes, ele conclui a tarefa movendo as classes para o mesmo arquivo. Explicando melhor: após terminar a tarefa de programação que ficou sob sua responsabilidade, ele escolhe uma das classes, digamos a classe B, e a move para o mesmo arquivo da classe A. Agindo dessa maneira, ele estará melhorando qual propriedade de projeto? Por outro lado, qual propriedade de projeto estará sendo afetada de modo negativo? Justifique.

3. **Classitis** é o nome dado por John Ousterhout à proliferação de pequenas classes em um sistema. Segundo ele, *classitis* pode resultar em classes que individualmente são simples, mas que aumentam a complexidade total de um sistema. Usando os conceitos de acoplamento e coesão, como podemos explicar o problema causado por essa "doença"?

4. Defina: (a) acoplamento aceitável; (b) acoplamento ruim; (c) acoplamento estrutural; (d) acoplamento evolutivo (ou lógico).

5. Dê um exemplo de: (1) acoplamento estrutural e aceitável; (2) acoplamento estrutural e ruim.

6. É possível que uma classe A esteja acoplada a uma classe B sem ter uma referência para B em seu código? Se sim, esse acoplamento será aceitável ou ruim?

7. Suponha um programa onde todo o código está implementado no método main. Ele tem um problema de coesão ou acoplamento? Justifique.

8. Qual princípio de projeto é violado pelo seguinte código?

```
void onclick() {
num1 = textfield1.value();
c1 = BD.getConta(num1)
num2 = textfield2.value();
c2 = BD.getConta(num2)
valor = textfield3.value();
beginTransaction();
try {
c1.retira(valor);
c2.deposita(valor);
commit();
}
catch() {
rollback();
}
}
```

9. Costuma-se afirmar que existem três conceitos chaves em orientação a objetos: encapsulamento, polimorfismo e herança. Suponha que você tenha sido encarregado de projetar uma nova linguagem de programação. Suponha ainda que você poderá escolher apenas dois dos três conceitos que mencionamos. Qual dos conceitos eliminaria então da sua nova linguagem? Justifique sua resposta.

10. Qual princípio de projeto é violado pelo seguinte código? Como você poderia alterar o código do método para atender a esse princípio?

```
void sendMail(ContaBancaria conta, String msg) {
Cliente cliente = conta.getCliente();
String mail = cliente.getMailAddress();
“Envia mail”
}
```

11. Qual princípio de projeto é violado pelo seguinte código? Como você poderia alterar o código do método para atender a esse princípio?

```
void imprimeDataContratacao(Funcionario func) {
Date data = func.getDataContratacao();
String msg = data.format();
System.out.println(msg);
}
```

12. As pré-condições de um método são expressões booleanas envolvendo seus parâmetros (e, possivelmente, o estado de sua classe) que devem ser verdadeiras antes da sua execução. De forma semelhante, as pós-condições são expressões booleanas envolvendo o resultado do método. Considerando essas definições, qual princípio de projeto é violado pelo código abaixo?

```
class A {
int f(int x) { // pre: x > 0
...
return exp;
} // pos: exp > 0
...
}
```

```
class B extends A {
int f(int x) { // pre: x > 10
...
return exp;
} // pos: exp > -50
...
}
```

13. Por que a métrica LCOM mede a ausência e não a presença de coesão? Justifique.

14. Qual das seguintes classes é mais coesa? Justifique computando os valores de LCOM de cada uma delas.

```
class A {
X x = new X();
void f() {
x.m1();
}
void g() {
x.m2();
}
void h() {
x.m3();
}
}
```

```
class B {
X x = new X();
Y y = new Y();
Z z = new Z();
void f() {
x.m();
}
void g() {
y.m();
}
void h() {
z.m();
}
}
```

15. Todos os métodos de uma classe devem ser considerados no cálculo de LCOM? Sim ou não? Justifique.

16. Calcule o CBO e LCOM da seguinte classe:

```
class A extends B {
C f1, f2, f3;
void m1(D p) {
“usa f1 e f2”
}
void m2(E p) {
“usa f2 e f3”
}
void m3(F p) {
“usa f3”
}
}
```

17. A definição de complexidade ciclomática é independente de linguagem de programação. Sim ou não? Justifique.

18. Dê um exemplo de código com complexidade ciclomática mínima. Qual é essa complexidade?

19. Cristina Lopes — professora da Universidade da Califórnia, em Irvine, nos EUA — é autora de um livro sobre estilos de programação (link). Ela discute no livro diversos estilos para implementação de um mesmo problema, chamado frequência de termos. Dado um arquivo texto, deve-se listar as n-palavras mais frequentes em ordem decrescente de frequência e ignorando *stop words*, isto é, artigos, preposições, etc. O código fonte em Python de todas as versões analisadas no livro está publicamente disponível no GitHub (e, para esse exercício, fizemos um fork do repositório original). Faça uma análise de duas dessas versões:

- Monolítica, disponível neste link.
- Orientada a objetos, disponível neste link.

Primeiro, revise e estude o código das duas versões (cada versão tem menos de 100 linhas). Em seguida, argumente sobre as vantagens da solução OO sobre a versão monolítica. Para isso, tente extrapolar o tamanho do sistema. Suponha que ele será implementado por desenvolvedores diferentes e que cada um ficará responsável por uma parte do projeto.