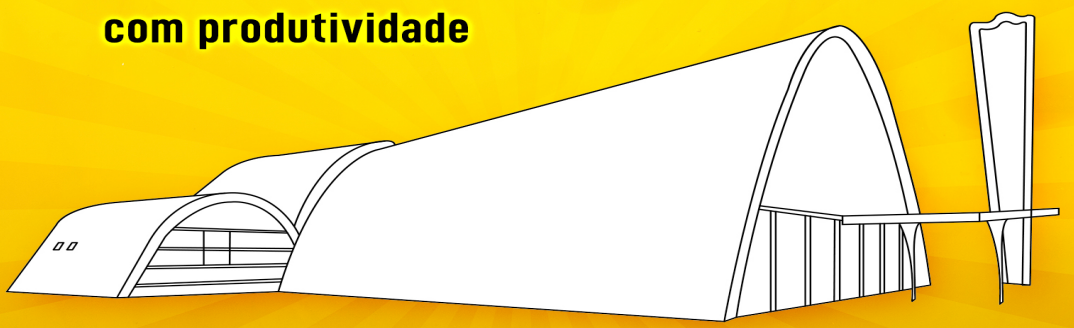
ENGENHARIA DE SOFTWARE MODERNA

Princípios e práticas para desenvolvimento de software com produtividade



MARCO TULIO VALENTE

Versão 2020.1.0 - LeanPub

Para Cynthia, Daniel e Mariana.

Conteúdo

Capítulo 1

Requisitos

> *The hardest single part of building a software system is deciding precisely what to build.* – Frederick Brooks

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 usuário, as quais são os principais instrumentos de Engenharia de Requisitos quando se usa Métodos Ágeis de Desenvenvolvimento. 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 falar 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 largamente usados hoje em dia para validar e definir os requisitos de produtos de software.

1.1 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 esse 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 de programação, usando o melhor processo de desenvolvimento, com alta cobertura de testes, etc e ele não atender às necessidades e restrições de seus usuários.

Por isso, problemas na especificação de requisitos têm um custo alto. Eles podem, por exemplo, implicar em trabalho extra, quando se descobre — após o sistema ter sido implementado — 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. 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.

Requisito Não-Funcional	**Métrica**
Desempenho	Transações por segundo, tempo de resposta, latência, vazão (throughput)
Espaço	Uso de disco, uso de RAM, uso de cache

Requisito Não-Funcional	Métrica
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 novos usuários
Portabilidade	% de linhas de código que são portáveis entre plataformas

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.

1.2 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.

Em Engenharia de Requisitos, as atividades relacionadas com a descoberta e entendimento dos requisitos de um sistema são chamadas de **Elicitação de Requisitos**. Segundo o Dicionário Houaiss, elicitar (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 (*etnos*, 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 exige-se um **Documento de Especificação de Requisitos**, onde 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 uma padrão para Documentos de Especificação de Requisitos, denominado **Padrão IEEE 830**. Ele foi proposto no contexto de Processos Waterfall, onde o desenvolvimento inicia-se com uma longa fase de levantamento de requisitos. As principais seções de um documento de requisitos no padrão IEEE 830 são as seguintes:

- Requisito 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

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 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 certo 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 a informações do negócio (18%)

1.2.1 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 críticos — 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 organização de certificação, principalmente no caso de sistemas que lidam com vidas humanas, como sistemas das áreas médicas, de transporte ou militar. Na Seção 3.4, iremos estudar **Casos de Uso**, que são documentos bastante 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.

1.3 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 de três partes, todas começando com a letra C e que vamos documentar usando a seguinte equação:

História de Usuário = Cartão + Conversas + 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 logo 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 usuário, 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 **sucintas**. Na verdade, até admite-se 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 sucintas 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 pensar e listar os principais usuários que vão interagir com o sistema. Assim, evita-se que as histórias fiquem enviesadas e cubram as necessidades de apenas certos tipos de usuários. Definidos esses **papéis de usuários** (*user roles*), costuma-se escrever as histórias no seguinte modelo:

> Como um [papel de usuário], eu gostaria de [realizar alguma coisa 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.

1.3.1 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. 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 layout 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 as novas aquisições da biblioteca

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 as novas aquisições da biblioteca

Em seguida, mostramos as histórias propostas por professores. É importante mencionar que, de fato, os professores foram os usuários que lembraram de requisitar as histórias a seguir. Eles podem ter feito isso, por exemplo, em um workshop de escrita de histórias. Mas isso não implica 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.

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 da universidade

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 que estão estragados	Como funcionário da biblioteca, eu gostaria de obter estatísticas sobre o acervo atual
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 sprint review quando se usa Scrum.

Pesquisa por livros, informando ISBN
Pesquisa por livros, informando autor; retorna livros cujo autor contém a string de busca
Pesquisa por livros, informando título; retorna livros cujo título contém a string de busca
Pesquisa 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**.

1.4 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 usuário. 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 para realizar um determinado objetivo. Tipicamente, esse ator é um usuário humano (embora, mais raramente, possa ser um outro sistema de software ou hardware). Em qualquer caso, o importante é que os atores sejam entidades externas 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 onde tudo dá certo, às vezes chamado também de "fluxo feliz". Já a segunda lista inclui **extensões ao 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 exemplo de caso de uso, referente a um sistema bancário, e que especifica uma transferência entre contas, realizada por um cliente do banco.

Transferir Valores entre Contas
Ator: Cliente do Banco
Fluxo normal:
1. Autenticar Cliente
2. Cliente informa agência e conta de destino da transferência
3. Ciente 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:
2. a. Se conta e agência incorretas, solicitar nova conta e agência
3. a. Se valor acima do saldo atual, solicitar novo valor
4. a. Data informada deve ser a data atual ou no máximo um ano a frente
5. a. Se data informada é a data atual, transferência é imediata
6. b. Se data informada é uma data futura, sistema agenda 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 de 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, vamos descrever algumas boas práticas de 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 9 passos no fluxo normal. Ele afirma literalmente o seguinte:

"eu raramente encontro um caso de uso bem escrito com mais de 9 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 um produto que pretenda 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 onde os usuários e desenvolvedores se referem às mesmas

coisas usando nomes diferentes e, pior ainda, se referem a coisas diferentes pelo mesmo nome".

1.4.1 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 a seguir. 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. Um 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 (sprints) 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 90, por Ivar Jacobson, um dos pais da UML e também do Processo Unificado (UP). Mais especificamente, casos de uso foram concebidos para ser um dos principais produtos da fase de Elaboração do UP. Conforme afirmamos no Capítulo 2, métodos como o UP dão ênfase à comunicação escrita entre usuários e desenvolvedores, na forma de documentos como casos de uso.

1.5 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. Em outras palavras, se é para um sistema falhar — por não ter sucesso, usuários, mercado, etc — 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 restringe-se 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**). Às vezes, 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. Nesse caso, repete-se o ciclo, voltando para o passo construir.
- Pode-se concluir que o teste foi bem sucedido e que 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 features.

- Por fim, pode-se concluir que que o MVP falhou, após várias tentativas. Nesse caso, 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 acionáveis** (*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 acionável — 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 também **métricas de funil** (*funnel metrics*), que capturam o nível de interação dos usuários com um sistema. Por exemplo, um "funil" pode incluir as seguintes métricas:

- Aquisição: número de clientes que visitaram o seu sistema.
- Ativação: número de clientes que, por exemplo, criaram uma conta no sistema.
- Retenção: número de clientes que retornaram ao sistema, após criarem uma conta.
- Receita: número de clientes que fizeram uma compra e, portanto, geraram receita.

- Recomendação: número de clientes que recomendaram o sistema para outros clientes.

1.5.1 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, onde 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. 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.

1.5.2 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 Web frequentemente estava fora do ar.

1.5.3 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 tem 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.

1.6 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 uso por grupos distintos de usuário. Ao final do teste, decide-se qual versão despertou maior interesse desses usuários. Portanto, testes A/B consistem em 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 layouts 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 dos elementos 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 também 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 compra 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 muito 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 usar essa versão 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 significância 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, não podemos ter certeza de que esse ganho é causado pelo tratamento B. Por isso, 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.

1.6.1 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 aleatórias, 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 20

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 onde os dois grupos, controle e tratamento, executam a mesma versão do sistema. Logo, assumindo-se uma significância 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 tomadas em um teste A/B. Alguns autores chegam a recomendar que não se deve iniciar testes A/B ante se realizar alguns testes A/A (link). Caso os testes A/A não fahem, 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 elicitar requisitos de forma antecipada e escrever especificações. Adicionalmente, testes A/B permitem detectar cenários onde os usuários começam a usar novas features de modo

inesperado. Dentre outras coisas, isso permite que os engenheiros aprendam com a diversidade de usuários e apreciem as diferentes visões que tais usuários têm do Facebook." (link)
Na Netflix, "os desenvolvedores tratam cada funcionalidade como um experimento, o que faz com 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. 3rd Edition. Wiley, 2008.

Exercícios de Fixação

1. [POSCOMP 2010, adaptado] Sobre Engenharia de Requisitos, marque V ou 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, entre 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? Responsa usando o acrônimo 3C's.

4. Suponha uma rede social como o Facebook. (1) Escreva um conjunto de 5 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 5 anti-patterns para histórias de usuário. 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).

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

Comprar Livro
Ator: 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 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 onde 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 no Google) 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 decidiu começar com um MVP. (a) Quais seriam as features desse MVP? (b) Ele seria desenvolvido para quais plataformas de hardware e sistemas operacionais? (c) Elabore um rascunho rápido da sua interface com o usuário. (d) 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 à segunda questão, use uma calculadora de tamanho de amostras de testes A/B, como aquela que citamos na Seção 3.6.