

# **Engenharia de Software Moderna**

**Vários Autores**

# Prefácio

*A inutilidade dos prefácios é um lugar comum da história dos prefácios, portanto serei breve.* – Eduardo Giannetti

A ideia de escrever este livro surgiu no início de 2019, quando fui alocado para ministrar a disciplina Engenharia de Software, do Bacharelado em Ciência da Computação, da UFMG. Para preparar o curso, comecei com uma análise dos principais livros de Engenharia de Software. Para minha surpresa, percebi que eles tinham mudado pouco desde que cursei a disciplina na minha graduação há mais de 25 anos!

Meu objetivo era escolher um livro que permitisse, no início de uma aula, dizer para os alunos: hoje vamos estudar tal assunto, que corresponde a tal capítulo do livro-texto. No final da aula, gostaria de sugerir aos alunos: para fixar a matéria que acabamos de ver, sugiro que façam tais exercícios. No entanto, infelizmente, não encontrei esse livro. Em vez disso, tive que fazer uma extensa pesquisa e leitura de pelo menos 15 livros.

Como resultado, preparei mais de 600 slides, que considero conter o principal material que deve ser tratado em uma disciplina de graduação em Engenharia de Software, especificamente em cursos que possuem uma única disciplina na área. Porém, estudar apenas por slides não proporciona a mesma experiência de aprendizado obtida com a leitura atenta de um texto completo e contextualizado.

Assim, surgiu a ideia de transformar os slides em um livro que pudesse desempenhar o papel do sonhado livro-texto, já visando a futuras ofertas da disciplina. E que fosse útil também para outros professores, que devem enfrentar problemas semelhantes ao meu quando têm que ministrar um curso de graduação em Engenharia de Software.

Gostaria, então, de destacar os seguintes pontos sobre o conteúdo, a organização e a estratégia usada na escrita do livro:

- Ele foi escrito para ser um livro moderno, com ênfase em técnicas e princípios que são largamente usados na construção de software nos dias de hoje.
- Por outro lado, o livro também cobre técnicas e princípios tradicionais, porém de forma rápida. O motivo é que achamos importante cobrir a história da área, mencionando o que deu certo e o que não deu. Achamos que essa visão é importante na formação e amadurecimento dos alunos.
- O livro inclui inúmeros exemplos e discussões de casos reais. Para isso, criei a seção Mundo Real, na qual os assuntos são ilustrados com exemplos recentes e reais, provenientes de grandes empresas de software e também de artigos científicos, buscando-se apresentar o que se faz de melhor tanto na indústria como na academia.
- O livro foi escrito em português, pois o objetivo é contribuir, primeiro, com os cursos e alunos brasileiros. Porém, optamos por não traduzir alguns termos — como refactoring, branches e sprint — pois achamos que eles são usados, também sem tradução, pelos desenvolvedores brasileiros no seu dia a dia.
- Apesar de tratar de temas modernos, a intenção foi escrever um livro duradouro. Por isso, temas e tecnologias que ainda não passaram pelo teste do tempo não são abordados (ou são abordados de forma rápida). Um exemplo são os últimos frameworks e arquiteturas para implementação de sistemas, que tendem a mudar rapidamente. Também não acoplamos o livro a nenhuma linguagem de programação. Por exemplo, os trechos de código são mostrados em uma sintaxe neutra.
- Ele não é um livro extenso, com diversos capítulos que, na prática, tratam de assuntos que não são importantes nos cursos atuais.
- Cada vez mais, engenheiros de software têm que escrever código. Hoje, há pouco espaço para dizer que eu não preciso programar, pois sou arquiteto ou analista. Por isso, nos capítulos de projeto, testes e refatoração procuramos seguir a recomendação de Linus Torvalds (criador do Linux): falar é fácil, mas mostre-me o código. Assim, esses

capítulos incluem dezenas de exemplos de código, que simulam problemas e soluções típicos de sistemas reais.

- Ao longo da escrita, procuramos praticar o que enfatizamos no livro, principalmente no que diz respeito à prevalência atual de métodos de desenvolvimento ágeis. Assim, o livro foi escrito seguindo princípios ágeis. Cada capítulo foi tratado como sendo um sprint; uma vez pronto, ele foi disponibilizado para uso, para receber críticas e sugestões. Como ocorre com software, acho arriscado, nos dias de hoje, passar anos escrevendo um manuscrito, trancado em minha sala, para só então torná-lo público.

## Público-Alvo

O livro destina-se a alunos de cursos de graduação. Ele foi escrito para ser adotado em cursos que possuem uma única disciplina de Engenharia de Software, com 60 horas. Porém, achamos também que ele pode ser usado — junto com outros livros — em cursos com duas ou mais disciplinas na área.

Além disso, ele pode ser adotado em cursos técnicos e tecnológicos. Na verdade, fizemos um esforço para usar uma linguagem clara, próxima à linguagem coloquial, exatamente para não criar barreiras à adoção do livro. Por fim, também escrevemos o livro pensando em profissionais da área, que estão em busca de aperfeiçoamento em temas e métodos modernos de Engenharia de Software.

## Pré-requisitos

Os leitores devem ter domínio de conceitos básicos de programação e de algoritmos e estruturas de dados. Além disso, recomendamos domínio de orientação a objetos. Supondo um curso de graduação de 4 anos, os alunos devem estar na metade final para melhor aproveitar o conteúdo do livro.

## Website

O livro possui uma versão aberta, em HTML, disponível em:

<https://engsoftmoderna.info>

Nesse site, estão também disponíveis diversos exercícios de múltipla escolha. Não menos importante: para reportar qualquer erro, mesmo que um simples erro ortográfico, você pode usar este [formulário](#).

Marco Túlio Valente Belo Horizonte, 10 de fevereiro de 2020.

# Cap 1 Introdução

Neste primeiro capítulo, vamos definir e contextualizar o que é Engenharia de Software (Seção 1.1) e dar uma visão geral dos principais assuntos estudados nessa área da Computação (Seção 1.2). O objetivo é propiciar ao leitor uma visão horizontal da área de Engenharia de Software, antes de nos aprofundarmos em temas específicos. Além disso, sendo Engenharia de Software uma área bastante ampla, vamos caracterizar os tipos de sistemas de software que podem se beneficiar das técnicas e conceitos apresentados neste livro (Seção 1.3). O objetivo é, logo no início, evitar falsas expectativas em relação ao conteúdo do trabalho. Por fim, iremos apresentar a estrutura e os assuntos tratados nos capítulos restantes do livro (Seção 1.4).

## Definições, Contexto e História

No mundo moderno, tudo é software. Hoje em dia, por exemplo, empresas de qualquer tamanho dependem dos mais diversos sistemas de informação para automatizar seus processos. Governos também interagem com os cidadãos por meio de sistemas computacionais, por exemplo, para coletar impostos ou realizar eleições. Empresas vendem, por meio de sistemas de comércio eletrônico, uma gama imensa de produtos, diretamente para os consumidores. Software está também embarcado em diferentes dispositivos e produtos de engenharia, incluindo automóveis, aviões, satélites, robôs, etc. Por fim, software está contribuindo para renovar indústrias e serviços tradicionais, como telecomunicações, transporte em grandes centros urbanos, hospedagem, lazer e publicidade.

Portanto, devido a sua relevância no nosso mundo, não é surpresa que exista uma área da Computação destinada a investigar os desafios e propor soluções que permitam desenvolver sistemas de software — principalmente aqueles mais complexos e de maior tamanho — de forma produtiva e com qualidade. Essa área é chamada de **Engenharia de Software**.

Engenharia de Software trata da aplicação de abordagens sistemáticas, disciplinadas e quantificáveis para desenvolver, operar, manter e evoluir software. Ou seja, Engenharia de Software é a área da Computação que se

preocupa em propor e aplicar princípios de engenharia na construção de software.

Historicamente, a área surgiu no final da década de 60 do século passado. Nas duas décadas anteriores, os primeiros computadores modernos foram projetados e começaram a ser usados principalmente para resolução de problemas científicos. Ou seja, nessa época software não era uma preocupação central, mas sim construir máquinas que pudessem executar alguns poucos programas. Em resumo, computadores eram usados por poucos e para resolver apenas problemas científicos.

No entanto, progressos contínuos nas tecnologias de construção de hardware mudaram de forma rápida esse cenário. No final da década de 60, computadores já eram mais populares, já estavam presentes em várias universidades norte-americanas e europeias e já chegavam também em algumas grandes empresas. Os cientistas da computação dessa época se viram diante de um novo desafio: como os computadores estavam se tornando mais populares, novas aplicações não apenas se tornavam possíveis, mas começavam a ser demandadas pelos usuários dos grandes computadores da época. Na verdade, os computadores eram grandes no sentido físico e não em poder de processamento, se comparado com os computadores atuais. Dentre essas novas aplicações, as principais eram sistemas comerciais, como folha de pagamento, controle de clientes, controle de estoques, etc.

**Conferência da OTAN:** Em outubro de 1968, um grupo de cerca de 50 renomados Cientistas da Computação se reuniu durante uma semana em Garmisch, na Alemanha, em uma conferência patrocinada por um comitê científico da OTAN, a organização militar que congrega os países do Atlântico Norte (veja uma foto da reunião na próxima figura). O objetivo da conferência era chamar a atenção para um problema crucial do uso de computadores, o chamado software. A conferência produziu um relatório, com mais de 130 páginas, que afirmava a necessidade de que software fosse construído com base em princípios práticos e teóricos, tal como ocorre em ramos tradicionais e bem estabelecidos da Engenharia. Para deixar essa proposta mais clara, decidiu-se cunhar o termo Engenharia de Software. Por isso, a Conferência da OTAN é considerada o marco histórico de criação da área de Engenharia de Software.



Cientistas na conferência da OTAN de 1968 sobre Engenharia de Software.  
Reprodução gentilmente autorizada pelo Prof. Robert McClure.

O comentário a seguir, de um dos participantes da Conferência da OTAN, ilustra os desafios que esperavam a recém-criada área de pesquisa:

O problema é que certas classes de sistemas estão colocando demandas sobre nós que estão além das nossas capacidades e das teorias e métodos de projeto que conhecemos no presente tempo. Em algumas aplicações não existe uma crise, como rotinas de ordenação e folhas de pagamento, por exemplo. Porém, estamos tendo dificuldades com grandes aplicações. Não podemos esperar que a produção de tais sistemas seja fácil.

Passado mais de meio século da Conferência da OTAN, os avanços obtidos em técnicas e métodos para construção de software são notáveis. Hoje, já se tem conhecimento de que software — na maioria das vezes — não deve ser construído em fases estritamente sequenciais, como ocorre com produtos tradicionais de engenharia, tais como Engenharia Civil, Engenharia Mecânica, Engenharia Eletrônica, etc. Já existem também padrões que podem ser usados por Engenheiros de Software em seus novos sistemas, de forma que eles não precisem reinventar a roda toda vez que enfrentarem um novo problema de projeto. Bibliotecas e frameworks para os mais diversos fins estão largamente disponíveis, de forma que desenvolvedores de software podem reusar código sem se preocupar com detalhes inerentes a tarefas como implementar interfaces gráficas, criar estruturas de dados, acessar bancos de dados, criptografar mensagens, etc. Prosseguindo, as mais variadas técnicas de testes podem (e devem) ser usadas para garantir que os sistemas em construção tenham qualidade e que falhas não ocorram quando eles entrarem em produção e forem usados por clientes reais. Sabe-se também que sistemas envelhecem, como outros produtos de engenharia. Logo, software também precisa de manutenção, não apenas corretiva, para corrigir bugs reportados por usuários, mas também para garantir que os sistemas continuem fáceis de manter e entender, mesmo com o passar dos anos.

## Não existe bala de prata

Como começamos a afirmar no parágrafo anterior, desenvolvimento de software é diferente de qualquer outro produto de Engenharia, principalmente quando se compara software com hardware. Frederick Brooks, Prêmio Turing em Computação (1999) e um dos pioneiros da área de Engenharia de Software, foi um dos primeiros a chamar a atenção para esse fato. Em 1987, em um ensaio intitulado *Não Existe Bala de Prata: Essência e Acidentes em Engenharia de Software* ([link](#)), ele discorreu sobre as particularidades da área de Engenharia de Software.

Segundo Brooks, existem dois tipos de dificuldades em desenvolvimento de software: **dificuldades essenciais** e **dificuldades accidentais**. As essenciais são da natureza da área e dificilmente serão superadas por qualquer nova tecnologia ou método que se invente.

Daí a menção à bala de prata no título do ensaio. Diz a lenda que uma bala de prata é a única maneira de matar um lobisomem, desde que usada em uma noite de lua cheia. Ou seja, por causa das dificuldades essenciais, não podemos esperar soluções milagrosas em Engenharia de Software, na forma de balas de prata. O interessante é que, mesmo conhecendo o ensaio de Brooks, sempre surgem novas tecnologias que são vendidas como se fossem balas de prata.

Segundo Brooks, as dificuldades essenciais são as seguintes:

1. **Complexidade:** dentre as construções que o homem se propõe a realizar, software é uma das mais desafiadoras e mais complexas que existe. Na verdade, como dissemos antes, mesmo construções de engenharia tradicional, como um satélite, uma usina nuclear ou um foguete, são cada vez mais dependentes de software.
2. **Conformidade:** pela sua natureza software tem que se adaptar ao seu ambiente, que muda a todo momento no mundo moderno. Por exemplo, se as leis para recolhimento de impostos mudam, normalmente espera-se que os sistemas sejam rapidamente adaptados à nova legislação. Brooks comenta que isso não ocorre, por exemplo, na Física, pois as leis da natureza não mudam de acordo com os caprichos dos homens.
3. **Facilidade de mudanças:** que consiste na necessidade de evoluir sempre, incorporando novas funcionalidades. Na verdade, quanto mais bem sucedido for um sistema de software, mais demanda por mudanças ele recebe.
4. **Invisibilidade:** devido à sua natureza abstrata, é difícil visualizar o tamanho e consequentemente estimar o esforço de construir um sistema de software.

As dificuldades (2), (3) e (4) são específicas de sistemas de software, isto é, elas não ocorrem em outros produtos de Engenharia, pelo menos na mesma intensidade. Por exemplo, quando a legislação ambiental muda, os fabricantes de automóveis têm anos para se conformar às novas leis. Adicionalmente, carros não são alterados, pelo menos de forma essencial,

com novas funcionalidades, após serem vendidos. Por fim, um carro é um produto físico, com peso, altura, largura, assentos, forma geométrica, etc., o que facilita sua avaliação e precificação por consumidores finais.

Ainda segundo Brooks, desenvolvimento de software enfrenta também dificuldades acidentais. No entanto, elas estão associadas a problemas tecnológicos, que os Engenheiros de Software podem resolver, se devidamente treinados e caso tenham acesso às devidas tecnologias e recursos. Como exemplo, podemos citar as seguintes dificuldades: um compilador que produz mensagens de erro obscuras, uma IDE que possui muitos bugs e frequentemente sofre travamentos, um framework que não possui documentação, uma aplicação Web com uma interface pouco intuitiva, etc. Todas essas dificuldades dizem respeito à solução adotada e, portanto, não são uma característica inerente dos sistemas mencionados.

**Mundo Real:** Para ilustrar a complexidade envolvida na construção de sistemas de software reais, vamos dar alguns números sobre o tamanho desses sistemas, em linhas de código. Por exemplo, o sistema operacional Linux, em sua versão 4.1.3, de 2017, possui cerca de 25 milhões de linhas de código e contribuições de quase 1.700 engenheiros ([link](#)). Para mencionar um segundo exemplo, os sistemas do Google somavam 2 bilhões de linhas de código, distribuídas por 9 milhões de arquivos, em janeiro de 2015 ([link](#)). Nesta época, cerca de 40 mil solicitações de mudanças de código (commits) eram realizadas, em média, por dia, pelos cerca de 25 mil Engenheiros de Software empregados pelo Google nessa época.

## O que se Estuda em Engenharia de Software?

Para responder a essa pergunta, vamos nos basear no *Guide to the Software Engineering Body of Knowledge*, também conhecido pela sigla SWEBOK ([link](#)). Trata-se de um documento, organizado pela IEEE Computer Society (uma sociedade científica internacional), com o apoio de diversos pesquisadores e de profissionais da indústria. O objetivo do SWEBOK é precisamente documentar o corpo de conhecimento que caracteriza a área que hoje chamamos de Engenharia de Software.

O SWEBOK define 12 áreas de conhecimento em Engenharia de Software:

1. Engenharia de Requisitos
2. Projeto de Software
3. Construção de Software
4. Testes de Software
5. Manutenção de Software
6. Gerência de Configuração
7. Gerência de Projetos
8. Processos de Software
9. Modelos de Software
10. Qualidade de Software
11. Prática Profissional
12. Aspectos Econômicos

Na verdade, o SWEBOK inclui mais três áreas de conhecimento: Fundamentos de Computação, Fundamentos de Matemática e Fundamentos de Engenharia. No entanto, sendo áreas de fronteira, elas não serão tratadas neste capítulo.

No restante desta seção, vamos brevemente discutir e comentar sobre cada uma das 12 áreas listadas acima. O nosso objetivo é propiciar ao leitor um panorama do conhecimento que se adquiriu ao longo dos anos em Engenharia de Software e, assim, informá-lo sobre *o que* se estuda nessa área.

## **Engenharia de Requisitos**

Os requisitos de um sistema definem *o que* ele deve fazer e *como* ele deve operar. Assim, a Engenharia de Requisitos inclui o conjunto de atividades realizadas com o objetivo de definir, analisar, documentar e validar os requisitos de um sistema. Em uma primeira classificação, os requisitos podem ser **funcionais** ou **não-funcionais**.

Requisitos funcionais definem *o que* um sistema deve fazer; isto é, quais funcionalidades ou serviços ele deve implementar.

Já os requisitos não-funcionais definem *como* um sistema deve operar, sob quais restrições e com qual qualidade de serviço. São exemplos de requisitos não-funcionais: desempenho, disponibilidade, tolerância a falhas, segurança, privacidade, interoperabilidade, capacidade, manutenibilidade e usabilidade.

Por exemplo, suponha um sistema de *home-banking*. Nesse caso, os requisitos funcionais incluem informar o saldo da conta, informar o extrato, realizar transferência entre contas, pagar um boleto bancário, cancelar um cartão de débito, etc. Já os requisitos não-funcionais, dentre outros, incluem:

- Desempenho: informar o saldo da conta em menos de 3 segundos;
- Disponibilidade: estar no ar 99% do tempo;
- Tolerância a falhas: continuar operando mesmo se um determinado centro de dados cair;
- Segurança: criptografar todos os dados trocados com as agências;
- Privacidade: não disponibilizar para terceiros dados de clientes;
- Interoperabilidade: integrar-se com os sistemas do Banco Central;
- Capacidade: ser capaz de armazenar dados de 1 milhão de clientes;
- Usabilidade: ter uma versão para deficientes visuais.

## Projeto de Software

Durante o projeto de um sistema de software, são definidas suas principais unidades de código, porém apenas no nível de interfaces, incluindo **interfaces providas** e **interfaces requeridas**. Interfaces providas são aqueles serviços que uma unidade de código torna público para uso pelo resto do sistema. Interfaces requeridas são aquelas interfaces das quais uma unidade de código depende para funcionar.

Portanto, durante o projeto de um sistema de software, não entramos em detalhes de implementação de cada unidade de código, tais como detalhes de implementação dos métodos de uma classe, caso o sistema seja implementado em uma linguagem orientada a objetos.

Por exemplo, durante o projeto de um sistema de *home-banking*, pode-se propor uma classe para representar contas bancárias, como a seguinte:

```
class ContaBancaria {  
    private Cliente cliente;  
    private double saldo;  
    public double getSaldo() { ... }  
    public String getNomeCliente() { ... }  
    public String getExtrato (Date inicio) { ... }  
    ...  
}
```

Primeiro, é importante mencionar que a implementação acima é bem simples, pois o nosso objetivo é didático, isto é, diferenciar projeto de software de sua implementação. Para atingir esse objetivo, o importante é mencionar que `ContaBancaria` oferece uma interface para as demais classes do sistema, na forma de três métodos públicos, que constituem a interface provida pela classe. Por outro lado, `ContaBancaria` também depende de uma outra classe, `Cliente`; logo, a interface de `Cliente` é uma interface requerida por `ContaBancaria`. Muitas vezes, interfaces requeridas são chamadas de dependências. Isto é, `ContaBancaria` possui uma dependência para `Cliente`.

Quando o projeto é realizado em um nível mais alto e as unidades de código possuem maior granularidade — são pacotes, por exemplo — ele é classificado como um projeto arquitetural. Ou seja, **arquitetura de software**

trata da organização de um sistema em um nível de abstração mais alto do que aquele que envolve classes ou construções semelhantes.

## Construção de Software

Construção trata da implementação, isto é, codificação do sistema. Nesse momento, existem diversas decisões que precisam ser tomadas, como, por exemplo: definir os algoritmos e estruturas de dados que serão usados, definir os frameworks e bibliotecas de terceiros que serão usados; definir técnicas para tratamento de exceções; definir padrões de nomes, leiaute e documentação de código e, por último, mas não menos importante, definir as ferramentas que serão usadas no desenvolvimento, incluindo compiladores, ambientes integrados de desenvolvimento (IDEs), depuradores, gerenciadores de bancos de dados, ferramentas para construção de interfaces, etc.

## Testes de Software

Teste consiste na execução de um programa com um conjunto finito de casos, com o objetivo de verificar se ele possui o comportamento esperado. A seguinte frase, bastante famosa, de Edsger W. Dijkstra — também prêmio Turing em Computação (1982) — sintetiza não apenas os benefícios de testes, mas também suas limitações:

Testes de software mostram a presença de bugs, mas não a sua ausência.

Pelo menos três pontos podem ser comentados sobre testes, ainda neste capítulo de Introdução.

Primeiro, existem diversos tipos de testes. Por exemplo, **testes de unidade** (quando se testa uma pequena unidade do código, como uma classe), **testes de integração** (quando se testa uma unidade de maior granularidade, como um conjunto de classes), **testes de performance** (quando se submete o sistema a uma carga de processamento para verificar seu desempenho), **testes de usabilidade** (quando o objetivo é verificar a usabilidade da interface do sistema), etc.

Segundo, testes podem ser usados tanto para verificação como para validação de sistemas. Verificação tem como o objetivo garantir que um sistema atende à sua especificação. Já com validação, o objetivo é garantir que um sistema atende às necessidades de seus clientes. A diferença entre os conceitos só faz sentido porque pode ocorrer de a especificação de um sistema não expressar as necessidades de seus clientes. Por exemplo, essa diferença pode ser causada por um erro na fase de levantamento de requisitos; isto é, os desenvolvedores não entenderam os requisitos do sistema ou o cliente não foi capaz de explicá-los precisamente.

Existem duas frases, muito usadas, que resumem as diferenças entre verificação e validação:

- **Verificação:** estamos implementando o sistema corretamente? Isto é, de acordo com seus requisitos.
- **Validação:** estamos implementando o sistema correto? Isto é, aquele que os clientes ou o mercado está querendo.

Assim, quando se realiza um teste de um método, para verificar se ele retorna o resultado especificado, estamos realizando uma atividade de verificação. Por outro lado, quando realizamos um teste funcional e de aceitação, ao lado do cliente, isto é, mostrando para ele os resultados e funcionalidades do sistema, estamos realizando uma atividade de validação.

Terceiro, é importante definir e distinguir três conceitos relacionados a testes: **defeitos**, **bugs** e **falhas**. Para ilustrar a diferença entre eles, suponha o seguinte código para calcular a área de um círculo, dependendo de uma determinada condição:

```
if (condicao)
    area = pi * raio * raio * raio;
```

Esse código possui um defeito, pois a área de um círculo é  $\pi$  vezes raio ao quadrado, e não ao cubo. Bug é um termo mais informal, usado com objetivos às vezes diversos. Mas o uso mais comum é como sinônimo de defeito. Por fim, uma falha ocorre quando um código com defeito for executado — por exemplo, a condição do `if` do programa acima for

verdadeira — e, com isso, levar o programa a apresentar um resultado incorreto. Portanto, nem todo defeito ou bug ocasiona falhas, pois pode ser que o código defeituoso nunca seja executado.

Resumindo: código defeituoso é aquele que não está de acordo com a sua especificação. Se esse código for executado e de fato levar o programa a apresentar um resultado incorreto, dizemos que ocorreu uma falha.

**Aprofundamento:** Na literatura sobre testes, às vezes são mencionados os termos **erro** e **falta (fault)**. Quando isso ocorre, o significado é o mesmo daquele que adotamos para *defeito* neste livro. Por exemplo, o *IEEE Standard Glossary of Software Engineering Terminology* ([link](#)) define que falta é um passo, processo ou definição de dados incorretos em um programa de computador; os termos erro e bug são [também] usados para expressar esse significado. Resumindo, *defeito*, *erro*, *falta* e *bug* são sinônimos.

**Mundo Real:** Existe uma lista enorme de falhas de software, com consequências graves, tanto em termos financeiros como de vidas humanas. Um dos exemplos mais famosos é a explosão do foguete francês Ariane 5, lançado em 1996, de Kourou, na Guiana Francesa. Cerca de 30 segundos após o lançamento, o foguete explodiu devido a um comportamento inesperado de um dos sistemas de bordo, causando um prejuízo de cerca de meio bilhão de dólares. Interessante, o defeito que causou a falha no sistema de bordo do Ariane 5 foi bem específico, relativamente simples e restrito a poucas linhas de código, implementadas na linguagem de programação ADA, até hoje muito usada no desenvolvimento de software militar e espacial. Essas linhas eram responsáveis pela conversão de um número real, em ponto flutuante, com 64 bits, para um número inteiro, com 16 bits. Durante os testes e, provavelmente, lançamentos anteriores do foguete, essa conversão sempre foi bem sucedida: o número real sempre cabia em um inteiro. Porém, na data da explosão, alguma situação nunca testada previamente exigiu a conversão de um número maior do que o maior inteiro que pode ser representado em 16 bits. Com isso, gerou-se um resultado espúrio, que fez com que o sistema de controle do foguete funcionasse de forma errática, causando a explosão.

## Manutenção e Evolução de Software

Assim como sistemas tradicionais de Engenharia, software também precisa de manutenção. Neste livro, vamos usar a seguinte classificação para os tipos de manutenção que podem ser realizadas em sistemas de software: **corretiva, preventiva, adaptativa, refactoring e evolutiva**.

Manutenção corretiva tem como objetivo corrigir bugs reportados por usuários ou outros desenvolvedores.

Por sua vez, manutenção preventiva tem como objetivo corrigir bugs latentes no código, que ainda não causaram falhas junto aos usuários do sistema.

**Mundo Real:** Um exemplo de manutenção preventiva foram as atividades de manutenção realizadas por diversas empresas antes da virada do último milênio, de 1999 para 2000. Nessa época, diversos sistemas armazenavam o ano de uma data com dois dígitos, isto é, as datas tinham o formato DD-MM-AA. As empresas ficaram receosas de que, em 2000 e nos anos seguintes, algumas operações envolvendo datas retornassem valores incorretos, pois uma subtração 00 - 99, por exemplo, poderia dar um resultado inesperado. As empresas montaram então grupos de trabalho para realizar manutenções em seus sistemas e converter todas as datas para o formato DD-MM-AAAA. Como essas atividades foram realizadas antes da virada do milênio, elas são um exemplo de manutenção preventiva.

Manutenção adaptativa tem como objetivo adaptar um sistema a uma mudança em seu ambiente, incluindo tecnologia, legislação, regras de integração com outros sistemas ou demandas de novos clientes. Como exemplos de manutenção adaptativa podemos citar:

- A migração de um sistema de Python 2.7 para Python 3.0.
- A customização de um sistema para atender a requisitos de um novo cliente — isto é, quando se instala um sistema em um cliente é comum ter que realizar algumas alterações, para atender a particularidades de seu negócio.
- A adaptação de um sistema para atender a uma mudança de legislação ou outra mudança contextual.

Refactorings são modificações realizadas em um software preservando seu comportamento e visando exclusivamente a melhoria de seu código ou projeto. São exemplos de refactorings operações como renomeação de um método ou variável (para um nome mais intuitivo e fácil de lembrar), divisão de um método longo em dois métodos menores (para facilitar o entendimento) ou movimentação de um método para uma classe mais apropriada.

Manutenção evolutiva é aquela realizada para incluir uma nova funcionalidade ou introduzir aperfeiçoamentos importantes em funcionalidades existentes. Sistemas de software podem ser usados por décadas exatamente porque eles sofrem manutenções evolutivas, que preservam o seu valor para os clientes. Por exemplo, diversos sistemas bancários usados hoje em dia foram criados nas décadas de 70 e 80, em linguagens como COBOL. No entanto, eles já sofreram diversas evoluções e melhorias. Hoje, esses sistemas possuem interfaces Web e para celulares, que se integram aos módulos principais, implementados há dezenas de anos.

**Sistemas legados** são sistemas antigos, baseados em linguagens, sistemas operacionais e bancos de dados tecnologicamente ultrapassados. Por esse motivo, a manutenção desses sistemas costuma ser mais custosa e arriscada. Porém, é importante ressaltar que legado não significa irrelevante, pois muitas vezes esses sistemas realizam operações críticas para seus clientes.

**Aprofundamento:** Na literatura, existem classificações alternativas para os tipos de manutenção de software. Uma delas, proposta por Lientz & Swanson, em 1978 ([link](#)), classifica manutenção nas seguintes categorias: (1) Corretiva, exatamente como usado e definido neste livro; (2) Perfectiva, refere-se à adição de novas funcionalidades; neste livro, optamos por chamá-la de manutenção evolutiva; (3) Adaptativa, refere-se a mudanças no ambiente operacional do software, como um novo hardware ou sistema operacional; logo, não inclui, por exemplo, customizações para novos clientes, como proposto neste livro; (4) Preventiva, refere-se a mudanças que visam incrementar a manutenibilidade de um sistema; neste livro, optamos pelo termo mais comum hoje em dia, que é refactoring, e que iremos estudar no Capítulo 9.

## Gerência de Configuração

Atualmente, é inconcebível desenvolver um software sem um sistema de controle de versões, como git. Esses sistemas armazenam todas as versões de um software, não só do código fonte, mas também de documentação, manuais, páginas web, relatórios, etc. Eles também permitem restaurar uma determinada versão. Por exemplo, se foi realizada uma mudança no código que introduziu um bug crítico, pode-se com relativa facilidade recuperar e retornar para a versão antiga, anterior à introdução do bug.

No entanto, gerência de configuração é mais do que apenas usar um sistema como git. Ela inclui a definição de um conjunto de políticas para gerenciar as diversas versões de um sistema. Por exemplo, preocupa-se com o esquema usado para identificar as releases de um software; isto é, as versões de um sistema que serão liberadas para seus clientes finais. Um time de desenvolvedores pode definir que as releases de uma determinada biblioteca que eles estão desenvolvendo serão identificadas no formato  $x.y.z$ , onde  $x$ ,  $y$  e  $z$  são inteiros. Um incremento em  $z$  ocorre quando se lança uma nova release com apenas correções de bugs (normalmente, chamada de *patch*); um incremento em  $y$  ocorre quando se lança uma release da biblioteca com pequenas funcionalidades (normalmente, chamada de versão *minor*); por fim, um incremento em  $x$  ocorre quando se lança uma release com funcionalidades muito diferentes daquelas da última release (normalmente, chamada de versão *major*). Esse esquema de numeração de releases é conhecido como **versionamento semântico**.

## Gerência de Projetos

Desenvolvimento de software requer o uso de práticas e atividades de gerência de projetos, por exemplo, para negociação de contratos com clientes (com definição de prazos, valores, cronogramas, etc.), gerência de recursos humanos (incluindo contratação, treinamento, políticas de promoção, remuneração, etc.), gerência de riscos, acompanhamento da concorrência, marketing, finanças, etc. Em um projeto, normalmente usa-se o termo **stakeholder** para designar todas as partes interessadas no mesmo; ou seja, os stakeholders são aqueles que afetam ou que são afetados pelo projeto, podendo ser pessoas físicas ou organizações. Por exemplo, stakeholders comuns em projetos de software incluem, obviamente, seus desenvolvedores e seus clientes; mas, também, gerentes da equipe de

desenvolvimento, empresas subcontratadas, fornecedores de qualquer natureza, talvez algum nível de governo, etc.

Existe uma frase muito conhecida, também de Frederick Brooks, que captura uma peculiaridade de projetos de software. Segundo Brooks:

A inclusão de novos desenvolvedores em um projeto que está atrasado contribui para torná-lo ainda mais atrasado.

Essa frase ficou tão famosa que ela é hoje conhecida como **Lei de Brooks**. Basicamente, esse efeito acontece porque os novos desenvolvedores terão primeiro que entender e compreender todo o sistema, sua arquitetura e seu projeto (*design*), antes de começarem a produzir código útil. Além disso, equipes maiores exigem um maior esforço de comunicação e coordenação para tomar e explicar decisões. Por exemplo, se um time tem 3 desenvolvedores (d1, d2, d3), existem 3 canais de comunicação possíveis (d1-d2, d1-d3 e d2-d3); se ele cresce para 4 desenvolvedores, o número de canais duplica, para 6 canais. Se ele cresce para 10 desenvolvedores, passam a existir 45 canais de comunicação. Por isso, modernamente, software é desenvolvido em times pequenos, com uma dezena de engenheiros, se tanto.

**Tradução:** Em português, a palavra *projeto* pode se referir tanto a *design* como a *project*. Por exemplo, em uma subseção anterior introduzimos questões de projeto de software, isto é, *software design*, tratando de conceitos como interfaces, dependências, arquitetura, etc. Na presente seção, acabamos de discutir questões de gerência de projetos de software, isto é, *software project management*, tais como prazos, contratos, Lei de Brooks, etc. No restante deste livro, iremos traduzir apenas o uso mais comum em cada capítulo e manter o uso menos comum em inglês. Por exemplo, no Capítulo 2 (Processos de Desenvolvimento), usaremos projeto com tradução de *project*, pois é o uso mais comum neste capítulo. Já no Capítulo 5 (Princípios de Projeto) e no Capítulo 6 (Padrões de Projeto), *design* será traduzido para projeto, pois é o uso mais comum nesses capítulos, aparecendo inclusive no título dos mesmos.

**Aprofundamento:** A Lei de Brooks foi proposta em um livro clássico do autor sobre gerenciamento de projetos de software, chamado *The Mythical Man-Month*, cuja primeira edição foi publicada em 1975 ([link](#)). Nesse livro,

Brooks reporta as lições que aprendeu no início da sua carreira como gerente responsável pelos primeiros sistemas operacionais da IBM. Em 1995, uma segunda edição do livro foi lançada, em comemoração aos seus 20 anos. Essa edição incluiu um novo capítulo, com o artigo *No Silver Bullet — Essence and Accidents of Software Engineering*, publicado originalmente em 1987 (e que já comentamos nesta Introdução). Em 1999, Frederick Brooks ganhou o Prêmio Turing, considerado o Prêmio Nobel da Computação.

## Processos de Desenvolvimento de Software

Um processo de desenvolvimento define quais atividades e etapas devem ser seguidas para construir e entregar um sistema de software. Uma analogia pode ser feita, por exemplo, com a construção de prédios, que ocorre de acordo com algumas etapas: fundação, alvenaria, cobertura, instalações hidráulicas, instalações elétricas, acabamento, pintura, etc.

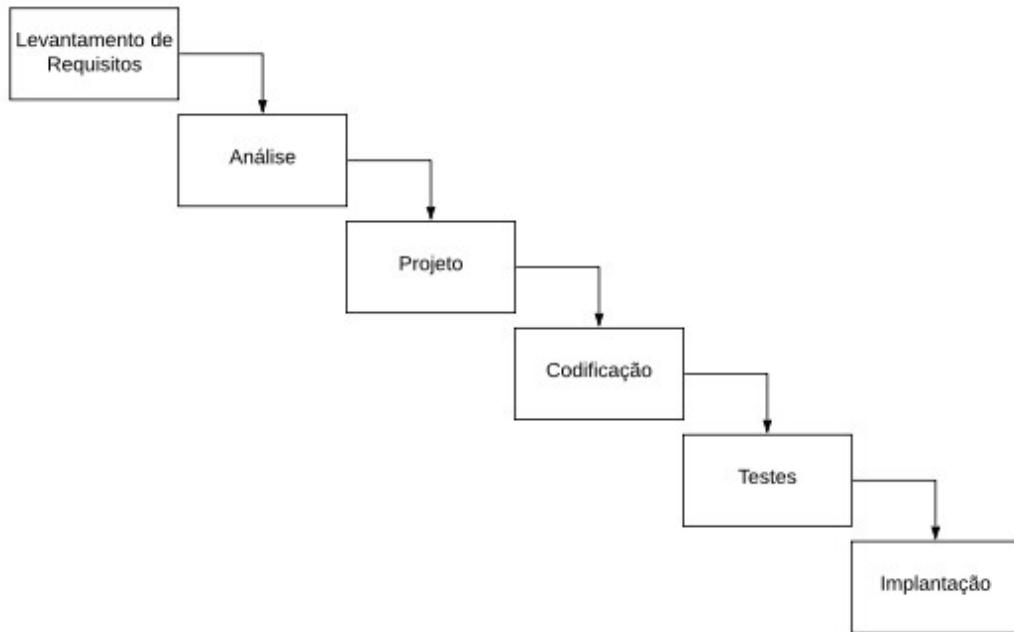
Historicamente, existem dois grandes tipos de processos que podem ser adotados na construção de sistemas de software:

- **Processos Waterfall** (ou em cascata)
- **Processos Ágeis** (ou incrementais ou iterativos).

Processos Waterfall foram os primeiros a serem propostos, ainda na década de 70, quando a Engenharia de Software começava a ganhar envergadura. De forma compreensível, eles foram inspirados nos processos usados em engenharias tradicionais, os quais são largamente sequenciais, como ilustrado no exemplo do prédio, usado no parágrafo inicial desta seção. Processos Waterfall foram muito usados até a década de 1990 e grande parte desse sucesso deve-se a uma padronização lançada pelo Departamento de Defesa Norte-Americano, em 1985. Basicamente, eles estabeleceram que todo software comprado ou contratado pelo Departamento de Defesa deveria ser construído usando Waterfall.

Processos Waterfall — também chamados de **processos dirigidos por planejamento** (*plan-driven*) — propõem que a construção de um sistema deve ser feita em etapas sequenciais, como em uma cascata de água, na qual

a água vai escorrendo de um nível para o outro. Essas etapas são as seguintes: levantamento de requisitos, análise (ou projeto de alto nível), projeto detalhado, codificação e testes. Finalizado esse pipeline, o sistema é liberado para produção, isto é, para uso efetivo pelos seus usuários, conforme ilustrado na próxima figura.



Fases de um processo Waterfall.

No entanto, processos Waterfall, a partir do final da década de 90, passaram a ser muito criticados, devido aos atrasos e problemas recorrentes em projetos de software, que ocorriam com frequência nessa época. O principal problema é que Waterfall pressupõe um levantamento completo de requisitos, depois um projeto detalhado, depois uma implementação completa, etc. Para só então validar o sistema com os usuários, o que pode acontecer anos após o início do projeto. No entanto, nesse período, o mundo pode ter mudado, bem como as necessidades dos clientes, que podem não mais precisar do sistema que ajudaram a especificar anos antes. Assim, reunidos em uma cidade de Utah, Estados Unidos, em fevereiro de 2001, um grupo de 17 Engenheiros de Software propôs um modo alternativo para construção de software, que eles chamaram de Ágil — nome do manifesto que eles produziram nessa reunião ([link](#)). Contrastando com processos Waterfall, a ideia de processos ágeis é que um sistema deve ser construído de

forma incremental e iterativa. Pequenos incrementos de funcionalidade são produzidos, em intervalos de cerca de um mês e, logo em seguida, validados pelos usuários. Uma vez que o incremento produzido seja aprovado, o ciclo se repete.

Processos ágeis tiveram um profundo impacto na indústria de software. Hoje, eles são usados pelas mais diferentes organizações que produzem software, desde pequenas empresas até as grandes companhias da Internet. Diversos métodos que concretizam os princípios ágeis foram propostos, tais como **XP, Scrum, Kanban e Lean Development**.

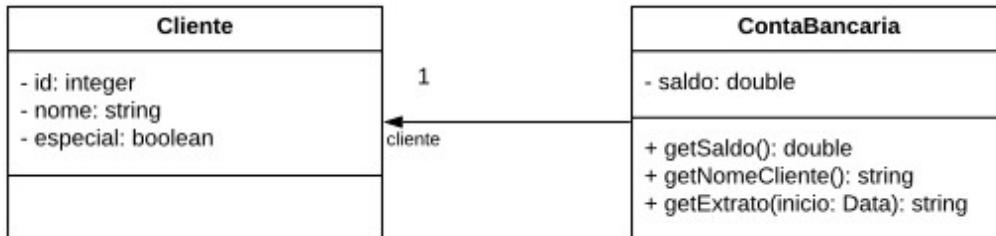
Esses métodos também ajudaram a disseminar diversas práticas de desenvolvimento de software, como **testes automatizados**, **test-driven development** (isto é, escrever os testes primeiro, antes do próprio código) e **integração contínua** (*continuous integration*). Integração contínua recomenda que desenvolvedores integrem o código que produzem imediatamente, se possível todo dia. O objetivo é evitar que desenvolvedores fiquem muito tempo trabalhando localmente, em sua máquina, sem integrar o código que estão produzindo no repositório principal do projeto. Quando o time de desenvolvimento é maior, isso aumenta as chances de conflitos de integração, que ocorrem quando dois desenvolvedores alteram em paralelo os mesmos trechos de código. O primeiro desenvolvedor a integrar seu código será bem sucedido; enquanto o segundo desenvolvedor será informado de que o trecho já foi modificado pelo primeiro.

## Modelos de Software

Um modelo oferece uma representação em mais alto nível de um sistema do que o seu código fonte. O objetivo é permitir que desenvolvedores possam analisar propriedades e características essenciais de um sistema, de modo mais fácil e rápido, sem ter que mergulhar nos detalhes do código. Modelos podem ser criados antes do código, por exemplo, ainda na fase de projeto. Nesse caso, eles são usados para apoiar **Engenharia Avante** (*Forward Engineering*); isto é, primeiro cria-se um modelo para ter um entendimento de mais alto nível de um sistema, antes de partir para a implementação do código. Por outro lado, eles podem ser criados para ajudar a entender uma base de código existente; nesse caso, eles são um instrumento de

**Engenharia Reversa** (*Reverse Engineering*). Em ambos os casos, modelos são uma forma de documentar o código de um sistema.

Frequentemente, modelos de software são baseados em notações gráficas. Por exemplo, **UML** (*Unified Modelling Language*) é uma notação que define mais de uma dezena de diagramas gráficos para representar propriedades estruturais e comportamentais de um sistema. Na próxima figura, mostra-se um diagrama UML — chamado Diagrama de Classes — para o exemplo de código usado na seção sobre Projeto de Software. Nesse diagrama, as caixas retangulares representam classes do sistema, incluindo seus atributos e métodos. As setas são usadas para denotar relacionamentos entre as classes. Existem editores para criar diagramas UML, que podem ser usados, por exemplo, em um cenário de Engenharia Avante.



Exemplo de Diagrama de Classe UML

## Qualidade de Software

Qualidade é um objetivo recorrente em produtos de engenharia. Fabricantes de automóveis, celulares, computadores, empresas de construção civil, etc., todos almejam e dizem que possuem produtos de qualidade. Esse contexto não é diferente quando o produto é um software. Segundo uma classificação proposta por Bertrand Meyer ([link](#)), qualidade de software pode ser avaliada em duas dimensões: **qualidade externa** ou **qualidade interna**.

Qualidade externa considera fatores que podem ser aferidos sem analisar o código. Assim, a qualidade externa de um software pode ser avaliada mesmo por usuários comuns, que não são especialistas em Engenharia de Software. Como exemplo, temos os seguintes fatores (ou atributos) de qualidade externa:

- Correção: o software atende à sua especificação? Nas situações normais, ele funciona como esperado?
- Robustez: o software continua funcionando mesmo quando ocorrem eventos anormais, como uma falha de comunicação ou de disco? Por exemplo, um software robusto não pode sofrer um *crash* (abortar) caso tais eventos anormais ocorram. Ele deve pelo menos avisar por qual motivo não está conseguindo funcionar conforme previsto.
- Eficiência: o software faz bom uso de recursos computacionais? Ou ele precisa de um hardware extremamente poderoso e caro para funcionar?
- Portabilidade: é possível portar esse software para outras plataformas e sistemas operacionais? Ele, por exemplo, possui versões para os principais sistemas operacionais, como Windows, Linux e macOS? Ou então, se for um app, ele possui versões para Android e iOS?
- Facilidade de Uso: o software possui uma interface amigável, mensagens de erro claras, suporta mais de uma língua, etc? Pode ser também usado por pessoas com alguma deficiência, como visual ou auditiva?
- Compatibilidade: o software é compatível com os principais formatos de dados de sua área? Por exemplo, se o software for uma planilha eletrônica, ele importa arquivos em formatos XLS e CSV?

Por outro lado, qualidade interna considera propriedades e características relacionadas com a implementação de um sistema. Portanto, a qualidade interna de um sistema somente pode ser avaliada por um especialista em Engenharia de Software e não por usuários leigos. São exemplos de fatores (ou atributos) de qualidade interna: modularidade, legibilidade do código, manutenibilidade e testabilidade.

Para garantir qualidade de software, diversas estratégias podem ser usadas. Primeiro, **métricas** podem ser usadas para acompanhar o desenvolvimento de um produto de software, incluindo métricas de código fonte e métricas de processo. Um exemplo de métrica de código é o número de linhas de um programa, que pode ser usado para dar uma ideia de seu tamanho. Métricas

de processo incluem, por exemplo, o número de defeitos reportados em produção por usuários finais em um certo intervalo de tempo.

Existem ainda práticas que podem ser adotadas para garantir a produção de software com qualidade. Modernamente, por exemplo, diversas organizações usam **revisões de código**, isto é, o código produzido por um desenvolvedor somente entra em produção depois de ser revisado e inspecionado por um outro desenvolvedor do time. O objetivo é detectar *bugs* antecipadamente, antes de o sistema entrar em produção. Além disso, revisões de código servem para garantir a qualidade interna do código — isto é, sua manutenibilidade, legibilidade, modularidade, etc. — e para disseminar boas práticas de Engenharia de Software entre os membros de um time de desenvolvimento.

A próxima figura mostra um exemplo de revisão de código, referente a um exemplo que usamos na seção sobre Testes de Software. Assumindo que a empresa que produziu esse código adotasse revisões de código, ele teria que ser analisado por um outro desenvolvedor, chamado de revisor, antes de entrar em produção. Esse revisor poderia perceber o bug e anotar o código com uma dúvida, antes de aprovar-lo. Em seguida, o responsável pelo código poderia concordar que, de fato, existe um bug, corrigir o código e submetê-lo de novo para revisão. Finalmente, ele seria aprovado pelo revisor. Existem diversas ferramentas para apoiar processos de revisão de código. No exemplo, usamos a ferramenta fornecida pelo GitHub.

The screenshot shows a GitHub pull request interface. At the top, it says "feature1". Below that, there's a diff view of a file. The first two lines are shown in a light blue background:

```
... ... @@ -0,0 +1,2 @@

```

Line 1 is highlighted in green and contains the code: `+ if (enabled)`. Line 2 is also highlighted in green and contains the code: `+ area = pi * raio * raio * raio;`.

Below the code, there's a user profile picture of a man with glasses, followed by the name "mtov" and the timestamp "12 minutes ago". There are also "Author" and "Owner" buttons. A comment from "mtov" reads: "Não seria pi \* raio \* raio?".

## Exemplo de revisão de código

### Prática Profissional

Como afirmado na frase de Bjarne Stroustrup que abre este capítulo, *nossa sociedade funciona à base de software*. Isso gera diversas oportunidades para os profissionais da área, mas também gera responsabilidades e pontos de preocupação. Questões sobre a prática profissional em Engenharia de Software iniciam-se no momento da formação, em nível de graduação, envolvendo a definição de currículos de referência e a necessidade de cursos específicos para a área, que constituam alternativas aos cursos de Ciência da Computação, Sistemas de Informação e Engenharia de Computação. Não menos importante, existem questões sobre a formação em nível técnico e tecnológico, anterior à formação universitária. Após a etapa de formação, existem questões sobre a regulamentação da profissão, por exemplo.

Por fim, mas muito atual e relevante, existem questionamentos sobre o papel e a **responsabilidade ética** dos profissionais formados em Computação, em uma sociedade na qual os relacionamentos humanos são cada vez mais mediados por algoritmos e sistemas de software. Neste sentido, as principais sociedades científicas da área possuem códigos que procuram ajudar os profissionais de Computação — não necessariamente apenas Engenheiros de Software — a exercer seu ofício de forma ética. Como exemplos, temos o Código de Ética da ACM ([link](#)) e da IEEE Computer Society ([link](#)). Esse último é interessante porque é específico para a prática de Engenharia de Software. Por exemplo, ele recomenda que:

Engenheiros de Software devem se comprometer em fazer da análise, especificação, projeto, desenvolvimento, teste e manutenção de software uma profissão benéfica e respeitada.

No Brasil, existe o Código de Ética da Sociedade Brasileira de Computação (SBC), que por ser sintético e bastante claro resolvemos reproduzir a seguir:

São deveres dos profissionais de Informática:

Art. 1o: Contribuir para o bem-estar social, promovendo, sempre que possível, a inclusão de todos os setores da sociedade.

Art. 2o: Exercer o trabalho profissional com responsabilidade, dedicação, honestidade e justiça, buscando sempre a melhor solução.

Art. 3o: Esforçar-se para adquirir continuamente competência técnica e profissional, mantendo-se sempre atualizado com os avanços da profissão.

Art. 4o: Atuar dentro dos limites de sua competência profissional e orientar-se por elevado espírito público.

Art. 5o: Guardar sigilo profissional das informações a que tiver acesso em decorrência das atividades exercidas.

Art. 6o: Conduzir as atividades profissionais sem discriminação, seja de raça, sexo, religião, nacionalidade, cor da pele, idade, estado civil ou qualquer outra condição humana.

Art. 7o: Respeitar a legislação vigente, o interesse social e os direitos de terceiros.

Art. 8o: Honrar compromissos, contratos, termos de responsabilidade, direitos de propriedade, copyrights e patentes.

Art. 9o: Pautar sua relação com os colegas de profissão nos princípios de consideração, respeito, apreço, solidariedade e da harmonia da classe.

Art. 10o: Não praticar atos que possam comprometer a honra, a dignidade e privacidade de qualquer pessoa.

Art. 11o: Nunca se apropriar de trabalho intelectual, iniciativas ou soluções encontradas por outras pessoas.

Art. 12o: Zelar pelo cumprimento deste código.

– Código de Ética da Sociedade Brasileira de Computação (SBC, 2013) ([link](#))

**Mundo Real:** O Stack Overflow realiza anualmente uma pesquisa com usuários da plataforma de perguntas e respostas. Em 2018, esse survey foi respondido por mais de 100 mil desenvolvedores, dos mais variados países. Dentre as perguntas, um grupo se referia a questões éticas ([link](#)). Uma delas perguntava se desenvolvedores têm a obrigação de considerar as implicações éticas do código que produzem. Quase 80% dos respondentes disseram que sim. Uma outra pergunta foi a seguinte: Quem, em última análise, é responsável por um código que colabora para um comportamento antiético? Nesse caso, 57,5% responderam que é a alta gerência da organização ou empresa, enquanto 23% disseram que é o próprio desenvolvedor. Quando perguntados se concordariam em escrever um código com dúvidas éticas, 58% responderam que não e 37% responderam que dependeria do código requisitado.

## Aspectos Econômicos

Diversas decisões e questões econômicas se entrelaçam com o desenvolvimento de sistemas. Por exemplo, uma startup de software deve decidir qual o modelo de rentabilização pretende adotar, se baseado em assinaturas ou em anúncios. Desenvolvedores de apps para celulares têm que decidir sobre o preço que irão cobrar pela sua aplicação, o que, dentre outras variáveis, requer conhecimento sobre o preço das apps concorrentes. Por isso, não é surpresa que grandes companhias de software atualmente empreguem economistas, para avaliarem os aspectos econômicos dos sistemas que produzem.

Para discutir um caso mais concreto, em economia existe uma preocupação frequente com os custos de oportunidade de uma decisão. Isto é, toda decisão possui um custo de oportunidade, que são as oportunidades preteridas quando se descartou uma das decisões alternativas. Em outras palavras, quando se descarta uma decisão Y e, no seu lugar, toma-se uma decisão X, os eventuais benefícios de Y passaram a ser oportunidades perdidas. Por exemplo, suponha que o principal sistema de sua empresa tenha uma lista de bugs B para ser corrigida. Existem benefícios em corrigir B? Claro, isso vai deixar os clientes atuais mais satisfeitos; eles não vão pensar em migrar para sistemas concorrentes, etc. Porém, existe também um custo de oportunidade nessa decisão. Especificamente, em vez de corrigir B, a empresa poderia investir em novas funcionalidades F, que poderiam ajudar

a ampliar a base de clientes. O que é melhor? Corrigir os bugs ou implementar novas funcionalidades? No fundo, essa é uma decisão econômica.

## Classificação de Sistemas de Software

Atualmente, como estamos ressaltando nesta Introdução, software permeia as mais distintas atividades humanas. Ou seja, temos software de todos os tamanhos, em todas as atividades, com os mais diferentes requisitos funcionais e não-funcionais, desenvolvidos por 1-2 desenvolvedores ou por grandes corporações da Internet, etc. O risco é então achar que existe um único modo de desenvolver software. Em outras palavras, que todo software deve ser construído usando o mesmo processo de desenvolvimento, os mesmos princípios de projeto, os mesmos mecanismos de garantia de qualidade, etc.

Uma classificação proposta por Bertrand Meyer ([link](#)) ajuda a distinguir e entender os diferentes sistemas de software que podem ser construídos e os princípios de Engenharia de Software mais recomendados para cada uma das categorias propostas. Segundo essa classificação, existem três tipos principais de software:

- **Sistemas A (Acute)**
- **Sistemas B (Business)**
- **Sistemas C (Casuais)**

Vamos discutir primeiro os Sistemas C e A (isto é, os sistemas em cada um dos extremos da classificação) e depois os Sistemas B.

Sistemas C (Casuais) não sofrem pressão para terem níveis altos de qualidade. São sistemas que podem ter alguns bugs, os quais não vão comprometer fundamentalmente o seu funcionamento. Como exemplo, podemos citar um script feito para um trabalho acadêmico, um programa de conversão de dados (que vai ser usado uma única vez, para converter os dados para um novo banco de dados que está sendo comprado pela

empresa), um sistema para controlar os sócios do Diretório Acadêmico da universidade, um sistema para gerenciar as salas disponíveis para reuniões em uma empresa, etc. Por isso, Sistemas C não precisam ter níveis altos de qualidade interna; por exemplo, podem ter parte do código duplicado. Também não precisam ter desempenho ou uma boa interface. Em geral, são desenvolvidos por 1-2 programadores; ou seja, são sistemas pequenos e não críticos. Por tudo isso, eles não se beneficiam tanto das práticas, técnicas e processos estudados neste livro. Pelo contrário, no caso de Sistemas C, o maior risco é **over-engineering**, ou seja, o uso de recursos mais sofisticados em um contexto que não demanda tanta preocupação. Como se diz coloquialmente, Engenharia de Software nesse contexto equivale a usar uma bala de canhão para matar formigas.

No outro extremo, temos os Sistemas A (de *acute*, ou de missão crítica). São sistemas nos quais qualquer falha pode causar um imenso prejuízo, incluindo a perda de vidas humanas. São sistemas para controlar um carro autônomo, uma usina nuclear, um avião, os equipamentos de uma UTI, um trem de metrô, etc. O exemplo do sistema de controle do foguete Ariane 5, usado na seção sobre Testes de Software, é um exemplo de Sistema A. O desenvolvimento desses sistemas deve ser feito de acordo com processos rígidos, incluindo rigorosa revisão de código e certificação por organizações externas. É comum exigir redundância não apenas em hardware, mas também no próprio software. Por exemplo, o sistema roda de forma paralela em duas máquinas e uma decisão somente é tomada caso ambas as instâncias cheguem ao mesmo resultado. Por fim, sistemas A muitas vezes são especificados em uma linguagem formal, baseada em teoria de conjuntos ou lógica.

**Aviso:** Por tudo que foi afirmado no parágrafo anterior, **sistemas A (isto é, de missão crítica) não serão tratados neste livro.**

Sobram os sistemas B (*Business*), que são exatamente aqueles que vão se beneficiar dos conceitos estudados neste livro. Esses sistemas incluem as mais variadas aplicações corporativas (financeiras, recursos humanos, logística, vendas, contabilidade, etc.), sistemas Web dos mais variados tipos, desde sistemas com poucas páginas até grandes redes sociais ou sistemas de busca. Outras aplicações incluem bibliotecas e frameworks de software, aplicações de uso geral (como editores de texto, planilhas, editores de

imagens, etc.) e sistemas de software básico (como compiladores, gerenciadores de bancos de dados, IDEs, etc.). Nesses sistemas, os princípios e práticas de Engenharia de Software estudados neste livro podem contribuir com dois benefícios principais: (1) eles podem tornar mais produtivo o desenvolvimento de Sistemas B; (2) eles podem propiciar a construção de Sistemas B com melhor qualidade, tanto interna (por exemplo, sistemas mais fáceis de serem mantidos) como externa (por exemplo, sistemas com menor quantidade de bugs em produção).

## Próximos Capítulos

Este livro possui **10 capítulos e um apêndice**:

**Capítulo 2: Processos**, com foco em processos ágeis de desenvolvimento, especificamente XP, Scrum e Kanban. Tomamos a decisão de focar em métodos ágeis porque eles são largamente usados hoje em dia no desenvolvimento dos mais variados tipos de sistemas, dos mais variados domínios e tamanhos. Tratamos também de processos tradicionais, como Waterfall e o Processo Unificado, porém de forma resumida e, também, para fazer o contraste com métodos ágeis.

**Capítulo 3: Requisitos**, que inicia com uma discussão sobre a importância de requisitos e os principais tipos de requisitos. Então, apresentamos duas técnicas para levantamento e validação de requisitos: Histórias de Usuário (usadas com métodos ágeis) e Casos de Uso (uma técnica tradicional, que é mais usada com métodos dirigidos por planejamento e documentação). Por fim, apresentamos dois assuntos que, apesar de importantes e atuais, não são ainda tratados nos livros tradicionais: Produto Mínimo Viável (MVPs) e Testes A/B. Argumentamos que esses dois conceitos não são importantes apenas em startups, mas também em empresas que desenvolvem software para mercados mais estáveis.

**Capítulo 4: Modelos**, que tem foco no uso de UML para elaboração de esboços (*sketches*) de software. Modernamente, concordamos que UML não é mais usada para os fins que ela foi concebida na década de 90, ou seja, para criação de modelos detalhados de software. Praticamente, não existem mais casos de empresas que investem meses — ou anos — na elaboração de

diagramas gráficos antes de começar a implementar qualquer linha de código. Porém, se não tratássemos de UML no livro ficaríamos com a sensação de que após o banho, jogamos o bebê fora, junto com a água da bacia. Se por um lado não faz sentido estudar todos os diagramas da UML em detalhes, por outro lado existem elementos importantes em alguns desses diagramas. Além disso, desenvolvedores, com frequência, elaboram pequenos esboços de software, por exemplo, para comunicar e discutir ideias de design com outros desenvolvedores. Assim, conhecimento básico de UML pode ser interessante para desenhar esses esboços, inclusive para evitar a criação de uma nova linguagem de modelagem.

**Capítulo 5: Princípios de Projeto**, que trata de dois temas que devem ser do conhecimento de todo projetista de software. São eles: (1) propriedades (ou considerações) importantes em projeto de software, incluindo integridade conceitual, ocultamento de informação, coesão e acoplamento; (2) princípios de projeto, os quais constituem recomendações mais específicas para construção de bons projetos de software, tais como responsabilidade única, prefira composição a herança, aberto/fechado, Demeter, etc.

**Capítulo 6: Padrões de Projeto**, os quais constituem um catálogo de soluções para problemas comuns de projeto de software. Neste capítulo, vamos estudar os principais padrões de projeto definidos no livro clássico sobre o tema. A discussão de cada padrão será dividida em três partes: (1) um contexto, isto é, um sistema no qual o padrão pode ser útil; (2) um problema no projeto desse sistema; (3) uma solução para esse problema por meio de padrões. Iremos também apresentar diversos exemplos de código, para facilitar o entendimento e a discussão prática do uso de cada padrão. O código de alguns exemplos mais complexos será disponibilizado no GitHub.

**Capítulo 7: Arquitetura**, que inicia com uma apresentação e discussão sobre Arquitetura de Software. O objetivo é deixar claro que arquitetura deve ser vista como projeto em alto nível, envolvendo pacotes, camadas ou serviços, em vez de classes individuais. Em seguida, discutimos cinco padrões arquiteturais: arquitetura em camadas (incluindo 3-camadas), arquitetura MVC (incluindo single-page applications), microsserviços, arquiteturas orientadas por filas de mensagens e arquiteturas publish/subscribe. Essas duas últimas são comuns na construção de sistemas

distribuídos fracamente acoplados. Por fim, apresentamos um anti-padrão arquitetural, chamado *big ball of mud*, que é um termo usado para designar sistemas sem organização arquitetural. Esses sistemas poderiam até possuir uma arquitetura no seu início, mas depois o projeto arquitetural deles foi sendo abandonado, transformando-os em um espaguete de dependências entre os seus módulos.

**Capítulo 8: Testes**, com ênfase em testes de unidade, usando frameworks como JUnit. O capítulo inclui dezenas de exemplos de testes de unidade e discute diversos aspectos desses testes. Por exemplo, discutimos bons princípios para escrita de testes de unidade e também test smells, isto é, padrões de testes que não são recomendados. Em seguida, tratamos de testabilidade, isto é, discutimos a importância de escrever código que possa ser facilmente testado. O capítulo inclui uma seção inteira sobre mocks e stubs, os quais são objetos que viabilizam o teste de unidade de código com dependências mais complexas, como dependências para bancos de dados e outros sistemas externos. Finalizada a discussão sobre testes de unidade, também discutimos, porém de forma mais resumida, dois outros tipos de testes: testes de integração e testes de sistema. Esses testes verificam propriedades de unidades maiores de código, como as classes responsáveis por um determinado serviço ou funcionalidade (testes de integração) ou mesmo todas as classes de um sistema (testes de sistema). Para terminar, incluímos uma discussão sobre outros testes, como testes caixa-preta (ou testes funcionais), testes caixa-branca (ou testes estruturais), testes de aceitação e também testes para verificar requisitos não-funcionais, como desempenho, falhas e usabilidade.

**Capítulo 9: Refactoring**, cujo principal conteúdo é uma apresentação dos principais refactorings que podem ser realizados para melhorar a qualidade interna de um sistema de software. A apresentação inclui vários exemplos de código fonte, alguns deles de refactorings reais, realizados em sistemas de código aberto. O objetivo é transmitir ao leitor uma experiência prática de refatoração, que o ajude a desenvolver o hábito de frequentemente reservar tempo para melhorar a qualidade interna do código que ele vai desenvolver. No capítulo, também apresentamos uma lista de code smells, isto é, indicadores de que uma estrutura de código não está cheirando bem e que, portanto, deve ser objeto de uma refatoração.

**Capítulo 10: DevOps**, que é um movimento que tenta aproximar os times de desenvolvimento (Devs) e de operações (Ops) de uma empresa desenvolvedora de software. O time de operações é responsável por manter o software em funcionamento, sendo formado por administradores de rede, administradores de bancos de dados, técnicos de suporte, etc. Em uma cultura tradicional, esses dois times tendem a atuar de forma independente. Ou seja, o time de desenvolvimento desenvolve o sistema e depois joga ele por cima da parede (*throw it over the wall*) que separa o departamento de desenvolvimento do departamento de operações. Assim, os dois times não conversam nem no momento da passagem de bastão (*hand-off*) de uma área para outra. Para resolver esse problema, DevOps propõe uma interação constante entre as áreas Devs e Ops, desde os primeiros dias do desenvolvimento. O objetivo é acelerar a entrada em produção de um sistema. Além de uma introdução a DevOps, vamos estudar algumas práticas essenciais quando uma empresa adota essa cultura, incluindo Controle de Versões, Integração Contínua e Deployment/Entrega Contínua.

**Apêndice A: Git**, que apresenta e mostra exemplos de uso dos principais comandos do sistema git. Atualmente, é inconcebível não usar controle de versões em qualquer sistema, mesmo naqueles mais simples. Por isso, fizemos questão de acrescentar esse apêndice no livro. Git é o sistema de controle de versões mais usado atualmente.

## Bibliografia

Pierre Bourque, Richard Fairley. Guide to the Software Engineering Body of Knowledge, Version 3.0, IEEE Computer Society, 2014.

Armando Fox, David Patterson. Construindo Software como Serviço: Uma Abordagem Ágil Usando Computação em Nuvem. Strawberry Canyon LLC. 1a edição, 2014.

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

## Exercícios de Fixação

1. Segundo Frederick Brooks, desenvolvimento de software enfrenta dificuldades essenciais (para as quais não há bala de prata) e acidentais (para as quais existe uma solução melhor). Dê um exemplo de dificuldade accidental que já tenha experimentado ao desenvolver programas, mesmo que pequenos. Sugestão: elas podem estar relacionadas a ferramentas que tenha usado, como compiladores, IDEs, bancos de dados, sistemas operacionais, etc.
2. Diferencie requisitos funcionais de requisitos não-funcionais.
3. Explique por que testes podem ser considerados tanto uma atividade de verificação como de validação de software. Qual tipo de teste é mais adequado se o objetivo for verificação? Qual tipo de teste é mais adequado se o objetivo for validar um sistema de software?
4. Por que testes não conseguem provar a *ausência* de bugs?
5. Suponha um programa que tenha uma única entrada: um inteiro de 64 bits. Em um teste exaustivo, temos que testar esse programa com todos os possíveis inteiros (logo, 2<sup>64</sup>). Se cada teste levar 1 nanosegundo (10<sup>-9</sup> segundos), quanto tempo levará esse teste exaustivo?
6. Se considerarmos o contexto histórico, por que foi natural que os primeiros processos de desenvolvimento de software tivessem características sequenciais e fossem baseados em planejamento e documentação detalhados?
7. Alguns estudos mostram que os custos com manutenção e evolução podem alcançar 80% ou mais dos custos totais alocados a um sistema de software, durante todo o seu ciclo de vida. Explique por que esse valor é tão alto.
8. Refactoring é uma transformação de código que preserva comportamento. Qual o significado da expressão *preservar comportamento*? Na prática, qual restrição ela impõe a uma operação de refactoring?
9. Dê exemplos de sistemas A (*Acute*, ou críticos) e B (*Business*, ou comerciais) com os quais já tenha interagido.

10. Dê exemplos de sistemas C (casuais) que você já tenha desenvolvido.

11. Em 2015, descobriu-se que mais de 11 milhões de carros da Volkswagen emitiam poluentes dentro das normas legais apenas quando eles estavam sendo testados em um laboratório de certificação. Fora do laboratório, os carros emitiam mais poluentes, para melhorar o desempenho. Ou seja, o código incluía uma estrutura de decisão como a seguinte (meramente ilustrativa, para fins deste exercício):

```
if "Carro sendo testado em um laboratório"
    "Emita poluentes dentro das normas"
else
    "Emita poluentes fora das normas"
```

O que você faria se seu chefe pedisse para escrever um *if* como o acima? Para mais informações sobre esse episódio, consulte essa página da [Wikipedia](#).

# Cap 2 Processos

Este capítulo inicia com uma apresentação sobre a importância de processos de software (Seção 2.1). Em seguida, discutimos questões gerais e preliminares sobre processos ágeis de desenvolvimento de software (Seção 2.2), incluindo uma apresentação sobre o contexto histórico que motivou o surgimento desse tipo de processo. As próximas seções tratam de três métodos ágeis: Extreme Programming (Seção 2.3), Scrum (Seção 2.4) e Kanban (Seção 2.5). Depois, temos uma seção dedicada a discutir quando métodos ágeis não são recomendados (Seção 2.6). Por fim, na Seção 2.7, discutimos alguns processos tradicionais, principalmente o Processo Unificado.

## Importância de Processos

A produção de um carro em uma fábrica de automóveis segue um processo bem definido. Sem estender a explicação, primeiro, as chapas de aço são cortadas e prensadas, para ganhar a forma de portas, tetos e capôs. Depois, o carro é pintado e instalaram-se painel, bancos, cintos de segurança e a fiação. Por fim, instala-se a parte mecânica, incluindo motor, suspensão e freios.

Assim como carros, software também é produzido de acordo com um **processo**, embora certamente menos mecânico e mais dependente de esforço intelectual. Um processo de desenvolvimento de software define um conjunto de passos, tarefas, eventos e práticas que devem ser seguidos por desenvolvedores de software, na produção de um sistema.

Alguns críticos de processos de software costumam fazer a seguinte pergunta: por que eu preciso seguir um processo? E complementam, perguntando também o seguinte: Qual processo Linus Torvalds usou na implementação do sistema operacional Linux? Ou que Donald Knuth usou na implementação do formatador de textos TeX?

Na verdade, a segunda parte da pergunta não faz muito sentido, pois tanto o Linux (no seu início) e o TeX são projetos individuais, liderados por um único desenvolvedor. Nesses casos, a adoção de um processo é menos

importante. Ou, dizendo de outra forma, o processo em tais projetos é pessoal, composto pelos princípios, práticas e decisões tomadas pelo seu único desenvolvedor; e que terão impacto apenas sobre ele mesmo.

Porém, os sistemas de software atuais são por demais complexos para serem desenvolvidos por uma única pessoa. Por isso, casos de sistemas desenvolvidos por heróis serão cada vez mais raros. Na prática, os sistemas modernos — que nos interessam neste livro — são desenvolvidos em **equipes**.

E essas equipes, para produzir software com qualidade e produtividade, precisam de um ordenamento, mesmo que mínimo. Por isso, empresas dão tanto valor a processos de software. Eles são o instrumento de que as empresas dispõem para coordenar, motivar, organizar e avaliar o trabalho de seus desenvolvedores, de forma a garantir que eles trabalhem com produtividade e produzam sistemas alinhados com os objetivos da organização. Sem um processo — mesmo que simplificado e leve, como os processos ágeis que estudaremos neste capítulo — existe o risco de que os times de desenvolvimento passem a trabalhar de forma descoordenada, gerando produtos sem valor para o negócio da empresa. Por fim, processos são importantes não apenas para a empresa, mas também para os desenvolvedores, pois permitem que eles tomem consciência das tarefas e resultados que se esperam deles. Sem um processo, os desenvolvedores podem se sentir perdidos, trabalhando de forma errática e sem alinhamento com os demais membros do time.

Neste capítulo, vamos estudar alguns processos de software. Na verdade, no Capítulo 1 já comentamos sobre Processos Waterfall e Ágeis. Na próxima seção, vamos retomar essa discussão e, em seguida, descrever alguns métodos de desenvolvimento de software.

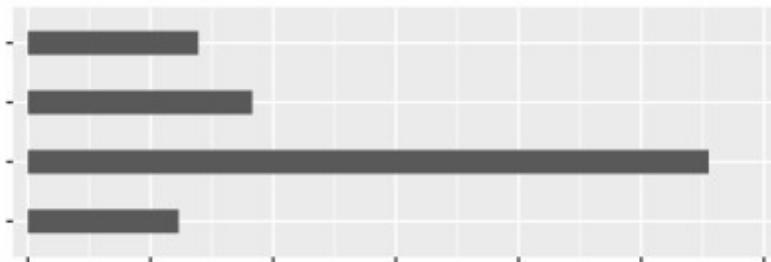
## Manifesto Ágil

Os primeiros processos de desenvolvimento de software — do tipo Waterfall, propostos ainda na década de 70 — eram estritamente sequenciais, começando com uma fase de especificação de requisitos até chegar às fases finais de implementação, testes e manutenção do sistema.

Se considerarmos o contexto histórico, essa primeira visão de processo era natural, visto que projetos de Engenharia tradicional também são sequenciais e precedidos de um planejamento detalhado. Todas as fases também geram documentações detalhadas do produto que está sendo desenvolvido. Por isso, nada mais natural que a nascente Engenharia de Software se espelhasse nos processos de áreas mais tradicionais, como Engenharia Eletrônica, Civil, Mecânica, Aeronáutica, etc.

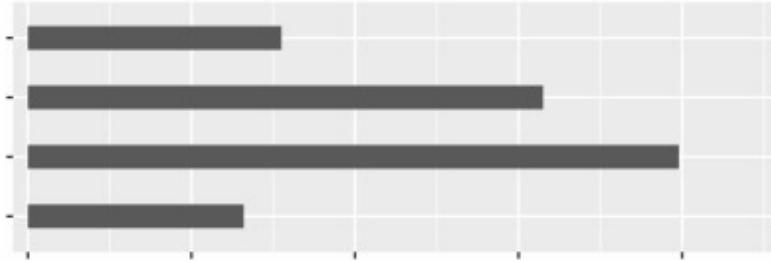
No entanto, após cerca de uma década, começou-se a perceber que software é diferente de outros produtos de Engenharia. Essa percepção foi ficando clara devido aos problemas frequentes enfrentados por projetos de software nas décadas de 70 a 90. Por exemplo, os cronogramas e orçamentos desses projetos não eram obedecidos. Não raro, projetos inteiros eram cancelados, após anos de trabalho, sem entregar um sistema funcional para os clientes.

Em 1994, um relatório produzido pela empresa de consultoria Standish Group revelou informações mais detalhadas sobre os projetos de software da época. Por exemplo, o relatório, que ficou conhecido pelo sugestivo nome de CHAOS Report ([link](#)), mostrou que mais de 55% dos projetos estourava os prazos planejados entre 51% e 200%; pelo menos 12% estouravam os prazos acima de 200%, conforme mostra o próximo gráfico.



CHAOS Report (1994): percentual de projetos que estourava seus prazos (para cada faixa de estouro).

Os resultados em termos de custos não eram mais animadores: quase 40% dos projetos ultrapassava o orçamento entre 51% e 200%, como mostra o seguinte gráfico:



CHAOS Report (1994): percentual de projetos que estourava seus orçamentos (para cada faixa de estouro).

Em 2001, um grupo de profissionais da indústria se reuniu na cidade de Snowbird, no estado norte-americano de Utah, para discutir e propor uma alternativa aos processos do tipo Waterfall que então predominavam. Essencialmente, eles passaram a defender que software é diferente de produtos tradicionais de Engenharia. Por isso, software também demanda um processo de desenvolvimento diferente.

Por exemplo, os requisitos de um software mudam com frequência, mais do que os requisitos de um computador, de um avião ou de uma ponte. Além disso, os clientes frequentemente não têm uma ideia precisa do que querem. Ou seja, corre-se o risco de projetar por anos um produto que depois de pronto não será mais necessário, ou porque o mundo mudou ou porque os planos e as necessidades dos clientes mudaram. Eles diagnosticaram ainda problemas nos documentos prescritos por processos Waterfall, incluindo documentos de requisitos, fluxogramas, diagramas, etc. Esses documentos eram detalhados, pesados e extensos. Assim, rapidamente se tornavam obsoletos, pois quando os requisitos mudavam os desenvolvedores não propagavam as alterações para a documentação, mas apenas para o código.

Então eles decidiram lançar as bases para um novo conceito de processo de software, as quais foram registradas em um documento que chamaram de **Manifesto Ágil**. Por ser curto, iremos reproduzir o texto do manifesto:

Por meio deste trabalho, passamos a valorizar:

**Indivíduos e interações**, mais do que processos e ferramentas

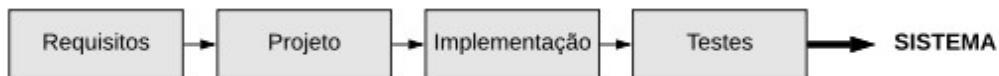
**Software em funcionamento**, mais do que documentação abrangente

**Colaboração com o cliente**, mais do que negociação de contratos

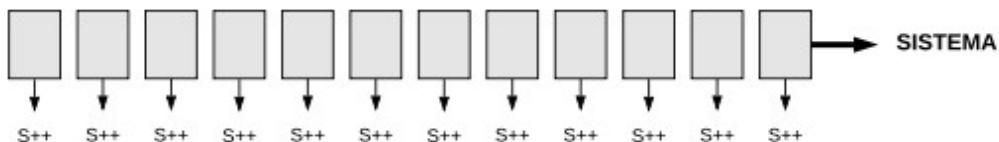
**Resposta a mudanças**, mais do que seguir um plano.

A característica principal de processos ágeis é a adoção de **ciclos curtos e iterativos de desenvolvimento**, por meio dos quais um sistema é implementado de forma gradativa; começando por aquilo que é mais urgente para o cliente. De início, implementa-se uma primeira versão do sistema, com as funcionalidades que segundo o cliente são para ontem, isto é, possuem prioridade máxima. Em seguida, essa versão é validada pelo cliente. Se ela for aprovada, um novo ciclo — ou **iteração** — inicia-se, com mais algumas funcionalidades, também priorizadas pelos clientes. Normalmente, esses ciclos são curtos, com duração de um mês, talvez até um pouco menos. Assim, o sistema vai sendo construído de forma incremental, sendo cada incremento devidamente aprovado pelos clientes. O desenvolvimento termina quando o cliente decide que todos os requisitos estão implementados.

As próximas figuras comparam desenvolvimento em Waterfall e Ágil.



Desenvolvimento usando um Processo Waterfall. O sistema fica pronto apenas no final.



Desenvolvimento usando um Processo Ágil. A cada iteração (representada pelos retângulos) gera-se um incremento no sistema ( $S++$ ), que já pode ser validado e testado pelos usuários finais.

No entanto, a figura anterior pode sugerir que, em desenvolvimento ágil, cada iteração é um mini-waterfall, incluindo todas as fases de um processo Waterfall. Isso não é verdadeiro; em geral, as iterações em métodos ágeis não são um pipeline de tarefas, como em Waterfall (mais sobre isso nas próximas seções). A figura também pode sugerir que ao final de cada iteração tem-se que colocar um sistema em produção, para uso pelos usuários finais. Isso também não é verdade. De fato, o objetivo é entregar um sistema funcional, isto é, que realize tarefas úteis. Porém, a decisão de colocá-lo em produção envolve outras variáveis, como riscos para o negócio da empresa, disponibilidade de servidores, campanhas de marketing, elaboração de manuais, treinamento de usuários, etc.

Outras características de processos ágeis incluem:

- **Menor ênfase em documentação**, ou seja, apenas o essencial deve ser documentado.
- **Menor ênfase em planos detalhados**, pois muitas vezes nem o cliente, nem os Engenheiros de Software têm, no início de um projeto, uma ideia clara dos requisitos que devem ser implementados. Esse entendimento vai surgir ao longo do caminho, à medida que incrementos de produto sejam produzidos e validados. Em outras palavras, o importante em desenvolvimento ágil é conseguir avançar, mesmo em ambientes com informações imperfeitas, parciais e sujeitas a mudanças.
- **Inexistência de uma fase dedicada a design (*big design up front*)**. Em vez disso, o design também é incremental. Ele evolui à medida que o sistema vai nascendo, ao final de cada iteração.
- **Desenvolvimento em times pequenos**, com cerca de uma dezena de desenvolvedores. Ou, em outras palavras, times que possam ser alimentados com duas pizzas, conforme popularizado pelo CEO da Amazon, Jeff Bezos.
- **Ênfase em novas práticas de desenvolvimento** (pelo menos, para o início dos anos 2000), como **programação em pares, testes**

**automatizados e integração contínua.**

Devido a essas características, processos ágeis são considerados **processos leves**, com poucas prescrições e documentos.

No entanto, as características acima são genéricas e abrangentes; por isso alguns métodos foram propostos para ajudar desenvolvedores a adotar os princípios ágeis, de forma mais concreta. O interessante é que todos eles foram propostos, pelo menos na primeira versão, antes do encontro de Utah, em 2001, que lançou o Manifesto Ágil.

Neste capítulo, vamos estudar três métodos ágeis:

- **Extreme Programming (XP)**, proposto por Kent Beck, em um livro lançado em 1999 ([link](#)). Uma segunda edição do livro, incluindo uma grande revisão, foi lançada em 2004. Neste capítulo, vamos nos basear nessa edição mais recente.
- **Scrum**, proposto por Jeffrey Sutherland e Ken Schwaber, em um artigo publicado em 1995 ([link](#)).
- **Kanban**, cujas origens remontam a um sistema de controle de produção que começou a ser usado nas fábricas da Toyota, ainda na década de 50 ([link](#)). Nos últimos 10 anos, Kanban tem sido gradativamente adaptado para uso no desenvolvimento de software.

**Aprofundamento:** Neste livro, usamos os termos **processos** e **métodos**. Processo é o conjunto de passos, etapas e tarefas que se usa para construir um software. Toda organização usa um processo para desenvolver seus sistemas, o qual pode ser ágil ou waterfall, por exemplo. Ou, talvez, esse processo pode ser caótico. Porém, o ponto que queremos reforçar é que sempre existe um processo. Já método, no nosso contexto, define e especifica um determinado processo de desenvolvimento (a palavra método tem sua origem no grego, onde significa caminho para se chegar a um objetivo). Assim, XP, Scrum e Kanban são métodos ágeis ou, de modo mais extenso, são métodos que definem práticas, atividades, eventos e técnicas compatíveis com princípios ágeis de desenvolvimento de software. Aproveitando que estamos tratando de definições, frequentemente usa-se

também o termo **metodologia** quando se fala de processos de software. Por exemplo, é comum ver referências a metodologias para desenvolvimento de software, metodologias ágeis, metodologia orientada a objetos, etc. A palavra metodologia, no sentido estrito, denota o ramo da lógica que se ocupa dos métodos das diferentes ciências, segundo o Dicionário Houaiss. No entanto, a palavra também pode ser usada como sinônimo de método, segundo o mesmo dicionário. Apesar disso, neste livro evitamos usar o termo metodologia e tentamos employar sempre o termo método.

**Aviso:** Todo método de desenvolvimento deve ser entendido como um conjunto de recomendações; cabe a uma organização analisar cada uma e decidir se ela faz sentido no seu contexto. Como resultado, a organização pode ainda decidir por adaptar essas recomendações para atender às suas necessidades. Logo, provavelmente, não existem duas organizações que seguem exatamente o mesmo processo de desenvolvimento, mesmo que elas digam que estão desenvolvendo usando Scrum, por exemplo.

**Mundo Real:** O sucesso e impacto de processos ágeis foi impressionante. Hoje, a grande maioria das empresas que desenvolvem software, independente de seu tamanho ou do foco de seu negócio, usam princípios ágeis, em maior ou menor escala. Para citar alguns dados, em 2018, o Stack Overflow survey incluiu uma pergunta sobre o método de desenvolvimento mais usado pelos respondentes ([link](#)). Essa pergunta recebeu 57 mil respostas de desenvolvedores profissionais e a grande maioria mencionou métodos ou práticas ágeis, incluindo aquelas que vamos estudar neste capítulo, como Scrum (63% das respostas), Kanban (36%) e Extreme Programming (16%). Apenas 15% dos participantes marcaram Waterfall como resposta.

## Extreme Programming

Segundo seu autor, XP é um método leve recomendado para desenvolver software com requisitos vagos ou sujeitos a mudanças; isto é, basicamente sistemas comerciais, na classificação que adotamos no Capítulo 1. Sendo um método ágil, XP possui todas as características que mencionamos na seção anterior, isto é: adota ciclos curtos e iterativos de desenvolvimento, concede menos ênfase para documentação e para planos detalhados, propõe que o

design de um sistema também seja definido de forma incremental e sugere que as equipes de desenvolvimento sejam pequenas.

Porém, XP não é um método prescritivo, que define um passo a passo detalhado para construção de software. Em vez disso, XP é definido por meio de um conjunto de **valores**, **princípios** e **práticas de desenvolvimento**. Ou seja, XP é inicialmente definido de forma abstrata, usando-se de valores e princípios que devem fazer parte da cultura e dos hábitos de times de desenvolvimento de software. Depois, esses valores e princípios são concretizados em uma lista de práticas de desenvolvimento. Frequentemente, quando decidem adotar XP, desenvolvedores e organizações concentram-se nas práticas. Porém, os valores e princípios são componentes chaves do método, pois são eles que dão sentido às práticas propostas em XP. Sendo mais claro, se uma organização não está preparada para trabalhar no modelo mental de XP — representado pelos seus valores e princípios — recomenda-se também não adotar suas práticas.

Neste capítulo, vamos primeiro apresentar os valores e princípios de XP. Veja uma lista deles a seguir:

- **Valores:** comunicação, simplicidade, feedback, coragem, respeito e qualidade de vida.
- **Princípios:** humanidade, economicidade, benefícios mútuos, melhorias contínuas, falhas acontecem, baby steps e responsabilidade pessoal.

Em seguida, vamos descrever as práticas. Para facilitar a explicação delas, resolvemos organizá-las em três grupos: práticas sobre o processo de desenvolvimento, práticas de programação e práticas de gerenciamento de projetos. Veja a seguir uma lista das práticas em cada grupo:

- **Práticas sobre o Processo de Desenvolvimento:** representante dos clientes, histórias dos usuários, iterações, releases, planejamento de releases, planejamento de iterações, planning poker, slack.
- **Práticas de Programação:** design incremental, programação pareada, desenvolvimento dirigido por testes (TDD), build automatizado,

integração contínua.

- **Práticas de Gerenciamento de Projetos:** métricas, ambiente de trabalho, contratos com escopo aberto.

## Valores

XP defende que o desenvolvimento de projetos de software seja norteado por três valores principais: comunicação, simplicidade e feedback. Na verdade, argumenta-se que esses valores são universais, para convívio humano. Ou seja, eles não servem apenas para guiar projetos de desenvolvimento, mas a própria vida em sociedade. Uma boa **comunicação** é importante em qualquer projeto, não apenas para evitar, mas também para aprender com erros. O segundo valor de XP é **simplicidade**, pois em todo sistema complexo e desafiador existem sistemas ou subsistemas mais simples, que às vezes não são considerados. Por último, existem riscos em todos os projetos de software: os requisitos mudam, a tecnologia muda, a equipe de desenvolvimento muda, o mundo muda, etc. Um valor que ajuda a controlar tais riscos é estar aberto ao **feedback** dos stakeholders, a fim de que correções de rota sejam implementadas o quanto antes. Em outras palavras, é difícil desenvolver o sistema de software certo em uma primeira e única tentativa. Frederick Brooks tem uma frase conhecida sobre esse fenômeno:

Planeje-se para jogar fora partes de seu sistema, pois você fará isso.

Por isso, feedback é um valor essencial para garantir que as partes ou versões que serão descartadas sejam identificadas o quanto antes, de forma a diminuir prejuízos e retrabalho. Além dos três valores mencionados, XP também defende outros valores, como **coragem, respeito e qualidade de vida**.

## Princípios

Os valores que mencionamos são abstratos e universais. Por outro lado, as práticas que vamos mencionar mais adiante são procedimentos concretos e pragmáticos. Assim, para unir esses dois extremos, XP defende que projetos de software devem seguir um conjunto de princípios. A imagem que se

apresenta é de um rio: de um lado estão os valores e de outro as práticas. Os princípios — que descreveremos agora — fazem o papel de uma ponte ligando esses dois lados. Alguns dos principais princípios de XP são os seguintes:

**Humanidade** (*humanity*, em inglês). Software é uma atividade intensiva no uso de capital humano. O principal recurso de uma empresa de software não são seus bens físicos — computadores, prédios, móveis ou conexões de Internet, por exemplo — mas sim seus colaboradores. Um termo que reflete bem esse princípio é *peopleware*, que foi cunhado por Tom DeMarco, em um livro com o mesmo título ([link](#)). A ideia é que a gestão de pessoas — incluindo fatores como expectativas, crescimento, motivação, transparência e responsabilidade — é fundamental para o sucesso de projetos de software.

**Economicidade** (*economics*, em inglês). Se por um lado, *peopleware* é fundamental, por outro lado software é uma atividade cara, que demanda a alocação de recursos financeiros consideráveis. Logo, tem-se que ter consciência de que o outro lado, isto é, quem está pagando as contas do projeto, espera resultados econômicos e financeiros. Por isso, na grande maioria dos casos, software não pode ser desenvolvido apenas para satisfazer a vaidade intelectual de seus desenvolvedores. Software não é uma obra de arte, mas algo que tem que gerar resultados econômicos, como defendido por esse princípio de XP.

**Benefícios Mútuos.** XP defende que as decisões tomadas em um projeto de software têm que beneficiar múltiplos stakeholders. Por exemplo, o contratante do software deve garantir um bom ambiente de trabalho (*peopleware*); em contrapartida, a equipe deve entregar um sistema que agregue valor ao seu negócio (*economicidade*). Mais um exemplo: ao escrever testes um desenvolvedor se beneficia, pois eles ajudam a detectar bugs no seu código; mas testes também ajudam outros desenvolvedores, que futuramente terão mais segurança de que o seu código não vai introduzir regressões — isto é, bugs — em código que está funcionando. Um terceiro e último exemplo: refactoring é uma atividade que torna o código mais limpo e fácil de entender, tanto para quem o escreveu, como para quem futuramente terá que mantê-lo. A frase todo negócio tem que ser bom para os dois lados resume bem esse terceiro princípio de XP.

**Melhorias Contínuas** (no livro de XP, o nome original é *improvements*): Como expressa a frase de Kent Beck que abre este capítulo, nenhum processo de desenvolvimento de software é perfeito. Por isso, é mais seguro trabalhar com um sistema que vai sendo continuamente aprimorado, a cada iteração, com o feedback dos clientes e de todos os membros do time. Pelo mesmo motivo, XP não recomenda investir um grande montante de tempo em um design inicial e completo. Em vez disso, o design do sistema também é incremental, melhorando a cada iteração. Por fim, as próprias práticas de desenvolvimento podem ser aprimoradas; para isso, o time deve reservar tempo para refletir sobre elas.

**Falhas Acontecem.** Desenvolvimento de software não é uma atividade livre de riscos. Como discutido no Capítulo 1, software é uma das mais complexas construções humanas. Logo, falhas são esperadas em projetos de desenvolvimento de software. No contexto desse princípio, falhas incluem bugs, funcionalidades que não se mostraram interessantes para os usuários finais e requisitos não-funcionais que não estão sendo plenamente atendidos, como desempenho, usabilidade, privacidade, disponibilidade, etc. Evidentemente, XP não advoga que essas falhas devem ser acobertadas. Porém, elas não devem ser usadas para punir membros de um time. Pelo contrário, falhas fazem parte do jogo, se um time pretende entregar software com rapidez.

**Baby Steps.** É melhor um progresso seguro, testado e validado, mesmo que pequeno, do que grandes implementações com riscos de serem descartadas pelos usuários. O mesmo vale para testes (que são úteis mesmo quando as unidades testadas são de menor granularidade), integração de código (é melhor integrar diariamente, do que passar pelo stress de fazer uma grande integração após semanas de trabalho) e refatorações (que devem ocorrer em pequenos passos, quando é mais fácil verificar que o comportamento do sistema está sendo preservado). Em resumo, o importante é garantir melhorias contínuas, não importando que sejam pequenas, desde que na direção correta. Essas pequenas melhorias são melhores do que grandes revoluções, as quais costumam não apresentar resultados positivos, pelo menos quando se trata de desenvolvimento de software.

**Responsabilidade Pessoal** (que usamos como tradução para *accepted responsibility*). De acordo com esse princípio, desenvolvedores devem ter

uma ideia clara de seu papel e responsabilidade na equipe. O motivo é que responsabilidade não pode ser transferida, sem que a outra parte a aceite. Por isso, XP defende que o engenheiro de software que implementa uma *história* — termo que o método usa para requisitos — deve ser também aquele que vai testá-la e mantê-la.

**Mundo Real:** Um dos primeiros sistemas a adotar XP foi um sistema de folha de pagamentos da fabricante de automóveis Chrysler, chamado Chrysler Comprehensive Compensation (C3) ([link](#)). O projeto desse sistema começou no início de 1995 e, como não apresentou resultados concretos, ele foi reiniciado no ano seguinte, sob a liderança de Kent Beck. Outro membro conhecido da comunidade ágil, Martin Fowler, participou do projeto, como consultor. No desenvolvimento do sistema C3, foram usadas e testadas diversas ideias do método que poucos anos depois receberia o nome de XP.

## Práticas sobre o Processo de Desenvolvimento

XP — como outros métodos ágeis — recomenda o envolvimento dos clientes com o projeto. Ou seja, além de desenvolvedores, os times incluem pelo menos um **representante dos clientes**, que deve entender do domínio do sistema que será construído. Uma das funções desse representante é escrever as **histórias de usuário** (*user stories*), que é o nome que XP dá para os documentos que descrevem os requisitos do sistema a ser implementado. No entanto, histórias são documentos resumidos, com apenas duas ou três sentenças, com as quais o representante dos clientes define o que ele deseja que o sistema faça, usando sua própria linguagem.

Iremos aprofundar o estudo sobre histórias de usuários no Capítulo 3. Mas, por enquanto, gostaríamos de adiantar que as histórias são escritas em cartões de papel, normalmente a mão. Ou seja, em vez de documentos de requisitos detalhados, histórias são documentos simples, que focam nas funcionalidades do sistema, sempre na visão de seus usuários. Como exemplo, mostramos a seguir uma história de um sistema de perguntas e respostas — semelhante ao famoso Stack Overflow — que usaremos neste capítulo para explicar XP.

### Postar Pergunta

*Um usuário, quando logado no sistema, deve ser capaz de postar*

*perguntas. Como é um site sobre programação, as perguntas podem incluir blocos de código, os quais devem ser apresentados com um leiaute diferenciado.*

Observe que a história tem um título (Postar Pergunta) e uma breve descrição, que não ocupa mais do que duas ou três sentenças. Costuma-se dizer que histórias são um lembrete para que depois esse requisito seja verbalmente detalhado pelo representante dos clientes.

Depois de escritas pelo representante dos clientes, as histórias são estimadas pelos desenvolvedores. Ou seja, são os desenvolvedores que definem, mesmo que preliminarmente, quanto tempo será necessário para implementar as histórias escritas pelo representante dos clientes. Frequentemente, a duração de uma história é estimada em **story points**, em vez de horas ou homens/hora. Nesses casos, usa-se uma escala inteira para classificar histórias como possuindo um certo número de story points. O objetivo é definir uma ordem relativa entre as histórias. As histórias mais simples são estimadas como tendo tamanho igual a 1 story point; histórias que são cerca de duas vezes mais complexas do que as primeiras são estimadas como tendo 2 story points e assim por diante. Muitas vezes, usa-se uma sequência de Fibonacci para definir a escala de possíveis story points, como em 1, 2, 3, 5, 8, 13 story points. Nesse caso, o objetivo é criar uma escala que torne as tarefas progressivamente mais difíceis e, ao mesmo tempo, permita ao time realizar comparações similares à seguinte: será que o esforço para implementar essa tarefa que planejamos estimar com 8 story points é equivalente ao esforço de implementar uma tarefa na escala anterior (5 story points) e mais uma tarefa na próxima escala inferior (3 pontos)? Se sim, 8 story points é uma boa estimativa. Senão, o melhor é estimar a história com 5 story points.

**Aprofundamento:** Uma técnica usada para estimar o tamanho de histórias é conhecida como **Planning Poker**. Ela funciona assim: o representante dos clientes seleciona uma história e a lê para os desenvolvedores. Após a leitura, os desenvolvedores interagem com o representante dos clientes para tirar possíveis dúvidas e conhecer melhor a história. Feito isso, cada desenvolvedor faz sua estimativa para o tamanho da história, de forma independente. Depois disso, eles ao mesmo tempo levantam cartões com a estimativa que pensaram, em story points. Esses cartões foram distribuídos

antes e neles constam os números 1, 2, 3, 5, etc. Se houver consenso, o tamanho da história está estimado e passa-se para a próxima história. Senão, o time deve iniciar uma discussão, para esclarecer a razão das diferentes estimativas. Por exemplo, os desenvolvedores responsáveis pelas estimativas mais discrepantes podem explicar o motivo da sua proposta. Feito isso, realiza-se uma nova votação e o processo se repete, até que o consenso seja alcançado.

A implementação das histórias ocorre em **iterações**, as quais têm uma duração fixa e bem definida, variando de uma a três semanas, por exemplo. As iterações, por sua vez, formam ciclos mais longos, chamados de **releases**, de dois a três meses, por exemplo. A **velocidade** de um time é o número de story points que ele consegue implementar em uma iteração. Sugere-se que o representante dos clientes escreva histórias que requeiram pelo menos uma release para serem implementadas. Ou seja, em XP, o horizonte de planejamento é uma release, isto é, alguns meses.

**Aviso:** Em XP, a palavra release tem um sentido diferente daquele que se usa em gerência de configuração. Em gerência de configuração, uma release é uma versão de um sistema que será disponibilizada para seus usuários finais. Como já mencionamos em um aviso anterior, não necessariamente a versão do sistema ao final de uma release de XP precisa entrar em produção.

Em resumo, para começar a usar XP precisamos de:

- Definir a duração de uma iteração.
- Definir o número de iterações de uma release.
- Um conjunto de histórias, escritas pelo representante dos clientes.
- Estimativas para cada história, feitas pelos desenvolvedores.
- Definir a velocidade do time, isto é, o número de story points que ele consegue implementar por iteração.

Uma vez definidos os parâmetros e documentos acima, o representante do cliente deve priorizar as histórias. Para isso, ele deve definir quais histórias

serão implementadas nas iterações da primeira release. Nessa priorização, deve-se respeitar a velocidade do time de desenvolvimento. Por exemplo, suponha que a velocidade de um time seja de 25 story points por iteração. Nesse caso, o representante do cliente não pode alocar histórias para uma iteração cujo somatório de story points ultrapasse esse limite. A tarefa de alocar histórias a iterações e releases é chamada de **planejamento de releases** (ou então *planning game*, que foi o nome adotado na primeira edição do livro de XP).

Por exemplo, suponha o fórum de perguntas e respostas que mencionamos antes. A próxima tabela resume o resultado de um possível planejamento de releases. Nessa tabela, estamos assumindo que o representante dos clientes escreveu 8 histórias, que cada release possui duas iterações e que a velocidade do time é de 21 story points por iteração (veja que o somatório dos story points de cada iteração é exatamente igual a 21).

História	Story Points	Iteração	Release
Cadastrar usuário	8	1	1
Postar perguntas	5	1	1
Postar respostas	3	1	1
Tela de abertura	5	1	1
Gamificar perguntas/respostas	5	2	1
Pesquisar perguntas/respostas	8	2	1
Adicionar tags	5	2	1
Comentar perguntas/respostas	3	2	1

A tabela anterior serve para reforçar dois pontos já mencionados: (1) as histórias em XP representam funcionalidades do sistema que se pretende construir; isto é, a implementação do sistema é dirigida por suas funcionalidades; (2) os desenvolvedores não opinam sobre a ordem de implementação das histórias; isso é decidido pelo representante dos clientes, que deve ser alguém capacitado e com autoridade para definir o que é mais urgente e importante para a empresa que está contratando o desenvolvimento do sistema.

Uma vez realizado o planejamento de uma release, começam as iterações. Antes de mais nada, o time de desenvolvimento deve se reunir para realizar o **planejamento da iteração**. O objetivo desse planejamento é decompor as histórias de uma iteração em tarefas, as quais devem corresponder a atividades de programação que possam ser alocadas para um dos desenvolvedores do time. Por exemplo, a seguinte lista mostra as tarefas para a história Postar Perguntas, que é a primeira história que será implementada em nosso sistema de exemplo.

- Projetar e testar a interface Web, incluindo leiaute, CSS templates, etc.
- Instalar banco de dados, projetar e criar tabelas.
- Implementar a camada de acesso a dados.
- Instalar servidor e testar framework web.
- Implementar camada de controle, com operações para cadastrar, remover e atualizar perguntas.
- Implementar interface Web.

Como regra geral, as tarefas não devem ser complexas, devendo ser possível concluir-las em alguns dias.

Resumindo, um projeto XP é organizado em:

- releases, que são conjunto de iterações, com duração total de alguns meses.
- iterações, que são conjuntos de tarefas resultantes da decomposição de histórias, com duração total de algumas semanas.
- tarefas, com duração de alguns dias.

Definidas as tarefas, o time deve decidir qual desenvolvedor será responsável por cada uma. Feito isso, começa de fato a iteração, com a

implementação das tarefas.

Uma iteração termina quando todas as suas histórias estiverem implementadas e validadas pelo representante dos clientes. Assim, ao fim de uma iteração, as histórias devem ser mostradas para o representante dos clientes, que deve concordar que elas, de fato, atendem ao que ele especificou.

XP defende ainda que os times, durante uma iteração, programem algumas **folgas** (*slacks*), que são tarefas que podem ser adiadas, caso necessário. Como exemplos, podemos citar o estudo de uma nova tecnologia, a realização de um curso online, preparar uma documentação ou manual ou mesmo desenvolver um projeto paralelo. Algumas empresas, como o Google, por exemplo, são famosas por permitir que seus desenvolvedores usem 20% de seu tempo para desenvolver um projeto pessoal ([link](#)). No caso de XP, folgas têm dois objetivos principais: (1) criar um buffer de segurança em uma iteração, que possa ser usado caso alguma tarefa demande mais tempo do que o previsto; (2) permitir que os desenvolvedores respirem um pouco, pois o ritmo de trabalho em projetos de desenvolvimento de software costuma ser intenso e desgastante. Logo, os desenvolvedores precisam de um tempo para realizarem algumas tarefas sem que exista uma cobrança imediata de resultados.

## Perguntas Frequentes

Vamos agora responder algumas perguntas sobre as práticas de XP que acabamos de explicar.

**Qual a duração ideal de uma iteração?** Difícil precisar, pois depende das características do time, da empresa contratante, da complexidade do sistema a ser desenvolvido, etc. Iterações curtas — por exemplo, de uma semana — propiciam feedback mais rápido. Porém, requerem um maior comprometimento dos clientes, pois toda semana um novo incremento de produto deve ser validado. Além disso, requerem que as histórias sejam mais simples. Por outro lado, iterações mais longas — por exemplo, de um mês — permitem que o time planeje e conclua as tarefas com mais tranquilidade. Porém, demora-se um pouco mais para receber feedback dos clientes. Esse feedback pode ser importante quando os requisitos são pouco claros. Por

isso, uma escolha de compromisso seria algo como 2 ou 3 semanas. Outra alternativa recomendada consiste em experimentar, isto é, testar e avaliar diferentes durações, antes de decidir.

**O que o representante dos clientes faz durante as iterações?** No início de uma release, cabe ao representante dos clientes escrever as histórias das iterações que farão parte dessa release. Depois, no final de cada iteração, cabe a ele validar e aprovar a implementação das histórias. Porém, durante as iterações, ele deve estar fisicamente disponível para tirar dúvidas do time. Veja que uma história é um documento muito resumido, logo é natural que surjam dúvidas durante a sua implementação. Por isso, o representante dos clientes deve estar sempre disponível para se reunir com os desenvolvedores, para tirar dúvidas e explicar detalhes relativos à implementação de histórias.

**Como escolher o representante dos clientes?** Antes de mais nada, deve ser alguém que conheça o domínio do sistema e que tenha autoridade para priorizar histórias. Conforme detalhado a seguir, existem pelo menos três perfis de representante dos clientes:

- Suponha o desenvolvimento interno de um sistema, isto é, o departamento de sistemas da empresa X está desenvolvendo um sistema para um outro departamento Y, da mesma empresa. Nesse caso, o representante dos clientes deve ser um funcionário do departamento Y.
- Suponha que o time de desenvolvimento foi contratado para desenvolver um sistema para a empresa X. Ou seja, trata-se de um desenvolvimento terceirizado. Nesse caso, o representante do cliente deve ser um funcionário da empresa X, com pleno domínio da área do sistema e que vai ser um dos seus principais usuários, quando ele ficar pronto.
- Suponha que o time de desenvolvimento de uma empresa X foi designado para fazer um sistema para um público externo à empresa. Por exemplo, um sistema como aquele usado neste capítulo, similar ao Stack Overflow. Logo, os clientes do sistema não são funcionários de X, mas sim clientes externos. Nesse caso, o representante dos clientes deve ser alguém da área de marketing, vendas ou negócios da empresa

X. Em última instância, pode ser o dono da empresa. Em qualquer caso, a sugestão é que seja uma pessoa próxima do problema e o mais distante possível da solução. Por isso mesmo, deve-se evitar que ele seja um desenvolvedor ou um gerente de projeto. O tipo de representante dos clientes que mencionamos neste item é, às vezes, chamado de um **user proxy**.

**Como definir a velocidade do time?** Não existe bala de prata para essa questão. Essa definição depende da experiência do time e de seus membros. Se eles já participaram de projetos semelhantes àquele que estão iniciando, certamente essa deve ser uma questão menos difícil. Caso contrário, tem-se que experimentar e ir calibrando a velocidade nas iterações seguintes.

**Histórias podem incluir tarefas de instalação de infraestrutura de software?** Não, histórias são especificadas pelo representante dos clientes, que é um profissional leigo em Engenharia de Software. Portanto, ele não costuma ter conhecimento de infraestrutura de software. No entanto, uma história pode dar origem a uma tarefa como instalar e testar o banco de dados. Resumindo, histórias estão associadas a requisitos funcionais; para implementá-las criam-se tarefas, que podem estar associadas a requisitos funcionais, não-funcionais ou tarefas técnicas, como instalação de bancos de dados, servidores, frameworks, etc.

**A história X depende da história Y, mas o representante dos clientes priorizou X antes de Y. O que devo fazer?** Por exemplo, suponha que no sistema de exemplo o representante dos clientes tenha alocado a história Postar Pergunta para a iteração 2 e a história Postar Resposta para a iteração 1. A pergunta então é a seguinte: o time deve respeitar essa alocação? Sim, pois a regra é clara: o representante dos clientes é a autoridade final quando se trata de definir a ordem de implementação das histórias. Logo, pode-se perguntar em seguida: como que vamos postar respostas, sem ter as perguntas? Para isso, basta implementar algumas perguntas fixas, que não possam ser modificadas pelos usuários. Na iteração 1, quando o cliente abrir o sistema, essas perguntas vão aparecer por default, talvez com um layout bem simples, e então o cliente vai poder usar o sistema apenas para responder essas perguntas fixas.

**Quando um projeto XP termina?** Quando o representante dos clientes decide que as histórias já implementadas são suficientes e que não há mais nada de relevante que deva ser implementado.

## Práticas de Programação

O nome Extreme Programming foi escolhido porque XP propõe um conjunto de práticas de programação inovadoras, principalmente para a época na qual foram propostas, no final da década de 90. Na verdade, XP é um método que dá grande importância a tarefas de programação e produção de código. Essa importância tem que ser entendida no contexto da época, quando havia uma diferença entre analistas e programadores. Analistas eram encarregados de elaborar o projeto em alto nível de um sistema, definindo seus principais componentes, classes e interfaces. Para isso, recomendava-se o uso de uma linguagem de modelagem gráfica, como UML, que veremos no Capítulo 4 deste livro. Concluída a fase de análise e projeto, começava a fase de codificação, que ficava a cargo dos programadores. Assim, na prática, existia uma hierarquia nesses papéis, sendo o papel de analista o de maior prestígio. Métodos ágeis — e, particularmente, XP — acabaram com essa hierarquia e passaram a defender a produção de código com funcionalidades, logo nas primeiras semanas de um projeto.

Mas XP não apenas acabou com a grande fase de projeto e análise, logo no início dos projetos. O método também propôs um novo conjunto de práticas de programação, incluindo programação em pares, testes automatizados, desenvolvimento dirigido por testes (TDD), builds automatizados, integração contínua, etc. A maioria dessas práticas passou a ser largamente adotada pela indústria de software e hoje são praticamente obrigatórias na maioria dos projetos — mesmo naqueles que não usam um método ágil.

Nessa seção, vamos estudar as práticas de programação de XP.

**Design Incremental.** Como afirmado nos parágrafos anteriores, em XP não há uma fase de design e análise detalhados, conhecida como *Big Design Up Front* (BDUF), a qual é uma das principais fases de processos do tipo Waterfall. A ideia é que o time deve reservar tempo para definir o design do sistema que está sendo desenvolvido. Porém, isso deve ser uma atividade contínua e incremental, em vez de estar concentrada no início do projeto,

antes de qualquer codificação. No caso, simplicidade é o valor de XP que norteia essa opção por um design também incremental. Argumenta-se que quando o design é confinado no início do projeto, correm-se diversos riscos, pois os requisitos ainda não estão totalmente claros para o time, e nem mesmo para o representante dos clientes. Por exemplo, pode-se supervalorizar alguns requisitos, que mais tarde irão se revelar menos importantes; de forma inversa, pode-se subvalorizar outros requisitos, que depois, com o decorrer da implementação, irão assumir um maior protagonismo. Isso sem falar que novos requisitos podem surgir ao longo do projeto, tornando o design inicial desatualizado e menos eficiente.

Por isso, XP defende que o momento ideal para pensar em *design* é quando ele se revelar importante. Frequentemente, duas frases são usadas para motivar e justificar essa prática: faça a coisa mais simples que possa funcionar (*do the simplest thing that could possibly work*) e você não vai precisar disso (*you aren't going to need it*), essa última conhecida pela sigla YAGNI.

Duas observações são importantes para melhor entender a proposta de design incremental. Primeiro, times experientes costumam ter uma boa aproximação do design logo na primeira iteração. Por exemplo, eles já sabem que se trata de um sistema com interface Web, com uma camada de lógica não trivial, mas também não tão complexa, e depois com uma camada de persistência e um banco de dados, certamente relacional. Ou seja, apenas a sentença anterior já define muito do design que deve ser adotado. Como uma segunda observação, nada impede que na primeira iteração o time crie uma tarefa técnica para discutir e refinar o design que vão começar a adotar no sistema.

Por fim, design incremental somente é possível caso seja adotado em conjunto com as demais práticas de XP, principalmente **refactoring**. XP defende que refactoring deve ser continuamente aplicado para melhorar a qualidade do design. Por isso, toda oportunidade de refatoração, visando facilitar o entendimento e a evolução do código, não pode ser deixada para depois.

**Programação em Pares.** Junto com design incremental, programação em pares é uma das práticas mais polêmicas de XP. Apesar de polêmica, a ideia

é simples: toda tarefa de codificação — incluindo implementação de uma nova história, de um teste, ou a correção de um bug — deve ser realizada por dois desenvolvedores trabalhando juntos, compartilhando o mesmo teclado e monitor. Um dos desenvolvedores é o **líder** (ou *driver*) da sessão, ficando com o teclado e o mouse. Ao segundo desenvolvedor cabe a função de revisor e questionador, no bom sentido, do trabalho do líder. Esse segundo desenvolvedor é chamado de **navegador**. O nome vem dos ralis automobilísticos, nos quais os pilotos são acompanhados de um navegador.

Com programação em pares espera-se melhorar a qualidade do código e do design, pois duas cabeças pensam melhor do que uma. Além disso, programação em pares contribui para disseminar o conhecimento sobre o código, que não fica nas mãos e na cabeça de apenas um desenvolvedor. Por exemplo, não é raro encontrar sistemas nos quais um determinado desenvolvedor tem dificuldade para sair de férias, pois apenas ele conhece uma parte crítica do código. Como uma terceira vantagem, programação por pares pode ser usada para treinar desenvolvedores menos experientes em tecnologias de desenvolvimento, algoritmos e estruturas de dados, padrões e princípios de design, escrita de testes, técnicas de depuração, etc.

Por outro lado, existem também custos econômicos derivados da adoção de programação em pares, já que dois programadores estão sendo alocados para realizar uma tarefa que, a princípio, poderia ser realizada por apenas um. Além disso, muitos desenvolvedores não se sentem confortáveis com a prática. Para eles é desconfortável — do ponto de vista emocional e cognitivo — discutir cada linha de código e cada decisão de implementação com um colega. Para aliviar esse desconforto, XP propõe que os pares sejam trocados a cada sessão. Elas podem durar, por exemplo, 50 minutos, seguidos de uma pausa de 10 minutos para descanso. Na próxima sessão, os pares e papéis (líder vs revisor) são trocados. Assim, se em uma sessão você atuou como revisor do programador X, na sessão seguinte você passará a ser o líder, mas tendo outro desenvolvedor Y como revisor.

**Mundo Real:** Em 2008, dois pesquisadores da Microsoft Research, Andrew Begel e Nachiappan Nagappan, realizaram um survey com 106 desenvolvedores da empresa, para capturar a percepção deles sobre programação em pares ([link](#)). Quase 65% dos desenvolvedores responderam positivamente a uma primeira pergunta sobre se programação em pares

estaria funcionando bem para eles (*pair programming is working well for me*). Quando perguntados sobre os benefícios de programação em pares, as respostas foram as seguintes: redução no número de bugs (62%), produção de código de melhor qualidade (45%), disseminação de conhecimento sobre o código (40%) e oportunidade de aprendizado com os pares (40%). Por outro lado, os custos da prática foram apontados como sendo seu principal problema (75%). Sobre as características do par ideal, a resposta mais comum foi complementaridade de habilidades (38%). Ou seja, desenvolvedores preferem parear com uma pessoa que o ajude a superar seus pontos fracos.

Mais recentemente, diversas empresas passaram a adotar a prática de **revisão de código**. A ideia é que todo código produzido por um desenvolvedor tem que ser revisado e comentado por um outro desenvolvedor, porém de forma *offline* e assíncrona. Ou seja, nesses casos, o revisor não estará fisicamente ao lado do líder.

**Propriedade Coletiva do Código.** A ideia é que qualquer desenvolvedor — ou par de desenvolvedores trabalhando junto — pode modificar qualquer parte do código, seja para implementar uma nova feature, para corrigir um bug ou para aplicar um refactoring. Por exemplo, se você descobriu um bug em algum ponto do código, vá em frente e corrija-o. Para isso, você não precisa de autorização de quem implementou esse código ou de quem realizou a última manutenção nele.

**Testes Automatizados.** Essa é uma das práticas de XP que alcançou maior sucesso. A ideia é que testes manuais — um ser humano executando o programa, fornecendo entradas e checando as saídas produzidas — é um procedimento custoso e que não pode ser reproduzido a todo momento. Logo, XP propõe a implementação de programas — chamados de testes — para executar pequenas unidades de um sistema, como métodos, e verificar se as saídas produzidas são aquelas esperadas. Essa prática prosperou porque, mais ou menos na mesma época de sua proposição, foram desenvolvidos os primeiros frameworks de testes de unidade — como o JUnit, cuja primeira versão, implementada por Kent Beck e Erich Gamma, é de 1997. No momento, não iremos estender a discussão sobre testes automatizados porque temos um capítulo exclusivo para isso no livro (Capítulo 8).

**Desenvolvimento Dirigido por Testes (TDD).** Essa é outra prática de programação inovadora proposta por XP. A ideia é simples: se em XP todo método deve possuir testes, por que não escrevê-los primeiro? Isto é, implementa-se o teste de um método e, só então, o seu código. TDD, que também é conhecido como *test-first programming*, possui duas motivações principais: (1) evitar que a escrita de testes seja sempre deixada para amanhã, pois eles são a primeira coisa que se deve implementar; (2) ao escrever um teste, o desenvolvedor se coloca no papel de cliente do método testado, isto é, ele primeiro pensa na sua interface, em como os clientes devem usá-lo, para então pensar na implementação. Com isso, incentiva-se a criação de métodos mais amigáveis, do ponto de vista da interface provida para os clientes. Não iremos alongar a explicação sobre TDD, pois dedicaremos uma seção inteira para essa prática no capítulo de Testes.

**Build Automatizado.** Build é o nome que se dá para a geração de uma versão de um sistema que seja executável e que possa ser colocada em produção. Logo, inclui não apenas a compilação do código, mas a execução de outras ferramentas como linkeditores e empacotadores de código em arquivos WAR, JAR, etc. No caso de XP, a execução dos testes é outra etapa fundamental do processo de build. Para automatizar esse processo, são usadas ferramentas, como o sistema Make, que faz parte das distribuições do sistema operacional Unix desde a década de 1970. Mais recentemente, surgiram outras ferramentas de build, como Ant, Maven, Gradle, Rake, MSBuild, etc. Primeiro, XP defende que o processo de build seja automatizado, sem nenhuma intervenção dos desenvolvedores. O objetivo é liberá-los das tarefas de rodar scripts, informar parâmetros de linhas de comando, configurar ferramentas, etc. Assim, eles podem focar apenas na implementação de histórias. Segundo, XP defende que o processo de build seja o mais rápido possível, para que os desenvolvedores recebam rapidamente feedback sobre possíveis problemas, como um erro de compilação. Na segunda versão do livro de XP, recomenda-se um limite de 10 minutos para a conclusão de um build. No entanto, dependendo do tamanho do sistema e de sua linguagem de programação, pode ser difícil atender a esse limite. Por isso, o mais importante é focar na regra geral: sempre procurar automatizar e reduzir o tempo de build de um sistema.

**Integração Contínua.** Sistemas de software são desenvolvidos com o apoio de sistemas de controle de versões (VCS, ou *Version Control System*), que

armazenam o código fonte do sistema e de arquivos relacionados, como arquivos de configuração, documentação, etc. Hoje o sistema de controle de versão mais usado é o git, por exemplo. Quando se usa um VCS, desenvolvedores têm que primeiro baixar (*pull*) o código fonte para sua máquina local, antes de começar a trabalhar em uma tarefa. Feito isso, eles devem subir o código modificado (*push*). Esse último passo é chamado de **integração** da modificação no código principal, armazenado no VCS. Porém, entre um *pull* e um *push*, outro desenvolvedor pode ter modificado o mesmo trecho de código e realizado a sua integração. Nesse caso, quando o primeiro desenvolvedor tentar subir com seu código, o sistema de controle de versão irá impedir a integração, dizendo que existe um **conflito**.

Conflitos são ruins, porque eles devem ser resolvidos manualmente. Se um desenvolvedor A modificou a inicialização de uma variável *x* com o valor 10; e outro desenvolvedor B, trabalhando em paralelo com A, gostaria de inicializar *x* com 20, isso representa um conflito. Para resolvê-lo, A e B devem sentar e discutir qual o melhor valor para inicializar *x*. No entanto, esse é um cenário simples. Conflitos podem ser mais complexos, envolver grandes trechos de código e mais do que dois desenvolvedores. Por isso, a resolução de conflitos de integração costuma demandar um grande esforço, resultando no que se chama de **integration hell**.

Por outro lado, evitar completamente conflitos é impossível. Por exemplo, não é possível conciliar automaticamente os interesses de dois desenvolvedores quando um deles precisa inicializar *x* com 10 e outro com 20. Porém, pode-se pelo menos tentar diminuir o número e o tamanho dos conflitos. Para conseguir isso, a ideia de XP é simples: desenvolvedores devem integrar seu código sempre, se possível todos os dias. Essa prática é chamada de **integração contínua**. O objetivo é evitar que desenvolvedores passem muito tempo trabalhando localmente, em suas tarefas, sem integrar o código. E, com isso, pelo menos diminuir as chances e o tamanho dos conflitos.

Para garantir a qualidade do código que está sendo integrado com frequência quase diária, costuma-se usar também um **serviço de integração contínua**. Antes de realizar qualquer integração, esse serviço faz o build do código e executa os testes. O objetivo é garantir que o código não possui erros de compilação e que ele passa em todos os testes. Existem diversos serviços de

integração contínua, como Jenkins, TravisCI, CircleCI, etc. Por exemplo, quando se desenvolve um sistema usando o GitHub, pode-se ativar esses serviços na forma de plugins. Se o repositório GitHub for público, o serviço de integração contínua é gratuito; se for privado, deve-se pagar uma assinatura.

Iremos estudar mais sobre Integração Contínua no capítulo sobre DevOps.

**Mundo Real:** Em 2010, Laurie Williams, professora da Universidade da Carolina do Norte, nos EUA, pediu que 326 desenvolvedores respondessem a um questionário sobre a experiência deles com métodos ágeis ([link](#)). Em uma das questões, pedia-se aos participantes para ranquear a importância de práticas ágeis, usando uma escala de 1 a 5, na qual o score 5 deveria ser dado apenas para práticas essenciais em desenvolvimento ágil. Três práticas ficaram empatadas em primeiro lugar, com score médio 4.5 e desvio padrão de 0.8. São elas: integração contínua, iterações curtas (menos de 30 dias) e definição de critérios para tarefas concluídas (*done criteria*). Por outro lado, dentre as práticas nas últimas posições podemos citar planning poker (score médio 3.1) e programação em pares (score médio 3.3).

## Práticas de Gerenciamento de Projetos

**Ambiente de Trabalho.** XP defende que o projeto seja desenvolvido com um time pequeno, com menos de 10 desenvolvedores, por exemplo. Todos eles devem estar dedicados ao projeto. Isto é, deve-se evitar times fracionados, nos quais alguns desenvolvedores trabalham apenas alguns dias da semana no projeto e os outros dias em um outro projeto, por exemplo.

Além disso, XP defende que todos os desenvolvedores trabalhem em uma mesma sala, para facilitar comunicação e feedback. Também propõe que o espaço de trabalho seja informativo, isto é, que sejam, por exemplo, fixados cartazes nas paredes, com as histórias da iteração, incluindo seu estado: histórias pendentes, histórias em andamento e histórias concluídas. A ideia é permitir que o time possa visualizar o trabalho que está sendo realizado.

Outra preocupação de XP é garantir jornadas de trabalho sustentáveis. Empresas de desenvolvimento de software são conhecidas por exigirem longas jornadas de trabalho, com diversas horas extras e trabalho nos finais

de semana. XP defende que essa prática não é sustentável e que as jornadas de trabalho devem ser sempre próximas de 40 horas, mesmo na véspera de entregas. O interessante é que XP é um método proposto por desenvolvedores, com grande experiência em projetos reais de desenvolvimento de software. Logo, eles devem ter sentido na própria pele os efeitos de longas jornadas de trabalho. Dentre outros problemas, elas podem causar danos à saúde física e mental dos desenvolvedores, assim como incentivar a rotatividade do time, cujos membros vão estar sempre pensando em um novo emprego.

**Contratos com Escopo Aberto.** Quando uma empresa terceiriza o desenvolvimento, existem duas possibilidades de contrato: com escopo fechado ou com escopo aberto. Em contratos com escopo fechado, a empresa contratante define, mesmo que de forma mínima, os requisitos do sistema e a empresa contratada define um preço e um prazo de entrega. XP advoga que esses contratos são arriscados, pois os requisitos mudam e nem mesmo o cliente sabe antecipadamente o que ele quer que o sistema faça, de modo preciso. Assim, contratos com escopo fixo podem fazer com que a contratada entregue um sistema com problemas de qualidade e mesmo com alguns requisitos implementados de modo parcial ou com bugs, apenas para não ter que pagar possíveis multas. Por outro lado, quando o escopo é aberto, o pagamento ocorre por hora trabalhada. Por exemplo, combina-se que a contratada vai alojar um time com um certo número de desenvolvedores para trabalhar integralmente no projeto, usando as práticas de XP. Combina-se também um preço para a hora de cada desenvolvedor. O cliente define as histórias e faz a validação delas ao final de cada iteração. O contrato pode ser rescindido ou renovado após um certo número de meses, o que dá ao cliente a liberdade de mudar de empresa caso não esteja satisfeito com a qualidade do serviço prestado. Como usual em XP, o objetivo é abrir um fluxo de comunicação e feedback entre contratada e contratante, em vez de forçar a primeira a entregar um produto com problemas conhecidos, apenas para cumprir um contrato. Na verdade, contratos com escopo aberto são mais compatíveis com os princípios do Manifesto Ágil, que explicitamente valoriza colaboração com o cliente, mais do que negociação de contratos.

**Métricas de Processo.** Para que gerentes e executivos possam acompanhar um projeto XP recomenda-se o uso de duas métricas principais: número de bugs em produção (que deve ser idealmente da ordem de poucos bugs por

ano) e intervalo de tempo entre o início do desenvolvimento e o momento em que o projeto começar a gerar os seus primeiros resultados financeiros (que também deve ser pequeno, da ordem de um ano, por exemplo).

## Scrum

Scrum é um método ágil, iterativo e incremental para gerenciamento de projetos. Foi proposto por Jeffrey Sutherland e Ken Schwaber, em um artigo publicado pela primeira vez em 1995 ([link](#)). Dentre os métodos ágeis, Scrum é o mais conhecido e usado. Provavelmente, parte do sucesso do método seja explicada pela existência de uma indústria associada à sua adoção, a qual inclui a produção de livros, diversos cursos, consultorias e certificações.

Uma pergunta que vamos responder logo no início desta seção diz respeito às diferenças entre Scrum e XP. Existem diversas pequenas diferenças, mas a principal delas é a seguinte:

- XP é um método ágil voltado exclusivamente para projetos de desenvolvimento de software. Para isso, XP inclui um conjunto de práticas de programação, como testes de unidade, programação em pares, integração contínua e design incremental, que foram estudadas na seção anterior, dedicada a XP.
- Scrum é um método ágil para gerenciamento de projetos, que não necessariamente precisam ser projetos de desenvolvimento de software. Por exemplo, a escrita deste livro — como comentaremos daqui a pouco — é um projeto que está sendo realizado usando conceitos de Scrum. Tendo um foco mais amplo que XP, Scrum não propõe nenhuma prática de programação.

Dentre os métodos ágeis, Scrum é também aquele que é melhor definido. Essa definição inclui um conjunto preciso de **papéis**, **artefatos** e **eventos**, que são listados a seguir. No resto desta seção, vamos explicar cada um deles.

- **Papéis:** Dono do Produto, Scrum Master, Desenvolvedor.

- **Artefatos:** Backlog do Produto, Backlog do Sprint, Quadro Scrum, Gráfico de Burndown.
- **Eventos:** Planejamento do Sprint, Sprint, Reuniões Diárias, Revisão do Sprint, Retrospectiva.

## Papéis

Times Scrum são formados por um Dono de Produto (*Product Owner* ou apenas PO), um Scrum Master e de três a nove desenvolvedores.

O **Dono do Produto** tem exatamente o mesmo papel do Representante dos Clientes em XP, por isso não vamos explicar de novo a sua função em detalhes. Mas ele, como o próprio nome indica, deve possuir a visão do produto que será construído, sendo responsável também por maximizar o retorno do investimento feito no projeto. Como em XP, cabe ao Dono do Produto escrever as histórias dos usuários e, por isso, ele deve estar sempre disponível para tirar dúvidas do time.

O **Scrum Master** é um papel característico e único de Scrum. Trata-se do especialista em Scrum do time, sendo responsável por garantir que as regras do método estão sendo seguidas. Para isso, ele deve continuamente treinar e explicar os princípios de Scrum para os demais membros do time. Ele também deve desempenhar funções de um facilitador dos trabalhos e removedor de impedimentos. Por exemplo, suponha que um time esteja enfrentando problemas com um dos servidores de bancos de dados, cujos discos estão apresentando problemas todos os dias. Cabe ao Scrum Master intervir junto aos níveis adequados da empresa para garantir que esse problema de hardware não atrapalhe o avanço do time. Por outro lado, ele não é um gerente de projeto tradicional. Por exemplo, ele não é o líder do time, pois todos em um time Scrum têm o mesmo nível hierárquico.

Costuma-se dizer que times Scrum são **cross-funcionais** (ou multidisciplinares), isto é, eles devem incluir — além do Dono do Produto e do Scrum Master — todos os especialistas necessários para desenvolver o produto, de forma a não depender de membros externos. No caso de projetos de software, isso inclui desenvolvedores *front-end*, desenvolvedores *back-end*, especialistas em bancos de dados, projetistas de interfaces, etc. Cabe a

esses especialistas tomar todas as decisões técnicas do projeto, incluindo definição da linguagem de programação, arquitetura e frameworks que serão usados no desenvolvimento. Cabe a eles também estimar o tamanho das histórias definidas pelo Dono do Produto, usando uma unidade como story points, de modo semelhante ao que vimos em XP.

## Principais Artefatos e Eventos

Em Scrum, os dois artefatos principais são o Backlog do Produto e o Backlog do Sprint e os principais eventos são sprints e o planejamento de sprints, conforme descreveremos a seguir.

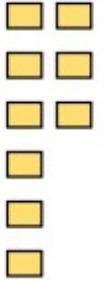
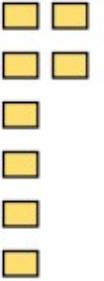
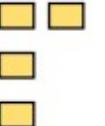
- O **Backlog do Produto** é uma lista de histórias, ordenada por prioridades. Assim como em XP, as histórias são escritas e priorizadas pelo Dono do Produto e constituem uma descrição resumida das funcionalidades que devem ser implementadas no projeto. É importante mencionar ainda que o Backlog do Produto é um artefato dinâmico, isto é, ele deve ser continuamente atualizado, de forma a refletir mudanças nos requisitos e na visão do produto. Por exemplo, à medida que o desenvolvimento avança, ideias de novas funcionalidades podem surgir, enquanto outras podem perder importância. Todas essas atualizações devem ser realizadas pelo Dono do Produto. Na verdade, é o fato de ser o dono do Backlog do Produto que faz o Dono do Produto receber esse nome.
- **Sprint** é o nome dado por Scrum para uma iteração. Ou seja, como todo método ágil, Scrum é um método iterativo, no qual o desenvolvimento é dividido em sprints, de até um mês. Ao final de um sprint, deve-se entregar um produto com valor tangível para o cliente. O resultado de um sprint é chamado de um produto potencialmente pronto para entrar em produção (*potentially shippable product*). Lembre que o adjetivo potencial não torna a entrada em produção obrigatória, conforme discutido na Seção 2.2.
- O **Planejamento do Sprint** é uma reunião na qual todo o time se reúne para decidir as histórias que serão implementadas no sprint que vai se iniciar. Portanto, ele é o evento que marca o início de um sprint. Essa

reunião é dividida em duas partes. A primeira é comandada pelo Dono do Produto. Ele propõe histórias para o sprint e o restante do time decide se tem **velocidade** para implementá-las. A segunda parte é comandada pelos desenvolvedores. Nela, eles quebram as histórias em tarefas e estimam a duração delas. No entanto, o Dono do Produto deve continuar presente nessa parte final, para tirar dúvidas sobre as histórias selecionadas para o sprint. Por exemplo, pode-se decidir cancelar uma história, pois ela se revelou mais complexa ao ser quebrada em tarefas.

- O **Backlog do Sprint** é o artefato gerado ao final do Planejamento do Sprint. Ele é uma lista com as tarefas do sprint, bem como inclui a duração das mesmas. Como o Backlog do Produto, o Backlog do Sprint também é dinâmico. Por exemplo, tarefas podem se mostrar desnecessárias e outras podem surgir, ao longo do sprint. Pode-se também alterar a estimativa de horas previstas para uma tarefa. Porém, o que não pode ser alterado é o **objetivo do sprint** (*sprint goal*), isto é, a lista de histórias que o dono do produto selecionou para o sprint e que o time de desenvolvimento se comprometeu a implementar na duração do mesmo. Assim, Scrum é um método adaptável a mudanças, mas desde que elas ocorram entre sprints. Ou seja, no time-box de um sprint, a equipe de desenvolvimento deve ter tranquilidade e segurança para trabalhar com uma lista fechada de histórias.

Terminada a reunião de planejamento, tem início o sprint. Ou seja, o time começa a trabalhar na implementação das tarefas do backlog. Além de cross-funcionais, os times Scrum são **auto-organizáveis**, isto é, eles têm autonomia para decidir como e por quem as histórias serão implementadas.

Ao lado do Backlog do Sprint, costuma-se anexar um quadro com tarefas *a fazer, em andamento e finalizadas*. Esse quadro — também chamado de **Quadro Scrum** (*Scrum Board*) — pode ser fixado nas paredes do ambiente de trabalho, permitindo que o time tenha diariamente uma sensação visual sobre o andamento do sprint. Veja um exemplo na próxima figura.

Backlog	To Do	Doing	Testing	Done
				

Exemplo de Quadro Scrum, mostrando as histórias selecionadas para o sprint e as tarefas nas quais elas foram quebradas. Cada tarefa nesse quadro pode estar em um dos seguintes estados: a fazer, em andamento, em teste ou concluída.

Uma decisão importante em projetos Scrum envolve os critérios para considerar uma história ou tarefa como concluídas (*done*). Esses critérios devem ser combinados com o time e ser do conhecimento de todos os seus membros. Por exemplo, em um projeto de desenvolvimento de software, para que uma história seja marcada como concluída pode-se exigir a implementação de testes de unidade, que devem estar todos passando, bem como a revisão do código por um outro membro do time. Além disso, o código deve ter sido integrado com sucesso no repositório do projeto. O objetivo desses critérios é evitar que os membros — de forma apressada e valendo-se de código de baixa qualidade — consigam mover suas tarefas para a coluna concluído.

Um outro artefato comum em Scrum é o **Gráfico de Burndown**. A cada dia do sprint, esse gráfico mostra quantas horas são necessárias para se implementar as tarefas que ainda não estão concluídas. Isto é, no dia X do sprint ele informa que restam tarefas a implementar que somam Y horas. Logo, a curva de um gráfico de burndown deve ser declinante, atingindo o valor zero ao final do sprint, caso ele seja bem sucedido. Mostra-se a seguir um exemplo, assumindo-se um sprint com duração de 15 dias.

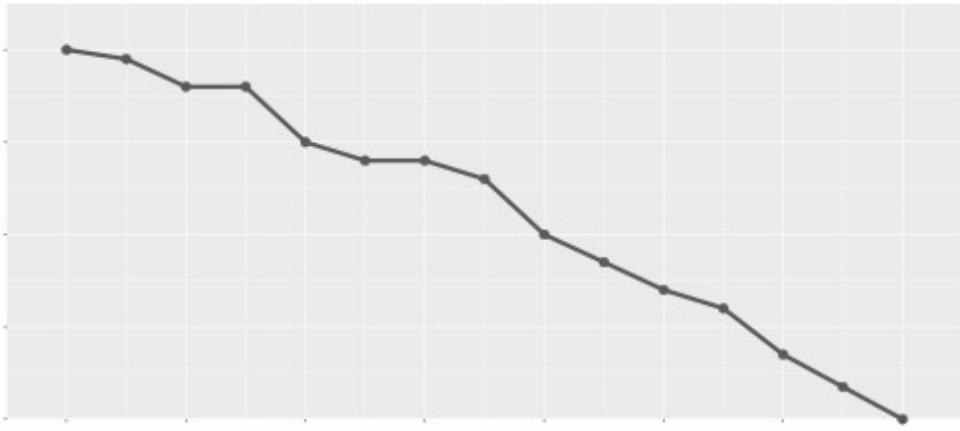


Gráfico de Burndown, assumindo um sprint de 15 dias.

## Outros Eventos

Vamos agora descrever mais três eventos Scrum, especificamente Reuniões Diárias, Revisão do Sprint e Retrospectiva.

Scrum propõe que sejam realizadas **Reuniões Diárias**, de cerca de 15 minutos, das quais devem participar todos os membros do time. Essas reuniões para serem rápidas devem ocorrer com os membros em pé, daí serem também conhecidas como **reuniões em pé** (*standup meetings*, ou ainda *daily scrum*). Nelas, cada membro do time deve responder a três perguntas: (1) o que ele fez no dia anterior; (2) o que ele pretende fazer no dia corrente; (3) e se ele está enfrentando algum problema mais sério, isto é, um impedimento, na sua tarefa. Essas reuniões têm como objetivo melhorar a comunicação entre os membros do time, fazendo com que eles compartilhem e socializem o andamento do projeto. Por exemplo, dois desenvolvedores podem tomar ciência, durante a reunião diária, que eles vão começar a modificar o mesmo trecho de código. Portanto, seria recomendável que eles se reunissem, separadamente do resto do time, para discutir essas modificações. E, com isso, minimizar as chances de possíveis conflitos de integração.

A **Revisão do Sprint** (*Sprint Review*) é uma reunião para mostrar os resultados de um sprint. Dela devem participar todos os membros do time e idealmente outros stakeholders, convidados pelo Dono do Produto, que

estejam envolvidos com o resultado do sprint. Durante essa reunião o time demonstra, ao vivo, o produto para os clientes. Como resultado, todas as histórias do sprint podem ser aprovadas pelo Dono do Produto. Por outro lado, caso ele detecte problema em alguma história, ela deve voltar para o Backlog do Produto, para ser retrabalhada em um próximo sprint. O mesmo deve ocorrer com as histórias que o time não concluiu durante o sprint.

A **Retrospectiva** é a última atividade de um sprint. Trata-se de uma reunião do time Scrum, com o objetivo de refletir sobre o sprint que está terminando e, se possível, identificar pontos de melhorias no processo, nas pessoas, nos relacionamentos e nas ferramentas usadas. Apenas para dar um exemplo, como resultado de uma retrospectiva, o time pode acordar sobre a importância de todos estarem presentes, pontualmente, nas reuniões diárias, pois nos últimos sprints alguns membros estão se atrasando. Veja, portanto, que uma retrospectiva não é uma reunião para lavar a roupa suja e para membros ficarem discutindo entre si. Se for necessário, isso deve ser feito em particular, em outras reuniões ou com a presença de gerentes da organização. Depois da retrospectiva, o ciclo se repete, com um novo sprint.

Uma característica marcante de todos os eventos Scrum é terem uma duração bem definida, que é chamada de **time-box** da atividade. Por isso, esse termo aparece sempre em documentos Scrum. Por exemplo, veja essa frase do Scrum Guide oficial: o coração do método Scrum é um sprint, que tem um time-box de um mês ou menos e durante o qual um produto *done*, usável e que potencialmente pode ser colocado em produção é criado ([link](#)). O objetivo da fixação de *time boxes* é criar um fluxo contínuo de trabalho, bem como fomentar o compromisso da equipe com o sucesso do sprint e evitar a perda de foco.

A próxima tabela mostra o time-box dos eventos Scrum. No caso de eventos com um time-box máximo (exemplo: planejamento do sprint), o valor recomendado refere-se a um sprint de um mês. Se o sprint for menor, o time-box sugerido deve ser também menor.

Evento	Time-box
Planejamento do Sprint	máximo de 8 horas
Sprint	menos de 1 mês

Reunião Diária	15 minutos
Revisão do Sprint	máximo de 4 horas
Retrospectiva	máximo de 3 horas

## Exemplo: Escrita de um Livro

Este livro está sendo escrito usando artefatos e eventos de Scrum. Claro que apenas alguns, pois o livro tem um único autor que, em certa medida, desempenha todos os papéis previstos por Scrum. Logo no início do projeto, os capítulos do livro foram planejados, constituindo assim o Backlog do Produto. A escrita de cada capítulo é considerada como sendo um sprint. Na reunião de Planejamento do Sprint, define-se a divisão do capítulo em seções, que são equivalentes às tarefas. Então começa-se a escrita de cada capítulo, isto é, tem início um sprint. Via de regra, os sprints são planejados para ter uma duração de três meses. Para ficar mais claro, mostra-se a seguir o backlog do sprint atual, bem como o estado de cada tarefa, exatamente no momento em que se está escrevendo este parágrafo:

História	A fazer	Em andamento	Concluídas
Cap. 2 - Processos de Desenvolvimento			

Kanban

Quando não usar Métodos Ágeis

Outros Processos

Exercícios

| Scrum |

Introdução

Manifesto Ágil

XP

|

Decidiu-se adotar um método ágil para escrita do livro para minimizar os riscos de desenvolver um produto que não atende às necessidades de nossos clientes, que, nesta primeira versão, são estudantes e professores brasileiros de disciplinas de Engenharia de Software, principalmente em nível de graduação. Assim, ao final de cada sprint, um capítulo é lançado e divulgado publicamente, de forma a receber feedback. Com isso, evita-se uma solução Waterfall, por meio da qual o livro seria escrito por cerca de dois anos, sem receber qualquer feedback.

Para finalizar, vamos comentar sobre o critério para conclusão de um capítulo, ou seja, para definir que um capítulo está finalizado (*done*). Esse critério requer a leitura e revisão completa do capítulo, pelo autor do livro. Concluída essa revisão, o capítulo é divulgado preliminarmente para os membros do Grupo de Pesquisa em Engenharia de Software Aplicada, do DCC/UFMG.

## Perguntas Frequentes

Antes de concluir a seção, vamos responder algumas perguntas sobre Scrum:

**O que significa a palavra Scrum?** O nome não é uma sigla, mas uma referência à reunião de jogadores realizada em uma partida de rugby para decidir quem vai ficar com a bola após uma infração involuntária.

**O que é um squad?** Esse termo é um sinônimo para time ágil ou time Scrum. O nome foi popularizado pela Spotify. Assim como times Scrum, squads são pequenos, cross-funcionais e auto-organizáveis. É comum ainda usar o nome tribo para denotar um conjunto de squads.

**O Dono do Produto pode ser um comitê?** Em outras palavras, pode existir mais de um Dono de Produto em um time Scrum? A resposta é não. Apenas um membro do time exerce essa função. O objetivo é evitar decisões por comitê, que tendem a gerar produtos lotados de funcionalidades, mas que foram implementadas apenas para atender a determinados membros do comitê. Porém, nada impede que o Dono do Produto faça a ponte entre o time e outros usuários com amplo domínio da área do produto que está

sendo construído. Na verdade, essa é uma tarefa esperada de Donos de Produto, pois às vezes existem requisitos que são do domínio de apenas alguns colaboradores da organização. Cabe então ao Dono do Produto intermediar as conversas entre os desenvolvedores e tais usuários.

**O Scrum Master deve exercer seu papel em tempo integral?** Idealmente, sim. Porém, em times maduros, que conhecem e praticam Scrum há bastante tempo, às vezes não é necessário ter um Scrum Master em tempo integral. Nesses casos, existem duas possibilidades: (1) permitir que o Scrum Master desempenhe esse papel em mais de um time Scrum; (2) alocar a responsabilidade de Scrum Master a um dos membros do time. No entanto, caso a segunda alternativa seja adotada, o Scrum Master não deve ser também o Dono do Produto. A razão é que uma das responsabilidades do Scrum Master é exatamente acompanhar e auxiliar o Dono do Produto em suas tarefas de escrever e priorizar histórias do usuário.

**O Scrum Master precisa ter um diploma de nível superior em um curso da área de Computação?** Não, pois sua função envolve remover impedimentos e assegurar que o time esteja seguindo os princípios de Scrum. Portanto, ele não é um resolvedor de problemas técnicos, tais como bugs, uso correto de frameworks, implementação de funcionalidades, etc. Por outro lado, isso não impede que um desenvolvedor técnico, com um curso superior na área de Computação, assuma as funções de Scrum Master, como vimos na resposta anterior. Existem também certificações para Scrum Master, as quais podem ser requeridas por empresas que decidem adotar Scrum.

**Além de histórias, quais outros itens podem fazer parte do Backlog do Produto?** Também podem ser cadastrados no Backlog do Produto itens como bugs — principalmente aqueles mais complexos e que demandam dias para serem resolvidos — e também melhorias em histórias já implementadas.

**Existem gerentes quando se usa Scrum?** A resposta é sim! De fato, times Scrum são autônomos para implementar as histórias priorizadas pelo Dono do Produto. Porém, um projeto demanda diversas outras decisões que devem ser tomadas em um nível gerencial. Dentre essas decisões, podemos citar as seguintes:

- Contratar e alocar membros para os times Scrum; ou seja, os desenvolvedores não têm autonomia para escolher em quais times eles vão trabalhar. Essa é uma decisão de mais alto nível e, portanto, tomada por gerentes.
- Decidir os objetivos e responsabilidades de cada time, incluindo o sistema — ou parte de um sistema — que o time irá desenvolver usando Scrum. Por exemplo, um time não decide, por conta própria, que a organização precisa de um novo sistema de contabilidade e então começa a desenvolvê-lo. Essa é uma decisão estratégica, que cabe aos gerentes e executivos da organização.
- Gerenciar e administrar questões de recursos humanos, incluindo contratações de novos funcionários, desligamentos de funcionários, promoções, transferências, treinamentos, etc.
- Avaliar se os resultados produzidos pelos times Scrum estão, de fato, gerando benefícios e valor para a organização.

## Kanban

A palavra japonesa *kanban* significa cartão visual ou cartão de sinalização. Desde a década de 50, o nome também é usado para denotar o processo de produção *just-in-time* usado em fábricas japonesas, principalmente naquelas da Toyota, onde ele foi usado pela primeira vez. O processo também é conhecido como Sistema de Produção da Toyota (TPS) ou, mais recentemente, por manufatura *lean*. Em uma linha de montagem, os cartões são usados para controlar o fluxo de produção.

No caso de desenvolvimento de software, Kanban foi usado pela primeira vez na Microsoft, em 2004, como parte de um esforço liderado por David Anderson, então um funcionário da empresa ([link](#)). Segundo Anderson, Kanban é um método que ajuda times de desenvolvimento a trabalhar em ritmo sustentável, eliminando desperdício, entregando valor com frequência e fomentando uma cultura de melhorias contínuas.

Para começar a explicar Kanban, vamos usar uma comparação com Scrum. Primeiro, Kanban é mais simples do que Scrum, pois não usa nenhum dos eventos de Scrum, incluindo sprints. Também, não existe nenhum dos papéis (Dono do Produto, Scrum Master, etc.), pelo menos da forma rígida preconizada por Scrum. Por fim, não existe nenhum dos artefatos Scrum, com uma única e central exceção: o quadro de tarefas, que é chamado de **Quadro Kanban** (*Kanban Board*), e que inclui também o Backlog do Produto.

O Quadro Kanban é dividido em colunas, da seguinte forma:

- A primeira coluna é o backlog do produto. Como em Scrum, usuários escrevem as histórias, que vão para o Backlog.
- As demais colunas são os passos que devem ser seguidos para transformar uma história do usuário em uma funcionalidade executável. Por exemplo, pode-se ter colunas como Especificação, Implementação e Revisão de Código. A ideia, portanto, é que as histórias sejam processadas passo a passo, da esquerda para a direita, como em uma linha de montagem. Além disso, cada coluna é dividida em duas subcolunas: em execução e concluídas. Por exemplo, a coluna implementação tem duas subcolunas: tarefas em implementação e tarefas implementadas. As tarefas concluídas em um passo estão aguardando serem puxadas, por um membro do time, para o próximo passo. Por isso, Kanban é chamado de um sistema *pull*.

Mostra-se abaixo um exemplo de Quadro Kanban. Observe que existe uma história no backlog (H3), além de uma história (H2) que já foi puxada por algum membro do time para o passo de especificação. Existem ainda quatro tarefas (T6 a T9) que foram especificadas a partir de uma história anterior. Continuando, existem duas tarefas em implementação (T4 e T5) e existe uma tarefa implementada e aguardando ser puxada para revisão de código (T3). No último passo, existe uma tarefa em revisão (T2) e uma tarefa cujo processamento está concluído (T1). Por enquanto, não se preocupe com a sigla WIP que aparece em todos os passos, exceto no backlog. Vamos explicá-la em breve. Além disso, representamos as histórias e tarefas pelas letras H e T, respectivamente. Porém, em um quadro real, ambas são cartões

auto-adesivos com uma pequena descrição. O Quadro Kanban pode ser assim montado em uma das paredes do ambiente de trabalho do time.

Backlog	Especificação WIP		Implementação WIP		Revisão de Código WIP	
H3	em espec.  H2	especificadas  T6 T7 T8 T9	em implementação  T4  T5	implementadas  T3	em revisão  T2	revisadas  T1

Agora, mostraremos uma evolução do projeto. Isto é, alguns dias depois, o Quadro Kanban passou para o seguinte estado (as tarefas que avançaram no quadro estão sublinhadas e em vermelho).

Backlog	Especificação		Implementação		Revisão de Código	
	WIP		WIP		WIP	
H3	em espec.	especificadas <b>T8 T9</b> <b>T10 T11 T12</b>	em implementação <b>T4</b> <b>T5</b> <b>T6</b> <b>T7</b>	implementadas	em revisão <b>T3</b>	revisadas <b>T1</b> <b>T2</b>

Veja que a história H2 desapareceu, pois foi quebrada em três tarefas (T10, T11 e T12). O objetivo da fase de especificação é exatamente transformar uma história em uma lista de tarefas. Continuando, T6 e T7 — que antes estavam esperando — entraram em implementação. Já T3 entrou na fase de revisão de código. Por fim, a revisão de T2 terminou. Veja ainda que, neste momento, não existe nenhuma tarefa implementada e aguardando ser puxada para revisão.

Como em outros métodos ágeis, times Kanban são auto-organizáveis. Isso significa que eles têm autonomia para definir qual tarefa vai ser puxada para o próximo passo. Eles também são cross-funcionais, isto é, devem incluir membros capazes de realizar todos os passos do Quadro Kanban.

Por fim, resta explicar o conceito de **Limites WIP** (*Work in Progress*). Via de regra, métodos de gerenciamento de projetos têm como objetivo garantir um ritmo sustentável de trabalho. Para isso, deve-se evitar duas situações extremas: (1) o time ficar ocioso boa parte do tempo, sem tarefa para realizar; ou (2) o time ficar sobrecarregado de trabalho e, por isso, não

conseguir produzir software de qualidade. Para evitar a segunda situação — sobrecarga de trabalho — Kanban propõe um limite máximo de tarefas que podem estar em cada um dos passos de um Quadro Kanban. Esse limite é conhecido pelo nome Limite WIP, isto é, trata-se do limite máximo de cartões presentes em cada passo, contando aqueles na primeira coluna (em andamento) e aqueles na segunda coluna (concluídos) do passo. A exceção é o último passo, no qual o WIP aplica-se apenas à primeira subcoluna, já que não faz sentido aplicar um limite ao número de tarefas concluídas pelo time de desenvolvimento.

A seguir, reproduzimos o último Quadro Kanban, mas com os limites WIP. Eles são os números que aparecem abaixo do nome de cada passo, na primeira linha do quadro. Ou seja, nesse Quadro Kanban, admite-se um máximo de 2 histórias em especificação; 5 tarefas em implementação; e 3 tarefas em revisão. Vamos deixar para explicar o limite WIP do passo Especificação por último. Mas podemos ver que existem 4 tarefas em implementação (T4, T5, T6 e T7). Portanto, abaixo do WIP desse passo, que é igual a 5 tarefas. Em Revisão de Código, o limite é de 3 tarefas e também está sendo respeitado, pois existe apenas uma tarefa em revisão (T3). Veja que para fins de verificar o limite WIP contam-se as tarefas em andamento (1a subcoluna de cada passo) e concluídas (2a subcoluna de cada passo), com exceção do último passo, no qual consideram-se apenas as tarefas da primeira subcoluna (T3, no exemplo).

Backlog	Especificação 2		Implementação 5		Revisão de Código 3	
H3	em espec.	especificadas <b>T8 T9 T10 T11 T12</b>	em implementação <b>T4 T5 T6 T7</b>	implementadas	em revisão <b>T3</b>	revisadas <b>T1 T2</b>

Agora vamos explicar o WIP do passo Especificação. Para verificar o WIP desse passo, deve-se somar as histórias em especificação (zero no quadro anterior) e as histórias que já foram especificadas. No caso, temos duas histórias especificadas. Ou seja, T8 e T9 são tarefas que resultaram da especificação de uma mesma história. E as tarefas T10, T11 e T12 são resultado da especificação de uma segunda história. Portanto, para fins do cálculo de WIP, temos duas histórias no passo, o que está dentro do seu limite, que também é 2. Para facilitar a visualização, costuma-se representar as tarefas resultantes da especificação de uma mesma história em uma única linha. Seguindo esse padrão, para calcular o WIP do passo Especificação, deve-se somar as histórias da primeira subcoluna (zero, no nosso exemplo) com o número de linhas na segunda coluna (duas).

Ainda no quadro anterior, e considerando os limites WIP, tem-se que:

- A história H3, que está no backlog, não pode ser puxada para o passo Especificação, pois o WIP desse passo está no limite.
- Uma das tarefas já especificadas (T8 a T12) pode ser puxada para implementação, pois o WIP do passo está em 4, enquanto o limite é 5.

- Uma ou mais tarefas em implementação (T4 a T7) podem ser finalizadas, o que não altera o WIP do passo.
- A revisão de T3 pode ser finalizada.

Apenas reforçando, o objetivo dos limites WIP é evitar que os times Kanban fiquem sobrecarregados de trabalho. Quando um desenvolvedor tem muitas tarefas para realizar — porque os limites WIP não estão sendo respeitados — a tendência é que ele não consiga concluir nenhuma dessas tarefas com qualidade. Como usual em qualquer atividade humana, quando assumimos muitos compromissos, a qualidade de nossas entregas cai muito. Kanban reconhece esse problema e, para que ele não ocorra, cria uma trava automática para impedir que os times aceitem trabalhos além da sua capacidade de entrega. Essas travas, que são os limites WIP, servem para uso interno do time e, mais importante ainda, para uso externo. Ou seja, elas são o instrumento de que um time dispõe para recusar trabalho extra que está sendo empurrado de cima para baixo por gerentes da organização, por exemplo.

## Calculando os Limites WIP

Resta-nos agora explicar como os limites WIP são definidos. Existe mais de uma alternativa, mas vamos adotar uma adaptação de um algoritmo proposto por Eric Brechner — um engenheiro da Microsoft — em seu livro sobre o uso de Kanban no desenvolvimento de software ([link](#)). O algoritmo é descrito a seguir.

Primeiro, temos que estimar quanto tempo em média uma tarefa vai ficar em cada passo do Quadro Kanban. Esse tempo é chamado de **lead time (LT)**. No nosso exemplo, vamos supor os seguintes valores:

- LT(especificação) = 5 dias
- LT(implementação) = 12 dias
- LT(revisão) = 6 dias

Veja que essa estimativa considera uma tarefa média, pois sabemos que vão existir tarefas mais complexas e mais simples. Veja ainda que o *lead time* inclui o tempo em fila, isto é, o tempo que a tarefa vai ficar na 2a subcoluna dos passos do Quadro Kanban aguardando ser puxada para o passo seguinte.

Em seguida, deve-se estimar o **throughput (TP)** do passo com maior *lead time* do Quadro Kanban, isto é, o número de tarefas produzidas por dia nesse passo. No nosso exemplo, e na maioria dos projetos de desenvolvimento de software, esse passo é a Implementação. Assim, suponha que o time seja capaz de sustentar a implementação de 8 tarefas por mês. O **throughput** desse passo é então:  $8 / 21 = 0.38$  tarefas/dia. Veja que consideramos que um mês tem 21 dias úteis.

Por fim, o WIP de cada passo é assim definido:

$$\mathbf{WIP(\text{passo}) = TP * LT(\text{passo})}.$$

onde throughput refere-se ao throughput do passo mais lento, conforme calculado no item anterior.

Logo, teremos os seguintes resultados:

- $\mathbf{WIP(\text{especificação}) = 0.38 * 5 = 1.9}$
- $\mathbf{WIP(\text{implementação}) = 0.38 * 12 = 4.57}$
- $\mathbf{WIP(\text{revisão}) = 0.38 * 6 = 2.29}$

Arredondando para cima, os resultados finais ficam assim:

- $\mathbf{WIP(\text{especificação}) = 2}$
- $\mathbf{WIP(\text{implementação}) = 5}$
- $\mathbf{WIP(\text{revisão}) = 3}$

No algoritmo proposto por Eric Brechner, sugere-se ainda adicionar uma margem de erro de 50% nos WIPs calculados, para acomodar variações no tamanho das tarefas, tarefas bloqueadas devido a fatores externos, etc. No entanto, como nosso exemplo é ilustrativo, não vamos ajustar os WIPs que foram calculados acima.

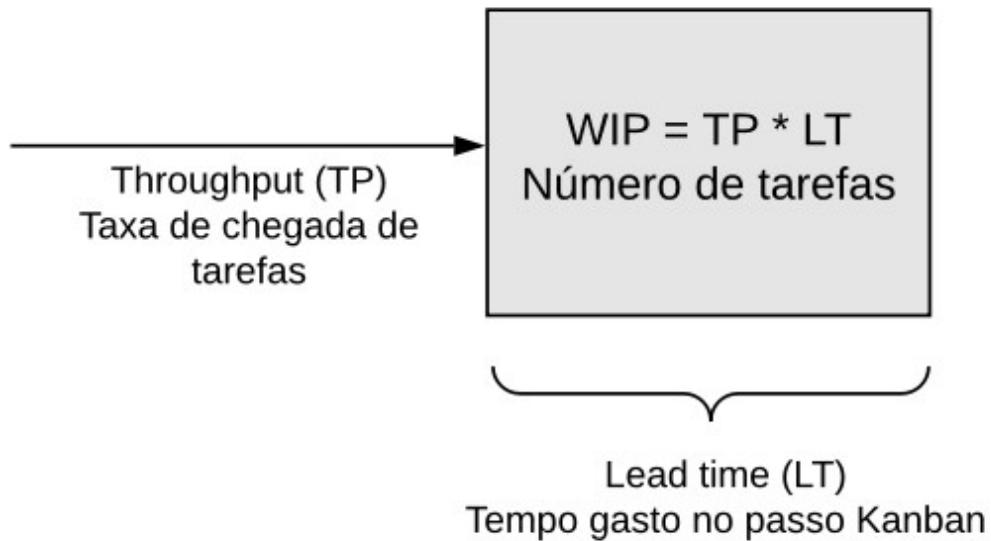
Conforme afirmado, limites WIPs são o recurso oferecido por Kanban para garantir um ritmo de trabalho sustentável e a entrega de sistemas de software com qualidade. O papel desses limites é contribuir para que os desenvolvedores não fiquem sobrecarregados de tarefas e, consequentemente, propensos a baixar a qualidade de seu trabalho. Na verdade, todo método de desenvolvimento de software tende a oferecer esse tipo de recurso. Por exemplo, em Scrum existe o conceito de sprints com time-boxes definidos, cujo objetivo é evitar que times aceitem trabalhar em histórias que ultrapassam a sua velocidade de entrega. Adicionalmente, uma vez iniciado, o objetivo de um sprint não pode ser alterado, de forma a blindar o time de mudanças diárias de prioridade. No caso de métodos Waterfall, o recurso para garantir um fluxo de trabalho sustentável e de qualidade é a existência de uma fase detalhada de especificação de requisitos. Com essa fase, a intenção era oferecer aos desenvolvedores uma ideia clara do sistema que eles deveriam implementar.

## Lei de Little

O procedimento para cálculo de WIPs explicado anteriormente é uma aplicação direta da **Lei de Little**, um dos resultados mais importantes da Teoria de Filas ([link](#)). A Lei de Little diz que o número de itens em um sistema de filas é igual à taxa de chegada desses itens multiplicado pelo tempo que cada item fica no sistema. Traduzindo para o nosso contexto, o sistema é um passo de um processo Kanban e os itens são tarefas. Assim, temos que:

- WIP: número de tarefas em um dado passo de um processo Kanban.
- Throughput (TP): taxa de chegada dessas tarefas nesse passo.
- Lead Time (LT): tempo que cada tarefa fica nesse passo.

Ou seja, de acordo com a Lei de Little: **WIP = TP \* LT**. Visualmente, podemos representar a Lei de Little como mostrado na próxima figura.



Lei de Little:  $WIP = TP * LT$

## Perguntas Frequentes

Antes de concluir, vamos responder algumas perguntas sobre Kanban:

**Quais são os papéis que existem em Kanban?** Ao contrário de Scrum, Kanban não define uma lista fixa de papéis. Cabe ao time e à organização definir os papéis que existirão no processo de desenvolvimento, tais como Dono do Produto, Testadores, etc.

**Como as histórias dos usuários são priorizadas?** Kanban é um método de desenvolvimento mais leve que Scrum e mesmo que XP. Um dos motivos é que ele não define critérios de priorização de histórias. Como respondido na pergunta anterior, não se define, por exemplo, que o time tem que possuir um Dono do Produto, responsável por essa priorização. Veja que essa é uma possibilidade, isto é, pode existir um Dono do Produto em times Kanban. Mas outras soluções também são possíveis, como priorização externa, por um gerente de produto.

**Times Kanban podem realizar eventos típicos de Scrum, como reuniões diárias, revisões e retrospectivas?** Sim, apesar de Kanban não prescrever a realização desses eventos. Porém, também não há um voto explícito aos mesmos. Cabe ao time decidir quais eventos são importantes, quando eles devem ser realizados, com que duração, etc.

**Em vez de um Quadro Kanban físico, com adesivos em uma parede ou em um quadro branco, pode-se usar um software para gerenciamento de projetos?** Kanban não proíbe o uso de software de gerenciamento de projetos. Porém, recomenda-se o uso de um quadro físico, pois um dos princípios mais importantes de Kanban é a visualização do trabalho pelo time, de forma que seus membros possam a qualquer momento tomar conhecimento do trabalho em andamento e de possíveis problemas e gargalos que estejam ocorrendo. Em alguns casos, chega-se a recomendar a adoção de ambas soluções: um quadro físico, mas com um backup em um software de gerenciamento de projetos, que possa ser acessado pelos gerentes e executivos da organização.

## Quando Não Usar Métodos Ágeis?

Apesar de métodos ágeis — como aqueles estudados nas seções anteriores — terem alcançado um sucesso inquestionável, é bom lembrar que não existe bala de prata em Engenharia de Software, conforme comentamos no Capítulo 1. Assim, neste capítulo vamos discutir cenários e domínios nos quais práticas ágeis podem não ser adequadas.

No entanto, a pergunta que abre essa seção não admite uma resposta simples, como, por exemplo, sistemas das áreas X, Y e Z não devem usar métodos ágeis; e os demais devem usar. Em outras palavras, sistemas de qualquer área podem se beneficiar de pelo menos algumas das práticas propostas por métodos ágeis. Por outro lado, existem práticas que não são recomendadas para determinados tipos de sistemas, organizações e contextos. Assim, vamos responder a pergunta proposta em uma granularidade mais fina. Isto é, vamos comentar a seguir sobre quando **não** usar determinadas práticas de desenvolvimento ágil.

- **Design Incremental.** Esse tipo de design faz sentido quando o time tem uma primeira visão do design do sistema. Se o time não tem essa visão, ou o domínio do sistema é novo e complexo, ou o custo de mudanças futuras é muito alto, recomenda-se adotar uma fase de design e análise inicial, antes de partir para iterações que requeiram implementações de funcionalidades.
- **Histórias do Usuário.** Histórias são um método leve para especificação de requisitos, que depois são clarificados com o envolvimento cotidiano de um representante dos clientes no projeto. Porém, em certos casos, pode ser importante ter uma especificação detalhada de requisitos no início do projeto, principalmente se ele for um projeto de uma área totalmente nova para o time de desenvolvedores.
- **Envolvimento do Cliente.** Se os requisitos do sistema são estáveis e de pleno conhecimento do time de desenvolvedores, não faz sentido ter um Representante dos Clientes ou Dono do Produto integrado ao time. Por exemplo, esse papel não é importante no desenvolvimento de um compilador para uma linguagem conhecida, com uma gramática e semântica consolidadas.
- **Documentação Leve e Simplificada.** Em certos domínios, documentações detalhadas de requisitos e de projeto são mandatórias. Por exemplo, sistemas cujas falhas podem causar a morte de seres humanos, como aqueles das áreas médicas e de transporte, costumam demandar certificação por uma entidade externa, que pode exigir uma documentação detalhada, além do código fonte.
- **Times Auto-organizáveis.** Times ágeis são autônomos e empoderados para trabalhar sem interferências durante o time-box de uma iteração. Consequentemente, eles não precisam prestar contas diárias para os gerentes e executivos da organização. No entanto, essa característica pode ser incompatível com os valores e cultura de certas organizações, principalmente aquelas com uma tradição de níveis hierárquicos e de controle rígidos.
- **Contratos com Escopo Aberto.** Em contratos com escopo aberto, a remuneração é por hora trabalhada. Assim, a empresa que assina o

contrato não tem — no momento da assinatura — uma ideia precisa de quais funcionalidades serão implementadas e nem do prazo e custo do sistema. Algumas organizações podem não se sentir seguras para assinar esse tipo de contrato, principalmente quando elas não têm uma experiência prévia com desenvolvimento ágil ou referências confiáveis sobre a empresa contratada.

Para concluir, é importante mencionar que duas práticas ágeis são atualmente adotadas na grande maioria de projetos de software:

- Times pequenos, pois o esforço de sincronização cresce muito quando os times são compostos por dezenas de membros.
- Iterações (ou sprints), mesmo que com duração maior do que aquela típica de métodos ágeis. Por exemplo, iterações com duração de dois ou três meses, em vez de iterações com menos de 30 dias. Na verdade, entre o surgimento de Waterfall e de métodos ágeis, alguns métodos iterativos foram propostos, isto é, métodos com pontos de validação ao longo do desenvolvimento. Na próxima seção, iremos estudar dois desses métodos.

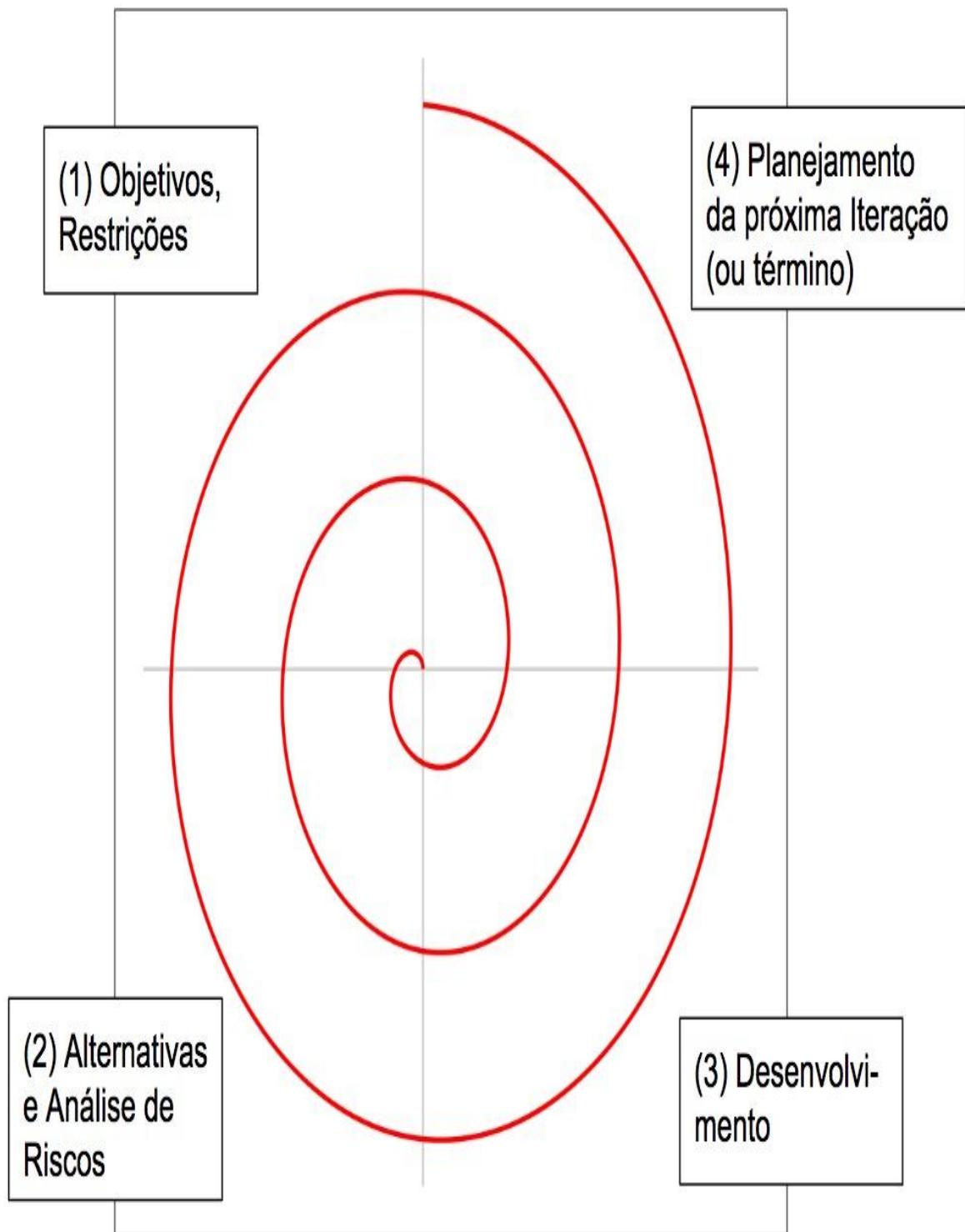
## Outros Métodos Iterativos

A transição entre Waterfall — dominante nas décadas de 70 e 80 — e métodos ágeis — que começaram a surgir na década de 90, mas que só ganharam popularidade no final dos anos 2000 — foi gradativa. Como em métodos ágeis, esses métodos surgidos nesse período de transição possuem o conceito de iterações. Ou seja, eles não são estritamente sequenciais, como em Waterfall. Porém, as iterações têm duração maior do que aquela usual em desenvolvimento ágil. Em vez de poucas semanas, elas duram alguns meses. Por outro lado, eles preservam características relevantes de Waterfall, como ênfase em documentação e em uma fase inicial de levantamento de requisitos e depois de design.

Um exemplo de proposta de processo surgida nessa época é o **Modelo em Espiral**, proposto por Barry Boehm, em 1986 ([link](#)). Nesse modelo, um sistema é desenvolvido na forma de uma espiral de iterações. Cada iteração,

ou volta completa na espiral, inclui quatro etapas (acompanhe também na próxima figura):

- Definição de objetivos e restrições, tais como custos, cronogramas, etc.
- Avaliação de alternativas e análise de riscos. Por exemplo, pode-se chegar à conclusão de que é mais interessante comprar um sistema pronto, do que desenvolver o sistema internamente.
- Desenvolvimento e testes, por exemplo, usando Waterfall. Ao final dessa etapa, deve-se gerar um protótipo que possa ser demonstrado aos usuários do sistema.
- Planejamento da próxima iteração ou então tomar a decisão de parar, pois o que já foi construído é suficiente para atender às necessidades da organização.



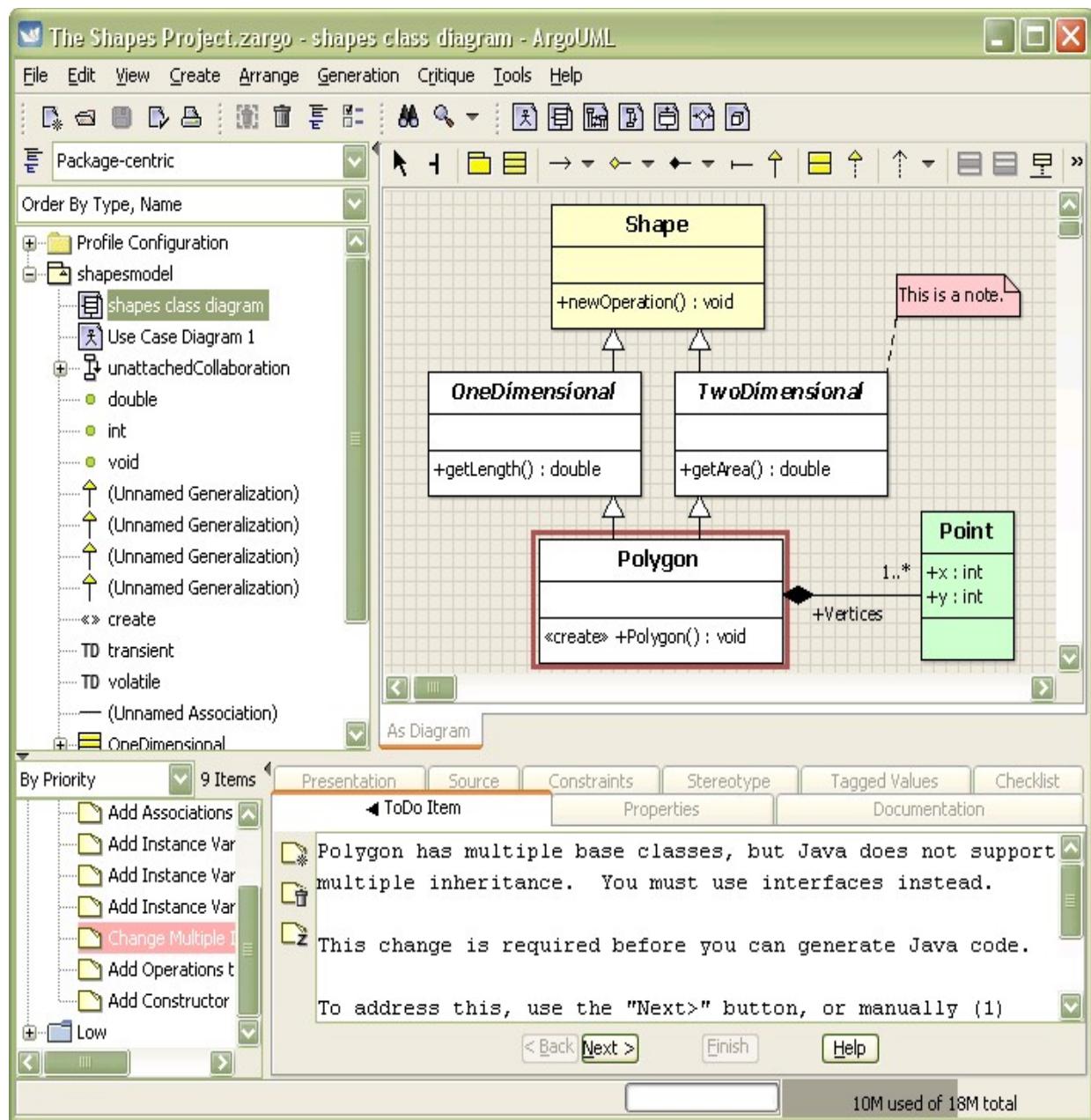
Modelo Espiral. Cada iteração é dividida em quatro etapas.

Assim, o Modelo em Espiral produz, a cada iteração, versões mais completas de um sistema, começando da versão gerada no centro da espiral. Porém, cada iteração, somando-se as quatro fases, pode levar de 6 a 24 meses. Portanto, mais do que em XP e Scrum. Uma outra característica importante é a existência de uma fase explícita de análise de riscos, da qual devem resultar medidas concretas para mitigar os riscos identificados no projeto.

O **Processo Unificado (UP)**, proposto no final da década de 90, é outro exemplo de método iterativo de desenvolvimento. UP foi proposto por profissionais ligados a uma empresa de consultoria e de ferramentas de apoio ao desenvolvimento de software chamada Rational, que em 2003 seria comprada pela IBM. Por isso, o método é também chamado de **Rational Unified Process (RUP)**.

Devido a suas origens, UP é vinculado a duas tecnologias específicas:

- UP é baseado na linguagem de modelagem UML. Todos os seus resultados são documentados e representados usando-se diagramas gráficos de UML. No Capítulo 4, iremos estudar UML com mais calma. Por enquanto, vamos ressaltar que a proposta era ter uma linguagem de modelagem unificada (UML) e também um processo unificado (UP), ambos propostos pelo mesmo grupo de profissionais.
- UP é associado a ferramentas de apoio ao projeto e análise de software, conhecidas como **ferramentas CASE** (*Computer-Aided Software Engineering*). O nome é uma analogia com ferramentas CAD (*Computer Aided-Design*), usadas em projetos de Engenharia Civil, Engenharia Mecânica, Arquitetura, etc. A ideia era que o projeto e análise de um sistema deveriam ser integralmente baseados em diagramas UML. Mas esses diagramas não seriam desenhados em papel e sim usando-se ferramentas computacionais (veja um exemplo na figura da próxima página). A Rational, além de propor o método UP, também desenvolvia e vendia licenças de uso de ferramentas CASE.



Projeto usando ferramenta CASE. Figura gentilmente cedida pelos desenvolvedores do sistema ArgoUML.

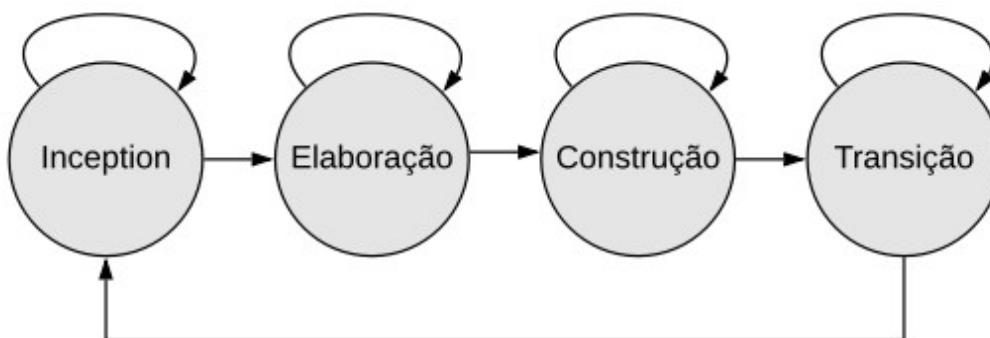
UP propõe que o desenvolvimento seja decomposto nas seguintes fases:

- **Inception** (às vezes, traduzida como iniciação ou concepção): que inclui análise de viabilidade, definição de orçamentos, análise de riscos e definição de escopo do sistema. Ao final dessa fase, o caso de negócios

(*business case*) do sistema deve estar bem claro. Pode-se inclusive decidir que não vale a pena desenvolver o sistema, mas sim comprar um sistema pronto.

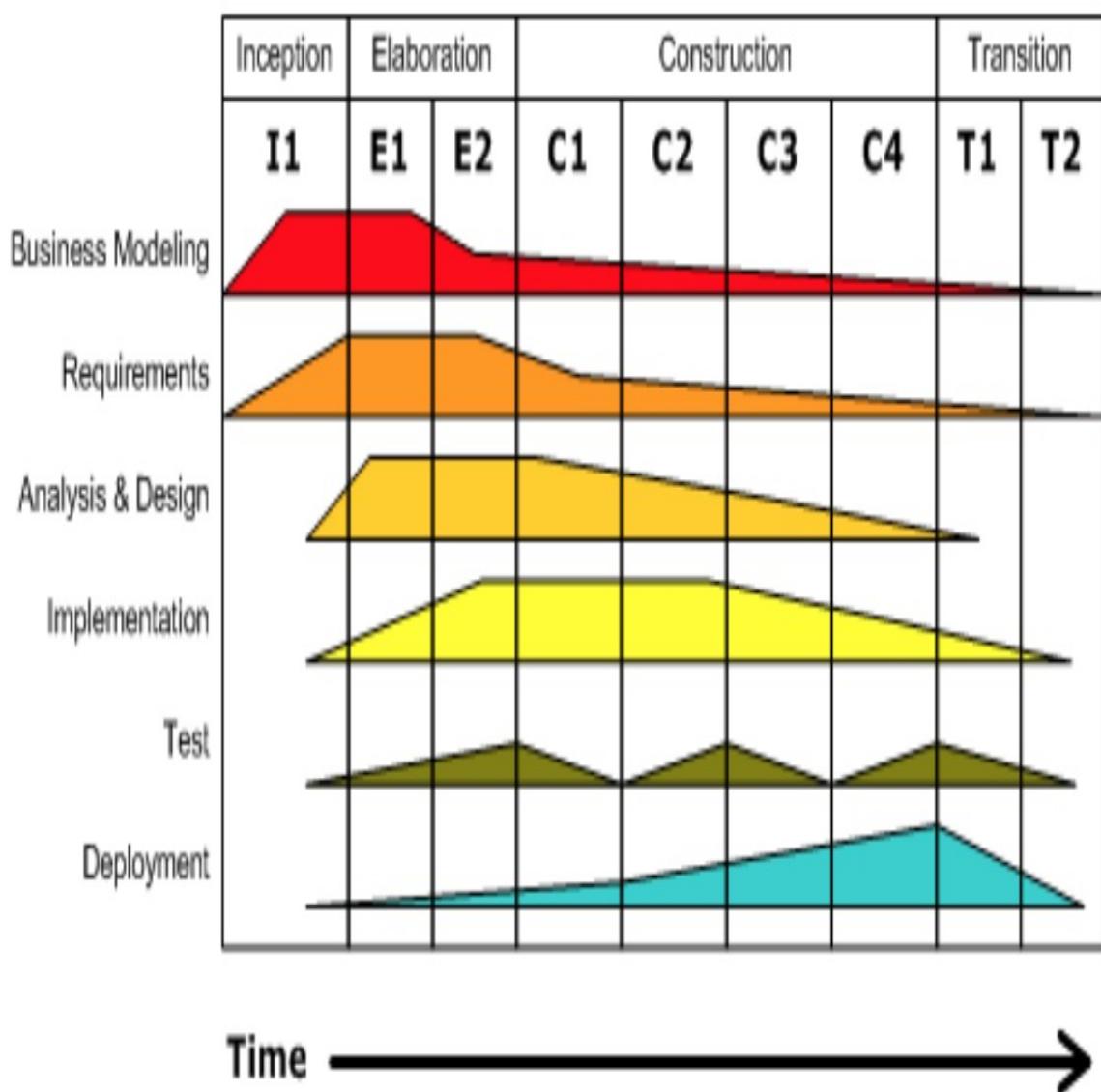
- **Elaboração:** que incluiu especificação de requisitos (via casos de uso de UML), definição da arquitetura do sistema, bem como de um plano para o seu desenvolvimento. Ao final dessa fase, todos os riscos identificados na fase anterior devem estar devidamente controlados e mitigados.
- **Construção:** na qual se realiza o projeto de mais baixo nível, implementação e testes do sistema. Ao final dessa fase, deve ser disponibilizado um sistema funcional, incluindo documentação e manuais, que possam ser validados pelos usuários.
- **Transição:** na qual ocorre a disponibilização do sistema para produção, incluindo a definição de todas as rotinas de implantação, como políticas de backup, migração de dados de sistemas legados, treinamento da equipe de operação, etc.

Assim como no Modelo Espiral, pode-se repetir várias vezes o processo; ou seja, o desenvolvimento é incremental, com novas funcionalidades sendo entregues a cada iteração. Adicionalmente, pode-se repetir cada uma das fases. Por exemplo, construção — em uma dada iteração — pode ser dividida em duas subfases, cada uma construindo uma parte do produto. A próxima figura ilustra o modelo de iterações de UP.



Fases e iterações do Processo Unificado (UP). Repetições são possíveis em cada fase (auto-laços). E também pode-se repetir todo o fluxo (laço externo), para gerar mais um incremento de produto.

UP define também um conjunto de disciplinas de engenharia que incluem por exemplo: modelagem de negócios, definição de requisitos, análise e design, implementação, testes e implantação. Essas disciplinas — ou fluxos de trabalho — podem ocorrer em qualquer fase. Porém, espera-se que algumas disciplinas sejam mais intensas em determinadas fases, como mostra a próxima figura. No projeto ilustrado, tarefas de modelagem de negócio estão concentradas nas fases iniciais do projeto (inception e elaboração) e quase não ocorrem nas fases seguintes. Por outro lado, implementação está concentrada na fase de Construção.



Fases (na horizontal) e disciplinas (na vertical) de um projeto desenvolvido usando UP. A área da curva mostra a intensidade da disciplina durante cada fase (imagem da Wikipedia, licença: domínio público).

## Bibliografia

Kent Beck, Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2nd edition, 2004.

Kent Beck. *Embracing Change with Extreme Programming*. IEEE Computer, vol. 32, issue 10, p. 70-77, 1999.

Kent Beck, Martin Fowler. *Planning Extreme Programming*. Addison-Wesley, 2000.

Ken Schwaber, Jeff Sutherland. *The Scrum Guide*, 2017.

Kenneth Rubin. *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley, 2012

Eric Brechner. *Agile Project Management with Kanban*. Microsoft Press, 2015.

David Anderson. *Kanban*. Blue Hole Press, 2013.

Ian Sommerville. *Engenharia de Software*. Pearson, 10a edição, 2019.

Hans van Vliet. *Software Engineering: Principles and Practice*. 3rd edition. Wiley, 2008.

Armando Fox, David Patterson. *Construindo Software como Serviço: Uma Abordagem Ágil Usando Computação em Nuvem*. Strawberry Canyon LLC. 1a edição, 2014.

## Exercícios de Fixação

1. Como XP preconiza que devem ser os contratos de desenvolvimento de software?

2. Quais as diferenças entre XP e Scrum?
3. Times Scrum são ditos cross-funcionais e auto-organizáveis. Por quê? Defina esses termos.
4. Em Scrum, qual a diferença entre as histórias do topo e do fundo do Backlog do Produto?
5. O que são e para que servem story points?
6. Em Scrum, qual a diferença entre uma sprint review e uma retrospectiva?
7. Um sprint pode ser cancelado? Se sim, por quem e por qual motivo? Para responder a essa questão, consulte antes o Scrum Guide ([link](#)), que é o guia que documenta a versão oficial de Scrum.
8. Procure pensar em um sistema de uma área da qual tenha algum conhecimento. (a) Escreva então uma história para esse sistema (veja que histórias são especificações resumidas de funcionalidades, com 2-3 sentenças). (b) Em seguida, quebre a história que definiu em algumas tarefas (de forma semelhante ao que fizemos no sistema similar ao Stack Overflow, usado como exemplo na seção sobre XP). (c) Existem dependências entre essas tarefas? Ou elas podem ser implementadas em qualquer ordem?
9. Suponha dois times, A e B, atuando em projetos diferentes, contratados por empresas distintas, sem conexões entre eles. Porém, ambos os times adotam sprints de 15 dias e ambos possuem 5 desenvolvedores. Nos seus projetos, o time A considera que sua velocidade é de 24 pontos. Já o time B assume uma velocidade de 16 pontos. Pode-se afirmar que A é 50% mais produtivo que B? Justifique sua resposta.
10. Quais são as principais diferenças entre Scrum e Kanban?
11. Quais são as diferenças entre um Quadro Scrum e um Quadro Kanban?
12. Qual o erro existe no seguinte Quadro Kanban?

Backlog	Especificação (2)		Implementação (5)		Validação (3)	
X X	X	XXXX	X	X	X	
X X			X	X	X	
X X			X	X	X	

13. Suponha o seguinte quadro Kanban. Neste momento, o time não consegue trabalhar na especificação de novas histórias, pois o WIP do passo Especificação está sendo totalmente preenchido por itens esperando movimentação para o passo seguinte (Implementação). O que seria mais recomendado neste momento: (a) desrespeitar o WIP e já puxar uma nova história do Backlog para Especificação; ou (b) ajudar o time nas três tarefas em Validação, de forma a desbloquear o fluxo do processo.

Backlog	Especificação (2)		Implementação (5)		Validação (3)	
X		XXX	X	X	X	
X X		XXX	X		X	
X			X		X	
			X			

14. Por que se recomenda que os limites WIP calculados usando a Lei de Little sejam incrementados, por exemplo em 50%, de forma a admitir uma margem de erro? Em outras palavras, quais eventos podem originar esses erros na estimativa dos WIPs?

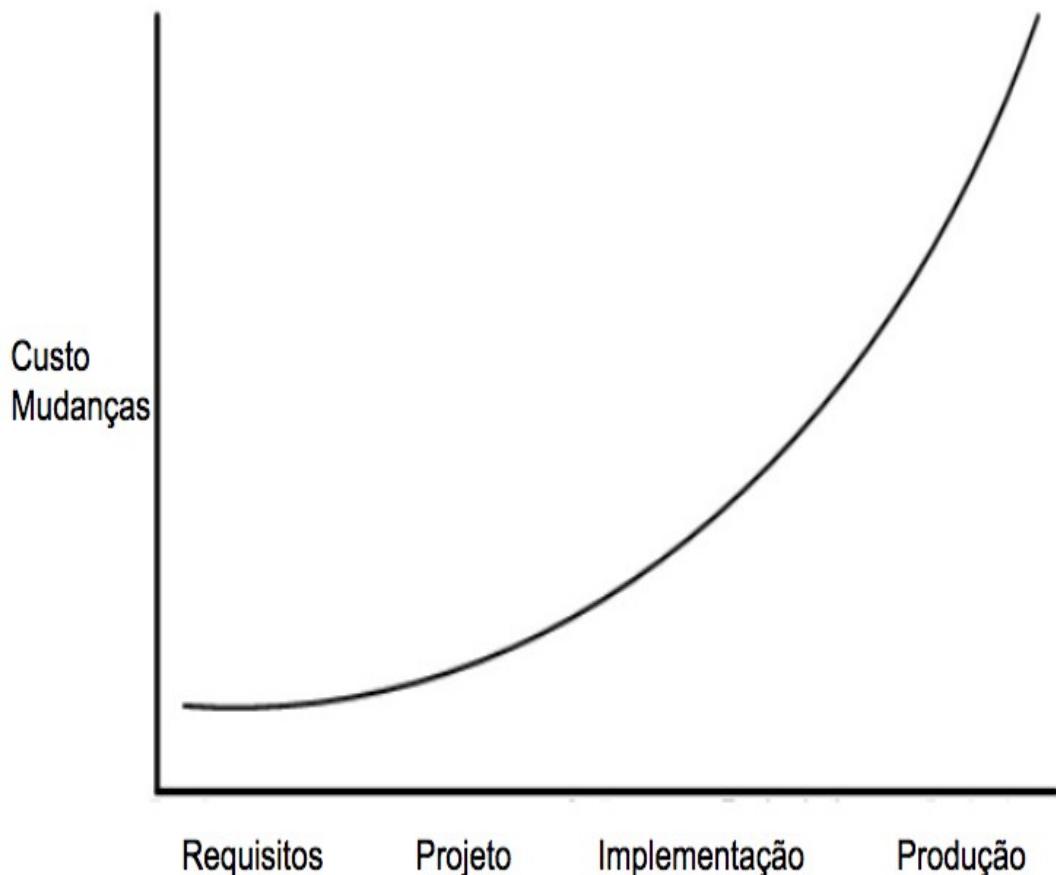
15. Descreva os principais recursos oferecidos por Waterfall, Scrum e Kanban para controlar riscos e garantir um fluxo de trabalho sustentável e que propicie o desenvolvimento de software com qualidade.

16. Seja um processo Kanban, dividido em quatro passos. A tabela a seguir informa o *lead time* de cada um deles e o throughput do passo C, que é o passo mais lento. Com base nesses valores, calcule o WIP de cada passo (última coluna da tabela).

Passo	Lead Time (médio, dias)	Throughput (tarefas/dia)	WIP
A	4	-	
B	3	-	
C	10	0.5	
D	5	-	

Passo	Lead Time (médio, dias)	Throughput (tarefas/dia)	WIP
A	4	-	
B	3	-	
C	10	0.5	
D	5	-	

17. Seja o seguinte gráfico, que mostra — para um determinado sistema — como os custos de mudanças variam conforme a fase do desenvolvimento em que elas são realizadas. (a) Qual método de desenvolvimento você recomendaria para esse sistema? Justifique sua resposta. (b) Sistemas de quais domínios podem ter uma curva de custos de mudanças semelhante a essa?



18. O artigo *Development and Deployment at Facebook* ([link](#)) apresenta os métodos e práticas de desenvolvimento de software usados no Facebook. Na sua primeira seção (páginas 2-3; figura 2), os autores fazem uma distinção entre alguns métodos de desenvolvimento, baseando-se na frequência com que versões de um sistema são liberadas para uso quando se adota cada um deles. Complete a seguinte tabela informando a frequência de releases mencionada no artigo para alguns métodos e políticas de liberação de software.

<b>Método</b>	<b>Frequência de novas releases</b>
Waterfall	Raramente
Evolucionário	Intervalos regulares
Ágil	Constantemente

Facebook

Deployment Contínuo

19. Por que métodos como o Processo Unificado (UP) e Espiral não são considerados ágeis? E qual a diferença deles para o Modelo Waterfall?

# Cap 3 Requisitos

Este capítulo inicia com uma apresentação sobre a importância e os diversos tipos de requisitos de software (Seção 3.1). Em seguida, caracterizamos e apresentamos as atividades que compõem o que chamamos de Engenharia de Requisitos (Seção 3.2). As quatro seções seguintes apresentam quatro técnicas e documentos para especificação e validação de requisitos. Na Seção 3.3, tratamos de histórias de usuários, as quais são os principais instrumentos de Engenharia de Requisitos quando se usa Métodos Ágeis de Desenvolvimento. Em seguida, na Seção 3.4 tratamos de casos de uso, que são documentos tradicionais e mais detalhados para especificação de requisitos. Na Seção 3.5, vamos tratar de Produto Mínimo Viável (MVP), muito usados modernamente para prospectar e validar requisitos. Para concluir, na Seção 3.6 tratamos de Testes A/B, também muito usados hoje em dia para validar e definir os requisitos de produtos de software.

## Introdução

**Requisitos** definem o que um sistema deve fazer e sob quais restrições. Requisitos relacionados com a primeira parte dessa definição — o que um sistema deve fazer, ou seja, suas funcionalidades — são chamados de **Requisitos Funcionais**. Já os requisitos relacionados com a segunda parte — sob que restrições — são chamados de **Requisitos Não-Funcionais**.

Vamos usar novamente o exemplo do Capítulo 1, relativo a um sistema de home-banking, para ilustrar a diferença entre esses dois tipos de requisitos. Em um sistema de home-banking, os requisitos funcionais incluem informar o saldo e extrato de uma conta, realizar transferências entre contas, pagar um boleto bancário, cancelar um cartão de débito, dentre outros. Já os requisitos não-funcionais estão relacionados com a qualidade do serviço prestado pelo sistema, incluindo características como desempenho, disponibilidade, níveis de segurança, portabilidade, privacidade, consumo de memória e disco, dentre outros. Portanto, os requisitos não-funcionais definem restrições ao funcionamento do sistema. Por exemplo, não basta que o sistema de home-banking implemente todas as funcionalidades requeridas pelo banco.

Adicionalmente, ele deve ter uma disponibilidade de 99,9% — a qual funciona, portanto, como uma restrição ao seu funcionamento.

Como expresso por Frederick Brooks na sentença que abre este capítulo, a definição dos requisitos é uma etapa crucial da construção de qualquer sistema de software. De nada adianta ter um sistema com o melhor design, implementado na mais moderna linguagem, usando o melhor processo de desenvolvimento, com alta cobertura de testes e ele não atender às necessidades de seus usuários. Problemas na especificação de requisitos também têm um custo alto. Eles podem requerer trabalho extra, quando se descobre — após o sistema ficar pronto — que os requisitos foram especificados de forma incorreta ou que requisitos importantes não foram especificados. No limite, corre-se o risco de entregar um sistema que vai ser rejeitado pelos seus usuários, pois ele não resolve os seus problemas.

Requisitos funcionais, na maioria das vezes, são especificados em linguagem natural. Por outro lado, requisitos não-funcionais são especificados de forma quantitativa usando-se métricas, como aquelas descritas na próxima tabela.

<b>Requisito</b>	<b>Métrica</b>
<b>Não-Funcional</b>	
Desempenho	Transações por segundo, tempo de resposta, latência, vazão (throughput)
Espaço	Uso de disco, RAM, cache
Confiabilidade	% de disponibilidade, tempo médio entre falhas (MTBF)
Robustez	Tempo para recuperar o sistema após uma falha (MTTR); probabilidade de perda de dados após uma falha
Usabilidade	Tempo de treinamento de usuários
Portabilidade	% de linhas de código portáveis

O uso de métricas evita especificações genéricas, como o sistema deve ser rápido e ter alta disponibilidade. Em vez disso, é preferível definir que o sistema deve ter 99,99% de disponibilidade e que 99% de todas as transações realizadas em qualquer janela de 5 minutos devem ter um tempo de resposta máximo de 1 segundo.

Alguns autores, como Ian Sommerville ([link](#)), também classificam requisitos em **requisitos de usuário** e **requisitos de sistema**. Requisitos de usuários são requisitos de mais alto nível, escritos por usuários, normalmente em linguagem natural e sem entrar em detalhes técnicos. Já requisitos de sistema são técnicos, precisos e escritos pelos próprios desenvolvedores. Normalmente, um requisito de usuário é expandido em um conjunto de requisitos de sistema. Suponha, por exemplo, um sistema bancário. Um requisito de usuário — especificado pelos funcionários do banco — pode ser o seguinte: o sistema deve permitir transferências de valores para uma conta corrente de outro banco, por meio de TEDs. Esse requisito dá origem a um conjunto de requisitos de sistema, os quais vão detalhar e especificar o protocolo a ser usado para realização de tais transferências entre bancos. Portanto, requisitos de usuário estão mais próximos do problema, enquanto que requisitos de sistema estão mais próximos da solução.

## Engenharia de Requisitos

**Engenharia de Requisitos** é o nome que se dá ao conjunto de atividades relacionadas com a descoberta, análise, especificação e manutenção dos requisitos de um sistema. O termo engenharia é usado para reforçar que essas atividades devem ser realizadas de modo sistemático, ao longo de todo o ciclo de vida de um sistema e, sempre que possível, valendo-se de técnicas bem definidas.

As atividades relacionadas com a descoberta e entendimento dos requisitos de um sistema são chamadas de **Elicitação de Requisitos**. Segundo o Dicionário Houaiss, eliciar (ou eliciar) significa fazer sair, expulsar, expelir. No nosso contexto, o termo designa as interações dos desenvolvedores de um sistema com os seus stakeholders, com o objetivo de fazer sair, isto é, descobrir e entender os principais requisitos do sistema que se pretende construir.

Diversas técnicas podem ser usadas para elicitação de requisitos, incluindo entrevistas com stakeholders, aplicação de questionários, leitura de documentos e formulários da organização que está contratando o sistema, realização de workshops com os usuários, implementação de protótipos e análise de cenários de uso. Existem ainda técnicas de elicitação de requisitos

baseadas em estudos etnográficos. O termo tem sua origem na Antropologia, onde designa o estudo de uma cultura em seu ambiente natural (*ethnos*, em grego, significa povo ou cultura). Por exemplo, para estudar uma nova tribo indígena descoberta na Amazônia, o antropólogo pode se mudar para a aldeia e passar meses convivendo com os índios, para entender seus hábitos, costumes, linguagem, etc. De forma análoga, em Engenharia de Requisitos, etnografia designa a técnica de elicitação de requisitos que recomenda que o desenvolvedor se integre ao ambiente de trabalho dos stakeholders e observe — normalmente, por alguns dias — como ele desenvolve suas atividades. Veja que essa observação é silenciosa, isto é, o desenvolvedor não interfere e opina sobre as tarefas e eventos que estão sendo observados.

Após elicitados, os requisitos devem ser: (1) documentados, (2) verificados e validados e (3) priorizados.

No caso de desenvolvimento ágil, a documentação de requisitos é feita de forma simplificada, por meio de **histórias do usuário**, conforme estudamos no Capítulo 2. Por outro lado, em alguns projetos, ainda se exige um **Documento de Especificação de Requisitos**, no qual todos os requisitos do software que se pretende construir — incluindo requisitos funcionais e não-funcionais — são documentados em linguagem natural (português, inglês, etc.). Na década de 90, chegou-se a propor um padrão para Documentos de Especificação de Requisitos, denominado **Padrão IEEE 830**. Ele foi proposto no contexto de Processos Waterfall, isto é, processos que possuem uma longa fase inicial de levantamento de requisitos. As principais seções de um documento de requisitos no padrão IEEE 830 são mostradas na figura da próxima página.

- \* Requisitos Relacionados com Interfaces Externas
  - \* Interfaces com o Usuário
  - \* Interfaces com Hardware
  - \* Interfaces com Outros Sistemas de Software
  - \* Interfaces de Comunicação
- \* Requisitos Funcionais
  - \* Requisito Funcional #1
  - \* Requisito Funcional #2
  - \* ....
- \* Requisitos de Desempenho
- \* Requisitos de Projeto
- \* Outros Requisitos

## Documento de Requisitos no Padrão IEEE 830

Após sua especificação, os requisitos devem ser verificados e validados. O objetivo é garantir que eles estejam corretos, precisos, completos, consistentes e verificáveis, conforme discutido a seguir.

- Requisitos devem estar **corretos**. Um contra-exemplo é a especificação de forma incorreta da fórmula para remuneração das cadernetas de poupança em um sistema bancário. Evidentemente, uma imprecisão na descrição dessa fórmula irá resultar em prejuízos para o banco ou para seus clientes.
- Requisitos devem ser **precisos**, isto é, não devem ser ambíguos. No entanto, ambiguidade ocorre com mais frequência do que gostaríamos quando usamos linguagem natural. Por exemplo, considere essa condição: para ser aprovado um aluno precisa obter 60 pontos no semestre ou 60 pontos no Exame Especial e ser frequente. Veja que ela admite duas interpretações. A primeira é a seguinte: (60 pontos no semestre ou 60 pontos no Exame Especial) e ser frequente. Porém, pode-se interpretar também como: 60 pontos no semestre ou (60 pontos

no Exame Especial e ser frequente). Conforme você observou, tivemos que usar parênteses para eliminar a ambiguidade na ordem das operações e e ou.

- Requisitos devem ser **completos**. Isto é, não podemos esquecer de especificar certos requisitos, principalmente se eles forem importantes no sistema que se pretende construir.
- Requisitos devem ser **consistentes**. Um contra-exemplo ocorre quando um stakeholder afirma que a disponibilidade do sistema deve ser 99,9% e outro considera que 90% já é suficiente.
- Requisitos devem ser **verificáveis**, isto é, deve ser possível testar se os requisitos estão sendo atendidos. Um contra-exemplo é um requisito que apenas requer que o sistema seja amigável. Como os desenvolvedores vão saber se estão atendendo a essa expectativa dos clientes?

Por fim, os requisitos devem ser priorizados. Às vezes, o termo requisitos é interpretado de forma literal, isto é, como uma lista de funcionalidades e restrições obrigatórias em sistemas de software. No entanto, nem sempre aquilo que é especificado pelos clientes será implementado nas releases iniciais. Por exemplo, restrições de prazo e custos podem postergar a implementação de certos requisitos.

Adicionalmente, os requisitos podem mudar, pois o mundo muda. Por exemplo, no sistema bancário que usamos como exemplo, as regras de remuneração das cadernetas de poupança precisam ser atualizadas toda vez que forem modificadas pelos órgãos federais responsáveis pela definição das mesmas. Logo, se existe um documento de especificação de requisitos, documentando tais regras, ele deve ser atualizado, assim como o código fonte do sistema. Chama-se de **rastreabilidade** (*traceability*) a capacidade de dado um trecho de código identificar os requisitos implementados por ele e vice-versa (isto é, dado um requisito, identificar os trechos de código que o implementam).

Antes de concluir, é importante mencionar que Engenharia de Requisitos é uma atividade multidisciplinar e complexa. Por exemplo, fatores políticos

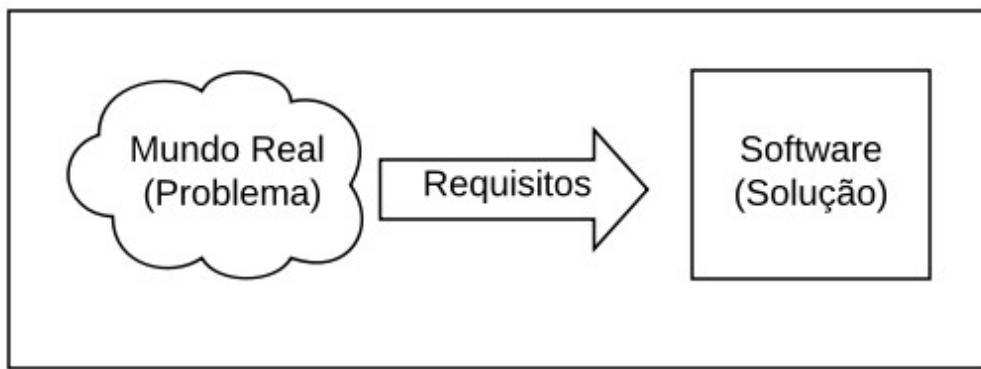
podem fazer com que certos stakeholders não colaborem com a elicitação de requisitos que ameacem seu poder e status na organização. Outros stakeholders simplesmente podem não ter tempo para se reunir com os desenvolvedores, a fim de explicar os requisitos do sistema. A especificação de requisitos pode ser impactada ainda por uma barreira cognitiva entre os stakeholders e desenvolvedores. Devido a essa barreira, os desenvolvedores podem não entender a linguagem e os termos usados pelos stakeholders. Veja que esses últimos tendem a ser especialistas de longa data na área do sistema. Portanto, eles podem se expressar usando uma linguagem muito específica.

**Mundo Real:** Para entender os desafios enfrentados em Engenharia de Requisitos, em 2016, cerca de duas dezenas de pesquisadores coordenaram um survey com 228 empresas que desenvolvem software, distribuídas por 10 países, incluindo o Brasil ([link](#)). Quando questionadas sobre os principais problemas enfrentados na especificação de requisitos, as dez respostas mais comuns foram as seguintes (incluindo o percentual de empresas que apontou cada problema):

- Requisitos incompletos ou não-documentados (48%)
- Falhas de comunicação entre membros do time e os clientes (41%)
- Requisitos em constante mudança (33%)
- Requisitos especificados de forma abstrata (33%)
- Restrições de tempo (32%)
- Problemas de comunicação entre os próprios membros do time (27%)
- Stakeholders com dificuldades de separar requisitos e soluções (25%)
- Falta de apoio dos clientes (20%)
- Requisitos inconsistentes (19%)
- Falta de acesso às necessidades dos clientes ou do negócio (18%)

## O que Vamos Estudar?

A próxima figura resume um pouco o que foi estudado sobre requisitos até agora. Ela mostra que os requisitos são a ponte que liga um problema do mundo real a um sistema de software que o soluciona. Usaremos essa figura para motivar e apresentar os temas que estudaremos no restante deste capítulo.



Requisitos são a “ponte” que liga um problema do mundo real a um sistema de software que o soluciona.

A figura serve para ilustrar uma situação muito comum em Engenharia de Requisitos: sistemas cujos requisitos mudam com frequência ou cujos usuários não sabem especificar com precisão o sistema que querem. Na verdade, já estudamos sobre tais sistemas no Capítulo 2, quando tratamos de Métodos Ágeis. Conforme visto, quando os requisitos mudam frequentemente e o sistema não é de missão crítica, não vale a pena investir anos na elaboração de um Documento Detalhado de Requisitos. Corre-se o risco de quando ele ficar pronto, os requisitos já estarem obsoletos — ou um concorrente já ter construído um sistema equivalente e dominado o mercado. Em tais sistemas, como vimos no Capítulo 2, pode-se adotar documentos simplificados de especificação de requisitos — chamados de **Histórias de Usuários** — e incorporar um representante dos clientes, em tempo integral, ao time de desenvolvimento, para tirar dúvidas e explicar os requisitos para os desenvolvedores. Dada a importância de tais cenários — sistemas cujos requisitos são sujeitos a mudanças, mas não críticas — iremos iniciar com o estudo de Histórias de Usuários na Seção 3.3.

Por outro lado, existem também aqueles sistemas com requisitos mais estáveis. Nesses casos, pode ser importante investir em especificações de requisitos mais detalhadas. Tais especificações podem ser também requisitadas por certas empresas, que preferem contratar o desenvolvimento de um sistema apenas após conhecer todos os seus requisitos. Por último, eles podem ser requisitados por organizações de certificação, principalmente no caso de sistemas que lidam com vidas humanas, como sistemas das áreas médica, de transporte ou militar. Na Seção 3.4, iremos estudar **Casos de Uso**, que são documentos detalhados para especificação de requisitos.

Uma terceira situação é quando não sabemos nem mesmo se o problema que vamos resolver é de fato um problema. Ou seja, podemos até levantar todos os requisitos desse problema e implementar um sistema que o resolva. Porém, não temos certeza de que esse sistema terá sucesso e usuários. Nesses casos, o mais prudente é dar um passo atrás e primeiro testar a relevância do problema que se planeja resolver por meio de um sistema de software. Um possível teste envolve a construção de um **Produto Mínimo Viável (MVP)**. Um MVP é um sistema funcional, mas que possui apenas o conjunto mínimo de funcionalidades necessárias para comprovar a viabilidade de um produto ou sistema. Dada a importância contemporânea de tais cenários — sistemas para resolver problemas em mercados desconhecidos ou incertos — estudaremos mais sobre MVPs na Seção 3.5.

## Histórias de Usuários

Documentos de requisitos tradicionais, como aqueles produzidos quando se usa Waterfall, possuem centenas de páginas e levam às vezes mais de um ano para ficarem prontos. Além disso, eles sofrem dos seguintes problemas: (1) durante o desenvolvimento, os requisitos mudam e os documentos ficam obsoletos; (2) descrições em linguagem natural são ambíguas e incompletas; então os desenvolvedores têm que voltar a conversar com os clientes durante o desenvolvimento para tirar dúvidas; (3) quando essas conversas intermediárias não ocorrem, os riscos são ainda maiores: no final da codificação, o cliente pode simplesmente concluir que esse não é mais o sistema que ele queria, pois suas prioridades mudaram, sua visão do negócio mudou, os processos internos de sua empresa mudaram, etc. Por isso, uma longa fase inicial de especificação de requisitos é cada vez mais rara, pelo

menos em sistemas comerciais, como aqueles que estão sendo tratados neste livro.

Os profissionais da indústria que propuseram métodos ágeis perceberam — ou sofreram com — tais problemas e propuseram uma técnica pragmática para solucioná-los, que ficou conhecida pelo nome de **Histórias de Usuários**. Conforme sugerido por Ron Jeffries em um livro sobre desenvolvimento ágil ([link](#)), uma história de usuário é composta por três partes, todas começando com a letra C e que vamos documentar usando a seguinte equação:

$$\text{História de Usuário} = \text{Cartão} + \text{Conversas} + \text{Confirmação}$$

A seguir, exploramos cada uma dessas partes de uma história:

- **Cartão**, usado pelos clientes para escrever, na sua linguagem e em poucas sentenças, uma funcionalidade que esperam ver implementada no sistema.
- **Conversas** entre clientes e desenvolvedores, por meio das quais os clientes explicam e detalham o que escreveram em cada cartão. Como dito antes, a visão de métodos ágeis sobre Engenharia de Requisitos é pragmática: como especificações textuais e completas de requisitos não funcionam, elas foram eliminadas e substituídas por comunicação verbal entre desenvolvedores e clientes. Por isso, métodos ágeis — conforme estudamos no Capítulo 2 — incluem nos times de desenvolvimento um representante dos clientes, que participa do time em tempo integral.
- **Confirmação**, que é basicamente um teste de alto nível — de novo especificado pelo cliente — para verificar se a história foi implementada conforme esperado. Portanto, não se trata de um teste automatizado, como um teste de unidades, por exemplo. Mas a descrição dos cenários, exemplos e casos de teste que o cliente irá usar para confirmar a implementação da história. Por isso, são também chamados de **testes de aceitação** de histórias. Eles devem ser escritos o

quanto antes, preferencialmente no início de uma iteração. Alguns autores recomendam escrevê-los no verso dos cartões da história.

Portanto, especificações de requisitos por meio de histórias não consistem apenas de duas ou três sentenças, como alguns críticos de métodos ágeis podem afirmar. A maneira correta de interpretar uma história de usuário é a seguinte: a história que se escreve no cartão é um lembrete do representante dos clientes para os desenvolvedores. Por meio dele, o representante dos clientes declara que gostaria de ver um determinado requisito funcional implementado na próxima iteração (ou sprint). Mais ainda, durante todo o sprint ele se compromete a estar disponível para refinar a história e explicá-la para os desenvolvedores. Por fim, ele também se compromete a considerar a história implementada desde que ela satisfaça os testes de confirmação que ele mesmo especificou.

Olhando na perspectiva dos desenvolvedores, o processo funciona assim: o representante dos clientes está nos pedindo a história resumida nesse cartão. Logo, nossa obrigação no próximo sprint é implementá-la. Para isso, poderemos contar com o apoio integral dele para conversar e tirar dúvidas. Além disso, ele já definiu os testes que vai usar na reunião de revisão do sprint (ou sprint review) para considerar a história implementada. Combinamos ainda que ele não pode mudar de ideia e, ao final do sprint, usar um teste completamente diferente para testar nossa implementação.

Resumindo, quando usamos histórias de usuários, atividades de Engenharia de Requisitos ocorrem ao longo de todo o desenvolvimento, em praticamente todos os dias de uma iteração. Consequentemente, troca-se um documento de requisitos com centenas de páginas por conversas frequentes, nas quais o representante dos clientes explica os requisitos para os desenvolvedores da equipe. Prosseguindo na comparação, histórias de usuários favorecem comunicação verbal, em vez de comunicação escrita. E por isso elas são também compatíveis com os princípios do Manifesto Ágil, que reproduzimos a seguir: (1) indivíduos e interações, mais do que processos e ferramentas; (2) software em funcionamento, mais do que documentação abrangente; (3) colaboração com o cliente, mais do que negociação de contratos; (4) resposta a mudanças, mais do que seguir um plano.

Boas histórias devem possuir as seguintes características (cujas iniciais em inglês dão origem ao acrônimo INVEST):

- Histórias devem ser **independentes**: dadas duas histórias X e Y, deve ser possível implementá-las em qualquer ordem. Para isso, idealmente, não devem existir dependências entre elas.
- Histórias devem ser abertas para **negociação**. Frequentemente, costuma-se dizer que histórias (o cartão) são convites para conversas entre clientes e desenvolvedores durante um sprint. Logo, ambos devem estar abertos a ceder em suas opiniões durante essas conversas. Os desenvolvedores devem estar abertos para implementar detalhes que não estão expressos ou que não cabem nos cartões da história. E os clientes devem aceitar argumentos técnicos vindos dos desenvolvedores, por exemplo sobre a inviabilidade de implementar algum detalhe da história conforme inicialmente vislumbrado.
- Histórias devem agregar **valor** para o negócio dos clientes. Histórias são propostas, escritas e priorizadas pelos clientes e de acordo com o valor que elas agregam ao seu negócio. Por isso, não existe a figura de uma história técnica, como a seguinte: o sistema deve ser implementado em JavaScript, usando React no front-end e Node.js no backend.
- Deve ser viável **estimar** o tamanho de uma história. Por exemplo, quantos dias serão necessários para implementá-la. Normalmente, isso requer que a história seja pequena, como veremos no próximo item, e que os desenvolvedores tenham experiência na área do sistema.
- Histórias devem ser **succintas**. Na verdade, até se admite histórias complexas e grandes, as quais são chamadas de **épicos**. Porém, elas ficam posicionadas no final do backlog, o que significa que ainda não se tem previsão de quando elas serão implementadas. Por outro lado, as histórias do topo do backlog e que, portanto, serão implementadas em breve, devem ser succintas e pequenas, para facilitar o entendimento e estimativa das mesmas. Assumindo-se que um sprint tem duração máxima de um mês, deve ser possível implementar as histórias do topo do backlog em menos de uma semana.

- Histórias devem ser **testáveis**, isto é, elas devem ter critérios de aceitação objetivos. Como exemplo, podemos citar: o cliente pode pagar com cartões de crédito. Uma vez definidas as bandeiras de cartões de crédito que serão aceitas, essa história é testável. Por outro lado, a seguinte história é um contra-exemplo: um cliente não deve esperar muito para ter sua compra confirmada. Essa é uma história vaga e, portanto, com um critério de aceitação também vago.

Antes de começar a escrever histórias, recomenda-se listar os principais usuários que vão interagir com o sistema. Assim, evita-se que as histórias fiquem enviesadas e atendam às necessidades de apenas certos usuários. Definidos esses **papéis de usuários** (*user roles*), costuma-se escrever as histórias no seguinte formato:

Como um [papel de usuário], eu gostaria de [realizar algo com o sistema]

Vamos mostrar exemplos de histórias nesse formato na próxima seção. Antes, gostaríamos de comentar que, logo no início do desenvolvimento de um sistema, costuma-se realizar um **workshop de escrita de histórias**. Esse workshop reúne em uma sala representantes dos principais usuários do sistema, que discutem os objetivos do sistema, suas principais funcionalidades, etc. Ao final do workshop, que dependendo do tamanho do sistema pode durar uma semana, deve-se ter em mãos uma boa lista de histórias de usuários, que demandem alguns sprints para serem implementadas.

## Exemplo: Sistema de Controle de Bibliotecas

Nesta seção, vamos mostrar exemplos de histórias para um sistema de controle de bibliotecas. Elas estão associadas a três tipos de usuários: usuário típico, professor e funcionário da biblioteca.

Primeiro, mostramos histórias propostas por usuários típicos (veja a seguir). Qualquer usuário da biblioteca se encaixa nesse papel e, portanto, pode realizar as operações mencionadas nessas histórias. Observe que as histórias são resumidas e não detalham como cada operação será implementada. Por exemplo, uma história documenta que o sistema deve permitir pesquisas por

livros. No entanto, existem diversos detalhes que a história omite, incluindo os campos de pesquisa, os filtros que poderão ser usados, o número máximo de resultados retornados em cada pesquisa, o leiaute das telas de pesquisa e de resultados, etc. Mas lembre-se que uma história é uma promessa: o representante dos clientes promete ter tempo para definir e explicar tais detalhes em conversas com os desenvolvedores, durante o sprint no qual a história será implementada. Conforme já comentado, quando se usa histórias, essa comunicação verbal entre desenvolvedores e representante dos clientes é a principal atividade de Engenharia de Requisitos.

Como usuário típico, eu gostaria de realizar empréstimos de livros

Como usuário típico, eu gostaria de devolver um livro que tomei emprestado

Como usuário típico, eu gostaria de renovar empréstimos de livros

Como usuário típico, eu gostaria de pesquisar por livros

Como usuário típico, eu gostaria de reservar livros que estão emprestados

Como usuário típico, eu gostaria de receber e-mails com novas aquisições

Em seguida, mostramos as histórias propostas por professores:

Como professor, eu gostaria de realizar empréstimos de maior duração

Como professor, eu gostaria de sugerir a compra de livros

Como professor, eu gostaria de doar livros para a biblioteca

Como professor, eu gostaria de devolver livros em outras bibliotecas

É importante mencionar que, de fato, os professores foram os usuários que lembraram de requisitar as histórias acima. Eles podem ter feito isso, por exemplo, em um workshop de escrita de histórias. Mas isso não significa que apenas professores poderão fazer uso dessas histórias. Por exemplo, ao

detalhar as histórias em um sprint, o representante dos clientes (*product owner*) pode achar interessante permitir que qualquer usuário faça doações de livros e não apenas professores. Por fim, a última história sugerida por professores — permitir devoluções em outras bibliotecas da universidade — pode ser considerada como um **épico**, isto é, uma história mais complexa. Como a universidade possui mais de uma biblioteca, o professor gostaria de realizar um empréstimo na Biblioteca Central e devolver o livro na biblioteca do seu departamento, por exemplo. No entanto, essa funcionalidade requer a integração dos sistemas das duas bibliotecas e, também, pessoal disponível para transportar o livro para sua biblioteca original.

Por fim, mostramos as histórias propostas pelos funcionários da biblioteca, durante o workshop de escrita de histórias. Veja que, geralmente, são histórias relacionadas com a organização da biblioteca e também para garantir o seu bom funcionamento.

Como funcionário da biblioteca, eu gostaria de cadastrar novos usuários

Como funcionário da biblioteca, eu gostaria de cadastrar novos livros

Como funcionário da biblioteca, eu gostaria de dar baixa em livros estragados

Como funcionário da biblioteca, eu gostaria de obter estatísticas sobre o acervo

Como funcionário da biblioteca, eu gostaria que o sistema envie e-mails de cobrança para alunos com empréstimos atrasados

Como funcionário da biblioteca, eu gostaria que o sistema aplicasse multas quando da devolução de empréstimos atrasados

Antes de concluir, vamos mostrar um teste de aceitação para a história pesquisar por livros. Para confirmar a implementação dessa história, o representante dos clientes definiu que gostaria de ver as seguintes pesquisas serem realizadas com sucesso. Elas serão demonstradas e testadas durante a reunião de entrega de histórias — chamada de Revisão do Sprint em Scrum.

Pesquisar por livros, informando ISBN

Pesquisar por livros, informando autor; retorna livros cujo autor contém a string de busca

Pesquisar por livros, informando título; retorna livros cujo título contém a string de busca

Pesquisar por livros cadastrados na biblioteca desde uma data até a data atual

**Aprofundamento:** Testes de aceitação devem ser especificados pelo representante dos clientes. Com isso, procura-se evitar o que se denomina de **gold plating**. Em Engenharia de Requisitos, a expressão designa a situação na qual os desenvolvedores decidem, por conta própria, sofisticar a implementação de algumas histórias — ou requisitos, de forma mais genérica —, sem que isso tenha sido pedido pelos clientes. Em uma tradução literal, os desenvolvedores ficam cobrindo as histórias com camadas de ouro, quando isso não irá gerar valor para os usuários do sistema.

## Perguntas Frequentes

Antes de finalizar, e como comum neste livro, vamos responder algumas perguntas sobre histórias de usuários:

**Como especificar requisitos não-funcionais usando histórias?** Essa é uma questão de tratamento mais desafiador quando se usa métodos ágeis. De fato, o representante dos clientes (ou dono do produto) pode escrever uma história dizendo que o tempo de resposta máximo do sistema deve ser de 1 segundo. No entanto, não faz sentido alocar essa história a uma iteração, pois ela deve ser uma preocupação durante todas as iterações do projeto. Por isso, a melhor solução é pedir ao dono do produto para escrever histórias sobre requisitos não-funcionais, mas usá-las principalmente para reforçar os critérios de conclusão de histórias (*done criteria*). Por exemplo, para considerar que uma história esteja concluída ela deverá passar por uma revisão de código que tenha como objetivo detectar problemas de desempenho. Antes de disponibilizar para produção qualquer release do sistema, pode-se também realizar um teste de desempenho, para garantir que

o requisito não-funcional especificado na história esteja sendo atendido. Em resumo, pode-se — e deve-se — escrever histórias sobre requisitos não-funcionais, mas elas não vão para o backlog do produto. Em vez disso, elas são usadas para refinar os critérios de conclusão de histórias.

**É possível criar histórias para estudo de uma nova tecnologia?** Conceitualmente, a resposta é que não se deve criar histórias exclusivamente para aquisição de conhecimento, pois histórias devem sempre ser escritas e priorizadas pelos clientes. E elas devem ter valor para o negócio. Logo, não vale a pena violar esse princípio e permitir que os desenvolvedores criem uma história como estudar o emprego do framework X na implementação da interface Web. Por outro lado, esse estudo pode ser uma tarefa, necessária para implementar uma determinada história. Tarefas para aquisição de conhecimento são chamadas de **spikes**.

## Casos de Uso

**Casos de uso** (*use cases*) são documentos textuais de especificação de requisitos. Como veremos nesta seção, eles incluem descrições mais detalhadas do que histórias de usuários. Recomenda-se que casos de uso sejam escritos na fase de Especificação de Requisitos, considerando que estamos seguindo um processo de desenvolvimento do tipo Waterfall. Eles são escritos pelos próprios desenvolvedores do sistema — às vezes, chamados de Engenheiros de Requisitos durante essa fase do desenvolvimento. Para isso, os desenvolvedores podem se valer, por exemplo, de entrevistas com os usuários do sistema. Apesar de escritos pelos desenvolvedores, casos de uso podem ser lidos, entendidos e validados pelos usuários, antes de as fases de design e implementação terem início.

Casos de uso são escritos na perspectiva de um **ator** que deseja usar o sistema com um objetivo. Tipicamente, esse ator é um usuário humano (embora possa ser um outro sistema de software ou hardware). Ou seja, normalmente, o ator é uma entidade externa ao sistema.

Explicando com mais detalhes, um caso de uso enumera os passos que um ator realiza em um sistema com um determinado objetivo. Na verdade, um caso de uso inclui duas listas de passos. A primeira representa o **fluxo**

**normal** de passos necessários para concluir uma operação com sucesso. Ou seja, o fluxo normal descreve um cenário em que tudo dá certo, às vezes chamado também de fluxo feliz. Já a segunda lista inclui **extensões do fluxo normal**, as quais representam alternativas de execução de um passo normal ou então situações de erro. Ambos os fluxos — normal e extensões — serão posteriormente implementados no sistema. Mostra-se a seguir um caso de uso, referente a um sistema bancário e que especifica uma transferência entre contas, por um cliente do banco.

## **Transferir Valores entre Contas**

**Autor:** Cliente do Banco

### **Fluxo normal:**

- 1 - Autenticar Cliente (sublinhado)
- 2 - Cliente informa agência e conta de destino da transferência
- 3 - Cliente informa valor que deseja transferir
- 4 - Cliente informa a data em que pretende realizar a operação
- 5 - Sistema efetua transferência
- 6 - Sistema pergunta se o cliente deseja realizar uma nova transferência

### **Extensões:**

- 2a - Se conta e agência incorretas, solicitar nova conta e agência
- 3a - Se valor acima do saldo atual, solicitar novo valor
- 4a - Data informada deve ser a data atual ou no máximo um ano a frente
- 5a - Se data informada é a data atual, transferir imediatamente
- 5b - Se data informada é uma data futura, agendar transferência

Vamos agora detalhar alguns pontos pendentes sobre casos de uso, usando o exemplo anterior. Primeiro, todo caso de uso deve ter um nome, cuja primeira palavra deve ser um verbo no infinitivo. Em seguida, ele deve informar o ator principal do caso de uso. Um caso de uso pode também incluir um outro caso de uso. No nosso exemplo, o passo 1 do fluxo normal inclui o caso de uso autenticar cliente. A sintaxe para tratar inclusões é simples: menciona-se o nome do caso de uso a ser incluído, que deve estar sublinhado. A semântica também é clara: todos os passos do caso de uso incluído devem ser executados antes de prosseguir. Ou seja, a semântica é a mesma de macros em linguagens de programação.

Por último, temos as extensões, as quais têm dois objetivos:

- Detalhar algum passo do fluxo normal. No nosso exemplo, usamos extensões para especificar que a transferência deve ser imediatamente realizada se a data informada for a data corrente (extensão 5a). Caso contrário, temos um agendamento da transferência, que vai ocorrer na data futura que foi informada (extensão 5b).
- Tratar erros, exceções, cancelamentos, etc. No nosso exemplo, usamos uma extensão para especificar que um novo valor deve ser solicitado, caso não exista saldo suficiente para a transferência (extensão 3a).

Devido à existência de fluxos de extensão, recomenda-se evitar comandos de decisão (se) no fluxo normal de casos de uso. Quando uma decisão entre dois comportamentos normais for necessária, pense em defini-la como uma extensão. Esse é um dos motivos pelos quais os fluxos de extensão, em casos de uso reais, frequentemente possuem mais passos do que o fluxo normal. No nosso exemplo simples, quase já temos um empate: seis passos normais contra cinco extensões.

Algumas vezes, descrições de casos de uso incluem seções adicionais, tais como: (1) propósito do caso de uso; (2) pré-condições, isto é, o que deve ser verdadeiro antes da execução do caso de uso; (3) pós-condições, isto é, o que deve ser verdadeiro após a execução do caso de uso; e (4) uma lista de casos de uso relacionados.

Para concluir, seguem algumas boas práticas para escrita de casos de uso:

- As ações de um caso de uso devem ser escritas em uma linguagem simples e direta. Escreva casos de uso como se estivesse no início do ensino fundamental é uma sugestão ouvida com frequência. Sempre que possível, use o ator principal como sujeito das ações, seguido de um verbo. Por exemplo, o cliente insere o cartão no caixa eletrônico. Porém, se a ação for realizada pelo sistema, escreva algo como: o sistema valida o cartão inserido.
- Casos de uso devem ser pequenos, com poucos passos, principalmente no fluxo normal, para facilitar o entendimento. Alistair Cockburn, autor de um conhecido livro sobre casos de uso ([link](#)), recomenda que eles devem ter no máximo nove passos no fluxo normal. Ele afirma literalmente o seguinte: eu raramente encontro um caso de uso bem escrito com mais de nove passos no cenário principal de sucesso. Portanto, se você estiver escrevendo um caso de uso e ele começar a ficar extenso, tente quebrá-lo em dois casos de uso menores. Outra alternativa consiste em agrupar alguns passos. Por exemplo, os passos usuário informa login e usuário informa senha podem ser agrupados em usuário informa login e senha.
- Casos de uso não são algoritmos escritos em pseudo-código. O nível de abstração é maior do que aquele necessário em algoritmos. Lembre-se de que os usuários do sistema cujos requisitos estão sendo documentados devem ser capazes de ler, entender e descobrir problemas em casos de uso. Por isso, evite os comandos se, repita até, etc. Por exemplo, em vez de um comando de repetição, você pode usar algo como: o cliente pesquisa o catálogo até encontrar o produto que pretende comprar.
- Casos de uso não devem tratar de aspectos tecnológicos ou de design. Além disso, eles não precisam mencionar a interface que o ator principal usará para se comunicar com o sistema. Por exemplo, não se deve escrever algo como: o cliente pressiona o botão verde para confirmar a transferência. Lembre-se que estamos na fase de documentação de requisitos e que decisões sobre tecnologia, design,

arquitetura e interface com o usuário ainda não estão em nosso radar. O objetivo deve ser documentar o que o sistema deverá fazer e não como ele irá implementar os requisitos especificados.

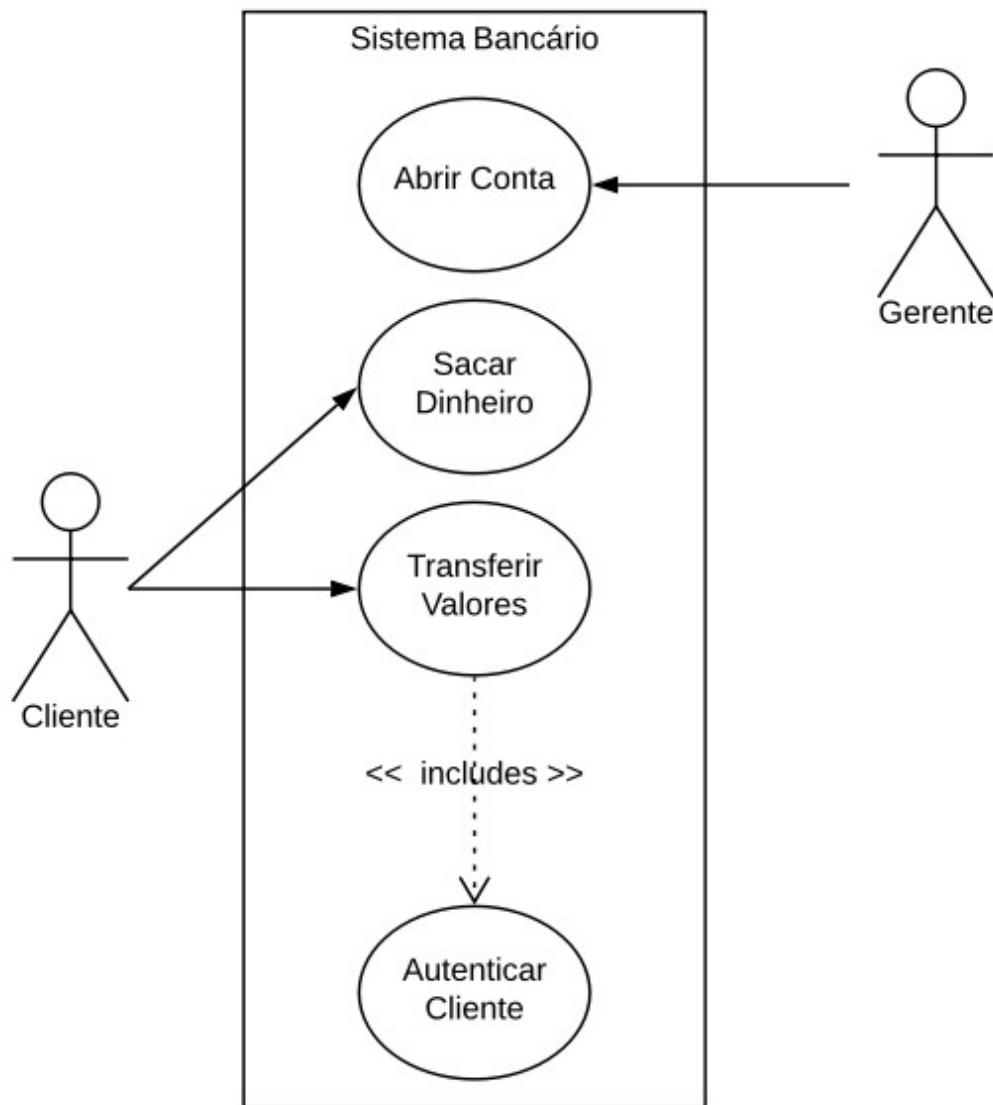
- Evite casos de uso muito simples, como aqueles com apenas operações CRUD (Cadastrar, Recuperar, Atualizar ou *Update* e Deletar). Por exemplo, em um sistema acadêmico não faz sentido ter casos de uso como Cadastrar Professor, Recuperar Professor, Atualizar Professor e Deletar Professor. No máximo, crie um caso de uso Gerenciar Professor e explique brevemente que ele inclui essas quatro operações. Como a semântica delas é clara, isso pode ser feito em uma ou duas sentenças. Aproveitando, gostaríamos de mencionar que não necessariamente o fluxo normal de um caso de uso precisa ser uma enumeração de ações. Em algumas situações, como a que estamos mencionando, é mais prático usar um texto livre.
- Padronize o vocabulário adotado nos casos de uso. Por exemplo, evite usar o nome Cliente em um caso de uso e Usuário em outro. No livro *The Pragmatic Programmer* ([link](#)), David Thomas e Andrew Hunt recomendam a criação de um **glossário**, isto é, um documento que lista os termos e vocabulário usados em um projeto. Segundo os autores, é muito difícil ser bem sucedido em um projeto no qual os usuários e desenvolvedores referem-se às mesmas coisas usando nomes diferentes e, pior ainda, referem-se a coisas diferentes pelo mesmo nome.

## Diagramas de Casos de Uso

No Capítulo 4, vamos estudar a linguagem de modelagem gráfica UML. No entanto, gostaríamos de adiantar e comentar sobre um dos diagramas UML, chamado **Diagrama de Casos de Uso**. Esse diagrama é um índice gráfico de casos de uso. Ele representa os atores de um sistema (como pequenos bonecos) e os casos de uso (como elipses). Mostram-se também dois tipos de relacionamento: (1) ligando ator com caso de uso, que indicam que um ator participa de um determinado caso de uso; (2) ligando dois casos de uso, que indicam que um caso de uso inclui ou estende outro caso de uso.

Um exemplo simples de Diagrama de Caso de Uso para o nosso sistema bancário é mostrado na figura da próxima página. Nele estão representados

dois atores: Cliente e Gerente. Cliente participa dos seguintes casos de uso: Sacar Dinheiro e Transferir Valores. E Gerente é o ator principal do caso de uso Abrir Conta. O diagrama também deixa explícito que Transferir Valores inclui o caso de uso Autenticar Cliente. Por fim, veja que os casos de uso são representados dentro de um retângulo, que delimita as fronteiras do sistema. Os dois atores são representados fora dessa fronteira.



### Exemplo de Diagrama UML de Casos de Uso

**Aprofundamento:** Neste livro, fazemos uma distinção entre casos de uso (documentos textuais para especificar requisitos) e diagramas de caso uso (índices gráficos de casos de uso, conforme proposto em UML). A mesma

decisão é adotada, por exemplo, por Craig Larman, em seu livro sobre UML e padrões de projeto ([link](#)). Ele afirma que casos de uso são documentos textuais e não diagramas. Portanto, a modelagem de casos de uso é essencialmente uma ação de redigir texto e não de desenhar diagramas. E também por Martin Fowler, que chega a afirmar que diagramas UML de caso de uso possuem pouco valor — a importância de casos de uso está no texto, que não é padronizado em UML. Portanto, quando for adotar casos de uso coloque sua energia no texto. Por outro lado, outros autores, para evitar qualquer confusão optam por usar o termo **cenários de uso**, em vez de casos de uso.

## Perguntas Frequentes

Vamos responder agora duas perguntas sobre casos de uso.

**Qual a diferença entre casos de uso e histórias de usuários?** A resposta simples é que casos de uso são especificações de requisitos mais detalhadas e completas do que histórias. Uma resposta mais elaborada é formulada por Mike Cohn em seu livro sobre histórias ([link](#)). Segundo ele, casos de uso são escritos em um formato aceito tanto por clientes como por desenvolvedores, de forma que cada um deles possa ler e concordar com o que está escrito. Portanto, o objetivo é documentar um acordo entre clientes e time de desenvolvimento. Histórias, por outro lado, são escritas para facilitar o planejamento de iterações e para servir como um lembrete para conversas sobre os detalhes das necessidades dos clientes.

**Qual a origem da técnica de casos de uso?** Casos de uso foram propostos no final da década de 80, por Ivar Jacobson, um dos pais da UML e também do Processo Unificado (UP) ([link](#)). Especificamente, casos de uso foram concebidos para ser um dos principais produtos da fase de Elaboração do UP. Conforme dito no Capítulo 2, UP enfatiza comunicação escrita entre usuários e desenvolvedores, usando documentos como casos de uso.

## Produto Mínimo Viável (MVP)

O conceito de MVP foi popularizado no livro **Lean Startup**, de Eric Ries ([link](#)). Por sua vez, o conceito de Lean Startup é inspirado nos princípios de

Manufatura Lean, desenvolvidos por fabricantes japoneses de automóveis, como a Toyota, desde o início dos anos 50. Já comentamos sobre Manufatura Lean no Capítulo 2, pois o processo de desenvolvimento Kanban também foi adaptado de princípios de gerenciamento de produção originados do que ficou conhecido depois como Manufatura Lean. Um dos princípios de Manufatura Lean recomenda eliminar desperdícios em uma linha de montagem ou cadeia de suprimentos. No caso de uma empresa de desenvolvimento de software, o maior desperdício que pode existir é passar anos levantando requisitos e implementando um sistema que depois não vai ser usado, pois ele resolve um problema que não interessa mais a seus usuários. Portanto, se é para um sistema falhar — por não ter sucesso, usuários ou mercado — é melhor falhar rapidamente, pois o desperdício de recursos será menor.

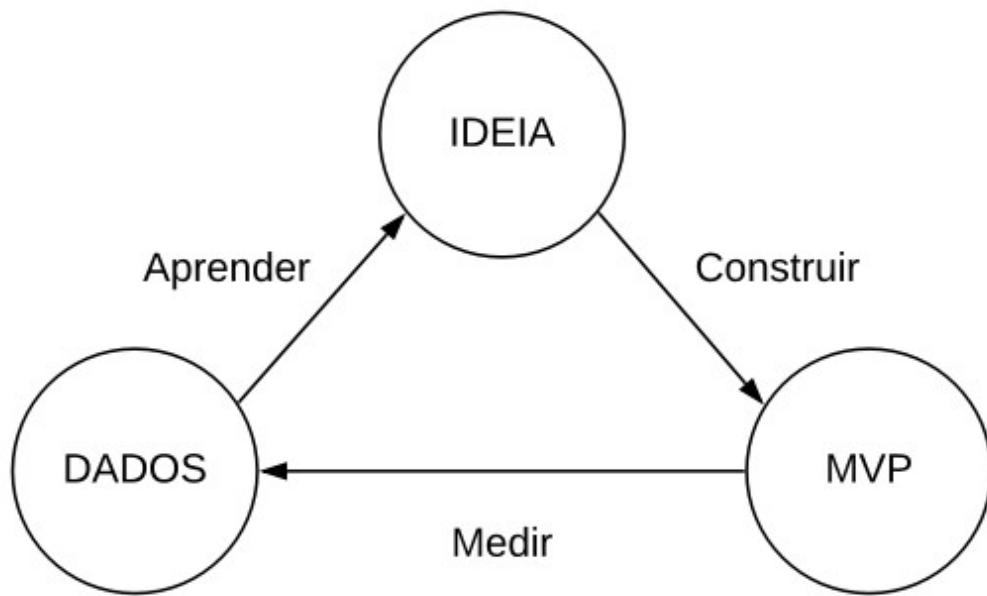
Sistemas de software que não atraem interesse podem ser produzidos por qualquer empresa. No entanto, eles são mais comuns em **startups**, pois por definição elas são empresas que operam em ambientes de grande incerteza. No entanto, Eric Ries também lembra que a definição de startup não se restringe a uma empresa formada por dois universitários que desenvolvem em uma garagem um novo produto de sucesso instantâneo. Segundo ele, qualquer pessoa que está criando um novo produto ou negócio sob condições de extrema incerteza é um empreendedor, quer saiba ou não, e quer trabalhe em uma entidade governamental, em uma empresa apoiada por capital de risco, em uma organização sem fins lucrativos ou em uma empresa com investidores financeiros decididamente voltada para o lucro.

Então, para deixar claro o nosso cenário, suponha que pretendemos criar um sistema novo, mas não temos certeza de que ele terá usuários e fará sucesso. Como comentado acima, não vale a pena passar um ou dois anos levantando os requisitos desse sistema, para então concluir que ele será um fracasso. Por outro lado, não faz muito sentido também realizar pesquisas de mercado, para aferir a receptividade do sistema antes de implementá-lo. Como ele é um sistema novo, com requisitos diferentes de quaisquer sistemas existentes, os resultados de uma pesquisa de mercado podem não ser confiáveis.

Uma solução consiste em implementar um sistema simples, com um conjunto de requisitos mínimos, mas que sejam suficientes para testar a viabilidade de continuar investindo no seu desenvolvimento. Em Lean

Startup, esse primeiro sistema é chamado de **Produto Mínimo Viável (MVP)**. Costuma-se dizer também que o objetivo de um MVP é testar uma hipótese de negócio.

Lean startup propõe um método sistemático e científico para construção e validação de MVPs. Esse método consiste em um ciclo com três passos: **construir, medir e aprender** (veja próxima figura). No primeiro passo (construir), tem-se uma ideia de produto e então implementa-se um MVP para testá-la. No segundo passo (medir), o MVP é disponibilizado para uso por clientes reais com o intuito de coletar dados sobre a sua viabilidade. No terceiro passo (aprender), as métricas coletadas são analisadas e geram o que se denomina de **aprendizado validado** (*validated learning*).



Método Lean Startup para validação de MVPs

O aprendizado obtido com um MVP pode resultar em três decisões:

- Pode-se concluir que ainda são necessários mais testes com o MVP, possivelmente alterando seu conjunto de requisitos, sua interface com os usuários ou o mercado alvo. Logo, repete-se o ciclo, voltando para o passo construir.

- Pode-se concluir que o teste foi bem sucedido e, portanto, achou-se um mercado para o sistema (um *market fit*). Neste caso, é hora de investir mais recursos para implementar um sistema com um conjunto mais robusto e completo de funcionalidades.
- Por fim, pode-se concluir que o MVP falhou, após várias tentativas. Então, restam duas alternativas: (1) perecer, isto é, desistir do empreendimento, principalmente se não existirem mais recursos financeiros para mantê-lo vivo; ou (2) realizar um **pivô**, isto é, abandonar a visão original e tentar um novo MVP, com novos requisitos e para um novo mercado, mas sem esquecer o que se aprendeu com o MVP anterior.

Ao tomar as decisões acima, um risco é usar apenas **métricas de vaidade** (*vanity metrics*). Essas são métricas superficiais que fazem bem para o ego dos desenvolvedores e gerentes de produto, mas que não ajudam a entender e aprimorar uma estratégia de mercado. O exemplo clássico é o número de pageviews em um site de comércio eletrônico. Pode fazer muito bem dizer que o site atrai milhões de clientes por mês, mas somente isso não vai ajudar a pagar as contas do empreendimento. Por outro lado, métricas que ajudam a tomar decisões sobre o futuro de um MVP são chamadas de **métricas açãoáveis** (*actionable metrics*). No caso de um sistema de comércio eletrônico, essas métricas incluiriam o percentual de visitantes que fecham compras, o valor de cada ordem de compra, o número de itens comprados, o custo de captação de novos clientes, etc. Ao monitorar essas métricas, pode-se concluir, por exemplo, que a maioria dos clientes compra apenas um item ao fechar uma compra. Como resultado concreto — ou açãoável — pode-se decidir incorporar um sistema de recomendação ao site ou, então, investigar o uso de um sistema de recomendação mais eficiente. Tais sistemas, dada uma compra em andamento, são capazes de sugerir novos itens para serem comprados. Assim, eles têm o potencial de incrementar o número de itens comprados em uma mesma transação.

Para avaliar MVPs que incluem vendas de produtos ou serviços, costuma-se usar **métricas de funil** (*funnel metrics*), que capturam o nível de interação dos usuários com um sistema. Um funil pode incluir as seguintes métricas:

- Aquisição: número de clientes que visitaram o seu sistema.
- Ativação: número de clientes que criaram uma conta no sistema.
- Retenção: clientes que retornaram ao sistema, após criarem uma conta.
- Receita: número de clientes que fizeram uma compra.
- Recomendação: clientes que recomendaram o sistema para terceiros.

## Exemplos de MVP

Um MVP não precisa ser um software real, implementado em uma linguagem de programação, com bancos de dados, integração com outros sistemas, etc. Dois exemplos de MVP que não são sistemas são frequentemente mencionados nos artigos sobre Lean Startup.

O primeiro é o caso da Zappos, uma das primeiras empresas a tentar vender sapatos pela Internet nos Estados Unidos. Em 1999, para testar de forma pioneira a viabilidade de uma loja de sapatos virtual, o fundador da empresa concebeu um MVP simples e original. Ele visitou algumas lojas de sapatos de sua cidade, fotografou diversos pares de sapato e criou uma página Web bastante simples, por meio da qual os clientes poderiam selecionar os sapatos que desejassem comprar. Porém, todo o processamento era feito de forma manual, incluindo a comunicação com a empresa de cartões de crédito, a compra dos sapatos nas lojas da cidade e a remessa para os clientes. Não existia nenhum sistema para automatizar essas tarefas. No entanto, com esse MVP baseado em tarefas manuais, o dono da Zappos conseguiu validar de forma rápida e barata a sua hipótese inicial, isto é, de que havia mercado para venda de sapatos pela Internet. Anos mais tarde, a Zappos foi adquirida pela Amazon, por mais de um bilhão de dólares.

Um segundo exemplo de MVP que não envolveu a disponibilização de um software real para os usuários vem do Dropbox (o sistema de compartilhamento e armazenamento de arquivos na nuvem). Para receber feedback sobre o produto que estavam desenvolvendo, um dos fundadores da empresa gravou um vídeo simples, quase amador, demonstrando em 3 minutos as principais funcionalidades e vantagens do sistema que estavam

desenvolvendo. O vídeo viralizou e contribuiu para aumentar a lista de usuários interessados em realizar um teste do sistema (de 5 mil para 75 mil usuários). Outro fato interessante é que os arquivos usados no vídeo tinham nomes engraçados e que faziam referência a personagens de histórias em quadrinhos. O objetivo era chamar a atenção de adotantes iniciais (*early adopters*), que são aquelas pessoas aficionadas por novas tecnologias e que se dispõem a serem as primeiras a testar e comprar novos produtos. A hipótese que se queria validar com o MVP em forma de vídeo é que havia usuários interessados em instalar um sistema de sincronização e backup de arquivos. Essa hipótese se revelou verdadeira pela atração de um grande número de adotantes iniciais dispostos a fazer um teste beta do Dropbox.

No entanto, MVPs também podem ser implementados na forma de sistemas de software reais, embora mínimos. Por exemplo, no início de 2018, nosso grupo de pesquisa na UFMG iniciou o projeto de um sistema para catalogar a produção científica brasileira em Ciência da Computação. A primeira decisão foi construir um MVP, cobrindo apenas artigos em cerca de 15 conferências da área de Engenharia de Software. Nessa primeira versão, o código implementado em Python tinha menos de 200 linhas. Os gráficos mostrados pelo sistema, por exemplo, eram planilhas do Google Spreadsheets embutidas em páginas HTML. Esse sistema — inicialmente chamado CoreBR — foi divulgado e promovido em uma lista de e-mails da qual participam os professores brasileiros de Engenharia de Software. Como o sistema atraiu um bom interesse, medido por meio de métricas como duração das sessões de uso, decidimos investir mais tempo na sua construção. Primeiro, seu nome foi alterado para CSIndexbr ([link](#)). Depois, expandimos gradativamente a cobertura para mais 20 áreas de pesquisa em Ciência da Computação e quase duas centenas de conferências. Passamos a cobrir também artigos publicados em mais de 170 periódicos. O número de professores com artigos indexados aumentou de menos de 100 para mais de 900 professores. A interface do usuário deixou de ser um conjunto de planilhas e passou a ser um conjunto de gráficos implementados em JavaScript.

## Perguntas Frequentes

Para finalizar, vamos responder algumas perguntas sobre MVPs.

**Apenas startups devem usar MVPs?** Definitivamente não. Como tentamos discutir nesta seção, MVPs são um mecanismo para lidar com incerteza. Isto é, quando não sabemos se os usuários vão gostar e usar um determinado produto. No contexto de Engenharia de Software, esse produto é um software. Claro que startups, por definição, são empresas que trabalham em mercados de extrema incerteza. Porém, incerteza e riscos também podem caracterizar software desenvolvido por diversos tipos de organização, privadas ou públicas; pequenas, médias ou grandes; e dos mais diversos setores.

**Quando não vale a pena usar MVPs?** De certo modo, essa pergunta foi respondida na questão anterior. Quando o mercado de um produto de software é estável e conhecido, não há necessidade de validar hipóteses de negócio e, portanto, de construir MVPs. Em sistemas de missão crítica, também não se cogita a construção de MVPs. Por exemplo, está fora de cogitação construir um MVP para um software de monitoramento de pacientes de UTIs.

**Qual a diferença entre MVPs e prototipação?** Prototipação é uma técnica conhecida em Engenharia de Software para elicitação e validação de requisitos. A diferença entre protótipos e MVPs está nas três letras da sigla, isto é, tanto no M, como no V e no P. Primeiro, protótipos não são necessariamente sistemas mínimos. Por exemplo, eles podem incluir toda a interface de um sistema, com milhares de funcionalidades. Segundo, protótipos não são necessariamente implementados para testar a viabilidade de um sistema junto aos seus usuários finais. Por exemplo, eles podem ser construídos para demonstrar o sistema apenas para os executivos de uma empresa contratante. Por isso mesmo, eles também não são produtos.

**Um MVP é um produto de baixa qualidade?** Essa pergunta é mais complexa de ser respondida. Porém, é verdade que um MVP deve ter apenas a qualidade mínima necessária para avaliar um conjunto de hipóteses de negócio. Por exemplo, o código de um MVP não precisa ser de fácil manutenção e usar os mais modernos padrões de design e frameworks de desenvolvimento, pois pode ser que o produto se mostre inviável e seja descartado. Na verdade, em um MVP, qualquer nível de qualidade a mais do que o necessário para iniciar o laço construir-medir-aprender é considerado desperdício. Por outro lado, é importante que a qualidade de um MVP não

seja tão ruim a ponto de impactar negativamente a experiência do usuário. Por exemplo, um MVP hospedado em um servidor Web com problemas de disponibilidade pode dar origem a resultados chamados de falsos negativos. Eles ocorrem quando a hipótese de negócio é falsamente invalidada. No nosso exemplo, o motivo do insucesso não estaria no MVP, mas sim no fato de os usuários não conseguirem acessar o sistema, pois o servidor frequentemente estava fora do ar.

## Construindo o Primeiro MVP

Lean startup não define como construir o primeiro MVP de um sistema. Em alguns casos isso não é um problema, pois os proponentes do MVP têm uma ideia precisa de suas funcionalidades e requisitos. Então, eles já conseguem implementar o primeiro MVP e, assim, iniciar o ciclo construir-medir-aprender. Por outro lado, em certos casos, mesmo a ideia do sistema pode não estar clara. Nesses casos, recomenda-se construir um protótipo antes de implementar o primeiro MVP.

**Design Sprint** é um método proposto por Jake Knapp, John Zeratsky e Braden Kowitz para testar e validar novos produtos por meio de protótipos, não necessariamente de software ([link](#)). As principais características de um design sprint — não confundir com um sprint, de Scrum — são as seguintes:

- Time-box. Um design sprint tem a duração de cinco dias, começando na segunda-feira e terminando na sexta-feira. O objetivo é descobrir uma primeira solução para um problema rapidamente.
- Equipes pequenas e multidisciplinares. Um design sprint deve reunir uma equipe multidisciplinar de sete pessoas. Ao definir esse tamanho, o objetivo é fomentar discussões — por isso, a equipe não pode ser muito pequena. Porém, procura-se evitar debates intermináveis — por isso, a equipe não pode também ser muito grande. Da equipe, devem participar representantes de todas as áreas envolvidas com o sistema que se pretende prototipar, incluindo pessoas de marketing, vendas, logística, etc. Por último, mas não menos importante, a equipe deve incluir um tomador de decisões, que pode ser, por exemplo, o próprio dono da empresa.

- Objetivos e regras claras. Os três primeiros dias do design sprint têm como objetivo convergir, depois divergir e, então, convergir novamente. Isto é, no primeiro dia, entende-se e delimita-se o problema que se pretende resolver. O objetivo é garantir que, nos dias seguintes, a equipe estará focada em resolver o mesmo problema (convergência). No segundo dia, possíveis alternativas de solução são propostas, de forma livre (divergência). No terceiro dia, escolhe-se uma solução vencedora, dentre as possíveis alternativas (convergência). Nessa escolha, a última palavra cabe ao tomador de decisões, isto é, um design sprint não é um processo puramente democrático. No quarto dia, implementa-se um protótipo, que pode ser simplesmente um conjunto de páginas HTML estáticas, sem qualquer código ou funcionalidade. No último dia, testa-se o protótipo com cinco clientes reais, com cada um deles usando o sistema em sessões individuais.

Antes de concluir, é importante mencionar que design sprint não é voltado apenas para definição de um protótipo de MVP. A técnica pode ser usada para propor uma solução para qualquer problema. Por exemplo, pode-se organizar um design sprint para reformular a interface de um sistema, já em produção, mas que está apresentando uma alta taxa de abandono.

## Testes A/B

**Testes A/B** (ou *split tests*) são usados para escolher, dentre duas versões de um sistema, aquela que desperta maior interesse dos usuários. As duas versões são idênticas, exceto que uma implementa um requisito A e outra implementa um requisito B, sendo que A e B são mutuamente exclusivos. Ou seja, queremos decidir qual requisito vamos de fato adotar no sistema. Para isso, as versões A e B são liberadas para grupos distintos de usuários. Ao final do teste, decide-se qual versão despertou maior interesse desses usuários. Portanto, testes A/B constituem uma abordagem dirigida por dados para seleção de requisitos (ou funcionalidades) que serão oferecidos em um sistema. O requisito vencedor será mantido no sistema e a versão com o requisito perdedor será descartada.

Testes A/B podem ser usados, por exemplo, quando se constrói um MVP (com requisitos A) e, depois de um ciclo construir-medir-aprender pretende-

se testar um novo MVP (com requisitos B). Um outro cenário muito comum são testes A/B envolvendo componentes de interfaces com o usuário. Por exemplo, dados dois leiautes da página de entrada de um site, um teste A/B pode ser usado para decidir qual resulta em maior engajamento por parte dos usuários. Pode-se testar também a cor ou posição de um botão da interface, as mensagens usadas, a ordem de apresentação de uma lista, etc.

Para aplicar testes A/B, precisamos de duas versões de um sistema, que vamos chamar de **versão de controle** (sistema original, com os requisitos A) e **versão de tratamento** (sistema com novos requisitos B). Para ser mais claro, e usando o exemplo do final da Seção 3.5, suponha que a versão de controle consiste de um sistema de comércio eletrônico que faz uso de um algoritmo de recomendação tradicional e a versão de tratamento consiste do mesmo sistema, mas com um algoritmo de recomendação supostamente mais eficaz. Logo, nesse caso, o teste A/B terá como objetivo definir se o novo algoritmo de recomendação é realmente melhor e, portanto, deve ser incorporado ao sistema.

Para rodar testes A/B, precisamos de uma métrica para medir os ganhos obtidos com a versão de tratamento. Essa métrica é genericamente chamada de **taxa de conversão**. No nosso exemplo, vamos assumir que ela é o percentual de visitas que se convertem em compras por meio de links recomendados. A expectativa é que o novo algoritmo de recomendação aumente esse percentual.

Por fim, precisamos instrumentar o sistema de forma que metade dos clientes use a versão de controle (com o algoritmo tradicional) e a outra metade use a versão de tratamento (com o novo algoritmo de recomendação, que está sendo testado). Além disso, é importante que essa seleção seja aleatória. Ou seja, quando um usuário entrar no sistema, iremos escolher aleatoriamente qual versão ele irá usar. Para isso, podemos modificar a página principal, incluindo o seguinte trecho de código:

```
version = Math.Random(); // número aleatório entre 0 e 1
if (version < 0.5)
    "execute a versão de controle"
else
    "execute a versão de tratamento"
```

Após um certo número de acessos, o teste é encerrado e verificamos se a versão de tratamento, de fato, aumentou a taxa de conversão de usuários. Se sim, passaremos a usá-la em todos os clientes. Se não, continuaremos com a versão de controle.

Uma questão fundamental em testes A/B é a determinação do tamanho da amostra. Em outras palavras, quantos clientes deveremos testar com cada uma das versões. Não iremos nos aprofundar na estatística desse cálculo, pois ela está fora do escopo do livro. Além disso, existem calculadoras de tamanho de amostras de testes A/B disponíveis na Web. No entanto, gostaríamos de mencionar que os testes podem demandar um número extremamente elevado de clientes, que somente estão ao alcance de sistemas populares, como grandes lojas de comércio eletrônico, serviços de busca, redes sociais, portais de notícias, etc. Para dar um exemplo, suponha que a taxa de conversão de clientes seja de 1% e que desejamos verificar se o tratamento introduz um ganho mínimo de 10% nessa taxa. Nesse caso, os grupos de controle e de tratamento devem possuir no mínimo 200 mil clientes, cada um, para que os resultados do teste tenham relevância estatística, considerando um nível de confiança de 95%. Sendo um pouco mais claro:

- Se após 200K acessos, a versão B aumentar a taxa de conversão em pelo menos 10% podemos ter certeza estatística de que esse ganho é causado pelo tratamento B (na verdade, podemos ter 95% de certeza). Logo, dizemos que o teste foi bem sucedido, isto é, ele foi ganho pela versão B.
- Caso contrário, a versão de tratamento B não atingiu os ganhos de conversão pretendidos. Assim, dizemos que o teste A/B falhou.

O tamanho da amostra de um teste A/B diminui bastante quando os testes envolvem eventos com maior taxa de conversão e que testam ganhos de maior proporção. No exemplo anterior, se a taxa de conversão fosse de 10% e a melhoria a ser testada fosse de 25%, o tamanho da amostra cairia para 1.800 clientes, para cada grupo. Esses valores foram estimados usando a calculadora de testes A/B da empresa Optimizely, disponível neste [link](#).

**Aprofundamento:** Em termos estatísticos, um Teste A/B é modelado como um **Teste de Hipótese**. Nesse tipo de teste, partimos de uma Hipótese Nula, que representa o status quo do sistema. Isto é, a Hipótese Nula assume que nada vai mudar e que, portanto, a versão B não é melhor do que a versão atual do sistema. Por outro lado, a hipótese que muda o status quo é chamada de Hipótese Alternativa. Por convenção, a Hipótese Nula é representada por H<sub>0</sub> e a Hipótese Alternativa por H<sub>1</sub>.

Um Teste de Hipótese é um procedimento de decisão que parte do princípio de que H<sub>0</sub> é verdadeira e, em seguida, tenta refutá-la. Para isso, um teste estatístico específico deve ser usado. Porém, esses testes não são totalmente confiáveis. Ou seja, eles sempre trabalham com uma probabilidade de erro. Por exemplo, qualquer que seja o teste, existe uma chance de refutar H<sub>0</sub> mesmo ela sendo verdadeira. Nesses casos, dizemos que ocorreu um Erro do Tipo I ou um falso positivo, pois concluímos indevidamente que a versão B é melhor do que a versão A.

Se erros do Tipo I não podem ser evitados, podemos, pelo menos, ter uma ideia da probabilidade com que eles ocorrem. Mais especificamente, em Testes A/B, existe um parâmetro de entrada, chamado Nível de Significância (*significance level*) e representado pela letra grega  $\alpha$  (alfa). Esse parâmetro define a probabilidade de ocorrência de erros do Tipo I.

Por exemplo, suponha que  $\alpha$  seja definido em 5%. Então, existe uma probabilidade de 5% de rejeitar H<sub>0</sub> indevidamente. No exemplo usado anteriormente nesta seção, em vez de  $\alpha$ , usamos como parâmetro de entrada o valor  $(1 - \alpha)$ , que é a probabilidade de rejeitar H<sub>0</sub> corretamente. Normalmente, esse valor é chamado de Nível de Confiança. Tomamos essa decisão porque  $(1 - \alpha)$  é o parâmetro de entrada mais comum de calculadoras de tamanho de amostras de Testes A/B.

## Perguntas Frequentes

Seguem algumas perguntas e esclarecimentos sobre testes A/B.

**Posso testar mais de duas variações?** Sim, a metodologia que explicamos adapta-se a mais de dois testes. Basta dividir os acessos em três grupos

aleatórios, por exemplo, se quiser testar três versões de um sistema. Esses testes, com mais de um tratamento, são chamados de Testes A/B/n.

**Posso terminar o teste A/B antes, se ele apresentar o ganho esperado?** Não, esse é um erro frequente e grave. Se o tamanho da amostra for de 200 mil usuários, o teste — de cada grupo — somente pode ser encerrado quando alcançarmos exatamente esse número de usuários. Sendo mais preciso, ele não deve terminar antes, com menos usuários, nem depois, com mais usuários. Um possível erro de desenvolvedores quando começam a usar testes A/B consiste em encerrar o teste no primeiro dia em que o ganho mínimo esperado for alcançado, sem testar o resto da amostra.

**O que é um teste A/A?** É um teste no qual os dois grupos, controle e tratamento, executam a mesma versão do sistema. Logo, assumindo-se uma confiança estatística de 95%, eles deveriam quase sempre falhar, pois a versão A não pode ser melhor do que ela mesma. Testes A/A são recomendados para testar e validar os procedimentos e decisões metodológicas que foram tomados em um teste A/B. Alguns autores chegam a recomendar que não se deve iniciar testes A/B antes de realizar alguns testes A/A ([link](#)). Caso os testes A/A não falhem, deve-se depurar o sistema de experimentação até descobrir a causa raiz (*root cause*) que está fazendo com que uma versão A seja considerada melhor do que ela mesmo.

**Qual a origem dos termos grupos de controle e de tratamento?** Os termos têm sua origem na área médica, mais especificamente em experimentos randomizados controlados (*randomized control experiments*). Por exemplo, para lançar uma nova droga no mercado, empresas farmacêuticas devem realizar esse tipo de experimento. São escolhidas duas amostras, chamadas de controle e de tratamento. Os participantes da amostra de controle recebem um placebo e os participantes da amostra de tratamento são tratados com a droga. Após o teste, comparam-se os resultados para verificar se o uso da droga foi efetivo. Experimentos randomizados controlados são um modo cientificamente aceito de provar causalidade. No nosso exemplo, eles podem, por exemplo, provar que a droga testada causou a cura de uma doença.

**Mundo Real:** Testes A/B são usados por todas as grandes empresas da Internet. A seguir, reproduzimos depoimentos de desenvolvedores e

cientistas de três empresas sobre esses testes:

- No Facebook, as inovações que os engenheiros implementam são imediatamente liberadas para uso por usuários reais. Isso permite que os engenheiros comparem cuidadosamente as novas funcionalidades com o caso base (isto é, como o site atual). Testes A/B são uma abordagem experimental para descobrir o que os clientes querem, a qual dispensa eliciar requisitos de forma antecipada e escrever especificações. Além disso, testes A/B permitem detectar cenários nos quais os usuários começam a usar novas funcionalidades de modo inesperado. Dentre outras coisas, isso permite que os engenheiros aprendam com a diversidade de usuários e apreciem as diferentes visões que eles têm do Facebook. ([link](#))
- Na Netflix, os desenvolvedores tratam cada funcionalidade como um experimento, o que faz com que certas funcionalidades possam morrer após serem liberadas para uso. Por exemplo, se um número pequeno de clientes estiver usando um novo elemento [de uma interface com o usuário], um experimento [isto é, um teste A/B] pode ser realizado, incluindo a movimentação do elemento para uma nova posição na tela. Se todos os experimentos falharem, a funcionalidade é removida do sistema. ([link](#))
- Na Microsoft, especificamente no serviço de buscas Bing, o uso de experimentos controlados cresceu exponencialmente ao longo dos anos, com mais de 200 experimentos concorrentes sendo executados a cada dia [dados de 2013]. Consideramos que o Sistema de Experimentos do Bing foi responsável por acelerar a inovação e aumentar a receita da empresa em milhões de dólares, por permitir a descoberta de ideias que foram avaliadas por milhares de experimentos controlados. ([link](#))

## Bibliografia

Mike Cohn. User Stories Applied: For Agile Software Development. Addison-Wesley, 2004.

Alistair Cockburn. Writing Effective Use Cases. Addison-Wesley, 2000.

Eric Ries. *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Business, 2011.

Jake Knapp, John Zeratsky, Braden Kowitz. *Sprint: How to Solve Big Problems and Test New Ideas in Just Five Days*. Simon & Schuster, 2016.

Ian Sommerville. *Engenharia de Software*. Pearson, 10a edição, 2019.

Hans van Vliet. *Software Engineering: Principles and Practice*. Wiley, 2008.

## Exercícios de Fixação

1. [POSCOMP 2010, adaptado] Sobre Engenharia de Requisitos, marque Verdadeiro (V) ou Falso (F).

( ) A Engenharia de Requisitos, como todas as outras atividades de Engenharia de Software, precisa ser adaptada às necessidades do processo, do projeto, do produto e do pessoal que está fazendo o trabalho.

( ) No estágio de levantamento e análise dos requisitos, os membros da equipe técnica de desenvolvimento do software trabalham com o cliente e os usuários finais do sistema para descobrir mais informações sobre o domínio da aplicação, que serviços o sistema deve oferecer, o desempenho exigido do sistema, as restrições de hardware, dentre outras informações.

( ) Na medida em que a informação de vários pontos de vista é coletada, os requisitos emergentes são consistentes.

( ) A validação de requisitos se ocupa de mostrar que estes realmente definem o sistema que o cliente deseja. Ela é importante porque a ocorrência de erros em um documento de requisitos pode levar a grandes custos relacionados ao retrabalho.

2. Cite o nome de pelo menos cinco técnicas para elicitação de requisitos.

3. Quais são as três partes de uma história de usuário? Responda usando o acrônimo 3C's.

4. Suponha uma rede social como o Facebook. (1) Escreva um conjunto de cinco histórias para essa rede, assumindo o papel de um usuário típico; (2) Pense agora em mais um papel de usuário e escreva pelo menos duas histórias para ele.

5. Em Engenharia de Software, *anti-patterns* são soluções não recomendadas para um certo problema. Escreva pelo menos cinco *anti-patterns* para histórias de usuários. Em outras palavras, descreva formatos de histórias que não são recomendados ou que não possuem propriedades recomendáveis.

6. Pense em um sistema e escreva uma história épica para o mesmo.

7. No contexto de requisitos, o que significa a expressão *gold plating*?

8. Escreva um caso de uso para um Sistema de Controle de Bibliotecas (similar ao que usamos para ilustrar a escrita de histórias na Seção 3.3.1).

9. O seguinte caso de uso possui apenas o fluxo normal. Escreva então algumas extensões para ele.

## **Comprar Livro**

**Autor:** Usuário da loja virtual

### **Fluxo normal:**

1. Usuário pesquisa catálogo de livros
2. Usuário seleciona livros e coloca no carrinho de compra
3. Usuário decide fechar a compra
4. Usuário seleciona endereço de entrega
5. Usuário seleciona tipo de entrega
6. Usuário seleciona modo de pagamento
7. Usuário confirma pedido

10. Para cada técnica de especificação e/ou validação de requisitos a seguir, descreva um sistema no qual o seu uso seria mais recomendado: (1) Histórias de Usuários; (2) Casos de Uso; (3) MVPs.

11. Qual a diferença entre um Produto Mínimo Viável (MVP) e o produto obtido na primeira iteração de um método ágil, como XP ou Scrum?

12. O artigo *Failures to be celebrated: an analysis of major pivots of software startups* ([link](#)) apresenta uma discussão sobre quase 50 casos reais de pivôs em startups da área de software. Na Seção 2.3, o artigo apresenta uma classificação de dez tipos de pivô comuns nessas startups. Leia essa parte do artigo, liste pelo menos cinco tipos de pivôs e faça uma breve descrição de cada um deles.

13. Quando começou, a EasyTaxi — a empresa brasileira de aplicativos para solicitação de táxis — construiu um MVP que usava um software muito simples e uma parte operacional realizada de forma manual. Pesquise na Internet sobre esse MVP (basta usar as palavras EasyTaxi e MVP) e faça uma descrição do mesmo.

14. Suponha que estamos em 2008, quando ainda não existia Spotify, e você decidiu criar uma startup para oferecer um serviço de streaming de músicas na Internet. Então, como primeiro passo, você implementou um MVP.

1. Quais seriam as principais funcionalidades desse MVP?
2. Ele seria desenvolvido para qual hardware e sistema operacional?
3. Elabore um rascunho rápido da sua interface com o usuário.
4. Quais métricas você usaria para medir o sucesso/fracasso do MVP?

15. Suponha que você seja responsável por um sistema de comércio eletrônico. Suponha que na versão atual desse sistema (versão A) a mensagem do carrinho de compra seja Adicionar ao Carrinho. Suponha que você pretenda fazer um teste A/B testando a mensagem alternativa Compre Já, a qual vai corresponder à versão B do teste.

1. Qual seria a métrica usada como taxa de conversão nesse teste?

2. Supondo que no sistema original a taxa de conversão seja de 5% e que você deseja avaliar um ganho de 1% com a mensagem da versão B, qual seria o tamanho da amostra que deveria testar em cada uma das versões? Para responder, use uma calculadora de tamanho de amostras de testes A/B, como aquela que citamos na Seção 3.6.

# Cap 4 Modelos

Este capítulo inicia com uma apresentação genérica sobre modelos de software (Seção 4.1). Em seguida, apresentamos uma visão geral sobre UML, que é a notação gráfica mais utilizada para construção de modelos de software (Seção 4.2). Também deixamos claro que vamos estudar UML visando a criação de esboços de software (*sketches*) e não desenhos técnicos detalhados (*blueprints*). Nas seções seguintes, apresentamos quatro diagramas UML com um maior nível de detalhes: Diagramas de Classes (Seção 4.3), Diagramas de Pacotes (Seção 4.4), Diagramas de Sequência (Seção 4.5) e Diagramas de Atividades (Seção 4.6).

## Modelos de Software

Como vimos no capítulo anterior, requisitos documentam o que um sistema deve fazer, valendo-se de um nível de abstração próximo do problema e de seus usuários. Por outro lado, o código fonte é uma representação concreta, de baixo nível e executável do comportamento de um sistema. Portanto, existe uma lacuna entre esses dois mundos: requisitos e código fonte. Para preencher essa lacuna, desde a fundação da área, Engenheiros de Software investem na criação de **modelos**, os quais são criados para ajudar no entendimento e análise de um sistema. Para cumprir essa missão, os modelos usados em Engenharia de Software são mais detalhados do que requisitos, mas ainda menos complexos do que o código fonte de um sistema.

Modelos são largamente usados também em outras engenharias. Por exemplo, uma engenheira civil pode decidir criar uma maquete para mostrar como será a ponte que ela foi contratada para construir. Em seguida, ela pode criar um modelo matemático e físico da ponte e usá-lo para simular e provar propriedades da mesma, tais como carga máxima, resistência a ventos, ondas, terremotos, etc.

Infelizmente, modelos de software — pelo menos até hoje — são menos efetivos do que os modelos matemáticos usados em outras engenharias. O motivo é que ao abstrair detalhes eles também descartam parte da complexidade que é essencial aos sistemas modelados. Frederick Brooks

comenta sobre essa questão em seu ensaio clássico *Não Existe Bala de Prata* ([link](#)):

A complexidade de um software é uma propriedade essencial e não acidental. Portanto, representações de uma entidade de software que abstraem sua complexidade normalmente também abstraem sua essência. Por três séculos, matemáticos e físicos obtiveram grandes avanços construindo modelos simplificados de um fenômeno complexo, derivando propriedades de tais modelos e verificando tais propriedade por meio de experimentos. Esse paradigma funcionou porque as complexidades ignoradas não são propriedades essenciais do fenômeno sob estudo. Porém, essa abordagem não funciona quando as complexidades são essenciais.

A frase que abre esse capítulo, do estatístico britânico George Box, também remete a uma reflexão sobre o uso prático de modelos. Apesar de a frase se referir a modelos matemáticos, ela se aplica a outros modelos, inclusive modelos de software. Segundo Box, todos os modelos são errados, pois são simplificações ou aproximações da realidade. Por isso, a questão principal consiste em avaliar se, apesar dessas simplificações, um modelo continua sendo uma abstração útil para o estudo de alguma propriedade do objeto ou fenômeno que ele modela.

Nesta introdução, estamos procurando calibrar as expectativas associadas ao estudo de modelos de software. Por um lado, como afirmamos, eles não têm a mesma efetividade de modelos em outras Engenharias. Além disso, via de regra, modelos de software não são formalismos matemáticos, mas sim representações gráficas de determinadas dimensões de um sistema de software. Por outro lado, isso não significa dizer que modelos de software são inúteis, a ponto de não merecer um capítulo em um livro sobre práticas de Engenharia de Software Moderna. Se não criarmos expectativas irrealistas, eles podem ter um papel importante no desenvolvimento de sistemas de software, tal como veremos na próxima seção.

Se pensarmos em termos de atividades de desenvolvimento de software, a criação de modelos é considerada uma atividade de projeto (*design*). Durante o levantamento de requisitos, as atenções estão voltadas para a definição do problema que será resolvido pelo sistema. Quando se avança para atividades

de projeto, o problema já deve estar devidamente entendido e as atenções se voltam para a concepção de uma solução capaz de resolvê-lo. Após essa solução ser projetada, ela deve ser implementada, usando-se linguagens de programação, bibliotecas, frameworks, bancos de dados, etc.

Especificamente, neste capítulo, iremos estudar um subconjunto dos diagramas propostos pela UML (*Unified Modelling Language*). Vamos começar descrevendo a história e o contexto que levou à criação da UML. Em seguida, vamos estudar alguns dos principais diagramas UML com mais detalhes.

**Aprofundamento:** Desde a década de 70, pesquisadores têm investigado o uso de modelos matemáticos em Engenharia de Software, por meio do que se chama de **Métodos Formais**. Esses métodos valem-se de uma notação matemática — baseada em lógica, teoria de conjuntos ou Redes de Petri, por exemplo — para derivar **especificações formais** para sistemas de software. Além de serem precisas e não-ambíguas, especificações formais podem ser usadas para provar propriedades de um sistema mesmo antes de sua implementação. Por exemplo, em tese, poderia-se provar que um sistema concorrente não possui deadlocks ou condições de corrida. Pode parecer ambicioso, mas isso ocorre em outras Engenharias. Retomando o exemplo do início da seção, engenheiros civis usam há séculos modelos matemáticos para provar, por exemplo, que uma ponte — antes de ser construída — vai suportar determinada carga e certas condições climáticas. No entanto, o uso de formalismos e especificações matemáticas em Engenharia de Software não avançou como em outras Engenharias. Por isso, eles são pouco usados atualmente, com exceção talvez de alguns sistemas de missão crítica.

## UML

UML é uma notação gráfica para modelagem de software. A linguagem define um conjunto de diagramas para documentar e ajudar no design de sistemas de software, particularmente sistemas orientados a objetos. As origens de UML datam da década de 80, quando o paradigma de orientação a objetos estava amadurecendo e vivendo seu auge. Assim, surgiram diversas linguagens orientadas a objetos, como C++, e também algumas notações gráficas para modelagem de software. Lembre-se que os sistemas

na década de 80 eram desenvolvidos segundo o Modelo Waterfall, que prescreve uma grande e longa fase de design. A proposta de UML era que nessa fase seriam criados modelos gráficos, que depois seriam repassados para os programadores, para serem convertidos em código fonte.

Na verdade, UML é o resultado de um esforço para unificar as notações gráficas que surgiram no final das décadas de 80 e início da década de 90. Especificamente, a primeira versão de UML foi proposta em 1995, como resultado da unificação de notações que estavam sendo desenvolvidas de forma independente por três Engenheiros de Software conhecidos na época: Grady Booch, Jim Rumbaugh e Ivar Jacobson. Nessa época, surgiram também ferramentas para desenhar diagramas UML, as quais foram chamadas de **ferramentas CASE** (*Computer-Aided Software Engineering*). O nome é inspirado em ferramentas CAD (*Computer Aided Design*), usadas para criar modelos para produtos de Engenharia tradicional, como casas, pontes, automóveis, aviões, etc. Por isso, era importante ter uma padronização de UML, de forma que um diagrama criado em uma ferramenta CASE pudesse ser aberto e editado em uma outra ferramenta, de uma empresa diferente. De fato, em 1997, UML passou a ser um padrão gerenciado pela OMG, que é uma organização de padronização financiada por indústrias de software. Desde o início, o desenvolvimento de UML foi comandado por consultores influentes e por grandes empresas de ferramentas ou consultoria, como a Rational, que depois viria a ser comprada pela IBM.

## Como usar UML?

Martin Fowler, em seu livro sobre UML ([link](#)), propõe uma classificação sobre formas de uso dessa linguagem de modelagem. Segundo ele, existem três formas de uso de UML: como blueprint, como linguagem de programação ou como esboço. Vamos descrever cada uma delas nos próximos parágrafos.

**UML como blueprint** corresponde ao uso de UML vislumbrado por seus criadores, ainda na década de 90. Nessa forma de uso, defende-se que, após o levantamento de requisitos, seja produzido um conjunto de modelos — ou plantas técnicas (*blueprints*) — documentando diversos aspectos de um sistema e sempre usando diagramas UML. Esses modelos seriam criados por

analistas de sistemas, usando-se ferramentas CASE e, depois, repassados a programadores para codificação. Logo, UML como *blueprint* é recomendado quando se emprega processos de desenvolvimento do tipo Waterfall ou quando se adota o Processo Unificado (UP). Na verdade, UP foi proposto por pessoas com forte ligação com UML. No entanto, como já discutimos no Capítulo 2, o uso de UML na construção de modelos detalhados e completos é cada vez mais raro. Por exemplo, com métodos ágeis não existe uma longa fase inicial de design (*big design up front*). Em vez disso, decisões de design são tomadas e refinadas ao longo do desenvolvimento, em cada uma das iterações (ou *sprints*). Por isso, não iremos neste capítulo nos aprofundar no uso de UML como *blueprint*.

**UML como linguagem de programação** corresponde ao uso de UML vislumbrado pela OMG, após a padronização da linguagem de modelagem. De forma ambiciosa e pelo menos durante um período, vislumbrou-se a geração de código automaticamente a partir de modelos UML. Em outras palavras, não haveria mais uma fase de codificação, pois o código seria gerado diretamente a partir da compilação de modelos UML. Essa forma de uso é conhecida como **Desenvolvimento Dirigido por Modelos** (*Model Driven Development ou MDD*). Para que MDD fosse viável, UML foi expandida e ganhou novos recursos e diagramas. Foi a partir desse momento que a linguagem ganhou a reputação de ser pesada e complexa. Porém, mesmo com adição de complexidade extra, o uso de UML para geração de código não se tornou comum, pelo menos na grande maioria dos sistemas.

Resta então o terceiro uso, **UML como esboço**, que corresponde à forma que vamos estudar neste capítulo. Nela, usamos UML para construir diagramas leves e informais de partes de um sistema, vindo daí o nome esboço (*sketch*). Esses diagramas são usados para comunicação entre os desenvolvedores, em duas situações principais:

- **Engenharia Avante** (*Forward Engineering*): quando os desenvolvedores usam modelos UML para discutir e analisar alternativas de design, antes que exista qualquer código. Por exemplo, suponha que uma história tenha sido alocada para o sprint corrente. Antes de implementar a história, os desenvolvedores podem se reunir e fazer um esboço das principais classes que deverão ser criadas no

sistema, bem como dos relacionamentos entre elas. O objetivo é validar a proposta de tais classes antes de começar a codificar.

- **Engenharia Reversa** (*Reverse Engineering*): quando os desenvolvedores usam modelos UML para analisar e discutir uma funcionalidade que já se encontra implementada no código fonte. Por exemplo, um desenvolvedor mais experiente pode desenhar alguns diagramas UML para explicar para um desenvolvedor recém-contratado como uma funcionalidade está implementada. Normalmente, é mais fácil conduzir essa explicação usando modelos e diagramas gráficos do que analisar e explicar cada linha de código. Ou seja, aplica-se aqui o ditado segundo o qual uma figura vale mais do que mil palavras.

Nas duas situações, o objetivo não é gerar modelos completos e detalhados. Por isso, não se considera o uso de ferramentas complexas e caras, como ferramentas CASE. Muito menos se cogita a geração automática de código a partir desses esboços. Muitas vezes, os diagramas são desenhados em um quadro e, depois, fotografados e apagados. Adicionalmente, usa-se apenas um subconjunto dos diagramas UML.

Como os esboços são pequenos e informais, pode-se questionar a necessidade de uma linguagem padronizada nos cenários que mencionamos. No entanto, consideramos que é melhor usar uma notação existente há anos, mesmo que de forma parcial, do que inventar uma notação própria. Especificamente, o emprego de UML como esboço contribui para evitar dois extremos. Por um lado, ele não assume o emprego rígido, detalhado e sistemático de UML. Por outro lado, evita-se o uso de uma notação informal e *ad hoc*, cuja semântica pode não ser clara para todos os desenvolvedores. Além disso, UML costuma ser usada em livros, tutoriais e documentos que explicam o uso de frameworks ou técnicas de programação. Por exemplo, no Capítulo 6, usaremos diagramas UML para ilustrar o funcionamento de alguns padrões de projeto. Caso o leitor não tenha tido contato com UML, pode ser que ele tenha dificuldade para entender o conceito que está sendo explicado.

Sintetizando a descrição que acabamos de fazer, modelos de software, como diagramas UML, são usados para comunicação entre desenvolvedores. Ou seja, eles são escritos por e para desenvolvedores. Trata-se de uma diferença

importante para documentos de requisitos, que, conforme vimos no capítulo anterior, são escritos por desenvolvedores, mas de forma que eles possam ser lidos e verificados pelos usuários finais do sistema.

**Mundo Real:** No segundo semestre de 2013, Sebastian Baltes e Stephan Diehl — ambos pesquisadores da Universidade de Trier, na Alemanha — pediram 394 desenvolvedores para responder um questionário sobre o emprego de esboços (*sketches*) em atividades de projeto de software ([link](#)). Esses desenvolvedores estavam distribuídos por mais de 32 países, embora a maioria fosse da Alemanha (54%). A análise das respostas obtidas revelou resultados interessantes sobre o uso de esboços em atividades de projeto e desenvolvimento de software, conforme descrito a seguir:

- 24% dos desenvolvedores que participaram da pesquisa criaram o último esboço no mesmo dia em que responderam ao questionário e 39% no intervalo de tempo máximo de uma semana, antes da resposta. Portanto, esses percentuais indicam que esboços são criados com frequência por desenvolvedores de software.
- 58% dos últimos esboços criados pelos participantes foram depois arquivados, seja em papel (6%), digitalmente (42%) ou de ambas as formas (10%). Isso sugere que os desenvolvedores consideram que os esboços carregam informação importante, que talvez seja útil no futuro.
- 40% dos esboços foram feitos em papel, 18% em quadros e 39% em computadores.
- 52% dos esboços foram feitos para ajudar no projeto (*design*) da arquitetura do sistema, 48% para ajudar no projeto de novas funcionalidades, 46% para explicar alguma tarefa para um outro desenvolvedor, 45% para analisar requisitos e 44% para ajudar no entendimento de uma tarefa. A soma dos percentuais ultrapassa 100% porque os participantes podiam marcar mais de uma resposta.
- 48% dos esboços continham algum elemento de UML e 9% eram integralmente baseados em diagramas UML. Portanto, esses percentuais reforçam a importância de estudar UML, não como notação

para documentação detalhada de sistemas (*blueprints*), mas para ajudar na construção de modelos informais e parciais.

## Diagramas UML

Os diagramas UML são classificados em dois grandes grupos:

- **Diagramas Estáticos (ou Estruturais)** modelam a estrutura e organização de um sistema, incluindo informações sobre classes, atributos, métodos, pacotes, etc. Neste capítulo, vamos estudar dois diagramas estáticos: Diagramas de Classes e Diagramas de Pacotes.
- **Diagramas Dinâmicos (ou Comportamentais)** modelam eventos que ocorrem durante a execução de um sistema. Por exemplo, eles podem modelar uma sequência de chamadas de métodos. Neste capítulo, vamos estudar dois diagramas dinâmicos: Diagramas de Sequência e Diagramas de Atividades.

Para entender melhor a diferença entre esses grupos de diagramas, diagramas estáticos lidam apenas com informações que estão disponíveis, por exemplo, quando da compilação do código resultante dos modelos. Essa visão é estática porque ela não muda, a não ser que sejam realizadas mudanças nos modelos. Já os diagramas dinâmicos fornecem uma visão de tempo de execução. Eles são dinâmicos porque é comum ter execuções diferentes de um mesmo programa. Por exemplo, os usuários podem executar o programa com entradas diferentes, selecionar opções e menus diferentes, etc. Em resumo, se estiver interessado em modelar a estrutura de um programa, você deve usar diagramas estáticos. Se seu interesse for modelar o comportamento de um programa — isto é, o que pode acontecer durante sua execução, quais métodos são de fato executados, etc. — você deve usar algum diagrama dinâmico da UML. Por fim, gostaríamos de lembrar que tratamos de Diagramas de Casos de Uso no Capítulo 3, quando apresentamos técnicas para especificação de requisitos.

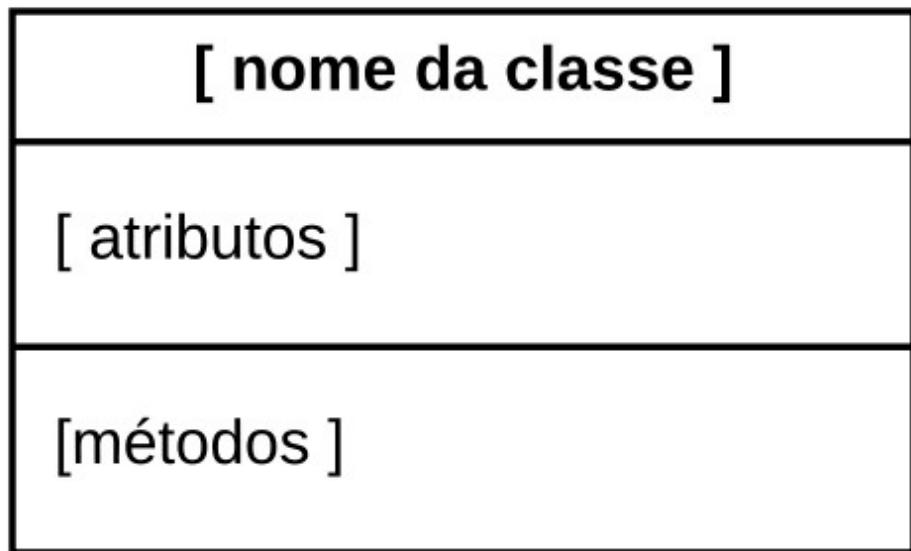
**Aviso:** Existem diversas versões de UML. No restante deste capítulo vamos usar a **versão de UML que é adotada na 3a edição do livro UML Distilled, de Martin Fowler** ([link](#)). Esse livro foi o primeiro trabalho a

discutir o uso de UML como esboço (*sketches*). Na verdade, vamos estudar um pequeno subconjunto da versão 2.0. Além de tratar de apenas quatro diagramas, não vamos cobrir todos os recursos de cada um deles. O nosso desafio ao escrever este capítulo foi selecionar os 20% (ou menos) dos recursos de UML que são responsáveis por 80% (ou mais) de seu uso prático nos dias de hoje. Para se ter uma ideia do nível de detalhe alcançado por UML, a especificação da versão mais recente da linguagem — versão 2.5.1, no momento da escrita deste capítulo — possui 796 páginas. Ela pode ser encontrada no site da OMG ([link](#)).

## Diagrama de Classes

Diagramas de classes são os diagramas mais usados da UML. Eles oferecem uma representação gráfica para um conjunto de classes, provendo informações sobre atributos, métodos e relacionamentos que existem entre as classes modeladas.

Um diagrama de classes é desenhado usando-se retângulos e setas. Cada uma das classes é representada por meio de um retângulo com três compartimentos, conforme mostra a figura a seguir. Esses compartimentos contêm o nome da classe (normalmente, em negrito), seus atributos e métodos, como também ilustrado a seguir:



Mostra-se a seguir um diagrama com duas classes: Pessoa e Fone.

Pessoa	Fone
<ul style="list-style-type: none"> <li>- nome: String</li> <li>- sobrenome: String</li> <li>- fone: Fone</li> </ul>	<ul style="list-style-type: none"> <li>- codigo: String</li> <li>- numero: String</li> <li>- celular: Boolean</li> </ul>
<ul style="list-style-type: none"> <li>+ setPessoa(nome, sobrenome, fone)</li> <li>+ getPessoa(): Pessoa</li> </ul>	<ul style="list-style-type: none"> <li>+ setFone(codigo, numero, celular)</li> <li>+ getFone(): String</li> <li>+ isCelular(): Boolean</li> </ul>

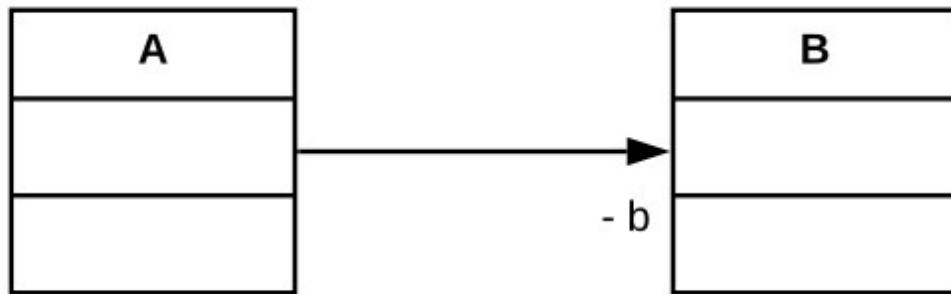
Nesse diagrama, pode-se conferir que a classe Pessoa tem três atributos — nome, sobrenome e fone — e dois métodos — setPessoa e getPessoa. Os três atributos são privados, conforme indicado pelo sinal - antes de cada um. Informa-se também o tipo de cada atributo. Por sua vez, os dois métodos são públicos, conforme indicado pelo sinal +. O diagrama possui ainda uma segunda classe, chamada Fone, com três atributos privados — codigo, numero e celular — e três métodos públicos — setFone, getFone e isCelular. No caso dos métodos, informamos também o nome de seus parâmetros e o tipo de retorno.

Porém, se fosse somente isso, os diagramas dariam a impressão de que as classes de um sistema são ilhas sem comunicação entre si. No entanto, um

dos principais objetivos de diagramas de classe é mostrar visualmente os relacionamentos que existem entre as classes de um sistema. Por isso, eles incluem também linhas e setas, as quais são usadas para representar três tipos de relacionamentos: **associação**, **herança** e **dependência**. Vamos tratar de cada um deles nos próximos parágrafos.

## Associações

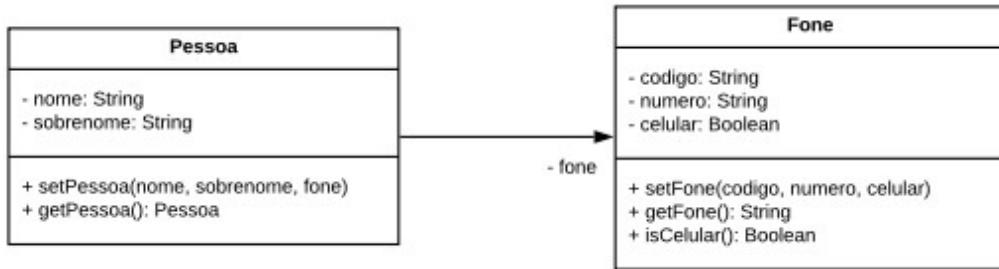
Quando uma classe A possui um atributo b de um tipo B, dizemos que existe uma associação de A para B, a qual é representada por meio de uma seta, também de A para B. Na extremidade da seta, informa-se o nome do atributo de A responsável pela associação — no nosso caso, b. Veja o exemplo abaixo (nele, só mostramos as informações que nos interessam; por isso, o compartimento de atributos e métodos está vazio):



Para ficar ainda mais claro, vamos mostrar como seria o código das classes A e B:

```
class A {  
    ...  
    private B b;  
    ...  
}  
  
class B {  
    ...  
}
```

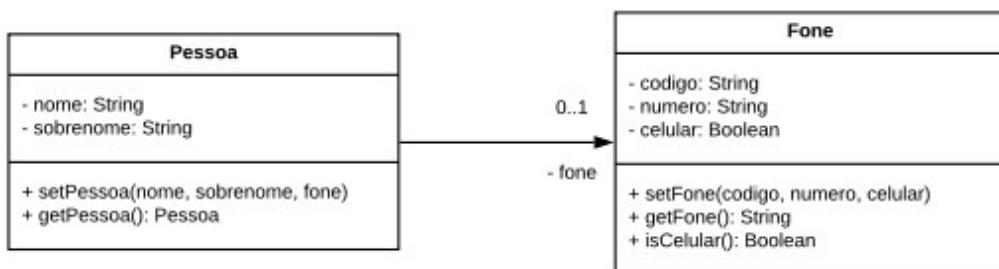
Portanto, usando associações, podemos transformar o primeiro diagrama que mostramos nesta seção, com as classes Pessoa e Fone, no seguinte diagrama:



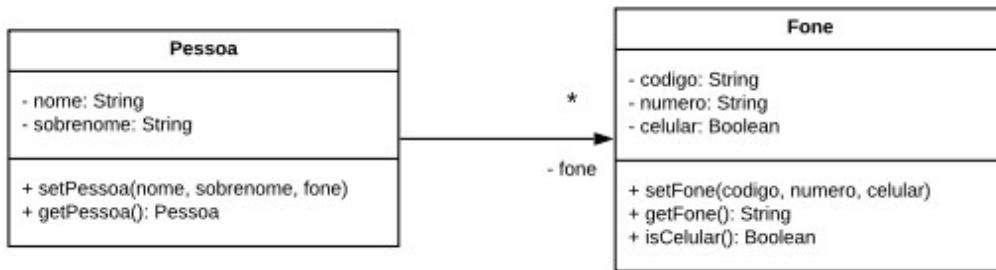
As duas versões do diagrama são semanticamente idênticas. A diferença é que na primeira versão as classes aparecem ilhadas. Já na segunda versão, mostrada acima, fica visualmente claro que existe uma associação de **Pessoa** para **Fone**. Reforçando, em ambos diagramas, **Pessoa** tem um atributo **fone** do tipo **Fone**. Porém, na primeira versão, esse atributo é mostrado dentro do compartimento de atributos da classe **Pessoa**. Já na segunda versão, ele é apresentado fora desse compartimento. Mais especificamente, na extremidade da seta que liga **Pessoa** a **Fone**. O objetivo é deixar claro que o atributo pertence a **Pessoa**, mas ele aponta para um objeto do tipo **Fone**.

Frequentemente, associações incluem informações de **multiplicidade**, que indicam quantos objetos podem estar associados ao atributo responsável pela associação. As informações de multiplicidade mais comuns são as seguintes: 1 (exatamente um objeto), 0..1 (zero ou um objeto) e \* (zero ou mais objetos).

No próximo exemplo, incluímos informação sobre a multiplicidade da associação entre **Pessoa** e **Fone**, que no caso definimos como sendo 0..1. Essa informação consta acima do nome do atributo responsável pela associação, no caso, `fone`. E ela explicita que uma **Pessoa** pode ter zero ou um único telefone. Usando termos de programação, o atributo `fone` de **Pessoa** pode ter o valor `null`, isto é, a **Pessoa** em questão não tem **Fone** associado. Ou então ela pode se associar a um único objeto do tipo **Fone**.



No próximo exemplo, a semântica já é diferente. Nesse caso, uma Pessoa pode estar associada a múltiplos objetos do tipo Fone, inclusive a nenhum. Essa multiplicidade é representada pelo \* que adicionamos logo acima da seta da associação.



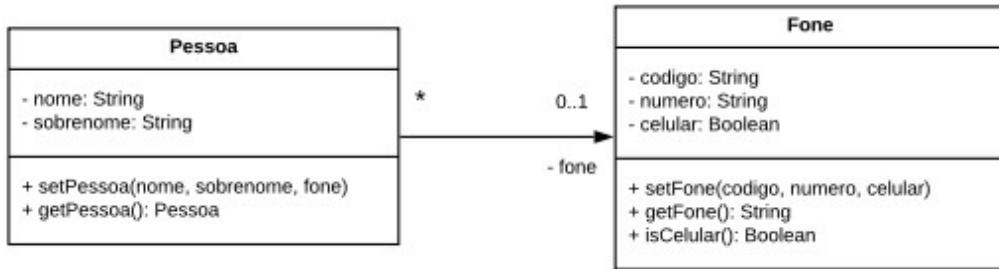
Neste segundo exemplo com informações de multiplicidade, o tipo do atributo fone deve ser um vetor de Fone. Para que fique claro, mostramos o código das classes a seguir:

```

class Pessoa {
    private Fone[] fone;
    ...
}
class Fone {
    ...
}
  
```

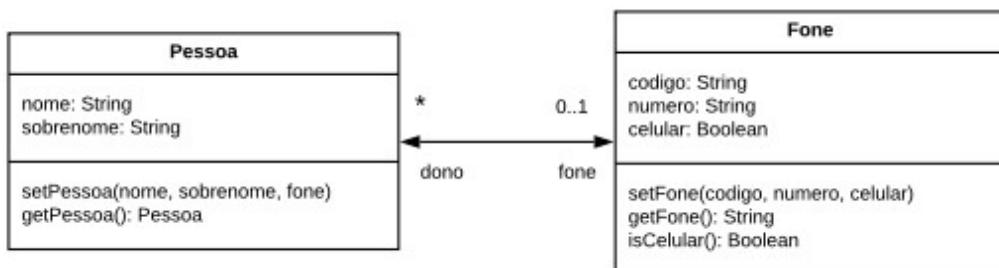
Talvez o leitor possa se perguntar o que é mais correto: uma Pessoa ter no máximo um Fone (isto é, 0..1) ou uma Pessoa ter qualquer quantidade de Fone (isto é, \*)? A resposta é simples: depende dos requisitos do sistema. Ou seja, quem deve responder a essa pergunta são os usuários do sistema que estamos modelando. Para nós, o que importa é que diagramas de classe são capazes de modelar qualquer um dos dois cenários.

Em alguns casos, informações de multiplicidade são também mostradas na extremidade contrária da seta, como no exemplo abaixo:



Nesse diagrama, a multiplicidade `0..1` — da extremidade com a seta — indica que uma **Pessoa** pode ter zero ou um único **Fone**. Mas o mais importante é explicar a multiplicidade que foi adicionada na extremidade oposta da seta, isto é, a multiplicidade `*`. Ela indica que um **Fone** pode estar associado a mais de uma **Pessoa**. Em outras palavras, duas pessoas, distintas, podem compartilhar o mesmo objeto do tipo **Fone**. No entanto, a associação continua sendo unidirecional, isto é, **Pessoa** tem um atributo `fone` que representa o seu **Fone**. Porém, **Fone** não possui um atributo para armazenar as diversas pessoas a que ele pode estar associado. Tentando ser mais claro, dada uma **Pessoa** pode-se recuperar o seu **Fone**. Para isso, basta acessar o atributo `fone`. Mas dado um **Fone** não é possível saber, pelo menos via atributos, a quais objetos do tipo **Pessoa** ele está associado.

Para concluir, suponha que seja importante navegar nos dois sentidos da associação, isto é, de **Pessoa** para **Fone** e também de **Fone** para **Pessoa**. A solução para essa exigência é simples: basta tornar a **associação bidirecional**, isto é, adicionar uma seta em cada extremidade da linha que conecta as classes, como mostrado no próximo diagrama.



Para não deixar dúvidas sobre a semântica de uma associação bidirecional, mostramos também o código das duas classes:

```
class Pessoa {  
    ...  
    private Fone fone;  
    ...  
}  
  
class Fone {  
    ...  
    private Pessoa[] dono;  
    ...  
}
```

Nesse código, Pessoa possui um atributo privado fone do tipo Fone, que pode ser null; com isso, satisfazemos a extremidade 0..1 da associação bidirecional. Por outro lado, Fone possui um vetor privado, de nome dono, que referencia objetos do tipo Pessoa; assim, satisfazemos a extremidade \* da mesma associação.

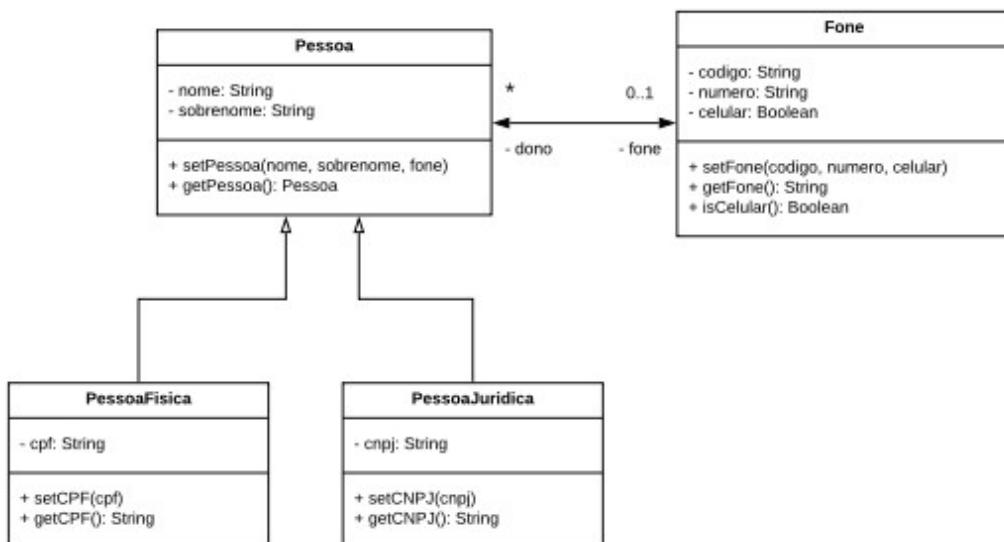
No último diagrama de classes, omitimos todos os símbolos de visibilidade, tanto pública (+) como privada (-). Isso foi feito, de forma deliberada, para destacar que estamos tratando do uso de UML para criação de esboços, quando os diagramas são criados para discutir e ilustrar uma ideia de projeto. Logo, nesse contexto, não faz sentido exigir que os diagramas sejam sintaticamente perfeitos. Por isso, pequenos erros ou omissões são tolerados, principalmente quando não há prejuízo para o propósito do diagrama.

**Aprofundamento:** UML — dependendo da versão que está sendo usada — admite notações diferentes para associações. Por exemplo, algumas vezes, informa-se um nome para a associação, o qual é mostrado logo acima e ao longo da seta que une as duas classes. Outras vezes, no caso de associações bidirecionais, as duas setas são omitidas — pois a padronização de UML define o seguinte: uma associação em que nenhuma das extremidades é marcada com uma seta de navegabilidade é navegável em ambas as direções. No entanto, essas notações alternativas tendem a ser confusas ou mesmo ambíguas. Por exemplo, Gonzalo Génova e mais dois pesquisadores da Universidade de Madrid, na Espanha, fazem a seguinte observação sobre o uso de associações bidirecionais sem setas: infelizmente, isso pode introduzir ambiguidade na notação gráfica, porque não conseguimos mais distinguir entre associações bidirecionais e associações sem especificação de navegabilidade em uma de suas extremidades. ([link](#), Seção 3, quarto

parágrafo). Existem ainda dois conceitos frequentemente mencionados quando estudamos associações em UML: composição e agregação. Composição é uma relação na qual a classe de destino *não* pode existir de forma independente da classe de origem. Por outro lado, quando as duas classes têm ciclos de vida independentes, temos uma relação de agregação. No entanto, na prática, esses conceitos também geram confusão e, por isso, resolvemos não os incluir na explicação sobre diagramas de classes. A mesma opinião é compartilhada por outros autores. Por exemplo, Fowler afirma que agregação é algo estritamente sem sentido; portanto, eu recomendo que você ignore esse conceito em seus diagramas ([link](#), página 68).

## Herança

Em diagramas de classes, relações de herança são representadas por meio de setas com a extremidade não preenchida. Essas setas são usadas para conectar subclasses à sua classe base. No próximo exemplo, elas indicam que PessoaFisica e PessoaJuridica são subclasses de Pessoa. Como usual em orientação a objetos, subclasses herdam todos os atributos e métodos da classe base, mas também podem adicionar novos membros. Por exemplo, apenas PessoaFisica tem cpf e apenas PessoaJuridica tem cnpj.



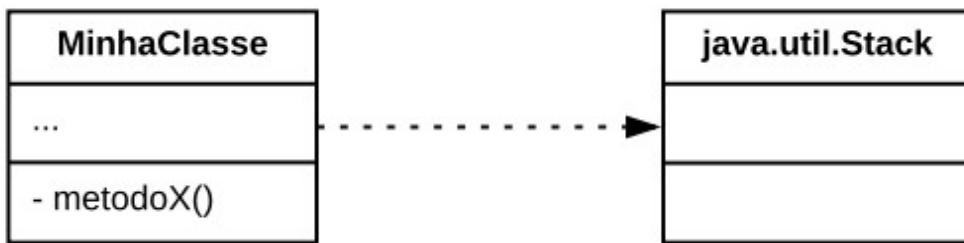
## Dependências

Existe uma dependência de uma classe A para uma classe B, representada por uma seta com uma linha tracejada de A para B, quando a classe A usa a classe B, porém esse uso não ocorre por meio de associação (isto é, A não tem um atributo do tipo B) ou herança (isto é, A não é uma subclasse de B). Dependências ocorrem, por exemplo, quando um método de A declara um parâmetro ou variável local do tipo B ou quando um método de A lança uma exceção do tipo B. Uma dependência é considerada uma modalidade menos forte de relacionamento entre classes do que relacionamentos que ocorrem por meio de associação e herança.

Para ilustrar o uso de dependências, considere o seguinte trecho de código:

```
import java.util.Stack;  
  
class MinhaClasse {  
    ...  
    private void metodoX() {  
        Stack stack = new Stack();  
        ...  
    } ...  
}
```

Observe que o `metodoX` de `MinhaClasse` possui uma variável local do tipo `java.util.Stack`. Nesse caso, dizemos que existe uma dependência de `MinhaClasse` para `java.util.Stack`, a qual é modelada da seguinte forma:



Algumas vezes, logo acima e ao longo da seta tracejada, informa-se o tipo da dependência, usando-se palavras como `create` (para indicar que a classe de origem instancia objetos da classe de destino da dependência) ou `call` (para indicar que a classe de origem chama métodos da classe de destino). Essas palavras são escritas entre sinais de menor (<<) e maior (>>). No diagrama a

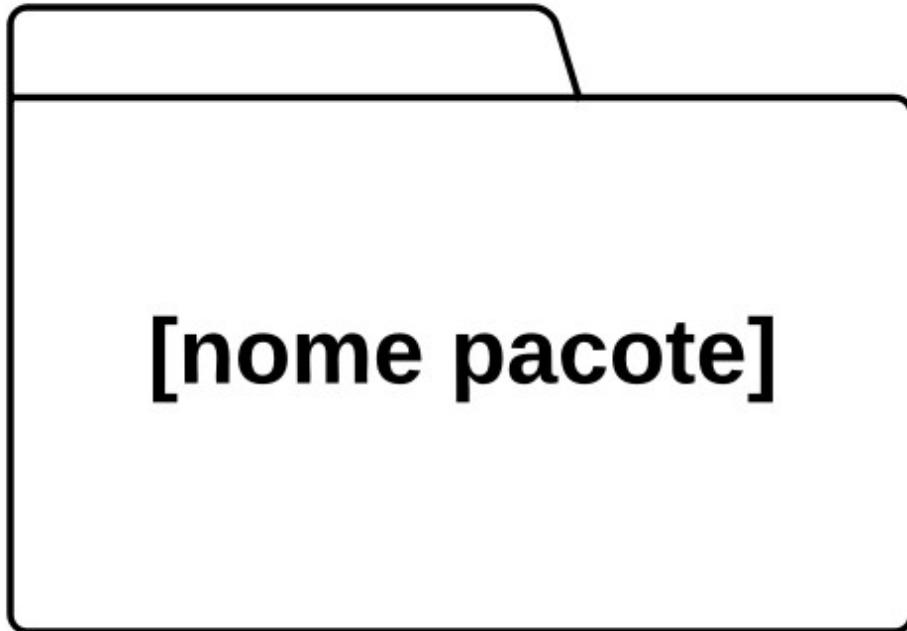
seguir, por exemplo, fica claro o tipo de dependência que ShapeFactory estabelece com a classe Shape.



Uma classe pode ter dependências para um grande número de classes. No entanto, não se costuma representar todas elas em diagramas de classes, mas apenas as mais importantes e que estão diretamente relacionadas com a funcionalidade ou propriedade do sistema que pretendemos esboçar.

## Diagrama de Pacotes

Diagramas de pacotes são recomendados quando se pretende oferecer um modelo de mais alto nível de um sistema, que mostre apenas grupos de classes — isto é, pacotes — e as dependências entre eles. Para isso, UML define um retângulo especial para representar pacotes, mostrado abaixo:



Ao contrário dos retângulos de classes, o retângulo de pacotes inclui apenas o nome do pacote (em negrito). Ele possui ainda um detalhe na parte de cima, na forma de um trapézio, para melhor diferenciá-lo dos retângulos de classe.

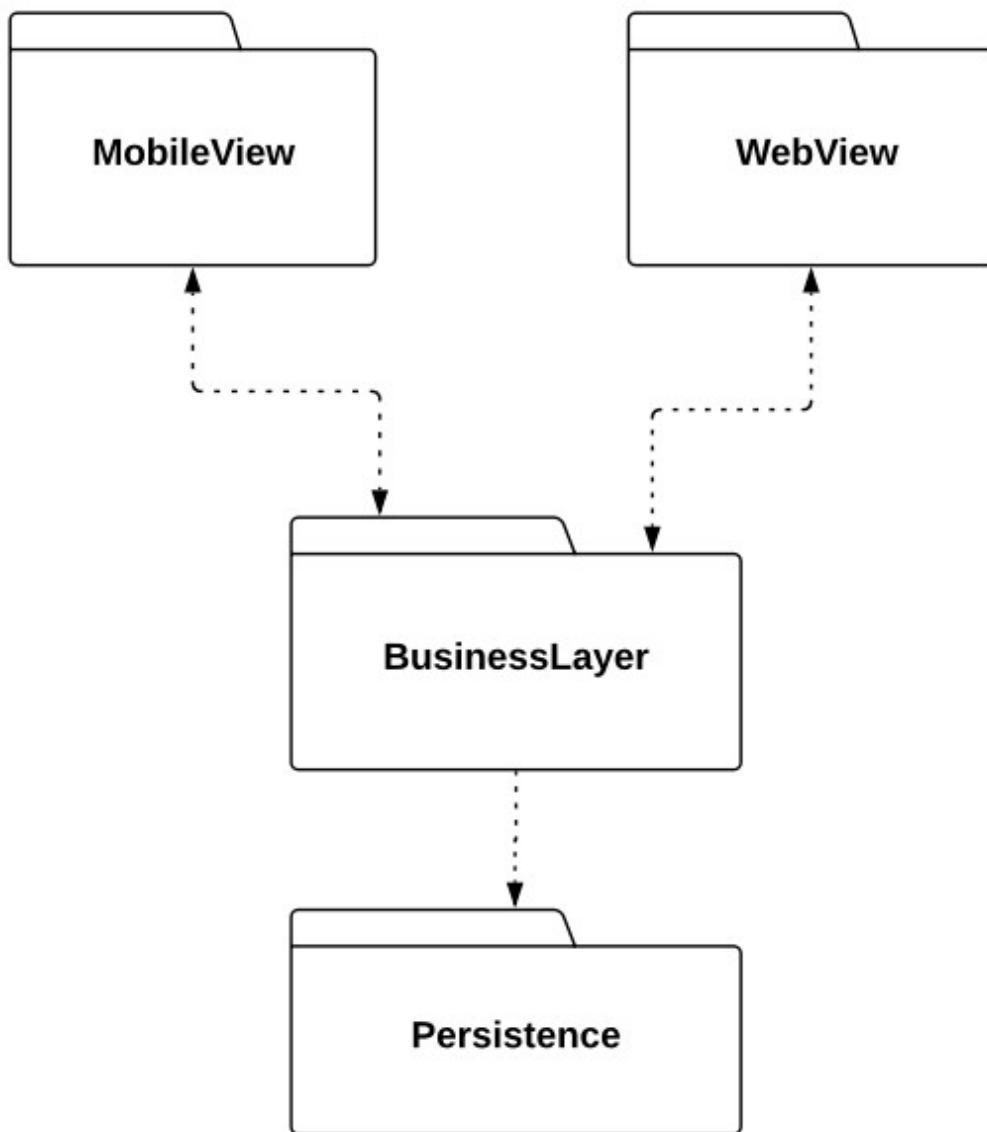
A figura da próxima página mostra um exemplo de diagrama de pacotes: Nesse diagrama, podemos ver que o sistema possui quatro pacotes principais: MobileView, WebView, BusinessLayer e Persistence. Podemos ver ainda as dependências — setas tracejadas — que existem entre eles. Ambos os pacotes View usam classes de BusinessLayer. Por outro lado, as classes de BusinessLayer também usam classes da View, por exemplo, para notificá-las da ocorrência de algum evento. Por isso, as setas que ligam os pacotes de View a BusinessLayer são bidirecionais. Por fim, apenas classes do pacote BusinessLayer usam classes do pacote Persistence.

Para concluir, gostaríamos de acrescentar duas observações:

- Dependências não incluem informações sobre quantas classes do pacote de origem dependem de classes do pacote de destino. Por exemplo,

suponha dois pacotes P1 e P2, ambos com 100 classes. Suponha ainda que uma única classe de P1 use uma única classe de P2. Mesmo nesse caso, dizemos que existe uma dependência de P1 para P2.

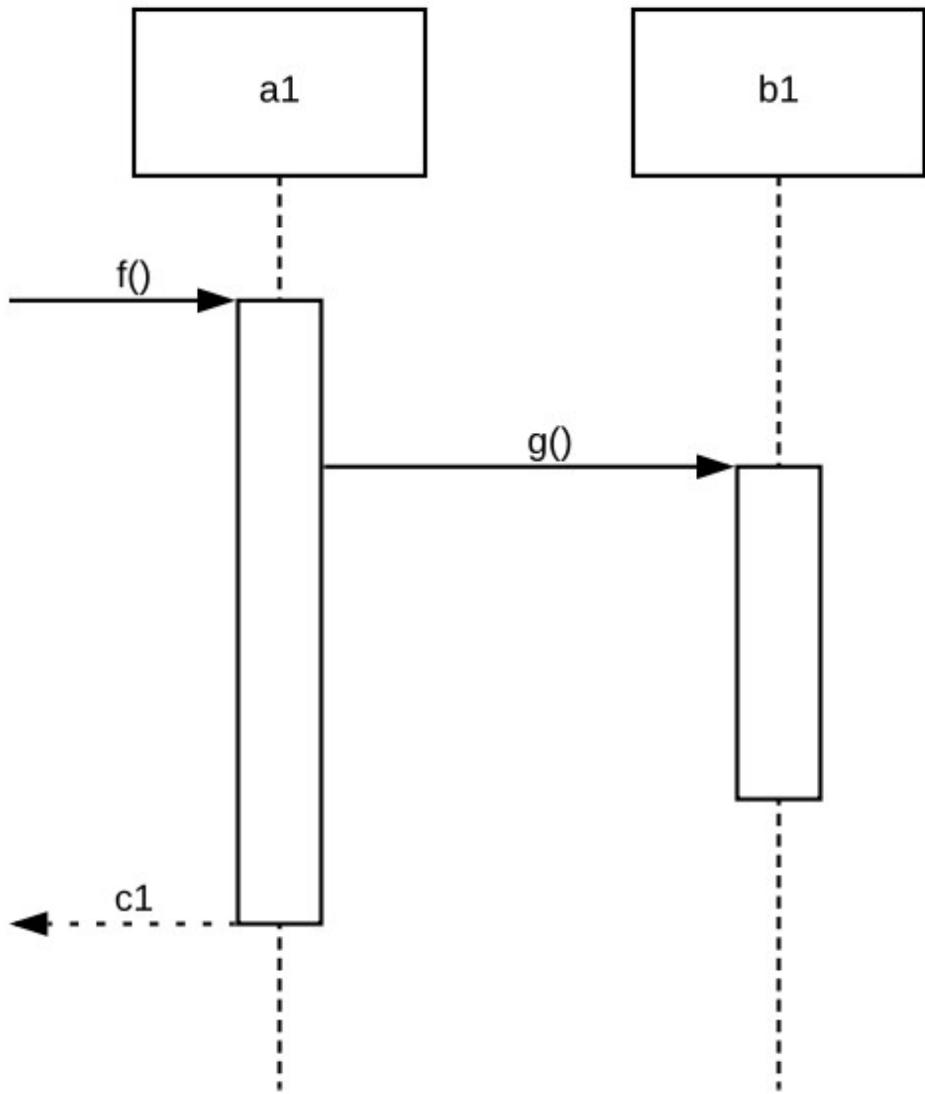
- Em diagramas de pacotes, temos um único tipo de seta, sempre tracejada, que representa qualquer tipo de relacionamento, seja ele por meio de associação, herança ou dependência simples. Essa semântica é diferente daquela que apresentamos para setas tracejadas em diagramas de classes. Nesses últimos, relações de associação e herança são representadas por meio de setas contínuas. Apenas as demais dependências são representadas por meio de setas tracejadas.



# Diagrama de Sequência

Diagramas de sequência são diagramas dinâmicos, também chamados de comportamentais. Por isso, em vez de classes, eles modelam objetos de um sistema. Adicionalmente, eles incluem informações sobre quais métodos desses objetos são executados em um determinado cenário de uso de um programa. Logo, eles são usados quando se pretende explicar o comportamento de um sistema, em um determinado cenário. Por exemplo, no final desta seção, vamos apresentar um diagrama de sequência que ilustra os métodos que são chamados quando um cliente chega em um caixa eletrônico e solicita uma operação de retirada de valores.

Antes disso, para iniciar a apresentação de diagramas de sequência, vamos usar o diagrama da próxima página. Apesar de simples, esse diagrama serve para mostrar a dinâmica e a notação usada por diagramas de sequência. Como já dissemos, diagramas de sequência modelam objetos, os quais são representados por meio de retângulos, com o nome dos objetos modelados. Esses retângulos ficam dispostos logo na primeira linha do diagrama. Portanto, dois objetos são representados no diagrama anterior, de nomes a1 e b1. Abaixo de cada objeto, desenha-se uma linha vertical, a qual pode assumir duas formas: (1) quando ela é desenhada de forma tracejada, o objeto está inativo, isto é, nenhum de seus métodos está sendo executado; (2) quando a linha fica cheia, ganhando um formato retangular, um dos métodos do objeto foi chamado e encontra-se em execução. Quando essa execução termina, a linha volta a ficar tracejada. Além disso, o início da chamada é indicado por uma seta na horizontal, com o nome do método chamado. O retorno da chamada é indicado por uma seta tracejada, com o nome do objeto retornado. No entanto, às vezes a seta de retorno é omitida, como no caso da chamada do método g. Existem dois motivos para essa omissão: (1) o tipo de retorno é void; ou (2) o objeto de retorno não é relevante, a ponto de merecer ser representado no diagrama.

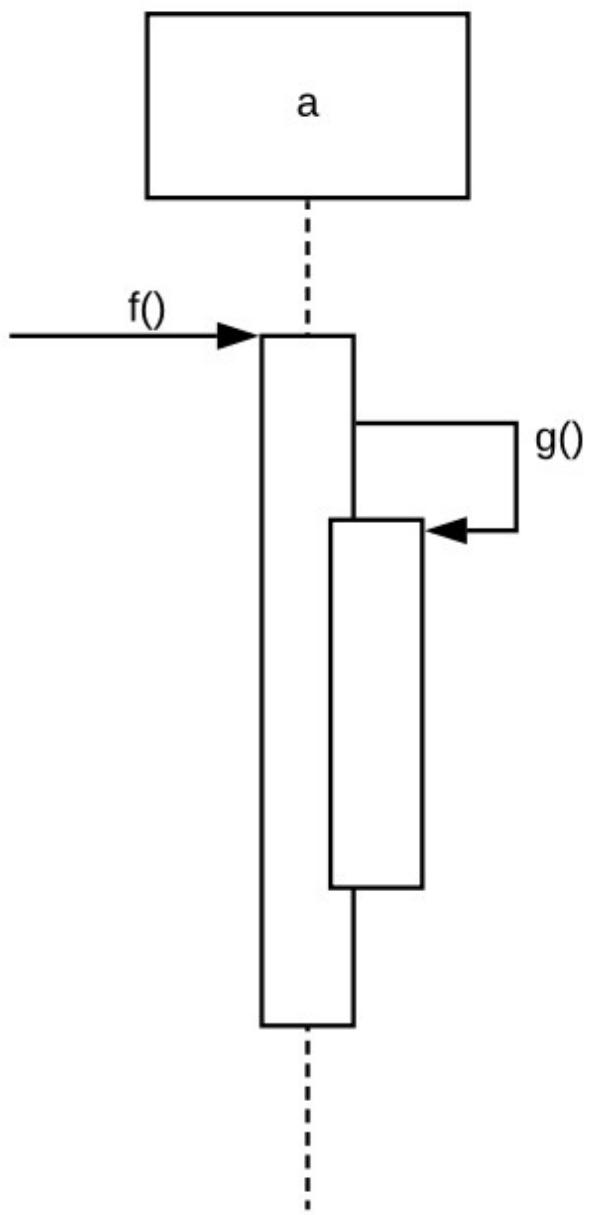


No diagrama de sequência mostrado acima representamos apenas dois objetos (**a1** e **b1**). Mas um diagrama de sequência pode ter mais objetos. No entanto, esse número não pode crescer tanto, pois o diagrama acaba ficando complexo e de difícil entendimento. Por exemplo, pode não ser possível representá-lo em uma única folha de papel ou em uma tela de computador.

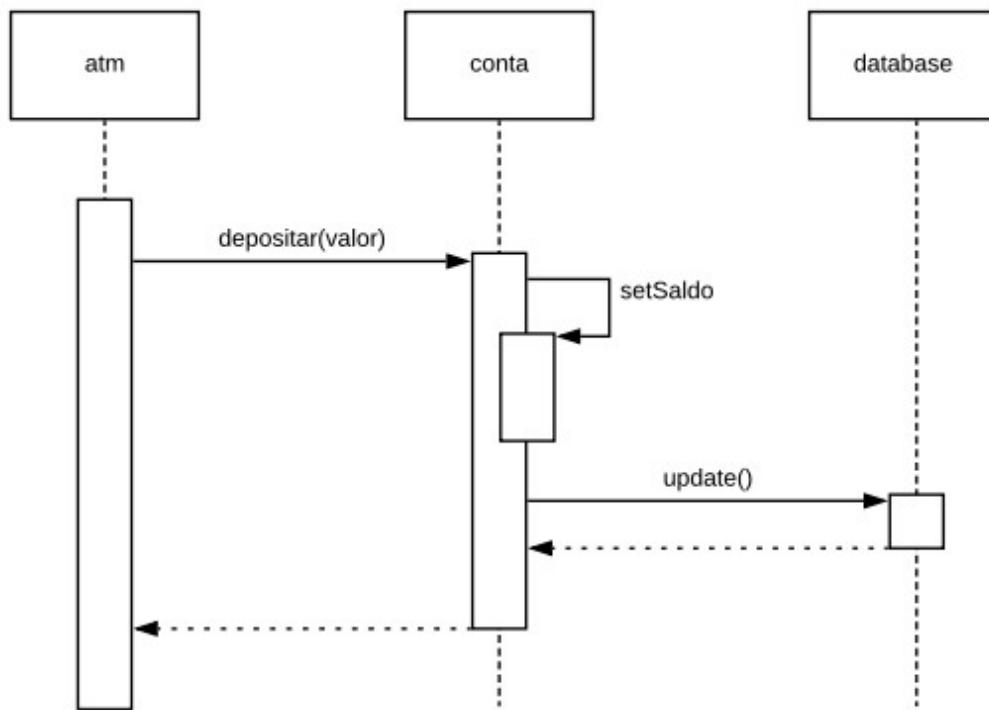
Um objeto pode ficar ativo e inativo diversas vezes em um mesmo diagrama. Ou seja, ele pode executar um método; ficar inativo; executar um novo método; ficar inativo, etc. Existe ainda um caso especial, quando um objeto chama um método dele mesmo, isto é, quando ele chama um método usando **this**. Para ilustrar esse caso, suponha o seguinte programa.

```
class A {  
    void g() {  
        ...  
    }  
  
    void f() {  
        ...  
        g();  
        ...  
    }  
  
    main() {  
        A a = new A();  
        a.f();  
    }  
}
```

A execução desse programa é representada pelo diagrama de sequência a seguir. Observe como a chamada de `g()` feita por `f()` é representada por meio de um novo retângulo, que sai do retângulo que representa a ativação da função `f()`.

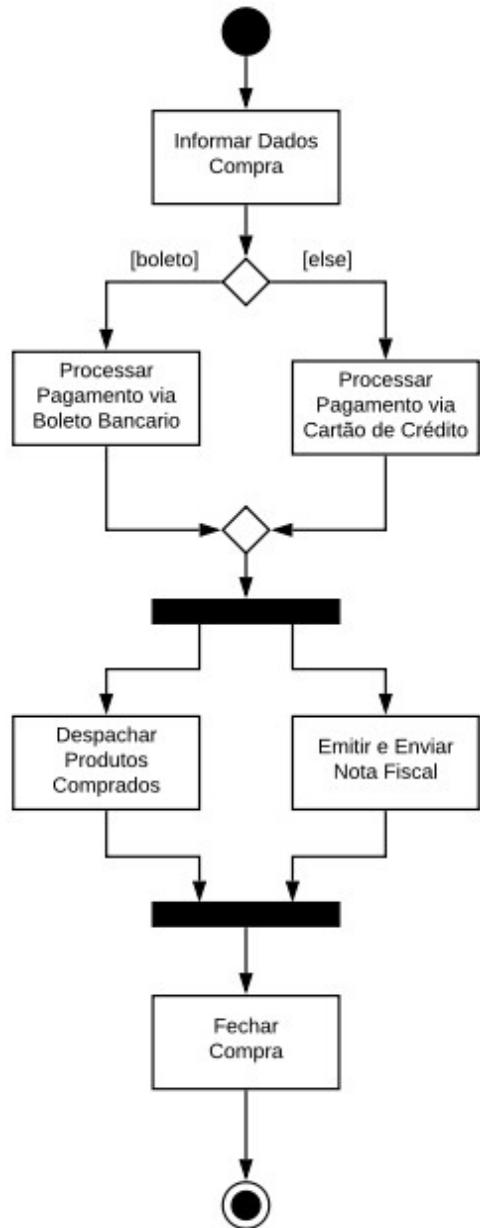


Para concluir, o próximo diagrama mostra um cenário mais real, que ilustra os métodos chamados quando o cliente de uma caixa eletrônico solicita um depósito de certo valor em sua conta.

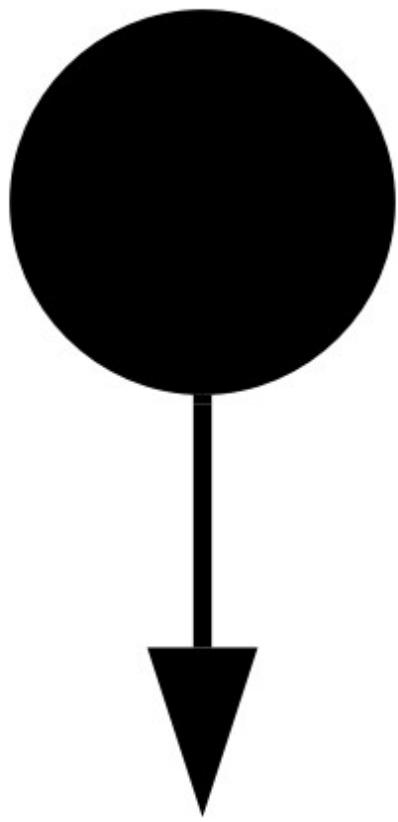


## Diagrama de Atividades

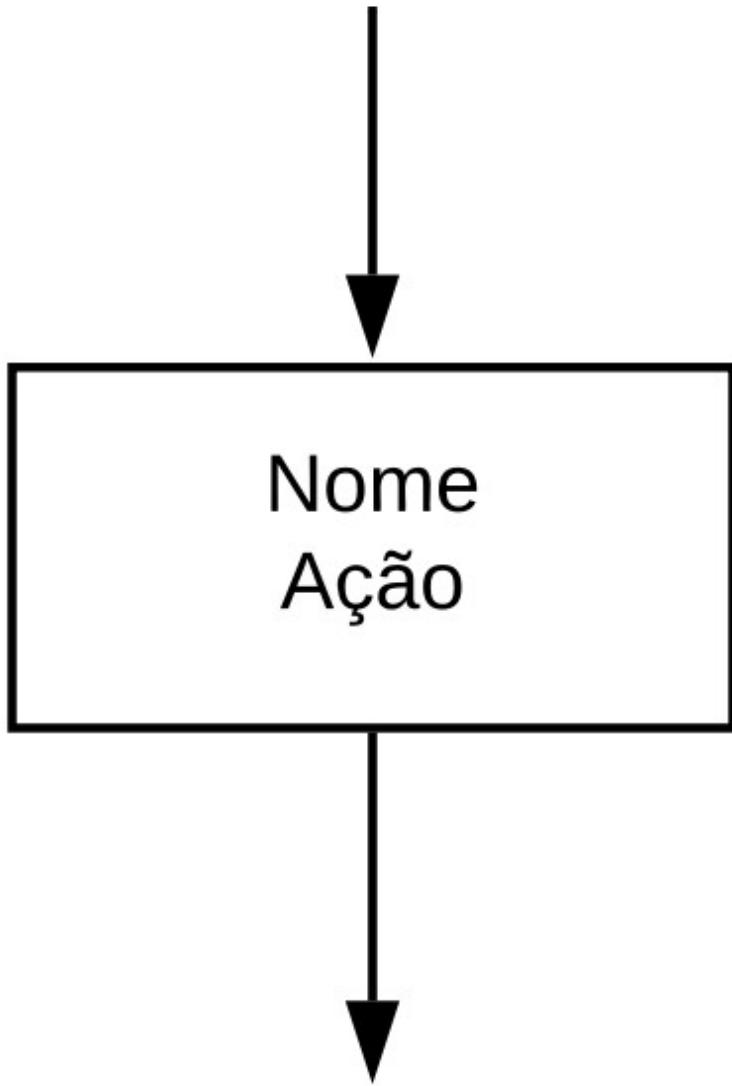
Diagramas de atividades são usados para representar, em alto nível, um processo ou fluxo de execução. Os principais elementos desses diagramas são **ações** representadas por retângulos. Existem ainda elementos de **controle**, que definem a ordem de execução das ações. A figura da próxima página mostra um diagrama de atividades que modela o processo seguido após um usuário fechar uma compra em uma loja virtual. Para isso, assume-se que os produtos comprados já estão no carrinho de compra. Para entender o funcionamento de um diagrama de atividades (como aquele mostrado na figura), devemos assumir que existe uma ficha (*token*) imaginária que caminha pelos nodos do diagrama. A seguir, explicamos o comportamento de cada nodo de um diagrama de atividades, assumindo a existência dessa ficha.



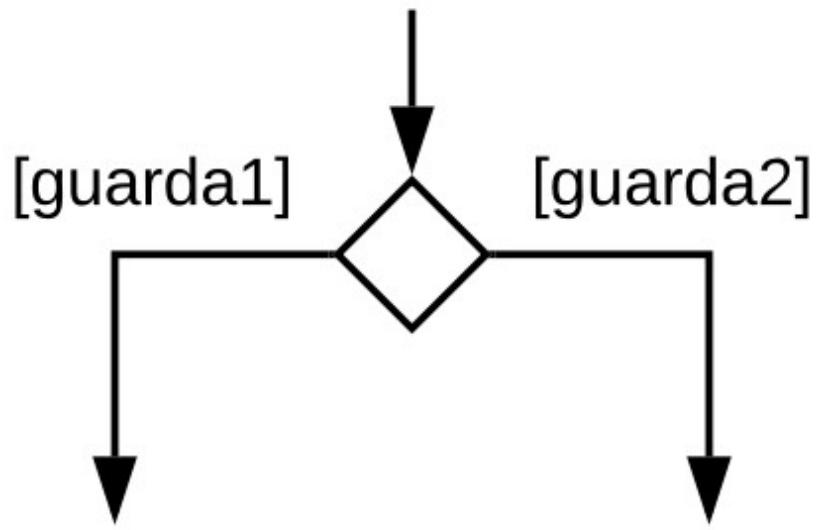
**Nodo Inicial:** Cria uma ficha para dar início à execução do processo. Feito isso, repassa a ficha para seu único fluxo de saída. Por definição, o nodo inicial não possui fluxo de entrada.



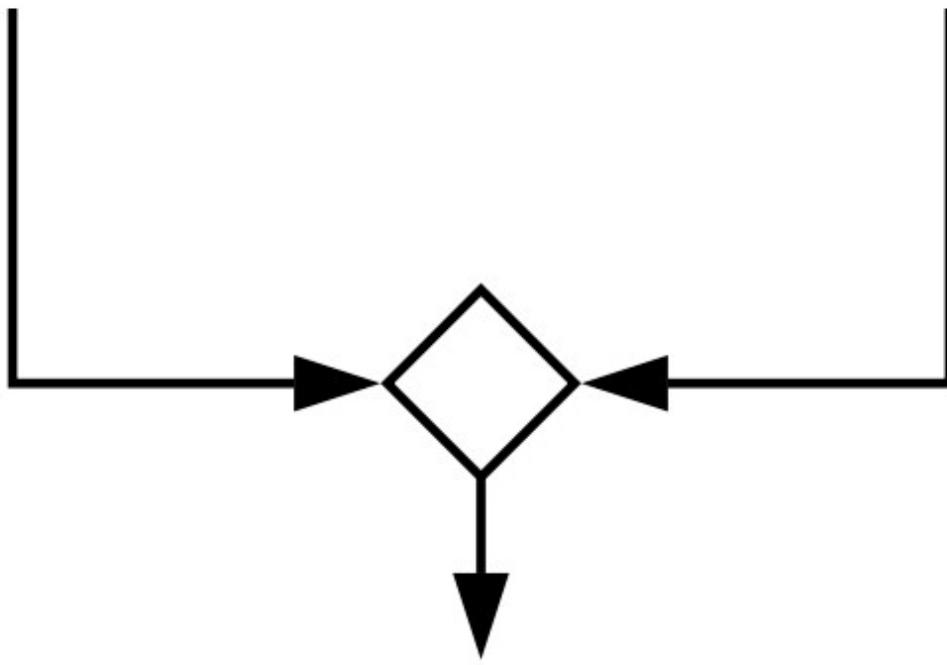
**Ações:** Possuem um único fluxo de entrada e um único fluxo de saída. Para uma ação ser executada uma ficha precisa chegar no seu fluxo de entrada. Após a execução, repassa-se a ficha para o fluxo de saída.



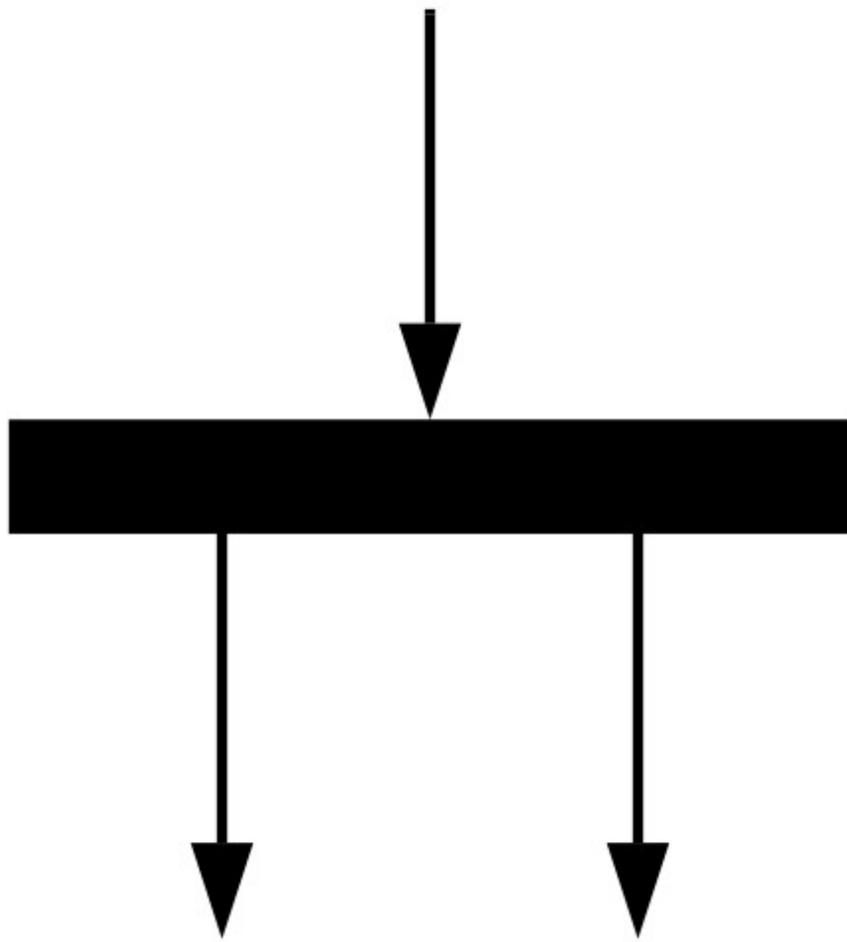
**Decisões:** Possuem um único fluxo de entrada e dois ou mais fluxos de saída. Cada fluxo de saída possui uma variável booleana associada, chamada de guarda. Para se tomar uma decisão, precisa-se receber uma ficha no fluxo de entrada. Quando isso acontece, a ficha é repassada apenas para o fluxo de saída cuja condição é verdadeira.



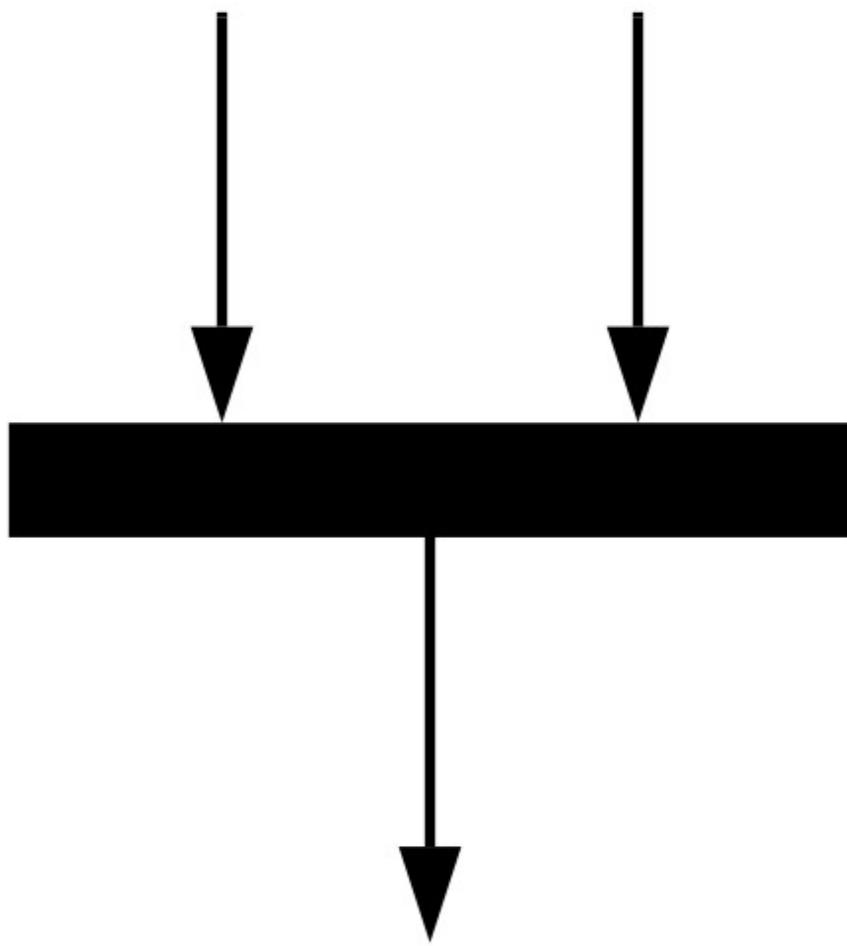
**Merges:** Podem possuir vários fluxos de entrada, mas um único fluxo de saída. Quando uma ficha chega em um dos fluxos de entrada, fazem seu repasse para o fluxo de saída. São usados para unir os fluxos de nodos de decisão.



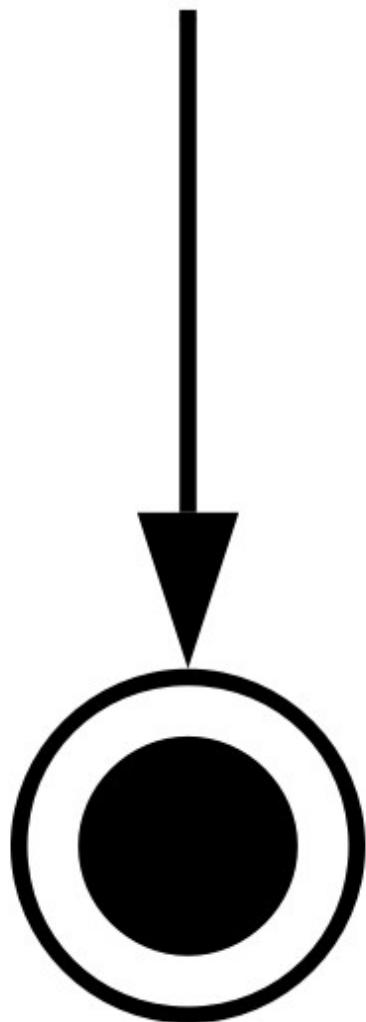
**Forks:** Possuem um único fluxo de entrada e um ou mais fluxos de saída. Atuam como multiplicadores de ficha: quando recebem uma ficha no fluxo de entrada, criam e repassam fichas idênticas em cada fluxo de saída. Como resultado, passam a existir múltiplos processos em execução de forma paralela.



**Joins:** Possuem vários fluxos de entrada, mas um único fluxo de saída. Atuam como sorvedouros de fichas: esperam que fichas cheguem em todos os fluxos de entrada. Quando isso acontece, repassam uma única ficha para o fluxo de saída. Logo, são usados para sincronizar processos. Em outras palavras, transformar vários fluxos de execução em um único fluxo.



**Nodo Final:** Pode possuir mais de um fluxo de entrada; mas não possui fluxos de saída. Quando uma ficha chega em um dos fluxos de entrada, encerra-se a execução do diagrama de atividades.



**Aprofundamento:** Existem pelo menos três outras alternativas para modelagem de fluxos e processos:

- **Fluxogramas**, os quais foram propostos tão logo se começou a desenvolver os primeiros programas para computadores modernos. Diagramas de atividades são parecidos com fluxogramas; porém, eles incluem suporte a concorrência, por meio de *forks* e *joins*. Por outro lado, fluxogramas modelam processos sequenciais.

- **Redes de Petri** é uma notação gráfica, proposta pelo matemático alemão Carl Adam Petri, em 1962, para modelagem de sistemas concorrentes. Redes de Petri possuem uma representação gráfica e também usam fichas (*tokens*) para marcar o estado corrente do sistema. Elas têm ainda a vantagem de possuir uma definição mais formal, principalmente quando comparada com a definição de diagramas de atividades. Por outro lado, esses últimos tendem a oferecer uma notação mais simples e fácil de entender.
- **BPMN** (*Business Process Model and Notation*) é um esforço mais recente, que teve início nos anos 2000, visando a proposição de uma notação gráfica mais amigável para modelagem de processos de negócio do que aquela oferecida por diagramas de atividades. Um dos objetivos é propiciar que analistas de negócio possam ler, interpretar e validar diagramas BPMN.

## Bibliografia

Martin Fowler. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley, 2003.

Grady Booch, James Rumbaugh, Ivar Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 2005.

Craig Larman. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice-Hall, 2004.

## Exercícios de Fixação

1. Explique e discuta os três usos possíveis de UML:

1. Como blueprint (ou plantas técnicas detalhadas)
2. Como sketches (esboços)
3. Como linguagem de programação.

2. Descreva cenários de uso de diagramas de classes UML como instrumento dos seguintes tipos de engenharia:

1. Engenharia Reversa

2. Engenharia Avante (*Forward Engineering*).

3. Modele os cenários descritos a seguir usando Diagramas de Classe UML. Veja que as classes são grafadas em uma fonte diferente.

1. ContaBancaria possui exatamente um Cliente. Um Cliente, por sua vez, pode ter várias ContaBancaria. Existe navegabilidade em ambos os sentidos.

2. ContaPoupanca e ContaSalario são subclasses de ContaBancaria.

3. No código de ContaBancaria declara-se uma variável local do tipo BancoDados.

4. Um ItemPedido se refere a um único Produto (sem navegabilidade). Um Produto pode ter vários ItemPedido (com navegabilidade).

5. A classe Aluno possui atributos nome, matricula, curso (todos privados); e métodos getCurso() e cancelaMatricula(), ambos públicos.

4. (ENADE 2014, Tec. e Análise de Sistemas) Construa um diagrama de classes para representar as seguintes classes e associações:

- Uma revista científica possui título, ISSN e periodicidade;
- Essa revista publica diversas edições com os seguintes atributos: número da edição, volume da edição e data da edição. Importante destacar que cada instância da classe edição relaciona-se única e exclusivamente a uma instância da classe revista científica, não podendo relacionar-se com nenhuma outra;

- Um artigo possui título e nome do autor. Um artigo é um conteúdo exclusivo de uma edição. E uma edição obrigatoriamente tem que possuir no mínimo 10 e no máximo 15 artigos.

5. Crie diagramas de classes para os seguintes trechos de código:

1.

```
public class HelloWorldSwing {  
  
    public static void main(String[] args) {  
        JFrame frame = new JFrame("Hello world!");  
        frame.setVisible(true);  
    }  
  
}
```

2.

```
class HelloWorldSwing extends JFrame {  
  
    public HelloWorldSwing() {  
        super("Hello world!");  
    }  
  
    public static void main(String[] args) {  
        HelloWorldSwing frame = new HelloWorldSwing();  
        frame.setVisible(true);  
    }  
  
}
```

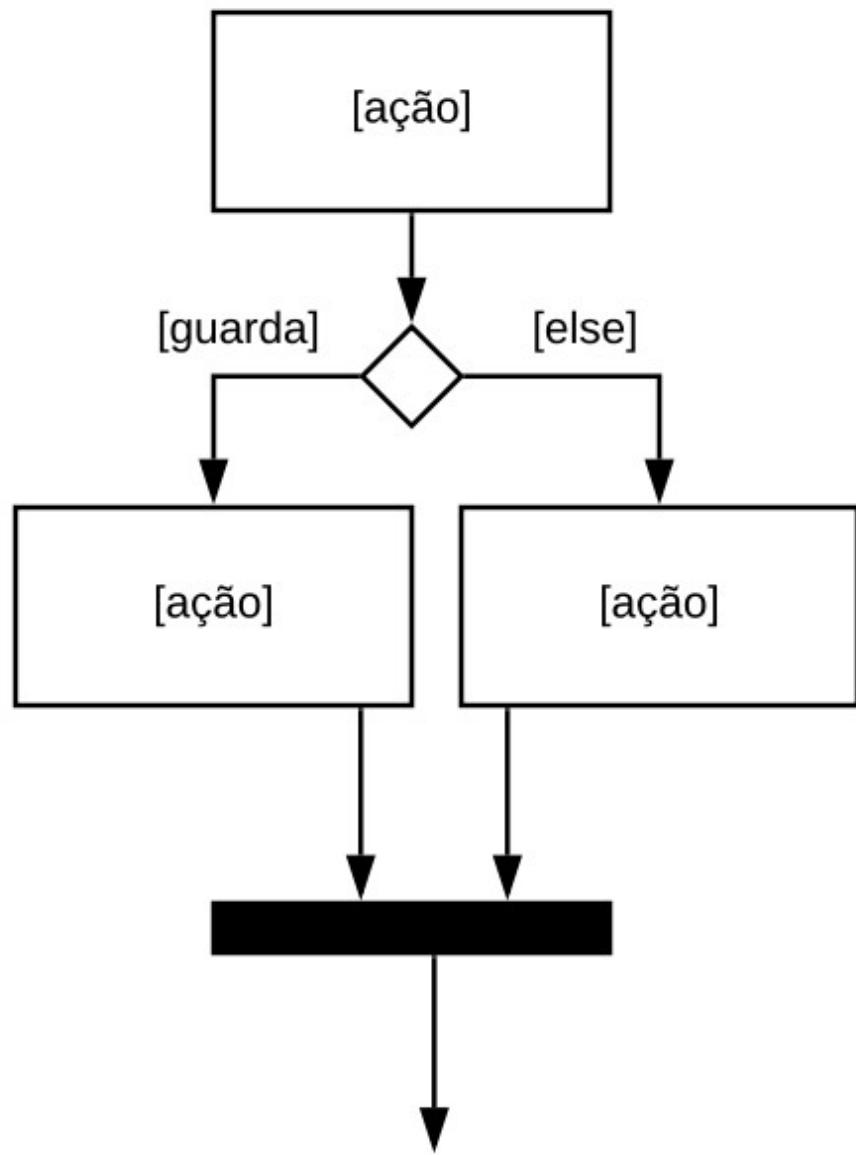
6. Mostre o diagrama de sequência relativo ao seguinte código. O diagrama deve começar com a seguinte chamada `a._m5()`.

```
A a = new A(); // variáveis globais  
B b = new B();  
C c = new C();  
  
class C {  
    void m1() { ... }  
}
```

```
class B {  
    void m2() { ... c.m1(); ... this.m3(); ... }  
    void m3() { ... c.m1(); ... }  
    void m4() { ... }  
}  
  
class A {  
    void m5() { ... b.m2(); ... b.m3(); ... b.m4(); ... }  
}
```

7. Em diagramas de atividades, explique a diferença entre um nodo de *merge* e um nodo de *join*.

8. Qual é o erro do seguinte diagrama de atividades? Refaça o diagrama de forma a refletir corretamente a intenção do projetista.



# Cap 5 Princípios de Projeto

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).

## 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.

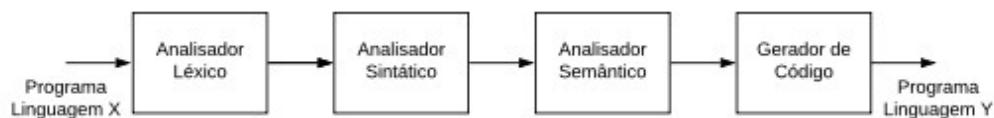
Essa explicação pode passar a impressão de que projeto é uma atividade simples. No entanto, no projeto de software temos que combater um grande inimigo: a **complexidade** que caracteriza sistemas modernos de software. Talvez, por isso, Ousterhout mencione que a decomposição de um problema em partes 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. Dentre outros, funções, classes, interfaces, pacotes e bibliotecas 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.

## 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.



### Principais módulos de um compilador

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 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 inicial.

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();
```

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

## 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.

## 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 sempre 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, no qual 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 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 exemplos de falta de integridade conceitual em nível de código:

- 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).

## 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++ e Ruby. 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 e componentes.

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  $C_i$  é responsável pelos problemas de desempenho do sistema. Quando detalhes de implementação de  $C_i$  são ocultados do resto do sistema, fica mais fácil trocar sua implementação por uma classe  $C'_i$ , 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 demandar 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.

## 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 de 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 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");
    }
}
```

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.

**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 lista 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 leitura direta em bases de dados de outros sistemas, sem memória compartilhada ou variáveis globais ou qualquer tipo de *backdoor*. 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!

## 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, no qual 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).

## 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 afirmado que toda classe deve ter uma única responsabilidade no sistema. Ou, ainda, afirmado 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 projetos de software, a qual é 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.

## 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 seria 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`.

## 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 frequência.

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:

Maximize 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.

## Exemplos

**Exemplo 1:** Suponha a classe Estacionamento, usada na Seção 5.3, 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 {  
    private void f() {  
        int total; ...  
        File arq = File.open("arq1.db");  
        total = arq.readInt();  
        ...  
    }  
}  
  
class B {  
    private void g() {  
        int total;  
        // computa valor de total  
        File arq = File.open("arq1.db");  
        arq.writeInt(total);  
        ...  
        arq.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 {  
    private void f(B b) {  
        int total;  
        total = b.getTotal();  
        ...  
    }  
}  
  
class B {  
    int total;  
  
    public int getTotal() {  
        return total;  
    }  
  
    private void g() {  
        // computa valor de total  
        File arq = File.open("arq1");  
        arq.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. Ou seja, 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 demandam mudanças em um outro elemento. Acoplamento pode dar origem a uma relação bem útil 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 demandam 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.

## SOLID e Outros 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:

- Single Responsibility Principle
- Open Closed/Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency 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 frequência. O mesmo acontece com tecnologias de implementação, como bibliotecas e frameworks.

## 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 no qual 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 no 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`), conforme mostrado no código a seguir. 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;  
    }  
  
}
```

## 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 têm 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 `Mx` e `My`. O primeiro conjunto é usado por clientes `Cx` (que então não usam os métodos `My`). De forma inversa, os métodos `My` são usados apenas por clientes `Cy` (que não usam os métodos `Mx`). Consequentemente, essa interface deveria ser quebrada em duas interfaces menores e específicas: uma interface contendo apenas os métodos `Mx` e a segunda interface contendo apenas os métodos `My`.

**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();  
    ...  
}
```

## 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 `projeto` 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() {
    ...
    ProjetoLG projeto = new ProjetoLG();
    ...
    g(projeto);
}

void g(Projeto projeto) {
    ...
}
```

**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.

## 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 extends 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:

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

Solução via Composição:

```
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 se adota 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 no qual 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 **reúso caixa-branca**, pois as subclasses costumam ter acesso a detalhes de implementação da classe base. Por outro lado, composição é um mecanismo de **reúso 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.

## 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
}
```

O jornaleiro tem acesso à carteira do seu cliente — via `getCarteira()` — e então ele mesmo retira o valor do jornal dela. Porém, nenhum cliente aceitaria que um jornaleiro tivesse essa liberdade. 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.

## 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 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;
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.

## 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. Em um desses trabalhos, 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();  
    ...  
}
```

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  
...  
f(new B2()); // e de qualquer 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:  $1 \leq n \leq 1$  milhão. Ou seja, o método deve ser capaz de retornar qualquer número primo, para  $n$  variando de 1 até 1 milhão. Nesse exemplo, uma possível 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).

## Métricas de Código Fonte

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 que calculam essas métricas de forma automática. Algumas delas funcionam como plugins de IDEs conhecidas.

## 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 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.

## 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 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, computar o seguinte conjunto:

$$M(C) = \{ (f_1, f_2) \mid f_1 \text{ e } f_2 \text{ são métodos de } C \}$$

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

$$A(f) = \text{conjunto de atributos da classe que são acessados por um método } f$$

O valor de LCOM de C é assim definido:

$$P = | \{ (f_1, f_2) \text{ in } M(C) \mid A(f_1) \text{ e } A(f_2) \text{ são conjuntos disjuntos} \} |$$

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, isto é, a interseção deles é vazia.

**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)	$A(m1) = \{a1, a2\}$ $A(m2) = \{a1, a3\}$	$\{a1\}$
(m1,m3)	$A(m1) = \{a1, a2\}$ $A(m3) = \{a3\}$	$\emptyset$
(m2,m3)	$A(m2) = \{a1, a3\}$ $A(m3) = \{a3\}$	$\{a3\}$

### Exemplo de cálculo de LCOM

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.

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 (f1,f2), onde f1 manipula alguns atributos e f2 manipula atributos diferentes.

Para cálculo de LCOM 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 cálculo de LCOM. A versão que apresentamos, chamada de LCOM1, 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.

## 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 de que A depende,  $CBO(A) = 9$ .

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.

## 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:

$$CC = \text{número de comandos de decisão em uma função} + 1$$

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 em que 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 será um acoplamento ruim?

7. Suponha um programa em que 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 endereco = 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. 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"  
    }  
}
```

```
    }  
}
```

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

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

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.

# Cap 6 Padrões de Projeto

Este capítulo inicia com uma introdução ao conceito e aos objetivos de padrões de projeto (Seção 6.1). Em seguida, discutimos com detalhes dez padrões de projetos: Fábrica, Singleton, Proxy, Adaptador, Fachada, Decorador, Strategy, Observador, Template Method e Visitor. Cada um desses padrões é discutido em uma seção separada (Seções 6.1 a 6.11). A apresentação de cada padrão é organizada em três partes: (1) um contexto, isto é, um sistema no qual o padrão poderia ser útil; (2) um problema no projeto desse sistema; (3) uma solução para esse problema por meio de padrões. Na Seção 6.12, discutimos brevemente mais alguns padrões. Terminamos o capítulo alertando que padrões de projeto não são uma bala de prata, ou seja, discutimos situações nas quais o uso de padrões de projeto não é recomendado (Seção 6.13).

## Introdução

Padrões de projeto são inspirados em uma ideia proposta por Christopher Alexander, um arquiteto — de construções civis e não de software — e professor da Universidade de Berkeley. Em 1977, Alexander lançou um livro chamado *A Patterns Language*, no qual ele documenta diversos padrões para construção de cidades e prédios. Segundo Alexander:

Cada padrão descreve um problema que sempre ocorre em nosso contexto e uma solução para ele, de forma que possamos usá-la um milhão de vezes.

Em 1995, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides lançaram um livro adaptando as ideias de Alexander para o mundo de desenvolvimento de software ([link](#)). Em vez de propor um catálogo de soluções para projeto de cidades e prédios, eles propuseram um catálogo com soluções para resolver problemas recorrentes em projeto de software. Eles deram o nome de **Padrões de Projeto** às soluções propostas no livro. Eles definem padrões de projeto da seguinte forma:

Padrões de projeto descrevem objetos e classes que se relacionam para resolver um problema de projeto genérico em um contexto particular.

Assim, para entender os padrões propostos pela *Gang of Four* — nome pelo qual ficaram conhecidos os autores e também o livro de padrões de projeto — precisamos entender: (1) o problema que o padrão pretende resolver; (2) o contexto em que esse problema ocorre; (3) a solução proposta. Neste livro, vamos descrever alguns padrões de projeto, sempre focando nesses elementos: contexto, problema e solução. Iremos também mostrar vários exemplos de código fonte.

Além de oferecer soluções prontas para problemas de projeto, padrões de projeto transformaram-se em um vocabulário largamente adotado por desenvolvedores de software. Assim, é comum ouvir desenvolvedores dizendo que usaram uma fábrica para resolver um certo problema, enquanto que um segundo problema foi resolvido por meio de decoradores. Ou seja, eles apenas mencionam o nome do padrão e subentendem que a solução adotada já está clara. De forma semelhante, o vocabulário de padrões de projeto é muito usado na documentação de sistemas. Por exemplo, a figura da próxima página mostra a documentação de uma das classes da biblioteca padrão de Java. Podemos ver que o nome da classe termina em `Factory` — que é um dos padrões de projeto que vamos estudar daqui a pouco. Na descrição da classe, volta-se a mencionar que ela é uma fábrica. Portanto, desenvolvedores que conhecem esse padrão de projeto terão mais facilidade para entender e usar a classe em questão.

```
public abstract class DocumentBuilderFactory  
extends Object
```

Defines a factory API that enables applications to obtain a parser that produces DOM object trees from XML documents.

Documentação de uma classe Factory da API de Java

Um desenvolvedor pode se beneficiar do domínio de padrões de projeto em dois cenários principais:

- Quando ele estiver implementando o seu próprio sistema. Nesse caso, conhecer padrões de projeto pode ajudá-lo a adotar, no seu sistema, uma solução de projeto já testada e validada.
- Quando ele estiver usando um sistema de terceiros, como o pacote de Java que implementa a classe `DocumentBuilderFactory` da figura. Nesse caso, conhecimento de padrões de projeto pode ajudá-lo a entender o comportamento e a estrutura da classe que ele precisa usar.

É importante entender que padrões de projeto visam a criação de projetos de software flexíveis e extensíveis. Neste livro, antes de explicar cada um dos padrões, vamos apresentar um contexto e um trecho de código que funciona e produz um resultado. Porém, ele não dá origem a um projeto flexível. Para deixar essa inflexibilidade clara, apresentaremos um cenário de extensão do código mostrado, envolvendo a implementação de novos requisitos. Então vamos argumentar que essa extensão exigirá algum esforço, que poderá ser minimizado se usarmos um padrão de projeto.

Os quatro autores do livro de padrões de projeto defendem que devemos projetar um sistema pensando em mudanças que inevitavelmente vão ocorrer — eles chamam essa preocupação de *design for change*. Conforme afirmado por eles na frase que abre este capítulo, se *design for change* não for uma preocupação, os desenvolvedores correm o risco de em breve ter que planejar um profundo reprojeto de seus sistemas.

No livro sobre padrões de projeto, são propostos 23 padrões, divididos nas seguintes três categorias (os padrões que estudaremos neste capítulo estão em negrito, seguido do número da seção em que eles são apresentados):

- **Criacionais:** padrões que propõem soluções flexíveis para criação de objetos. São eles: **Abstract Factory (6.2)**, Factory Method, **Singleton (6.3)**, **Builder (6.12)** e Prototype.

- **Estruturais:** padrões que propõem soluções flexíveis para composição de classes e objetos. São eles: **Proxy (6.4)**, **Adapter (6.5)**, **Facade (6.6)**, **Decorator (6.7)**, Bridge, Composite e Flyweight.
- **Comportamentais:** padrões que propõem soluções flexíveis para interação e divisão de responsabilidades entre classes e objetos. São eles: **Strategy (6.8)**, **Observer (6.9)**, **Template Method (6.10)**, **Visitor (6.11)**, Chain of Responsibility, Command, Interpreter, **Iterator (6.12)**, Mediator, Memento e State.

**Tradução:** Por ser uma tradução direta, vamos traduzir os nomes dos padrões Fábrica Abstrata, Método Fábrica, Adaptador, Fachada, Decorador, Observador e Iterador. Os demais serão referenciados usando o nome original.

## Fábrica

**Contexto:** Suponha um sistema distribuído baseado em TCP/IP. Nesse sistema, três funções `f`, `g` e `h` criam objetos do tipo `TCPChannel` para comunicação remota, como mostra o próximo código.

```
void f() {
    TCPChannel c = new TCPChannel();
    ...
}

void g() {
    TCPChannel c = new TCPChannel();
    ...
}

void h() {
    TCPChannel c = new TCPChannel();
    ...
}
```

**Problema:** Suponha que — em determinadas configurações do sistema — precisaremos usar UDP para comunicação. Portanto, se considerarmos esse requisito, o sistema não atende ao Princípio Aberto/Fechado, isto é, ele não está fechado para modificações e aberto para extensões nos protocolos de comunicação usados. Sendo mais claro, gostaríamos de parametrizar o

código acima para criar objetos dos tipos `TCPChannel` ou `UDPChannel`, dependendo dos clientes. O problema é que o operador `new` deve ser seguido do nome literal de uma classe. Esse operador — pelo menos em linguagens como Java, C++ e C# — não permite que a classe dos objetos que se pretende criar seja passada como um parâmetro. Resumindo, o problema consiste em parametrizar a instanciação dos canais de comunicação no código acima, de forma que ele consiga trabalhar com protocolos diferentes.

**Solução:** A solução que vamos descrever baseia-se no padrão de projeto **Fábrica**. Esse padrão possui algumas variações, mas no nosso problema vamos adotar um método estático que: (1) apenas cria e retorna objetos de uma determinada classe; (2) e também oculta o tipo desses objetos por trás de uma interface. Um exemplo é mostrado a seguir:

```
class ChannelFactory {  
    public static Channel create() { // método fábrica estático  
        return new TCPChannel();  
    }  
}  
  
void f() {  
    Channel c = ChannelFactory.create();  
    ...  
}  
  
void g() {  
    Channel c = ChannelFactory.create();  
    ...  
}  
void h() {  
    Channel c = ChannelFactory.create();  
    ...  
}
```

Nessa nova versão, as funções `f`, `g` e `h` não têm consciência do tipo de `Channel` que vão criar e usar. Elas chamam um **Método Fábrica Estático**, que instancia e retorna um objeto de uma classe concreta — para ser claro, essa variante do padrão Fábrica não foi proposta no livro da Gangue dos Quatro, mas sim alguns anos depois por Joshua Bloch ([link](#)). É importante também destacar que as três funções usam sempre uma interface `Channel` para manipular os objetos criados pelo método fábrica estático. Ou seja,

aplicamos o princípio Prefira Interfaces a Classes (ou Inversão de Dependências).

No novo código, o sistema continua funcionando com canais do tipo `TCPChannel`. Porém, se quisermos mudar o tipo de canal, temos agora que modificar um único elemento do código: o método `create` da classe `channelFactory`. Dizendo de outra forma, um método fábrica estático funciona como um aspirador de métodos `new`: todas as chamadas antigas de `new` migram para uma única chamada, no método fábrica estático.

Existem ainda algumas variações do padrão Fábrica. Em uma delas, uma classe abstrata é usada para concentrar vários métodos fábrica. Essa classe recebe então o nome de **Fábrica Abstrata**. Um exemplo é mostrado no código a seguir:

```
abstract class ProtocolFactory { // Fábrica Abstrata
    abstract Channel createChannel();
    abstract Port createPort();
    ...
}

void f(ProtocolFactory pf) {
    Channel c = pf.createChannel();
    Port p = pf.createPort();
    ...
}
```

No exemplo acima, omitimos as classes que estendem a classe abstrata `ProtocolFactory` e que vão implementar, de fato, os métodos concretos para criação de canais e portas de comunicação. Podemos, por exemplo, ter duas subclasses: `TCPProtocolFactory` e `UDPProtocolFactory`.

## Singletton

**Contexto:** Suponha uma classe `Logger`, usada para registrar as operações realizadas em um sistema. Um uso dessa classe é mostrado a seguir:

```
void f() {
    Logger log = new Logger();
    log.println("Executando f");
    ...
}
```

```

}

void g() {
    Logger log = new Logger();
    log.println("Executando g");
    ...
}

void h() {
    Logger log = new Logger();
    log.println("Executando h");
    ...
}

```

**Problema:** No código anterior, cada método que precisa registrar eventos cria sua própria instância de Logger. No entanto, gostaríamos que todos os usos de Logger tivessem como alvo a mesma instância da classe. Em outras palavras, não queremos uma proliferação de objetos Logger. Em vez disso, gostaríamos que existisse, no máximo, uma única instância dessa classe e que ela fosse usada em todas as partes do sistema que precisam registrar algum evento. Isso é importante, por exemplo, caso o registro de eventos seja feito em arquivos. Se for possível a criação de vários objetos Logger, todo novo objeto instanciado vai apagar o arquivo anterior, criado por outros objetos do tipo Logger.

**Solução:** A solução para esse problema consiste em transformar a classe Logger em um **Singleton**. Esse padrão de projeto define como implementar classes que terão, como o próprio nome indica, no máximo uma instância. Mostramos a seguir a versão de Logger que funciona como um Singleton:

```

class Logger {

    private Logger() {} // proíbe clientes chamar new Logger()

    private static Logger instance; // instância única

    public static Logger getInstance() {
        if (instance == null) // 1a vez que chama-se getInstance
            instance = new Logger();
        return instance;
    }

    public void println(String msg) {
        // registra msg no console, mas poderia ser em arquivo
        System.out.println(msg);
    }
}

```

```
    }
}
```

Primeiro, essa classe tem um construtor *default* privado. Com isso, um erro de compilação ocorrerá quando qualquer código fora da classe tentar chamar `new Logger()`. Além disso, um atributo estático armazena a instância única da classe. Quando precisarmos dessa instância, devemos chamar o método público e estático `getInstance()`. Um exemplo é mostrado a seguir:

```
void f() {
    Logger log = Logger.getInstance();
    log.println("Executando f");
    ...
}

void g() {
    Logger log = Logger.getInstance();
    log.println("Executando g");
    ...
}

void h() {
    Logger log = Logger.getInstance();
    log.println("Executando h");
    ...
}
```

Nesse novo código, temos certeza de que as três chamadas de `getInstance` retornam a mesma instância de `Logger`. Todas as mensagens serão então registradas usando-se essa instância.

Dentre os padrões de projeto propostos no livro da Gangue dos Quatro, `Singleton` é o mais polêmico e criticado. O motivo é que ele pode ser usado para camuflar a criação de variáveis e estruturas de dados globais. No nosso caso, a instância única de `Logger` é, na prática, uma variável global que pode ser lida e alterada em qualquer parte do programa. Para isso, basta chamar `Logger.getInstance()`. Como vimos no Capítulo 5, variáveis globais representam uma forma de acoplamento ruim (ou forte) entre classes, isto é, uma forma de acoplamento que não é mediada por meio de interfaces estáveis. Porém, no caso de `Logger`, o uso de `Singleton` não gera preocupações, pois ele é exatamente aquele recomendado pelo padrão: temos um recurso que é único — um arquivo de log, no caso — e queremos refletir

essa característica no projeto, garantindo que ele seja manipulado por meio de uma classe que, por construção, possuirá no máximo uma instância.

Em resumo: Singleton deve ser usado para modelar recursos que, conceitualmente, devem possuir no máximo uma instância durante a execução de um programa. Por outro lado, um uso abusivo do padrão ocorre quando ele é adotado como um artifício para criação de variáveis globais.

Por fim, existe mais uma crítica ao uso de Singletons: eles tornam o teste automático de métodos mais complicado. O motivo é que o resultado da execução de um método pode agora depender de um estado global armazenado em um Singleton. Por exemplo, suponha um método `m` que retorna o valor de `x + y`, onde `x` é um parâmetro de entrada e `y` é uma variável global, que é parte de um Singleton. Logo, para testar esse método precisamos fornecer o valor `x`; o que é bastante fácil, pois ele é um parâmetro do método. Mas também precisamos garantir que `y` terá um valor conhecido; o que pode ser mais difícil, pois ele é um atributo de uma outra classe.

**Código Fonte:** O código do exemplo de Singleton usado nesta seção está disponível neste [link](#).

## Proxy

**Contexto:** Suponha uma classe `BookSearch`, cujo principal método pesquisa por um livro, dado o seu ISBN:

```
class BookSearch {  
    ...  
    Book getBook(String ISBN) { ... }  
    ...  
}
```

**Problema:** Suponha que nosso serviço de pesquisa de livros esteja ficando popular e ganhando usuários. Para melhorar seu desempenho, pensamos em introduzir um sistema de cache: antes de pesquisar por um livro, iremos verificar se ele está no cache. Se sim, o livro será imediatamente retornado. Caso contrário, a pesquisa prosseguirá segundo a lógica normal do método `getBook()`. Porém, não gostaríamos que esse novo requisito — pesquisa em

cache — fosse implementado na classe BookSearch. O motivo é que queremos manter a classe coesa e aderente ao Princípio da Responsabilidade Única. Na verdade, o cache será implementado por um desenvolvedor diferente daquele que é responsável por manter BookSearch. Além disso, vamos usar uma biblioteca de cache de terceiros, com diversos recursos e customizações. Por isso, achamos importante separar, em classes distintas, o interesse pesquisar livros por ISBN (que é um requisito funcional) do interesse usar um cache nas pesquisas por livros (que é um requisito não-funcional).

**Solução:** O padrão de projeto **Proxy** defende a inserção de um objeto intermediário, chamado proxy, entre um objeto base e seus clientes. Assim, os clientes não terão mais uma referência direta para o objeto base, mas sim para o proxy. Por sua vez, o proxy possui uma referência para o objeto base. Além disso, o proxy deve implementar as mesmas interfaces do objeto base.

O objetivo de um proxy é mediar o acesso a um objeto base, agregando-lhe funcionalidades, sem que ele tome conhecimento disso. No nosso caso, o objeto base é do tipo BookSearch; a funcionalidade que pretendemos agregar é um cache; e o proxy é um objeto da seguinte classe:

```
class BookSearchProxy implements BookSearchInterface {  
    private BookSearchInterface base;  
  
    BookSearchProxy (BookSearchInterface base) {  
        this.base = base;  
    }  
  
    Book getBook(String ISBN) {  
        if("livro com ISBN no cache")  
            return "livro do cache"  
        else {  
            Book book = base.getBook(ISBN);  
            if(book != null)  
                "adicone book no cache"  
            return book;  
        }  
    }  
    ...  
}
```

Deve ser criada também uma interface BookSearchInterface, não mostrada no código. Tanto a classe base como a classe do proxy devem implementar essa interface. Isso permitirá que os clientes não tomem conhecimento da existência de um proxy entre eles e o objeto base. Mais uma vez, estamos lançando mão do Princípio Prefira Interfaces a Classes.

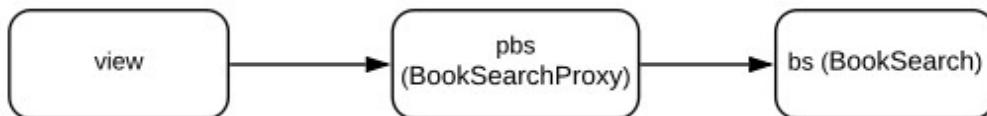
O próximo código ilustra a instanciação do proxy. Primeiro, mostramos o código antes do proxy. Nesse código (a seguir), um objeto BookSearch é criado no programa principal e depois passado como parâmetro de qualquer classe ou função que precise do serviço de pesquisa de livros, como a classe View.

```
void main() {  
    BookSearch bs = new BookSearch();  
    ...  
    View view = new View(bs);  
    ...  
}
```

Com a decisão de usar um proxy, vamos ter que modificar esse código para instanciar o proxy (código a seguir). Além disso, View passou a receber como parâmetro de sua construtora uma referência para o proxy, em vez de uma referência para o objeto base.

```
void main() {  
    BookSearch bs = new BookSearch();  
    BookSearchProxy pbs;  
    pbs = new BookSearchProxy(bs);  
    ...  
    View view = new View(pbs);  
    ...  
}
```

A próxima figura ilustra os objetos e as referências entre eles, considerando a solução que usa um proxy:



Padrão de projeto Proxy

Além de ajudar na implementação de caches, proxies podem ser usados para implementar outros requisitos não-funcionais. Alguns exemplos incluem:

- Comunicação com um cliente remoto, isto é, pode-se usar um proxy para encapsular protocolos e detalhes de comunicação. Esses proxies são chamados de **stubs**.
- Alocação de memória por demanda para objetos que consomem muita memória. Por exemplo, uma classe pode manipular uma imagem em alta resolução. Então, podemos usar um proxy para evitar que a imagem fique carregada o tempo todo na memória principal. Ela somente será carregada, possivelmente do disco, antes da execução de alguns métodos.
- Controlar o acesso de diversos clientes a um objeto base. Por exemplo, os clientes devem estar autenticados e ter permissão para executar certas operações do objeto base. Com isso, a classe do objeto base pode se concentrar na implementação de requisitos funcionais.

## Adaptador

**Contexto:** Suponha um sistema que tenha que controlar projetores multimídia. Para isso ele deve instanciar objetos de classes fornecidas pelos fabricantes de cada projetor, como ilustrado a seguir:

```
class ProjetoSamsung {  
    public void turnOn() { ... }  
    ...  
}  
  
class ProjetoLG {  
    public void enable(int timer) { ... }  
    ...  
}
```

Para simplificar, estamos mostrando apenas duas classes. Porém, um cenário real pode envolver classes de outros fabricantes de projetores. Também estamos mostrando apenas um método de cada classe, mas elas podem conter outros métodos. Particularmente, o método mostrado é responsável

por ligar o projetor. No caso dos projetores da Samsung, esse método não possui parâmetros. No caso dos projetores da LG podemos passar um intervalo em minutos para ligação do projetor. Se esse parâmetro for igual a zero, o projetor é ligado imediatamente. Veja ainda que o nome dos métodos é diferente nas duas classes.

**Problema:** No sistema de controle de projetores multimídia, queremos usar uma interface única para ligar os projetores, independentemente de marca. O próximo código mostra essa interface e uma classe cliente do sistema:

```
interface Projetor {  
    void liga();  
}  
...  
class SistemaControleProjetores {  
    void init(Projetor projetor) {  
        projetor.liga(); // liga qualquer projetor  
    }  
}
```

Porém, as classes de cada projetor — mostradas anteriormente — foram implementadas pelos seus fabricantes e estão prontas para uso. Ou seja, não temos acesso ao código dessas classes para fazer com que elas implementem a interface `Projetor`.

**Solução:** O padrão de projeto **Adaptador** — também conhecido como **Wrapper** — é uma solução para o nosso problema. Recomenda-se usar esse padrão quando temos que converter a interface de uma classe para outra interface, esperada pelos seus clientes. No nosso exemplo, ele pode ser usado para converter a interface `Projetor` — usada no sistema de controle de projetores — para as interfaces (métodos públicos) das classes implementadas pelos fabricantes dos projetores.

Um exemplo de classe adaptadora, de `ProjetorSamsung` para `Projetor`, é mostrado a seguir:

```
class AdaptadorProjetorSamsung implements Projetor {
```

```

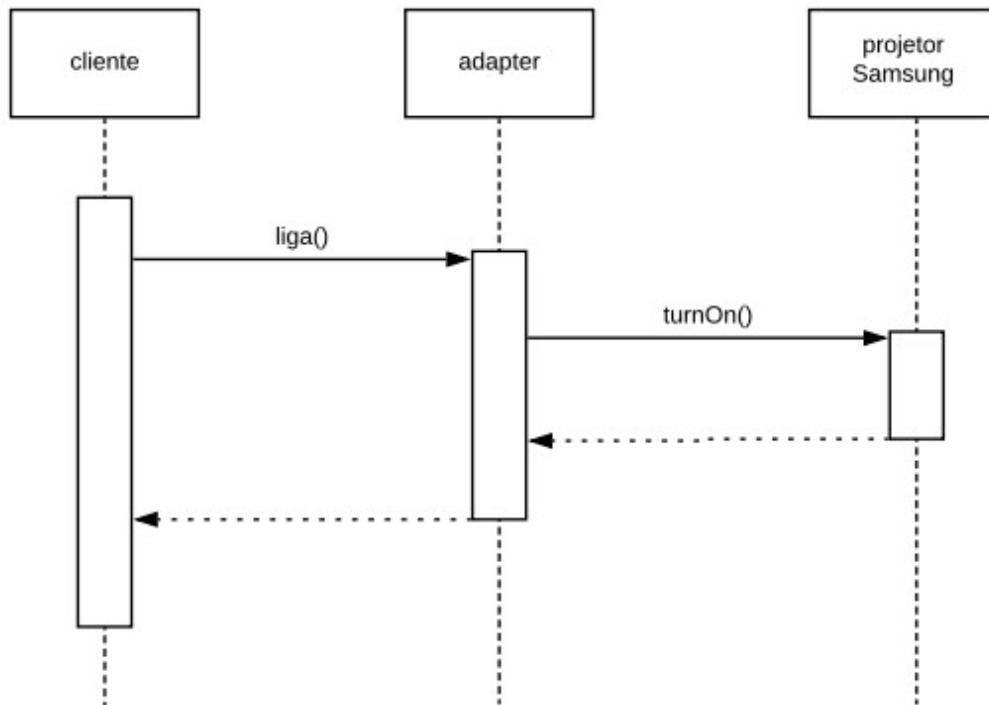
private ProjetorSamsung projetor;

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

public void liga() {
    projetor.turnOn();
}
}

```

A classe `AdaptadorProjetorSamsung` implementa a interface `Projetor`. Logo, objetos dessa classe podem ser passados como parâmetro do método `init()` do sistema para controle de projetores. A classe `AdaptadorProjetorSamsung` também possui um atributo privado do tipo `ProjetorSamsung`. A sequência de chamadas é então a seguinte (acompanhe também pelo diagrama de sequência UML, mostrado na próxima página): primeiro, o cliente — no nosso caso, representado pelo método `init` — chama `liga()` da classe adaptadora; em seguida, a execução desse método chama o método equivalente — no caso, `turnOn()` — do objeto que está sendo adaptado; isto é, um objeto que acessa projetores Samsung.



Padrão de projeto Adaptador

Se quisermos manipular projetores LG, vamos ter que implementar uma segunda classe adaptadora. No entanto, seu código será parecido com `AdaptadorProjetoSamsung`.

## Fachada

**Contexto:** Suponha que implementamos um interpretador para uma linguagem X. Esse interpretador permite executar programas X a partir de uma linguagem hospedeira, no caso Java. Se quiser tornar o exemplo mais real, imagine que X é uma linguagem para consulta a dados, semelhante a SQL. Para executar programas X, a partir de um código em Java, os seguintes passos são necessários:

```
Scanner s = new Scanner("prog1.x");
Parser p = new Parser(s);
AST ast = p.parse();
CodeGenerator code = new CodeGenerator(ast);
code.eval();
```

**Problema:** Como a linguagem X está ficando popular, os desenvolvedores estão reclamando da complexidade do código acima, pois ele requer conhecimento de classes internas do interpretador de X. Logo, os usuários frequentemente pedem uma interface mais simples para chamar o interpretador da linguagem X.

**Solução:** O padrão de projeto **Fachada** é uma solução para o nosso problema. Uma Fachada é uma classe que oferece uma interface mais simples para um sistema. O objetivo é evitar que os usuários tenham que conhecer classes internas desse sistema; em vez disso, eles precisam interagir apenas com a classe de Fachada. As classes internas ficam encapsuladas por trás dessa Fachada.

No nosso problema, a Fachada poderia ser:

```
class InterpretadorX {
    private String arq;
    InterpretadorX(arq) {
        this.arq = arq;
```

```

    }

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

```

Assim, os desenvolvedores que precisam executar programas X, a partir de Java, poderão fazê-lo por meio de uma única linha de código:

```
new InterpretadorX("prog1.x").eval();
```

Antes de implementar a fachada, os clientes precisavam criar três objetos de tipos internos do interpretador e chamar dois métodos. Agora, basta criar um único objeto e chamar eval.

## Decorador

**Contexto:** Vamos voltar ao sistema de comunicação remota usado para explicar o Padrão Fábrica. Suponha que as classes TCPChannel e UDPChannel implementam uma interface Channel:

```

interface Channel {
    void send(String msg);
    String receive();
}

class TCPChannel implements Channel {
    ...
}

class UDPChannel implements Channel {
    ...
}

```

**Problema:** Os clientes dessas classes precisam adicionar funcionalidades extras em canais, tais como buffers, compactação das mensagens, log das mensagens trafegadas, etc. Mas essas funcionalidades são opcionais: dependendo do cliente precisamos de apenas algumas funcionalidades ou,

talvez, nenhuma delas. Uma primeira solução consiste no uso de herança para criar subclasses com cada possível seleção de funcionalidades. No quadro abaixo, mostramos algumas das subclasses que teríamos que criar (extends significa relação de herança):

- TCPZipChannel extends TCPChannel
- TCPBufferedChannel extends TCPChannel
- TCPBufferedZipChannel extends TCPZipChannel extends TCPChannel
- TCPLogChannel extends TCPChannel
- TCPLogBufferedZipChannel extends TCPBufferedZipChannel  
extends TCPZipChannel extends TCPChannel
- UDPZipChannel extends UDPChannel
- UDPBufferedChannel extends UDPChannel
- UDPBufferedZipChannel extends UDPZipChannel extends UDPChannel
- UDPLogChannel extends UDPChannel
- UDPLogBufferedZipChannel extends UDPBufferedZipChannel  
extends UDPZipChannel extends UDPChannel

Nessa solução, usamos herança para implementar subclasses para cada conjunto de funcionalidades. Suponha que o usuário precise de um canal UDP com buffer e compactação. Para isso, tivemos que implementar UDPBufferedZipChannel como subclasse de UDPZipChannel, que por sua vez foi implementada como subclasse de UDPChannel. Como o leitor deve ter percebido, uma solução via herança é quase que inviável, pois ela gera

uma explosão combinatória do número de classes relacionadas com canais de comunicação.

**Solução:** O Padrão Decorador representa uma alternativa a herança quando se precisa adicionar novas funcionalidades em uma classe base. Em vez de usar herança, usa-se composição para adicionar tais funcionalidades dinamicamente nas classes base. Portanto, Decorador é um exemplo de aplicação do princípio de projeto Prefira Composição a Herança, que estudamos no capítulo anterior.

No nosso problema, ao optarmos por decoradores, o cliente poderá configurar um Channel da seguinte forma:

```
channel = new ZipChannel(new TCPChannel());  
// TCPChannel que compacte/descompacte dados
```

```
channel = new BufferChannel(new TCPChannel());  
// TCPChannel com um buffer associado
```

```
channel = new BufferChannel(new UDPChannel());  
// UDPChannel com um buffer associado
```

```
channel = new BufferChannel(new ZipChannel(new TCPChannel()));  
// TCPChannel com compactação e um buffer associado
```

Portanto, em uma solução com decoradores, a configuração de um Channel é feita no momento da sua instanciação, por meio de uma sequência aninhada de operadores new. O new mais interno sempre cria uma classe base, no nosso exemplo TCPChannel ou UDPChannel. Feito isso, os operadores mais externos são usados para decorar o objeto criado com novas funcionalidades.

Falta então explicar as classes que são os decoradores propriamente ditos, como ZipChannel e BufferChannel. Primeiro, elas são subclasses da seguinte classe que não aparece no exemplo, mas que é fundamental para o funcionamento do padrão Decorador:

```
class ChannelDecorator implements Channel {  
  
    private Channel channel;  
  
    public ChannelDecorator(Channel channel) {  
        this.channel = channel;
```

```

    }

    public void send(String msg) {
        channel.send(msg);
    }

    public String receive() {
        return channel.receive();
    }
}

```

Essa classe tem duas características importantes:

- Ela é uma `Channel`, isto é, ela implementa essa interface e, portanto, os seus dois métodos. Assim, sempre que for esperado um objeto do tipo `Channel` podemos passar um objeto do tipo `ChannelDecorator` no lugar.
- Ela possui internamente um objeto do tipo `Channel` para o qual delega as chamadas aos métodos `send` e `receive`. Em outras palavras, um decorador, no nosso caso, vai sempre referenciar um outro decorador. Após implementar a funcionalidade que lhe cabe — um buffer, compactação, etc. — ele repassa a chamada para esse decorador.

Por fim, chegamos aos decoradores reais. Eles são subclasses de `ChannelDecorator`, como no código a seguir, que implementa um decorador que compacta e descompacta as mensagens trafegadas pelo canal:

```

class ZipChannel extends ChannelDecorator {

    public ZipChannel(Channel c) {
        super(c);
    }

    public void send(String msg) {
        "compacta mensagem msg"
        super.send(msg);
    }

    public String receive() {
        String msg = super.receive();
        "descompacta mensagem msg"
    }
}

```

```
    return msg;  
}  
  
}
```

Para entender o funcionamento de `ZipChannel`, suponha o seguinte código:

```
Channel c = new ZipChannel(new TCPChannel());  
c.send("Hello, world")
```

A chamada de `send` na última linha do exemplo dispara as seguintes execuções de métodos:

- Primeiro, executa-se `ZipChannel.send`, que vai compactar a mensagem.
- Após a compactação, `ZipChannel.send` chama `super.send`, que vai executar `ChannelDecorator.send`, pois `ChannelDecorator` é a superclasse da classe `ZipChannel`.
- `ChannelDecorator.send` apenas repassa a chamada para o `Channel` por ele referenciado, que no caso é um `TCPChannel`.
- Finalmente, chegamos a `TCPChannel.send`, que vai transmitir a mensagem via TCP.

**Código Fonte:** O código do exemplo de Decorador usado nesta seção está disponível neste [link](#).

## Strategy

**Contexto:** Suponha que estamos implementando um pacote de estruturas de dados, com a seguinte classe lista:

```
class MyList {  
    ... // dados de uma lista  
    ... // métodos de uma lista: add, delete, search  
  
    public void sort() {
```

```
    ... // ordena a lista usando Quicksort
}
}
```

**Problema:** os nossos clientes estão solicitando que novos algoritmos de ordenação possam ser usados para ordenar os elementos da lista. Explicando melhor, eles querem ter a opção de alterar e definir, por conta própria, o algoritmo de ordenação. No entanto, a versão atual da classe sempre ordena a lista usando o algoritmo Quicksort. Se lembarmos dos princípios de projeto que estudamos no capítulo anterior, podemos dizer que a classe `MyList` não segue o princípio Aberto/Fechado, considerando o algoritmo de ordenação.

**Solução:** o Padrão **Strategy** é a solução para o nosso problema de abrir a classe `MyList` para novos algoritmos de ordenação, mas sem alterar o seu código fonte. O objetivo do padrão é parametrizar os algoritmos usados por uma classe. Ele prescreve como encapsular uma família de algoritmos e como torná-los intercambiáveis. Assim, seu uso é recomendado quando uma classe é usuária de um certo algoritmo (de ordenação, no nosso exemplo). Porém, como existem diversos algoritmos com esse propósito, não se quer antecipar uma decisão e implementar apenas um deles no corpo da classe, como ocorre na primeira versão de `MyList`.

Mostra-se a seguir o novo código de `MyList`, usando o Padrão **Strategy** para configuração do algoritmo de ordenação:

```
class MyList {
    ... // dados de uma lista
    ... // métodos de uma lista: add, delete, search

    private SortStrategy strategy;

    public MyList() {
        strategy = new QuickSortStrategy();
    }

    public void setSortStrategy(SortStrategy strategy) {
        this.strategy = strategy;
    }

    public void sort() {
```

```

        strategy.sort(this);
    }
}

```

Nessa nova versão, o algoritmo de ordenação transformou-se em um atributo da classe `MyList` e um método `set` foi criado para configurar esse algoritmo. O método `sort` repassa a tarefa de ordenação para um método de mesmo nome do objeto com a estratégia de ordenação. Nessa chamada, passa-se `this` como parâmetro, pois o algoritmo a ser executado deve ter acesso à lista para ordenar seus elementos.

Para encerrar a apresentação do padrão, mostramos o código das classes que implementam as estratégias — isto é, os algoritmos — de ordenação:

```

abstract class SortStrategy {
    abstract void sort(MyList list);
}

class QuickSortStrategy extends SortStrategy {
    void sort(MyList list) { ... }
}

class ShellSortStrategy extends SortStrategy {
    void sort(MyList list) { ... }
}

```

## Observador

**Contexto:** Suponha que estamos implementando um sistema para controlar uma estação meteorológica. Nesse sistema, temos que manipular objetos de duas classes: `Temperatura`, que são objetos de modelo que armazenam as temperaturas monitoradas na estação meteorológica; e `Termometro`, que é uma classe usada para criar objetos visuais que exibem as temperaturas sob monitoramento. Termômetros devem exibir a temperatura atual que foi monitorada. Se a temperatura mudar, os termômetros devem ser atualizados.

**Problema:** Não queremos acoplar `Temperatura` (classe de modelo) a `Termometro` (classe de interface). O motivo é simples: classes de interface mudam com frequência. Na versão atual, o sistema possui uma interface textual, que exibe temperaturas em Celsius no console do sistema operacional. Mas, em breve, pretendemos ter interfaces Web, para celulares

e para outros sistemas. Pretendemos também oferecer outras interfaces de termômetros, tais como digital, analógico, etc. Por fim, temos mais classes semelhantes a Temperatura e Termometro em nosso sistema, tais como: PressaoAtmosferica e Barometro, UmidadeDoAr e Higrometro, VelocidadeDoVento e Anemometro, etc. Logo, na medida do possível, gostaríamos de reusar o mecanismo de notificação também nesses outros pares de classes.

**Solução:** O padrão **Observador** é a solução recomendada para o nosso contexto e problema. Esse padrão define como implementar uma relação do tipo um-para-muitos entre objetos sujeito e observadores. Quando o estado de um sujeito muda, seus observadores devem ser notificados.

Primeiro, vamos mostrar um programa principal para o nosso problema:

```
void main() {
    Temperatura t = new Temperatura();
    t.addObserver(new TermometroCelsius());
    t.addObserver(new TermometroFahrenheit());
    t.setTemp(100.0);
}
```

Esse programa cria um objeto do tipo Temperatura (um sujeito) e então adiciona dois observadores nele: um TermometroCelsius e um TermometroFahrenheit. Por fim, define-se o valor da temperatura para 100 graus Celsius. A suposição é que temperaturas são, por *default*, monitoradas na escala Celsius.

As classes Temperatura e TermometroCelsius são mostradas a seguir:

```
class Temperatura extends Subject {

    private double temp;

    public double getTemp() {
        return temp;
    }

    public void setTemp(double temp) {
        this.temp = temp;
        notifyObservers();
    }
}
```

```

}

class TermometroCelsius implements Observer {

    public void update(Subject s){
        double temp = ((Temperatura) s).getTemp();
        System.out.println("Temperatura Celsius: " + temp);
    }
}

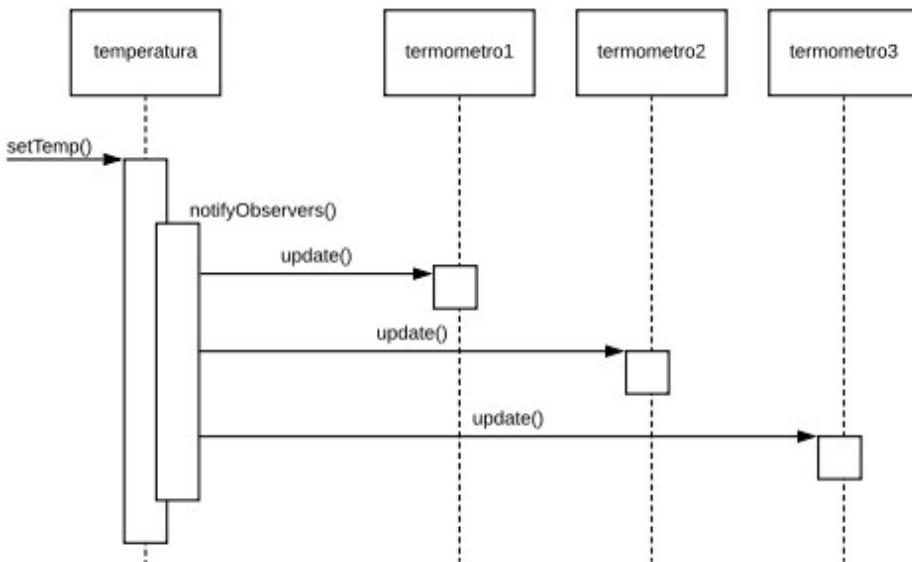
```

Veja que `Temperatura` herda de uma classe chamada `Subject`. Na solução proposta, todos os sujeitos devem estender essa classe. Ao fazer isso, eles herdam dois métodos:

- `addObserver`. No exemplo, esse método é usado no programa principal para adicionar dois termômetros como observadores de uma instância de `Temperatura`.
- `notifyObservers`. No exemplo, esse método é chamado por `Temperatura` para notificar seus observadores de que o seu valor foi alterado no método `setTemp`.

A implementação de `notifyObservers` — que é omitida no exemplo — chama o método `update` dos objetos que se registraram como observadores de uma determinada instância de `Temperatura`. O método `update` faz parte da interface `Observer`, que deve ser implementada por todo observador, como é o caso de `TermometroCelsius`.

A figura da próxima página mostra um diagrama de sequência UML que ilustra a comunicação entre uma temperatura (sujeito) e três possíveis termômetros (observadores). Assume-se que os três termômetros estão registrados como observadores da temperatura. A sequência de chamadas começa com temperatura recebendo uma chamada para executar `setTemp()`.



## Padrão de projeto Observador

O padrão Observador possui as seguintes vantagens principais:

- Ele não acopla os sujeitos a seus observadores. Na verdade, os sujeitos — como Temperatura, no exemplo — não conhecem os seus observadores. De forma genérica, os sujeitos publicam um evento anunciando a mudança de seu estado — chamando `notifyObservers` — e os observadores interessados são notificados. Esse comportamento facilita o reúso dos sujeitos em diversos cenários e, também, a implementação de diversos tipos de observadores para o mesmo tipo de sujeito.
- Uma vez implementado, o padrão Observador disponibiliza um mecanismo de notificação que pode ser reusado por diferentes pares de sujeito-observador. Por exemplo, podemos reusar a classe `Subject` e a interface `Observer` para notificações envolvendo pressão atmosférica e barômetros, umidade do ar e higrômetros, velocidade do vento e anemômetros, etc.

**Código Fonte:** Se quiser conferir o código completo do nosso exemplo de Observador, incluindo o código das classes `Subject` e da interface `Observer`, acesse o seguinte [link](#).

# Template Method

**Contexto:** Suponha que estamos desenvolvendo uma folha de pagamento. Nela, temos uma classe Funcionario, com duas subclasses: FuncionarioPublico e FuncionarioCLT.

**Problema:** Pretendemos padronizar um modelo (ou template) para cálculo dos salários na classe base Funcionario, que possa depois ser herdado pelas suas subclasses. Assim, as subclasses terão apenas que adaptar a rotina de cálculo de salários às suas particularidades. Mais especificamente, as subclasses saberão exatamente os métodos que precisam implementar para calcular o salário de um funcionário.

**Solução:** O padrão de projeto **Template Method** resolve o problema que enunciamos. Ele especifica como implementar o esqueleto de um algoritmo em uma classe abstrata X, mas deixando pendente alguns passos — ou métodos abstratos. Esses métodos serão implementados nas subclasses de X. Em resumo, um Template Method permite que subclasses customizem um algoritmo, mas sem mudar a sua estrutura geral implementada na classe base.

Um exemplo de Template Method para o nosso contexto e problema é mostrado a seguir:

```
abstract class Funcionario {  
  
    double salario;  
    ...  
    abstract double calcDescontosPrevidencia();  
    abstract double calcDescontosPlanoSaude();  
    abstract double calcOutrosDescontos();  
  
    public double calcSalarioLiquido() { // template method  
        double prev = calcDescontosPrevidencia();  
        double saude = calcDescontosPlanoSaude();  
        double outros = calcOutrosDescontos();  
        return salario - prev - saude - outros;  
    }  
}
```

Nesse exemplo, `calcSalarioLiquido` é um método template para cálculo do salário de funcionários. Ele padroniza que temos que calcular três descontos: para a previdência, para o plano de saúde do funcionário e outros descontos. Feito isso, o salário líquido é o salário do funcionário subtraído desses três descontos. Porém, em `Funcionario`, não sabemos ainda como calcular os descontos, pois eles variam conforme o tipo de funcionário (público ou CLT). Logo, são criados métodos abstratos para representar cada um desses passos da rotina de cálculo de salários. Como eles são abstratos, a classe `Funcionario` também foi declarada como abstrata. Como o leitor já deve ter percebido, subclasses de `Funcionario` — como `FuncionarioPublico` e `FuncionarioCLT` — vão herdar o método `calcSalarioLiquido`, que não precisará sofrer nenhuma modificação. No entanto, caberá às subclasses implementar os três passos (métodos) abstratos: `calcDescontosPrevidencia`, `calcDescontosPlanoSaude` e `calcOutrosDescontos`.

Métodos template permitem que código antigo chame código novo. No exemplo, a classe `Funcionario` provavelmente foi implementada antes de `FuncionarioPublico` e `FuncionarioCLT`. Logo, dizemos que `Funcionario` é mais antiga do que as suas subclasses. Mesmo assim, `Funcionario` inclui um método que vai chamar código novo, implementado nas subclasses. Esse recurso de sistemas orientados a objetos é chamado de **inversão de controle**. Ele é fundamental, por exemplo, para implementação de **frameworks**, isto é, aplicações semi-prontas, que antes de serem usadas devem ser customizadas por seus clientes. Apesar de não ser o único instrumento disponível para esse fim, métodos template constituem uma alternativa interessante para que um cliente implemente o código faltante em um framework.

## Visitor

**Contexto:** Suponha o sistema de estacionamentos que usamos no Capítulo 5. Suponha que nesse sistema existe uma classe `Veiculo`, com subclasses `carro`, `Onibus` e `Motocicleta`. Essas classes são usadas para armazenar informações sobre os veículos estacionados no estacionamento. Suponha ainda que todos esses veículos estão armazenados em uma lista. Dizemos que essa lista é uma estrutura de dados **polimórfica**, pois ela pode armazenar objetos de classes diferentes, desde que eles sejam subclasses de `Veiculo`.

**Problema:** Com frequência, no sistema de estacionamentos, temos que realizar uma operação em todos os veículos estacionados. Como exemplo, podemos citar: imprimir informações sobre os veículos estacionados, persistir os dados dos veículos ou enviar uma mensagem para os donos dos veículos.

No entanto, o objetivo é implementar essas operações fora das classes de Veiculo por meio de um código como o seguinte:

```
interface Visitor {  
    void visit(Carro c);  
    void visit(Onibus o);  
    void visit(Motocicleta m);  
}  
  
class PrintVisitor implements Visitor {  
    public void visit(Carro c) { "imprime dados de carro" }  
    public void visit(Onibus o) { "imprime dados de onibus" }  
    public void visit(Motocicleta m) {"imprime dados de moto"}  
}
```

Nesse código, a classe PrintVisitor inclui métodos que imprimem os dados de um Carro, Onibus e Motocicleta. Uma vez implementada essa classe, gostaríamos de usar o seguinte código para visitar todos os veículos do estacionamento::

```
PrintVisitor visitor = new PrintVisitor();  
foreach (Veiculo veiculo: listaDeVeiculosEstacionados) {  
    visitor.visit(veiculo); // erro de compilação  
}
```

No entanto, no código mostrado, o método visit a ser chamado depende do tipo dinâmico do objeto alvo da chamada (visitor) e do tipo dinâmico de um parâmetro (veiculo). Porém, em linguagens como Java, C++ ou C# apenas o tipo do objeto alvo da chamada é considerado na escolha do método a ser chamado. Dizendo de outro modo, em Java e em linguagens similares, o compilador somente conhece o tipo estático de veiculo, que é Veiculo. Por isso, ele não consegue inferir qual implementação de visit deve ser chamada.

Para ficar mais claro, o seguinte erro ocorre ao compilar o código anterior:

```

visitor.visit(veiculo);
^
method PrintVisitor.visit(Carro) is not applicable
  (argument mismatch; Veiculo cannot be converted to Carro)
method PrintVisitor.visit(Onibus) is not applicable
  (argument mismatch; Veiculo cannot be converted to Onibus)

```

Na verdade, esse código somente compila em linguagens que oferecem **despacho duplo** de chamadas de métodos (*double dispatch*). Nessas linguagens, os tipos do objeto alvo e de um dos parâmetros de chamada são usados para escolher o método que será invocado. No entanto, despacho duplo somente está disponível em linguagens mais antigas e menos conhecidas hoje em dia, como Common Lisp.

Portanto, o nosso problema é o seguinte: como simular *double dispatch* em uma linguagem como Java? Se conseguirmos fazer isso, poderemos contornar o erro de compilação que ocorre no código que mostramos.

**Solução:** A solução para o nosso problema consiste em usar o padrão de projeto **Visitor**. Esse padrão define como adicionar uma operação em uma família de objetos, sem que seja preciso modificar as classes dos mesmos. Além disso, o padrão Visitor deve funcionar mesmo em linguagens com *single dispatching* de métodos, como Java.

Como primeiro passo, temos que implementar um método `accept` em cada classe da hierarquia. Na classe raiz, ele é abstrato. Nas subclasses, ele recebe como parâmetro um objeto do tipo `Visitor`. E a sua implementação apenas chama o método `visit` desse `Visitor`, passando `this` como parâmetro. Porém, como a chamada ocorre no corpo de uma classe, o compilador conhece o tipo de `this`. Por exemplo, na classe `Carro`, o compilador sabe que o tipo de `this` é `Carro`. Logo, ele sabe que deve chamar a implementação de `visit` que tem `Carro` como parâmetro. Para ser preciso, o método exato a ser chamado depende do tipo dinâmico do objeto alvo da chamada (`v`). Porém, isso não é um problema, pois significa que temos um caso de *single dispatch*, que é permitido em linguagens como Java.

```

abstract class Veiculo {
    abstract public void accept(Visitor v);
}

class Carro extends Veiculo {

```

```

...
    public void accept(Visitor v) {
        v.visit(this);
    }
    ...
}

class Onibus extends Veiculo {
    ...
    public void accept(Visitor v) {
        v.visit(this);
    }
    ...
}

// Idem para Motocicleta

```

Por último, temos que modificar o laço que percorre a lista de veículos estacionados. Agora, chamaremos os métodos `accept` de cada veículo, passando o visitor como parâmetro.

```

PrintVisitor visitor = new PrintVisitor();
foreach (Veiculo veiculo: listaDeVeiculosEstacionados) {
    veiculo.accept(visitor);
}

```

Resumindo, visitors facilitam a adição de um método em uma hierarquia de classes. Um visitor congrega operações relacionadas — no exemplo, impressão de dados de `Veiculo` e de suas subclasses. Mas poderia também existir um segundo visitor, com outras operações — por exemplo, persistir os objetos em disco. Por outro lado, a adição de uma nova classe na hierarquia, por exemplo, `Caminhao`, requer a atualização de todos os visitors com um novo método: `visit(Caminhao)`.

Antes de concluir, é importante mencionar que visitors possuem uma desvantagem importante: eles podem forçar uma quebra no encapsulamento das classes que serão visitadas. Por exemplo, `Veiculo` pode ter que implementar métodos públicos expondo seu estado interno para que os visitors tenham acesso a eles.

**Código Fonte:** O código do exemplo de Visitor usado nesta seção está disponível neste [link](#).

# Outros Padrões de Projeto

**Iterador** é um padrão de projeto que padroniza uma interface para caminhar sobre uma estrutura de dados. Normalmente, essa interface inclui métodos como `hasNext()` e `next()`, como mostrado no seguinte exemplo:

```
List<String> list = Arrays.asList("a", "b", "c");
Iterator it = list.iterator();
while(it.hasNext()) {
    String s = (String) it.next();
    System.out.println(s);
}
```

Um iterador permite percorrer uma estrutura de dados sem conhecer o seu tipo concreto. Em vez disso, basta conhecer os métodos da interface `Iterator`. Iteradores também permitem que múltiplos caminhamentos sejam realizadas de forma simultânea em cima da mesma estrutura de dados.

**Builder** é um padrão de projeto que facilita a instanciação de objetos que têm muitos atributos, sendo alguns deles opcionais. Se o valor desses atributos opcionais não for informado, eles devem ser inicializados com um valor *default*. Em vez de criar diversos construtores, um método para cada combinação possível de parâmetros, podemos delegar o processo de inicialização dos campos de um objeto para uma classe `Builder`. Um exemplo é mostrado a seguir, para uma classe `Livro`.

```
Livro esm = new Livro.Builder().
    setNome("Engenharia Soft Moderna").
    setEditora("UFMG").setAno(2020).build();

Livro gof = new Livro.Builder().setName("Design Patterns").
    setAutores("GoF").setAno(1995).build();
```

Uma primeira alternativa ao uso de um `Builder` seria implementar a instanciação por meio de construtores. Porém, teríamos que criar diversos construtores, pois `Livro` possui diversos atributos, nem todos obrigatórios. Além disso, a chamada desses construtores poderia gerar confusão, pois o desenvolvedor teria que conhecer exatamente a ordem dos diversos parâmetros. Com o padrão `Builder`, os métodos `set` deixam claro qual atributo de `Livro` está sendo inicializado. Uma segunda alternativa seria

implementar os métodos set diretamente na classe `Livro`. Porém, isso quebraria o princípio de ocultamento da informação, pois tornaria possível alterar, a qualquer momento, qualquer atributo da classe. Por outro lado, com um `Builder`, os atributos somente podem ser definidos em tempo de instanciação da classe.

Em tempo, a versão de `Builder` que apresentamos não corresponde à descrição original do padrão contida no livro da Gangue dos Quatro. Em vez disso, apresentamos uma versão proposta por Joshua Bloch ([link](#)). Acreditamos que essa versão, hoje em dia, corresponde ao uso mais comum de `Builders`. Ela é usada, por exemplo, em classes da API de Java, como `Calendar.Builder` ([link](#)).

**Código Fonte:** O código do exemplo de `Builder` — incluindo as classes `Livro` e `Livro.Builder` — está disponível neste [link](#). Ao estudá-lo, você perceberá que `Livro.Builder` é uma classe interna, pública e estática de `Livro`. Por isso, é que podemos chamar `new Livro.Builder()` diretamente, sem precisar de instanciar antes um objeto do tipo `Livro`.

## Quando Não Usar Padrões de Projeto?

Padrões de projeto têm como objetivo tornar o projeto de um sistema mais flexível. Por exemplo, fábricas facilitam trocar o tipo dos objetos manipulados por um programa. Um decorador permite personalizar uma classe com novas funcionalidades, tornando-a flexível a outros cenários de uso. O padrão `Strategy` permite configurar os algoritmos usados por uma classe, apenas para citar alguns exemplos.

Porém, como quase tudo em Computação, o uso de padrões também tem um custo. Por exemplo, uma fábrica requer a implementação de pelo menos mais uma classe no sistema. Para citar um segundo exemplo, `Strategy` requer a criação de uma classe abstrata e mais uma classe para cada algoritmo. Por isso, a adoção de padrões de projeto exige uma análise cuidadosa. Para ilustrar esse tipo de análise, vamos continuar a usar os exemplos de Fábrica e `Strategy`:

- Antes de usar uma fábrica, devemos fazer (e responder) a seguinte pergunta: vamos mesmo precisar criar objetos de tipos diferentes no nosso sistema? Existem boas chances de que tais objetos sejam, de fato, necessários? Se sim, então vale a pena usar uma Fábrica para encapsular a criação de tais objetos. Caso contrário, é melhor criar os objetos usando o operador new, que é a solução nativa para criação de objetos em linguagens como Java.
- De forma semelhante, antes de incluir o padrão Strategy em uma certa classe devemos nos perguntar: vamos mesmo precisar de parametrizar os algoritmos usados na implementação dessa classe? Existem, de fato, usuários que vão precisar de algoritmos alternativos? Se sim, vale a pena usar o padrão Strategy. Caso contrário, é preferível implementar o algoritmo diretamente no corpo da classe.

Apesar de usarmos apenas dois padrões como exemplo, perguntas semelhantes podem ser feita para outros padrões.

No entanto, em muitos sistemas observa-se um uso exagerado de padrões de projeto, em situações nas quais os ganhos de flexibilidade e extensibilidade são questionáveis. Existe até um termo para se referir a essa situação: **paternite**, isto é, uma inflamação associada ao uso precipitado de padrões de projeto.

John Ousterhout tem um comentário relacionado a essa doença:

O maior risco de padrões de projetos é a sua super-aplicação (*over-application*). Nem todo problema precisa ser resolvido por meio dos padrões de projeto; por isso, não tente forçar um problema a caber em um padrão de projeto quando uma abordagem tradicional funcionar melhor. O uso de padrões de projeto não necessariamente melhora o projeto de um sistema de software; isso só acontece se esse uso for justificado. Assim como ocorre com outros conceitos, a noção de que padrões de projetos são uma boa coisa não significa que quanto mais padrões de projeto usarmos, melhor será nosso sistema.

Ousterhout ilustra seu argumento citando o emprego de decoradores durante a abertura de arquivos em Java, como mostrado no seguinte trecho de

código:

```
FileInputStream fs = new FileInputStream(fileName);
BufferedInputStream bs = new BufferedInputStream(fs);
ObjectInputStream os = new ObjectInputStream(bs);
```

Segundo Ousterhout, decoradores adicionam complexidade desnecessária ao processo de criação de arquivos em Java. O principal motivo é que, via de regra, iremos sempre nos beneficiar de um buffer ao abrir qualquer arquivo. Portanto, buffers de entrada/saída deveriam ser oferecidos por *default*, em vez de por meio de uma classe decoradora específica. Assim, e de acordo com esse raciocínio, as classes `FileInputStream` e `BufferedInputStream` poderiam ser fundidas em uma única classe.

## Bibliografia

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

Joshua Bloch. Effective Java. 3rd edition. Prentice Hall, 2017.

Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra. Head First Design Patterns: A Brain-Friendly Guide. O'Reilly, 2004.

Eduardo Guerra. Design Patterns com Java: Projeto Orientado a Objetos guiado por Padrões. Caso do Código, 2014.

Fernando Pereira, Marco Túlio Valente, Roberto Bigonha, Mariza Bigonha. Arcademis: A Framework for Object Oriented Communication Middleware Development. Software: Practice and Experience, 2006.

Fábio Tirelo, Roberto Bigonha, Mariza Bigonha, Marco Túlio Valente. Desenvolvimento de Software Orientado por Aspectos. XXIII Jornada de Atualização em Informática (JAI), 2004.

## Exercícios de Fixação

1. (ENADE 2011, adaptado) Sobre padrões de projeto, assinale V ou F.

( ) Prototype é um tipo de padrão estrutural.

( ) Singleton tem por objetivo garantir que uma classe tenha ao menos uma instância e fornecer um ponto global de acesso para ela.

( ) Template Method tem por objetivo definir o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses.

( ) Iterator fornece uma maneira de acessar sequencialmente os elementos de um objeto agregado sem expor sua representação subjacente.

2. Dê o nome dos seguintes padrões de projeto:

1. Oferece uma interface unificada e de alto nível que torna mais fácil o uso de um sistema.
2. Garante que uma classe possui, no máximo, uma instância e oferece um ponto único de acesso a ela.
3. Facilita a construção de objetos complexos com vários atributos, sendo alguns deles opcionais.
4. Converte a interface de uma classe para outra interface esperada pelos clientes. Permite que classes trabalhem juntas, o que não seria possível devido à incompatibilidade de suas interfaces.
5. Oferece uma interface ou classe abstrata para criação de uma família de objetos relacionados.
6. Oferece um método para centralizar a criação de um tipo de objeto.
7. Funciona como um intermediário que controla o acesso a um objeto base.
8. Permite adicionar dinamicamente novas funcionalidades a uma classe.

9. Oferece uma interface padronizada para caminhar em estruturas de dados.
  10. Permite parametrizar os algoritmos usados por uma classe.
  11. Torna uma estrutura de dados aberta a extensões, isto é, permite adicionar uma função em cada elemento de uma estrutura de dados, mas sem alterar o código de tais elementos.
  12. Permite que um objeto avise outros objetos de que seu estado mudou.
  13. Define o esqueleto de um algoritmo em uma classe base e delega a implementação de alguns passos para subclasses.
3. Dentre os padrões de projeto que respondeu na questão (2), quais são padrões criacionais?
4. Considerando as respostas da questão (2), liste padrões de projeto que:
1. Ajudam a tornar uma classe aberta a extensões, sem que seja preciso modificar o seu código fonte (isto é, padrões que colocam em prática o princípio Aberto/Fechado).
  2. Ajudam a desacoplar dois tipos de classes.
  3. Ajudam a incrementar a coesão de uma classe (isto é, fazem com que a classe tenha Responsabilidade Única).
  4. Simplificam o uso de um sistema.
5. Qual a semelhança entre Proxy, Decorador e Visitor? E qual a diferença entre esses padrões?
6. No exemplo de Adaptador, mostramos o código de uma única classe adaptadora (`AdaptadorProjetoSamsung`). Escreva o código de uma classe semelhante, mas que adapte a interface `Projeto` para a interface `ProjetoLG` (o código de ambas interfaces é mostrado na Seção 6.5). Chame essa classe de `AdaptadorProjetoLG`.

7. Suponha uma classe base A. Suponha que queremos adicionar quatro funcionalidades opcionais F1, F2, F3 e F4 em A. Essas funcionalidades podem ser adicionadas em qualquer ordem, isto é, a ordem não é importante. Se usarmos herança, quantas subclasses de A teremos que implementar? Se optarmos por uma solução por meio de decoradores, quantas classes teremos que implementar (sem contar a classe A). Justifique e explique sua resposta.

8. No exemplo de Decorador, mostramos o código de um único decorador (`ZipChannel`). Escreva o código de uma classe semelhante, mas que imprima a mensagem a ser transmitida ou recebida no console. Chame essa classe decoradora de `LogChannel`.

9. Dado o código abaixo de uma classe `Subject` (do padrão Observador):

```
interface Observer {  
    public void update(Subject s);  
}  
  
class Subject {  
  
    private List<Observer> observers=new ArrayList<Observer>();  
  
    public void addObserver(Observer observer) {  
        observers.add(observer);  
    }  
  
    public void notifyObservers() {  
        (A)  
    }  
}
```

Implemente o código de `notifyObservers`, comentado com um (A) acima.

10. Suponha a API de Java para E/S. Suponha que para evitar o que chamamos de paternite, você fez a união das classes `FileInputStream` e `BufferedInputStream` em uma única classe. Como discutimos na Seção 6.13, o mecanismo de buffer será ativado por *default* na classe que você criou. Porém, como você tornaria possível desativar buffers nessa nova classe, caso isso fosse necessário?

11. Suponha o exemplo de Visitor que usamos na Seção 6.11. Especificamente, suponha o seguinte código, mostrado no final da seção.

```
PrintVisitor visitor = new PrintVisitor();  
  
foreach(Veiculo veiculo: listaDeVeiculosEstacionados) {  
    veiculo.accept(visitor);  
}
```

Suponha que listaDeVeiculosEstacionados armazene três objetos: umCarro, umOnibus e umOutroCarro. Desenhe um diagrama de sequência UML que mostre os métodos executados por esse trecho de código (suponha que ele é executado por um objeto main).

12. Em uma entrevista dada ao site InformIT, em 2009, por ocasião dos 15 anos do lançamento da primeira edição do GoF, três dos autores do livro mencionaram que — se tivessem que lançar uma segunda edição do trabalho — provavelmente manteriam os padrões originais e incluiriam alguns novos, que se tornaram comuns desde o lançamento da primeira edição, em 1994. Um dos novos padrões que eles mencionaram na entrevista é chamado de **Null Object**. Estude e explique o funcionamento e os benefícios desse padrão de projeto. Para isso, você vai encontrar diversos artigos na Web. Mas se preferir consultar um livro, uma boa referência é o Capítulo 25 do livro *Agile Principles, Patterns, and Practices in C#*, de Robert C. Martin e Micah Martin. Ou então o refactoring chamado Introduce Null Object do livro de Refactoring de Martin Fowler.

# Cap 7 Arquitetura

Este capítulo inicia com uma introdução ao conceito de arquitetura de software. Em seguida, discutimos diversos padrões arquiteturais, incluindo: Arquitetura em Camadas e, especificamente, Arquitetura em Três Camadas (Seção 7.2), Arquitetura MVC (Seção 7.3) e Arquitetura baseada em Microsserviços (Seção 7.4). No caso de microsserviços, procuramos mostrar o contexto que levou ao surgimento desse padrão arquitetural, bem como discutimos seus principais benefícios e desafios. Em seguida, discutimos dois padrões arquiteturais usados para garantir escalabilidade e desacoplamento em sistemas distribuídos: Filas de Mensagens (Seção 7.5) e Publish/Subscribe (Seção 7.6). Terminamos o capítulo discutindo outros padrões arquiteturais (Seção 7.7) e dando um exemplo de anti-padrão arquitetural (Seção 7.8).

## Introdução

Existe mais de uma definição para arquitetura de software. Uma das mais comuns considera que arquitetura preocupa-se com projeto em mais alto nível. Ou seja, o foco deixa de ser a organização e interfaces de classes individuais e passa a ser em unidades de maior tamanho, sejam elas pacotes, componentes, módulos, subsistemas, camadas ou serviços — o nome não importa tanto neste primeiro momento. De forma genérica, os termos que acabamos de mencionar devem ser entendidos como conjuntos de classes relacionadas.

Além de possuírem um maior tamanho, os componentes arquiteturais devem ser relevantes para que um sistema atenda a seus objetivos. Por exemplo, suponha que você trabalhe em um sistema de informações. Certamente, esse sistema inclui um módulo de persistência, que faz a interface com o banco de dados. Esse módulo é fundamental em sistemas de informações, pois eles têm como objetivo principal automatizar e persistir informações relativas a processos de negócio. Por outro lado, suponha agora que você trabalhe em um sistema que usa técnicas de inteligência artificial para diagnosticar doenças. O sistema também possui um módulo de persistência para armazenar dados das doenças que são diagnosticadas. Porém, esse módulo,

além de simples, não é relevante para o objetivo principal do sistema. Logo, ele não faz parte da sua arquitetura.

Existe ainda uma segunda definição para arquitetura de software. Tal como expresso na frase de Ralph Johnson que abre esse capítulo, ela considera que arquitetura de software inclui as decisões de projeto mais importantes em um sistema. Essas decisões são tão importantes que, uma vez tomadas, dificilmente poderão ser revertidas no futuro. Portanto, essa segunda forma de definir arquitetura é mais ampla do que a primeira que apresentamos. Ela considera que arquitetura não é apenas um conjunto de módulos, mas um conjunto de decisões. É verdade que dentre essas decisões, inclui-se a definição dos módulos principais de um sistema. Mas outras decisões também são contempladas, como a escolha da linguagem de programação e do banco de dados que serão usados no desenvolvimento. De fato, uma vez que um sistema é implementado com um determinado banco de dados, dificilmente consegue-se migrar para um outro banco de dados. Prova disso é que ainda hoje temos exemplos de sistemas críticos que funcionam com bancos de dados não-relacionais e que são implementados em linguagens como COBOL.

**Padrões arquiteturais** propõem uma organização de mais alto nível para sistemas de software, incluindo seus principais módulos e as relações entre eles. Essas relações definem, por exemplo, que um módulo A pode (ou não pode) usar os serviços de um módulo B. Neste capítulo, vamos estudar padrões arquiteturais que dão origem às seguintes arquiteturas: Arquitetura em Camadas (Seção 7.2), Arquitetura Model-View-Controller ou MVC (Seção 7.3), Microsserviços (Seção 7.4), Arquitetura Orientada a Mensagens (Seção 7.5) e Arquitetura Publish/Subscribe (Seção 7.6).

Para finalizar, iremos apresentar de forma breve outros padrões arquiteturais, como pipes e filtros (Seção 7.7). Vamos também dar um exemplo de um anti-padrão arquitetural, conhecido como *big ball of mud* (Seção 7.8).

**Aprofundamento:** Alguns autores — como Taylor et al. ([link](#)) — fazem uma distinção entre padrões e **estilos arquiteturais**. Segundo eles, padrões focam em soluções para problemas específicos de arquitetura; enquanto estilos arquiteturais propõem que os módulos de um sistema devem ser organizados de uma determinado modo, o que não necessariamente ocorre

visando resolver um problema específico. Assim, para esses autores, MVC é um padrão arquitetural que resolve o problema de separar apresentação e modelo em sistemas de interfaces gráficas. Por outro lado, Pipes & Filtros constituem um estilo arquitetural. Neste capítulo, porém, não vamos fazer essa distinção. Em vez disso, chamaremos todos eles de padrões arquiteturais.

## Debate Tanenbaum-Torvalds

No início de 1992, um debate acalorado sobre a arquitetura de sistemas operacionais tomou conta de um grupo de discussão da Internet. Apesar de vários desenvolvedores e pesquisadores terem participado da discussão, ela ficou conhecida como **Debate Tanenbaum-Torvalds** ([link](#), apêndice A, página 102). Tanenbaum (Andrew) é um pesquisador da área de sistemas operacionais, autor de livros-texto na área e professor da Vrije Universiteit, em Amsterdã, na Holanda. E Torvalds (Linus) na época era estudante de Computação na Universidade de Helsinki, na Finlândia.

A discussão começou quando Tanenbaum postou uma mensagem no grupo com o título Linux está obsoleto. O seu principal argumento era que o Linux seguia uma **arquitetura monolítica**, na qual todas as funções do sistema operacional — como gerenciamento de processos, gerenciamento de memória e sistemas de arquivos, por exemplo — são implementadas em um único arquivo executável, que roda em modo supervisor. Desde essa época, Tanenbaum argumentava que a melhor solução para sistemas operacionais era uma **arquitetura microkernel**, na qual o kernel fica responsável apenas pelas funções mais básicas do sistema. As demais funções rodam como processos independentes e fora do kernel. Linus respondeu à mensagem de forma enfática, alegando que pelo menos o Linux já era uma realidade na época, enquanto que o sistema baseado em um microkernel que estava sendo desenvolvido por Tanenbaum apresentava diversos problemas e bugs. A discussão continuou forte e Tanenbaum chegou a declarar que Torvalds tinha sorte por não ter sido seu aluno; se fosse, ele certamente não teria obtido uma boa nota com o projeto monolítico do Linux. Um comentário interessante foi feito em seguida por Ken Thompson, um dos projetistas das primeiras versões do Unix:

Na minha opinião, é mais fácil implementar um sistema operacional com um kernel monolítico. Mas é também mais fácil que ele se transforme em uma bagunça à medida que o kernel é modificado.

Na verdade, Thompson previu o futuro, pois em 2009, Linus declarou o seguinte em uma conferência:

Não somos mais o kernel simples, pequeno e hiper-eficiente que imaginei há 15 anos. Em vez disso, nosso kernel está ficando grande e inchado. E sempre que adicionamos novas funcionalidades, o cenário piora.

Esse comentário consta de uma página da Wikipedia ([link](#)) e foi objeto de diversas matérias em sites de tecnologia na época. Ele revela que arquitetura não são apenas decisões importantes e difíceis de reverter. Muitas vezes, são também decisões que levam anos para que seus efeitos negativos fiquem mais claros e começem a causar problemas.

## Arquitetura em Camadas

**Arquitetura em camadas** é um dos padrões arquiteturais mais usados, desde que os primeiros sistemas de software de maior porte foram construídos nas décadas de 60 e 70. Em sistemas que seguem esse padrão, as classes são organizadas em módulos de maior tamanho, chamados de **camadas**. As camadas são dispostas de forma hierárquica, como em um bolo. Assim, uma camada somente pode usar serviços — isto é, chamar métodos, instanciar objetos, estender classes, declarar parâmetros, lançar exceções, etc. — da camada imediatamente inferior.

Dentre outras aplicações, arquiteturas em camadas são muito usadas na implementação de protocolos de rede. Por exemplo, HTTP é um protocolo de aplicação, que usa serviços de um protocolo de transporte; por exemplo, TCP. Por sua vez, TCP usa serviços de um protocolo de rede; por exemplo, IP. Finalmente, a camada IP usa serviços de um protocolo de comunicação; por exemplo, Ethernet.

Uma arquitetura em camadas partitiona a complexidade envolvida no desenvolvimento de um sistema em componentes menores (as camadas).

Como uma segunda vantagem, ela disciplina as dependências entre essas camadas. Como dissemos, a camada  $n$  somente pode usar serviços da camada  $n-1$ . Isso ajuda no entendimento, manutenção e evolução de um sistema. Por exemplo, torna-se mais fácil trocar uma camada por outra (por exemplo, mudar de TCP para UDP). Fica mais fácil também o reúso de uma camada por mais de uma camada superior. Por exemplo, a camada de transporte pode ser usada por vários protocolos de aplicação, como HTTP, SMTP, DHCP, etc.

**Aprofundamento:** Uma das primeiras propostas de arquitetura em camadas foi elaborada por Edsger W. Dijkstra, em 1968, para um sistema operacional denominado THE ([link](#)). As camadas propostas por Dijkstra foram as seguintes: multiprogramação (camada 0), alocação de memória (camada 1), comunicação entre processos (camada 2), gerenciamento de entrada/saída (camada 3) e programas dos usuários (camada 4). Dijkstra conclui o artigo destacando que os benefícios de uma estrutura hierárquica são mais importantes ainda em projetos de maior porte.

## Arquitetura em Três Camadas

Esse tipo de arquitetura é comum na construção de sistemas de informação corporativos. Até o final da década de 80, aplicações corporativas — como folhas de pagamento, controle de estoque ou sistemas financeiros — executavam em **mainframes**, que eram computadores fisicamente grandes e também muito caros. As aplicações eram monolíticas e acessadas por meio de terminais burros, isto é, sem qualquer capacidade de processamento e com uma interface totalmente textual. Com o avanço nas tecnologias de rede e de hardware, foi possível migrar esses sistemas de mainframes para outras plataformas. Foi nessa época que arquiteturas em três camadas se tornaram uma alternativa muito comum.

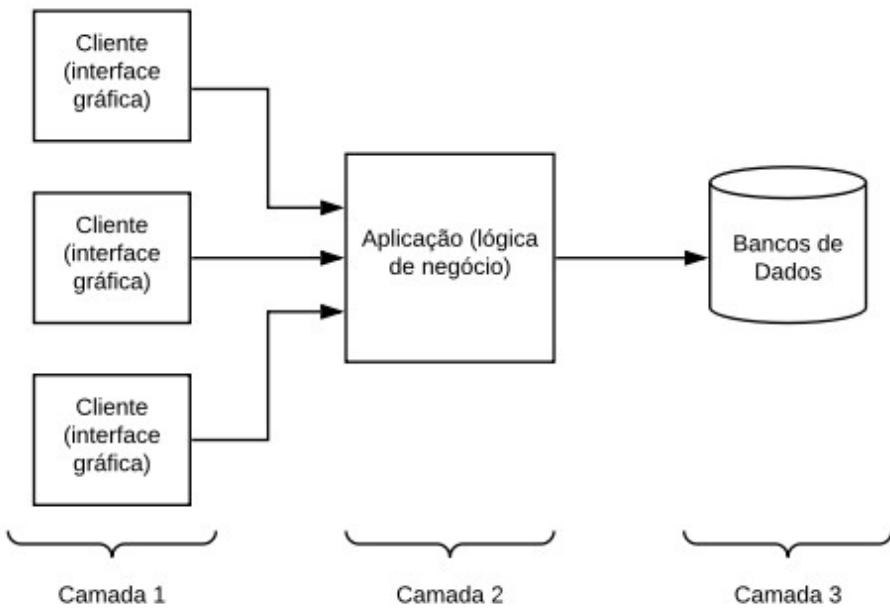
As três camadas dessa arquitetura são as seguintes:

- Interface com o Usuário, também chamada de camada de apresentação, é responsável por toda interação com o usuário. Ela trata tanto da exibição de informação, como da coleta e processamento de entradas e eventos de interfaces, tais como cliques em botões, marcação de texto,

etc. A camada de interface pode ser uma aplicação desktop, em Windows ou outro sistema operacional com interface gráfica, como também Web. Por exemplo, um sistema acadêmico deve prover uma interface para os professores lançarem as notas de suas disciplinas. O elemento principal dessa interface pode ser um formulário com duas colunas: nome do aluno e nota. O código que implementa esse formulário deve estar na camada de interface.

- Lógica de Negócio, também conhecida como camada de aplicação, implementa as regras de negócio do sistema. No sistema acadêmico que estamos usando como exemplo, podemos ter a seguinte regra de negócio: as notas são maiores ou iguais a zero e menores ou iguais ao valor da avaliação. Quando um professor solicitar o lançamento das notas de uma disciplina, cabe à camada de lógica verificar se essa regra é obedecida. Uma outra regra de negócio pode ser a seguinte: após o lançamento de qualquer nota, os alunos devem ser avisados por meio de um e-mail.
- Banco de Dados, que armazena os dados manipulados pelo sistema. Por exemplo, no nosso sistema acadêmico, após lançamento e validação das notas, elas são salvas em um banco de dados.

Normalmente, uma arquitetura em três camadas é uma arquitetura distribuída. Isto é, a camada de interface executa na máquina dos clientes. A camada de negócio executa em um servidor, muitas vezes chamado de servidor de aplicação. E, por fim, temos o banco de dados. A figura da próxima página mostra um exemplo, que assume que a interface oferecida aos clientes é uma interface gráfica.



## Arquitetura em três camadas

Em sistemas três camadas, a camada de aplicação pode ter diversos módulos, incluindo uma fachada, para facilitar o acesso ao sistema pelos clientes, e um módulo de persistência, com a função de isolar o banco de dados dos demais módulos.

Por fim, gostaríamos de mencionar que é possível ter sistemas em **duas camadas**. Nesses casos, as camadas de interface e de aplicação são unidas em uma única camada, que executa no cliente. A camada restante continua sendo o banco de dados. A desvantagem de arquiteturas em duas camadas é que todo o processamento ocorre nos clientes, que, portanto, devem ter um maior poder de computação.

## Arquitetura MVC

O padrão arquitetural MVC (Model-View-Controller) foi proposto no final da década de 70 e, em seguida, usado na implementação de Smalltalk-80, que é considerada uma das primeiras linguagens orientadas a objetos. Além de utilizarem conceitos de orientação a objetos, programas em Smalltalk foram pioneiros no uso de interfaces gráficas, com janelas, botões, scroll bars, mouse, etc. Isso em uma época em que os sistemas operacionais

ofereciam apenas interfaces de linha de comando e os programas tinham uma interface textual, isto é, as telas eram uma matriz de caracteres, com, por exemplo, 25 linhas e 80 colunas.

MVC foi o padrão arquitetural escolhido pelos projetistas de Smalltalk para implementação de interfaces gráficas. Especificamente, MVC define que as classes de um sistema devem ser organizadas em três grupos:

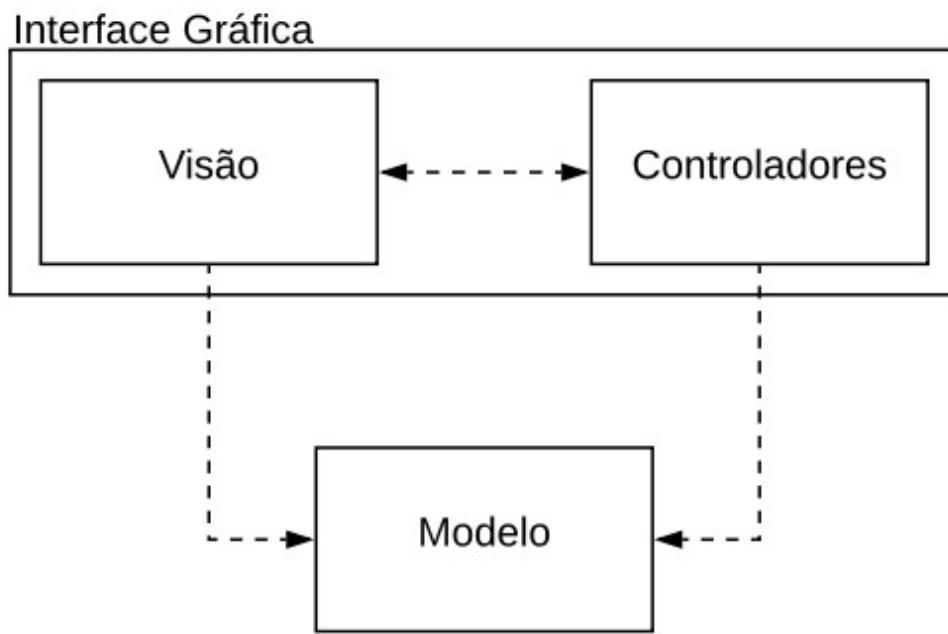
- Visão: classes responsáveis pela apresentação da interface gráfica do sistema, incluindo janelas, botões, menus, barras de rolagem, etc.
- Controladoras: classes que tratam e interpretam eventos gerados por dispositivos de entrada, como mouse e teclado. Como resultado de tais eventos, Controladoras podem solicitar uma alteração no estado do Modelo ou da Visão. Suponha, por exemplo, uma Calculadora. Quando o usuário clica em um botão +, uma classe Controladora deve capturar esse evento e chamar um método do Modelo. Como um segundo exemplo, quando o usuário clicar no botão Dark UI, cabe também a uma classe Controladora solicitar à Visão para mudar as cores da interface gráfica para tons mais escuros.
- Modelo: classes que armazenam os dados manipulados pela aplicação e que têm a ver com o domínio do sistema em construção. Assim, classes de modelo não têm qualquer conhecimento ou dependência para classes de Visão e Controladoras. Além de dados, classes de modelo podem conter métodos que alteram o estado dos objetos de domínio.

Portanto, em uma arquitetura MVC, a interface gráfica é formada por objetos de visão e por controladores. Porém, em muitos sistemas não existe uma distinção clara entre Visão e Controladores. Segundo Fowler ([link](#), página 331), mesmo a maioria das versões de Smalltalk não separa esses dois componentes. Por isso, fica mais fácil entender da seguinte forma:

$$\text{MVC} = (\text{Visão} + \text{Controladores}) + \text{Modelo} = \text{Interface Gráfica} + \text{Modelo}$$

A próxima figura mostra as dependências entre as classes de uma arquitetura MVC. A figura primeiro reforça que a interface gráfica é composta pela Visão e por Controladores. Podemos observar também que a Interface

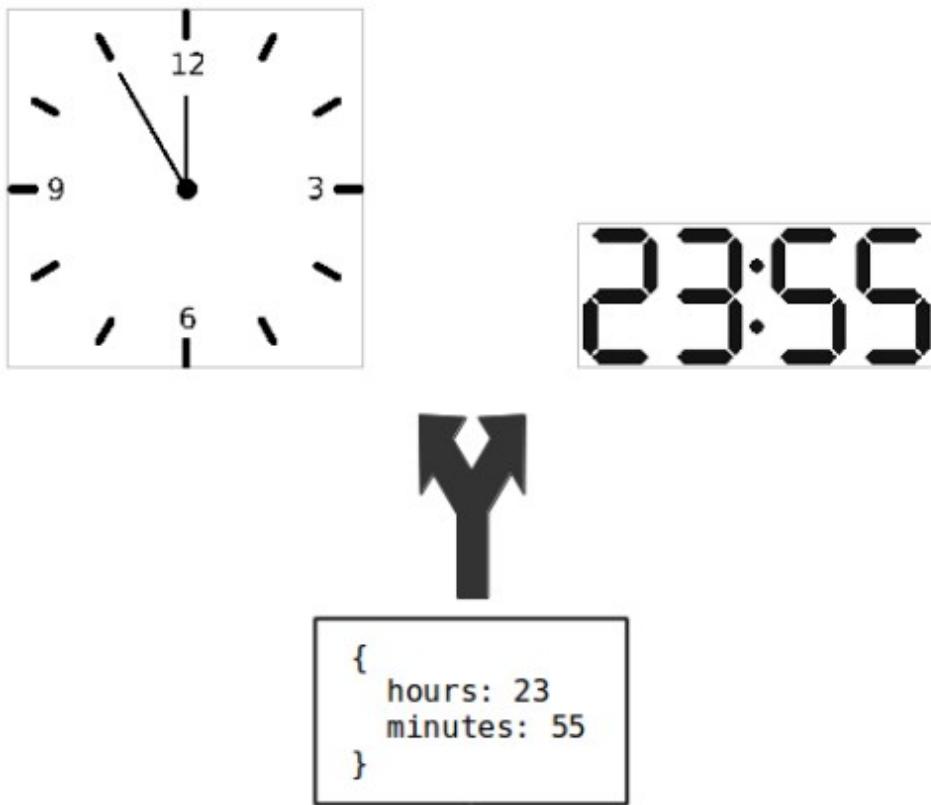
Gráfica pode depender do Modelo. Porém, classes de Modelo não têm dependências para classes da Interface Gráfica. Na verdade, podemos entender a Interface Gráfica como sendo observadora do Modelo. Quando o estado dos objetos do Modelo é alterado, deve-se atualizar automaticamente a interface do sistema.



### Arquitetura MVC

Dentre as vantagens de arquiteturas MVC, podemos citar:

- MVC favorece a especialização do trabalho de desenvolvimento. Por exemplo, pode-se ter desenvolvedores especialistas na implementação de interfaces gráficas, os quais são chamados de desenvolvedores de front-end. Por outro lado, desenvolvedores de classes de Modelo não precisam conhecer e implementar código de interface com usuários.
- MVC permite que classes de Modelo sejam usadas por diferentes Visões, como ilustrado na próxima figura. Neste exemplo, um objeto de Modelo armazena dois valores: hora e minutos. Esses dados são apresentados em duas visões diferentes. Na primeira, como um relógio analógico. Na segunda, como um relógio digital.



Sistema MVC com mais de uma visão (interface gráfica)

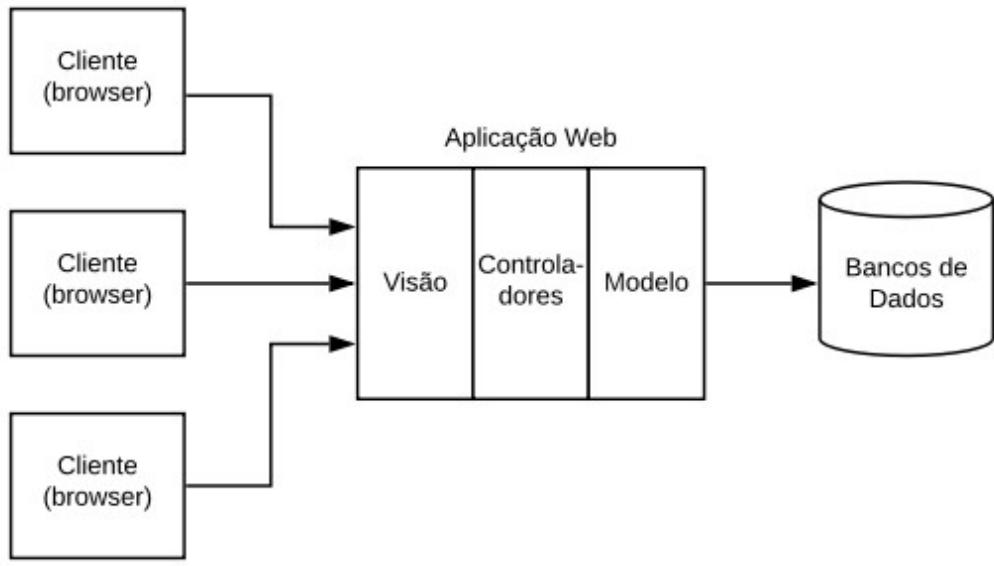
- MVC favorece testabilidade. Como veremos no próximo capítulo, é mais fácil testar objetos não-visuais, isto é, não relacionados com a implementação de interfaces gráficas. Por isso, ao separar objetos de apresentação de objetos de Modelo, fica mais fácil testar esses últimos.

Vamos concluir com um resumo sobre MVC, na visão de Fowler e Beck ([link](#), Cap. 12, pág. 370):

O coração e a parte mais preciosa de MVC está na separação entre código de interface com o usuário (a Visão, também chamada de apresentação) e a lógica do domínio (o Modelo). As classes de apresentação implementam apenas a lógica necessária para lidar com a interface do usuário. Por outro lado, objetos de domínio não incluem código visual, mas apenas lógica de negócios. Isso separa duas partes complexas de sistemas de software em partes que são mais fáceis de se modificar. Também permite várias apresentações da mesma lógica de negócio.

**Pergunta Frequentes: Qual a diferença entre MVC e três camadas?** A resposta vai ser um pouco longa e vamos nos basear na evolução histórica dessas arquiteturas:

- Como comentamos, MVC surgiu no final da década de 70, para ajudar na construção de interfaces gráficas. Isto é, aplicações que incluem uma interface com janelas, botões, caixas de texto, etc. Como exemplo, podemos citar um pacote de escritório, com aplicações como Word, Excel e Powerpoint, no caso do sistema operacional Windows.
- Na década de 90, as tecnologias de redes, sistemas distribuídos e bancos de dados se tornaram comuns. Viabilizou-se então a construção de aplicações distribuídas com três camadas. Nesse caso, MVC pode ser usado na implementação da camada de interface, que pode, por exemplo, ser uma aplicação nativa em Windows, implementada usando-se linguagens como Visual Basic ou Java (neste último caso, usando-se frameworks como Swing). Resumindo, a aplicação, como um todo segue, uma arquitetura em três camadas, mas usa MVC na camada de interface com o usuário.
- No início dos anos 2000, a Web se popularizou e a interface das aplicações migrou para HTML e, depois, para HTML e JavaScript. A confusão entre os termos MVC e três camadas surgiu então nessa época, principalmente devido ao aparecimento de frameworks para implementação de sistemas Web que se denominaram frameworks MVC. Como exemplo, podemos citar Spring (para Java), Ruby on Rails, Django (para Python) e CakePHP. Na verdade, esses frameworks expandiram e adaptaram o conceito de MVC para Web. Por exemplo, eles forçam a organização de um sistema Web em três partes (veja também na próxima figura): visão, composta por páginas HTML; controladores, que processam uma solicitação e geram uma nova visão como resposta e modelo, que é a camada que persiste os dados em um banco de dados.



## Arquitetura MVC Web

Logo, apesar de sistemas Web serem parecidos com sistemas três camadas, os frameworks Web mais populares optaram por usar termos típicos de MVC para nomear seus componentes. Portanto, a melhor maneira de responder à pergunta é afirmar que existem duas vertentes de sistemas MVC: a vertente clássica, que surgiu com Smalltalk-80 e a vertente Web, que se tornou comum na década de 90 e início dos anos 2000. Essa última vertente lembra bastante sistemas três camadas.

## Exemplo: Single Page Applications

Em uma aplicação Web tradicional, com formulários, menus e botões, toda vez que o usuário gera um evento — por exemplo, clica em um botão Gravar — ocorre uma interação entre o navegador e o servidor Web. Isto é, o navegador envia informações para o servidor Web, que as processa e devolve uma nova página para ser exibida para o usuário. Essas aplicações são então menos interativas e responsivas, devido ao atraso da comunicação entre navegador e servidor Web.

Recentemente, surgiu um novo tipo de sistema Web, chamado de **Single Page Applications (SPAs)**. Essas aplicações são mais parecidas com aplicações desktop do que com aplicações Web tradicionais. Ao se entrar em uma SPA, ela carrega para o navegador todo o código, incluindo páginas

HTML e scripts em CSS e JavaScript. Com isso, apesar de usar um navegador, o usuário tem a impressão de que ele está usando uma aplicação local, pois não ocorre mais uma atualização da página do navegador toda vez que ele gera certos eventos. Diversas aplicações modernas são SPAs, sendo o GMail, provavelmente, o exemplo mais conhecido. Evidentemente, continua existindo uma parte da aplicação no servidor, com a qual a SPA comunica-se frequentemente. Por exemplo, quando chega um novo e-mail o GMail atualiza a lista de mensagens na caixa de entrada. Para que isso ocorra de forma automática, a comunicação entre a SPA e o servidor deve ser assíncrona.

Existem diversos frameworks — todos em JavaScript — para implementação de SPAs. A seguir, mostramos um exemplo usando Vue.js.

```
<html>
<script src="https://cdn.jsdelivr.net/npm/vue"></script>

<body>

<h3>Uma Simples SPA</h3>

<div id="ui">
  Temperatura: {{ temperatura }}
  <p><button v-on:click="incTemperatura">Incrementa
  </button></p>
</div>

<script>
var model = new Vue({
  el: '#ui',
  data: {
    temperatura: 60
  },
  methods: {
    incTemperatura: function() {
      this.temperatura++;
    }
  }
})
</script>

</body>
</html>
```

Essa aplicação apresenta uma temperatura na tela do navegador e um botão para incrementá-la (veja figura a seguir).

## Uma Simples SPA

Temperatura: 60

Incrementa

Interface da Single-Page Application do exemplo

**Código Fonte:** O código do exemplo está disponível neste [link](#). Se quiser executar a aplicação no seu navegador, basta usar esse [link](#).

O interessante é que SPAs seguem uma arquitetura parecida com MVC. No exemplo mostrado, a interface da SPA, contendo visão e controle, é implementada em HTML, mais precisamente no código delimitado pela tag `<div>`. O modelo é implementado em JavaScript, usando-se Vue.js. O código do modelo está delimitado pela tag `<script>`

# Cap 8 Testes

Este capítulo inicia com uma introdução a testes, na qual usamos uma pirâmide para classificar os principais tipos de testes, de acordo com a sua granularidade e frequência. Também esclarecemos que o nosso foco no capítulo são testes de unidade. Assim, começamos com uma seção sobre conceitos e funcionamento básico desse tipo de teste (Seção 8.2). Em seguida, tratamos de aspectos avançados e complementares, ainda sobre testes de unidade, incluindo princípios para escrita de tais testes (Seção 8.3), cobertura de testes (Seção 8.4), projeto de software para facilitar a implementação de testes de unidade (Seção 8.5) e objetos mocks, os quais são usados para facilitar a implementação de testes de unidade (Seção 8.6). Na Seção 8.7, apresentamos o conceito de Desenvolvimento Dirigido por Testes (ou *Test-Driven Development*, TDD). Em seguida, tratamos dos testes da parte de cima da pirâmide de testes, ou seja, Testes de Integração (Seção 8.8) e Testes de Sistemas (Seção 8.9). Para fechar o capítulo, a Seção 8.10 cobre de forma rápida outros tipos de testes, tais como: testes caixa-preta e caixa-branca, testes de aceitação e testes de requisitos não-funcionais.

## Introdução

Software é uma das construções humanas mais complexas, como discutimos na Introdução deste livro. Portanto, é compreensível que sistemas de software estejam sujeitos aos mais variados tipos de erros e inconsistências. Para evitar que tais erros cheguem aos usuários finais e causem prejuízos de valor incalculável, é fundamental introduzir atividades de teste em projetos de desenvolvimento de software. De fato, teste é uma das práticas de programação mais valorizadas hoje em dia, em qualquer tipo de software. É também uma das práticas que sofreram mais transformações nos anos recentes.

Quando o desenvolvimento era em cascata, os testes ocorriam em uma fase separada, após as fases de levantamento de requisitos, análise, projeto e codificação. Além disso, existia uma equipe separada de testes, responsável por verificar se a implementação atendia aos requisitos do sistema. Para garantir isso, frequentemente os testes eram manuais, isto é, uma pessoa

usava o sistema, informava dados de entrada e verificava se as saídas eram aquelas esperadas. Assim, o objetivo de tais testes era apenas detectar bugs, antes que o sistema entrasse em produção.

Com métodos ágeis, a prática de testes de software foi profundamente reformulada, conforme explicamos a seguir:

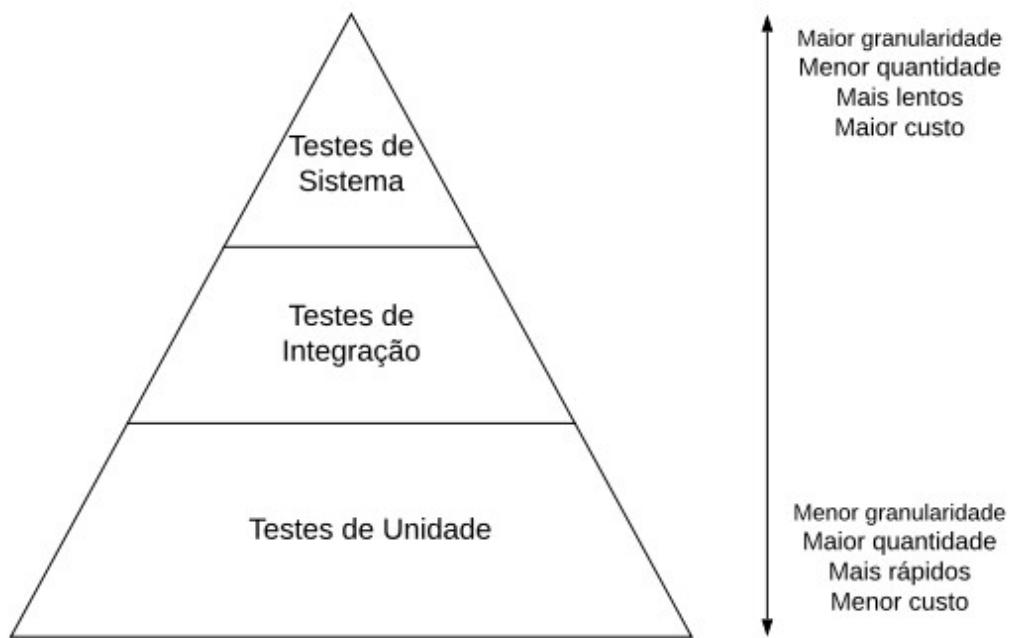
- Grande parte dos testes passou a ser automatizada, isto é, além de implementar as classes de um sistema, os desenvolvedores passaram a implementar também código para testar tais classes. Assim, os programas tornaram-se **auto-testáveis**.
- Testes não são mais implementados após todas as classes de um sistema ficarem prontas. Muitas vezes, eles são implementados até mesmo antes dessas classes.
- Não existem mais grandes equipes de testes — ou elas são responsáveis por testes específicos. Em vez disso, o desenvolvedor que implementa uma classe também deve implementar os seus testes.
- Testes não são mais um instrumento exclusivo para detecção de bugs. Claro, isso continua sendo importante, mas testes ganharam novas funções, como garantir que uma classe continuará funcionando após um bug ser corrigido em uma outra parte do sistema. E testes são também usados como documentação para o código de produção.

Essas transformações tornaram testes uma das práticas de programação mais valorizadas em desenvolvimento moderno de software. É nesse contexto que devemos entender a frase de Michael Feathers que abre esse capítulo: se um código não é acompanhado de testes, ele pode ser considerado de baixa qualidade ou até mesmo um código legado.\_

Neste capítulo, vamos focar em **testes automatizados**, pois testes manuais dão muito trabalho, são demorados e caros. Pior ainda, eles devem ser repetidos toda vez que o sistema sofrer uma modificação.

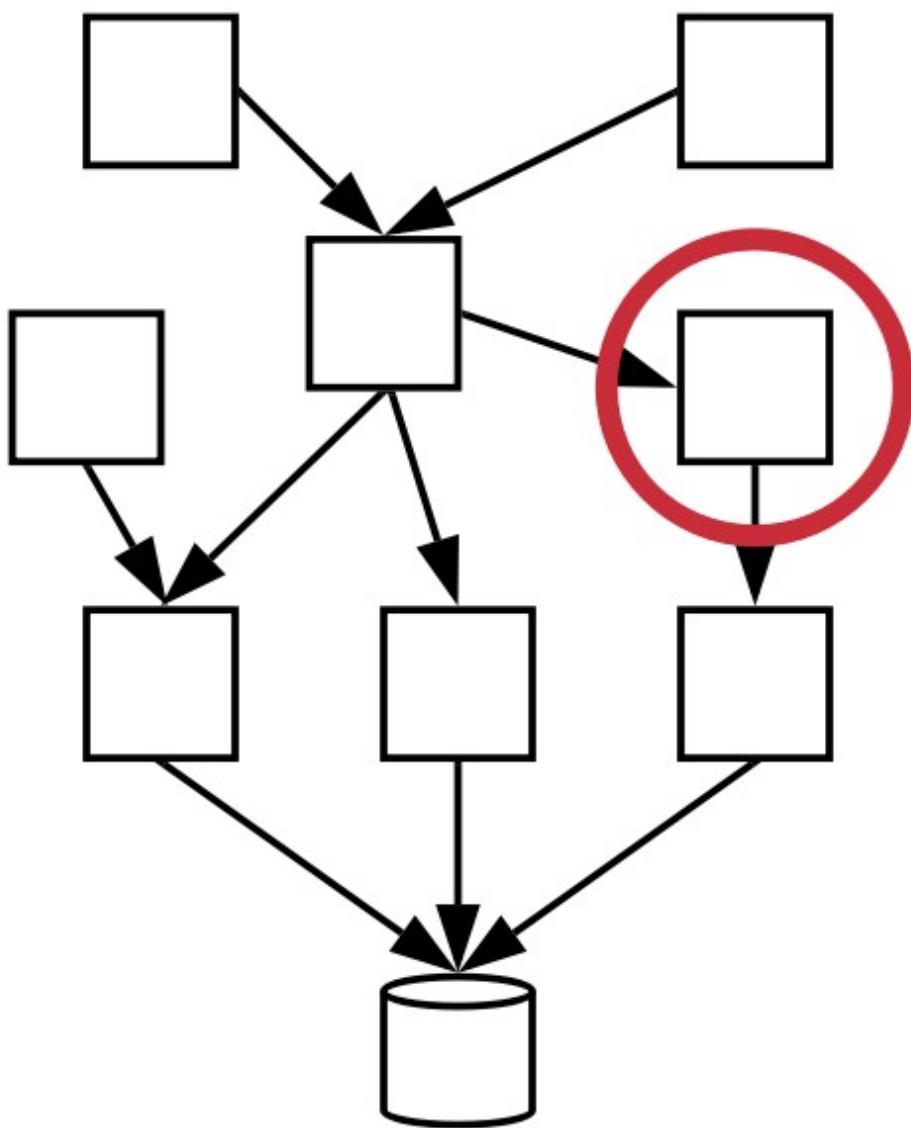
Uma forma interessante de classificar testes automatizados é por meio de uma **pirâmide de testes**, originalmente proposta por Mike Cohn ([link](#)).

Como mostra a próxima figura, essa pirâmide partitiona os testes de acordo com sua granularidade.

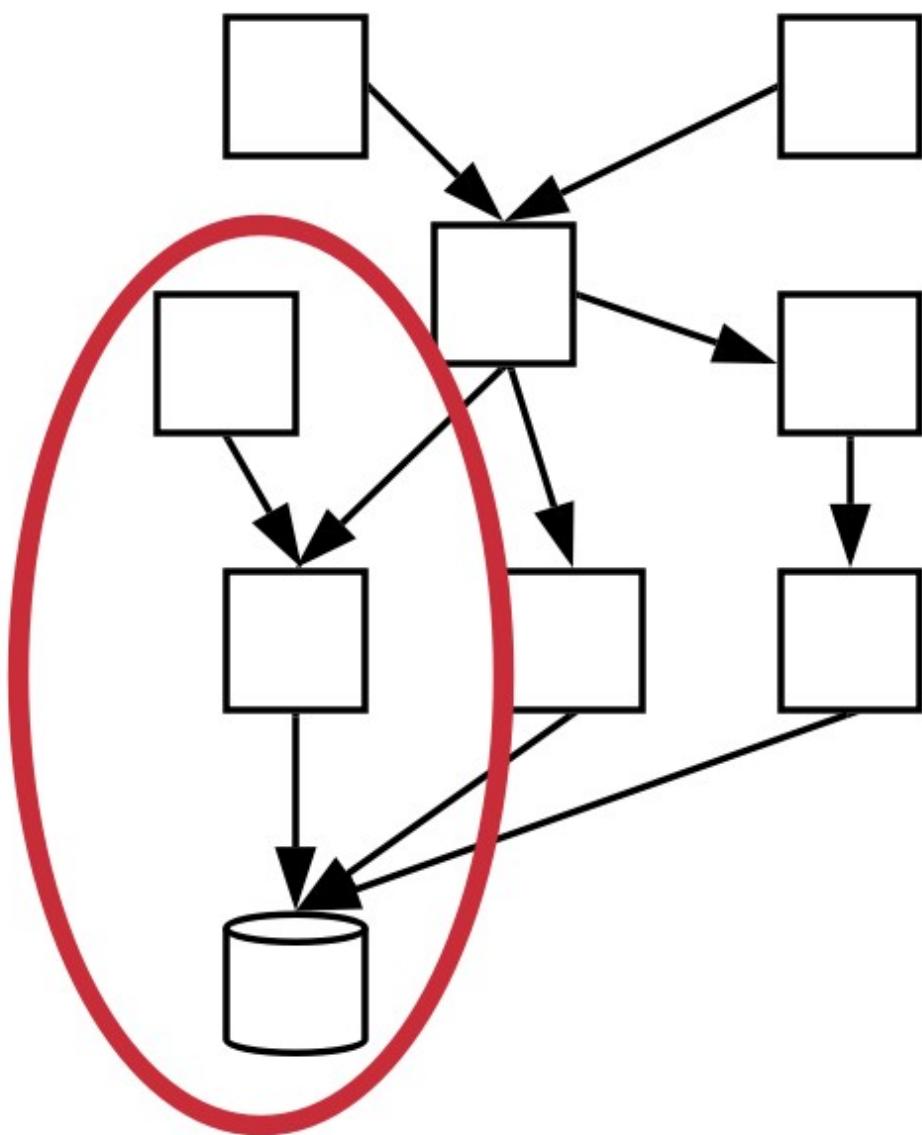


Pirâmide de testes

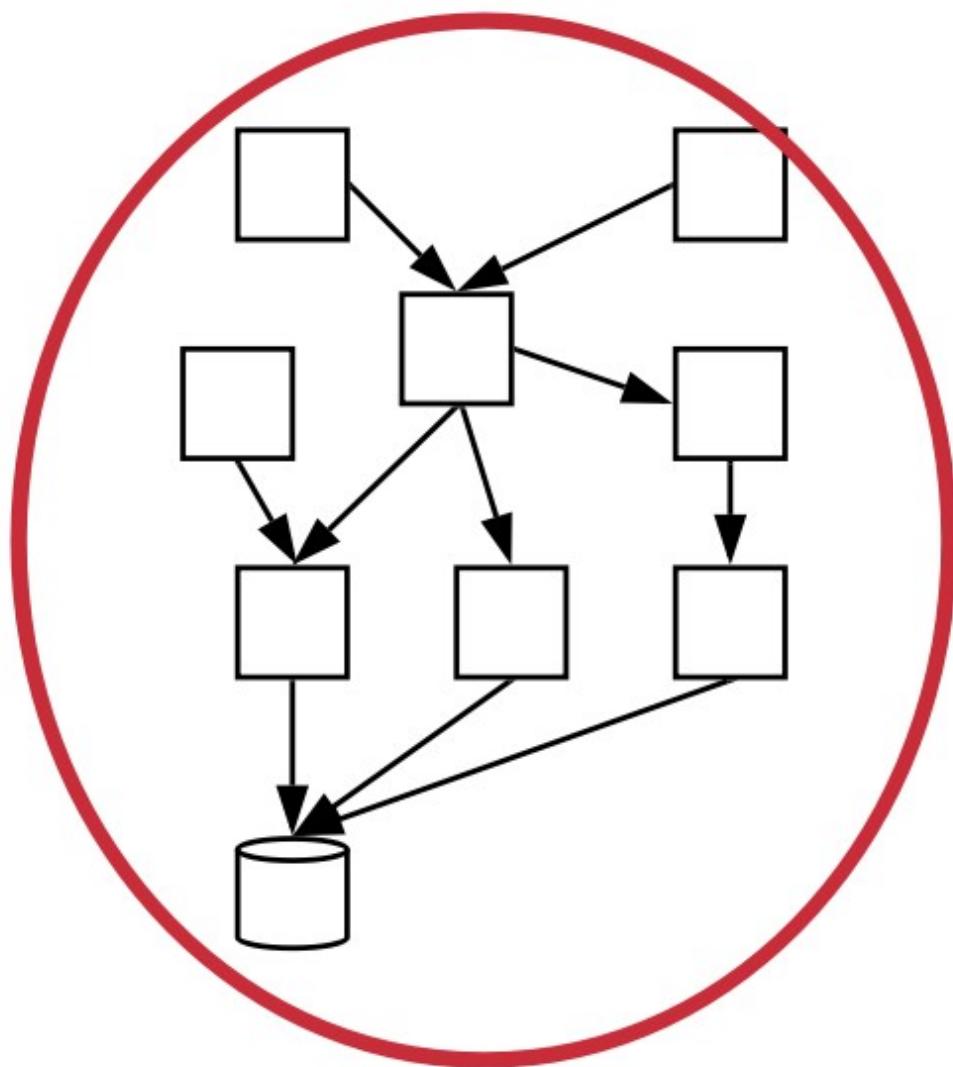
Particularmente, os testes são divididos em três grupos. **Testes de unidade** verificam automaticamente pequenas partes de um código, normalmente uma classe apenas (acompanhe também pelas figuras da próxima página). Eles formam a base da pirâmide, ou seja, a maior parte dos testes estão nessa categoria. Testes de unidade são simples, mais fáceis de implementar e executam rapidamente. No próximo nível, temos **testes de integração** ou **testes de serviços**, que verificam uma funcionalidade ou transação completa de um sistema. Logo, são testes que usam diversas classes, de pacotes distintos, e podem ainda testar componentes externos, como bancos de dados. Testes de integração demandam mais esforço para serem implementados e executam de forma mais lenta. Por fim, no topo da pirâmide, temos os **testes de sistema**, também chamados de **testes de interface com o usuário**. Eles simulam, da forma mais fiel possível, uma sessão de uso do sistema por um usuário real. Como são testes de ponta a ponta (*end-to-end*), eles são mais caros, mais lentos e menos numerosos. Testes de interface costumam ser também frágeis, isto é, mínimas alterações nos componentes da interface podem demandar modificações nesses testes.



Escopo de testes de unidade



Escopo de testes de integração



## Escopo de testes de sistema

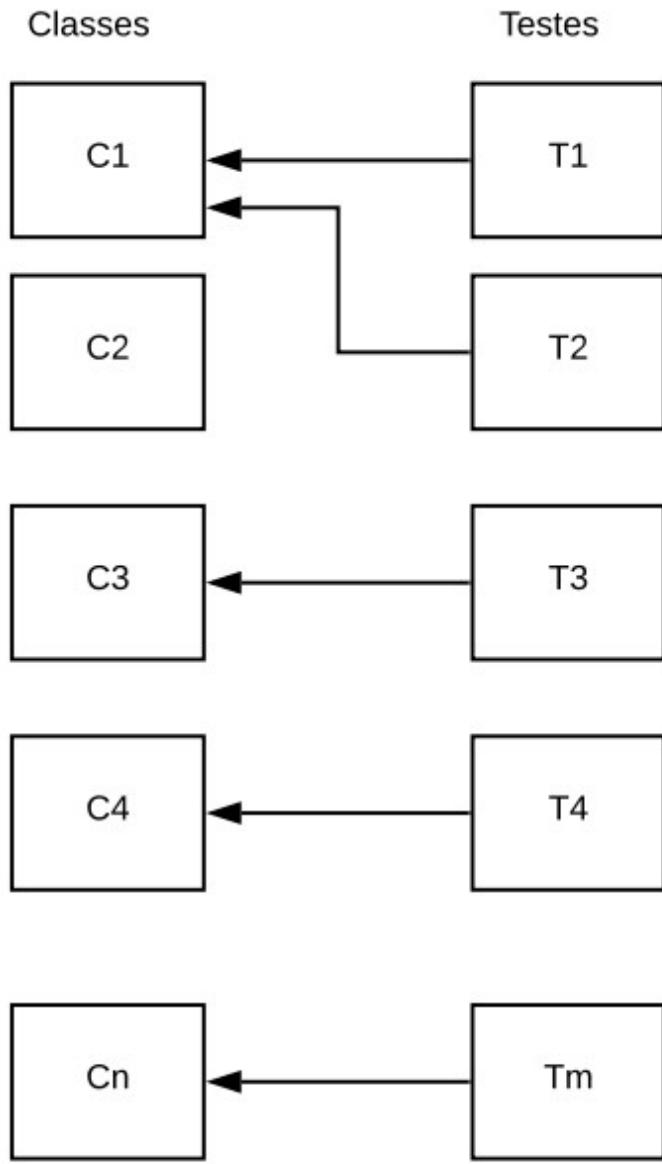
Uma recomendação genérica é que esses três testes sejam implementados na seguinte proporção: 70% como testes de unidades; 20% como testes de serviços e 10% como testes de sistema ([link](#), Capítulo 3).

Neste capítulo, vamos estudar os três tipos de testes da pirâmide de testes. O espaço que dedicaremos a cada teste também será compatível com seu espaço na pirâmide. Ou seja, falaremos mais de testes de unidade do que de testes de sistema, pois os primeiros são muito mais comuns. Antes de começar de fato, gostaríamos de relembrar alguns conceitos que apresentamos na Introdução. Diz-se que um código possui um **defeito** — ou

um **bug**, de modo mais informal — quando ele não está de acordo com a sua especificação. Se um código com defeito for executado e levar o programa a apresentar um resultado ou comportamento incorreto, dizemos que ocorreu uma **falha** (*failure*, em inglês).

## Testes de Unidade

Testes de unidade são testes automatizados de pequenas unidades de código, normalmente classes, as quais são testadas de forma isolada do restante do sistema. Um teste de unidade é um programa que chama métodos de uma classe e verifica se eles retornam os resultados esperados. Assim, quando se usa testes de unidades, o código de um sistema pode ser dividido em dois grupos: um conjunto de classes — que implementam os requisitos do sistema — e um conjunto de testes, conforme ilustrado na próxima figura.



### Correspondência entre classes e testes

A figura mostra um sistema com  $n$  classes e  $m$  testes. Como pode ser observado, não existe uma correspondência de 1 para 1 entre classes e testes. Por exemplo, uma classe pode ter mais de um teste. É o caso da classe C1, que é testada por T1 e T2. Provavelmente, isso ocorre porque C1 é uma classe importante, que precisa ser testada em diferentes contextos. Por outro lado, C2 não possui testes, ou porque os desenvolvedores esqueceram de implementar ou porque ela é uma classe menos importante.

Testes de unidade são implementados usando-se frameworks construídos especificamente para esse fim. Os mais conhecidos são chamados de frameworks **xUnit**, onde o **x** designa a linguagem usada na implementação dos testes. O primeiro desses frameworks — chamado **sUnit** — foi implementado por Kent Beck no final da década de 80 para Smalltalk. Neste capítulo, nossos testes serão implementados em Java, usando o **JUnit**. A primeira versão do JUnit foi implementada em conjunto por Kent Beck e Erich Gamma, em 1997, durante uma viagem de avião entre a Suíça e os EUA.

Hoje, existem versões de frameworks xUnit para as principais linguagens. Logo, uma das vantagens de testes de unidade é que os desenvolvedores não precisam aprender uma nova linguagem de programação, pois os testes são implementados na mesma linguagem do sistema que pretende-se testar.

Para explicar os conceitos de testes de unidade, vamos usar uma classe Stack:

```
import java.util.ArrayList;
import java.util.EmptyStackException;

public class Stack<T> {

    private ArrayList<T> elements = new ArrayList<T>();
    private int size = 0;

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return (size == 0);
    }

    public void push(T elem) {
        elements.add(elem);
        size++;
    }

    public T pop() throws EmptyStackException {
        if (isEmpty())
            throw new EmptyStackException();
        T elem = elements.remove(size-1);
        size--;
    }
}
```

```
        return elem;
    }
}
```

JUnit permite implementar classes que vão testar — de forma automática — classes da aplicação, como a classe `Stack`. Por convenção, classes de teste têm o mesmo nome das classes testadas, mas com um sufixo `Test`. Portanto, nossa primeira classe de teste vai se chamar `StackTest`. Já os métodos de teste começam com o prefixo `test` e devem, obrigatoriamente, atender às seguintes condições: (1) serem públicos, pois eles serão chamados pelo JUnit; (2) não possuírem parâmetros; (3) possuírem a anotação `@Test`, a qual identifica métodos que deverão ser executados durante um teste.

Mostramos a seguir nosso primeiro teste de unidade:

```
import org.junit.Test;
import static org.junit.Assert.assertTrue;

public class StackTest {

    @Test
    public void testEmptyStack() {
        Stack<Integer> stack = new Stack<Integer>();
        boolean empty = stack.isEmpty();
        assertTrue(empty);
    }
}
```

Nessa primeira versão, a classe `StackTest` possui um único método de teste, público, anotado com `@Test` e chamado `testEmptyStack()`. Esse método apenas cria uma pilha e testa se ela está vazia.

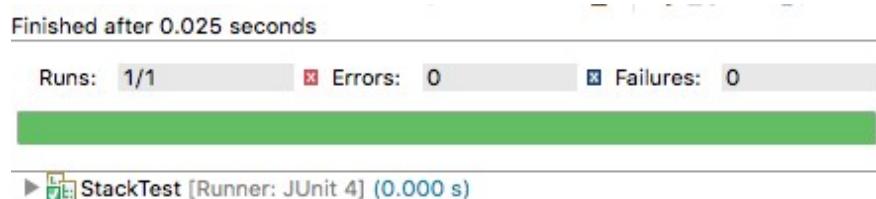
Métodos de teste têm a seguinte estrutura:

- Primeiro, cria-se o contexto do teste, também chamado de **fixture**. Para isso, deve-se instanciar os objetos que se pretende testar e, se for o caso, inicializá-los. No nosso primeiro exemplo, essa parte do teste inclui apenas a criação de uma pilha de nome `stack`.

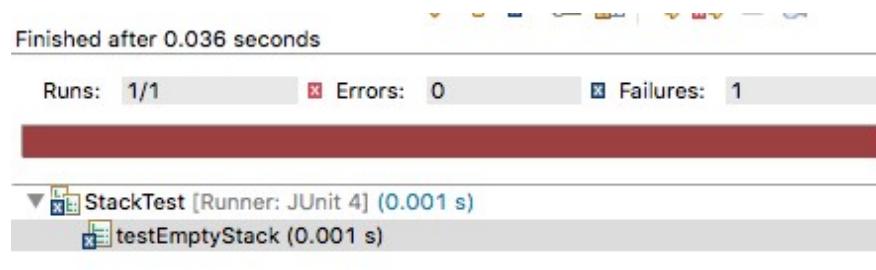
- Em seguida, o teste deve chamar um dos métodos da classe que está sendo testada. No exemplo, chamamos o método `isEmpty()` e armazenamos o seu resultado em uma variável local.
- Por fim, devemos testar se o resultado do método é aquele esperado. Para isso, deve-se usar um comando chamado **assert**. Na verdade, o JUnit oferece diversas variações de assert, mas todas têm o mesmo objetivo: testar se um determinado resultado é igual a um valor esperado. No exemplo, usamos `assertTrue`, que verifica se o valor passado como parâmetro é verdadeiro.

IDEs oferecem opções para rodar apenas os testes de um sistema, por exemplo, por meio de uma opção de menu chamada Run as Test. Ou seja, se o desenvolvedor chamar Run, ele irá executar o seu programa normalmente, começando pelo método `main`. No entanto, se ele optar pela opção Run as Test ele não irá executar o programa, mas apenas os seus testes de unidade.

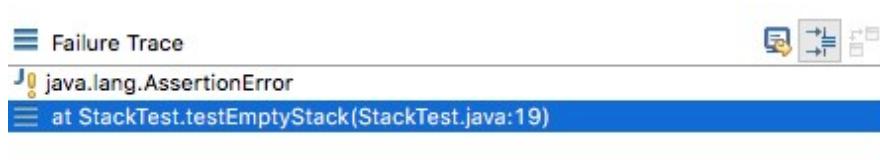
A próxima figura mostra o resultado da execução do nosso primeiro teste. O resultado é mostrado na própria IDE e a barra verde informa que todos os testes passaram. Pode-se observar também que o teste roda rapidamente, em 0.025 segundos.



Porém, suponha que tivéssemos cometido um erro na implementação da classe `Stack`. Por exemplo, suponha que o atributo `size` fosse inicializado com o valor 1, em vez de zero. Nesse caso, a execução dos testes iria falhar, como mostrado pela barra vermelha na IDE:



A mensagem de erro informa que houve uma falha durante a execução de `testEmptyStack`. Falha (*failure*) é o termo usado pelo JUnit para indicar testes cujo comando `assert` não foi satisfeito. Em uma outra janela da IDE, pode-se descobrir que a asserção responsável pela falha encontra-se na linha 19 do arquivo `StackTest.java`.



Para concluir, vamos mostrar o código completo do teste de unidade:

```
import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertEquals;

public class StackTest {

    Stack<Integer> stack;

    @Before
    public void init() {
        stack = new Stack<Integer>();
    }

    @Test
    public void testEmptyStack() {
        assertTrue(stack.isEmpty());
    }

    @Test
    public void testNotEmptyStack() {
        stack.push(10);
        assertFalse(stack.isEmpty());
    }

    @Test
    public void testSizeStack() {
        stack.push(10);
        stack.push(20);
        stack.push(30);
        int size = stack.size();
```

```

        assertEquals(3, size);
    }

    @Test
    public void testPushPopStack() {
        stack.push(10);
        stack.push(20);
        stack.push(30);
        int result = stack.pop();
        result = stack.pop();
        assertEquals(20, result);
    }

    @Test(expected = java.util.EmptyStackException.class)
    public void testEmptyStackException() {
        stack.push(10);
        int result = stack.pop();
        result = stack.pop();
    }

}

```

A classe StackTest tem cinco métodos de teste — todos com anotações @Test. Existe ainda um método, chamado `init()`, com uma anotação @Before. Esse método é executado pelo JUnit antes de qualquer método de teste. O JUnit funciona do seguinte modo: para cada classe de teste, ele chama cada um de seus métodos @Test. Cada método executa em uma instância diferente da classe de teste. Ou seja, antes de chamar um método @Test, o JUnit instancia um objeto da sua classe. Se essa classe tiver um método @Before, ele é executado antes do método @Test. No exemplo, usamos um método @Before para criar uma instância de Stack, a qual é usada depois pelos métodos @Test. Assim, evitamos repetir esse código de instanciação nos testes.

Para ficar um pouco mais claro, mostramos a seguir o algoritmo usado pelo JUnit para executar os testes de um programa:

```

para cada classe de teste TC
    para cada método m de TC com anotação @Test
        o = new TC();           // instancia objeto de teste
        se C possui um método b com anotação @Before
            então o.b();       // chama método @Before
        o.m();                  // chama método @Test

```

Voltando à classe StackTest, outro método interessante é aquele que testa a situação na qual a execução de um pop() levanta uma EmptyStackException. Veja que esse método — o último do nosso teste — não possui assert. O motivo é que um assert seria um código morto na sua implementação. A chamada de um pop() em uma pilha vazia terminaria a execução do método com uma exceção EmptyStackException. Ou seja, o assert não seria executado. Por isso, a anotação @Test tem um atributo especial que serve para especificar a exceção que deve ser levantada pelo método de teste. Em resumo, testEmptyStackException vai passar se a sua execução levantar uma EmptyStackException. Caso contrário, ele vai falhar.

**Código Fonte:** O código fonte de stack e StackTest está disponível neste [link](#).

**Aviso:** O framework JUnit possui várias versões. Neste capítulo, vamos usar a versão 4.12.

## Definições

Antes de avançar, vamos apresentar algumas definições:

- **Teste:** método que implementa um teste. O nome deriva da anotação @Test. São também chamados de método de teste (*test method*).
- **Fixture:** estado do sistema que será testado por um ou mais métodos de teste, incluindo dados, objetos, etc. O termo é reusado da indústria manufatureira, onde *fixture* é um equipamento que fixa uma peça que se pretende construir (veja uma [foto](#) na Wikipedia). No contexto de testes de unidade, a função de uma fixture é fixar o estado — isto é, os dados e objetos — exercitados no teste.
- **Casos de Teste (Test Case):** classe com os métodos de teste. O nome tem origem nas primeiras versões do JUnit. Nessas versões, os métodos de testes eram implementados em classes que herdavam de uma classe TestCase.

- **Suíte de Testes (Test Suite)**: conjunto de casos de teste, os quais são executados pelo framework de testes de unidade (que no nosso caso é o JUnit).
- **Sistema sob Teste (System Under Test, SUT)**: sistema que está sendo testado. É um nome genérico, usado também em outros tipos de testes, não necessariamente de unidades. Às vezes, usa-se também o termo **código de produção**, ou seja, código que vai ser executado pelos clientes do sistema.

## Quando Escrever Testes de Unidade?

Existem duas respostas principais para essa pergunta. Primeiro, pode-se escrever os testes após implementar uma pequena funcionalidade. Por exemplo, pode-se implementar alguns métodos e, em seguida, seus testes, que devem passar. Isto é, pode-se programar um pouco e escrever testes; programar mais um pouco e escrever novos testes, etc.

Alternativamente, pode-se escrever os testes primeiro, antes de qualquer código de produção. No início, esses testes não vão passar, pois isso somente vai acontecer depois que o código sob teste for implementado. Em outras palavras, inicia-se com um código que apenas compila e cujos testes, portanto, falham. Implementa-se então o código de produção e testa-se novamente. Agora, os testes devem passar. Esse estilo de desenvolvimento chama-se **Test-Driven Development**. Iremos discuti-lo com detalhes na Seção 8.7.

No entanto, existem duas respostas complementares para a questão sobre quando devemos escrever testes. Por exemplo, quando um usuário reportar um bug, pode-se começar sua análise escrevendo um teste que reproduz o bug e que, portanto, vai falhar. No passo seguinte, deve-se corrigir o bug. Se a correção for bem sucedida, o teste vai passar e ganhamos mais um teste para a suíte de testes.

Pode-se escrever testes também quando se estiver depurando um trecho de código. Por exemplo, evite escrever um `System.out.println` para testar manualmente o resultado de um método. Em vez disso, escreva um método de teste. Quando se usa um comando `println`, ele em algum momento é

removido. Já um teste tem a vantagem de contribuir com mais um teste para a suíte de testes.

Ainda sobre a pergunta principal desta seção, o que não é recomendável é deixar para implementar todos os testes após o sistema ficar pronto — tal como ocorria, por exemplo, com desenvolvimento em Waterfall. Se deixarmos para escrever os testes por último, eles podem ser construídos de forma apressada e com baixa qualidade. Ou então pode ser que eles nem sejam implementados, pois o sistema já estará funcionando e novas prioridades podem ter sido alocadas para o time de desenvolvimento. Por fim, não é recomendável que os testes sejam implementados por um outro time ou mesmo por uma outra empresa de desenvolvimento. Em vez disso, recomenda-se que o desenvolvedor de uma classe seja também responsável pela implementação de seus testes de unidade.

## Benefícios

O principal benefício de testes de unidade é encontrar bugs, ainda na fase de desenvolvimento e antes que o código entre em produção, quando os custos de correção e os prejuízos podem ser maiores. Portanto, se um sistema de software tem bons testes, é mais difícil que os usuários finais sejam surpreendidos com bugs.

Porém, existem dois outros benefícios que também são muito importantes. Primeiro, testes de unidade funcionam como uma rede de proteção contra **regressões** no código. Dizemos que uma regressão ocorre quando uma modificação realizada no código de um sistema — seja para corrigir um bug, implementar uma nova funcionalidade ou realizar uma refatoração — acaba por introduzir um bug ou outro problema semelhante no código. Ou seja, dizemos que o código regrediu porque algo que estava funcionando deixou de funcionar após a mudança que foi realizada. Regressões são mais raras quando se tem bons testes. Para isso, após concluir uma mudança o desenvolvedor deve rodar a suíte de testes. Se a mudança tiver introduzido alguma regressão, existe uma boa chance de que ela seja detectada pelos testes. Ou seja, antes da mudança os testes estavam passando, mas após a mudança algum teste começou a falhar.

Além de serem usados para detecção prematura de bugs e regressões no código, testes de unidade também ajudam na documentação e especificação do código de produção. De fato, ao olhar e analisar os testes implementados em StackTest podemos entender diversos aspectos do comportamento da classe Stack. Por isso, muitas vezes, antes de manter um código com o qual ele não tenha familiaridade, um desenvolvedor começa analisando os seus testes.

**Mundo Real:** Dentre as práticas de desenvolvimento propostas originalmente por métodos ágeis, testes de unidade é provavelmente a que alcançou o maior impacto e que é mais largamente usada. Hoje, os mais diversos sistemas de software, de empresas dos mais diferentes tamanhos, são desenvolvidos com o apoio de testes de unidade. A seguir, vamos destacar os casos de duas grandes empresas de software: Google e Facebook. Os comentários foram extraídos de artigos que documentam o processo e as práticas de desenvolvimento de software dessas empresas:

- Testes de unidade são fortemente encorajados e amplamente praticados no Google. Todo código de produção deve ter testes de unidade e nossa ferramenta de revisão de código automaticamente destaca código submetido sem os correspondentes testes. Os revisores de código normalmente exigem que qualquer mudança que adiciona novas funcionalidades deve também adicionar os respectivos testes. ([link](#))
- No Facebook, engenheiros são responsáveis pelos testes de unidade de qualquer código novo que eles desenvolvam. Além disso, esse código deve passar por testes de regressão, os quais são executados automaticamente, como parte dos processos de commit e push. ([link](#))

## Princípios e Smells

Nesta seção, vamos agrupar a apresentação de princípios e anti-padrões para implementação de testes de unidade. O objetivo é discutir questões importantes para a implementação de testes que tenham qualidade e que possam ser facilmente mantidos e entendidos.

### Princípios FIRST

Testes de unidades devem satisfazer às seguintes propriedades (cujas iniciais dão origem à palavra FIRST, em Inglês):

**Rápidos (Fast):** desenvolvedores devem executar testes de unidades frequentemente, para obter feedback rápido sobre bugs e regressões no código. Por isso, é importante que eles sejam executados rapidamente, em questões de milisegundos. Se isso não for possível, pode-se dividir uma suíte de testes em dois grupos: testes que executam rapidamente e que, portanto, serão frequentemente chamados; e testes mais demorados, que serão, por exemplo, executados uma vez por dia.

**Independentes:** a ordem de execução dos testes de unidade não é importante. Para quaisquer testes T1 e T2, a execução de T1 seguida de T2 deve ter o mesmo resultado da execução de T2 e depois T1. Pode acontecer ainda de T1 e T2 serem executados de forma concorrente. Para que os testes sejam independentes, T1 não deve alterar alguma parte do estado global do sistema que depois será usada para computar o resultado de T2 e vice-versa.

**Determinísticos (Repeatable):** testes de unidade devem ter sempre o mesmo resultado. Ou seja, se um teste T é chamado  $n$  vezes, o resultado deve ser o mesmo nas  $n$  execuções. Isto é, ou T passa em todas as execuções; ou ele sempre falha. Testes com resultados não-determinísticos são chamados de **Testes Flaky** (ou **Testes Erráticos**). Concorrência é uma das principais responsáveis por comportamento flaky. Um exemplo é mostrado a seguir:

```
@Test
public void exemploTesteFlaky {
    TaskResult resultado;
    MyMath m = new MyMath();
    m.asyncPI(10, resultado);
    Thread.sleep(1000);
    assertEquals(3.1415926535, resultado.get());
}
```

Esse teste chama uma função que calcula o valor de PI, com uma certa precisão, e de forma assíncrona — isto é, a função realiza o seu cálculo em uma nova thread, que ela mesma cria internamente. No exemplo, a precisão requerida são 10 casas decimais. O teste faz uso de um sleep para esperar que a função assíncrona termine. No entanto, isso torna o seu

comportamento não-determinístico: se a função terminar antes de 1000 milissegundos, o teste irá passar; mas se a execução, por alguma circunstância particular, demorar mais, o teste irá falhar. Uma possível alternativa seria testar apenas a versão síncrona da função. Se essa versão não existir, um refactoring poderia ser realizado para extraí-la do código da versão assíncrona. Na Seção 8.5, iremos discutir mais sobre questões relativas à testabilidade do código de produção.

Pode parecer que testes flaky são raros, mas um estudo divulgado pelo Google, com seus próprios testes, revelou que cerca de 16% deles estão sujeitos a resultados não-determinísticos ([link](#)). Ou seja, esses testes podem falhar não porque um bug foi introduzido no código, mas por causa de eventos não determinísticos, como uma thread que levou mais tempo para executar. Testes flaky são ruins porque eles atrasam o desenvolvimento: os programadores perdem um tempo para investigar a falha, para então descobrir que ela é um alarme falso.

**Auto-verificáveis (Self-checking):** O resultado de um teste de unidades deve ser facilmente verificável. Para interpretar o resultado do teste, o desenvolvedor não deve, por exemplo, ter que abrir e analisar um arquivo de saída ou fornecer dados manualmente. Em vez disso, o resultado dos testes deve ser binário e mostrado na IDE, normalmente por meio de componentes que ficam com a cor verde (para indicar que todos os testes passaram) ou com a cor vermelha (para indicar que algum teste falhou). Adicionalmente, quando um teste falha, deve ser possível identificar essa falha de forma rápida, incluindo a localização do comando assert que falhou.

**Escritos o quanto antes (Timely)**, se possível antes mesmo do código que vai ser testado, como já comentamos no final da Seção 8.2 e iremos discutir com mais profundidade na seção sobre Desenvolvimento Dirigido por Testes (Seção 8.7).

## Test Smells

**Test Smells** representam estruturas e características preocupantes no código de testes de unidade, as quais, a princípio deveriam ser evitadas. O nome é uma adaptação, para o contexto de testes, do conceito de **Code Smells** ou **Bad Smells**, que iremos estudar no Capítulo 9. No entanto, neste capítulo,

vamos aproveitar e já comentar sobre smells que podem ocorrer no código de testes.

Um **Teste Obscuro** é um teste longo, complexo e difícil de entender. Como afirmamos, testes devem ser usados também para auxiliar na documentação do sistema sob teste. Por isso, é importante que eles tenham uma lógica clara e de rápido entendimento. Idealmente, um teste deve, por exemplo, testar um único requisito do sistema sob teste.

Um **Teste com Lógica Condisional** inclui código que pode ou não ser executado. Isto é, são testes com comandos `if` ou laços, enquanto que o ideal é que os testes de unidade sejam lineares. Lógica condicional em testes é considerada um smell porque ela prejudica o entendimento do teste.

**Duplicação de Código em Testes** ocorre, como o próprio nome sugere, quando temos código repetido em diversos métodos de teste.

No entanto, um test smell não deve ser interpretado ao pé da letra, isto é, como sendo uma situação que deve ser evitada a todo custo. Em vez disso, eles devem ser considerados como um alerta para os implementadores do teste. Ao identificar um test smell, os desenvolvedores devem refletir sobre se não é possível ter um teste mais simples e menor, com um código linear e sem duplicação de comandos.

Por fim, assim como ocorre com código de produção, código de testes deve ser frequentemente refatorado, de modo a garantir que ele permaneça simples, fácil de entender e livre dos test smells que comentamos nesta seção.

## Número de assert por Teste

Alguns autores ([link](#)) recomendam que deve existir no máximo um assert por teste. Ou seja, eles recomendam escrever um código como o seguinte.

```
@Test  
public void testEmptyStack() {  
    assertTrue(stack.isEmpty());  
}  
  
@Test
```

```
public void testNotEmptyStack() {  
    stack.push(10);  
    assertFalse(stack.isEmpty());  
}
```

Em outras palavras, *não* se recomenda usar dois comandos assert no mesmo método, como no código a seguir:

```
@Test  
public void testEmptyStack() {  
    assertTrue(stack.isEmpty());  
    stack.push(10);  
    assertFalse(stack.isEmpty());  
}
```

O primeiro exemplo, que divide o teste de pilha vazia em dois testes, tende a ser mais legível e fácil de entender do que o segundo, que faz tudo em um único teste. Além disso, quando o teste do primeiro exemplo falha, é mais simples detectar o motivo da falha do que no segundo exemplo, que pode falhar por dois motivos.

No entanto, não devemos ser dogmáticos no emprego dessa regra ([link](#), Capítulo 4). O motivo é que existem casos nos quais justifica-se ter mais de um assert por método. Por exemplo, suponha que precisamos testar uma função `getBook` que retorna um objeto com dados de um livro, incluindo título, autor, ano e editora. Nesse caso, justifica-se ter quatro comandos assert no mesmo teste, cada um verificando um dos campos do objeto retornado pela função, como mostra o seguinte código.

```
@Test  
public void testBookService() {  
    BookService bs = new BookService();  
    Book b = bs.getBook(1234);  
    assertEquals("Engenharia Software Moderna", b.getTitle());  
    assertEquals("Marco Túlio Valente", b.getAuthor());  
    assertEquals("2020", b.getYear());  
    assertEquals("ASERG/DCC/UFMG", b.getPublisher());  
}
```

Uma segunda exceção é quando temos um método simples, que pode ser testado por meio de um único assert. Para ilustrar, mostramos o teste da função `repeat` da classe `Strings` da biblioteca google/guava ([link](#)):

```
@Test
public void testRepeat() {
    String input = "20";
    assertEquals("", Strings.repeat(input,0));
    assertEquals("20", Strings.repeat(input,1));
    assertEquals("2020", Strings.repeat(input,2));
    assertEquals("202020", Strings.repeat(input,3));
    ...
}
```

Nesse teste, temos quatro comandos `assertEquals`, os quais testam, respectivamente, o resultado da repetição de uma determinada string zero, uma, duas e três vezes.

## Cobertura de Testes

Cobertura de testes é uma métrica que ajuda a definir o número de testes que precisamos escrever para um programa. Ela mede o percentual de comandos de um programa que são cobertos por testes, isto é:

$$\text{cobertura de testes} = (\text{número de comandos executados pelos testes}) / (\text{total de comandos do programa})$$

Existem ferramentas para cálculo de cobertura de testes. Na próxima figura, mostramos um exemplo de uso da ferramenta que acompanha a IDE Eclipse. As linhas com fundo verde — coloridas automaticamente por essa ferramenta — indicam as linhas cobertas pelos cinco testes implementados em `StackTest`. As únicas linhas não coloridas de verde são responsáveis pela assinatura dos métodos de `Stack` e, portanto, não correspondem a comandos executáveis. Assim, a cobertura dos testes do nosso primeiro exemplo é de 100%, pois a execução dos métodos de testes resulta na execução de todos os comandos da classe `Stack`.

```

public class Stack<T> {
    private ArrayList<T> elements = new ArrayList<T>();
    private int size = 0;

    public int size() {
        return size;
    }

    public boolean isEmpty(){
        return (size == 0);
    }

    public void push(T elem) {
        elements.add(elem);
        size++;
    }

    public T pop() throws EmptyStackException {
        if (isEmpty())
            throw new EmptyStackException();
        T elem = elements.get(size-1);
        size--;
        return elem;
    }
}

```

Suponha agora que não tivéssemos implementado `testEmptyStackException`. Isto é, não iríamos testar o levantamento de uma exceção pelo método `pop()`, quando chamado com uma pilha vazia. Nesse caso, a cobertura dos testes cairia para 92.9%, como ilustrado a seguir:

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
▼ Stack	97.1 %	132	4	136
▼ src	97.1 %	132	4	136
▼ (default package)	97.1 %	132	4	136
▶ Stack.java	92.9 %	52	4	56
▶ StackTest.java	100.0 %	80	0	80

Nesse caso, a ferramenta de cálculo de cobertura de testes marcaria as linhas da classe `Stack` da seguinte forma:

```

public class Stack<T> {
    private ArrayList<T> elements = new ArrayList<T>();
    private int size = 0;

    public int size() {
        return size;
    }

    public boolean isEmpty(){
        return (size == 0);
    }

    public void push(T elem) {
        elements.add(elem);
        size++;
    }

    public T pop() throws EmptyStackException {
        if (isEmpty())
            throw new EmptyStackException();
        T elem = elements.get(size-1);
        size--;
        return elem;
    }
}

```

Como afirmamos, as linhas verdes são cobertas pela execução dos testes. Porém, existe um comando marcado de amarelo. Essa cor indica que o comando é um desvio (no caso, um `if`) e que apenas um dos caminhos possíveis do desvio (no caso, o caminho `false`) foi exercitado pelos testes de unidade. Por fim, o leitor já deve ter observado que existe uma linha em vermelho. Essa cor indica linhas que não foram cobertas pelos testes de unidade.

Em Java, ferramentas de cobertura de testes trabalham instrumentando os bytecodes gerados pelo compilador da linguagem. Como mostrado na figura com as estatísticas de cobertura, o programa anterior, após compilado, possui 52 instruções cobertas por testes de unidade, de um total de 56 instruções. Portanto, sua cobertura é  $52 / 56 = 92.9\%$ .

## Qual a Cobertura de Testes Ideal?

Não existe um número mágico e absoluto para cobertura de testes. A resposta varia de projeto para projeto, dependendo da complexidade dos requisitos, da criticidade do projeto, etc. Mas, em geral, não precisa ser 100%, pois sempre existem métodos triviais em um sistema; por exemplo, *getters* e *setters*. Também sempre temos métodos cujo teste é mais desafiador, como métodos de interface com o usuário ou métodos com comportamento assíncrono.

Portanto, não recomenda-se fixar um valor de cobertura que tenha que ser sempre atingido. Em vez disso, deve-se monitorar a evolução dos valores de cobertura ao longo do tempo, para verificar se os desenvolvedores, por exemplo, não estão relaxando na escrita de testes. Recomenda-se também avaliar cuidadosamente os trechos não cobertos por testes, para confirmar que eles não são relevantes ou então são difíceis de serem testados.

Feitas essas considerações, times que valorizam a escrita de testes costumam atingir facilmente valores de cobertura próximos de 70% ([link](#)). Por outro lado, valores abaixo de 50% tendem a ser preocupantes ([link](#)). Por fim, mesmo quando se usa TDD, a cobertura de testes costuma não chegar a 100%, embora normalmente fique acima de 90% ([link](#)).

**Mundo Real:** Em uma conferência de desenvolvedores do Google, em 2014, foram apresentadas algumas estatísticas sobre a cobertura de testes dos sistemas da empresa (veja os [slides](#) e também o [vídeo](#)). Na mediana, os sistemas do Google tinham 78% de cobertura, em nível de comandos. Segundo afirmou-se na palestra, a recomendação seria atingir 85% de cobertura na maioria dos sistemas, embora essa recomendação não seria escrita em pedra, ou seja, não teria que ser seguida de forma dogmática. Mostrou-se também que a cobertura variava por linguagem de programação. A menor cobertura era dos sistemas em C++, um pouco inferior a 60% na média dos projetos. A maior foi medida para sistemas implementados em Python, um pouco acima de 80%.

## Outras Definições de Cobertura de Testes

A definição de métrica de cobertura, apresentada acima, foi baseada em comandos, pois trata-se de sua definição mais comum. Porém, existem definições alternativas, tais como **cobertura de funções** (percentual de funções que são executadas por um teste), **cobertura de chamadas de funções** (dentre todas as linhas de um programa que chamam funções, quantas são, de fato, exercitadas por testes), **cobertura de branches** (% de branches de um programa que são executados por testes; um comando `if` sempre gera dois branches: quando a condição é verdadeira e quando ela é falsa). Cobertura de comandos e de branches são também chamadas de **Cobertura C0** e **Cobertura C1**, respectivamente. Para ilustrar a diferença entre ambas vamos usar a seguinte classe (primeiro código) e seu teste de unidade (segundo código):

```
public class Math {  
  
    public int abs(int x) {  
        if (x < 0) {  
            x = -x;  
        }  
        return x;  
    }  
  
}  
  
public class MathTest {  
  
    @Test  
    public void testAbs() {  
        Math m = new Math();  
        assertEquals(1, m.abs(-1));  
    }  
  
}
```

Supondo cobertura de comandos, temos uma cobertura de 100%. Porém, supondo uma cobertura de branches, o valor é 50%, pois dentre as duas condições possíveis do comando `if(x < 0)`, testamos apenas uma delas (a condição verdadeira). Se quisermos ter uma cobertura de branches de 100% teríamos que adicionar mais um comando `assert`, como: `assertEquals(1, m.abs(1))`. Logo, cobertura de branches é mais rigorosa do que cobertura de comandos.

# Testabilidade

Testabilidade é uma medida de quanto fácil é implementar testes para um sistema. Como vimos, é importante que os testes sigam os princípios FIRST, que eles tenham poucos assert e uma alta cobertura. No entanto, é importante também que o projeto do código de produção favoreça a implementação de testes. O termo em inglês para isso é **design for testability**. Em outras palavras, às vezes, parte relevante do esforço para escrita de bons testes deve ser alocada no projeto do sistema sob teste e não exatamente no projeto dos testes.

A boa notícia é que código que segue as propriedades e princípios de projeto que discutimos no Capítulo 5 — tais como coesão alta, acoplamento baixo, responsabilidade única, separação entre apresentação e modelo, inversão de dependências, Demeter, dentre outros — tende a apresentar boa testabilidade.

## Exemplo: Servlet

Servlet é uma tecnologia de Java para implementação de páginas Web dinâmicas. A seguir mostramos uma servlet que calcula o índice de massa corporal de uma pessoa, dado o seu peso e altura. O nosso objetivo é didático. Logo, não vamos detalhar todo o protocolo para implementação de servlets. Além disso, a lógica de domínio desse exemplo é simples, consistindo na seguinte fórmula: peso / (altura \* altura). Mas tente imaginar que essa lógica poderia ser mais complexa e que, mesmo assim, a solução que vamos apresentar continuaria válida.

```
public class IMCServlet extends HttpServlet {  
  
    public void doGet(HttpServletRequest req,  
                      HttpServletResponse res) {  
        res.setContentType("text/html");  
        PrintWriter out = res.getWriter();  
        String peso = req.getParameter("peso");  
        String altura = req.getParameter("altura");  
        try {  
            double p = Double.parseDouble(peso);  
            double a = Double.parseDouble(altura);  
            double imc = p / (a * a);  
            out.println("O IMC é: " + imc);  
        } catch (Exception e) {  
            out.println("Erro ao calcular IMC");  
        }  
    }  
}
```

```

        out.println("Índice de Massa Corporal (IMC): "+imc);
    }
    catch (NumberFormatException e) {
        out.println("Dados devem ser numéricos");
    }
}
}

```

Primeiro, veja que não é simples escrever um teste para `IMCServlet`, pois essa classe depende de diversos tipos do pacote de Servlets de Java. Por exemplo, não é trivial instanciar um objeto do tipo `IMCServlet` e depois chamar `doGet`. Se tomarmos esse caminho, teríamos que criar também objetos dos tipos `HttpServletRequest` e `HttpServletResponse`, para passar como parâmetro de `doGet`. No entanto, esses dois tipos podem depender de outros tipos e assim sucessivamente. Portanto, a testabilidade de `IMCServlet` é baixa.

Uma alternativa para testar o exemplo mostrado seria extrair a sua lógica de domínio para uma classe separada, como feito no código a seguir. Ou seja, a ideia consiste em separar apresentação (via Servlet) de lógica de domínio. Com isso, fica mais fácil testar a classe extraída, chamada `IMCModel`, pois ela não depende de tipos relacionados com Servlet. Por exemplo, é mais fácil instanciar um objeto da classe `IMCModel` do que da classe `IMCServlet`. É verdade que com essa refatoração não vamos testar o código completo. Porém, é melhor testar a parte de domínio do sistema do que deixar o código inteiramente descoberto de testes.

```

class IMCModel {
    public double calculaIMC(String p1, String a1)
        throws NumberFormatException {
        double p = Double.parseDouble(p1);
        double a = Double.parseDouble(a1);
        return p / (a * a);
    }
}

public class IMCServlet extends HttpServlet {
    IMCModel model = new IMCModel();

    public void doGet(HttpServletRequest req,
                      HttpServletResponse res) {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();

```

```

String peso = req.getParameter("peso");
String altura = req.getParameter("altura");
try {
    double imc = model.calculaIMC(peso, altura);
    out.println("Índice de Massa Corporal (IMC): " + imc);
}
catch (NumberFormatException e) {
    out.println("Dados devem ser numéricos");
}
}
}
}

```

**Código Fonte:** O código dessa servlet está disponível neste [link](#).

## Exemplo: Chamada Assíncrona

O próximo código mostra a implementação da função `asyncPI` que mencionamos na Seção 8.3 quando tratamos dos princípios FIRST e, especificamente, de testes determinísticos. Conforme explicamos nessa seção, não é simples testar uma função assíncrona, pois seu resultado é computado por uma thread independente. O exemplo que mostramos na Seção 8.3 usava um `sleep` para esperar o resultado ficar disponível. Porém, o uso desse comando torna o teste não-determinístico.

```

public class MyMath {

    public void asyncPI(int prec, TaskResult task) {
        new Thread (new Runnable() {
            public void run() {
                double pi = "calcula PI com precisão prec"
                task.setResult(pi);
            }
        }).start();
    }
}

```

A seguir mostramos uma solução para incrementar a testabilidade dessa classe. Primeiro, extraímos o código que implementa a computação de PI para uma função separada, chamada `syncPI`. Assim, apenas essa função seria testada por um teste de unidade. Em suma, vale a observação que fizemos antes: é melhor extrair uma função que seja fácil de ser testada, do que deixar o código sem testes.

```

public class MyMath {
    public double syncPI(int prec) {
        double pi = "calcula PI com precisão prec"
        return pi;
    }
    public void asyncPI(int prec, TaskResult task) {
        new Thread (new Runnable() {
            public void run() {
                double pi = syncPI(prec);
                task.setResult(pi);
            }
        }).start();
    }
}

```

**Código Fonte:** O código desse exemplo de chamada assíncrona está disponível neste [link](#).

## Mocks

Para explicar o papel desempenhado por mocks em testes de unidade, vamos começar com um exemplo motivador e discutir porque é difícil escrever um teste de unidade para ele. Em seguida, vamos introduzir o conceito de mocks como uma possível solução para testar esse exemplo.

**Aviso:** Neste capítulo, inicialmente, usamos **mock** como sinônimo de **stub**. No entanto, incluímos uma subseção mais à frente para ressaltar que alguns autores fazem uma distinção entre esses termos.

**Exemplo Motivador:** Para explicar o conceito de mocks, vamos partir de uma classe simples para pesquisa de livros, cujo código é mostrado a seguir. Essa classe, chamada BookSearch, implementa um método getBook, que pesquisa os dados de um livro em um serviço remoto. Esse serviço, por sua vez, implementa a interface BookService. Para o exemplo ficar mais real, suponha que BookService é uma API REST ou uma base de dados. O importante é que a pesquisa é realizada em outro sistema, que fica abstruído pela interface BookService. Esse serviço retorna o seu resultado como um documento JSON, isto é, um documento textual. Assim, cabe ao método getBook acessar o serviço remoto, obter a resposta em formato JSON e criar um objeto da classe Book para armazenar a resposta. Para simplificar o

exemplo, não mostramos o código da classe Book, mas ela é apenas uma classe com dados de livros e seus respectivos métodos get. Na verdade, para simplificar um pouco mais, o exemplo considera que Book possui um único campo, relativo ao seu título. Em um programa real, Book teria outros campos, que também seriam tratados em getBook.

```
import org.json.JSONObject;

public class BookSearch {

    BookService rbs;

    public BookSearch(BookService rbs) {
        this.rbs = rbs;
    }

    public Book getBook(int isbn) {
        String json = rbs.search(isbn);
        JSONObject obj = new JSONObject(json);
        String titulo;
        titulo = (String) obj.get("titulo");
        return new Book(titulo);
    }

}

public interface BookService {
    String search(int isbn);
}
```

**Problema:** Precisamos implementar um teste de unidade para BookSearch. Porém, por definição, um teste de unidade exercita um componente pequeno do código, como uma única classe. O problema é que para testar BookSearch precisamos de um BookService, que é um serviço externo. Ou seja, se não tomarmos cuidado, o teste de getBook vai alcançar um serviço externo. Isso é ruim por dois motivos: (1) o escopo do teste ficará maior do que uma única unidade de código; (2) o teste ficará mais lento, pois o serviço externo pode ser uma base de dados, armazenada em disco, ou então um serviço remoto, acessado via HTTP ou um protocolo similar. E devemos lembrar que testes de unidades devem executar rapidamente, conforme recomendado pelos princípios FIRST (Seção 8.3).

**Solução:** Uma solução consiste em criar um objeto que emula o objeto real, mas apenas para permitir o teste do programa. Esse tipo de objeto é chamado de **mock** (ou então **stub**). No nosso exemplo, o mock deve implementar a interface BookService e, portanto, o método search. Porém, essa implementação é parcial, pois o mock retorna apenas os títulos de alguns livros, sem acessar servidores remotos ou bancos de dados. Um exemplo é mostrado a seguir:

```
import static org.junit.Assert.*;
import org.junit.*;
import static org.junit.Assert.*;

class BookConst {

    public static String ESM =
        "{\"titulo\": \"Eng Soft Moderna\" }";

    public static String NULLBOOK =
        "{\"titulo\": \"NULL\" }";

}

class MockBookService implements BookService {

    public String search(int isbn) {
        if (isbn == 1234)
            return BookConst.ESM;
        return BookConst.NULLBOOK;
    }
}

public class BookSearchTest {

    private BookService service;

    @Before
    public void init() {
        service = new MockBookService();
    }

    @Test
    public void testGetBook() {
        BookSearch bs = new BookSearch(service);
        String titulo = bs.getBook(1234).getTitulo();
        assertEquals("Eng Soft Moderna", titulo);
```

```
    }  
}
```

Nesse exemplo, `MockBookService` é uma classe usada para criar mocks de `BookService`, isto é, objetos que implementam essa interface, mas com um comportamento trivial. No exemplo, o objeto mock, de nome `service`, somente retorna dados do livro cujo ISBN é 1234. O leitor pode então estar se perguntando: qual a utilidade de um serviço que pesquisa dados de um único livro? A resposta é que esse mock nos permite implementar um teste de unidade que não precisa acessar um serviço remoto, externo e lento. No método `testGetBook`, usa-se o mock para criar um objeto do tipo `BookSearch`. Em seguida, chama-se o método `getBook` para pesquisar por um livro e retornar o seu título. Por fim, executa-se um `assert`. Como o teste é baseado em um `MockBookService`, ele verifica se o título retornado é aquele do único livro pesquisado por tal mock.

Porém, talvez ainda reste uma pergunta: o que, de fato, `testGetBook` testa? Em outras palavras, qual requisito do sistema está sendo testado por meio de um objeto mock tão simples? Claro, nesse caso, não estamos testando o acesso ao serviço remoto. Como foi afirmado, esse é um requisito muito extenso para ser verificado via testes de unidade. Em vez disso, estamos testando se a lógica de instanciar um `Book` a partir de um documento JSON está funcionando. Em um teste mais real, poderíamos incluir mais campos em `Book`, além do título. Poderíamos também testar com mais alguns livros, bastando estender a capacidade do mock: em vez de retornar sempre o JSON do mesmo livro, ele retornaria dados de mais livros, dependendo do ISBN.

**Código Fonte:** O código do exemplo de mock usado nesta seção está disponível neste [link](#).

## Frameworks de Mocks

Mocks são tão comuns em testes de unidade que existem frameworks para facilitar a criação e programação de mocks (e/ou stubs). Não vamos entrar em detalhes desses frameworks, mas abaixo mostramos o teste anterior, mas com um mock instanciado por um framework chamado **mockito** ([link](#)), muito usado quando se escreve testes de unidade em Java que requerem mocks.

```

import org.junit.*;
import static org.junit.Assert.*;
import org.mockito.Mockito;
import static org.mockito.Mockito.when;
import static org.mockito.Matchers.anyInt;

public class BookSearchTest {

    private BookService service;

    @Before
    public void init() {
        service = Mockito.mock(BookService.class);
        when(service.search(anyInt()))
            .thenReturn(BookConst.NULLBOOK);
        when(service.search(1234)).thenReturn(BookConst.ESM);
    }

    @Test
    public void testGetBook() {
        BookSearch bs = new BookSearch(service);
        String titulo = bs.getBook(1234).getTitulo();
        assertEquals("Eng Soft Moderna", titulo);
    }
}

```

Primeiro, podemos ver que não existe mais uma classe MockBookService. O principal ganho de se usar um framework como o mockito é exatamente esse: não ter mais que escrever classes de mock manualmente. Em vez disso, um mock para BookService é criado pelo próprio framework usando-se dos recursos de **reflexão computacional** de Java. Para isso, basta usar a função `mock(type)`, como a seguir:

```
service = Mockito.mock(BookService.class);
```

No entanto, o mock service ainda está vazio e sem nenhum comportamento. Temos então que ensiná-lo a se comportar pelo menos em algumas situações. Especificamente, temos que ensiná-lo a responder a algumas pesquisas de livros. Para isso, o mockito oferece uma **linguagem de domínio específico** simples, baseada na mesma sintaxe de Java. Um exemplo é mostrado a seguir:

```
when(service.search(anyInt())).thenReturn(BookConst.NULLBOOK);  
when(service.search(1234)).thenReturn(BookConst.ESM);
```

Essas duas linhas programam o mock service. Primeiro, dizemos para ele retornar BookConst.NULLBOOK quando o seu método search for chamado com qualquer inteiro como argumento. Em seguida, abrimos uma exceção a essa regra geral: quando search for chamado com o inteiro 1234, ele deve retornar a string JSON com os dados do livro BookConst.ESM.

**Código Fonte:** O código desse exemplo, usando o Mockito, está neste [link](#).

## Mocks vs Stubs

Alguns autores, como Gerard Meszaros ([link](#)), fazem uma distinção entre **mocks** e **stubs**. Segundo eles, mocks devem verificar não apenas o estado do Sistema sob Testes (SUT), mas também o seu comportamento. Se os mocks verificam apenas o estado, eles deveriam ser chamados de stubs. No entanto, neste livro, não vamos fazer essa distinção, pois achamos que ela é útil e, portanto, os benefícios não compensam o custo de páginas extras para explicar conceitos semelhantes. Porém, apenas para esclarecer um pouco mais, um **teste comportamental** — também chamado de teste de interação — verifica eventos que ocorreram no SUT. Segue um exemplo:

```
void testBehaviour {  
    Mailer m = mock(Mailer.class);  
    sut.someBusinessLogic(m);  
    verify(m).send(anyString());  
}
```

Nesse exemplo, o comando `verify` — implementado pelo Mockito — é parecido com um `assert`. No entanto, ele verifica se um evento ocorreu com o mock passado como argumento. No caso, verificamos se o método `send` do mock foi executado pelo menos uma vez, usando qualquer string como argumento.

Segundo Gerard Meszaros, mocks e stubs são casos especiais de **objetos dublê** (*double*). O termo é inspirado em dublês de atores em filmes. Segundo Meszaros, existem pelo menos mais dois outros tipos de objetos dublê:

- **Objetos Dummy** são objetos que são passados como argumento para um método, mas que não são usados. Trata-se, portanto, de uma forma de duplê usada apenas para satisfazer o sistema de tipos da linguagem.
- **Objeto Fake** são objetos que possuem uma implementação mais simples do que o objeto real. Por exemplo, um objeto que simula em memória principal, por meio de tabelas hash, um objeto de acesso a bancos de dados.

## Exemplo: Servlet

Na seção anterior, mostramos o teste de uma servlet que calcula o Índice de Massa Corporal (IMC) de uma pessoa. No entanto, argumentamos que não iríamos testar a servlet completa porque ela possui dependências difíceis de serem recriadas em um teste. No entanto, agora sabemos que podemos criar mocks para essas dependências, isto é, objetos que vão simular as dependências reais, porém respondendo apenas às chamadas que precisamos no teste.

Primeiro, vamos reapresentar o código da servlet que queremos testar:

```
public class IMCServlet extends HttpServlet {
    IMCModel model = new IMCModel();

    public void doGet(HttpServletRequest req,
                      HttpServletResponse res) {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        String peso = req.getParameter("peso");
        String altura = req.getParameter("altura");
        double imc = model.calculaIMC(peso, altura);
        out.println("IMC: " + imc);
    }
}
```

Segue então o novo teste dessa servlet (ele é uma adaptação de um exemplo disponível em um [artigo](#) de autoria de Dave Thomas e Andy Hunt). Primeiro, podemos ver, no método `init`, que foram criados mocks para objetos dos tipos `HttpServletRequest` e `HttpServletResponse`. Esses

Mocks serão usados como parâmetros da chamada de `doGet` que faremos no método de teste. Ainda em `init`, criamos um objeto do tipo `StringWriter` que permite gerar saídas na forma de uma lista de `Strings`. Em seguida, esse objeto é encapsulado por um `PrintWriter`, que é o objeto usado como saída pela servlet — ou seja, trata-se de uma aplicação padrão de projeto Decorador, que estudamos no Capítulo 6. Por fim, programamos o mock de resposta: quando a servlet pedir um objeto de saída, por meio de uma chamada a `getWriter()`, ele deve retornar o objeto `PrintWriter` que acabamos de criar. Em resumo, fizemos tudo isso com o objetivo de alterar a saída da servlet para uma lista de strings.

```
public class IMCServletTest {  
  
    HttpServletRequest req;  
    HttpServletResponse res;  
  
    StringWriter sw;  
  
    @Before  
    public void init() {  
        req = Mockito.mock(HttpServletRequest.class);  
        res = Mockito.mock(HttpServletResponse.class);  
        sw = new StringWriter();  
        PrintWriter pw = new PrintWriter(sw);  
        when(res.getWriter()).thenReturn(pw);  
    }  
    // ...continua a seguir
```

Para concluir, temos o método de teste, mostrado a seguir.

```
// continuação de IMCServletTest  
@Test  
public void testDoGet() {  
    when(req.getParameter("peso")).thenReturn("82");  
    when(req.getParameter("altura")).thenReturn("1.80");  
    new IMCServlet().doGet(req, res);  
    assertEquals("IMC: 25.3\n", sw.toString());  
}  
}
```

Nesse teste, começamos programando o mock do objeto com os parâmetros de entrada da servlet. Quando a servlet pedir o parâmetro peso, o mock vai retornar 82; quando a servlet pedir o parâmetro altura, ele deve retornar 1.80.

Feito isso, o teste segue o fluxo normal de testes de unidades: chamamos o método que queremos testar, `doGet`, e verificamos se ele retorna o resultado esperado.

Esse exemplo serve também para ilustrar as desvantagens do uso de mocks. A principal delas é o fato de mocks aumentarem o acoplamento entre o teste e o método testado. Tipicamente, em testes de unidade, o método de teste chama o método testado e verifica seu resultado. Logo, ele se acopla apenas à assinatura deste método. Por isso, o teste não é quebrado quando apenas o código interno do método testado é modificado. No entanto, quando usamos mocks, isso deixa de ser verdade, pois o mock pode depender de estruturas internas do método testado, o que torna os testes mais frágeis. Por exemplo, suponha que a saída da servlet mude para Índice de Massa Corporal (IMC): [valor]. Nesse caso, teremos que lembrar de atualizar também o `assertEquals` do teste de unidade.

Por fim, não conseguimos criar mocks para todos os objetos e métodos. Em geral, as seguintes construções não são mockáveis: classes e métodos finais, métodos estáticos e construtores.

**Código Fonte:** O código do teste dessa servlet, usando mocks, está disponível neste [link](#).

## Desenvolvimento Dirigido por Testes (TDD)

Desenvolvimento Dirigido por Testes (*Test Driven Development*, TDD) é uma das práticas de programação propostas por Extreme Programming (XP). A ideia a princípio pode parecer estranha, talvez até absurda: dado um teste de unidade T para uma classe C, TDD defende que T deve ser escrito antes de C. Por isso, TDD é conhecido também como *Test-First Development*.

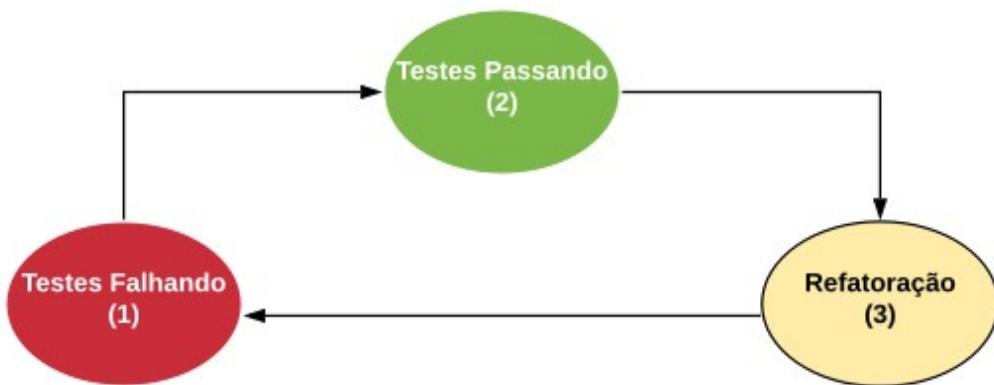
Quando se escreve o teste primeiro, ele vai falhar. Então, no fluxo de trabalho defendido por TDD, o próximo passo consiste em escrever o código que faz esse teste passar, mesmo que seja um código trivial. Em seguida, esse primeiro código deve ser finalizado e refinado. Por fim, se necessário,

ele deve ser refatorado, para melhorar seu projeto, legibilidade, manutenibilidade, para seguir princípios e padrões de projeto, etc.

TDD foi proposto com três objetivos principais em mente:

- TDD ajuda a evitar que os desenvolvedores esqueçam de escrever testes. Para isso, TDD promove testes à primeira atividade de qualquer tarefa de programação, seja ela corrigir um bug ou implementar uma nova funcionalidade. Sendo a primeira atividade, é mais difícil que a escrita de testes seja deixada para um segundo momento.
- TDD favorece a escrita de código com alta testabilidade. Essa característica é uma consequência natural da inversão do fluxo de trabalho proposta por TDD: como o desenvolvedor sabe que ele terá que escrever o teste T e depois a classe C, é natural que desde o início ele planeje C de forma a facilitar a escrita de seu teste. De fato, como mencionamos na Seção 8.4, sistemas que usam TDD têm alta cobertura de testes, normalmente acima de 90%.
- TDD é uma prática relacionada não apenas com testes, mas também com a melhoria do design de um sistema. Isso acontece porque o desenvolvedor, ao começar pela escrita de um teste T, coloca-se na posição de um usuário da classe C. Em outras palavras, com TDD, o primeiro usuário da classe é seu próprio desenvolvedor — lembre que T é um cliente de C, pois ele chama métodos de C. Por isso, espera-se que o desenvolvedor simplifique a interface de C, use nomes de identificadores legíveis, evite muitos parâmetros, etc.

Quando se trabalha com TDD, o desenvolvedor segue um ciclo composto por três estados, conforme mostra a próxima figura.



## Ciclos de TDD

De acordo com esse diagrama, a primeira meta é chegar no estado vermelho, quando o teste ainda não está passando. Pode parecer estranho, mas o estado vermelho já é uma pequena vitória: ao escrever um teste que falha, o desenvolvedor pelo menos tem em mãos uma especificação da classe que ele precisará implementar em seguida. Ou seja, ele já sabe o que tem que fazer. Conforme já mencionamos, nesse estado, é importante que o desenvolvedor pense também na interface da classe que ele terá que implementar, colocando-se na posição de um usuário da mesma. Por fim, é importante que ele entregue o código compilando. Para isso, ele deve escrever pelo menos o esqueleto da classe sob teste, isto é, a assinatura da classe e de seus métodos.

Em seguida, a meta é alcançar o estado verde. Para isso, deve-se implementar a funcionalidade completa da classe sob teste; quando isso ocorrer, os testes que estavam falhando vão começar a passar. No entanto, pode-se dividir essa implementação em pequenos passos. Talvez, nos passos iniciais, o código estará funcionando de forma parcial, por exemplo, retornando apenas constantes. Isso ficará mais claro no exemplo que daremos a seguir.

Por fim, deve-se analisar se existem oportunidades para refatorar o código da classe e do teste. Quando se usa TDD, o objetivo não é apenas alcançar o estado verde, no qual o programa está funcionando. Além disso, deve-se verificar a possibilidade de melhorar a qualidade do projeto do código. Por exemplo, verificar se não existe código duplicado, se não existem métodos muito longos que possam ser quebrados em métodos menores, se algum método pode ser movido para uma classe diferente, etc. Terminado o passo

de refatoração, podemos parar ou então reiniciar o ciclo, para implementar mais alguma funcionalidade.

## Exemplo: Carrinho de Compras

Para concluir, vamos ilustrar uma sessão de uso de TDD. Para isso, usaremos como exemplo o sistema de uma livraria virtual. Nesse sistema, temos uma classe Book, com atributos `titulo`, `isbn` e `preco`. E temos também a classe ShoppingCart, que armazena os livros que um cliente deseja comprar. Essa classe deve implementar métodos para: adicionar um livro no carrinho; retornar o preço total dos livros no carrinho; e remover um livro do carrinho. A seguir, mostramos a implementação desses métodos usando TDD.

**Estado Vermelho:** Começamos definindo que ShoppingCart terá um método `add` e um método `getTotal`. Além de decidir o nome de tais métodos, definimos os seus parâmetros e escrevemos o primeiro teste:

```
@Test
void testAddGetTotal() {
    Book b1 = new Book("book1", 10, "1");
    Book b2 = new Book("book2", 20, "2");
    ShoppingCart cart = new ShoppingCart();
    cart.add(b1);
    cart.add(b2);
    assertEquals(30.0, cart.getTotal());
}
```

Apesar de simples e de fácil entendimento, esse teste ainda não compila, pois não existe implementação para as classes Book e ShoppingCart. Então, temos que providenciar isso, como mostrado a seguir:

```
public class Book {
    public String title;
    public double price;
    public String isbn;

    public Book(String title, double price, String isbn) {
        this.title = title;
        this.price = price;
        this.isbn = isbn;
    }
}
```

```
}
```

```
public class ShoppingCart {
```

```
    public ShoppingCart() {}
```

```
    public void add(Book b) {}
```

```
    double getTotal() {
```

```
        return 0.0;
```

```
    }
```

```
}
```

A implementação de ambas as classes é muito simples. Implementamos apenas o mínimo para que o programa e o teste compilem. Observe, por exemplo, o método `getTotal` de `ShoppingCart`. Nessa implementação, ele sempre retorna 0.0. Apesar disso atingimos nosso objetivo: temos um teste compilando, executando e falhando! Ou seja, chegamos ao estado vermelho.

**Estado Verde:** o teste anterior funciona como uma especificação. Isto é, ele define o que temos que implementar em `ShoppingCart`. Logo, mãos à obra:

```
public class ShoppingCart {
```

```
    public ShoppingCart() {}
```

```
    public void add(Book b) {}
```

```
    double getTotal() {
```

```
        return 30.0;
```

```
    }
```

```
}
```

Porém, o leitor deve estar agora surpreso: essa implementação está incorreta! A construtora de `ShoppingCart` está vazia, a classe não possui nenhuma estrutura de dados para armazenar os itens do carrinho, `getTotal` retorna sempre 30.0, etc. Tudo isso é verdade, mas já temos uma nova pequena vitória: o teste mudou de cor, de vermelho para verde. Ou seja, ele está passando. Com TDD, os avanços são sempre pequenos. Em XP, esses avanços são chamados de **baby steps**.

Mas temos que prosseguir e dar uma implementação mais realista para `ShoppingCart`. Segue então ela:

```

public class ShoppingCart {

    private ArrayList<Book> items;

    private double total;

    public ShoppingCart() {
        items = new ArrayList<Book>();
        total = 0.0;
    }

    public void add(Book b) {
        items.add(b);
        total += b.price;
    }

    double getTotal() {
        return total;
    }

}

```

Agora dispomos de uma estrutura de dados para armazenar os itens do carrinho, um atributo para armazenar o valor total do carrinho, uma classe construtora, um método add que adiciona os livros na estrutura de dados e incrementa o total do carrinho e assim por diante. No melhor do nosso juízo, essa implementação já implementa o que foi pedido e, por isso, podemos declarar que chegamos ao estado verde.

**Estado Amarelo:** agora temos que olhar para o código que foi implementado — um teste e duas classes — e colocar em prática as propriedades, princípios e padrões de projeto que aprendemos em capítulos anteriores. Ou seja: existe alguma coisa que podemos fazer para tornar esse código mais legível, fácil de entender e de manter? No caso, a ideia que pode surgir é encapsular os campos de Book. Todos eles atualmente são públicos e, por isso, seria melhor implementar apenas métodos get e set para dar acesso a eles. Como essa implementação é simples, não vamos mostrar o código refatorado de Book.

Então, fechamos uma volta no ciclo vermelho-verde-refatorar de TDD. Agora, podemos parar ou então pensar em implementar mais um requisito.

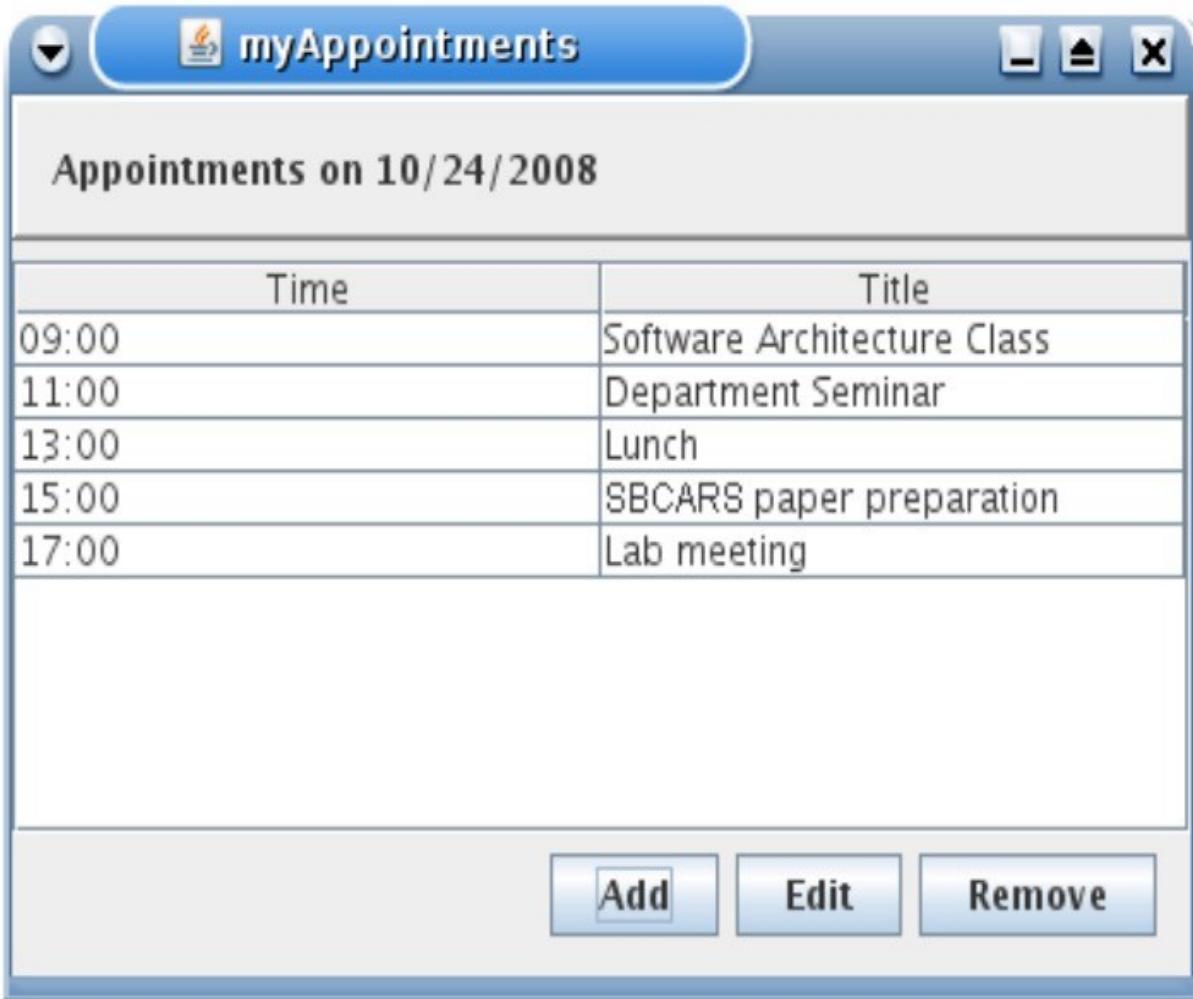
Por exemplo, podemos implementar um método para remover livros do carrinho. Para isso, temos que começar mais um ciclo.

## Testes de Integração

Com testes de integração — também chamados de **testes de serviços** — subimos para um nível intermediário da pirâmide de testes (veja uma figura com essa pirâmide na primeira seção do capítulo). Assim, o objetivo deixa de ser o teste de uma unidade pequena de código, como uma classe apenas. Em vez disso, testes de integração exercitam um serviço completo, isto é, uma funcionalidade de maior granularidade do sistema. Por isso, eles envolvem mais classes, às vezes de pacotes distintos. Também envolvem dependências e sistemas reais, como bancos de dados e serviços remotos. Além disso, quando se implementa testes de integração não faz mais sentido usar mocks ou stubs. Como são testes maiores, eles levam mais tempo para executar e, consequentemente, são chamados com menor frequência.

### Exemplo: Agenda de Compromissos

Suponha uma agenda simples que permita adicionar, remover e editar compromissos, conforme ilustrado na próxima figura.



## Interface da Agenda de Compromissos

Nesse sistema, existe uma classe com métodos para manipular a agenda, como mostrado a seguir:

```
public class AgendaFacade {  
    public AgendaFacade(DB db);  
    int addAppointment(Appointment p);  
    void removeAppointment(int id);  
    Appointment[] listAppointments();  
}
```

Assim, podemos escrever o seguinte teste de integração para essa classe:

```
@Test  
void AgendaFacadeTest() {  
    DB db = DB.create();
```

```

AgendaFacade agenda = new AgendaFacade(db);
Appointment app1 = new Appointment(...);
Appointment app2 = new Appointment(...);
Appointment app3 = new Appointment(...);
int id1 = agenda.addAppointment(app1);
int id2 = agenda.addAppointment(app2);
int id3 = agenda.addAppointment(app3);
Appointment [] apps = agenda.listAppointments();
assertEquals(3, apps.length);
}

```

Vale a pena mencionar dois pontos sobre esse teste. Primeiro, ele é implementado usando o JUnit, como os testes anteriores, de unidade, que estudamos neste capítulo. Ou seja, JUnit poder ser usado tanto para testes de unidade como para testes de integração. Segundo, ele é um teste de integração, pois a classe é testada com dependências reais, no caso para um banco de dados. No início do teste, cria-se um banco de dados com todas as tabelas vazias. Em seguida, três objetos são persistidos e depois lidos do banco de dados. Por fim, chama-se um assert. Assim, esse teste exercita os principais serviços da agenda, exceto aqueles relacionados com sua interface gráfica. Por isso, ele ainda não é um teste de sistema.

## Testes de Sistema

Testes de sistema estão posicionados no topo da pirâmide de testes. Trata-se de testes que simulam o uso de um sistema por um usuário real. Testes de sistema são também chamados de testes **ponta-a-ponta** (*end-to-end*) ou então **testes de interfaces**. São os testes mais caros, que demandam maior esforço para implementação e que executam em mais tempo.

### Exemplo: Teste de Sistemas Web

Selenium é um framework para automatizar testes de sistemas Web. O framework permite criar programas que funcionam como robôs que abrem páginas Web, preenchem formulários, clicam em botões, testam respostas, etc. Um exemplo — extraído e adaptado da documentação do Selenium ([link](#)) — é mostrado a seguir. Esse código simula um usuário de um navegador Firefox fazendo uma pesquisa no Google pela palavra software.

O código também imprime no console o título da página que lista os resultados da pesquisa.

```
public class SeleniumExample {  
  
    public static void main(String[] args) {  
        // cria um driver para acessar um servidor Web  
        WebDriver driver = new FirefoxDriver();  
  
        // instrui o driver para "navegar" pelo Google  
        driver.navigate().to("http://www.google.com");  
  
        // obtém um campo de entrada de dados, de nome "q"  
        WebElement element = driver.findElement(By.name("q"));  
  
        // preenche esse campo com as palavras "software"  
        element.sendKeys("software");  
  
        // submete os dados; como se fosse dado um "enter"  
        element.submit();  
  
        // espera a página de resposta carregar (timeout de 8s)  
        (new WebDriverWait(driver,8)).  
            until(new ExpectedCondition<Boolean>() {  
                public Boolean apply(WebDriver d) {  
                    return d.getTitle().toLowerCase().startsWith("software");  
                }  
            });  
  
        // resultado deve ser: "software - Google Search"  
        System.out.println("Page title is: "+driver.getTitle());  
  
        // fecha o navegador  
        driver.quit();  
    }  
}
```

Testes de interface são mais difíceis de escrever, pelo menos do que testes de unidade e mesmo do que testes de integração. Por exemplo, a API do Selenium é mais complexa do que aquela do JUnit. Além disso, o teste deve tratar eventos de interfaces, como timeouts que ocorrem quando uma página demora mais tempo do que o usual para ser carregada. Testes de interface também são mais frágeis, isto é, eles podem quebrar devido a pequenas mudanças na interface. Por exemplo, se o nome do campo de pesquisa da tela principal do Google mudar, o teste acima terá que ser atualizado. Porém,

se compararmos com a alternativa — realizar o teste manualmente — eles ainda são competitivos e apresentam ganhos.

## Exemplo: Teste de um Compilador

Quando se desenvolve um compilador, pode-se usar testes de unidade ou de integração. Já os testes de sistema de um compilador tendem a ser conceitualmente mais simples. O motivo é que a interface de um compilador não inclui janelas e telas com elementos gráficos. Em vez disso, um compilador recebe um arquivo de entrada e produz um arquivo de saída. Assim, o teste de sistema de um compilador C para uma linguagem X demanda a implementação de vários programas em X, exercitando diversos aspectos dessa linguagem. Para cada programa P, deve-se definir um conjunto de dados de entrada e um conjunto de dados de saída. Preferencialmente, essa saída deve ser em um formato simples, como uma lista de strings. Então, o teste de sistema do compilador ocorre da seguinte forma: chama-se C para compilar cada programa P; em seguida, executamos o resultado da compilação com a entrada definida anteriormente e verificamos se o resultado é o esperado. Esse teste é um teste de sistema, pois estamos exercitando todas as funcionalidades do compilador.

Quando comparados com testes de unidade, é mais difícil localizar o trecho de código responsável por uma falha em testes de sistema. Por exemplo, no caso do compilador, teremos a indicação de que um programa não está executando corretamente. Porém, normalmente não é trivial mapear essa falha para a unidade do compilador que gerou código de forma incorreta.

# Outros Tipos de Testes

## Testes Caixa-Preta e Caixa-Branca

Técnicas de teste podem ser classificadas como caixa-preta ou caixa-branca. Quando se usa uma **técnica caixa-preta**, os testes são escritos com base apenas na interface do sistema sob testes. Por exemplo, se a missão for testar um método como uma caixa-preta, a única informação disponível incluirá seu nome, parâmetros, tipos e exceções de retorno. Por outro lado, quando se usa uma **técnica caixa-branca**, a escrita dos testes considera informações

sobre o código e a estrutura do sistema sob teste. Por isso, técnicas de teste caixa-preta são também chamadas de **testes funcionais**. E técnicas caixa-branca são chamadas de **testes estruturais**.

No entanto, não é trivial classificar testes de unidade em uma dessas categorias. Na verdade, a classificação vai depender de como os testes são escritos. Se os testes de unidade forem escritos usando-se informações apenas sobre a interface dos métodos sob teste, eles são considerados como caixa-preta. Porém, se a escrita considerar informações sobre a cobertura dos testes, tais como desvios que são cobertos ou não, então eles são testes caixa-branca. Em resumo, testes de unidade sempre testam uma unidade pequena e isolada de código. Essa unidade pode ser testada na forma de uma caixa-preta (conhecendo-se apenas a sua interface e requisitos externos) ou na forma de uma caixa-branca (conhecendo-se e tirando-se proveito da sua estrutura interna, para elaboração de testes mais efetivos).

Uma observação semelhante pode ser feita sobre a relação entre TDD e testes caixa-preta/branca. Para esclarecer essa relação, vamos usar um comentário do próprio Kent Beck (fonte: *Test-Driven Development Violates the Dichotomies of Testing*, Three Rivers Institute, 2007):

No contexto de TDD, uma dicotomia incorreta ocorre entre testes caixa-preta e testes caixa-branca. Como testes em TDD são escritos antes do código que eles testam, eles talvez pudessem ser considerados como testes caixa-preta. No entanto, eu normalmente ganho inspiração para escrever o próximo teste depois que implemento e analiso o código verificado pelo teste anterior, o que é uma característica marcante de testes caixa-branca.

## Seleção de Dados de Teste

Quando se adota testes caixa-preta, existem técnicas para auxiliar na seleção das entradas que serão verificadas no teste. Partição via **Classe de Equivalência** é uma técnica que recomenda dividir as entradas de um problema em conjuntos de valores que têm a mesma chance de apresentar um bug. Esses conjuntos são chamados de classes de equivalência. Para cada classe de equivalência, recomenda-se testar apenas um dos seus valores, que pode ser escolhido randomicamente. Suponha uma função para calcular o

valor a pagar de imposto de renda, para cada faixa de salário, conforme tabela a seguir. Particionamento via classe de equivalência recomendaria testar essa função com quatro salários, um de cada faixa salarial.

Salário	Alíquota	Parcela a Deduzir
De 1.903,99 até 2.826,65	7,5%	142,80
De 2.826,66 até 3.751,05	15%	354,80
De 3.751,06 até 4.664,68	22,5%	636,13
Acima de 4.664,68	27,5%	869,36

**Análise de Valor Limite** (*Boundary Value Analysis*) é uma técnica complementar que recomenda testar uma unidade com os valores limites de cada classe de equivalência e seus valores subsequentes (ou antecedentes). O motivo é que bugs com frequência são causados por um tratamento inadequado desses valores de fronteira. Assim, no nosso exemplo, para a primeira faixa salarial, deveríamos testar com os seguintes valores:

- 1.903,98: valor imediatamente inferior ao limite inferior da primeira faixa salarial
- 1.903,99: limite inferior da primeira faixa salarial
- 2.826,65: limite superior da primeira faixa salarial
- 2.826,66: valor imediatamente superior ao limite superior da primeira faixa salarial

No entanto, como o leitor deve estar pensando, nem sempre é trivial encontrar as classes de equivalência para o domínio de entrada de uma função. Isto é, nem sempre todos os requisitos de um sistema são organizados em faixas de valores bem definidas como aquelas de nosso exemplo.

Para concluir, gostaríamos de lembrar que **testes exaustivos**, isto é, testar um programa com todas as entradas possíveis, na prática, é impossível, mesmo em programas pequenos. Por exemplo, imagine um compilador de uma linguagem X. É impossível testar esse compilador com todos os programas que podem ser implementados em X, até porque o número deles é infinito. Na verdade, mesmo uma função com apenas dois inteiros como

parâmetro pode levar séculos para ser testada exaustivamente com todos os possíveis pares de inteiros. **Testes randômicos**, quando os dados de teste são escolhidos aleatoriamente, também não são recomendados, na maioria dos casos. O motivo é que pode-se selecionar diferentes valores de uma mesma classe de equivalência, o que não é necessário. Por outro lado, algumas classes de equivalência podem ficar sem testes.

## Testes de Aceitação

São testes realizados pelo cliente, com dados do cliente. Os resultados desses testes irão determinar se o cliente está de acordo ou não com a implementação realizada. Se estiver de acordo, o sistema pode entrar em produção. Se não estiver de acordo, os devidos ajustes devem ser realizados. Por exemplo, quando se usa métodos ágeis, uma história somente é considerada completa após passar por testes de aceitação, realizados pelos usuários, ao final de um sprint, conforme estudamos no Capítulo 2.

Testes de aceitação possuem duas características que os distinguem de todos os testes que estudamos antes neste capítulo. Primeiro, são **testes manuais**, realizados pelos clientes finais do sistema. Segundo, eles não constituem exclusivamente uma atividade de verificação (como os testes anteriores), mas também uma atividade de validação do sistema. Lembre-se do capítulo de Introdução: verificação testa se fizemos o sistema corretamente, isto é, de acordo com a sua especificação e/ou requisitos. Já validação testa se fizemos o sistema correto, isto é, aquele que o cliente pediu e precisa.

Testes de aceitação podem ser divididos em duas fases. **Testes alfa** são realizados com alguns usuários, mas em um ambiente controlado, como a própria máquina do desenvolvedor. Se o sistema for aprovado nos testes alfa, pode-se realizar um teste com um grupo maior de usuários e não mais em um ambiente controlado. Esses testes são chamados de **testes beta**.

## Testes de Requisitos Não-Funcionais

Os testes anteriores, com exceção dos testes de aceitação, verificam apenas requisitos funcionais; logo, eles têm como objetivo encontrar bugs. Porém, é possível realizar também testes para verificar ou validar requisitos não-funcionais. Por exemplo, existem ferramentas que permitem a realização de

**testes de desempenho**, para verificar o comportamento de um sistema com alguma carga. Uma empresa de comércio eletrônico pode usar uma dessas ferramentas para simular o desempenho de seu site durante um grande evento, como uma Black-Friday, por exemplo. Já **testes de usabilidade** são usados para avaliar a interface do sistema e, normalmente, envolvem a observação de usuários reais usando o sistema. **Testes de falhas** simulam eventos anormais em um sistema, por exemplo a queda de alguns serviços ou mesmo de um data-center inteiro.

## Bibliografia

Gerard Meszaros. xUnit Test Patterns: Refactoring Test Code. Addison-Wesley, 2007.

Kent Beck, Erich Gamma. Test-infected: programmers love writing tests. Java Report, 3(7):37-50, 1998.

Kent Beck. Test-Driven Development: by Example, Addison-Wesley, 2002.

Dave Thomas and Andy Hunt. Mock Objects. IEEE Software, 2002

Maurício Aniche. Testes automatizados de software: um guia prático. Casa do Código, 2015.

Jeff Langr, Andy Hunt, Dave Thomas. Pragmatic Unit Testing in Java 8 with Junit. O'Reilly, 2015.

## Exercícios de Fixação

1. (ENADE 2011) Uma equipe está realizando testes com o código-fonte de um sistema. Os testes envolvem a verificação de diversos componentes individualmente, bem como das interfaces entre eles. Essa equipe está realizando testes de:

1. unidade
2. aceitação
3. sistema e aceitação

4. integração e sistema
5. unidade e integração

2. Descreva três benefícios associados ao uso de testes de unidade.
3. Suponha uma função `fib(n)`, que retorna o  $n$ -ésimo termo da sequência de Fibonacci, isto é,  $\text{fib}(0) = 0$ ,  $\text{fib}(1) = 1$ ,  $\text{fib}(2) = 1$ ,  $\text{fib}(3) = 2$ ,  $\text{fib}(4) = 3$ , etc. Escreva um teste de unidade para essa função.
4. Reescreva o seguinte teste, que verifica o levantamento de uma exceção `EmptyStackException`, para que ele fique mais simples e fácil de entender.

```
@Test
public void testEmptyStackException() {
    boolean sucesso = false;
    try {
        Stack<Integer> s = new Stack<Integer>();
        s.push(10);
        int r = stack.pop();
        r = stack.pop();
    } catch (EmptyStackException e) {
        sucesso = true;
    }
    assertTrue(sucesso);
}
```

5. Suponha que um programador escreveu o teste a seguir para a classe `ArrayList` de Java. Como você irá perceber, no código são usados diversos `System.out.println`. Ou seja, no fundo, ele é um teste manual, pois o desenvolvedor tem que conferir o seu resultado manualmente. Reescreva então cada um dos testes (de 1 a 6) como um teste de unidade, usando a sintaxe e os comandos do JUnit. Observação: se quiser executar o código, ele está disponível neste [link](#).

```
import java.util.List;
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        // teste 1
        List<Integer> s = new ArrayList<Integer>();
```

```
System.out.println(s.isEmpty());  
  
// teste 2  
s = new ArrayList<Integer>();  
s.add(1);  
System.out.println(s.isEmpty());  
  
// teste 3  
s = new ArrayList<Integer>();  
s.add(1);  
s.add(2);  
s.add(3);  
System.out.println(s.size());  
System.out.println(s.get(0));  
System.out.println(s.get(1));  
System.out.println(s.get(2));  
  
// teste 4  
s = new ArrayList<Integer>();  
s.add(1);  
s.add(2);  
s.add(3);  
int elem = s.remove(2);  
System.out.println(elem);  
System.out.println(s.get(0));  
System.out.println(s.get(1));  
  
// teste 5  
s = new ArrayList<Integer>();  
s.add(1);  
s.remove(0);  
System.out.println(s.size());  
System.out.println(s.isEmpty());  
  
// teste 6  
try {  
    s = new ArrayList<Integer>();  
    s.add(1);  
    s.add(2);  
    s.remove(2);  
}  
  
catch (IndexOutOfBoundsException e) {  
    System.out.println("IndexOutOfBounds");  
}  
}
```

}

6. Seja a seguinte função. Observe que ela possui quatro comandos, sendo dois deles `if`. Logo, esses dois `ifs` geram quatro branches:

```
void f(int x, int y) {  
    if (x > 0) {  
        x = 2 * x;  
        if (y > 0) {  
            y = 2 * y;  
        }  
    }  
}
```

Supondo o código acima, preencha a próxima tabela, com os valores da cobertura de comandos e cobertura de branches obtidos com os testes especificados na primeira coluna (ou seja, a primeira coluna define as chamadas da função `f` que o teste realiza).

Chamada feita pelo teste	Cobertura de comandos	Cobertura de branches
<code>f(0,0)</code>	0	0
<code>f(1,1)</code>	100%	100%
<code>f(0,0) e f(1,1)</code>	100%	100%

7. Suponha o seguinte requisito: alunos recebem conceito A em uma disciplina se tiverem nota maior ou igual a 90. Seja então a seguinte função que implementa esse requisito:

```
boolean isConceitoA(int nota) {  
    if (nota > 90)  
        return true;  
    else return false;  
}
```

O código dessa função possui três comandos, sendo um deles um `if`; logo, ela possui dois branches.

Responda agora às seguintes perguntas.

1. A implementação dessa função possui um bug? Se sim, quando esse bug resulta em falha?
  2. Suponha que essa função — exatamente como ela está implementada — seja testada com duas notas: 85 e 95. Qual a cobertura de comandos desse teste? E a cobertura de branches?
  3. Seja a seguinte afirmação: se um programa possui 100% de cobertura de testes, em nível de comandos, ele está livre de bugs. Ela é verdadeira ou falsa? Justifique.
8. Complete os comandos assert nos trechos indicados.

```
public void test1() {  
    LinkedList list = mock(LinkedList.class);  
    when(list.size()).thenReturn(10);  
    assertEquals(_____, _____);  
}  
  
public void test2() {  
    LinkedList list = mock(LinkedList.class);  
    when(list.get(0)).thenReturn("Engenharia");  
    when(list.get(1)).thenReturn("Software");  
    String result = list.get(0) + " " + list.get(1);  
    assertEquals(_____, _____);  
}
```

9. Suponha duas unidades de código A e B, sendo que A depende de B. Para permitir o teste de unidade de A foi criado um mock para B, chamado B'. O teste de unidade de A está passando. Porém, ao fazer o teste de integração com A e B, ele falha. Descreva um cenário mais real, no qual A, B, e B' sejam classes reais, com métodos reais, que realizam funções reais, etc. O cenário proposto deve incluir um bug associado ao mock B'. Ou seja, B' esconde um bug, que só vai aparecer no teste de integração. Dizendo de outra maneira, B' não simula precisamente o comportamento de B; quando B' é removido, no teste de integração, surge então um bug.

# Cap 9 Refactoring

Este capítulo inicia com uma introdução a refactorings, os quais são modificações realizadas em um programa para facilitar o seu entendimento e futura evolução. Na Seção 9.2, apresentamos uma série de operações de refactoring, incluindo exemplos de código, alguns deles de refactorings reais, realizados em sistemas de código aberto. Em seguida, na Seção 9.3, discutimos alguns aspectos sobre a prática de refactoring, incluindo a importância de uma boa suíte de testes de unidade. A Seção 9.4 apresenta os recursos oferecidos por IDEs para realização automatizada de refactorings. Para finalizar, a Seção 9.5 descreve uma lista de code smells, isto é, indicadores de que uma determinada estrutura de código não está cheirando bem e que, portanto, poderia ser objeto de uma refatoração.

## Introdução

No capítulo anterior, vimos que software precisa ser testado, como qualquer produto de engenharia. A mesma recomendação vale para atividades de manutenção. Isto é, software também precisa de manutenção. Na verdade, na Introdução deste livro, já comentamos que existem diversos tipos de manutenção que podem ser realizadas em sistemas de software. Quando um bug é detectado, temos que realizar uma **manutenção corretiva**. Quando os usuários ou o dono do produto solicitam uma nova funcionalidade, temos que realizar uma **manutenção evolutiva**. Quando uma regra de negócio ou alguma tecnologia usada pelo sistema muda, temos que reservar tempo para uma **manutenção adaptativa**.

Além disso, sistemas de software também envelhecem, como ocorre com os seres vivos. Ainda no início da década de 1970, Meir Lehman — então trabalhando na IBM — começou a observar e analisar esse fenômeno e, como resultado, enunciou um conjunto de leis empíricas sobre envelhecimento, qualidade interna e evolução de sistemas de software, que ficaram conhecidas **Leis da Evolução de Software** ou simplesmente **Leis de Lehman**. As duas primeiras leis enunciadas por Lehman foram as seguintes:

- Um sistema de software deve ser continuamente mantido para se adaptar ao seu ambiente. Esse processo deve continuar até o ponto em que se torna mais vantajoso substituí-lo por um sistema completamente novo.
- À medida que um sistema sofre manutenções, sua complexidade interna aumenta e a qualidade de sua estrutura deteriora-se, a não ser que um trabalho seja realizado para estabilizar ou evitar tal fenômeno.

A primeira lei justifica a necessidade de manutenções adaptativas e também evolutivas em sistemas de software. Ela também menciona que sistemas podem morrer, isto é, pode chegar a um ponto em que vale mais a pena descontinuar o desenvolvimento de um sistema e substituí-lo por um novo. Já a segunda Lei de Lehman afirma que manutenções tornam o código e a estrutura interna de um sistema mais complexos e difíceis de manter no futuro. Em outras palavras, existe uma deterioração natural da qualidade interna de um sistema, à medida que ele passa por manutenções e evoluções. No entanto, a segunda lei faz uma ressalva: um certo trabalho pode ser realizado para estabilizar ou mesmo evitar esse declínio natural da qualidade interna de sistemas de software. Modernamente, esse trabalho é chamado de **refactoring**.

Refactorings são transformações de código que melhoram a manutenibilidade de um sistema, mas sem afetar o seu funcionamento. Para explicar essa definição, vamos dividi-la em três partes. Primeiro, quando a definição menciona transformações de código, ela está se referindo a modificações no código, como dividir uma função em duas, renomear uma variável, mover uma função para outra classe, extrair uma interface de uma classe, etc. Em seguida, a definição menciona o objetivo de tais transformações: melhorar a manutenibilidade do sistema, isto é, melhorar sua modularidade, melhorar seu projeto ou arquitetura, melhorar sua testabilidade, tornar o código mais legível, mais fácil de entender e modificar, etc. Por fim, coloca-se uma restrição: não adianta melhorar a manutenibilidade do sistema e prejudicar o seu funcionamento. Ou seja, refactoring deve entregar o sistema funcionando exatamente como antes das transformações. Uma outra maneira de dizer isso é afirmando que refactorings devem preservar o comportamento ou a semântica do sistema.

No entanto, nas décadas de 70 e 80, quando as Leis de Lehman foram formuladas, o termo refactoring ainda não era usado. Um dos primeiros usos do termo em Engenharia de Software ocorreu em 1992, em uma tese de doutorado defendida por William Opdyke, na Universidade de Illinois, EUA ([link](#)). Em seguida, em 1999, refactoring — já com esse nome — foi incluído entre as práticas de programação preconizadas por Extreme Programming. Na primeira edição do livro de XP, recomenda-se que desenvolvedores devem realizar refactorings com o objetivo de reestruturar seus sistemas, sem mudar o comportamento deles e sim para remover duplicação de código, melhorar a comunicação com outros desenvolvedores, simplificar o código ou torná-lo mais flexível.

Em 2000, Martin Fowler lançou a primeira edição de um livro dedicado exclusivamente a refactoring, que alcançou grande sucesso e contribuiu para popularizar essa prática de programação. Um dos motivos desse sucesso foi o fato de o livro incluir um catálogo com dezenas de refactorings. De uma forma que lembra um catálogo de padrões de projeto (tal como estudamos no Capítulo 6), a apresentação dos refactorings começa dando um nome para eles, o que contribuiu para criar um vocabulário sobre refactoring. No livro, Fowler também apresenta a mecânica de funcionamento de cada refactoring, inclui exemplos de código e discute os benefícios e desvantagens dos refactorings.

Em seu livro, Fowler cita também a frase de Kent Beck que abre esse capítulo e que ressalta a importância de seguir bons hábitos de programação, os quais são fundamentais para preservar a saúde de um sistema, garantindo que ele continuará evoluindo por anos. Portanto, desenvolvedores não devem realizar apenas manutenções corretivas, adaptativas e evolutivas. É importante cuidar também da manutenibilidade do sistema, por meio da realização frequente de refactorings.

**Tradução:** Decidimos não traduzir refactoring quando usado como substantivo. O motivo é que achamos que o substantivo em inglês já faz parte do vocabulário dos desenvolvedores de software brasileiros. Porém, quando usada como verbo (*to refactor*), traduzimos para refatorar.

**Aviso:** O termo refactoring tornou-se comum em desenvolvimento de software. Por isso, às vezes ele é usado para indicar a melhoria de outros

requisitos não-funcionais, que não estão relacionados com manutenibilidade. Por exemplo, frequentemente, ouvimos desenvolvedores mencionar que vão refatorar o código para melhorar seu desempenho, para introduzir concorrência, para melhorar a usabilidade de sua interface, etc. No entanto, neste livro, vamos usar o termo restrito à sua definição original, isto é, apenas para denotar modificações de código que melhoram a sua manutenibilidade.

## Catálogo de Refactorings

Nesta seção, vamos apresentar os principais refactorings do catálogo de Fowler. Assim como adotado nesse catálogo, vamos comentar sobre os seguintes tópicos na apresentação de cada refactoring: nome, motivação, mecânica de aplicação e exemplos de uso. Além disso, vamos usar alguns exemplos de refactorings reais, realizados por desenvolvedores de sistemas de código aberto disponíveis no GitHub.

### Inline de Método

Esse refactoring funciona no sentido contrário a uma extração de método. Suponha um método pequeno, com uma ou duas linhas de código, e que seja chamado poucas vezes. O benefício proporcionado por esse método — em termos de reúso e incremento de legibilidade do código — é pequeno. Portanto, ele pode ser removido do sistema e seu corpo incorporado nos pontos de chamada. No entanto, é importante ressaltar que Inline de Métodos é uma operação mais rara e menos importante do que Extração.

**Exemplo:** A seguir, mostramos um exemplo de Inline de Método, realizado no sistema IntelliJ, uma IDE para Java. Primeiro, segue o código antes do inline. Podemos ver que o método `writeContentToFile` tem uma única linha de código e é chamado apenas uma vez, pelo método `write`.

```
private void writeContentToFile(final byte[] revision) {  
    getVirtualFile().setBinaryContent(revision);  
}  
  
private void write(byte[] revision) {  
    VirtualFile virtualFile = getVirtualFile();  
    ...
```

```
    if (document == null) {
        writeContentToFile(revision); // única chamada
    }
    ...
}
```

Os desenvolvedores do IntelliJ decidiram então remover `writeContentToFile` e expandir o seu corpo no único ponto de chamada. O código após o refactoring é mostrado a seguir.

```
private void write(byte[] revision) {
    VirtualFile virtualFile = getVirtualFile();
    ...
    if (document == null) {
        virtualFile.setBinaryContent(revision); // após inline
    }
    ...
}
```

## Movimentação de Método

Não é raro encontrar um método implementado na classe errada. Ou seja, apesar de implementado em uma classe A, um método f pode usar mais serviços de uma classe B. Por exemplo, ele pode ter mais dependências para elementos de B do que de sua classe A. Nesses casos, deve-se avaliar a possibilidade de mover f para a classe B. Esse refactoring pode melhorar a coesão da classe A, diminuir o acoplamento entre A e B e, em última instância, tornar ambas as classes mais fáceis de serem entendidas e modificadas.

Pelas suas características, Movimentação de Métodos é um dos refactorings com maior potencial para melhorar a modularização de um sistema. Como sua atuação não está restrita a uma única classe, Movimentação de Métodos pode ter um impacto positivo na arquitetura do sistema, garantindo que os métodos estejam nas classes apropriadas, tanto do ponto de vista funcional como arquitetural.

**Exemplo:** O sistema IntelliJ possui um pacote para execução de testes de unidade, via IDE. Esse pacote, por sua vez, possui uma classe com métodos de uso geral, chamada `PlatformTestUtil`. Conforme mostra o código a seguir, essa classe tinha um método chamado `averageAmongMedians`, que

calcula a média das medianas de uma parte de um vetor de inteiros. No entanto, esse método não possui relação com a execução de testes de unidade. Por exemplo, ele não usa métodos e atributos de `PlatformTestUtil`. Ou seja, ele é independente do resto da classe.

```
class PlatformTestUtil {  
    ...  
    public static long averageAmongMedians(long[] time,  
                                            int part) {  
        int n = time.length;  
        Arrays.sort(time);  
        long total = 0;  
        for (int i= n/2-n/part/2; i< n/2+n/part/2; i++) {  
            total += time[i];  
        }  
        int middlePartLength = n/part;  
        return middlePartLength == 0 ? 0:total/middlePartLength;  
    }  
    ...  
}
```

Pelos motivos expostos no parágrafo anterior, um dos desenvolvedores do IntelliJ decidiu mover `averageAmongMedians` para uma classe chamada `ArrayUtil`, que tem como objetivo disponibilizar funções utilitárias para manipulação de vetores. Logo, trata-se de uma classe mais relacionada com a funcionalidade provida por `averageAmongMedians`.

Após a movimentação para a nova classe, as chamadas de `averageAmongMedians` tiveram que ser atualizadas, como mostrado na figura a seguir.

```
- System.out.println("Average among the N/3 median times: " + PlatformTestUtil.averageAmongMedians(time, 3) + "ms");
+ System.out.println("Average among the N/3 median times: " + ArrayUtil.averageAmongMedians(time, 3) + "ms");
```

```
//System.out.println("JobLauncher.COUNT = " + JobLauncher.COUNT);
//System.out.println("JobLauncher.TINY = " + JobLauncher.TINY_COUNT);
```

```
@@ -205,7 +205,7 @@ public int compare(HighlightInfo o1, HighlightInfo o2) {
```

```
}
```

```
FileEditorManagerEx.getInstanceEx(getProject()).closeAllFiles();
```

```
- System.out.println("Average among the N/3 median times: " + PlatformTestUtil.averageAmongMedians(time, 3) + "ms");
+ System.out.println("Average among the N/3 median times: " + ArrayUtil.averageAmongMedians(time, 3) + "ms");
```

Atualizando chamadas de método estático após movimentação.  
averageAmongMedians foi movido de PlatformTestUtil para ArrayUtil.

No entanto, como o leitor pode observar nesse diff, isso não foi difícil, pois averageAmongMedians é um método estático. Logo, apenas o nome de sua classe teve que ser atualizado em cada ponto de chamada.

Em outros casos, no entanto, pode não ser tão simples atualizar as chamadas de um método após ele ser movido para uma nova classe. Isso acontece quando nos pontos de chamada não existem referências para objetos da nova classe do método. Uma solução consiste então em deixar uma implementação simples do método na classe de origem. Essa implementação apenas delega as chamadas para a nova classe do método. Por isso, nenhum cliente precisará ser alterado. Um exemplo é mostrado a seguir. Primeiro, o código original:

```

class A {
    B b = new B();
    void f { ... }
}

class B { ... }

class Cliente {
    A a = new A();
    void g() {
        a.f();
        ...
    }
}

```

E agora o código após a refatoração:

```

class A {
    B b = new B();
    void f {
        b.f(); // apenas delega chamada para B
    }
}

class B { // f foi movido de A para B
    void f { ... }
}

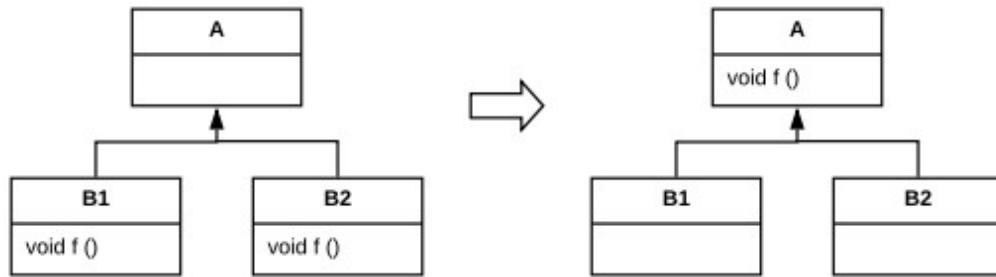
class Cliente {
    A a = new A();
    void g() {
        a.f(); // não precisa mudar
        ...
    }
}

```

Observe que o método `f` foi movido da classe `A` para a classe `B`. Porém, em `A` ficou uma versão do método que apenas repassa (ou delega) a chamada para `B`. Por isso, o código da classe `Cliente` não precisou ser alterado.

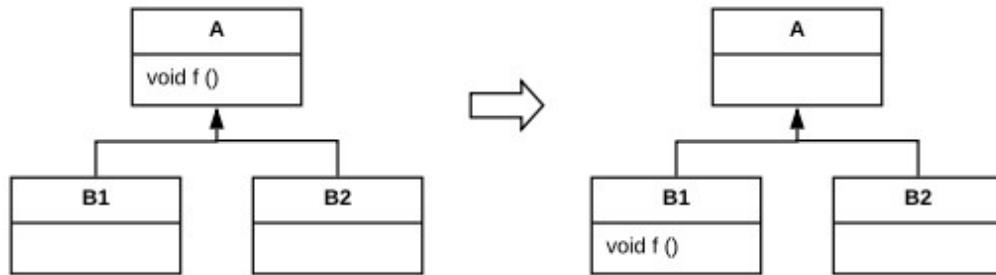
Quando ocorre em uma mesma hierarquia de classes, Movimentação de Métodos ganha nomes especiais. Por exemplo, quando o refactoring move um método de subclasses para uma superclasse, ele é chamado de **Pull Up Method**. Para ilustrar, suponha um mesmo método `f` implementado em duas subclasses `B1` e `B2`. Para evitar **duplicação de código**, pode-se então subir

com ambas implementações para a superclasse A, como mostra o diagrama de classes da próxima página.



### Pull Up Method

Por outro lado, quando um método é movido para baixo na hierarquia de classes, isto é, de uma superclasse para uma subclasse, dizemos que foi realizado um **Push Down Method**. Por exemplo, apesar de implementado na superclasse A, um método f pode ser do interesse de uma única subclasse, digamos que B1. Logo, podemos descer com sua implementação para B1, como mostrado no diagrama de classes da próxima página.



### Push Down Method

Para concluir, operações de refactoring podem ser feitas em sequência. Por exemplo, suponha a seguinte classe A com um método f:

```

class A {
    B b = new B();

    void f(){
        S1;
        S2;
    }
}
  
```

```
}
```

```
class B { ... }
```

Primeiro, vamos extrair um método g com o comando s2 de f:

```
class A {
    B b = new B();

    void g() { // novo método extraído de f()
        S2;
    }

    void f(){
        S1;
        g();
    }
}

class B { ... }
```

Na sequência, vamos mover g para uma classe B, como ilustrado a seguir:

```
class A {
    B b = new B();

    void f(){
        S1;
        b.g();
    }
}

class B {

    void g() { // movido de A para B
        S2;
    }
}
```

## Renomeação

Existe uma frase provocativa, atribuída a Phil Karlton, que afirma que existem apenas duas coisas difíceis em Ciência da Computação: invalidação

de cache e dar nomes às coisas. Como dar nomes é difícil, frequentemente temos que renomear um elemento de código, seja ele uma variável, função, método, parâmetro, atributo, classe, etc. Isso pode ocorrer porque o nome dado ao elemento não foi uma boa escolha. Um outro motivo é que a responsabilidade desse elemento pode ter mudado com o tempo e assim seu nome ficou desatualizado. Em ambos os casos, recomenda-se realizar um dos refactorings mais populares que existe: **renomeação**. Isto é, dar um nome mais adequado e significativo para o elemento de código.

Quando esse refactoring é aplicado, a parte mais complexa não é renomear o elemento, mas atualizar os pontos do código em que ele é referenciado. Por exemplo, se um método `f` é renomeado para `g`, todas as chamadas de `f` devem ser atualizadas. Na verdade, se `f` for muito usado, pode ser interessante extraí-lo para um novo método, com o novo nome, e manter o nome antigo, mas depreciado.

Para mostrar um exemplo, suponha o seguinte método `f`:

```
void f () {  
    // A  
}
```

Segue agora o código após o refactoring. Veja que extraímos um método `g`, com o código antigo de `f`, que foi então depreciado.

```
void g() {    // novo nome do método  
    // A  
}  
  
@deprecated  
void f() {    // mantém nome antigo  
    g();        // mas delega chamada para novo nome  
}
```

Depreciação é um mecanismo oferecido por linguagens de programação para indicar que um elemento de código está desatualizado e, portanto, desencorajar seu uso. Quando o compilador descobre que um trecho de código está usando um elemento depreciado, ele emite um *warning*. No exemplo anterior, o método `f` não foi simplesmente renomeado para `g`. Em vez disso, primeiro criou-se um método `g` com o código original de `f`. Em seguida, `f` foi depreciado e seu código modificado para apenas chamar `g`.

A estratégia baseada em depreciação torna a renomeação mais segura, pois ela não obriga a atualização de uma só vez de todas as chamadas de `f`, que podem ser muitas e espalhadas em diversos programas. Ou seja, ela dá tempo para que os clientes se adaptem à mudança e passem a usar o novo nome. Na verdade, refactorings — mesmo os mais simples, como uma renomeação — devem ser realizado em pequenos passos, ou em **baby steps**, para ter certeza que eles não vão prejudicar o correto funcionamento de um sistema.

## Outros Refactorings

Os refactorings apresentados anteriormente têm maior potencial para melhorar o projeto de um sistema, pois eles envolvem operações com um escopo global, como Movimentação de Métodos ou Extração de Classes. Porém, existem refactorings com escopo local, que melhoram, por exemplo, a implementação interna de um único método. Vamos a seguir descrever resumidamente alguns desses refactorings.

**Extração de Variáveis** é usado para simplificar expressões e torná-las mais fáceis de ler e entender. Seja o seguinte código de exemplo:

```
x1 = (-b + sqrt(b*b-4*a*c)) / (2*a);
```

Esse código pode ser refatorado para:

```
delta = b*b-4*a*c; // variável extraída
x1 = (-b + sqrt(delta)) / (2*a);
```

Veja que uma variável `delta` foi criada e inicializada com uma parte de uma expressão maior. Com isso, o código da expressão, após a refatoração, ficou menor e mais fácil de entender.

**Remoção de Flags** é um refactoring que sugere usar comandos como `break` ou `return`, em vez de variáveis de controle, também chamadas de flags. Seja o seguinte código de exemplo:

```
boolean search(int x, int[]a) {
    boolean achou = false;
    i = 0;
    while (i < a.length) && (!achou) {
```

```

        if (a[i] == x);
            achou = true;
        i++;
    }
    return achou;
}

```

Esse código pode ser refatorado da seguinte forma:

```

boolean search(int x, int[]a) {
    for (i = 0; i < a.length; i++)
        if (a[i] == x)
            return true;
    return false;
}

```

Observe que a função após a refatoração ficou menor e com uma lógica mais clara, graças ao uso de um comando `return` para retornar imediatamente assim que um determinado valor tenha sido encontrado em um vetor.

Existe também refactorings que tratam da simplificação de comandos condicionais. Um deles é chamado de **Substituição de Condicional por Polimorfismo**. Para entendê-lo, suponha um comando `switch` que retorna o valor da bolsa de pesquisa de um aluno, dependendo do seu tipo:

```

switch (aluno.type) {
    case "graduacao":
        bolsa = 500;
        break;
    case "mestrado":
        bolsa = 1500;
        break;
    case "doutorado":
        bolsa = 2500;
        break;
}

```

Em uma linguagem orientada a objetos, esse comando pode ser refatorado para apenas uma linha de código:

```
bolsa = aluno.getBolsa();
```

Na versão refatorada, o atributo `type` de `Aluno` não é mais necessário e, portanto, pode ser removido. Além disso, temos que implementar nas

subclasses de Aluno — por exemplo, AlunoGraduacao, AlunoMestrado e AlunoDoutorado — um método getBolsa(). Por fim, na superclasse Aluno esse método deve ser abstrato, isto é, ter apenas uma assinatura, sem corpo.

**Remoção de Código Morto** recomenda deletar métodos, classes, variáveis ou atributos que não estão sendo mais usados. Por exemplo, no caso de um método, pode não existir mais chamadas para ele. No caso de uma classe, ela pode não ser mais instanciada ou herdada por outras classes. No caso de um atributo, ele pode não ser usado no corpo da classe, nem em subclasses ou em outras classes. Pode parecer que Remoção de Código Morto é um refactoring raro, mas em grandes sistemas, desenvolvidos ao longo de anos por programadores diferentes, costuma existir uma quantidade considerável de código que não é mais chamado.

## Prática de Refactoring

Tendo apresentado diversos refactorings na seção anterior, vamos agora discutir como a prática de refactoring pode ser adotada em projetos reais de desenvolvimento de software.

Primeiro, a realização bem sucedida de refactorings depende da existência de bons testes, principalmente testes de unidade. Ou seja, sem testes fica arriscado realizar mudanças em um sistema, ainda mais quando elas não agregam novas funcionalidades ou corrigem bugs, como é o caso de refactorings. John Ousterhout tem o seguinte comentário sobre a importância de testes durante atividades de refactoring ([link](#)):

Testes, particularmente testes de unidade, desempenham um papel importante no projeto de software porque eles facilitam a realização de refactorings. Sem uma suíte de testes, torna-se arriscado realizar mudanças estruturais em um sistema. Como não há uma maneira fácil de encontrar bugs, é mais provável que eles fiquem escondidos até que o código entre em produção, quando é mais caro detectá-los e corrigi-los. Como resultado, desenvolvedores tendem a evitar refatorações em sistemas sem boas suítes de teste. Em vez disso, eles reduzem as modificações no código àquelas necessárias para implementar novas

funcionalidades ou corrigir bugs. Consequentemente, a complexidade vai se acumulando e erros de projeto não são corrigidos.

Uma segunda questão importante diz respeito ao momento em que o código deve ser refatorado. Existem dois modos principais de realizar refactorings: de forma oportunista ou de forma estratégica.

**Refactorings oportunistas** são realizados no meio de uma tarefa de programação, quando se descobre que um trecho de código não está bem implementado e que, portanto, pode ser melhorado. Isso pode acontecer quando se está corrigindo um bug ou implementando uma nova funcionalidade. Por exemplo, no meio dessas tarefas, pode-se perceber que o nome de um método não está claro, que um método está muito grande e difícil de entender, que um comando condicional está muito complexo, que um determinado código não é mais usado, etc. Assim, se o desenvolvedor descobrir problemas na implementação de um trecho de código, ele deve refatorá-lo imediatamente. Tentando explicar de um modo mais claro, suponha que um desenvolvedor trabalhe por uma hora na implementação de uma nova funcionalidade. É compreensível e desejável que parte desse tempo — talvez 20% ou mais — seja investido em refactorings. Kent Beck tem uma frase interessante sobre refactorings oportunistas:

Para cada mudança que você tiver que realizar em um sistema, primeiro torne essa mudança fácil (aviso: isso pode ser difícil), então realize a mudança facilmente.

A ideia de fundo dessa recomendação é que um desenvolvedor pode estar enfrentando dificuldades para implementar uma mudança exatamente porque o código não está preparado para acomodá-la. Assim, primeiro ele deve dar um passo atrás, isto é, refatorar o código de forma oportunista, para tornar a mudança em questão fácil. Feito isso, ele terá aberto caminho para dar dois passos a frente e implementar a mudança que ficou sob sua responsabilidade.

Na maior parte das vezes, os refactorings são oportunistas. No entanto, é possível ter também **refactorings planejados**. Normalmente, eles são mudanças mais profundas, demoradas e complexas, que não valem a pena encaixar no meio de uma outra tarefa de desenvolvimento. Em vez disso, eles devem ser realizados em sessões planejadas e dedicadas. Por exemplo,

esses refactorings podem envolver a quebra de um pacote em dois ou mais subpacotes, o que pode exigir atualizações em diversas partes de um sistema. Como um segundo exemplo, pode ser que o time de desenvolvimento negligenciou a prática de refactorings por muito tempo. Então, como existem muitos refactorings pendentes, é melhor planejá-los para um período de tempo específico. No entanto, como afirma Fowler tais episódios de refactorings planejados devem ser raros. A maior parte do esforço de refactoring deve ser do tipo normal e oportunista.

## Refactorings Automatizados

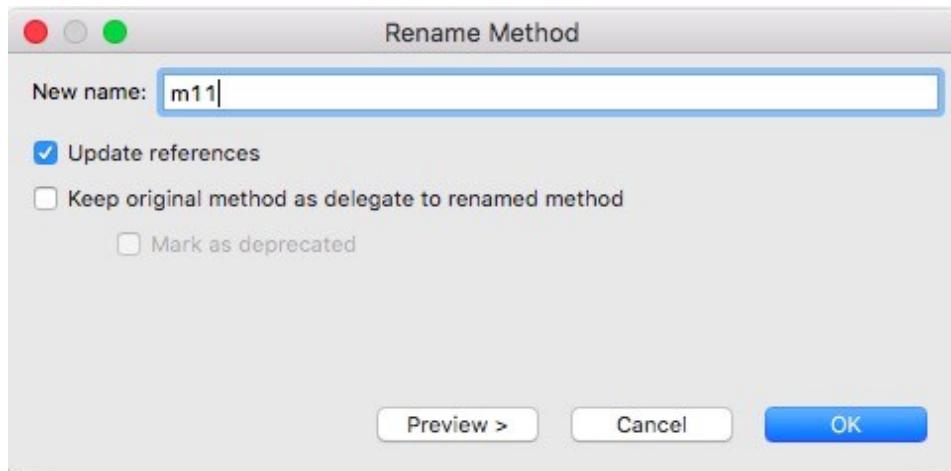
Diversas IDEs oferecem suporte para automatizar a realização de refactorings, da seguinte forma: o usuário seleciona o trecho de código que pretende refatorar e a operação de refactoring que deseja realizar. Então, a IDE realiza essa operação automaticamente. Para deixar o conceito mais claro, as próximas figuras ilustram um renomeação automática de método via uma IDE. Primeiro, o usuário marca o nome do método `m1`, como mostra a figura:

```
class A {  
    void m1() {}  
    void m2() {}  
    void m3() { m1(); }  
}  
  
class B {  
    void m4() { new A().m1(); }  
}
```

Em seguida, ele escolhe as opções de Refactor e Rename da sua IDE:

<b>Refactor</b>	►	Rename...	►	⌘R
Local History	►	Move...	►	⌘V
References	►	Change Method Signature...	►	⌘C
Declarations	►	Inline...	►	⌘I
 Add to Snippets...		Move Type to New File...		
 Coverage As	►	Extract Interface...		
 Run As	►	Extract Superclass...		
 Debug As	►	Use Supertype Where Possible...		
Team	►	Pull Up...		
Compare With	►	Push Down...		
Replace With	►	Extract Class...		
<input checked="" type="checkbox"/> Validate		Introduce Parameter Object...		
Preferences...		Introduce Indirection...		
 Remove from Context	►	Infer Generic Type Arguments...		

Então a IDE pergunta o novo nome que ele pretende dar ao método (acompanhe na figura a seguir). Nessa mesma caixa de diálogo, o desenvolvedor informa que deseja atualizar as referências para esse método, para que elas usem o novo nome.



Feito isso, a IDE realiza o refactoring automaticamente:

```
class A {  
    void m11() {}  
    void m2() {}  
    void m3() { m11(); }  
}  
  
class B {  
    void m4() { new A().m11(); }  
}
```

Primeiro, o nome do método na classe A foi renomeado para `m11`. Além disso, as chamadas feitas em `m3` e `m4` foram atualizadas para usar o novo nome.

Apesar de chamado de refactoring automatizado, o exemplo que mostramos deixa claro que cabe ao usuário indicar o trecho de código que deseja refatorar e o refactoring que deseja realizar. Ele também deve dar informações sobre esse refactoring. Por exemplo, o novo nome do identificador, no caso de uma renomeação. A partir daí é que o refactoring torna-se automatizado.

Antes de aplicar o refactoring, a IDE verifica se as suas **pré-condições** são verdadeiras, isto é, se a execução do refactoring — conforme requisitado pelo usuário — não vai causar um erro de compilação ou então mudar o comportamento do programa. No exemplo anterior, se o usuário pedir para renomear `m1` para `m2`, a IDE vai informar que esse refactoring não pode ser realizado, pois a classe já tem um método chamado `m2`.

## Verificação de Pré-condições de Refactorings

A verificação das pré-condições de refactorings não é uma tarefa trivial. Para ilustrar a complexidade dessa tarefa, vamos reusar um pequeno programa em Java, proposto por Friedrich Steimann e Andreas Thies ([link](#)). Conforme mostrado a seguir, esse programa inclui duas classes, A e B, implementadas em arquivos distintos, mas pertencentes a um mesmo pacote P1. Assim, a chamada de `m("abc")` no primeiro arquivo irá resultar na execução do método `m(String)` da classe B.

```

// arquivo A.java
package P1;

public class A {
    void n() {
        new B().m("abc"); // executa m(String) de B
    }
}

// arquivo B.java
package P1;

public class B {
    public void m(Object o) {...}
    void m(String s) {...}
}

```

No entanto, suponha que a classe B seja movida para um novo pacote; por exemplo, para um pacote P2:

```

// arquivo B.java
package P2; // novo pacote de B

public class B {
    public void m(Object o) {...}
    void m(String s) {...}
}

```

Apesar de parecer inofensiva, essa Movimentação de Classe muda o comportamento do programa. Na nova versão, a chamada de `m("abc")` resulta na execução de `m(Object)` e não mais de `m(String)`. O motivo é que a classe B não está mais no mesmo pacote da classe A. Por isso, `m(String)` deixou de ser visível para A, pois ele não é um método público. Para leitores que não conhecem Java, um método público de uma classe pública, como `m(Object)`, pode ser chamado em qualquer parte do código. Mas métodos sem modificador de visibilidade, como `m(String)`, somente podem ser chamados por código do mesmo pacote.

Em resumo, o exemplo de Movimentação de Classes que apresentamos não é um refactoring, pois ele não preserva o comportamento do programa. Se realizado com o apoio de uma IDE, caberia a ela detectar tal fato e avisar ao usuário que o refactoring não poderá ser realizado, pois ele muda o comportamento do programa.

# Code Smells

Code Smells — também conhecidos como **bad smells** — são indicadores de código de baixa qualidade, isto é, código difícil de manter, entender, modificar ou testar. Em resumo, código que não está cheirando bem e que portanto talvez possa ser refatorado. No entanto, nessa definição, o termo indicadores significa que não devemos considerar que todo code smell deve ser imediatamente refatorado. Essa decisão depende de outros fatores, como a importância do trecho de código e a frequência com que ele precisará ser mantido. Feita essa ressalva, vamos apresentar, no restante desta seção, alguns dos principais code smells.

## Código Duplicado

Duplicação de código é o principal code smell e aquele com o maior potencial para prejudicar a evolução de um sistema. Código duplicado aumenta o esforço de manutenção, pois alterações têm que ser replicadas em mais de uma parte do código. Consequentemente, corre-se o risco de alterar uma parte e esquecer de uma outra. Código duplicado também torna a base de código mais complexa, pois dados e comandos que poderiam estar modularizados em métodos ou classes ficam espalhados pelo sistema.

Para eliminar duplicação de código, pode-se usar os seguintes refactorings: Extração de Método (recomendado quando o código duplicado está dentro de dois ou mais métodos), Extração de Classe (recomendado quando o código duplicado refere-se a atributos e métodos que aparecem em mais de uma classe) e Pull Up Method (recomendado quando o código duplicado é um método presente em duas ou mais subclasses).

Trechos de código que possuem código idêntico são chamados de **clones**. No entanto, diferentes critérios podem ser usados para definir quando dois trechos de código A e B são, de fato, idênticos. Esses critérios dão origem a quatro tipos de clones, conforme descrito a seguir:

- Clone do Tipo 1: quando A e B têm o mesmo código, com diferenças apenas em comentários e espaços.

- Clone do Tipo 2: quando A e B são clones do Tipo 1, porém as variáveis usadas em A e B podem ter nomes diferentes.
- Clone do Tipo 3: quando A e B são clones do Tipo 2, porém com pequenas mudanças em comandos.
- Clone do Tipo 4: quando A e B são semanticamente equivalentes, mas suas implementações são baseadas em algoritmos diferentes.

**Exemplo:** Para ilustrar esses tipos de clones, vamos usar a seguinte função:

```
int fatorial(int n) {
    fat = 1;
    for (i = 1; i <= n; i++)
        fat = fat * i;
    return fat;
}
```

A seguir, mostramos quatro clones dessa função.

- Clone Tipo 1: insere um comentário e remove espaços entre os operadores.

```
int fatorial(int n) {
    fat=1;
    for (i=1; i<=n; i++)
        fat=fat*i;
    return fat; // retorna fatorial
}
```

- Clone Tipo 2: renomeia algumas variáveis.

```
int fatorial(int n) {
    f = 1;
    for (j = 1; j <= n; j++)
        f = f * j;
    return f;
}
```

- Clone Tipo 3: insere um comando simples, para imprimir o valor do fatorial.

```
int factorial(int n) {
    fat = 1;
    for (j = 1; j <= n; j++)
        fat = fat * j;
    System.out.println(fat); // novo comando
    return fat;
}
```

- Clone Tipo 4: implementa uma versão recursiva da função.

```
int factorial(int n) {
    if (n == 0)
        return 1;
    else return n*fatorial(n-1);
}
```

Em todos esses casos, não precisaríamos ter duas ou mais funções factorial no sistema. Apenas uma delas poderia ficar no código e a outra seria removida.

**Mundo Real:** Em 2013, Auki Yamashita e Leon Moonen, dois pesquisadores de um laboratório de pesquisa na Noruega, publicaram os resultados de um estudo exploratório sobre code smells envolvendo 85 desenvolvedores de software ([link](#)). Quando esses desenvolvedores foram questionados sobre os code smells com os quais eles tinham mais preocupação, a resposta mais comum foi Código Duplicado, com quase 20 pontos na escala usada pelos pesquisadores para ordenar as respostas. Em segundo lugar, com metade dos pontos, ficou Métodos Longos, que é o code smell que estudaremos a seguir.

## Métodos Longos

Em qualquer sistema, métodos devem ser pequenos, com nomes auto-explicativos e poucas linhas de código. **Métodos Longos** são considerados um code smell, pois eles tornam o código mais difícil de entender e manter. Quando nos deparamos com um método longo, devemos considerar a possibilidade de usar uma Extração de Método para quebrá-lo em métodos

menores. No entanto, não existe um limite máximo de linhas de código que possa ser usado arbitrariamente para classificar métodos longos, pois isso depende da linguagem de programação, da relevância do método, do domínio do sistema, etc. Porém, modernamente, existe uma tendência em escrever métodos pequenos, com menos de 20 linhas de código, por exemplo.

## Classes Grandes

Assim como métodos, classes não devem assumir muitas responsabilidades e prover serviços que não são coesos. Por isso, **Classes Grandes** (*Large Class*) é considerado um code smell, pois, assim como métodos longos, elas tornam o código mais difícil de entender e manter. Normalmente, é mais difícil também reusar essas classes em outro pacote ou sistema. Classes grandes são caracterizadas por um grande número de atributos, com baixa coesão entre eles. A solução para esse smell consiste em usar Extração de Classe para extrair uma classe menor A' a partir de uma classe grande A. Em seguida, a classe A passa a ter um atributo do tipo para A'.

**Aprofundamento:** Quando uma classe cresce tanto que ela passa a monopolizar grande parte da inteligência de um sistema, ela é chamada de **God Class** — ou então de **Blob**. Classes com nomes muito genéricos, como Manager, System ou Subsystem podem representar instâncias desse smell.

## Feature Envy

Esse smell designa um método que parece invejar os dados e métodos de uma outra classe. Dizendo de outro modo, ele acessa mais atributos e métodos de uma classe B do que de sua atual classe A. Portanto, deve-se analisar a possibilidade de usar Movimentação de Método para migrá-lo para a classe invejada.

O método `fireAreaInvalidated2`, mostrado no código a seguir, é um exemplo de Feature Envy. Podemos observar que ele realiza três chamadas de métodos, mas todas têm como alvo um mesmo objeto `abt` do tipo `AbstractTool`. Por outro lado, ele não acessa nenhum atributo ou chama qualquer método da sua classe atual. Logo, deve-se analisar a conveniência de mover esse método para `AbstractTool`.

```

public class DrawingEditorProxy
    extends AbstractBean implements DrawingEditor {
    ...
    void fireAreaInvalidated2 (AbstractTool abt , Double r ){
        Point p1 = abt.getView().drawingToView (...);
        Point p2 = abt.getView().drawingToView (...);
        Rectangle r=new Rectangle(p1.x,p1.y,p2.x-p1.x p2.y-p1.y);
        abt.fireAreaInvalidated (r);
    }
    ...
}

```

## Métodos com Muitos Parâmetros

Além de pequenos, métodos, na medida do possível, devem ter poucos parâmetros. Isto é, **Métodos com Muitos Parâmetros** é um smell, que pode ser eliminado de duas formas principais. Primeiro, deve-se verificar se um dos parâmetros pode ser obtido diretamente pelo método chamado, como mostrado a seguir:

```

p2 = p1.f();
g(p1, p2);

```

Nesse caso, p2 é desnecessário, pois ele pode ser obtido logo no início de g, da seguinte forma:

```

void g(p1) {
    p2 = p1.f(); ...
}

```

Uma outra possibilidade é criar um tipo que agrupe alguns dos parâmetros de um método. Por exemplo, suponha o seguinte método:

```

void f(Date inicio, Date fim) {
    ...
}

```

Pode-se criar uma classe DateRange para representar uma faixa de datas. O código refatorado ficaria assim:

```

class DateRange {
    Date inicio;
    Date fim;
}

```

```
}
```

```
void f(DateRange range) {
```

```
    ...
```

```
}
```

## Variáveis Globais

Conforme estudamos no capítulo sobre princípios de projeto, variáveis globais devem ser evitadas, pois elas dão origem a um tipo de **acoplamento ruim**. Por isso, elas também constituem um code smell. O principal motivo é que variáveis globais dificultam o entendimento de um módulo de forma independente dos demais módulos de um sistema. Para entender melhor, suponha a seguinte função:

```
void f(...) {
```

```
    // computa um determinado valor x
```

```
    return x + g; // onde g é uma variável global
```

```
}
```

Apenas analisando e estudando esse código, você consegue dizer o valor que `f` retorna? A resposta é negativa, pois não basta entender o código que precede o comando `return` da função. Precisamos conhecer também o valor de `g`. Porém, como `g` é uma variável global, seu valor pode ser alterado em qualquer parte do programa. Tal situação pode facilmente introduzir bugs nessa função, pois agora uma única linha de código distante e não relacionada com `f` pode influir no seu resultado. Para isso, basta que essa linha altere o valor de `g`. Antes de concluir, em linguagens como Java, atributos estáticos de classes funcionam exatamente como variáveis globais. Logo, eles também representam um code smell.

## Obsessão por Tipos Primitivos

Este code smell ocorre quando tipos primitivos (isto é, `int`, `float`, `String`, etc.) são usados no lugar de classes. Por exemplo, suponha que precisamos declarar uma variável para armazenar o CEP de um endereço. Na pressa para usar rapidamente a variável, podemos declará-la como sendo do tipo `String`, em vez de criar uma classe dedicada — por exemplo, `CEP` — para esse fim. A principal vantagem é que uma classe pode oferecer métodos para manipular os valores que a variável vai armazenar. Por exemplo, a

construtora da classe pode verificar se o CEP informado é válido antes de inicializar o objeto. Dessa forma, a classe assume essa responsabilidade e, consequentemente, evita que ela seja uma preocupação de seus clientes. Em resumo, não devemos ficar obcecados com tipos primitivos. Em vez disso, devemos analisar a possibilidade de criar classes que encapsulem valores primitivos e que ofereçam operações para manipulá-los. No próximo code smell, iremos complementar essa recomendação e sugerir que tais objetos, sempre que possível, devem ser também imutáveis.

## Objetos Mutáveis

Na segunda versão de seu livro, Fowler considera que **Objetos Mutáveis** são um code smell. Um objeto mutável é aquele cujo estado pode ser modificado. Por outro lado, um objeto imutável, uma vez criado, não pode mais ser alterado. Para viabilizar a criação de objetos imutáveis, classes devem declarar todos os seus atributos como privados e final (um atributo final somente pode ser usado para leitura). A classe também deve ser declarada final, para proibir a criação de subclasses. Se precisarmos alterar um objeto imutável, a única alternativa consiste em criar uma nova instância do objeto com o estado desejado.

Por exemplo, objetos do tipo `String` em Java são imutáveis, como ilustra o seguinte programa.

```
class Main {  
    public static void main(String[] args) {  
        String s1 = "Hello World";  
        String s2 = s1.toUpperCase();  
        System.out.println(s1);  
        System.out.println(s2);  
    }  
}
```

Esse programa vai imprimir Hello World e depois HELLO WORLD. O motivo é que o método `toUpperCase` não altera a string `s1`, mas apenas retorna uma cópia dela com as letras em maiúsculo.

Sempre que possível devemos criar objetos imutáveis, pois eles podem ser compartilhados de forma livre e segura com outras funções. Por exemplo, em Java, você pode passar uma string como parâmetro de uma função e ter

certeza de que essa função não vai mudar o seu conteúdo. Isso não ocorreria se strings fossem mutáveis, pois sempre haveria o risco de a função chamada alterar a string recebida como parâmetro. Como uma segunda vantagem, objetos imutáveis são por construção thread-safe, isto é, não é necessário sincronizar o acesso de threads aos seus métodos. O motivo é também simples: os problemas clássicos de sistemas concorrentes, como condições de corrida, ocorrem apenas quando múltiplas threads alteram o estado de um objeto. Se esse estado for imutável, tais problemas ficam automaticamente eliminados.

No entanto, precisamos entender esse code smell no contexto da linguagem de programação usada em um sistema. Por exemplo, em linguagens funcionais, objetos são imutáveis por definição, ou seja, esse code smell nunca vai ocorrer. Por outro lado, em linguagens imperativas, é normal ter um certo número de objetos mutáveis. Ou seja, nessas linguagens, o que temos que fazer é minimizar o número de tais objetos, sem, no entanto, imaginar que vamos eliminá-los por completo. Por exemplo, devemos considerar a possibilidade de tornar imutáveis objetos simples e pequenos, como aqueles das classes CEP, Moeda, Endereco, Data, Hora, Fone, Cor, Email, etc. Para ilustrar, mostramos a seguir a implementação de uma classe Data imutável:

```
final public class Data { // final: não pode ter subclasses
    final private int dia; // final: inicializado uma única vez
    final private int mes;
    final private int ano;

    private void check(int dia, int mes, int ano)
        throws InvalidDateException {
        // verifica se data válida
        // se não for, lança InvalidDateException
    }

    public Data(int dia, int mes, int ano)
        throws InvalidDateException {
        check(dia, mes, ano);
        this.dia = dia;
        this.mes = mes;
        this.ano = ano;
    }
}
```

```
// outros métodos  
}
```

## Classes de Dados

Tais classes possuem apenas atributos e nenhum método. No máximo, elas possuem getters e setters. Porém, como recorrente com code smells, não devemos considerar que **Classes de Dados** são sempre um erro de projeto. Em vez disso, o importante é analisar o código e verificar a possibilidade de mover comportamento para essas classes. Isto é, criar métodos nessas classes, para realizar operações que já estão sendo realizadas, mas de forma espalhada em outras classes.

## Comentários

Pode soar estranho ver comentários incluídos em uma lista de code smells. Por exemplo, em cursos de Introdução a Programação, os alunos são incentivados a comentar todo o código produzido, com o objetivo de ensinar a importância de documentação de código. No livro *Elements of Programming Style*, Brian Kerninghan — um dos criadores das primeiras versões do sistema operacional Unix e da linguagem de programação C — e P. J. Plauger dão uma recomendação que ajuda, de forma certeira, a esclarecer essa dúvida. Eles recomendam o seguinte:

Não comente código ruim, reescreva-o.

A ideia é que comentários não devem ser usados para explicar código ruim. Em vez disso, deve-se refatorar o código e, com isso, melhorar sua qualidade e legibilidade. Feito isso, existe uma boa chance de que o comentário não seja mais necessário. Um exemplo são métodos longos como aquele do programa a seguir.

```
void f() {  
    // task1  
    ...  
    // task2  
    ...  
    // taskn  
    ...  
}
```

Se usarmos Extração de Método para extrair o código comentado, teremos o código a seguir de melhor qualidade:

```
void task1 { ... }
void task2 { ... }
void taskn { ... }

void f {
    task1();
    task2();
    ...
    taskn();
}
```

Observe que, no método `f`, após a refatoração, comentários não são mais necessários, pois os nomes dos métodos chamados já revelam muito do que eles fazem.

**Aprofundamento:** **Débito técnico** é um termo cunhado por Ward Cunningham, em 1992, para designar os problemas técnicos que podem dificultar a manutenção e evolução de um sistema. Dentre outros, esses problemas incluem falta de testes, problemas arquiteturais (por exemplo, sistemas mais parecidos com uma *big ball of mud*), sistemas com um número grande de code smells ou sem qualquer documentação. A intenção de Cunningham foi criar um termo que pudesse ser compreendido por gerentes e pessoas sem conhecimento de princípios e práticas de Engenharia de Software. Assim, ele optou pelo termo débito para reforçar que esses problemas, caso não sejam resolvidos, em algum momento vão requerer o pagamento de juros. Tais juros vão se manifestar na forma de sistemas inflexíveis e difíceis de manter, nos quais a correção de bugs e a implementação de novas funcionalidades leva cada vez mais tempo e mostra-se mais arriscada.

Para ilustrar melhor o conceito, suponha que exista um débito técnico em um determinado módulo M de um sistema. Suponha ainda que a adição de uma nova funcionalidade F1 em M requer um esforço de 3 dias. Porém, se não houvesse o débito técnico, F1 poderia ser implementada em apenas 2 dias. Essa diferença de um dia constitui os juros cobrados pela existência do débito técnico em M. Uma alternativa seria então pagar o principal do débito, isto é, remover completamente o débito técnico de M. Mas isso pode

levar, por exemplo, 4 dias. Ou seja, se considerarmos que vamos estender o módulo com apenas F1, ainda não há vantagem. Porém, suponha que nos próximos meses vamos ter que implementar mais funcionalidades em M, tais como F2, F3, F4, etc. Nesse caso, a eliminação do principal do débito técnico pode compensar.

## Bibliografia

Martin Fowler. Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1st edition, 2000.

Martin Fowler. Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2nd edition, 2018.

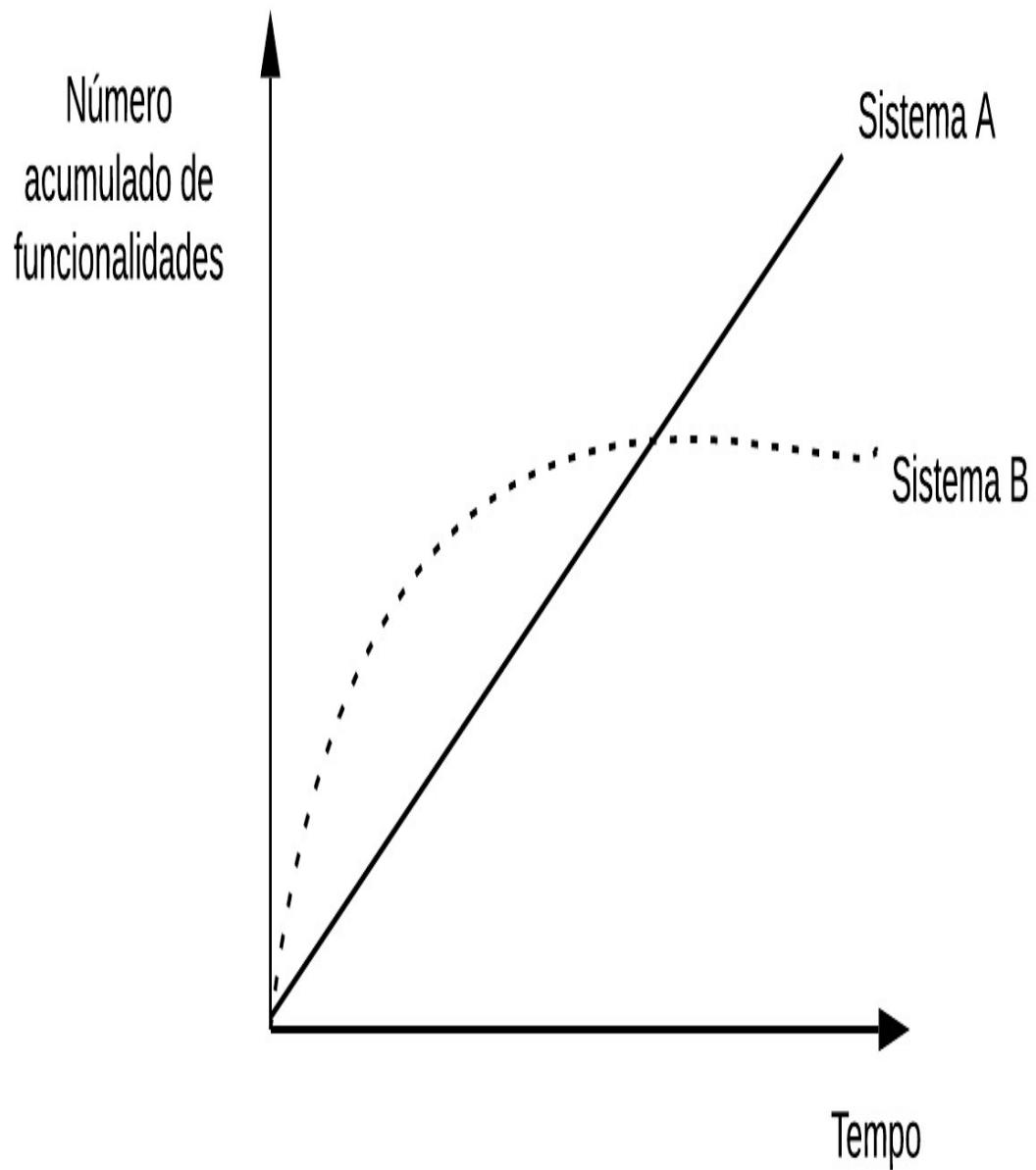
Danilo Silva, Nikolaos Tsantalis, Marco Túlio Valente. Why We Refactor? Confessions of GitHub Contributors. Foundations of Soft. Engineering, 2016.

## Exercícios de Fixação

1. Marque a alternativa FALSA:

1. refactorings melhoram o projeto de um sistema de software.
  2. refactorings tornam o código de um sistema mais fácil de ser entendido.
  3. refactorings facilitam a localização e a correção de bugs futuros.
  4. refactorings aceleram a implementação de novas funcionalidades.
  5. refactorings melhoram o desempenho de um sistema, em termos de tempo de execução.
2. O gráfico a seguir mostra o total acumulado de novas funcionalidades implementadas em dois sistemas (A e B), de domínios semelhantes, desenvolvidos por times semelhantes, usando as mesmas tecnologias. Em

qual dos dois sistemas você acha que refactorings foram realizados de forma sistemática? Justifique a sua resposta.



3. Descreva as diferenças entre refactorings oportunistas e refactorings planejados. Qual dessas formas de refactoring deve ser mais comum?

4. Escreva o nome de refactorings A e B que se executados em sequência não produzem impacto no código de um sistema. Ou seja, o refactoring B reverte as transformações realizadas pelo refactoring A.

5. Nos exemplos a seguir, extraia o código comentado com a palavra extrair para um método g.

```
class A {  
    void f() {  
        int x = 10  
        x++;  
        print x; // extrair  
    }  
}
```

```
class A {  
    void f() {  
        int x = 10  
        x++; // extrair  
        print x; // extrair  
    }  
}
```

```
class A {  
    void f() {  
        int x = 10  
        x++; // extrair  
        print x; // extrair  
        int y = x+1;  
        ...  
    }  
}
```

```
class A {  
    void f() {  
        int x = 10  
        int y; // extrair  
        y = h()*2; // extrair  
        print y; // extrair  
        int z = y+1;  
        ...  
    }  
}
```

```
    }  
}
```

6. A seguinte função calcula o n-ésimo termo da sequência de Fibonacci. O primeiro termo dessa sequência é 0; o segundo termo é 1; e a partir daí o n-ésimo termo é a soma dos dois termos anteriores.

```
int fib(int n) {  
    if (n == 1)  
        return 0;  
    if (n == 2)  
        return 1;  
    return fib(n-1) + fib(n-2);  
}
```

Descreva clones dos Tipos 1, 2, 3 e 4 para essa função. Não precisa implementar o código do clone, mas seja bem preciso na sua resposta ao indicar as diferenças entre cada clone e o código acima.

7. Seja o seguinte trecho de código de uma classe Moeda, que vai armazenar um determinado valor em reais.

```
class Moeda {  
    ...  
    private double valor = 0.0;  
    void add(Moeda m) {  
        this.valor = this.valor + m.getValor();  
    }  
    ...  
}
```

1. Descreva porque objetos da classe Moeda são mutáveis.
2. Reimplemente esse trecho da classe Moeda de forma a assegurar que os seus objetos sejam imutáveis (como vimos no capítulo, objetos mutáveis tendem a ser um code smell, principalmente no caso de objetos pequenos e simples, como provavelmente é o caso de objetos da classe em questão).
8. Como discutido no final da Seção 9.5, comentários que são usados para explicar código ruim são considerados um code smell. Nessas situações, o

ideal é tornar o código mais claro e, então, remover os comentários. A seguir, mostramos mais um caso de comentário que pode ser deletado. Explique por que esses comentários são desnecessários.

```
// classe Aluno
class Aluno {

    // matrícula do aluno
    int matricula;

    // data de nascimento do aluno
    Date dataNascimento;

    // endereço do aluno
    Endereco endereco;

    // construtor default da classe Aluno
    Aluno() {
        ...
    }
    ...
}
```

9. Use uma IDE, como o Eclipse ou IntelliJ para Java, para realizar um refactoring simples, em um de seus programas. Por exemplo, realize uma renomeação de método. Quais as vantagens de se realizar refactorings com o suporte de IDEs?

10. Use uma IDE para testar o exemplo de Movimentação de Classe discutido na Seção 9.4.1, isto é, o exemplo com classes A e B e pacotes P1 e P2. Se realizar a Movimentação de Classe discutida nesse exemplo via IDE, ocorrerá algum erro? Se sim, descreva o erro detectado pela IDE.

# Cap 10 DevOps

Este capítulo inicia discutindo o conceito de DevOps e seus benefícios (Seção 10.1). Apesar de ser um termo novo, existe uma tendência em ver DevOps como um movimento que visa introduzir práticas ágeis na última milha de um projeto de software, isto é, quando o sistema vai entrar em produção. Além de discutir o conceito, tratamos de três práticas importantes quando se adota DevOps. São elas: Controle de Versões (Seção 10.2), Integração Contínua (Seção 10.3) e Deployment Contínuo (Seção 10.4).

## Introdução

Até agora, neste livro, estudamos um conjunto de práticas para desenvolvimento de software com qualidade e agilidade. Por meio de métodos ágeis — como Scrum, XP ou Kanban —, vimos que o cliente deve participar desde o primeiro dia da construção de um sistema. Também estudamos práticas importantes para produção de software com qualidade, como testes de unidade e refactoring. Estudamos ainda princípios e padrões de projeto e também padrões arquiteturais.

Logo, após aplicar o que vimos, o sistema — ou um incremento dele, resultante de um sprint — está pronto para entrar em produção. Essa tarefa é conhecida pelos nomes de **implantação (deploy)**, **liberação (release)** ou **entrega (delivery)** do sistema. Independentemente do nome, ela não é tão simples e rápida como pode parecer.

Historicamente, em organizações tradicionais, a área de Tecnologia da Informação era dividida em dois departamentos:

- Departamento de Sistemas (ou Desenvolvimento), formado por desenvolvedores, programadores, analistas, arquitetos, etc.
- Departamento de Suporte (ou Operações), no qual ficavam alocados os administradores de rede, administradores de bancos de dados, técnicos de suporte, técnicos de infraestrutura, etc.

Hoje em dia, é fácil imaginar os problemas causados por essa divisão. Na maioria das vezes, a área de suporte tomava conhecimento de um sistema na véspera da sua implantação. Consequentemente, a implantação poderia atrasar por meses, devido a uma variedade de problemas que não tinham sido identificados. Dentre eles, podemos citar a falta de hardware para executar o novo sistema ou a nova funcionalidade, problemas de desempenho, incompatibilidades com o banco de dados de produção, vulnerabilidades de segurança, etc. No limite, esses problemas poderiam resultar no cancelamento da implantação e no abandono do sistema.

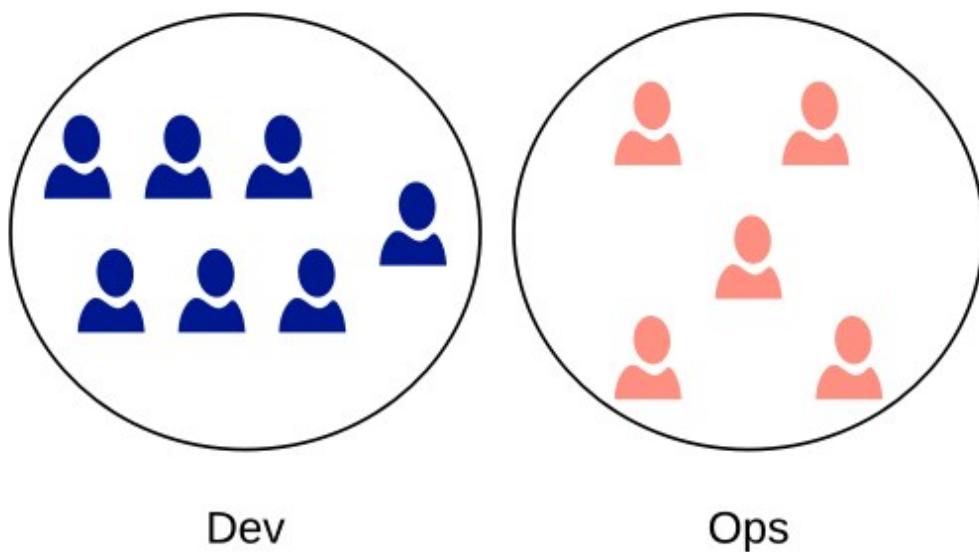
Resumindo, nesse modelo tradicional, existia um stakeholder importante — os administradores de sistemas ou *sysadmins* — que tomava conhecimento das características e requisitos não-funcionais de um novo software na véspera da implantação. Esse problema era agravado pelo fato de os sistemas serem monolitos, cuja implantação gerava todo tipo de preocupação, como mencionado no final do parágrafo anterior.

Então, para facilitar a implantação e entrega de sistemas, foi proposto o conceito de **DevOps**. Por ser um termo recente, ele ainda não possui uma definição consolidada. Mas seus proponentes gostam de descrever DevOps como um movimento que visa unificar as culturas de desenvolvimento (Dev) e de operação (Ops), visando permitir a implantação mais rápida e ágil de um sistema. Esse objetivo está refletido na frase que abre esse capítulo, de autoria de Gene Kim, Jez Humble, Patrick Debois e John Willes, todos eles membros de um grupo de desenvolvedores que ajudou a difundir os princípios de DevOps. Segundo eles, DevOps representa uma disruptão na cultura tradicional de implantação de sistemas ([link](#)):

Em vez de iniciar as implantações à meia-noite de sexta-feira e passar o fim de semana trabalhando para concluir-las, as implantações ocorrem em qualquer dia útil, quando todos estão na empresa e sem que os clientes percebam — exceto quando encontram novas funcionalidades e correções de bugs.

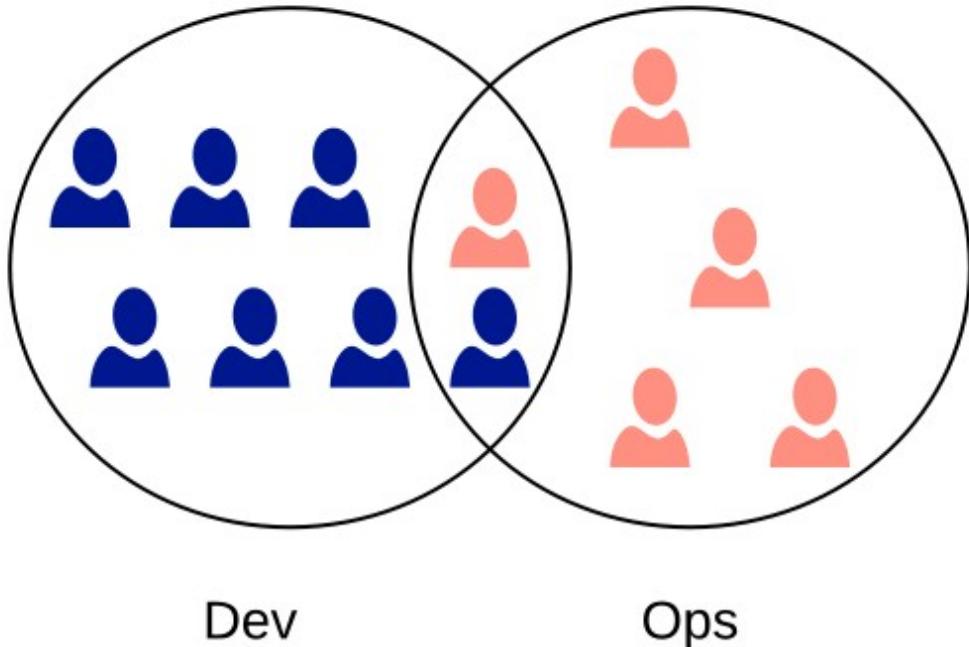
No entanto, DevOps não advoga a criação de um profissional novo, que fique responsável tanto pelo desenvolvimento como pela implantação de sistemas. Em vez disso, defende-se uma aproximação entre o pessoal de desenvolvimento e o pessoal de operações e vice-versa, visando fazer com

que a implantação de sistemas seja mais ágil e menos traumática. Tentando explicar com outras palavras, a ideia é evitar dois silos independentes: desenvolvedores e operadores, com pouca ou nenhuma interação entre eles, como ilustrado na figura a seguir.



Organização que não é baseada em DevOps, pois existe pouca comunicação entre Dev e Ops.

Em vez disso, defende-se que esses profissionais atuem em conjunto desde os primeiros sprints de um projeto, como ilustrado na figura a seguir. Para o cliente, o benefício deve ser a entrada em produção mais cedo do sistema que ele contratou.



Organização baseada em DevOps. Frequentemente, Devs e Ops sentam juntos para discutir questões sobre a entrega do sistema.

Quando migra-se para uma cultura de DevOps, os times ágeis podem incluir um profissional de operações, que participará dos trabalhos em tempo parcial ou mesmo em tempo integral. Sempre em função da demanda, esse profissional pode também participar de mais de um time. A ideia é que ele antecipe problemas de desempenho, segurança, incompatibilidades com outros sistemas, etc. Ele pode também, enquanto o código está sendo implementado, começar a trabalhar nos scripts de instalação, administração e monitoramento do sistema em produção.

De forma não menos importante, DevOps advoga a automatização de todos os passos necessários para colocar um sistema em produção e monitorar o seu correto funcionamento. Isso requer a adoção de práticas que já vimos neste livro, notadamente testes automatizados. Mas também requer o emprego de novas práticas e ferramentas, tais como Integração Contínua (*Continuous Integration*) e Deployment Contínuo (*Continuous Deployment*), que iremos estudar no presente capítulo.

**Mundo Real:** O termo DevOps começou a ser usado no final dos anos 2000 por profissionais frustrados com os atritos constantes entre as equipes de desenvolvimento e de operações. Então, eles convenceram-se de que uma solução seria a adoção de princípios ágeis não apenas na fase de desenvolvimento, mas também na fase de implantação de sistemas. Para citar uma data precisa, em Novembro de 2009 foi realizada, na Bélgica, a primeira conferência da indústria sobre o tema, chamada DevOpsDay. Considera-se que foi nesta conferência, organizada por Patrick Dubois, que a palavra DevOps foi cunhada ([link](#)).

Para finalizar, vamos discutir um conjunto de princípios para entrega de software, enunciados por Jez Humble e David Harley ([link](#)). Apesar de propostos antes da ideia de DevOps ganhar tração, eles estão completamente alinhados com essa ideia. Alguns desses princípios são os seguintes:

- **Crie um processo repetível e confiável para entrega de software.** Esse princípio é o mais importante deles. A ideia é que a entrega de software não pode ser um evento traumático, com passos manuais e sujeitos a surpresas. Em vez disso, colocar um software em produção deve ser tão simples como apertar um botão.
- **Automatize tudo que for possível.** Na verdade, esse princípio é um pré-requisito do princípio anterior. Advoga-se que todos os passos para entrega de um software devem ser automáticos, incluindo seu *build*, a execução dos testes, a configuração e ativação dos servidores e da rede, a carga do banco de dados, etc. De novo, idealmente, queremos apertar um botão e, em seguida, ver o sistema em produção.
- **Mantenha tudo em um sistema de controle de versões.** Tudo no enunciado do princípio refere-se não apenas a todo o código fonte, mas também arquivos e scripts de administração do sistema, documentação, páginas Web, arquivos de dados, etc. Consequentemente, deve ser simples restaurar e voltar o sistema para um estado anterior. Neste capítulo, iniciaremos estudando alguns conceitos básicos de **Controle de Versões**, na Seção 10.2. Além disso, no Apêndice A apresentamos o uso dos principais comandos do sistema Git, que é o sistema de controle de versões mais usado atualmente.

- **Se um passo causa dor, execute-o com mais frequência e o quanto antes.** Esse princípio não tem uma inspiração masoquista. Em vez disso, a ideia é antecipar os problemas, antes que eles se acumulem e as soluções fiquem complicadas. O exemplo clássico é o de **Integração Contínua**. Se um desenvolvedor passa muito tempo trabalhando de forma isolada, ele e o seu time podem depois ter uma grande dor de cabeça para integrar o código. Logo, como integração pode causar dor, a recomendação consiste em integrar código novo com mais frequência e o quanto antes, se possível, diariamente. Iremos estudar mais sobre integração contínua na Seção 10.3.
- **Concluído significa pronto para entrega.** Com frequência, desenvolvedores dizem que uma nova história está pronta (*done*). Porém, ao serem questionados se ela pode entrar em produção, começam a surgir pequenas pendências, tais como: a implementação ainda não foi testada com dados reais, ela ainda não foi documentada, ela ainda não foi integrada com o sistema X, etc. Esse princípio defende então que concluído, em projetos de software, deve ter uma semântica clara, isto é: 100% pronto para entrar em produção.
- **Todos são responsáveis pela entrega do software.** Esse último princípio alinha-se perfeitamente com a cultura de DevOps, que discutimos no início desta Introdução. Ou seja, não admite-se mais que os times de desenvolvimento e de operações trabalham em silos independentes e troquem informações apenas na véspera de uma implantação.

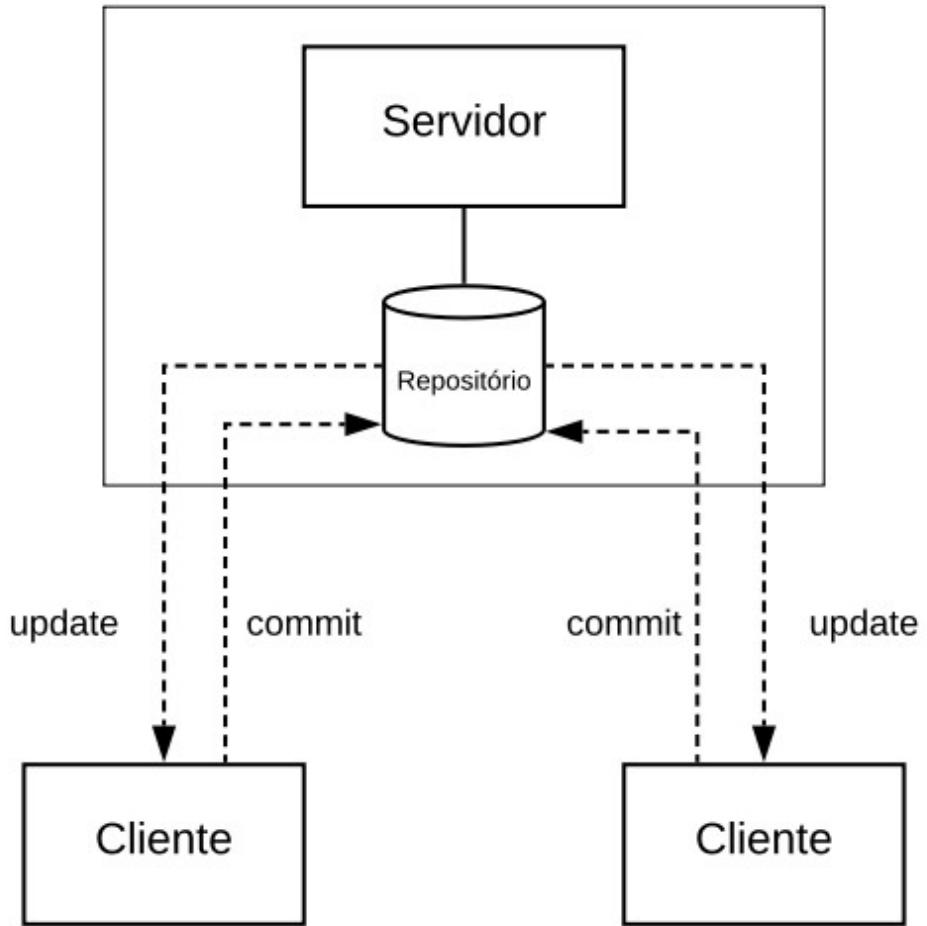
## Controle de Versões

Como mencionamos algumas vezes neste livro, software é desenvolvido em equipe. Por isso, precisamos de um servidor para armazenar o código fonte do sistema que está sendo implementado por um grupo de desenvolvedores. A existência desse servidor é fundamental para que esses desenvolvedores possam colaborar e para que os operadores saibam precisamente qual versão do sistema deve ser colocada em produção. Além disso, sempre é útil manter o histórico das versões mais importantes de cada arquivo. Isso permite, se

necessário, realizar uma espécie de undo no tempo, isto é, recuperar o código de um arquivo como ele estava há anos atrás, por exemplo.

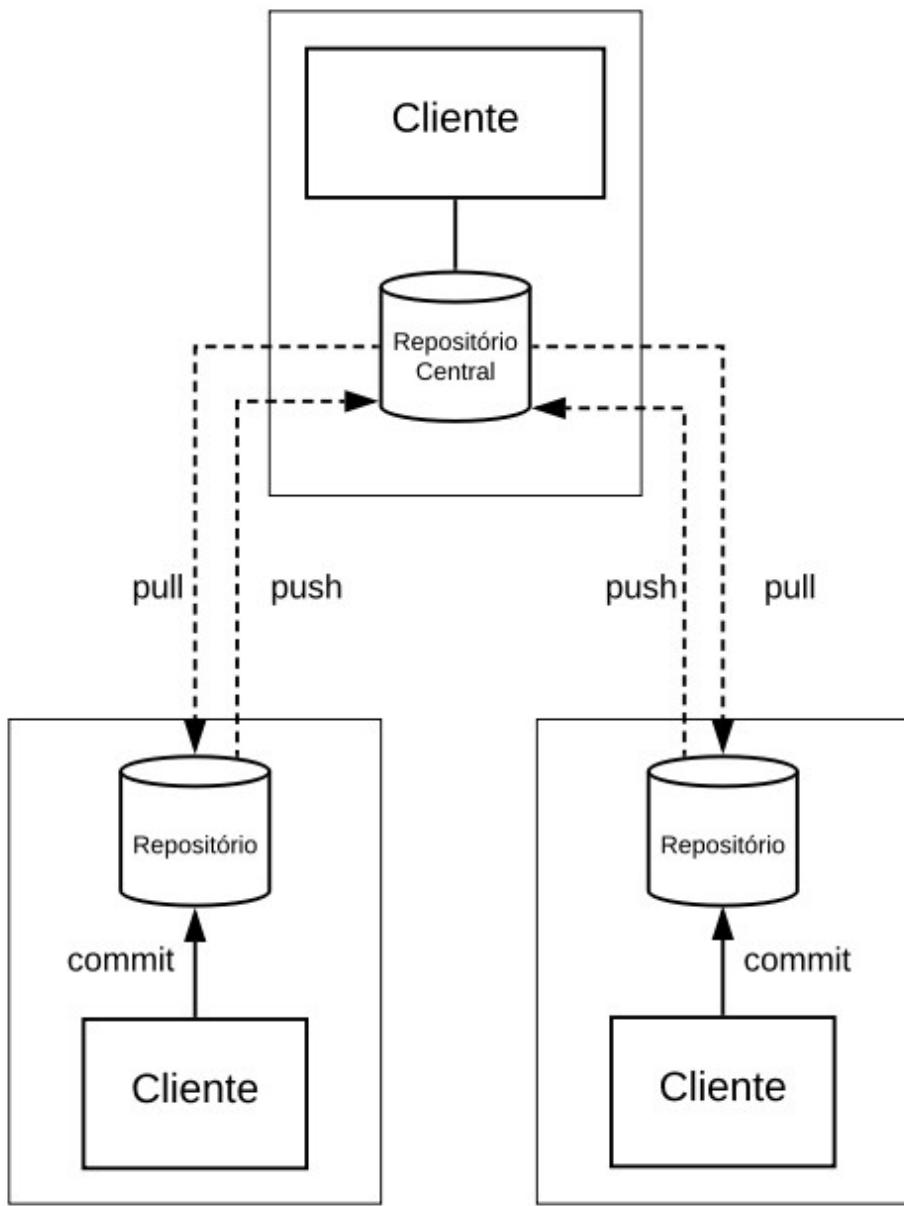
Um **Sistema de Controle de Versões** (VCS, na sigla em inglês) oferece os dois serviços mencionados no parágrafo anterior. Primeiro, ele oferece um **repositório** para armazenar a versão mais recente do código fonte de um sistema, bem como de arquivos relacionados, como arquivos de documentação, configuração, páginas Web, manuais, etc. Em segundo lugar, ele permite que se recupere versões mais antigas de qualquer arquivo, caso seja necessário. Como enunciamos na Introdução, modernamente é inconcebível desenvolver qualquer sistema, mesmo que simples, sem um VCS.

Os primeiros sistemas de controle de versões surgiram no início da década de 70, como o sistema SCCS, desenvolvido para o sistema operacional Unix. Em seguida, surgiram outros sistemas, como o CVS, em meados da década de 80, e depois o sistema Subversion, também conhecido pela sigla svn, no início dos anos 2000. Todos são sistemas centralizados e baseados em uma arquitetura cliente/servidor (veja figura na próxima página). Nessa arquitetura, existe um único servidor, que armazena o repositório e o sistema de controle de versões. Os clientes acessam esse servidor para obter a versão mais recente de um arquivo. Feito isso, eles podem modificar o arquivo, por exemplo, para corrigir um bug ou implementar uma nova funcionalidade. Por fim, eles atualizam o arquivo no servidor, realizando uma operação chamada **commit**, que torna o arquivo visível para outros desenvolvedores.



VCS Centralizado. Existe um único repositório, no nodo servidor.

No início dos anos 2000, começaram a surgir **Sistemas de Controle de Versões Distribuídos** (DVCS). Dentre eles, podemos citar o sistema BitKeeper, cujo primeiro release é de 2000, e os sistemas Mercurial e git, ambos lançados em 2005. Em vez de uma arquitetura cliente/servidor, um DVCS adota uma arquitetura peer-to-peer. Na prática, isso significa que cada desenvolvedor possui em sua máquina um servidor completo de controle de versões, que pode se comunicar com os servidores de outras máquinas, como ilustrado na próxima figura.



VCS Distribuído (DVCS). Cada cliente possui um servidor. Logo, a arquitetura é peer-to-peer.

Em teoria, quando se usa um DVCS, os clientes (ou *peers*) são funcionalmente equivalentes. Porém, na prática, costuma existir uma máquina principal, que armazena a versão de referência do código fonte. Na nossa figura, chamamos esse repositório de **repositório central**. Cada desenvolvedor pode trabalhar de forma independente e até mesmo offline em sua máquina cliente, realizando commits no seu repositório. De tempos em

tempos, ele deve sincronizar esse repositório com o central, por meio de duas operações: **pull** e **push**. Um pull atualiza o repositório local com novos commits disponíveis no repositório central. Por sua vez, um push faz a operação contrária, isto é, ele envia para o repositório central os commits mais recentes realizados pelo desenvolvedor em seu repositório local.

Quando comparado com VCS centralizados, um DVCS tem as seguintes vantagens:

- Pode-se trabalhar e gerenciar versões de forma offline, sem estar conectado a uma rede, pois os commits são realizados primeiro no repositório instalado localmente na máquina do desenvolvedor.
- Pode-se realizar commits com mais frequência, incluindo commits com implementações parciais, pois eles não vão chegar imediatamente até o repositório central.
- Commits são executados em menos tempo, isto é, eles são operações mais rápidas e leves. O motivo é que eles são realizados no repositório local de cada máquina.
- A sincronização não precisa ser sempre com o repositório central. Em vez disso, dois nodos podem também sincronizar os seus repositórios. Por exemplo, pode-se ter uma estrutura hierárquica dos repositórios. Nesses casos, os commits nascem nos repositórios que representam as folhas da hierarquia e vão subindo até chegar ao repositório central.

**Git** é um sistema de controle de versões distribuído cujo desenvolvimento foi liderado por Linus Torvalds, também responsável pela criação do sistema operacional Linux. Nos anos iniciais, o desenvolvimento do kernel do Linux usava um sistema de controle de versões comercial, chamado BitKeeper, que também possui uma arquitetura distribuída. No entanto, em 2005, a empresa proprietária do BitKeeper resolveu revogar as licenças gratuitas que eram usadas no desenvolvimento do Linux. Os desenvolvedores do sistema operacional, liderados por Torvalds, decidiram então iniciar a implementação de um DVCS próprio, ao qual deram o nome de Git. Assim como o Linux, o Git é um sistema de código aberto, que pode ser

gratuitamente instalado em qualquer máquina. O Git é também um sistema de linha de comando. Porém, existem clientes com interfaces gráficas, desenvolvidos por terceiros, que permitem usar o sistema sem ter que digitar comandos.

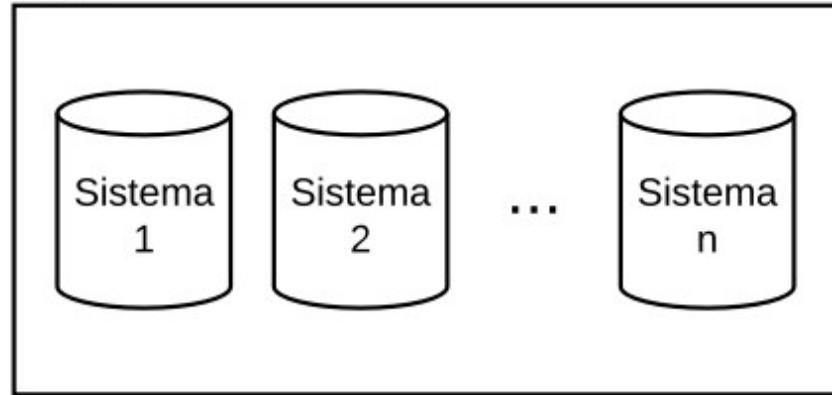
**GitHub** é um serviço de hospedagem de código que usa o sistema Git para prover controle de versões. O GitHub oferece repositórios públicos e gratuitos, para projetos de código aberto, e repositórios fechados e pagos, para uso por empresas. Assim, em vez de manter internamente um DVCS, uma empresa desenvolvedora de software pode alugar esse serviço do GitHub. Uma comparação pode ser feita com serviços de mail. Em vez de instalar um servidor de mail em uma máquina própria, uma empresa pode contratar esse serviço de terceiros, como do Google, via GMail. Apesar de o GitHub ser o mais popular, existem serviços semelhantes providos por outras empresas, como GitLab e BitBucket.

No Apêndice A, apresentamos e ilustramos os principais comandos do sistema Git. São explicados também os conceitos de forks e pull requests, os quais são específicos do GitHub.

## Multirepos vs Monorepos

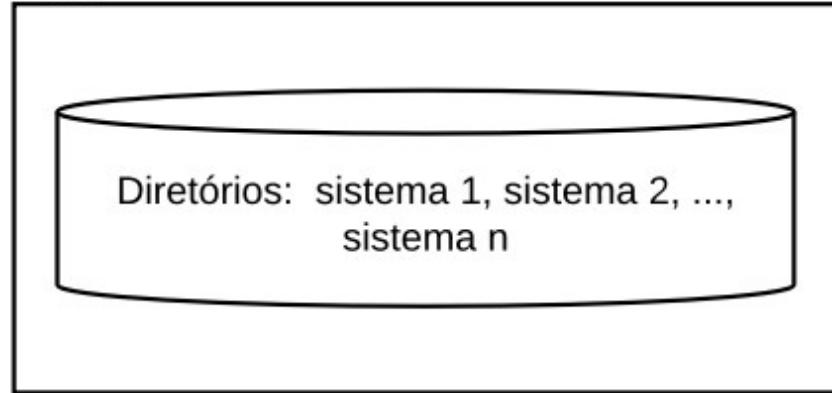
Um VCS gerencia repositórios. Assim, uma organização precisa decidir os repositórios que vai criar em seu VCS. Uma decisão tradicional consiste em criar um repositório para cada projeto ou sistema da organização. Porém, soluções baseadas em um único repositório estão sendo adotadas com mais frequência, principalmente por grandes empresas, como Google, Facebook e Microsoft. Essas duas alternativas — chamadas, respectivamente, de **multirepos** e **monorepos** — são ilustradas nas próximas duas figuras.

VCS



Multirepos: um VCS gerencia vários repositórios. Normalmente, um repositório por projeto ou sistema.

VCS



Monorepos: VCS gerencia um único repositório. Projetos são diretórios desse repositório.

Se pensarmos em contas do GitHub, podemos exemplificar da seguinte forma:

- Se optar por multirepos, uma organização terá vários repositórios, tais como `aserg-ufmg/sistema1`, `aserg-ufmg/sistema2`, `aserg-ufmg/sistema3`, etc.
- Se optar por monorepos, ela terá um único repositório — digamos, `aserg-ufmg/aserg-ufmg`. No diretório raiz desse repositório, teremos os subdiretórios `sistema1`, `sistema2`, `sistema3`, etc.

Dentre as vantagens de monorepos podemos citar:

- Como existe um único repositório, não há dúvida sobre qual repositório possui a versão mais atualizada de um arquivo. Isto é, com monorepos, existe uma única fonte de verdade sobre versões do código fonte.
- Monorepos incentivam o reúso e compartilhamento de código, pois os desenvolvedores têm acesso mais rápido a qualquer arquivo, de qualquer sistema.
- Mudanças são sempre atômicas. Com multirepos, dois commits podem ser necessários para implementar uma única mudança, caso ela afete dois sistemas. Com monorepos, a mesma mudança pode ser realizada por meio de um único commit.
- Facilita a execução de refactorings em larga escala. Por exemplo, suponha a renomeação de uma função utilitária que é usada em todos os sistemas da organização. Com monorepos, essa renomeação pode ser realizada com um único commit.

Por outro lado, monorepos requerem ferramentas para navegar em grandes bases de código. O motivo é que cada desenvolvedor terá em seu repositório local todos os arquivos de todos os sistemas da organização. Por isso, os responsáveis pelo monorepo do Google comentam que foram obrigados a implementar internamente um plug-in para a IDE Eclipse, que facilita o trabalho com uma base de código muito grande, como a que eles possuem na empresa ([link](#)).

## Integração Contínua

Para explicar o conceito de Integração Contínua (CI), iniciamos com uma subseção de motivação. Em seguida, apresentamos o conceito propriamente dito. Feito isso, discutimos outras práticas que uma organização deve adotar junto com CI. Terminamos com uma breve discussão sobre cenários que podem desmotivar o emprego de CI em uma organização.

## Motivação

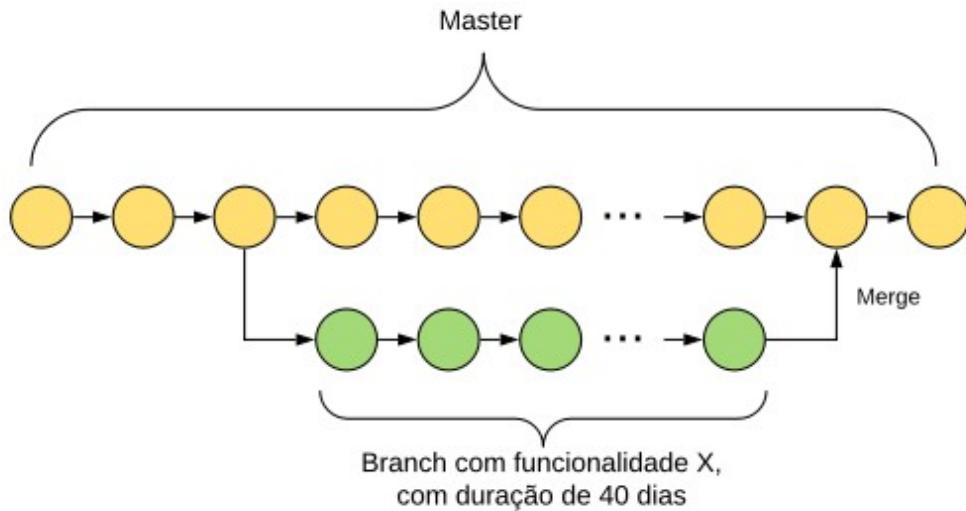
Antes de definir o que é integração contínua, vamos descrever o problema que levou à proposta dessa prática de integração de código. Tradicionalmente, era comum o uso de branches durante a implementação de novas funcionalidades. Branches podem ser entendidos como um sub-diretório interno e virtual, gerenciado pelo sistema de controle de versões. Nesses sistemas, existe um branch principal, conhecido pelo nome de **master** (quando usa-se Git) ou **trunk** (quando usa-se outros sistemas, como svn). Além do branch principal, os usuários podem criar seus próprios branches.

Por exemplo, antes de implementar uma nova funcionalidade, pode ser comum criar um branch para conter o seu código. Tais branches são chamados de **branches de funcionalidades (feature branches)** e, dependendo da complexidade da funcionalidade, eles podem levar meses para serem integrados de volta à linha principal de desenvolvimento. Logo, em sistemas maiores e mais complexos podem existir dezenas de branches ativos.

Quando a implementação da nova funcionalidade terminar, o código do branch deve ser copiado de volta para o master, por meio de um comando do sistema de controle de versões chamado **merge**. Nesse momento, uma variedade de conflitos pode ocorrer, os quais são conhecidos como **conflitos de integração** ou **conflitos de merge**.

Para ilustrar esse cenário, suponha que Alice criou um branch para implementar uma nova funcionalidade X em seu sistema. Como essa funcionalidade era complexa, Alice trabalhou de forma isolada no seu branch por 40 dias, conforme ilustrado na figura da próxima página (cada nodo desse grafo é um commit). Observe que enquanto Alice trabalhava —

realizando commits em seu branch — também ocorriam commits no branch principal.



Desenvolvimento usando branches de funcionalidades.

Então, após 40 dias, quando Alice integrou seu código no master, surgiram diversos conflitos. Alguns deles são descritos a seguir:

- Para implementar a funcionalidade X, o código desenvolvido por Alice chamava uma função  $f_1$ , que existia no master no momento da criação do branch. Porém, no intervalo de 40 dias, a assinatura dessa função foi modificada no master por outros desenvolvedores. Por exemplo, a função pode ter sido renomeada ou ter ganho um novo parâmetro. Ou ainda, em um cenário mais radical,  $f_1$  pode ter sido removida da linha principal de desenvolvimento.
- Para implementar a funcionalidade X, Alice mudou o comportamento de uma função  $f_2$  do master. Por exemplo,  $f_2$  retornava seu resultado em milhas e Alice alterou o seu código para que o resultado fosse retornado em quilômetros. Evidentemente, Alice atualizou todo o código que chamava  $f_2$  no seu branch, para considerar resultados em quilômetros. Porém, no período de 40 dias, surgiram novas chamadas de  $f_2$ , que foram integradas no master, mas supondo um resultado ainda em milhas.

Em sistemas grandes, com milhares de arquivos, dezenas de desenvolvedores e de branches de funcionalidades, os problemas causados por conflitos podem assumir proporções consideráveis e atrasar a entrada em produção de novas funcionalidades. Veja que a resolução de conflitos é uma tarefa manual, que requer análise e consenso entre os desenvolvedores envolvidos. Por isso, os termos **integration hell** ou **merge hell** são usados para descrever os problemas que ocorrem durante a integração de branches de funcionalidades.

Adicionalmente, branches de funcionalidades, principalmente aqueles com duração longa, ajudam a criar silos de conhecimento. Isto é, cada nova funcionalidade passa a ter um dono, pois um desenvolvedor ficou dedicado a ela por semanas. Por isso, esse desenvolvedor pode sentir-se confortável para adotar padrões diferentes do restante do time, incluindo padrões para leiaute do código, para organização de janelas e telas, para acesso a bancos de dados, etc.

## O que é Integração Contínua?

**Integração Contínua** (*Continuous Integration* ou CI) é uma prática de desenvolvimento proposta por Extreme Programming (XP). O princípio motivador da prática já foi comentado na Introdução do presente capítulo: se uma tarefa causa dor, não podemos deixar que ela acumule. Em vez disso, devemos quebrá-la em subtarefas que possam ser realizadas de forma frequente. Como essas subtarefas são pequenas e simples, a dor decorrente da sua realização será menor.

Adaptando para o contexto de integração de código, sabemos que grandes integrações são uma fonte de dor para os desenvolvedores, pois eles têm que resolver de forma manual diversos conflitos. Assim, CI recomenda integrar o código de forma frequente, isto é, contínua. Com isso, as integrações serão pequenas e irão gerar menos conflitos.

Kent Beck, em seu livro de XP, defende o uso de CI da seguinte forma ([link](#)):

Você deve integrar e testar o seu código em intervalos menores do que algumas horas. Programação em times não é um problema do tipo

dividir-e-conquistar. Na verdade, é um problema que requer dividir, conquistar e integrar. A duração de uma tarefa de integração é imprevisível e pode facilmente levar mais tempo do que a tarefa original de codificação. Assim, quanto mais tempo você demorar para integrar, maiores e mais imprevisíveis serão os custos.

Nessa citação, Beck defende várias integrações ao longo de um dia de trabalho de um desenvolvedor. No entanto, essa recomendação não é consensual. Outros autores, como Martin Fowler, mencionam pelo menos uma integração por dia por desenvolvedor ([link](#)), o que parece ser um limite mínimo para um time argumentar que está usando CI.

## **Boas Práticas para Uso de CI**

Quando usa-se CI, o master é constantemente atualizado com código novo. Para garantir que ele não seja quebrado — isto é, deixe de compilar ou possua bugs —, recomenda-se o uso de algumas práticas em conjunto com CI, as quais vamos discutir a seguir.

### **Build Automatizado**

Build é o nome usado para designar a compilação de todos os arquivos de um sistema, até a geração de uma versão executável. Quando se usa CI, o build deve ser automatizado, isto é, não incluir nenhum passo manual. Além disso, é importante que ele seja o mais rápido possível, pois com integração contínua ele será sempre executado. Alguns autores, por exemplo, chegam a recomendar um limite de 10 minutos para execução de um build ([link](#)).

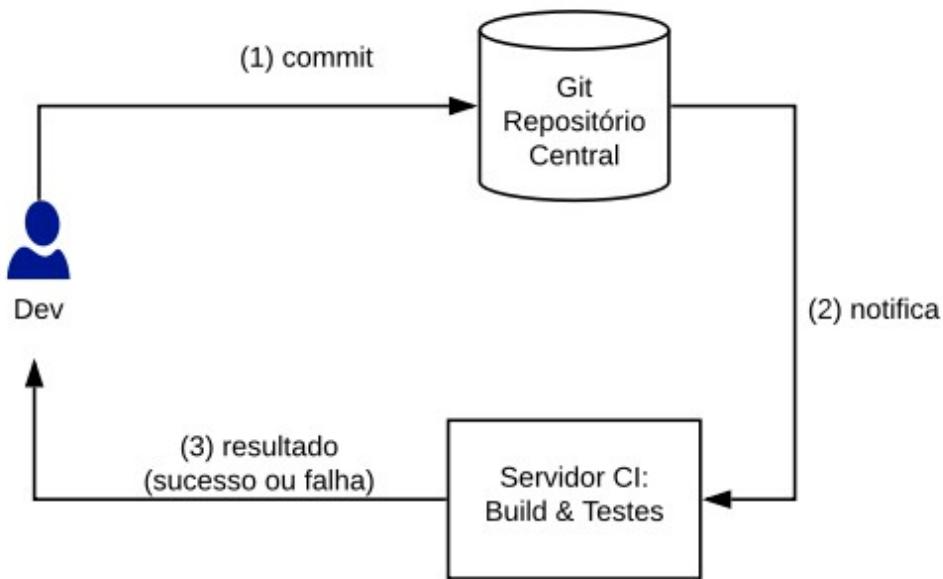
### **Testes Automatizados**

Além de garantir que o sistema compila sem erros após cada novo commit, é importante garantir também que ele continua com o comportamento esperado. Por isso, ao usar CI, deve-se ter uma boa cobertura de testes, principalmente testes de unidade, conforme estudamos no Capítulo 8.

### **Servidores de Integração Contínua**

Por fim, os builds e testes automatizados devem ser executados com frequência, se possível após cada novo commit realizado no master. Para isso, existem **Servidores de CI**, que funcionam da seguinte forma (acompanhe também pela próxima figura):

- Após um novo commit, o sistema de controle de versões avisa o servidor de CI, que clona o repositório e executa um build completo do sistema, bem como roda todos os testes.
- Após a execução do build e dos testes, o servidor notifica o usuário.



## Servidor de Integração Contínua

O objetivo principal de um servidor de integração contínua é evitar a integração de código com problemas, sejam eles de build ou de comportamento. Quando o build falha, costuma-se dizer que ele quebrou. Com frequência, o build na máquina do desenvolvedor pode ter sido concluído com sucesso. Mas ao ser executado no servidor de CI, ele pode falhar. Isso ocorre, por exemplo, quando o desenvolvedor esquece de realizar o commit de algum arquivo. Dependências incorretas são um outro motivo para quebra de builds. Por exemplo, o código pode ter sido compilado e testado na máquina do desenvolvedor usando a versão 2.0 de uma

determinada biblioteca, mas o servidor de CI realiza o build usando a versão 1.0.

Se o servidor de CI notificar o desenvolvedor de que seu código não passou nos testes ou quebrou o build, ele deve parar tudo o que está fazendo e providenciar a correção. Isso é importante porque um build quebrado impacta o trabalho dos outros desenvolvedores, pois eles não vão conseguir mais compilar ou executar o sistema. Costuma-se dizer que nada em uma empresa de software tem maior prioridade do que a correção de um build quebrado. No entanto, a solução pode ser simplesmente reverter o código para a versão anterior ao commit com problemas.

Ainda nesta linha de raciocínio, um desenvolvedor somente deve avançar para uma próxima tarefa de programação após receber o resultado do servidor de CI. Por exemplo, ele não deve começar a escrever código novo antes de ter certeza de que seu último commit passou pelo serviço de CI. Ele também não deve iniciar outras tarefas importantes, como entrar em uma reunião, sair para almoçar ou ir para a casa, antes do resultado desse servidor.

Existem diversos servidores de integração contínua no mercado. Alguns deles são oferecidos como um serviço independente, normalmente gratuito para repositórios de código aberto, mas pago para repositórios privados de empresas. Assim, se você possui um repositório aberto no GitHub, existe mais de uma opção gratuita para ativar um serviço de CI no mesmo.

Uma dúvida comum é se CI é compatível com o uso de branches. Mantendo coerência com a definição de CI, a melhor resposta é a seguinte: sim, desde que os branches sejam integrados de forma frequente no master, via de regra, todo dia. Dizendo de outra forma, CI não é incompatível com branches, mas apenas com branches com um tempo de vida elevado. Por exemplo, Martin Fowler tem a seguinte observação sobre o uso de branches, especificamente branches de funcionalidades, junto com CI ([link](#)):

Na maioria das vezes, branches de funcionalidades constituem uma abordagem incompatível com CI. Um dos princípios de CI é que todos devem enviar commits para a linha de desenvolvimento principal diariamente. Então, a não ser que os branches de funcionalidades durem

menos do que um dia, eles são um animal diferente de CI. É comum ouvir desenvolvedores dizendo que eles estão usando CI porque eles rodam builds automáticos, talvez usando um servidor de CI, após cada commit. Isso pode ser chamado de building contínuo e pode ser uma coisa boa. Porém, como não há integração, não podemos chamar essa prática de CI.

## Desenvolvimento Baseado no Trunk

Como vimos, ao adotar CI, branches devem durar no máximo um dia de trabalho. Logo, o custo/benefício de criá-los pode não compensar. Por isso, quando migram para CI, é comum que as organizações usem também **desenvolvimento baseado no trunk** (*trunk based development* ou TBD). Com TBD, não existem mais branches para implementação de novas funcionalidades ou para correção de bugs. Em vez disso, todo desenvolvimento ocorre no branch principal, também conhecido com trunk ou master.

**Mundo Real:** TBD é usado por grandes empresas desenvolvedoras de software, incluindo Google e Facebook:

- No Google, quase todo desenvolvimento ocorre no HEAD do repositório [isto é, no master]. Isso ajuda a identificar problemas de integração mais cedo e minimiza o esforço para realização de merges. ([link](#))
- No Facebook, todos engenheiros de front-end trabalham em um único branch que é mantido sempre estável, o que também torna o desenvolvimento mais rápido, pois não dispõe-se esforço na integração de branches de longa duração no trunk. ([link](#))

## Programação em Pares

Programação em Pares (*Pair Programming*) pode ser considerada uma forma contínua de revisão de código. Quando se adota essa prática, qualquer novo trecho de código é revisado por um outro desenvolvedor, que encontra-se sentado ao lado do desenvolvedor líder da sessão de programação. Portanto, assim como builds e testes contínuos, recomenda-se usar

programação em pares com CI. Porém, esse uso também não é obrigatório. Por exemplo, o código pode ser revisado após o commit ser realizado no master. No entanto, nesse caso, como o código já foi integrado, os custos de aplicar a revisão podem ser maiores.

## Quando não usar CI?

Os proponentes de CI definem um limite rígido para integrações no master: pelo menos uma integração por dia por desenvolvedor. No entanto, dependendo da organização, do domínio do sistema (que pode ser um sistema crítico) e do perfil dos desenvolvedores (que podem ser iniciantes), pode ser difícil seguir esse limite.

Por outro lado, é bom lembrar que esse limite não é uma lei da natureza. Por exemplo, talvez seja mais factível realizar uma integração a cada dois ou três dias. Na verdade, qualquer prática de Engenharia de Software — incluindo integração contínua — não deve ser considerada ao pé da letra, isto é, exatamente como está descrita no manual ou neste livro-texto. Adaptações justificadas pelo contexto da organização são possíveis e devem ser consideradas. Experimentação com diferentes intervalos de integração pode também ajudar a definir a melhor configuração para uma organização.

CI também não é compatível com projetos de código livre. Na maioria das vezes, os desenvolvedores desses projetos são voluntários e não têm disponibilidade para trabalhar diariamente no seu código. Nesses casos, um modelo baseado em Pull Requests e Forks, conforme usado pelo GitHub, é mais adequado.

## Deployment Contínuo

Com integração contínua, código novo é frequentemente integrado no branch principal. No entanto, esse código não precisa estar pronto para entrar em produção. Ou seja, ele pode ser uma versão preliminar, que foi integrado para que os outros desenvolvedores tomem ciência da sua existência e, consequentemente, evitem conflitos de integração futuros. Por exemplo, você pode integrar uma versão preliminar de uma tela, com uma

interface ainda ruim. Ou então, uma versão de uma função com problemas de desempenho.

Porém, existe mais um passo da cadeia de automação proposta por DevOps, chamado de **Deployment Contínuo (Continuous Deployment ou CD)**. A diferença entre CI e CD é simples, mas seus impactos são profundos: quando usa-se CD, todo novo commit que chega no master entra rapidamente em produção, em questões de horas, por exemplo. O fluxo de trabalho quando se usa CD é o seguinte:

- O desenvolvedor desenvolve e testa na sua máquina local.
- Ele realiza um commit e o servidor de CI executa novamente um build e os testes de unidade.
- Algumas vezes no dia, o servidor de CI realiza testes mais exaustivos com os novos commits que ainda não entraram em produção. Esses testes incluem, por exemplo, testes de integração, testes de interface e testes de desempenho.
- Se todos os testes passarem, os commits entram imediatamente em produção. E os usuários já vão interagir com a nova versão do código.

Dentre as vantagens de CD, podemos citar:

- CD reduz o tempo de entrega de novas funcionalidades. Por exemplo, suponha que as funcionalidades F1, F2,..., Fn estão previstas para uma nova release de um sistema. No modo tradicional, todas elas devem ser implementadas e testadas, antes da liberação da nova release. Por outro lado, com CD, as funcionalidades são liberadas assim que ficam prontas. Ou seja, CD diminui o intervalo entre releases. Passa-se a ter mais releases, mas com um menor número de funcionalidades.
- CD torna novas releases (ou implantações) um não-evento. Explicando melhor, não existe mais um dia D ou um deadline para entrega de novas releases. Deadlines são uma fonte de stress para desenvolvedores e operadores de sistemas de software. A perda de um deadline, por

exemplo, pode fazer com que uma funcionalidade somente entre em produção meses depois.

- Além de reduzir o stress causado por deadlines, CD ajuda a manter os desenvolvedores motivados, pois eles não ficam meses trabalhando sem receber feedback. Em vez disso, os desenvolvedores rapidamente recebem retorno — vindo de usuários reais — sobre o sucesso ou não de suas tarefas.
- Em linha com o item anterior, CD favorece experimentação e um estilo de desenvolvimento orientado por dados e feedback dos usuários. Novas funcionalidades entram rapidamente em produção. Com isso, recebe-se retorno dos usuários, que podem recomendar mudanças na implementação ou, no limite, o cancelamento de alguma funcionalidade.

**Mundo Real:** Diversas empresas que desenvolvem sistemas Web usam CD. Por exemplo, Savor e colegas reportam que no Facebook cada desenvolvedor coloca em produção, na média, 3.5 atualizações de software por semana ([link](#)). Em cada atualização, na média, 92 linhas de código são adicionadas ou modificadas. Esses números revelam que, para funcionar bem, CD requer que as atualizações de código sejam pequenas. Portanto, os desenvolvedores têm que desenvolver a habilidade de quebrar qualquer tarefa de programação (por exemplo, uma nova funcionalidade, mesmo que complexa) em partes pequenas, que possam ser implementadas, testadas, integradas e entregues rapidamente.

## Entrega Contínua (Continuous Delivery)

Deployment Contínuo (CD) não é recomendável para certos tipos de sistemas, incluindo sistemas desktop (como uma IDE ou um navegador Web), aplicações móveis e aplicações embutidas em hardware. Provavelmente, você não gostaria de ser notificado diariamente de que existe uma nova versão do navegador que usa em seu desktop, ou do sistema de rede social que usa em seu celular ou ainda de que um novo driver está disponível para sua impressora. Esses sistemas demandam um processo de instalação que não é transparente para seus usuários, como é a atualização de um sistema Web.

No entanto, mesmo nos sistemas mencionados no parágrafo anterior, pode-se usar um versão mais *fraca* de CD, chamada de **Entrega Contínua (Continuous Delivery)**. A ideia é simples: quando se usa entrega contínua, todo commit *pode* entrar em produção imediatamente. Ou seja, os desenvolvedores devem programar como se isso fosse acontecer. No entanto, existe uma autoridade externa — um gerente de projetos ou de releases, por exemplo — que toma a decisão sobre quando os commits, de fato, serão liberados para os usuários finais. Inclusive forças de mercado ou de estratégia da empresa podem influenciar nessa decisão. Uma outra maneira de explicar esses conceitos é por meio da seguinte diferença:

- **Deployment** é o processo de liberar uma nova versão de um sistema para seus usuários.
- **Delivery** é o processo de liberar uma nova versão de um sistema para ser objeto de deployment.

Quando adota-se Deployment Contínuo, ambos os processos são automáticos e contínuos. Porém, com Entrega Contínua, a entrega é realizada com frequência, mas o deployment depende de uma autorização manual.

**Mundo Real:** Vamos citar alguns dados sobre a frequência de deployments em sistemas não-Web. Por exemplo, o Google libera novas releases do navegador Chrome para o público a cada seis semanas. Até 2019, a IDE Eclipse tinha uma única nova release por ano. A partir de 2019, o sistema passou a ter uma nova release a cada 13 semanas. Um dos motivos foi permitir que os desenvolvedores liberem novas funcionalidades de forma rápida. Como um terceiro exemplo, a versão para Android do Facebook sofria uma atualização a cada oito semanas. Mais recentemente, o Facebook encurtou esse tempo para uma semana ([link](#)). Ou seja, as empresas estão lançando releases de forma mais rápida, para agradar os usuários, receber feedback, manter os desenvolvedores motivados e continuarem competitivas no mercado.

## Feature Flags

Nem sempre todo commit estará pronto para entrar imediatamente em produção. Por exemplo, um desenvolvedor pode estar trabalhando em uma nova funcionalidade X, mas ainda falta implementar parte de seus requisitos. Portanto, esse desenvolvedor pode se perguntar:

Se novas releases são liberadas quase todo dia, como evitar que minhas implementações parciais, que ainda não foram devidamente testadas ou que têm problemas de desempenho, cheguem até os usuários finais?

Uma solução seria não integrá-las no branch principal de desenvolvimento. Porém, não queremos mais usar essa prática, pois ela leva ao que chamamos de *integration (ou merge) hell*. Dizendo de outro modo, não queremos abrir mão de Integração Contínua e Desenvolvimento Baseado no Trunk.

Uma solução para esse problema é a seguinte: integre continuamente o código parcial da funcionalidade X, mas com ela desabilitada, isto é, qualquer código relativo a X estará guardado por uma variável booleana (um *flag*) que, enquanto a implementação de X não estiver concluída, vai avaliar como falso. Um exemplo hipotético é mostrado a seguir:

```
featureX = false;  
...  
if (featureX)  
    "aqui tem código incompleto de X"  
...  
if (featureX)  
    "mais código incompleto de X"
```

No contexto de deployment contínuo, variáveis usadas para evitar a entrada em produção de implementações parciais de funcionalidades são chamadas de **Feature Flags** ou **Feature Toggles**.

Para mostrar um segundo exemplo, suponha que você está trabalhando em uma nova página de um certo sistema. Então, você pode declarar um feature flag para desabilitar o carregamento da nova página, como mostrado a seguir:

```
nova_pag = false;  
...  
if (nova_pag)  
    "carregue nova página"
```

```
else
    "carregue página antiga"
```

Esse é o código que vai para produção enquanto a nova página não estiver pronta. Porém, durante a implementação, localmente, na sua máquina, você pode habilitar a nova página, fazendo o flag `nova_pag` receber `true`. Observe ainda que durante um certo intervalo de tempo vai existir duplicação de código entre as duas páginas. Porém, após a nova página ser aprovada, entrar em produção e receber feedback positivo dos clientes, o código da página antiga e o feature flag (`nova_pag`) podem ser removidos. Ou seja, a duplicação de código foi temporária.

**Mundo Real:** Pesquisadores de duas universidades canadenses, liderados pelos professores Peter Rigby e Bram Adams, realizaram um estudo sobre o uso de feature flags ao longo de 39 releases do navegador Chrome, relativas a cinco anos de desenvolvimento ([link](#)). Nesse período, eles encontraram mais de 2.400 feature flags distintos no código do navegador. Na primeira versão analisada, eles catalogaram 263 flags; na última versão, o número aumentou para 2.409 flags. Na média, uma nova release adicionava 73 novos flags e removia 43 flags. Por isso, o crescimento observado no estudo.

No entanto, alguns feature flags podem ser mantidos no código durante o processo de release do software. Isso pode ocorrer por dois motivos, conforme descrito a seguir.

Primeiro, feature flags ajudam a implementar o que chama-se de **release canário**. Nessa modalidade de release, uma nova funcionalidade — guardada por um feature flag — é disponibilizada inicialmente para um grupo pequeno de usuários. Por exemplo, para apenas 5% dos usuários. Com isso, os prejuízos causados por eventuais bugs não detectados nos testes da nova funcionalidade serão minimizados. Em seguida, caso a implantação seja bem sucedida, pode-se ampliar a base de usuários que terá acesso à nova funcionalidade de forma gradativa, até alcançar todos os usuários do sistema. O nome *release canário* é uma referência a uma prática comum na exploração de novas minas de carvão. Os mineiros costumavam adentrar essas minas com um canário em uma gaiola. Caso a mina possuísse algum gás tóxico, ele mataria primeiro o canário e, então, os mineiros poderiam recuar e evitar uma intoxicação.

Adicionalmente, feature flags ajudam a viabilizar **Testes A/B**, tal como estudamos no Capítulo 3. Apenas para relembrar, nesses testes, libera-se simultaneamente duas versões de uma funcionalidade (antiga e nova, por exemplo) para grupos distintos de usuários, com o objetivo de verificar se a nova funcionalidade de fato agrega valor ao sistema.

Para facilitar a execução de releases canários e testes A/B, pode-se usar uma estrutura de dados para armazenar os flags e seu estado (ligado ou desligado). Um exemplo é mostrado a seguir:

```
FeatureFlagsTable fft = new FeatureFlagsTable();
fft.addFeature("novo-carrinho-compras", false);
...
if (fft.IsEnabled("novo-carrinho-compras"))
    // processa compra usando novo carrinho
else
    // processa compra usando carrinho atual
...
```

Existem bibliotecas dedicadas a gerenciar feature flags, as quais disponibilizam classes semelhantes a `FeatureFlagsTable` do código acima. A vantagem nesse caso é que os flags podem ser setados externamente ao código, por exemplo, em um arquivo de configuração. Por outro lado, quando o flag é uma variável booleana, para alterar seu valor precisa-se editar e recompilar o código.

**Aprofundamento:** Nesta seção, nosso foco foi no uso de feature flags para evitar a entrada em produção de um determinado trecho de código, em um cenário de deployment contínuo. Feature flags com esse propósito são chamados também de **release flags**. No entanto, feature flags podem ser usados com outros propósitos. Um deles é gerar diferentes versões de um mesmo sistema de software. Por exemplo, suponha um sistema que tenha uma versão gratuita e uma versão paga. Os clientes da versão paga têm acesso a mais funcionalidades, cujo código é delimitado por feature flags. Nesse caso específico, os flags são chamadas de **flags de negócio (business flags)**.

## Bibliografia

Gene Kim, Jez Humble, John Willis, Patrick Debois. Manual de DevOps. Como Obter Agilidade, Confiabilidade e Segurança em Organizações Tecnológicas. Alta Books, 2018.

Jez Humble, David Farley. Entrega Contínua: Como Entregar Software de Forma Rápida e Confiável. Bookman, 2014.

Steve Matyas, Andrew Glover, Paul Duvall. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley, 2007.

## Exercícios de Fixação

1. Defina e descreva os objetivos de DevOps.
2. Em sites de oferta de empregos na área de TI, é comum encontrar vagas para Engenheiro DevOps, requerendo habilidades como as seguintes:

- Ferramentas de controle de versão (Git, Bitbucket, SVN, etc.)
- Gerenciadores de dependência e build (Maven, Gradle, etc.)
- Ferramentas de integração contínua (Jenkins, Bamboo, VSTS)
- Administração de servidores em Cloud (AWS e Azure)
- Sistemas Operacionais (Ubuntu, CentOS e Red Hat)
- Banco de dados (DynamoDB, Aurora Mysql)
- Docker e orquestração de Docker (Kubernetes, Mesos, Swarm)
- Desenvolvimento com APIs REST, Java

Considerando a definição de DevOps que usou como resposta no exercício anterior, você considera adequado que a função de um funcionário seja Engenheiro DevOps? Justifique a sua resposta.

3. Descreva duas vantagens de um Sistema de Controle de Versões Distribuído (DVCS), como o git.
4. Descreva uma desvantagem relacionada com o uso de mono-repositórios.
5. Defina (e diferencie) os seguintes termos: integração contínua (*continuous integration*); entrega contínua (*continuous delivery*) e deployment contínuo

(*continuous deployment*).

6. Por que integração contínua, entrega contínua e deployment contínuo são práticas importantes em DevOps? Na sua resposta, considere a definição de DevOps que usou no primeiro exercício desta lista.
7. Pesquise o significado da expressão Teatro de CI (*CI Theater*) e então descreva-o com suas próprias palavras.
8. Suponha que você foi contratado por uma empresa que fabrica impressoras. E que você ficou responsável por definir as práticas de DevOps que serão adotadas no desenvolvimento dos *drivers* (software) dessas impressoras. Qual das seguintes práticas você adotaria nesse desenvolvimento: deployment contínuo ou delivery contínuo? Justifique sua resposta.
9. Descreva um problema (ou dificuldade) que surge quando usamos feature flags para delimitar código que ainda não está pronto para entrar em produção.
10. Linguagens como C possuem suporte a diretivas de compilação condicional do tipo `#ifdef` e `#endif`. Pesquise o funcionamento e o uso dessas diretivas. Qual a diferença entre elas e feature flags?
11. Qual tipo de feature flags possui maior tempo de vida (isto é, permanece no código por mais tempo): *release flags* ou *business flags*? Justifique sua resposta.
12. Quando uma empresa migra para CI, normalmente ela não usa mais branches de funcionalidades (*feature branches*). Em vez disso, ela tem um único branch, que é compartilhado por todos os desenvolvedores. Essa prática é chamada Desenvolvimento Baseado no Trunk (ou TBD), conforme estudamos neste capítulo. No entanto, TBD não significa que branches não são mais usados nessas empresas. Descreva então um outro uso para branches, que não seja como *feature branches*.
13. Leia o seguinte [artigo](#) do blog oficial do GMail, que descreve uma grande atualização realizada na interface do sistema, em 2011. O artigo

chega a comparar os desafios dessa migração com aqueles de trocar os pneus de um carro com ele em movimento. Sobre esse artigo, responda então:

1. Qual tecnologia — que estudamos neste capítulo — foi fundamental para viabilizar essa atualização na interface do GMail? Qual nome o artigo dá para essa tecnologia?
2. E qual nome usamos no capítulo para referenciá-la?

# Apêndice A - Git

Neste apêndice, apresentamos e discutimos exemplos de uso do sistema Git, que é o sistema de controle de versões mais usado atualmente. Inspirados pela frase acima, de Linus Torvalds, criador do Git, vamos focar nos conceitos e comandos básicos desse sistema. Como sugere a frase, é importante dominar esses comandos antes de se aventurar no uso de comandos mais avançados. Caso o leitor não tenha conhecimento dos objetivos e vantagens proporcionados por um sistema de controle de versões, recomendamos primeiro a leitura da seção Controle de Versões, do Capítulo 10 deste livro.

## Init & Clone

Para começar a usar o git para gerenciar as versões de um sistema devemos executar um dos seguintes comandos: **init** ou **clone**. O comando **init** cria um repositório vazio. O segundo comando — **clone** — primeiro chama **init** para criar um repositório vazio. Em seguida, ele copia para esse repositório todos os commits de um repositório remoto, passado como parâmetro. Seja, por exemplo, o seguinte comando:

```
git clone https://github.com/NOME-USER/NOME-REPO
```

Esse comando clona para o diretório corrente um repositório armazenado remotamente no GitHub. Portanto, devemos usar **clone** quando vamos trabalhar em um projeto que já está em andamento e que já possui commits em um repositório central. No exemplo, esse repositório é disponibilizado pelo GitHub.

## Commit

Commits são usados para criar snapshots (ou fotografias) dos arquivos de um sistema. Uma vez tiradas essas fotografias, elas são armazenadas no sistema de controle de versões, de forma compactada e otimizada, para não ocupar muito espaço em disco. Posteriormente, pode-se recuperar qualquer

uma das fotografias, para, por exemplo, restaurar uma implementação antiga de um determinado arquivo.

Recomenda-se que desenvolvedores realizem commits periodicamente, sempre que tiverem efetuado uma mudança importante no código. Em sistemas de controle de versões distribuídos, como o git, os commits são primeiro armazenados no repositório local do desenvolvedor. Por isso, o custo de um commit é pequeno e, portanto, desenvolvedores podem realizar diversos commits ao longo de um dia de trabalho. Na verdade, o que não é recomendável é a realização de commits grandes, com modificações importantes em diversos arquivos. Também não recomenda-se que um commit inclua modificações relativas a mais de uma tarefa de manutenção. Por exemplo, não é recomendável corrigir dois bugs em um mesmo commit. Em vez disso, cada bug deve ser corrigido em um commit separado. Assim, facilita-se uma futura análise do código, caso, por exemplo, um cliente volte a reclamar que seu bug não foi corrigido.

Commits também possuem metadados, incluindo data, hora, autor e uma mensagem, que descreve a modificação realizada pelo commit. A próxima figura mostra uma página do GitHub que exibe os metadados principais de um commit do repositório google/guava. Pode-se observar que o commit refere-se a um refactoring, o que fica claro no seu título. Em seguida, o refactoring é explicado em detalhes na mensagem do commit. Na última linha da figura, podemos ver o nome do autor do commit e a informação de que ele foi realizado há 13 dias.

## Refactor AbstractGraphTest to allow for tests with ImmutableGraph.

[Browse files](#)

Reason: ImmutableGraph implementations are undertested compared to MutableGraph implementations. The current tests didn't catch a bug I deliberately introduced in [] Also, it would be nice to be able to share the incident edge order tests.

Note about design: I started out by making an AbstractGraphTest subclass for mutable graphs. However, that would lead to a duplication of all 7 subclasses and most of their tests. The issue is that directed/undirected and mutable/immutable are orthogonal and we also want to test allowSelfLoops=true/false and incidentEdgeOrder=unordered/stable. The proposed solution is somewhat unconventional, but at least allows us to share much more code between the tests.

RELENOTES=n/a

---

Created by MOE: <https://github.com/google/moe>

MOE\_MIGRATED\_REVID=283035873

---

from master (#3726)



nymanjens authored and nglorioso committed 13 days ago

1 parent 2ee7f9d commit 1c757483665f0ba8fed31a2af7e31643a4590256

## Commit no GitHub

Na última linha da figura também podemos observar que todo commit possui um identificador único, no caso:

1c757483665f0ba8fed31a2af7e31643a4590256

Esse identificador possui 20 bytes, normalmente representados em hexadecimal. Esses bytes correspondem a uma verificação de consistência (*check sum*) do conteúdo do commit, conforme computado por uma função hash SHA-1.

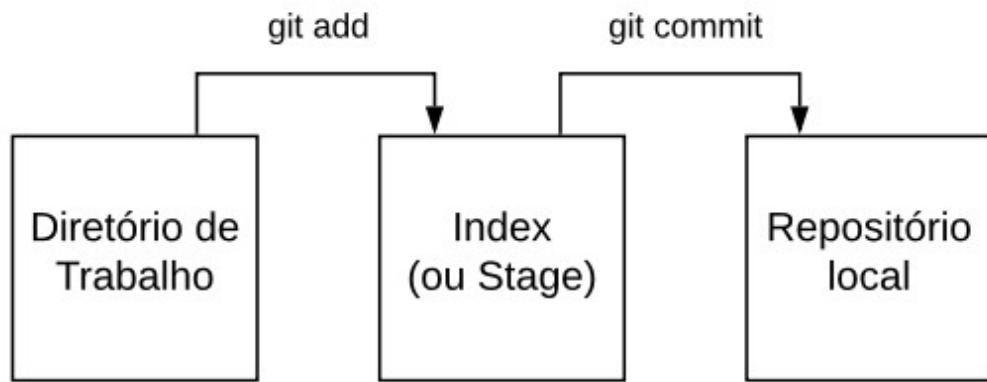
## Add

Na máquina local, o sistema git manipula três áreas distintas:

- Um **diretório de trabalho**, onde devemos salvar os arquivos que pretendemos versionar. Às vezes, essa área é chamada também de árvore de trabalho (*working tree*).
- O **repositório** propriamente dito, que armazena o histórico de commits.
- Uma área intermediária, chamada de **index** ou **stage**, que armazena temporariamente os arquivos que se pretende versionar. Tais arquivos são ditos rastreáveis (**tracked**).

Dentre essas três áreas, o desenvolvedor acessa apenas o diretório de trabalho, que funciona como um diretório comum do sistema operacional. As duas outras áreas são internas ao git e manipuladas exclusivamente por ele. Como qualquer diretório, o diretório de trabalho pode conter diversos arquivos. Porém, apenas aqueles adicionados ao index, por meio de um **add**, serão gerenciados pelo git.

Além de armazenar a lista de arquivos versionados, o index também armazena o conteúdo deles. Por isso, antes de fazer um commit devemos executar um add, para salvar o conteúdo do arquivo no index. Feito isso, podemos usar um commit para salvar no repositório local a versão adicionada ao index. Esse fluxo é ilustrado na próxima figura.



Comandos add e commit

**Exemplo:** Suponha o seguinte arquivo simples, mas suficiente para explicar os comandos add e commit.

```
// arq1
x = 10;
```

Após criar esse arquivo, o desenvolvedor executou o seguinte comando:

```
git add arq1
```

Esse comando adiciona o arquivo arq1 no index (ou stage). Porém, logo em seguida, o desenvolvedor modificou de novo o arquivo:

```
// arq1
x = 20; // novo valor de x
```

Feito isso, ele executou:

```
git commit -m "Alterando o valor de x"
```

A opção `-m` informa a mensagem que descreve o commit. Porém, o ponto que queremos ressaltar com esse exemplo é o seguinte: como o usuário não executou um novo add após mudar o valor de `x` para 20, a versão mais recente do arquivo não será salva pelo commit. Em vez disso, a versão de `arq1` que será versionada é aquela em que `x` tem o valor 10, pois ela é a versão que consta do index.

Para evitar o problema descrito nesse exemplo, é comum usar um commit da seguinte forma:

```
git commit -a -m "Alterando valor de x"
```

A opção `-a` indica que antes de executar o commit queremos adicionar no index todos os arquivos rastreados (*tracked*) que tenham sido modificados desde o último commit. Portanto, a opção `-a` não elimina a necessidade de usar `add`. O uso desse comando continua sendo necessário, pelo menos uma vez, para indicar ao git que desejamos tornar um determinado arquivo rastreável.

Da mesma forma que existe um `add`, também existe uma operação para remover um arquivo de um repositório git. Um exemplo é dado a seguir:

```
git rm arq1.txt  
git commit -m "Removendo arq1.txt"
```

Além de remover do repositório git local, o comando `rm` também remove o arquivo do diretório de trabalho.

## Status, Diff & Log

O comando `status` é um dos comandos git mais usados. Dentre outras informações, ele mostra o estado do diretório de trabalho e do index. Por exemplo, pode-se usar esse comando para obter informações sobre:

- Arquivos do diretório de trabalho que foram alterados pelo desenvolvedor, mas que ele ainda não adicionou no index.
- Arquivos do diretório de trabalho que não são rastreados pelo git, ou seja, eles ainda não foram objetos de um `add`.
- Arquivos que encontram-se no index, aguardando um `commit`.

O comando `git diff` é muito usado para destacar as modificações realizadas nos arquivos do diretório de trabalho e que ainda não foram movidas para o index (ou stage). Para cada arquivo modificado, ele mostra

as linhas que foram adicionadas (+) e removidas (-). Muitas vezes, usamos um `git diff` antes de um `add/commit` para ter certeza das mudanças que iremos perpetuar, em seguida, no sistema de controle de versões.

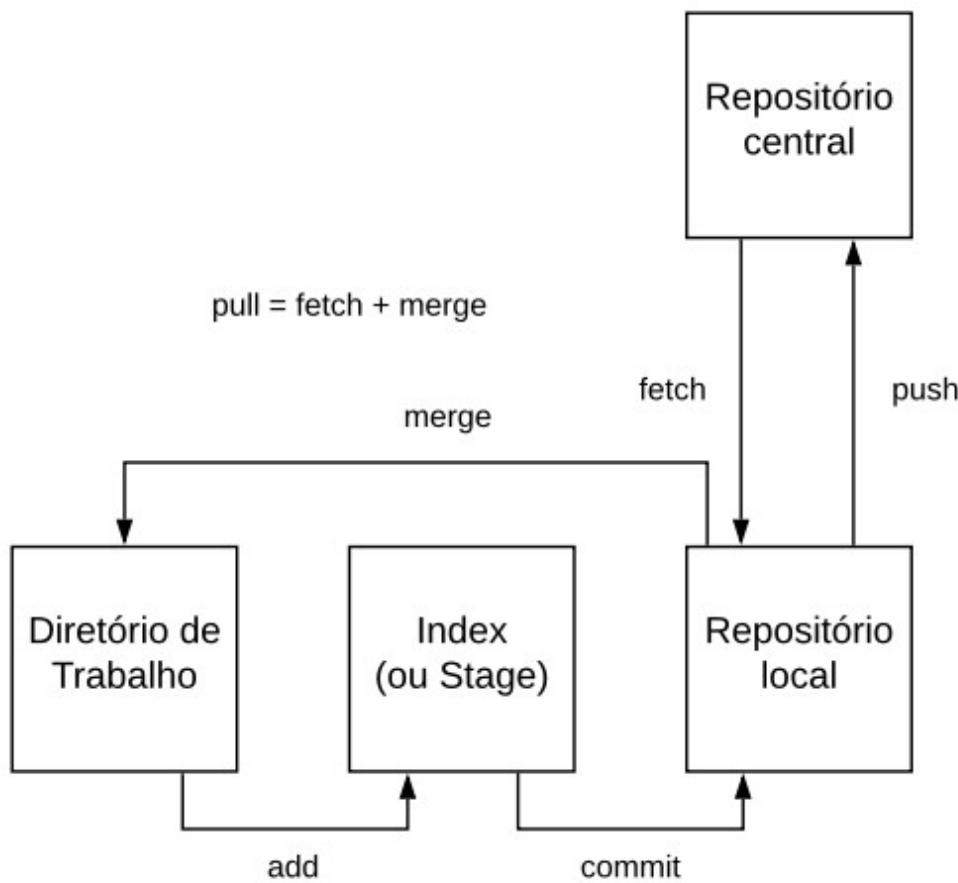
Já o comando `git log` lista informações sobre os últimos commits, como data, autor, hora e descrição do commit.

## Push & Pull

O comando **push** copia os commits mais recentes do repositório local para o repositório remoto. Portanto, ele é uma operação mais lenta, pois envolve comunicação pela rede. Um push deve ser usado quando o desenvolvedor deseja tornar uma modificação visível para os demais desenvolvedores. Para atualizar seu repositório local, os outros desenvolvedores do time devem executar um comando **pull**. Esse comando realiza duas operações principais:

- Primeiro, um `pull` copia os commits mais recentes do repositório central para o repositório local do desenvolvedor. Essa operação inicial é chamada de **fetch**.
- Em seguida, o comando `pull` atualiza os arquivos do diretório de trabalho. Essa operação é chamada de **merge**.

A próxima figura ilustra o funcionamento dos comandos `push` e `pull`.



## Comandos push e pull

**Exemplo:** Suponha que no repositório git central de um projeto exista o seguinte arquivo:

```
void f() {
    ...
}
```

Suponha que dois desenvolvedores, chamados Bob e Alice, realizaram um pull e, portanto, copiaram esse arquivo para o repositório local e para o diretório de trabalho de suas máquinas. A sintaxe desse comando é a seguinte:

```
git pull
```

No mesmo dia, Bob implementou uma segunda função g no arquivo:

```
void f() {    // antiga
    ...
}

void g() {    // implementada por Bob
    ...
}
```

Em seguida, Bob realizou um add, um commit e um push. Esse último comando tem a seguinte sintaxe:

```
git push origin master
```

O parâmetro `origin` é um valor default, usado pelo git, para indicar o repositório remoto, por exemplo, o repositório GitHub. Já o parâmetro `master` indica o branch principal. Iremos estudar mais sobre branches daqui a pouco.

Após executar o comando `push` acima, a nova versão do arquivo estará salva também no repositório remoto. Alguns dias depois, Alice decidiu que precisa alterar esse mesmo arquivo. Como ela ficou um tempo sem trabalhar no sistema, o recomendado é que ela execute primeiro um `pull`, para atualizar seu repositório local e seu diretório de trabalho com as mudanças ocorridas nesse período, como aquela realizada por Bob. Assim, após esse `pull`, o arquivo em questão será atualizado na máquina da Alice, para incluir a função `g` implementada por Bob.

## Conflitos de Merge

Conflitos de merge acontecem quando dois desenvolvedores alteram o mesmo trecho de código ao mesmo tempo. Para entender melhor essa situação, nada melhor do que usar um exemplo.

**Exemplo:** Suponha que Bob implementou o seguinte programa:

```
main() {
    print("Hello, world!");
}
```

Concluída a implementação, Bob realizou um add, seguido de um commit e um push.

Em seguida, Alice realizou um pull e obteve a versão do arquivo implementada por Bob. Então, Alice resolveu traduzir a mensagem do programa para português:

```
main() {  
    print("Olá, mundo!");  
}
```

Enquanto Alice fazia a tradução, Bob percebeu que escreveu Hello de forma errada, com apenas uma letra l. Porém, Alice foi mais rápida e realizou a trinca de comandos add, commit e push.

Bob, após corrigir o erro de ortografia, salvou o arquivo e também executou um add, seguido de um commit. Por fim, ele executou push, mas o comando falhou com a seguinte mensagem de erro:

Updates were rejected because the remote contains work that you do not have locally. This is usually caused by another repository pushing to the same ref. You may want to first integrate the remote changes (e.g., git pull ...) before pushing again.

A mensagem é bem clara: Bob não pode executar um push, pois o repositório remoto possui conteúdo novo, no caso, gerado por Alice. Antes de executar um push, Bob precisa executar um pull. Porém, ao fazer isso, ele recebe uma nova mensagem de erro:

CONFLICT (content): Merge conflict in arq2  
Automatic merge failed; fix conflicts and then commit the result.

Essa nova mensagem é também clara: existe um conflito de merge no arquivo arq2. Ao editar esse arquivo, Bob vai perceber que ele foi modificado pelo git, para destacar as linhas que geraram o conflito:

```
main() {  
<<<<< HEAD  
print("Hello, world!");  
=====
```

```
print("Olá, mundo!");
>>>>> f25bce8fea85a625b891c890a8eca003b723f21b
}
```

As linhas inseridas pelo git devem ser entendidas da seguinte forma:

- Entre <<<<< HEAD e ===== temos o código modificado por Bob, isto é, pelo desenvolvedor que não conseguiu dar um push e teve que dar um pull. HEAD designa que o código foi modificado no último commit realizado por Bob.
- Entre ===== e >>>>> f25bce8 ... temos o código modificado por Alice, isto é, pela desenvolvedora que executou com sucesso seu push. f25bce8... é o ID do commit no qual Alice modificou essa parte do código.

Cabe então a Bob resolver o conflito, o que é sempre uma tarefa manual. Para isso, ele tem que escolher o trecho de código que vai prevalecer — o seu código ou o da Alice — e editar o arquivo de acordo com tal escolha, para remover os delimitadores inseridos pelo git.

Vamos supor que Bob decida que o código de Alice é o certo, pois agora o sistema está usando mensagens em português. Logo, ele deve editar o arquivo, de forma que fique assim:

```
main() {
    print("Olá, mundo!");
}
```

Veja que Bob removeu os delimitadores inseridos pelo git (<<<<< HEAD , ===== e >>>>> f25bce8...). E também o comando `print` com a mensagem em inglês. Após deixar o código da forma correta, Bob deve executar novamente os comandos `add`, `commit` e `push`, que agora serão bem sucedidos.

Nesse exemplo, mostramos um conflito simples, que ficou restrito a única linha de um único arquivo. No entanto, um `pull` pode dar origem a conflitos mais complexos. Por exemplo, um mesmo arquivo pode apresentar vários conflitos. E também podemos ter conflitos em mais de um arquivo.

# Branches

O git organiza o diretório de trabalho em diretórios virtuais, chamados de **branches**. Até agora, não precisamos comentar sobre branches porque todo repositório possui um branch default, chamado de **master**, criado pelo comando `init`. Se não nos preocuparmos com branches, todo o desenvolvimento ocorrerá no master. Porém, em alguns casos, é interessante criar outros branches para melhor organizar o desenvolvimento. Para descrever o conceito de branches, vamos de novo usar um exemplo.

**Exemplo:** Suponha que Bob é responsável por manter uma determinada funcionalidade de um sistema. Para simplificar, vamos assumir que essa funcionalidade é implementada em uma única função `f`. Bob teve a ideia de mudar completamente a implementação de `f`, de forma que ela passe a usar algoritmos e estruturas de dados mais eficientes. Para isso, Bob vai precisar de algumas semanas. No entanto, apesar de estar otimista, Bob não tem certeza de que a nova implementação vai proporcionar os ganhos que ele imagina. Por fim, mas não menos importante, durante a implementação do novo código, Bob pode precisar do código original de `f`, para, por exemplo, corrigir bugs reportados pelos usuários.

Esse é um cenário interessante para Bob criar um branch para implementar e testar — de forma isolada — essa nova versão de `f`. Para isso, ele deve usar o comando:

```
git branch f-novo
```

Esse comando cria um novo branch, chamado `f-novo`, supondo que esse branch ainda não existe.

Para mudar do branch corrente para um novo branch, deve-se usar `git checkout [nome-branch]`. Para descobrir qual o nome do branch corrente, basta usar `git branch`. Na verdade, esse comando lista todos os branches e indica qual deles é o corrente.

Branches podem ser entendidos como “sub-diretórios virtuais” do diretório de trabalho. A principal diferença é que branches são gerenciados pelo git e não pelo sistema operacional. Por isso, optamos por chamá-los de virtuais.

Explorando mais essa comparação, podemos pensar que o comando `git branch [nome]` equivale ao comando `mkdir [nome]`, com a diferença que o git não apenas cria um branch mas copia para ele todos os arquivos do branch pai. Por outro lado, diretórios são criados vazios pelo sistema operacional. Já o comando `git checkout [nome]` lembra o comando `cd [nome]`. E `git status` lembra um misto de comandos `ls` e `pwd`. Também para reforçar essa comparação, existem certos comandos que permitem adicionar ao prompt do sistema operacional não apenas o nome do diretório corrente mas também o nome do branch corrente. Assim, o prompt pode ser exibido como `~/projetos/sistema/master>`.

Por outro lado, existe também uma diferença importante entre branches e diretórios. Um desenvolvedor somente pode alterar o branch corrente de A para B se as modificações que ele fez em A estiverem salvas. Isto é, se ele tiver realizado antes um `add` e `commit`. Caso ele tenha esquecido de chamar esses comandos, um comando `git checkout B` irá falhar com a seguinte mensagem de erro:

```
Your local changes to the following files would be overwritten by checkout:  
[list of files]  
Please commit your changes or stash them before you switch branches.
```

Voltando ao exemplo, após Bob ter criado o seu branch, ele deve proceder do seguinte modo. Quando ele quiser trabalhar na nova implementação de `f`, ele deve primeiro mudar o branch corrente para `f-novo`. Por outro lado, quando ele precisar modificar o código original de `f` — aquele que está em produção — ele deve se certificar de que o branch corrente é o `master`. Independentemente do branch em que estiver, Bob deve usar `add` e `commit` para salvar o estado do seu trabalho.

Bob vai continuar nesse fluxo, alternando entre os branches `f-novo` e `master`, até que a nova implementação de `f` esteja concluída. Quando isso acontecer, Bob vai precisar copiar o novo código de `f` para o código original. No entanto, como está usando branches, ele não precisa realizar essa operação de forma manual. O git oferece uma operação, chamada **merge**, que realiza exatamente essa cópia. A sintaxe é a seguinte:

```
git merge f-novo
```

Esse comando deve ser chamado no branch que irá receber as modificações realizadas em f-novo. No nosso caso, no branch master.

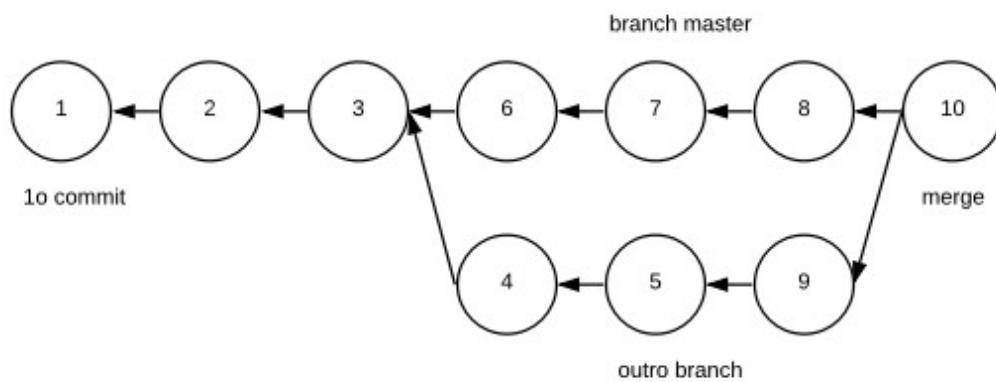
Como o leitor já deve estar pensando, um merge pode gerar conflitos, chamados também de **conflitos de integração**. No caso específico de merge de branches, esses conflitos vão ocorrer quando tanto o branch que está recebendo as modificações (master, no nosso exemplo) como o branch que está sendo integrado (f-novo, no exemplo) tiverem alterado os mesmos trechos de código. Conforme discutido na Seção A.6, o git irá delimitar os trechos com conflitos e caberá ao desenvolvedor que chamou o merge resolvê-los. Isto é, escolher o código que deve prevalecer.

Por fim, após realizar o merge, Bob pode remover o branch f-novo, caso não seja importante manter o histórico dos commits realizados para implementar a nova versão de f. Para deletar f-novo, ele deve executar o seguinte comando no master:

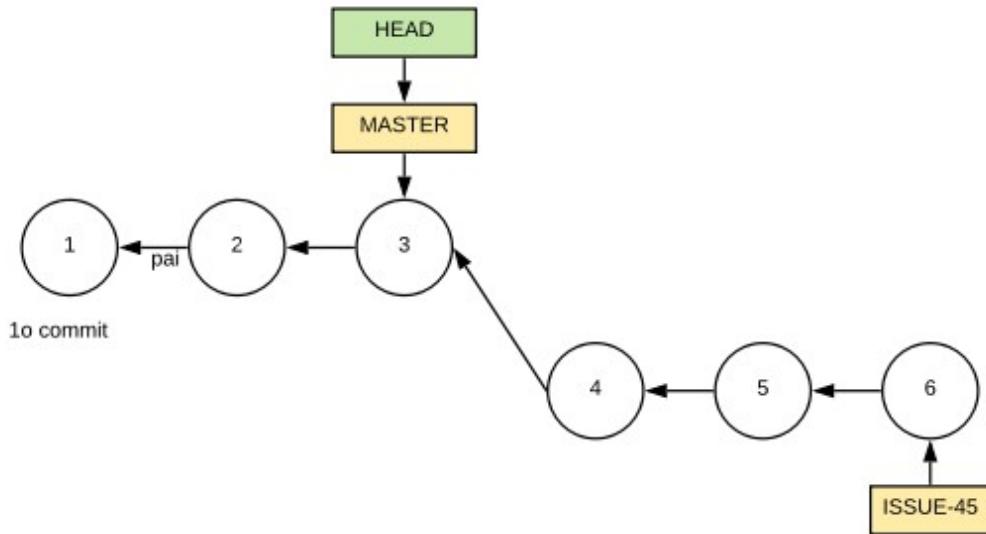
```
git branch -d f-novo
```

## Grafo de Commits

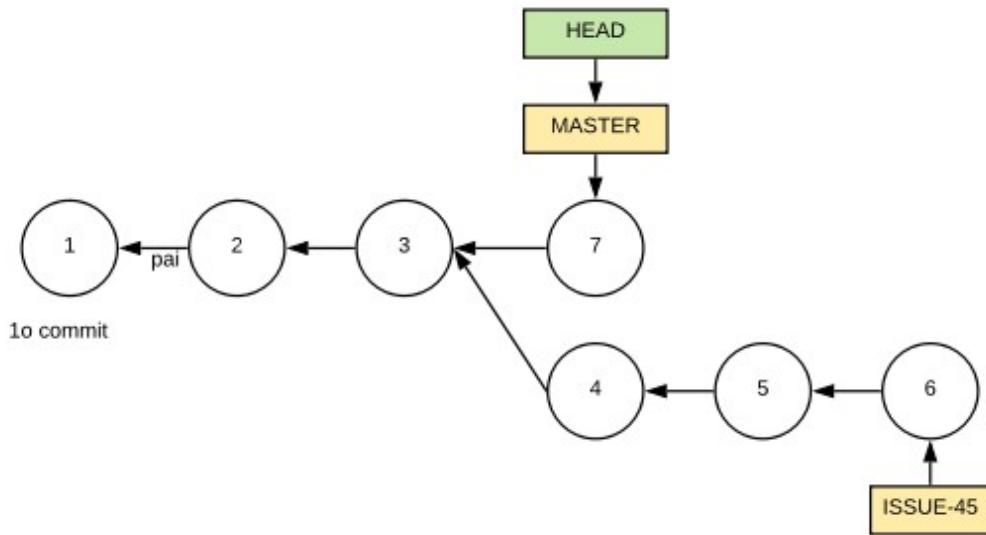
Commits podem possuir zero, um ou mais pais (ou antecessores). Como ilustra a próxima figura, o primeiro commit de um repositório não possui pai. Já um commit de merge possui dois ou mais pais, que representam os branches que foram unidos. Os demais commits possuem exatamente um nodo pai.



Um branch nada mais é do que uma variável interna do git que contém o identificador do último commit realizado no branch. Existe ainda uma variável chamada `HEAD`, que aponta para a variável do branch atual. Ou seja, `HEAD` contém o nome da variável que contém o identificador do último commit do branch atual. Um exemplo é mostrado a seguir:

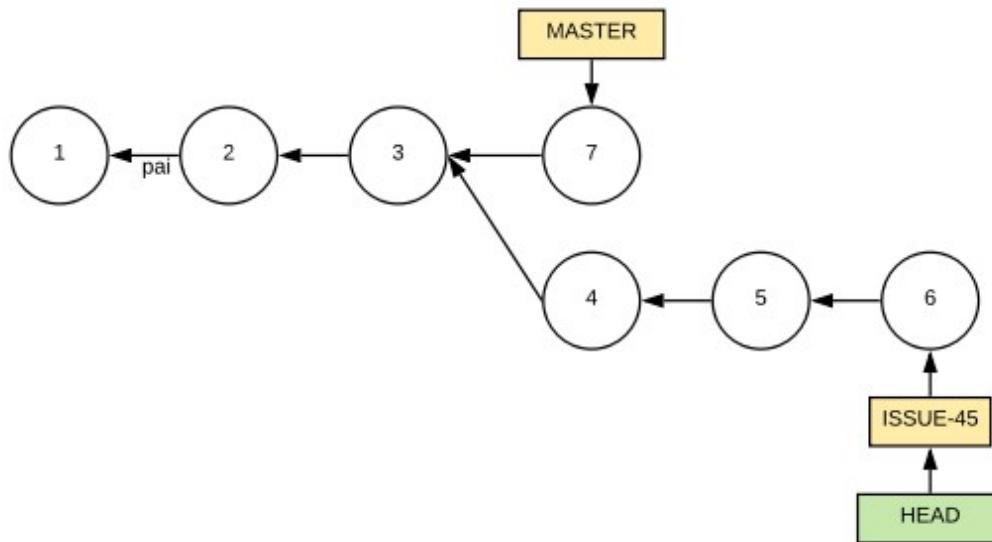


Nesse exemplo, existem dois branches, representados pelas variáveis `MASTER` e `ISSUE-45`. Cada uma delas aponta para o último commit de seu respectivo branch. A variável `HEAD` aponta para a variável `MASTER`. Isso significa que o branch atual é o `MASTER`. Se realizarmos um commit, a configuração mudará para:



O novo commit tem identificador 7. Ele foi realizado no `MASTER`, já que `HEAD` apontava para a variável desse branch. O pai do novo commit pai é o antigo `HEAD`, no caso o commit 3. A variável `MASTER` avançou e passou a apontar para o novo commit. Isso significa que, se não mudarmos de branch, o pai do próximo commit será o commit 7.

Porém, se mudarmos para o branch `ISSUE-45`, a configuração será aquela mostrada na próxima figura. A única mudança é que a variável `HEAD` passou a apontar para a variável do novo branch, isto é, para a variável `ISSUE-45`. Isso é suficiente para fazer com que o próximo commit seja realizado nesse branch, isto é, que ela tenha o commit 6 como pai.



## Branches Remotos

Até esse momento, trabalhamos com branches localmente, isto é, os branches que discutimos existem apenas no repositório local. No entanto, é possível realizar o push de um branch local para um repositório remoto. Para ilustrar esse recurso, vamos usar um exemplo semelhante ao da seção anterior.

**Exemplo:** Suponha que Bob criou um branch, chamado `g-novo`, para implementar uma nova funcionalidade. Ele realizou alguns commits nesse branch e agora gostaria de compartilhá-lo com Alice, para que ela

implemente parte da nova funcionalidade. Para isso, Bob deve usar o seguinte push:

```
git push -u origin g-novo
```

Esse comando realiza o push do branch corrente (g-novo) para o repositório remoto, chamado pelo git de `origin`. O repositório remoto pode, por exemplo, ser um repositório do GitHub. O parâmetro `-u` indica que, no futuro, vamos querer sincronizar os dois repositórios por meio de um `pull` (a letra do parâmetro vem da palavra *upstream*). Essa sintaxe vale apenas para o primeiro push de um branch remoto. Nos comandos seguintes, pode-se omitir o `-u`, isto é, usar apenas `git push origin g-novo`.

No repositório remoto, será criado um branch `g-novo`. Para trabalhar nesse branch, Alice deve primeiro criá-lo na sua máquina local, mas associado ao branch remoto, por meio dos seguintes comandos, que devem ser executados no `master`:

```
git pull
```

```
git checkout -t origin/g-novo
```

O primeiro comando é necessário para tornar o branch remoto visível na máquina local. Já o segundo comando cria um branch local, chamado `g-novo`, que Alice vai usar para rastrear mudanças no branch remoto `origin/g-novo`, conforme indica o parâmetro `-t`, que vem da palavra *tracking*. Em seguida, Alice pode realizar commits nesse branch. Por fim, quando estiver pronta para publicar suas mudanças, ela deve executar um `push`, com a sintaxe normal, isto é, sem o parâmetro `-u`.

Agora, Bob pode realizar um `pull`, concluir que a implementação da nova funcionalidade está finalizada e, portanto, pode ser integrada no `master`, por meio de um `merge`. Bob pode também deletar os branches local e remoto, usando os comandos:

```
git branch -d g-novo
```

```
git push origin --delete g-novo
```

E Alice também pode deletar seu branch local, chamando apenas:

```
git branch -d g-novo
```

## Pull Requests

Pull requests é um mecanismo que viabiliza que um branch seja revisado e discutido antes de ser integrado no branch principal. Quando se usa pull requests, um desenvolvedor sempre implementa novas funcionalidades em um branch separado. Concluída a implementação, ele não integra imediatamente o novo código no branch principal. Antes que isso ocorra, ele abre uma solicitação para que seu branch seja revisado e aprovado por um segundo desenvolvedor. Essa solicitação para revisão e integração de código é chamada de pull request. Trata-se de um mecanismo mais comum no GitHub, mas que possui equivalente em outros sistemas de controle de versões.

Modernamente, o processo de revisão e integração do código de um pull request ocorre via interface Web, provida, por exemplo, pelo GitHub. Porém, se essa interface não existisse, o revisor teria que começar o seu trabalho realizando um pull do branch para sua máquina local. Daí então a origem do nome, isto é, pull request é uma solicitação (*request*) para que um segundo desenvolvedor revise e integre um determinado branch. Para atender a essa solicitação, ele deve começar realizando um pull do branch.

A seguir, vamos detalhar o processo de submissão e revisão de pull requests por meio de um exemplo.

**Exemplo:** Suponha que Bob e Alice são membros de uma organização que mantém um repositório chamado awesome-git, com uma lista de links interessantes sobre git. Os links ficam armazenados no arquivo README.md desse repositório, isto é, na sua página principal, cujo endereço é [github.com/aserg-ufmg/awesome-git](https://github.com/aserg-ufmg/awesome-git). Qualquer membro da organização pode sugerir a adição de links nessa página. Mas veja que estamos usando a palavra sugerir. Isto é, eles não podem fazer um push diretamente no branch master. Em vez disso, a sugestão de link precisa ser revisada e aprovada por um outro membro do time.

Bob resolveu então sugerir a adição, nessa página, de um link para o presente apêndice do livro Engenharia de Software Moderna. Para isso, ele

primeiro clonou o repositório e criou um branch, chamado `livro-esm`, por meio dos seguintes comandos:

```
git clone https://github.com/aserg-ufmg/awesome-git.git  
git checkout livro-esm
```

Em seguida, Bob editou o arquivo `README.md`, adicionando a URL do apêndice. Por fim, ele realizou um `add`, um `commit` e fez um `push` do branch para o GitHub:

```
git add README.md  
git commit -m "Livro ESM"  
git push -u origin livro-esm
```

Na verdade, esses passos não são novidade em relação ao que vimos na seção anterior. No entanto, as diferenças começam agora. Primeiro, Bob deve ir na página do GitHub e selecionar o branch `livro-esm`. Feito isso, o GitHub mostrará um botão para criação de pull requests. Bob deve clicar nesse botão e descrever o seu pull request, como mostra a próxima figura.

The screenshot shows a GitHub pull request interface. At the top, there's a header with the title "Livro ESM". Below the header, there are two tabs: "Write" (which is selected) and "Preview". To the right of the tabs are various editing icons: bold (B), italic (i), quote ("), code (‘), backspace (⌫), list (list icon), numbered list (1, 2, 3 icon), checkmark (✓), at symbol (@), bookmark (star icon), and back arrow (left arrow icon). The main content area contains the text "Link para o apêndice sobre git do livro Engenharia de Software Moderna.". Below this text area, there's a dashed line followed by the instruction "Attach files by dragging & dropping, selecting or pasting them." On the far right of this line is a small "M+" icon. At the bottom right of the page is a green button labeled "Create pull request" with a dropdown arrow.

## Exemplo de pull request

Um pull request é uma solicitação para que um outro desenvolvedor revise e, se for o caso, realize o merge de um branch que você criou. Consequentemente, pull requests são um recurso para que uma organização passe a adotar **revisões de código**. Ou seja, desenvolvedores não integram diretamente o seu código no master do repositório remoto. Em vez disso, eles solicitam, via pull requests, que outros desenvolvedores revisem primeiro esse código e então façam o merge.

Na página do GitHub para criação de pull requests, Bob pode informar que deseja que seu código seja revisado pela Alice. Ela será então notificada que existe um pull request esperando sua revisão. Também via interface do GitHub, Alice pode revisar os commits do pull request criado por Bob, inclusive por meio de um diff entre o código novo e o código antigo. Se for o caso, Alice pode trocar mensagens com Bob, para esclarecer dúvidas sobre o novo código. Mais ainda, ela pode solicitar mudanças no código. Nesse caso, Bob deve providenciar as mudanças e realizar um novo add, commit e push. Então, os novos commit serão automaticamente anexados ao pull request, para que Alice possa conferir se o seu pedido foi atendido. Estando a modificação aprovada, Alice pode integrar o código no master, bastando para isso clicar em um dos botões da página de revisão de pull requests.

## Squash

Squash é um comando que permite unir diversos commits em um único commit. É uma operação recomendada, por exemplo, antes de submeter pull requests.

**Exemplo:** No exemplo anterior, suponha que o pull request criado por Bob tivesse cinco commits. Mais especificamente, ele está sugerindo o acréscimo de cinco novos links no repositório awesome-git, os quais foram coletados por ele ao longo de algumas semanas. Após a descoberta de cada link, Bob executou um commit na sua máquina local. E deixou para realizar o pull request apenas após acumular cinco commits.

Para facilitar a revisão de seu pull request por parte de Alice, Bob pretende unir esses cinco commits em um único commit. Assim, em vez de analisar cinco commits, Alice vai ter que analisar apenas um. Porém, a modificação submetida será exatamente a mesma, isto é, ela consiste na inclusão de cinco novos links na página. Porém, em vez de a solicitação estar distribuída em cinco commits (cada commit, adicionando um único link), ela estará concentrada em apenas um commit (adicionando cinco links).

Para realizar um squash, Bob deve chamar:

```
git rebase -i HEAD~5
```

O número 5 significa que pretende-se unir os cinco últimos commits do branch atual. Um editor de textos será aberto com uma lista contendo o ID e a descrição de cada um, como mostrado a seguir:

```
pick 16b5fcc Incluindo link 1
pick c964dea Incluindo link 2
pick 06cf8ee Incluindo link 3
pick 396b4a3 Incluindo link 4
pick 9be7fdb Incluindo link 5
```

Bob deve então usar o próprio editor para substituir a palavra pick por squash, exceto aquela da primeira linha. O arquivo ficará então assim:

```
pick 16b5fcc Incluindo link 1
squash c964dea Incluindo link 2
squash 06cf8ee Incluindo link 3
squash 396b4a3 Incluindo link 4
squash 9be7fdb Incluindo link 5
```

Bob deve então salvar o arquivo. Automaticamente, um novo editor será aberto, para ele informar a mensagem do novo commit — isto é, do commit que junta os cinco commits listados. Uma vez informada a mensagem, Bob deve salvar o arquivo e, então, o squash estará finalizado.

## Forks

Fork é o mecanismo que o GitHub oferece para clonar repositórios remotos, isto é, repositórios armazenados pelo próprio GitHub. Um fork é realizado via interface do GitHub. Na página de qualquer repositório, existe um botão para realizar essa operação. Se fizermos um fork do repositório torvalds/linux será criado uma cópia desse repositório na nossa conta do GitHub, chamado, por exemplo, mtov/linux.

Como fazemos sempre, vamos usar um exemplo para explicar essa operação.

**Exemplo:** Suponha o repositório [github.com/aserg-ufmg/awesome-git](https://github.com/aserg-ufmg/awesome-git), usado no exemplo sobre pull requests. Suponha ainda uma terceira desenvolvedora, chamada Carol. Porém, como Carol não é membro da organização ASERG/UFMG, ela não tem permissão para realizar push nesse repositório, como fez Bob no exemplo anterior. Apesar disso, Carol acha que

na lista atual falta um link importante e interessante, cuja inclusão ela gostaria de sugerir. Mas relembrando: Carol não pode seguir os mesmos passos usados por Bob no exemplo anterior, pois ela não tem permissão para dar push no repositório em questão.

Para resolver esse problema, Carol deve começar criando um fork do repositório. Para isso, basta clicar no botão fork, que existe na página de qualquer repositório no GitHub. Assim, Carol terá na sua conta do GitHub um novo repositório, cujo endereço será o seguinte: [github.com/carol/awesome-git](https://github.com/carol/awesome-git). Ela poderá clonar esse repositório para sua máquina local, criar um branch, adicionar o link que deseja na lista de links e realizar add, commit e push. Essa última operação será realizada no repositório resultante do fork. Por último, Carol deve ir na página do seu fork no GitHub e solicitar a criação de um pull request. Como o repositório é um fork, ela terá agora uma opção extra: destinar o pull request para o repositório original. Assim, caberá aos desenvolvedores do repositório original, como Bob e Alice, revisar e, se for o caso, aceitar o pull request.

Portanto, fork é um mecanismo que, quando combinado com pull requests, viabiliza que um projeto de código aberto receba contribuições de outros desenvolvedores. Explicando um pouco melhor, um projeto de código aberto pode receber contribuições — mais especificamente, commits — não apenas de seu time de desenvolvedores (Bob e Alice, no nosso exemplo), mas de um outro desenvolvedor com conta no GitHub (como é o caso de Carol).

## Bibliografia

- Scott Chacon; Ben Straub. Pro Git. 2a edição, Apress, 2014.
- Rachel M. Carmena. How to teach Git. Blog post ([link](#)).

## Exercícios de Fixação

Neste apêndice, mostramos diversos exemplos. Tente reproduzir cada um deles. Nos exemplos que envolvem repositórios remotos, a sugestão é usar um repositório do GitHub. Nos exemplos que envolvem dois usuários (Alice

e Bob, por exemplo), a sugestão é criar dois diretórios locais e usá-los para reproduzir os comandos de cada usuário.

Este livro foi formatado pelo autor usando o sistemas Pandoc, para conversão de Markdown para LaTeX e, em seguida, geração de um arquivo PDF. A fonte usada é Computer Modern, 11pt. A partir dos mesmos arquivos Markdown são geradas as versões MOBI (Kindle) e HTML.

# Vertopal.com

1. [Vertopal.com](#)

2. [Prefácio](#)

1. [Público-Alvo](#)

2. [Pré-requisitos](#)

3. [Website](#)

3. [Cap 1 Introdução](#)

1. [Definições, Contexto e História](#)

1. [Não existe bala de prata](#)

2. [O que se Estuda em Engenharia de Software?](#)

1. [Engenharia de Requisitos](#)

2. [Projeto de Software](#)

3. [Construção de Software](#)

4. [Testes de Software](#)

5. [Manutenção e Evolução de Software](#)

6. [Gerência de Configuração](#)

7. [Gerência de Projetos](#)

8. [Processos de Desenvolvimento de Software](#)

9. [Modelos de Software](#)

10. [Qualidade de Software](#)

11. [Prática Profissional](#)

12. [Aspectos Econômicos](#)

3. [Classificação de Sistemas de Software](#)

4. [Próximos Capítulos](#)

5. [Bibliografia](#)

6. [Exercícios de Fixação](#)

4. [Cap 2 Processos](#)

1. [Importância de Processos](#)

2. [Manifesto Ágil](#)

3. [Extreme Programming](#)

1. [Valores](#)

2. [Princípios](#)

3. [Práticas sobre o Processo de Desenvolvimento](#)

4. [Práticas de Programação](#)

5. [Práticas de Gerenciamento de Projetos](#)
4. [Scrum](#)
  1. [Papéis](#)
  2. [Principais Artefatos e Eventos](#)
  3. [Outros Eventos](#)
  4. [Exemplo: Escrita de um Livro](#)
  5. [Perguntas Frequentes](#)
5. [Kanban](#)
  1. [Calculando os Limites WIP](#)
  2. [Lei de Little](#)
  3. [Perguntas Frequentes](#)
6. [Quando Não Usar Métodos Ágeis?](#)
7. [Outros Métodos Iterativos](#)
8. [Bibliografia](#)
9. [Exercícios de Fixação](#)
5. [Cap 3 Requisitos](#)
  1. [Introdução](#)
  2. [Engenharia de Requisitos](#)
    1. [O que Vamos Estudar?](#)
  3. [Histórias de Usuários](#)
    1. [Exemplo: Sistema de Controle de Bibliotecas](#)
    2. [Perguntas Frequentes](#)
  4. [Casos de Uso](#)
    1. [Diagramas de Casos de Uso](#)
    2. [Perguntas Frequentes](#)
  5. [Produto Mínimo Viável \(MVP\)](#)
    1. [Exemplos de MVP](#)
    2. [Perguntas Frequentes](#)
    3. [Construindo o Primeiro MVP](#)
  6. [Testes A/B](#)
    1. [Perguntas Frequentes](#)
  7. [Bibliografia](#)
  8. [Exercícios de Fixação](#)
6. [Cap 4 Modelos](#)
  1. [Modelos de Software](#)
  2. [UML](#)
    1. [Como usar UML?](#)

- 2. [Diagramas UML](#)
- 3. [Diagrama de Classes](#)
  - 1. [Associações](#)
  - 2. [Herança](#)
  - 3. [Dependências](#)
- 4. [Diagrama de Pacotes](#)
- 5. [Diagrama de Sequência](#)
- 6. [Diagrama de Atividades](#)
- 7. [Bibliografia](#)
- 8. [Exercícios de Fixação](#)
- 7. [Cap 5 Princípios de Projeto](#)
  - 1. [Introdução](#)
    - 1. [Exemplo](#)
    - 2. [O Que Vamos Estudar?](#)
  - 2. [Integridade Conceitual](#)
  - 3. [Ocultamento de Informação](#)
    - 1. [Exemplo](#)
    - 2. [Getters e Setters](#)
  - 4. [Coesão](#)
    - 1. [Exemplos](#)
  - 5. [Acoplamento](#)
    - 1. [Exemplos](#)
  - 6. [SOLID e Outros Princípios de Projeto](#)
    - 1. [Princípio da Responsabilidade Única](#)
    - 2. [Princípio da Segregação de Interfaces](#)
    - 3. [Princípio de Inversão de Dependências](#)
    - 4. [Prefira Composição a Herança](#)
    - 5. [Princípio de Demeter](#)
    - 6. [Princípio Aberto/Fechado](#)
    - 7. [Princípio de Substituição de Liskov](#)
  - 7. [Métricas de Código Fonte](#)
    - 1. [Tamanho](#)
    - 2. [Coesão](#)
    - 3. [Acoplamento](#)
    - 4. [Complexidade](#)
  - 8. [Bibliografia](#)
  - 9. [Exercícios de Fixação](#)

## 8. [Cap 6 Padrões de Projeto](#)

1. [Introdução](#)
2. [Fábrica](#)
3. [Singleton](#)
4. [Proxy](#)
5. [Adaptador](#)
6. [Fachada](#)
7. [Decorador](#)
8. [Strategy](#)
9. [Observador](#)
10. [Template Method](#)
11. [Visitor](#)
12. [Outros Padrões de Projeto](#)
13. [Quando Não Usar Padrões de Projeto?](#)
14. [Bibliografia](#)
15. [Exercícios de Fixação](#)

## 9. [Cap 7 Arquitetura](#)

1. [Introdução](#)
  1. [Debate Tanenbaum-Torvalds](#)
2. [Arquitetura em Camadas](#)
  1. [Arquitetura em Três Camadas](#)
3. [Arquitetura MVC](#)
  1. [Exemplo: Single Page Applications](#)
4. [Microsserviços](#)
  1. [Gerenciamento de Dados](#)
  2. [Quando Não Usar Microsserviços?](#)
5. [Arquiteturas Orientadas a Mensagens](#)
  1. [Exemplo: Empresa de Telecomunicações](#)
6. [Arquiteturas Publish/Subscribe](#)
  1. [Exemplo: Companhia Aérea](#)
7. [Outros Padrões Arquiteturais](#)
8. [Anti-padrões Arquiteturais](#)
9. [Bibliografia](#)
10. [Exercícios de Fixação](#)

## 10. [Cap 8 Testes](#)

1. [Introdução](#)
2. [Testes de Unidade](#)

1. [Definições](#)
2. [Quando Escrever Testes de Unidade?](#)
3. [Benefícios](#)
3. [Princípios e Smells](#)
  1. [Princípios FIRST](#)
  2. [Test Smells](#)
  3. [Número de assert por Teste](#)
4. [Cobertura de Testes](#)
  1. [Qual a Cobertura de Testes Ideal?](#)
  2. [Outras Definições de Cobertura de Testes](#)
5. [Testabilidade](#)
  1. [Exemplo: Servlet](#)
  2. [Exemplo: Chamada Assíncrona](#)
6. [Mocks](#)
  1. [Frameworks de Mocks](#)
  2. [Mocks vs Stubs](#)
  3. [Exemplo: Servlet](#)
7. [Desenvolvimento Dirigido por Testes \(TDD\)](#)
  1. [Exemplo: Carrinho de Compras](#)
8. [Testes de Integração](#)
  1. [Exemplo: Agenda de Compromissos](#)
9. [Testes de Sistema](#)
  1. [Exemplo: Teste de Sistemas Web](#)
  2. [Exemplo: Teste de um Compilador](#)
10. [Outros Tipos de Testes](#)
  1. [Testes Caixa-Preta e Caixa-Branca](#)
  2. [Seleção de Dados de Teste](#)
  3. [Testes de Aceitação](#)
  4. [Testes de Requisitos Não-Funcionais](#)
11. [Bibliografia](#)
12. [Exercícios de Fixação](#)
11. [Cap 9 Refactoring](#)
  1. [Introdução](#)
  2. [Catálogo de Refactorings](#)
    1. [Inline de Método](#)
    2. [Movimentação de Método](#)
    3. [Renomeação](#)

- 4. [Outros Refactorings](#)
  - 3. [Prática de Refactoring](#)
  - 4. [Refactorings Automatizados](#)
    - 1. [Verificação de Pré-condições de Refactorings](#)
  - 5. [Code Smells](#)
    - 1. [Código Duplicado](#)
    - 2. [Métodos Longos](#)
    - 3. [Classes Grandes](#)
    - 4. [Feature Envy](#)
    - 5. [Métodos com Muitos Parâmetros](#)
    - 6. [Variáveis Globais](#)
    - 7. [Obsessão por Tipos Primitivos](#)
    - 8. [Objetos Mutáveis](#)
    - 9. [Classes de Dados](#)
    - 10. [Comentários](#)
  - 6. [Bibliografia](#)
  - 7. [Exercícios de Fixação](#)
12. [Cap 10 DevOps](#)
- 1. [Introdução](#)
  - 2. [Controle de Versões](#)
    - 1. [Multirepos vs Monorepos](#)
  - 3. [Integração Contínua](#)
    - 1. [Motivação](#)
    - 2. [O que é Integração Contínua?](#)
    - 3. [Boas Práticas para Uso de CI](#)
    - 4. [Quando não usar CI?](#)
  - 4. [Deployment Contínuo](#)
    - 1. [Entrega Contínua \(Continuous Delivery\)](#)
    - 2. [Feature Flags](#)
  - 5. [Bibliografia](#)
  - 6. [Exercícios de Fixação](#)
13. [Apêndice A - Git](#)
- 1. [Init & Clone](#)
  - 2. [Commit](#)
  - 3. [Add](#)
  - 4. [Status, Diff & Log](#)
  - 5. [Push & Pull](#)

6. [Conflitos de Merge](#)
7. [Branches](#)
  1. [Grafo de Commits](#)
8. [Branches Remotos](#)
9. [Pull Requests](#)
10. [Squash](#)
11. [Forks](#)
12. [Bibliografia](#)
13. [Exercícios de Fixação](#)