ENGENHARIA DE SOFTWARE

ENSINO A DISTÂNCIA

uniAvan
Centro Universitário Avantis

Ficha catalográfica elaborada na fonte pela Biblioteca do
Centro Universitário Avantis - UNIAVAN
Maria Helena Mafioletti Sampaio. CRB 14 – 276

Bombasar, James Roberto.
B699e Engenharia de software. /EAD/ [Caderno pedagógico].
James Roberto Bombasar. Balneário Camboriú: Faculdade Avantis, 2019.
101 p. il.

Inclui Índice
ISBN: 978-85-5456-221-2
ISBNe: 978-85-5456-220-5

1. Engenharia de software - Fundamentos. 2. Software.
3. Software - Ensino a Distância. I. Centro Universitário Avantis - UNIAVAN. II. Título.

CDD 21ª ed.
621.39 – Engenharia de Software.

PLANO DE ESTUDOS

EMENTA

Fundamentos da Engenharia de Software. Processos e Paradigmas de Desenvolvimento de Software. Documentação de Software. Teste de Software.

OBJETIVOS DA DISCIPLINA

- Compreender os conceitos fundamentais da engenharia de software;
- Compreender a dinâmica dos principais modelos genéricos de processo de desenvolvimento de software;
- Identificar modelos de processo adequados para o desenvolvimento de software em diferentes cenários;
- Aplicar modelos de processo no desenvolvimento de software;
- Compreender a importância da documentação de software;
- Identificar documentações úteis e necessárias em diferentes cenários de desenvolvimento de software;
- Produzir documentos de software;
- Compreender a importância dos testes de software para o desenvolvimento e a qualidade de software;
- Identificar tipos e estratégias de testes adequados em diferentes cenários de desenvolvimento de software;
- Projetar testes de softwares.

O PAPEL DA DISCIPLINA PARA A FORMAÇÃO DO ACADÊMICO

Engenharia de software é uma área da ciência da computação voltada ao estudo de metodologias e técnicas para a especificação, o desenvolvimento e a manutenção de software. Nesse sentido, o caderno de estudos de Engenharia de Software pretende ser seu aliado no desenvolvimento de habilidades e competências necessárias para a análise e o desenvolvimento de diferentes tipos de software, de maneira produtiva e organizada.

Você terá a oportunidade de conhecer os principais métodos e as ferramentas utilizados no desenvolvimento de softwares, incluindo modelos de processos, documentos e testes de software. O papel do engenheiro de software é tão importante quando o papel de qualquer outro profissional da área, pois é um dos principais responsáveis por garantir que processos de desenvolvimento de software sejam capazes de entregar produtos de qualidade e em conformidade com o que é esperado pelos clientes e usuários.

Bons estudos!

PROFESSOR

APRESENTAÇÃO DO AUTOR

JAMES ROBERTO BOMBASAR

Olá! Meu nome é James Roberto Bombasar, mas sou também conhecido como Beto, devido ao meu segundo nome. Sou bacharel em Sistemas de Informação, especialista em Ensino a Distância: docência e tutoria e mestre em Computação Aplicada.

Já atuei como programador, conteudista e analista de sistemas. Atualmente, trabalho no Centro Universitário Avantis (UniAvan) como docente no Ensino Superior.

Estou muito grato em poder compartilhar com você os conhecimentos que adquiri ao longo das minhas trajetórias acadêmica e profissional.

Pronto para começar?

Currículo Lattes: http://lattes.cnpq.br/4943270066505311
E-mail: james.bombasar@uniavan.edu.br

SUMÁRIO

unidade 1

FUNDAMENTOS DA ENGENHARIA DE SOFTWARE

1 INTRODUÇÃO À UNIDADE

Compreender os conceitos básicos da engenharia de software, sejam voltados para o desenvolvimento de produtos simples ou complexos, é o primeiro passo para que você seja capaz de atuar no processo de desenvolvimento de softwares. Como surgiu a engenharia de software? Qual a importância da engenharia de software? Qual o papel do engenheiro de software? Quais são as atividades básicas da engenharia de software? Esses são alguns exemplos de questionamentos que serão respondidos ao longo da primeira unidade do caderno de estudos.

Assim, o objetivo de aprendizagem da primeira unidade é compreender os conceitos fundamentais da engenharia de software. Serão explorados a crise e os mitos de software, as principais definições da área, os métodos, as ferramentas e o processo da engenharia de software, bem como o papel do engenheiro de software e as principais atividades por ele realizadas.

1.1 FUNDAMENTOS DA ENGENHARIA DE SOFTWARE

De acordo com Ferreira (2001, p. 289), o termo "engenharia" pode ser definido como a "aplicação de conhecimentos científicos e empíricos, e certas habilitações específicas, à criação de estruturas, dispositivos e processos para converter recursos naturais em formas adequadas ao atendimento das necessidades humanas". Em outras palavras, a engenharia pode ser entendida como a habilidade de transformar a natureza, por meio do uso de técnicas e ferramentas para resolver problemas.

Ao longo da história da humanidade, à medida que os problemas que surgiam se tornavam mais complexos, mais a engenharia necessitava de conhecimentos científicos. Agostinho, Amorelli e Barbosa (2015) relatam que a grande revolução na engenharia ocorreu a partir do século XVI; quando muitos problemas começaram a ser tratados de maneira racional, os artesãos deram espaço aos engenheiros e o sistema de aprendizado se voltou cada vez mais para o conteúdo científico.

Alguns marcos históricos que demonstram a crescente importância da engenharia ao longo da história foram a criação da primeira escola de engenharia das Américas no Brasil, em 1792, a fundação da Academia Militar de West Point nos EUA, em 1802, e a criação da Sociedade Profissional dos Engenheiros Civis na Inglaterra, em 1818 (AGOSTINHO; AMORELLI; BARBOSA, 2015).

A história da engenharia é bastante extensa, poderíamos falar sobre diversas outras áreas que surgiram para o atendimento das necessidades humanas, tais como a engenharia elétrica, a engenharia ambiental e a engenharia de produção. Contudo, o objetivo dessa breve contextualização histórica foi apenas chamar a sua atenção para a importância do conhecimento científico para as áreas de engenharia.

Agora, vamos direcionar o foco para a área da computação, na qual a necessidade de abordar problemas de desenvolvimento de software de maneira mais racional surgiu no final do século passado. Convido você a conhecer um pouco dessa história na próxima seção.

1.2 A CRISE DO SOFTWARE

Na segunda metade do século XX, houve uma grande evolução na área da computação, notadamente com o surgimento dos microprocessadores comerciais e computadores pessoais, que democratizaram o acesso à computação (TECMUNDO, 2016). Naturalmente, a popularização dos computadores veio acompanhada de um rápido crescimento da demanda por softwares. No entanto, a inexistência de ferramentas e métodos estabelecidos para o desenvolvimento de softwares fez com que projetos começassem a entregar softwares de má qualidade, estourando prazos e orçamentos. A esse fenômeno foi dado o nome de **crise do software**.

Figura 1: A crise do software.
Fonte:https://image.shutterstock.com/image-vector/cartoon-stick-figure-drawing-conceptual-450w-1381485443.jpg. Acesso em 09 setembro de 2019.

A noção da crise do software surgiu no final da década 60. Uma das primeiras referências ao termo foi feita por Dijkstra (1972), no artigo "The Humble Programmer" (O Programador Humilde). Apesar dos esforços da comunidade científica ao longo das últimas décadas para estabelecer ferramentas e métodos para o desenvolvimento de softwares, muitos projetos ainda enfrentam problemas na atualidade.

Para você ter uma ideia, o The Standish Group é uma empresa internacional independente de consultoria em pesquisa de Tecnologia da Informação (TI), conhecida por seus relatórios sobre projetos de implementação de sistemas de informação (THE STANDISH GROUP, 2019). O relatório "CHAOS Report 2015", produzido pelo The Standish Group em 2015, revela que mesmo após mais de 40 anos do marco inicial da crise do software, muitos projetos ainda foram desafiados ou cancelados (Figura 2).

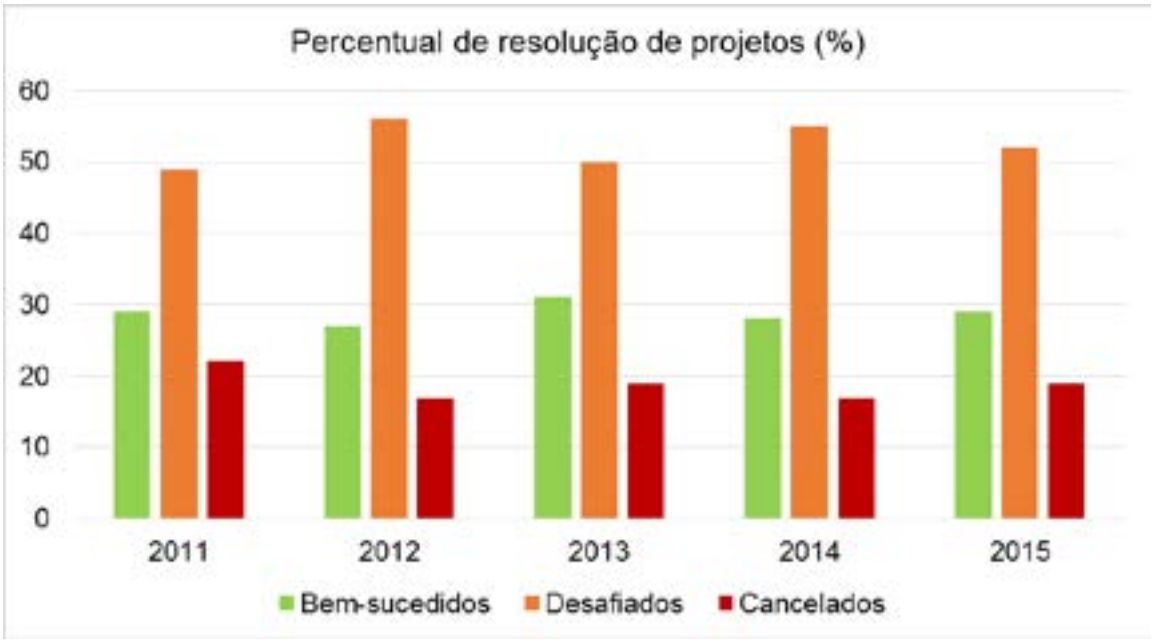


Figura 2: Percentual de resolução de projetos de implementação de sistemas de informação entre os anos de 2011 e 2015.
Fonte: Elaborado pelo autor com dados estraídos do CHAOS Report 2015 (THE STANDISH GROUP, 2015).

Segundo a definição do The Standish Group (2019), um projeto bem-sucedido é aquele completado no prazo e orçamento previstos, com todas os requisitos. Os projetos desafiados são aqueles concluídos com atraso, custo acima do estimado ou recursos e resultados aquém do especificado. Por sua vez, os projetos cancelados são aqueles interrompidos em algum ponto do desenvolvimento.

PARA REFLETIR

As causas que levam projetos de software a serem desafiados ou cancelados podem estar ligadas tanto à complexidade do processo de software quanto à imaturidade dos profissionais da área. Pesquise três exemplos de causas que podem levar projetos de software ao insucesso. Podem ser utilizados sites na Internet.

Nesta seção, você viu que a crise de software foi um fenômeno de fundamental importância para o surgimento da engenharia de software. Você viu também que até hoje projetos de desenvolvimento de software enfrentam problemas relacionados com prazos, orçamentos e qualidade. Neste momento, é natural que você esteja fazendo a seguinte pergunta:

Como evitar a crise de software?

Uma das soluções está no uso dos modelos, métodos e técnicas que serão apresentados ao longo deste caderno de estudos. Mas também é necessário que você, como futuro profissional da área, evolua seu paradigma sobre **o que é desenvolver um software**. Vamos iniciar essa evolução dando uma olhada em algumas definições de software e engenharia de software?

1.3 DEFINIÇÕES DE SOFTWARE E ENGENHARIA DE SOFTWARE

Antes de mais nada, é preciso estabelecer uma distinção clara entre os termos **programa**, **sistema** e **software**, que muitas vezes são utilizados como sinônimos, mas não são. A esta altura do curso, você já deve ter a noção de que um algoritmo é uma sequência finita de passos bem definidos que visa à solução de um determinado tipo de problema. Pois bem, quando um algoritmo é formalizado em uma linguagem de programação, para que possa ser executado por um computador, torna-se um programa. Assim, podemos dizer que um **programa** é um conjunto de instruções que descrevem ao computador como realizar uma tarefa.

Um **sistema** é tradicionalmente definido como um conjunto de "partes" inter-

relacionadas que interagem entre si para atingir a um objetivo. Desta forma, quando falamos em sistemas computacionais, estamos falando em um conjunto de programas ou softwares que trabalham em conjunto. É importante lembrar que o termo sistema não é utilizado somente na área da computação, existem sistemas de governo, sistemas biológicos, sistemas de pesquisa, dentre outros.

Já um **software** não é apenas um programa ou conjunto de programas, inclui também as estruturas de dados que possibilitam aos programas funcionar, bem como as documentações que descrevem a operação e o uso desses programas (PRESSMAN, 2016; SOMMERVILLE, 2011).

Por sua vez, a **engenharia de software** é uma disciplina que abrange um processo e um arcabouço de métodos e ferramentas para o desenvolvimento de softwares (PRESSMAN, 2016). Ela atua desde a especificação do software até sua manutenção e está presente em todos os aspectos do processo de desenvolvimento (SOMMERVILLE, 2011), conforme ilustra a Figura 3.

Figura 3: A engenharia de software e os aspectos do desenvolvimento de software.
Fonte: Adaptado pelo autor a partir de https://image.shutterstock.com/image-vector/software-engineering-chart-keywords-icons-450w-429107533.jpg. Acesso em 09 setembro de 2019.

Pressman (2016) explica que a engenharia de software está relacionada com a produção de softwares de qualidade dentro do cronograma e do orçamento. Esses dois últimos aspectos são tratados com ênfase na disciplina de Gerência de Projetos. No entanto, os

engenheiros de software também reconhecem que devem trabalhar dentro de limitações organizacionais e financeiras, então buscam soluções que atendam a essas limitações, mas com o foco voltado para a qualidade de software, conforme veremos na próxima seção.

SAIBA MAIS

"Software legado" e "**artefato de software"** são termos bastante utilizados na área de engenharia de software. Pesquise ao menos duas definições de cada um desses termos. Podem ser utilizados sites na Internet.

1.4 PROCESSO, MÉTODOS E FERRAMENTAS

Agora que você já sabe que a engenharia de software é uma disciplina que abrange um processo e um arcabouço de métodos e ferramentas, vamos entender como esses elementos se inter-relacionam entre si para atingir o objetivo de produzir softwares de qualidade.

E vamos iniciar fazendo uma analogia simples: quando um engenheiro civil constrói um uma casa ou prédio, segue um **modelo de processo** composto de atividades coordenadas, tais como fundação, estrutura, instalações, revestimentos e acabamentos. Ele também emprega **métodos**, que determinam de que modo cada atividade será executada. Para a construção da estrutura, por exemplo, podem ser empregados o método da alvenaria convencional ou o método da alvenaria estrutural, que dispensa o uso de vigas e pilares para a sustentação da obra. E, por fim, o engenheiro faz o uso de **ferramentas**, que fornecem suporte tanto para o processo quanto para os métodos.

Esse mesmo raciocínio se aplica quando estamos "construindo" um software. Utilizamos um **processo**, cujas atividades podem variar conforme as características do modelo, um conjunto de **métodos** (práticas) para executar essas atividades e um leque de **ferramentas** que fornece suporte automatizado ou semiautomatizado para os métodos e o processo (PRESSMAN, 2016). De acordo com Pressman (2016), todos esses elementos fazem parte de uma tecnologia em camadas que tem como pedra fundamental o comprometimento com a qualidade, conforme ilustrado na Figura 4.

Figura 4: Camadas da engenharia de software.
Fonte: Adaptado pelo autor a partir de Pressman (2016).

A camada de **processo** representa o conjunto de atividades relacionadas que levam à produção de um produto de software; ela estabelece o contexto no qual são aplicados métodos e possibilita o desenvolvimento de software de forma racional e dentro do prazo, servindo de base para o controle do gerenciamento de projetos (PRESSMAN, 2016; SOMMERVILLE, 2011). As atividades podem envolver tanto o desenvolvimento de novos softwares quanto a modificação ou evolução de softwares existentes.

De acordo com Sommerville (2011), existem muitos modelos de processo de software, mas todos contemplam no mínimo as quatro atividades fundamentais apresentadas na Figura 5.

Figura 5: Atividades fundamentais da engenharia de software.
Fonte: Adaptado pelo autor a partir de Sommerville (2011).

A atividade de **especificação** de software cuida de definir as funcionalidades do software (o que o software deve fazer) e as restrições a seu funcionamento. Existe uma subárea da engenharia de software, denominada engenharia de requisitos, que cuida especificamente da produção de documentos de requisitos e da manutenção desses documentos ao longo do tempo.

A atividade de **projeto e implementação** cuida de "construir" o software, ela envolve atividades como modelagem e programação. Neste ponto, é importante fazer uma breve distinção entre os termos "implementar" e "implantar". O primeiro significa "realizar", "fazer", e o segundo "fixar", "introduzir". Assim, na engenharia de software, o termo "implementação" é utilizado para se referir à construção do software, e o termo "implantação" para se referir à colocação do software em funcionamento no ambiente de produção.

Como você já sabe, a engenharia de software tem o comprometimento com a qualidade, e a qualidade de um produto de software está diretamente relacionada com o atendimento aos requisitos. Assim, a **validação** é uma atividade fundamental dentro do processo de desenvolvimento de software; é ela que cuida de garantir que as demandas do cliente sejam atendidas.

Um erro comum é pensar que o trabalho de desenvolvimento está terminado quando o software é colocado em uso. Muitos dos esforços da engenharia de software são consumidos após a entrega da primeira versão de um software, que deve evoluir para atender às necessidades de mudança do cliente. Por isto, a **evolução** também é uma atividade fundamental da engenharia de software.

Existem diferentes **modelos de processo** de desenvolvimento de software. Cada modelo define o seu próprio conjunto de atividades e como elas se organizam e se inter-relacionam ao longo do processo. Dedicaremos a Unidade 2 ao estudo desses modelos. Na próxima seção, vamos descobrir "quem é e onde vive" o engenheiro de software.

EXERCÍCIO

Segundo Pressman (2016), as atividades metodológicas da engenharia de software são complementadas por outras atividades chamadas de atividades de apoio, que auxiliam a equipe de a controlar o progresso, a qualidade, as mudanças e o risco. Pesquise quais são as atividades de apoio relacionadas no livro "Engenharia de Software: uma abordagem profissional", de Roger S. Pressman.

1.4 O PAPEL DO ENGENHEIRO DE SOFWARE

O profissional da engenharia de software está contemplado na Classificação Brasileira de Ocupações (CBO) pelos códigos 2122-15 (Engenheiro de software computacional básico) e 2122-05 (Engenheiro de sistemas computacionais) (MINISTÉRIO DO TRABALHO, 2019). A CBO, instituída com base legal na Portaria nº 397, de 10 de outubro de 2002, é um documento que visa expor as diversas atividades profissionais existentes em todo o país, sem diferenciar as profissões regulamentadas e as de livre exercício profissional (PORTAL EMPREGA BRASIL, 2019).

Para efeito de fiscalização do exercício profissional, o Conselho Federal de Engenharia e Agronomia, por meio da Resolução nº 1.100, de 24 de maio de 2018, inseriu o título de engenheiro (a) de software na Tabela de Títulos Profissionais do Sistema Confea/Crea, no grupo ou categoria Engenharia, modalidade Eletricista (CONSELHO FEDERAL DE ENGENHARIA E AGRONOMIA, 2018).

O Art. 2º da resolução nº 1.100, de 24 de maio de 2018 discriminou as atividades e competências profissionais do engenheiro de software da seguinte forma:

> Compete ao engenheiro de software as atribuições previstas no art. 7° da Lei nº 5.194, de 1966, combinadas com as atividades 1 a 18 do art. 5º, §1º, da Resolução nº 1.073, de 19 de abril de 2016, referentes a requisitos de software, sistemas e soluções de software, evolução de software, integração local e remota de sistemas de software (CONSELHO FEDERAL DE ENGENHARIA E AGRONOMIA, 2018, p. 239).

Dentre as atribuições previstas na Lei nº 5.194 e na Resolução nº 1.073, encontram-se atividades de estudo, pesquisa, treinamento, especificação, coleta de dados, projeto, padronização, produção técnica, análise, planejamento, execução, gestão, supervisão, coordenação, orientação técnica, auditoria, controle de qualidade, fiscalização, mensuração, assistência, assessoria e consultoria. Como você pode perceber, inúmeras são as atribuições de um engenheiro(a) de software, de forma que precisa estar presente ao longo de todo o processo de desenvolvimento de software.

Mas não é somente com as atribuições que um engenheiro de software deve se preocupar. Ele precisa exercer as atividades de maneira benéfica e responsável, em concordância com a segurança, saúde e o bem-estar da sociedade; conforme prevê o Código de Ética e de Prática Profissional da Engenharia de Software de Gotterbarn, Miller e Rogerson (1997), um padrão no ensino e na prática da engenharia de software, aprovado

segundo as recomendações da IEEE Computer Society (IEEE-CS) e da Association for Computing Machinery (ACM), e elaborado com base nos oitos princípios apresentados na Figura 6.

Figura 6: Os oito princípios da ética na engenharia de software.
Fonte: Adaptado pelo autor a partir de Gotterbarn, Miller e Rogerson (1997).

De uma forma geral, os primeiros princípios do Código de Ética de Gotterbarn, Miller e Rogerson (1997) estabelecem que os engenheiros de software devem atuar balanceando os seus próprios interesses com os interesses do **empregador**, do **cliente** e do bem-estar **público;** ou seja, garantindo a segurança do software e a comunicação sobre eventuais perigos que possa oferecer para o usuário, o público ou o meio ambiente.

Os princípios da **gestão** e do **produto** estabelecem que os engenheiros de software devem garantir a boa gestão dos projetos e a qualidade de seus produtos e suas modificações, com custos aceitáveis e prazos razoáveis. No ambiente de trabalho, os engenheiros de software devem ser justos e apoiar seus **colegas**, mantendo integridade e independência nos seus **julgamentos** profissionais. Por fim, o código de ética estabelece que o engenheiro de software deve promover o seu lado **pessoal**, participando de maneira contínua no aprendizado de sua profissão e promovendo uma abordagem ética na prática da mesma.

Em um ambiente de trabalho com tantas atividades e responsabilidades, é natural que surjam crenças e mal-entendidos entre as pessoas envolvidas no processo de desenvolvimento de software. Então, é hora de entrarmos na onda do programa de televisão MythBusters e desvendarmos alguns mitos relativos ao software.

1.5 MITOS RELATIVOS AO SOFTWARE

De acordo com Pressman (2016), os mitos criados em relação ao software e ao processo usado para desenvolvê-lo algumas vezes pode conter alguns elementos de verdade, e por isso são promulgados até mesmo por praticantes experientes. Alguns "resíduos" de mitos da engenharia de software permanecem até hoje, pois alguns hábitos e atitudes são difíceis de serem modificados.

Pressman (2016) classifica os mitos relativos ao software em: mitos dos clientes, mitos de gerenciamento e mitos dos profissionais da área. Os mitos dos clientes estão fortemente relacionados com a ansiedade pelo uso do software, muitas vezes potencializada pelas necessidades operacionais das empresas, e também pela falta de conhecimento dos clientes acerca da complexidade do processo de desenvolvimento de software, conforme ilustra a Figura 7.

Figura 7: Mitos dos clientes.
Fonte: Adaptado pelo autor a partir de Pressman (2016) e imagens de https://elements.envato.com/50-professionals-flat-multicolor-icons-SXSHQQ. Acesso em 13 setembro de 2019.

Quanto mais tardia for a mudança dos requisitos de um software, maior é o impacto sobre o prazo e os custos de desenvolvimento. O mesmo impacto ocorre, por exemplo, quando uma pessoa compra um apartamento em um edifício que está em construção. Se ela solicitar alguma alteração no layout do apartamento durante a fundação do edifício, não haverá um impacto significativo nos custos e na execução da obra. No entanto, se a mesma alteração for solicitada quando o edifício estiver em fase de acabamento, a construtora terá custos adicionais e provavelmente entregará a unidade ao cliente com atraso.

Os mitos de gerenciamento (Figura 8) normalmente surgem em função das pressões do mercado pela entrega de softwares de qualidade com preço bom e prazo curto. Os livros de engenharia de software fornecem muitos padrões genéricos, que precisam ser adaptados a cada projeto. É claro que uma equipe que desenvolve sempre o mesmo tipo de software pode ter seus próprios padrões bem estabelecidos. No entanto, esta maturidade pode levar vários anos para ser alcançada.

Para que a contratação de novos programadores seja capaz de acelerar o processo de desenvolvimento de software, é necessário que a equipe trabalhe com padrões muito bem estabelecidos. Esses padrões devem permitir, por exemplo, que diferentes partes de um mesmo programa possam ser desenvolvidas separadamente e integradas de maneira consistente. Algumas atividades da engenharia de software são fáceis de serem decompostas, mas outras não. Além disso, novos programadores precisam ser introduzidos de forma planejada e coordenada pelas pessoas que já estão trabalhando no projeto, e isso consome tempo.

Outro mito de gerenciamento é o de que a terceirização de projetos de software é um "mar de rosas". Segundo Pressman (2016), qualquer organização enfrentará dificuldades ao terceirizar um projeto se não souber gerenciá-lo e controlá-lo.

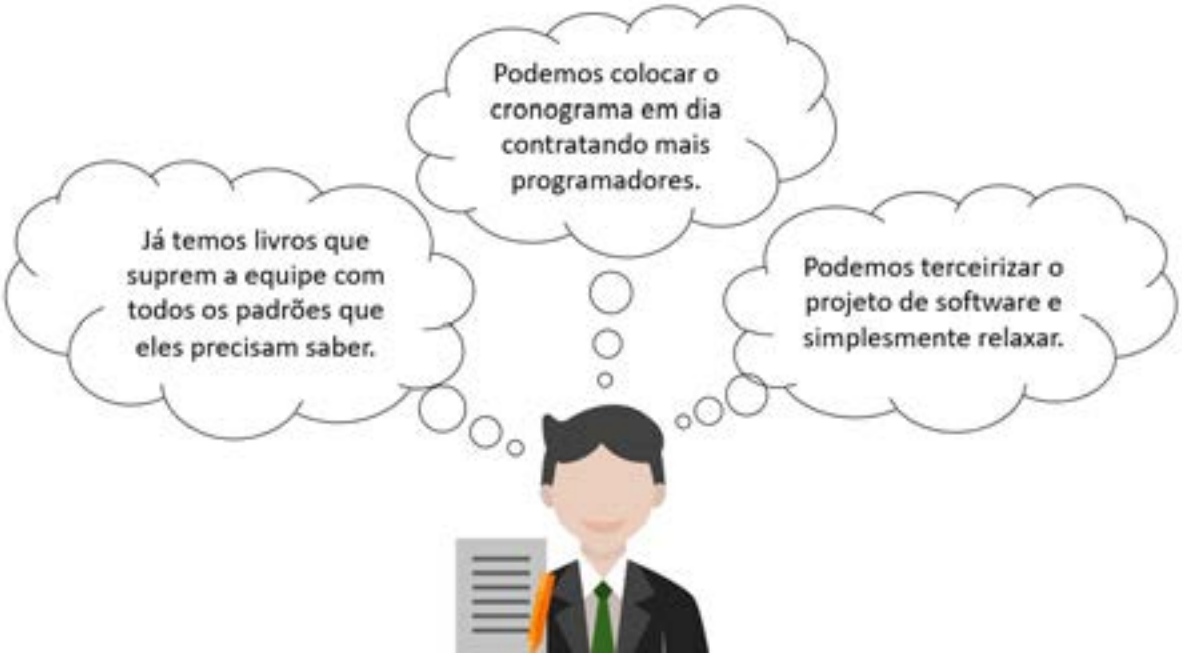


Figura 8: Mitos de gerenciamento.
Fonte: Adaptado pelo autor a partir de Pressman (2016) e imagens de https://elements.envato.com/50-professionals-flat-multicolor-icons-SXSHQQ. Acesso em 13 setembro de 2019.

Uma equipe de desenvolvimento de software pode envolver vários profissionais, como programadores, analistas, designers, dentre outros. Esses profissionais muitas

vezes ficam ansiosos pelo término do projeto e acabam alimentando crenças (Figura 9) de que determinadas atividades são desnecessárias.

Conforme discutimos na Seção 1.4, a evolução é uma atividade fundamental da engenharia de software, e muitos dos esforços de engenharia são consumidos após entrega da primeira versão do software. Assim, um erro comum é pensar que o trabalho de desenvolvimento está terminado quando o software é colocado em uso.

Na Seção 1.4, também vimos que o processo de desenvolvimento de software é formado por um conjunto de atividades que se inter-relacionam com o objetivo de produzir softwares de qualidade. Assim, é necessário realizar o controle de qualidade desde o início do processo de desenvolvimento, pois a falta de controle de qualidade de uma atividade certamente compromete a qualidade de todas as atividades subsequentes em um efeito cascata.

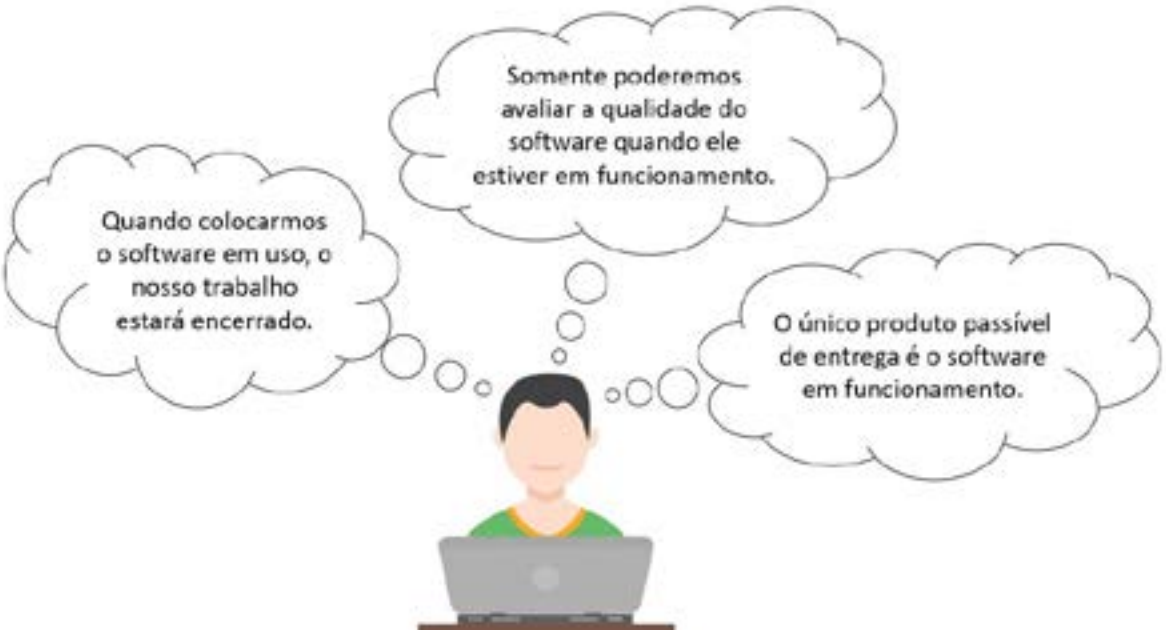


Figura 9: Mitos dos profissionais da área.
Fonte: Adaptado pelo autor a partir de Pressman (2016) e imagens de https://elements.envato.com/50-professionals-flat-multicolor-icons-SXSHQQ. Acesso em 13 setembro de 2019.

Outro mito dos profissionais da área diz que o único produto passível de entrega é o software em funcionamento. Segundo Pressman (2016), um software inclui muitos produtos derivados, tais como modelos, documentos e planos. Esses produtos também constituem uma importante base para a engenharia e o suporte de software.

AUTO ATIVIDADE

Proposta 1

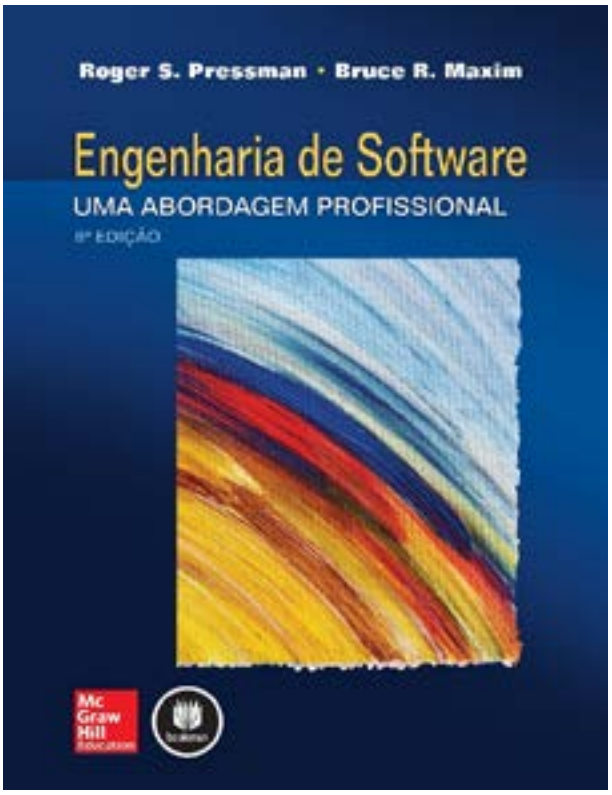
Utilize a plataforma on-line "chat" ou "fórum" e discuta com seus colegas de disciplina como os modelos de processos, métodos e ferramentas da engenharia de software podem auxiliar os profissionais da área a desenvolverem softwares de qualidade.

Proposta 2

Utilize a plataforma on-line "chat" ou "fórum" e socialize com os seus colegas de disciplina algum desenvolvimento de software que você já tenha vivido ou testemunhado. Foram utilizados modelos de processos, métodos e ferramentas da engenharia de software? Quais foram as dificuldades encontradas a longo do desenvolvimento?

SUGESTÃO DE LIVRO

PRESSMAN, Roger S; MAXIM, Bruce R. **Engenharia de Software**: uma abordagem profissional. 8. ed. Porto Alegre: AMGH, 2016. ISBN 978-0-07-802212-8.



O livro "Engenharia de Software: uma abordagem profissional", dos autores Roger S. Pressman e Bruce R. Maxim, é uma obra clássica para os profissionais de engenharia de software e que fornece uma leitura enriquecedora para quem deseja se aprofundar nos temas explorados neste caderno de estudos. O primeiro capítulo apresenta uma ampla contextualização sobre software e engenharia de software.

SUGESTÃO DE FILME

The Computers: The Remarkable Story of the ENIAC Programmers (2016)

O documentário de Kathy Kleiman conta a história de seis mulheres selecionadas para criar um algoritmo para o ENIAC, o primeiro computador totalmente eletrônico de grande escala, utilizado durante a 2ª Guerra Mundial para calcular trajetórias de projéteis. Na época, não existiam linguagens de programação ou ferramentas para ajudá-las, apenas os diagramas lógicos dos painéis que compunham o computador. As programadoras conseguiram se familiarizar com o ENIAC e descobriram que a melhor forma de programar era quebrar o cálculo em partes menores, que podiam ser executadas separadamente. Elas também desenvolveram métodos para corrigir rapidamente os bugs do ENIAC.

CONSIDERAÇÕES FINAIS

Nesta unidade, você viu que a engenharia de software é um importante arcabouço de métodos e ferramentas para a análise e o desenvolvimento de softwares de maneira produtiva e organizada, visando à qualidade e à conformidade do produto final. Em uma atualidade em que o desenvolvimento de softwares está cada vez mais complexo por conta da diversidade de tecnologias e funcionalidades exigidas pelo mercado, a engenharia de software se tornou uma disciplina vital para os profissionais e as empresas do setor.

Na vida profissional, já vi muitos programadores desprezarem a engenharia de software, considerando-a um conjunto de tarefas burocráticas que consome um tempo que poderia ser aproveitado em outras atividades. Essa imagem da engenharia de software é normalmente desconstruída no momento em que um programador tenta desenvolver seu primeiro software complexo de maneira totalmente empírica; e antes mesmo de concluir o desenvolvimento, dá-se conta que está diante de um artefato de difícil manutenção e evolução, bem como o tempo poupado com a atividade de engenharia começa a ser rapidamente consumido com refatorações (modificações estruturais) de código.

Espero que você não tenha a mesma experiência. Por isso, continue levando a sério o estudo deste caderno e tenha em mente que negar o conhecimento científico significa repetir erros do passado. Na próxima unidade, vamos conhecer alguns modelos genéricos de processo de desenvolvimento de software, mas, antes, vamos realizar alguns exercícios para verificar o seu aprendizado.

EXERCÍCIO FINAL

1. Na década de 1960, quando a engenharia de software era praticamente inexistente, a popularização dos computadores veio acompanhada de um rápido crescimento da demanda por softwares. No entanto, a inexistência de conhecimento científico para o desenvolvimento de softwares tornou os projetos difíceis de serem gerenciados. Muitos softwares eram de má qualidade e não satisfaziam aos requisitos dos clientes e usuários.

O fenômeno ao qual o texto se refere é chamado de:

A. Crise da engenharia de software.
B. Crise do software.
C. Crise da computação.
D. Crise dos computadores.
E. Crise da informática.

2. De acordo com Pressman (2016), a engenharia de software é uma disciplina que abrange um processo, cujas atividades podem variar conforme as características do modelo, um conjunto de métodos (práticas) para executar essas atividades, e um leque de ferramentas que fornece suporte automatizado ou semiautomatizado. Todos esses elementos fazem parte de uma tecnologia em camadas, que tem como pedra fundamental o comprometimento com a qualidade do produto de software.

Em relação ao processo, métodos e ferramentas da engenharia de software, avalie as afirmações a seguir:

I. O processo representa o conjunto de atividades relacionadas que levam à produção de um produto de software;
II. Os métodos determinam de que maneira cada atividade do processo será executada;
III. As ferramentas fornecem suporte automatizado ou semiautomatizado tanto para o processo quanto para os métodos.

É correto o que se afirma em:

A. II, apenas.
B. III, apenas.
C. I e II, apenas.
D. I e III, apenas.
E. I, II e III.

3. Um engenheiro de software deve estar presente ao longo de todo o processo de desenvolvimento de software, podendo desenvolver atividades de estudo, pesquisa, treinamento, especificação, coleta de dados, projeto, padronização, produção técnica, análise, planejamento, execução, gestão, coordenação, dentre outras. Em relação ao exercício da profissão de engenheiro de software, avalie as afirmações a seguir:

I. A atividade profissional do engenheiro de software está contemplada na Classificação Brasileira de Ocupações (CBO);
II. No Brasil, a profissão do engenheiro de software está regulamentada pela Resolução nº 1.100, de 24 de maio de 2018, do Conselho Federal de Engenharia e Agronomia;
III. Não existe um código de ética a ser respeitado pelo profissional de engenharia de software.

É correto o que se afirma em:

A. II, apenas.
B. III, apenas.
C. I e II, apenas.
D. I e III, apenas.
E. I, II e III.

REFERÊNCIAS

AGOSTINHO, Marcia; AMORELLI, Dirceu; BARBOSA, Simone. **Introdução à engenharia**. 1. ed. Rio de Janeiro: Lexicon, 2015.

CONSELHO FEDERAL DE ENGENHARIA E AGRONOMIA. Resolução nº 1100, de 24 de maio de 2018. **Diário Oficial da União**: edição 109, seção 1, Brasília, DF, p. 239, 8 junho 2018. Disponível em: <http://www.in.gov.br/materia/-/asset_publisher/KujrwOTZC2Mb/content/id/21012726/do1-2018-06-08-resolucao-n-1-100-de-24-de-maio-de-2018-21012669>. Acesso em: 10 set 2019.

DIJKSTRA, Edsger W. The humble programmer. **Communications of the ACM**, v. 15, n. 10, 1972. p. 859-866.

FERREIRA, Aurélio Buarque de Holanda. **Miniaurélio Século XXI**: O minidicionário da língua portuguesa. 5. ed. Rio de Janeiro: Nova Fronteira, 2001.

GOTTERBARN, D.; MILLER, K.; ROGERSON, S. Software engineering code of ethics. **Communications of the ACM**, v. 40, n. 11, p. 110-118, 1997.

MINISTÉRIO DO TRABALHO. **Listagem da CBO**. 2019. Disponível em: <http://www.mtecbo.gov.br/cbosite/pages/downloads.jsf>. Acesso em: 10 set 2019.

PORTAL EMPREGA BRASIL. **Classificação Brasileira de Ocupações (CBO)**. 2019. Disponível em: <https://empregabrasil.mte.gov.br/76/cbo/>. Acesso em: 11 set 2019.

PRESSMAN, Roger S.; MAXIM, Bruce R. **Engenharia de Software**: uma abordagem profissional. 8. ed. Porto Alegre: AMGH, 2016.

SOMMERVILLE, Ian. **Engenharia de software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.

TECMUNDO. **Das toneladas aos microchips**: a evolução dos computadores. 2016. Disponível em: <https://www.tecmundo.com.br/infografico/9421-a-evolucao-dos-computadores.htm>. Acesso em: 11 set 2019.

THE STANDISH GROUP. **About The Standish Group**. 2019. Disponível em: <https://www.standishgroup.com/about>. Acesso em: 10 set 2019.

_____. **CHAOS Report 2015**. 2015. Disponível em: <https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf>. Acesso em: 10 set 2019.

unidade 2

PROCESSOS E PARADIGMAS DE DESENVOLVIMENTO DE SOFTWARE

2 INTRODUÇÃO À UNIDADE

Na primeira unidade deste caderno de estudos, você viu que a engenharia de software é uma disciplina que está presente em todos os aspectos do processo de desenvolvimento de software, desde a especificação até a manutenção. Você viu também que o desenvolvimento de software se baseia na utilização de um processo e um arcabouço de métodos e ferramentas para atingir o objetivo de produzir softwares de qualidade, sendo que o modelo de processo adotado para o desenvolvimento de um software é que determina quais serão as atividades do processo e como será o inter-relacionamento entre elas.

Nesta segunda unidade, você irá conhecer alguns modelos de processos de desenvolvimento de software e compreender como eles funcionam. É importante que você tenha em mente que cada modelo de processo fornece um conjunto de diretrizes gerais para o desenvolvimento de softwares. Os modelos de processo muitas vezes precisam ser adaptados a cada novo projeto, em função das características da equipe e do software que está sendo desenvolvido. Ao final da leitura desta unidade, você deve ser capaz de:

- Compreender a dinâmica dos principais modelos genéricos de processo de desenvolvimento de software;
- Identificar modelos de processo adequados para o desenvolvimento de software em diferentes cenários;
- Aplicar modelos de processo no desenvolvimento de software.

2.1 PROCESSOS E PARADIGMAS DE DESENVOLVIMENTO DE SOFTWARE

De acordo com Ferreira (2001, p. 584 e 595), o termo "paradigma" pode ser definido como "*Modelo, padrão*", e o termo "processo" como o "Modo por que se realiza ou executa uma coisa". Na engenharia de software, o conjunto estruturado de atividades utilizado no desenvolvimento de software é referido tanto como "modelo de processo de desenvolvimento de software", como "paradigma de desenvolvimento de software"; ou seja, os termos "modelo de processo" e "paradigma" são utilizados no mesmo contexto (SOMMERVILLE, 2011).

Em nosso dia a dia, encontramos diferentes modelos de processos para a realização de tarefas. Por exemplo, para preparar um bolo, existem diversas receitas, assim como para

a elaboração de artigos científicos, existem diversos manuais. Esses modelos surgem para auxiliar as pessoas a realizar tarefas, e evoluem à medida que novas necessidades, métodos e ferramentas surgem.

Larman (2003) relata a aplicação de modelos de processo de desenvolvimento de software iterativos e incrementais a partir de meados da década 1950. No entanto, não há como negar que o fenômeno da crise do software, iniciado no final da década de 1960, foi um importante impulso para os profissionais e pesquisadores da área da computação trabalharem na elaboração de novos paradigmas de desenvolvimento de software. Foi nessa época que os problemas relativos a prazo, custo e qualidade de projetos de software ficaram mais evidentes por conta do rápido crescimento da demanda por softwares.

Quando pensamos sobre o que é um "processo", a primeira ideia que nos vem à mente é uma sequência de etapas encadeadas, na qual cada uma das etapas depende da total conclusão da etapa imediatamente anterior. E isso é natural, pois grande parte dos processos que realizamos ou testemunhamos no dia a dia seguem essa dinâmica. Para assar um bolo, primeiro devemos misturar todos os ingredientes. Para erguer as paredes de uma casa, primeiro devemos terminar a fundação, e assim por diante.

No entanto, essa ideia de linearidade não iterativa, em que cada etapa é executada uma única vez, tem sido pouco aplicada nos cenários de desenvolvimento de software atuais. Ao longo da história da engenharia de software, a crescente pressão sobre os prazos de execução dos projetos de software motivou o surgimento de modelos de processo iterativos, que preveem a entrega de diferentes versões de um mesmo software, com a adição ou aperfeiçoamento de "ingredientes" do software a cada nova versão. Nas próximas seções, estudaremos alguns modelos de processo de desenvolvimento de software comumente encontrados na literatura.

2.2 O MODELO EM CASCATA

"Gerenciando o desenvolvimento de grandes sistemas de software" (ROYCE, 1970, tradução nossa) foi a primeira publicação a explorar um modelo de processo de desenvolvimento de software, conhecido como "modelo em cascata" ou "ciclo de vida clássico" (Figura 10). Trata-se de um modelo derivado de processos mais gerais da engenharia de sistemas e que sugere uma abordagem sequencial e sistemática, que inicia com os requisitos do cliente e avança pelas atividades de análise, projeto, implementação, teste e operação, culminando com a manutenção contínua do software concluído (SBROCCO, 2012; PRESSMAN, 2016).

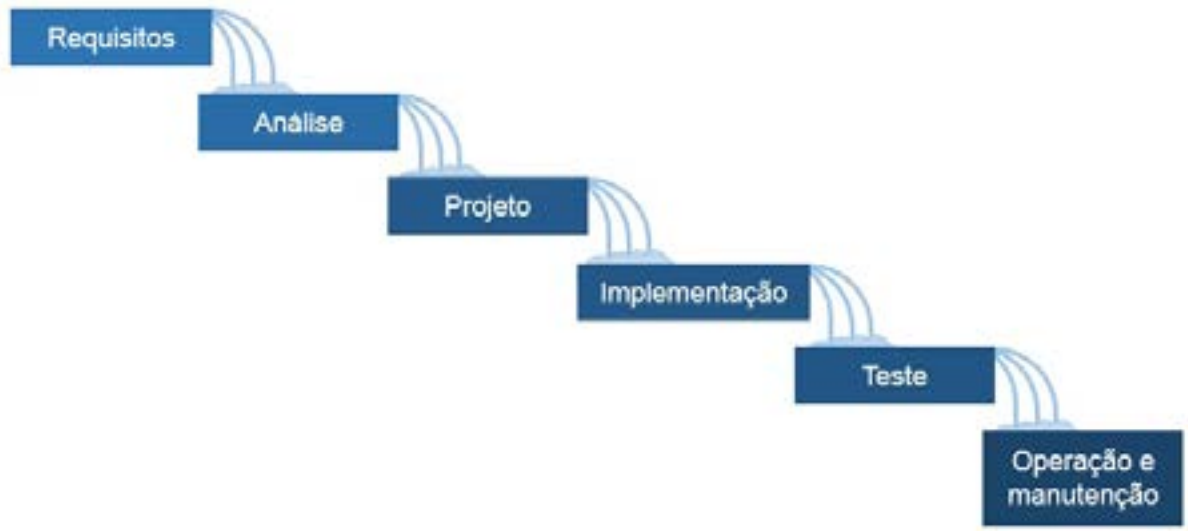


Figura 10: O modelo em cascata.
Fonte: Adaptado pelo autor a partir de Royce (1970).

No modelo em cascata, o trabalho começa pelo estabelecimento de requisitos para todos os elementos do sistema, desde as necessidades de interação com hardwares, pessoas e bases de dados, até as necessidades estratégicas no âmbito da área de negócios (SBROCCO, 2012). Na atividade de análise, o trabalho passa a estar focado na definição e documentação do subconjunto de requisitos específicos do software, tais como o domínio (âmbito) da informação, a funcionamento, as restrições, as metas, o desempenho e a interface (SOMMERVILLE, 2011; SBROCCO, 2012).

O projeto é a atividade que traduz os requisitos para uma representação das abstrações fundamentais do sistema de software e seus relacionamentos, por meio da produção de documentos de arquitetura, estruturas de dados, interfaces, detalhes procedimentais e outros tipos de documentos que permitem avaliar a qualidade do projeto antes que a implementação de código tenha início (SOMMERVILLE, 2011; SBROCCO, 2012).

Na implementação, o projeto é traduzido para uma linguagem de programação, ou seja, é gerado o código fonte do sistema. Essa atividade também é responsável pelos testes unitários, que verificam o funcionamento de cada procedimento ou função do sistema. Na atividade de teste, as unidades individuais são integradas e testadas como um todo para descobrir erros e garantir que os requisitos do software tenham sido atendidos.

Após o teste, o sistema de software é entregue ao cliente e colocado em fase de operação e manutenção. Alguns motivos que demandam a necessidade de realizar manutenção em um sistema de software são: (i) correção de erros que não foram descobertos em estágios iniciais do desenvolvimento; (ii) adaptações para acomodar mudanças no ambiente externo; (iii) melhoramentos funcionais ou de desempenho; e (iv) adição de funcionalidades em resposta a novos requisitos (SBROCCO, 2012).

No modelo em cascata puro, o resultado de cada atividade é a aprovação de um ou mais documentos que alimentam de informações as próximas atividades. As atividades são executadas em estrita sequência, ou seja, uma atividade não deve ser iniciada até que a atividade anterior seja concluída, o que facilita muito a gestão dos projetos. Por outro lado, é um modelo de baixa visibilidade para o cliente, que só vê o resultado no final e que dificulta o atendimento de demandas de alterações de requisitos (SOMMERVILLE, 2011; PRESSMAN, 2016).

Na prática, é sempre necessário considerar que, em qualquer atividade do processo, as atividades anteriores possam ser revisadas e seus resultados alterados, por exemplo, os documentos de projeto serem alterados durante a implementação, à medida que problemas vão sendo identificados (PAULA FILHO, 2009). E de fato, apesar de muitos profissionais da área tratarem o modelo em cascata como uma cachoeira de passagem única (Figura 10), Royce (1970) considerou a possibilidade de realimentação entre as atividades, permitindo que, em atividades posteriores, haja revisão e alteração dos resultados das atividades anteriores, conforme ilustra a Figura 11.

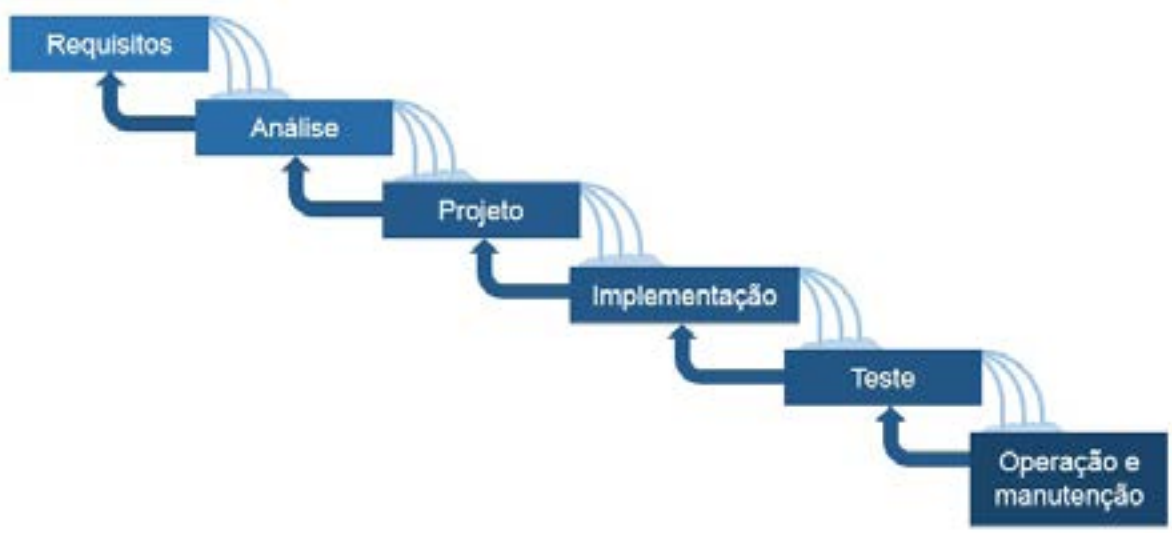


Figura 11: Modelo em cascata com realimentação.
Fonte: Adaptado pelo autor a partir de Royce (1970).

As iterações do modelo em cascata são realizadas na forma de uma cascata invertida, o que pode levar ao congelamento de partes do desenvolvimento e ao uso de artifícios de implementação para contornar problemas de projeto, resultando em sistemas mal estruturados e que não atendem aos requisitos (SOMMERVILLE, 2011; PRESSMAN, 2016). Além disso, as iterações podem ser dispendiosas e envolver significativo retrabalho.

Assim, o modelo em cascata é inapropriado para projetos de software que têm ritmos acelerados e estão constantemente sujeitos a mudanças, porém, pode servir para projetos de curta duração, tais como pequenas manutenções e evoluções de software, e projetos

cujos requisitos sejam inicialmente bem compreendidos e pouco suscetíveis a mudanças (SOMMERVILLE, 2011; PRESSMAN, 2016).

Existe ainda uma variação do modelo em cascata denominada modelo V (Figura 12), que descreve a relação entre as atividades de verificação (requisitos, projeto, implementação) e a atividade de teste, cujas ações estão associadas à garantia da qualidade de cada uma das atividades de verificação.

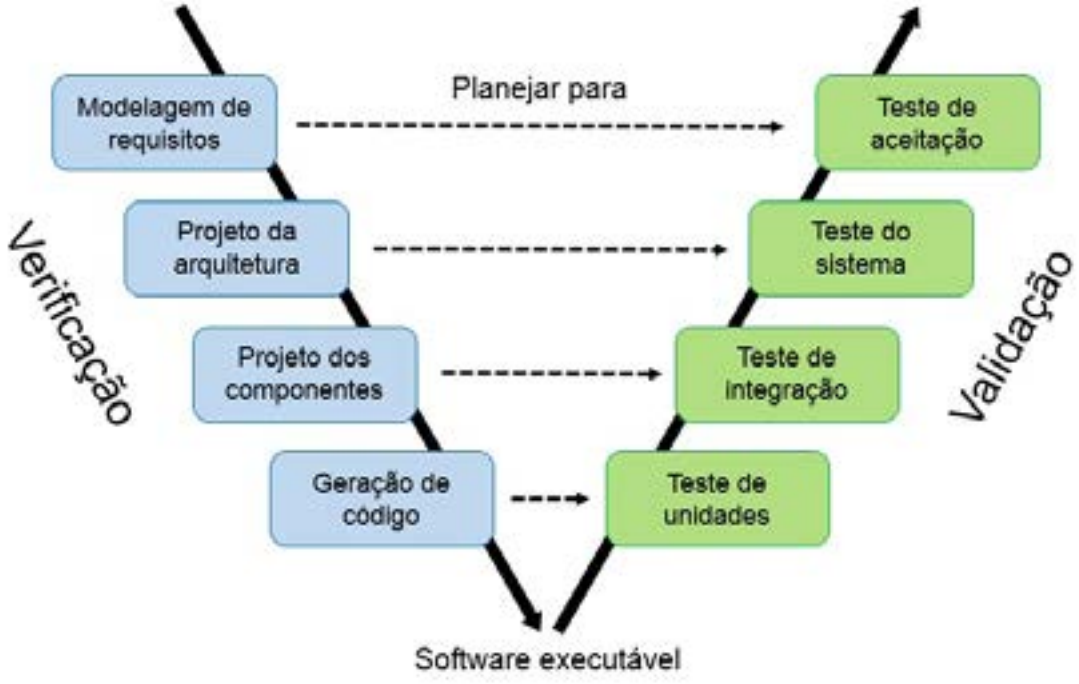


Figura 12: O modelo V.
Fonte: Adaptado pelo autor a partir de Pressman (2016).

Segundo Pressman (2016), o modelo V fornece apenas uma visão de como as atividades de verificação e validação são aplicadas, não existindo uma diferença fundamental entre o ciclo de vida clássico (cascata) e o modelo V.

PARA REFLETIR

Nesta seção, vimos que o modelo em cascata é um paradigma de desenvolvimento de software frequentemente citado como um modelo rígido e que pode levar a sistemas mal estruturados. No entanto, o modelo em cascata pode ser vantajoso em cenários de licitações e serviços específicos para órgãos públicos. Reflita sobre o porquê de o modelo em cascata apresentar possíveis vantagens nesses cenários. Podem ser realizadas pesquisas em sites na Internet.

2.3 MODELOS DE PROCESSO INCREMENTAL

Existem situações em que, apesar de os requisitos iniciais do software serem bem definidos, é necessário o rápido fornecimento de um determinado conjunto de funcionalidades ao cliente. Nessas situações, pode-se optar por um modelo de processo projetado para desenvolver o software de forma **iterativa** (na qual atividades se repetem) e **incremental** (na qual cada repetição realiza entregas) (PRESSMAN, 2016).

O modelo incremental aplica sequências lineares e cumulativas ao longo do tempo. Cada sequência linear gera entregáveis que incrementam o software. Por exemplo, ao desenvolver um sistema de e-commerce de maneira incremental, você poderia liberar as funcionalidades de cadastro de produtos no primeiro incremento; o carrinho de compras no segundo; a geração de relatórios no terceiro; e, finalmente, recursos avançados de inteligência artificial no quarto.

Em princípio, os incrementos são partes do produto de software que possuem a capacidade de operar e fornecer alguma funcionalidade para o cliente. No exemplo do sistema de e-commerce, enquanto você trabalha no desenvolvimento do carrinho de compras (segundo incremento), o cliente já poderia utilizar o primeiro incremento para cadastrar o estoque. No entanto, Sommerville (2011) destaca que entrega e implantação de incrementos em processos operacionais reais nem sempre é possível, e pode ser necessário desenvolver softwares de forma incremental apenas para expô-los à avaliação do cliente.

A cada iteração do modelo incremental, a equipe de desenvolvimento passa pelas clássicas atividades de requisitos, análise, projeto, implementação e teste em uma parte específica do produto de software (Figura 13). As atividades são repetidas até que nenhuma outra iteração seja necessária. Sob esse ponto de vista, o modelo iterativo e incremental pode ser visto como uma série consecutiva de miniprojetos que usam o modelo em cascata (SCHACH, 2010).

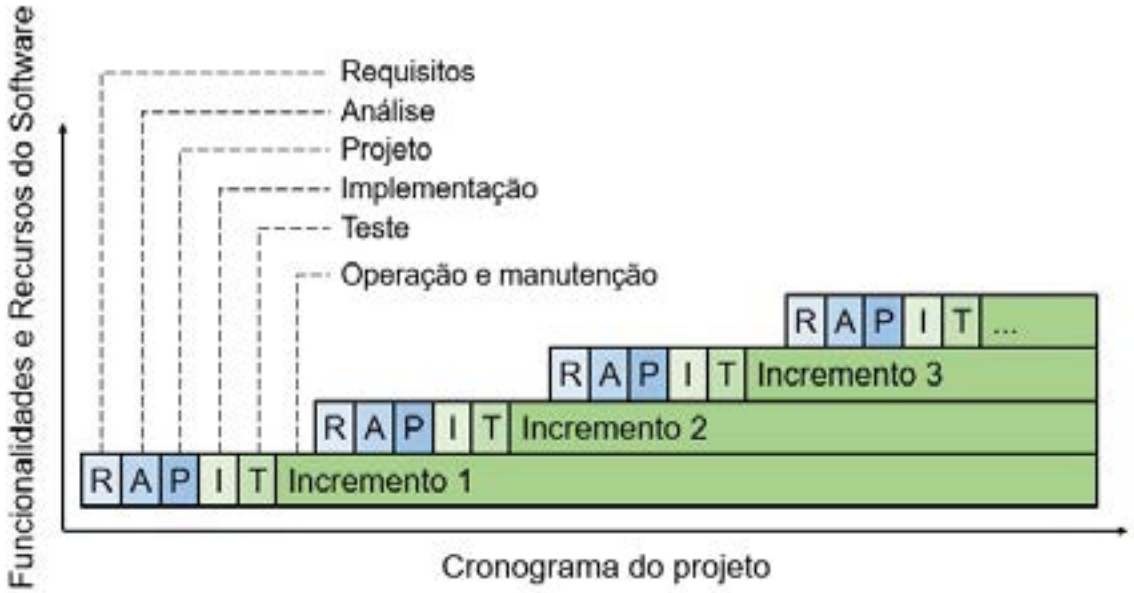


Figura 13: Modelo de processo incremental.
Fonte: Adaptado pelo autor a partir de Pressman (2016).

Antes da liberação de cada incremento, é essencial testar cada artefato e fazer as alterações necessárias, até que a equipe de desenvolvimento esteja satisfeita com os artefatos do incremento (SCHACH, 2010). Quando isto acontece, é desenvolvido um planejamento para o incremento seguinte, com base no resultado do uso e/ou avaliação do cliente sobre o incremento atual (PRESSMAN, 2016).

Frequentemente, os incrementos iniciais entregam o produto essencial (mais urgente ou importante) para que o cliente possa avaliar os requisitos básicos já nos estágios iniciais do desenvolvimento (SOMMERVILLE, 2011). Esses incrementos podem ser implementados com número mais reduzido de pessoal, que poderá ser aumentado se o produto essencial for bem acolhido (PRESSMAN, 2016).

Segundo Sommerville (2011), o desenvolvimento incremental de software é uma abordagem comum para o desenvolvimento de sistemas aplicativos e uma parte fundamental das metodologias ágeis de desenvolvimento de software. No entanto, é particularmente crítico para projetos complexos e de longa duração, que necessitam de modelos que permitam que várias equipes comecem a planejar e desenvolver diferentes partes do software desde o início do projeto (SOMMERVILLE, 2011).

EXERCÍCIO

Uma instituição de ensino superior deseja adquirir um software para gerenciamento da sua biblioteca. O sistema deverá permitir aos funcionários da biblioteca cadastrar usuários (alunos e funcionários), cadastrar livros e periódicos, consultar o acervo, efetuar empréstimos, efetuar devoluções e gerar relatórios de alunos, empréstimos, livros e periódicos. Supondo que você seja contratado para desenvolver esse software, proponha uma estratégia de "quebra" do sistema em partes menores para a aplicação do modelo de processo incremental, de forma que cada incremento seja capaz de operar e fornecer alguma funcionalidade para o cliente.

2.4 PROTOTIPAÇÃO

Existem situações em que o cliente não expõe detalhadamente os requisitos do sistema. Também existem situações em que o desenvolvedor está inseguro quanto ao modo de projetar o sistema. Nessas situações, pode-se optar por desenvolver uma versão simplificada da solução para ajudar a compreender melhor o que está para ser desenvolvido. Essa versão simplificada é chamada de protótipo.

Segundo Sommerville (2011) e Pressman (2016), embora a prototipação possa ser utilizada como um modelo de processo isolado, que evolui de uma solução simplificada até alcançar um produto acabado, é mais utilizada no contexto de qualquer um dos modelos de processo, como uma técnica de auxílio para as seguintes atividades:

- **Requisitos**. A prototipação pode auxiliar na identificação e omissões e erros nos requisitos ou, quando estes ainda são obscuros, na melhor compreensão do que deve ser construído. Os usuários finais podem se envolver na avaliação do protótipo para identificar pontos fortes e fracos do software e obter novas ideias para requisitos.
- **Projeto**. Um protótipo pode ser usado para apoiar o projeto de interface de usuário, principalmente no que se refere ao modelo de interação homem-máquina. Um protótipo também pode ser usado para avaliar a aderência e o desempenho de soluções de infraestrutura, tais como bancos de dados e sistemas operacionais.

A Figura 14 apresenta um modelo de processo de prototipação que pode ser usado para a identificação de requisitos de software. O processo inicia com a definição dos objetivos e limitações de escopo, para que os interessados possam entender a função do protótipo e, consequentemente, obter os benefícios que esperam do seu desenvolvimento (SOMMERVILLE, 2011; PRESSMAN, 2016).

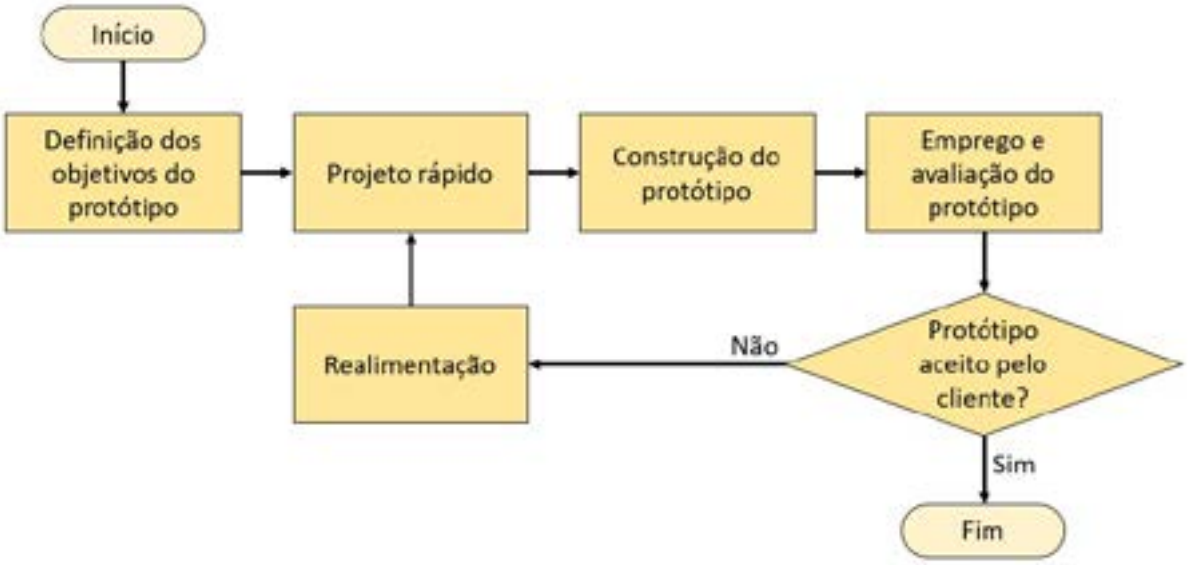


Figura 14: Modelo de processo de prototipação.
Fonte: Adaptado pelo autor a partir de Pressman (2016), Sommerville (2011) e Schach (2010).

Em seguida, a prototipação é planejada rapidamente e ocorre a modelagem de um "projeto rápido", que se concentra na representação dos aspectos visíveis aos usuários finais (PRESSMAN, 2016). Como exemplo, Schach (2010) utiliza o protótipo de um software de contas a pagar, contas a receber e estoques, que executa a captura de dados na tela, imprime relatórios, mas não realiza nenhuma atualização de arquivo ou tratamento de erro.

Schach (2010) entende que a estrutura interna de um protótipo rápido não é relevante porque o seu único objetivo é determinar quais são as necessidades do cliente. Além disso, a prototipagem rápida é essencial para que os custos sejam controlados e os interessados possam experimentar o protótipo no início do processo de software, conforme explica Sommerville (2011).

O projeto rápido leva à construção de um protótipo, que é empregado e avaliado. Nesta etapa, os usuários devem receber treinamento e ter tempo para se sentirem confortáveis e se situarem em um padrão natural de uso (SOMMERVILLE, 2011). O resultado da avaliação serve de realimentação para a próxima iteração. As iterações ocorrem conforme o protótipo se ajusta às necessidades do cliente, e uma vez que o cliente esteja satisfeito, o processo é encerrado e os desenvolvedores podem elaborar um documento

de especificações que atenda às verdadeiras necessidades do cliente (SCHACH, 2010).

Segundo Pressman (2016), embora um protótipo possa evoluir lentamente até se transformar no sistema real, na maioria dos projetos não há preocupação com a sua qualidade e a manutenção a longo prazo; a prototipação se preocupa apenas em desenvolver um produto "descartável" que sirva de base para o desenvolvimento de uma versão redesenhada, na qual os problemas identificados com o auxílio da prototipação são resolvidos.

Um problema geral com a prototipação pode surgir na etapa de emprego e avaliação do protótipo. Sommerville (2011) explica que os usuários envolvidos podem não receber treinamento suficiente, não terem tempo para se sentirem confortáveis e até mesmo não serem usuários típicos do sistema. Com isso, a participação dos usuários na avaliação pode não atingir os seus objetivos, que são a identificação de pontos fortes e fracos do software e obtenção de novas ideias para requisitos.

2.5 MODELO ESPIRAL

Nas seções anteriores você viu que (i) o modelo em cascata sugere uma abordagem sequencial e sistemática; (ii) o modelo de processo incremental foi projetado para desenvolver softwares de forma iterativa; e (iii) a prototipação evolui de uma solução simplificada até alcançar um produto acabado.

O modelo espiral proposto por Boehm (1988) reúne essas três características dos modelos estudados anteriormente em um só processo. Ou seja, o software é desenvolvido com a aplicação de uma sequência de atividades metodológicas, que são executadas de forma iterativa para produzir versões cada vez mais refinadas do software.

A Figura 15 mostra a representação gráfica do modelo proposto por Boehm (1988). Os quadrantes coloridos representam as quatro atividades metodológicas do modelo, e o espiral representa tanto a evolução do processo (dimensão angular) como a evolução do produto de software (dimensão radial). Desta forma, o modelo reflete o conceito subjacente de que cada iteração envolve uma evolução que aborda a mesma sequência de atividades para determinado nível de elaboração do produto (BOEHM, 1988). A dimensão radial também representa o custo acumulado na realização das etapas. Assim que esse processo evolucionário começa, a equipe de desenvolvimento realiza as atividades indicadas no circuito espiral, começando pelo centro no sentido horário (PRESSMAN, 2016).

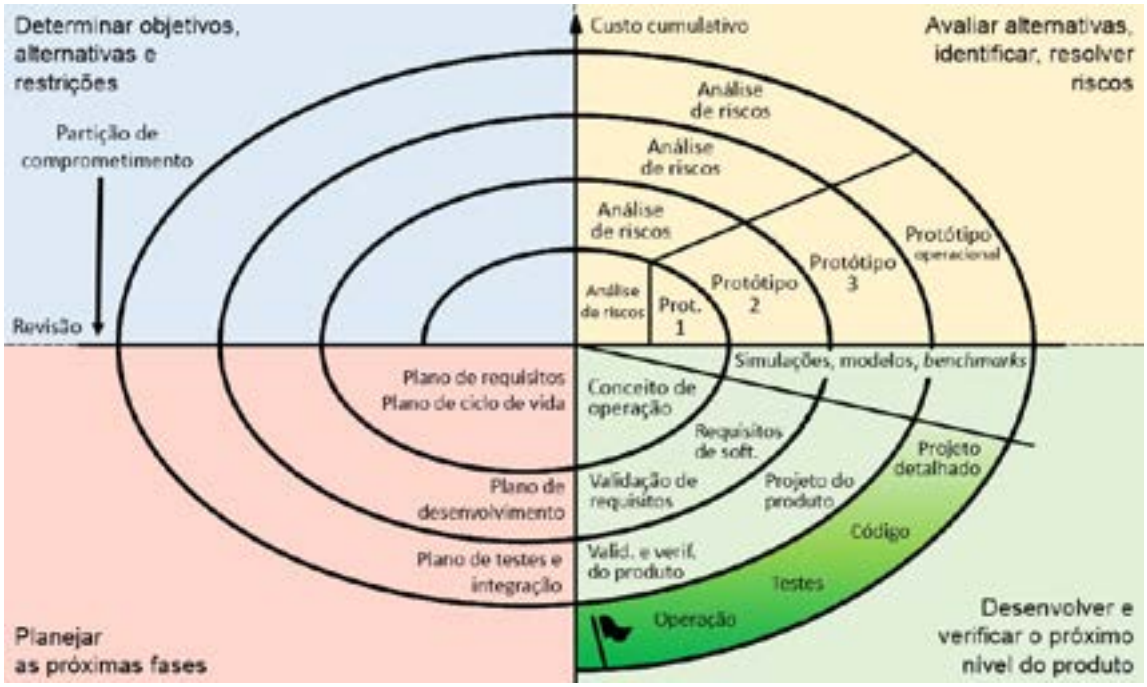


Figura 15: Modelo espiral.
Fonte: Adaptado pelo autor a partir de Boehm (1988).

Cada iteração começa com a identificação dos objetivos da parte do produto que está sendo elaborada, das alternativas para implementar essa parte do produto (design, reutilização, compra) e das restrições impostas à aplicação das alternativas (custo, cronograma, interface) (BOEHM, 1988).

O próximo passo é analisar os riscos das alternativas relativas aos objetivos e restrições. Se forem identificadas fontes significativas de risco, devem ser tomadas medidas para resolvê-las, tais como prototipação, simulação, modelos, *benchmarks*, dentre outras técnicas de resolução de riscos (BOEHM, 1988). A principal diferença entre o modelo espiral em relação a outros modelos de processo de software é o seu reconhecimento explícito dos riscos, que devem ser diretamente considerados em todos os estágios do projeto (SOMMERVILLE, 2011; PRESSMAN, 2016).

Se o protótipo inicial for suficiente para servir como uma base de baixo risco para a evolução do produto, as iterações subsequentes desenvolvem uma série de protótipos evolutivos (Protótipo 2, Protótipo 3, ...) (BOEHM, 1988). Por outro lado, se a prototipação já resolveu todos os riscos de desempenho ou de interface do usuário, e os riscos de desenvolvimento dominam, o processo segue o modelo de ciclo de vida clássico (Conceito, Requisitos, Projeto, ...) (BOEHM, 1988). Desta forma, o modelo espiral mantém a abordagem em etapas sistemáticas do ciclo de vida clássico ao mesmo tempo que usa a prototipação como mecanismo de redução de riscos em qualquer estágio do processo (PRESSMAN, 2016).

De acordo om Boehm (1988), cada iteração é concluída com uma revisão que envolve as principais pessoas ou organizações envolvidas no produto e abrange todos os produtos desenvolvidos durante a iteração, incluindo os planos para a próxima iteração e os recursos necessários para sua execução. O principal objetivo da revisão é garantir que todos os envolvidos se comprometam com a abordagem para a próxima iteração. Pressman (2016) explica que a cada passagem pela região de planejamento, o gerente de projeto pode fazer um ajuste no número de iterações planejadas, bem como no custo e cronograma do projeto, de acordo com o feedback (realimentação) do cliente.

Pressman (2016) considera que o modelo espiral é uma abordagem realista para o desenvolvimento em larga escala e que pode ser adaptada para ser aplicada ao longo da vida do software. Por exemplo, a primeira iteração pode ser usada para o desenvolvimento de um projeto de conceitos, e as iterações subsequentes para o desenvolvimento de versões cada vez mais evoluídas do software, permitindo aos desenvolvedores e clientes compreenderem melhor os riscos em cada nível evolucionário.

2.6 DESENVOLVIMENTO BASEADO EM COMPONENTES

Quando as pessoas envolvidas no desenvolvimento de um software descobrem que podem aproveitar implementações de outros projetos, fazem as modificações necessárias nessas implementações para incorporá-las a seus softwares. Assim, em diversos projetos, há algum reuso informal, que ocorre independentemente do modelo de processo de desenvolvimento usado (SOMMERVILLE, 2011). A ideia de que o processo de desenvolvimento de software deve se empenhar em produzir componentes que possam ser reusados foi fomentada pela primeira vez por McIlory (1968), durante a Conferência de Engenharia de Software da OTAN.

O desenvolvimento baseado em componentes depende de uma ampla base de componentes reusáveis e de um framework de integração desses componentes (SOMMERVILLE, 2011). Em alguns casos, os componentes são softwares comerciais de prateleira ou COTS (Commercial Off-The-Sele), desenvolvidos por vendedores que os oferecem como produtos com interfaces bem definidas, que permitem que os componentes sejam integrados em diversos projetos de software (PRESSMAN, 2016).

De acordo com Sommerville (2011), existem três tipos de componentes que podem ser usados em um desenvolvimento orientado a reuso: (i) web services disponíveis para invocação remota; (ii) coleções de objetos que são desenvolvidas como pacotes a serem integrados com frameworks de componentes; e (iii) sistemas de software stand-alone

(que são autossuficientes, não necessitam de um software auxiliar).

A Figura 16 apresenta o modelo de processo geral de desenvolvimento baseado no reuso, que inicia com a especificação de requisitos e segue com a busca e análise de componentes, modificação dos requisitos (em função dos componentes encontrados), projeto do sistema com reuso, desenvolvimento e integração dos componentes e validação de sistema (SOMMERVILLE, 2011). O que diferencia o desenvolvimento baseado em componentes dos modelos estudados nas seções anteriores são as etapas internas do processo, já que a especificação de requisitos e a validação de sistema são atividades comuns aos outros modelos.

Figura 16: Desenvolvimento baseado em componentes.
Fonte: Adaptado pelo autor a partir de Sommerville (2011).

Brown (1996) fornece uma visão de como ocorre o processo de integração dos componentes dentro da engenharia de software baseada em componentes (Figura 17). A primeira atividade (seleção) cuida de buscar componentes autônomos com potencial para serem reusados na construção do software. Podem ser selecionados tanto componentes COTS como componentes de sistemas já desenvolvidos.

A próxima atividade cuida de qualificar os componentes aptos para uso do desenvolvimento do sistema, considerando os requisitos do sistema e a usabilidade, performance e confiabilidade dos componentes. Em seguida, os componentes são adaptados para remover incompatibilidades. A Figura 17 ilustra a aplicação de uma técnica de adaptação denominada wrapping (encapsulamento). A atividade de composição cuida de agregar os componentes por meio de middlewares ou outros mecanismos de integração. Por fim, a atividade de atualização é responsável por gerenciar a configuração dos componentes, que podem ser substituídos à medida que surgem novas versões.

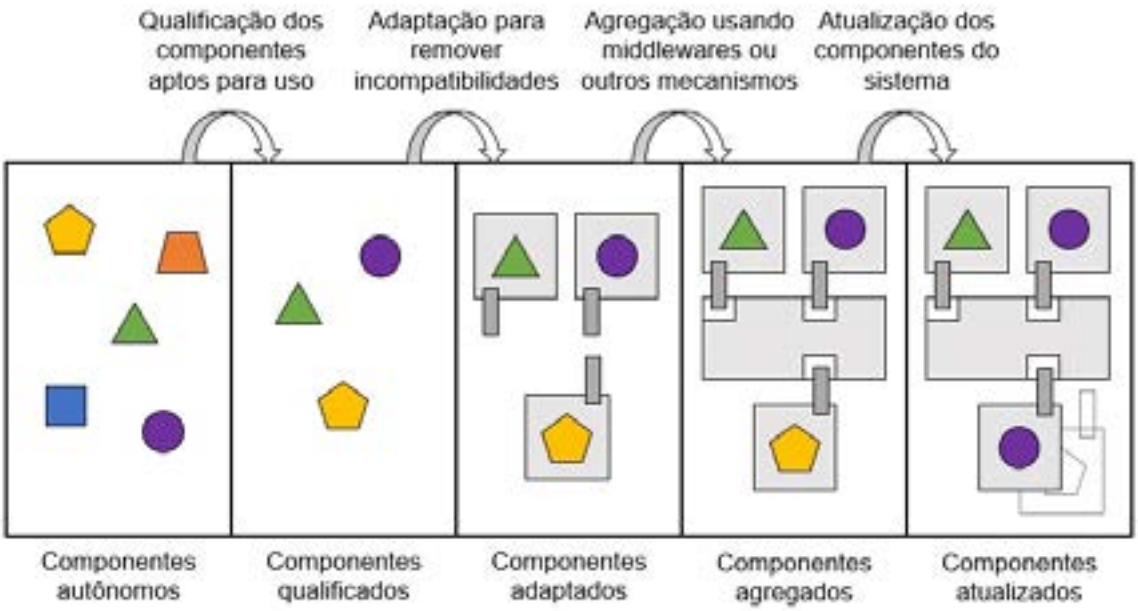


Figura 17: Processo de integração de componentes.
Fonte: Adaptado pelo autor a partir de Brown (1996).

Pressman (2016) considera que o modelo de desenvolvimento baseado em componentes incorpora as características evolucionária e iterativa do modelo espiral e proporciona uma série de benefícios mensuráveis aos engenheiros de software, tais como a redução dos custos e do tempo de execução do projeto. Por sua vez, Sommerville (2011) faz a ressalva de que o modelo pode levar a um software que não atende às reais necessidades dos usuários e a perda do controle sobre a evolução do sistema, já que as novas versões dos componentes não estão sob o domínio da organização que está desenvolvendo o software.

2.7 PROCESSO UNIFICADO (RUP)

O Processo Unificado é um modelo que propõe a implementação dos princípios do desenvolvimento ágil aproveitando as melhores características dos modelos de processo genéricos. Ele também é chamado de Processo Unificado Racional (RUP - Rational Unified Process), em homenagem a Rational Corporation da IBM, que contribuiu para o desenvolvimento do modelo e de ferramentas que dão suporte ao processo. O RUP reconhece a importância da comunicação com o cliente e apoia a prototipação e a entrega iterativa e incremental (PRESSMAN, 2016).

A Figura A mostra o modelo RUP em duas perspectivas: uma perspectiva que

representa o tempo e mostra o aspecto dinâmico (ciclos, fases, iterações) do processo à medida que ele é executado, e uma perspectiva estática, que mostra como o processo é descrito em termos de atividades, artefatos, pessoas e fluxos de trabalho (workflows) (IBM, 1998). Além dessas duas perspectivas, o RUP também fornece um conjunto de boas práticas a serem usadas durante o processo.

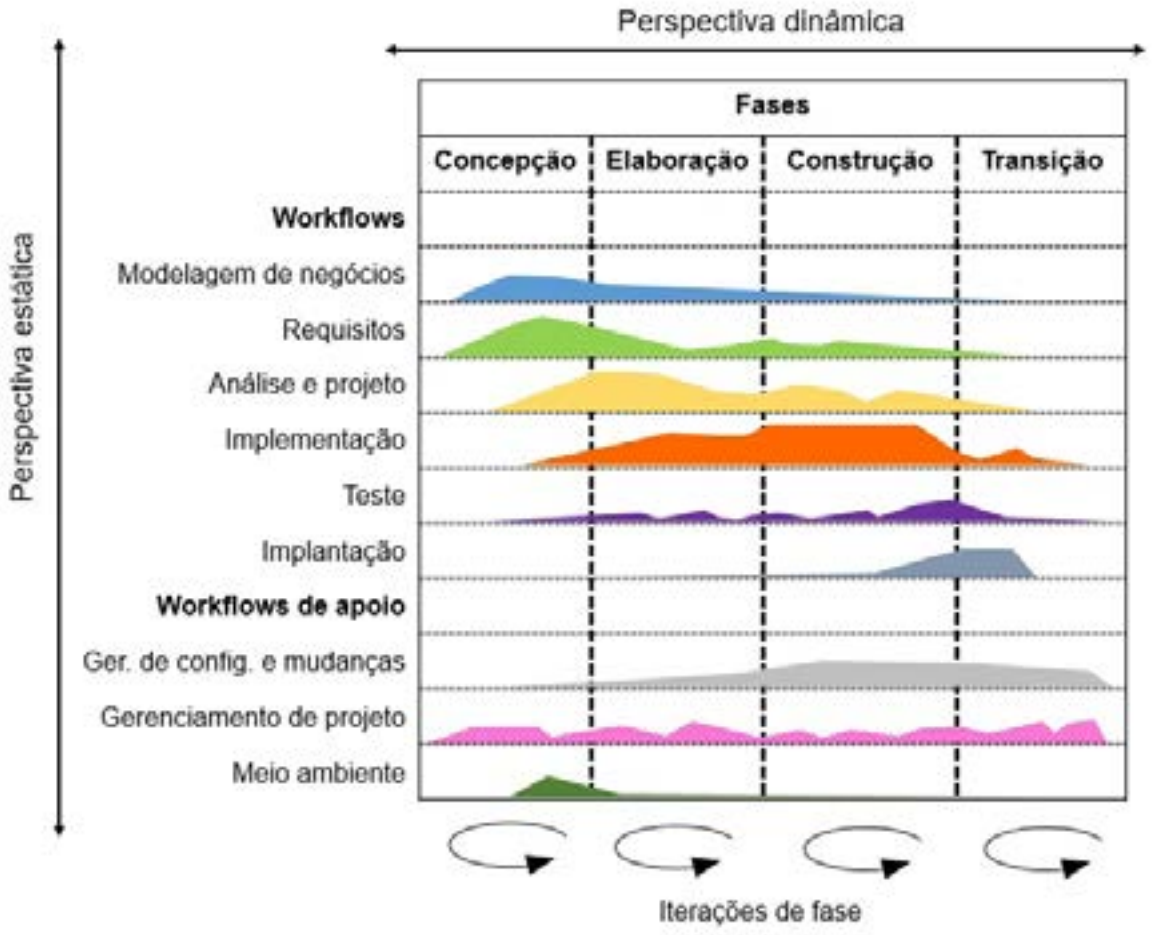


Figura 18: Processo Unificado da Rational (RUP).
Fonte: Adaptado pelo autor a partir de IBM (1998) e Sommerville (2011).

No RUP, tanto uma fase como todo o conjunto de fases pode ser executada de forma iterativa e incremental. Isso significa que as fases do RUP ocorrem de forma concorrente e escalonada, e não em sequência, como nos modelos genéricos (SOMMERVILLE, 2011; PRESSMAN, 2016).

Durante a fase de concepção, você estabelece o *business case* (caso de negócios) do sistema e delimita o escopo do projeto. Isso inclui um documento de visão geral (requisitos, recursos e restrições do projeto principal), um modelo de casos de uso de alto nível (com os casos mais significativos), uma avaliação inicial de risco e um plano

de projeto, que mostra as fases, as iterações e as datas dos principais marcos (IBM, 1998).

O propósito da fase de elaboração é: (i) analisar o domínio do problema; (ii) estabelecer uma base arquitetônica sólida, incluindo um protótipo de arquitetura executável; (iii) desenvolver plano de desenvolvimento para o projeto geral, mostrando iterações e critérios de avaliação para cada iteração; e (iv) eliminar os elementos de maior risco do projeto (IBM, 1998). Ao final da fase, é tomada a decisão pela continuação ou não do projeto.

Por sua vez, a fase de construção envolve o desenvolvimento (ou aquisição) e integração dos componentes de software e a execução de testes de unidade, integração e aceitação (PRESSMAN, 2016). O resultado da fase de construção é um produto de software versão "beta", pronto para colocar nas mãos de seus usuários finais (IBM, 1998). Por fim, a fase de transição cuida da colocação do sistema em ambiente de produção (SOMMERVILLE, 2011). Os usuários finais realizam testes beta e fornecem *feedback* dos defeitos e mudanças necessárias (PRESSMAN, 2016).

A perspectiva estática do modelo RUP mostra as atividades que ocorrem durante o processo, chamadas de *workflows*. Existem seis *workflows* centrais e três *workflows* de apoio. Todos os *workflows* podem estar ativos em todas as fases, conforme se observa nos gráficos coloridos da Figura 18. No entanto, perceba que os *workflows* de modelagem, requisitos e análise tendem a estar mais ativos nas fases iniciais, enquanto os demais *workflows* tendem a estar mais ativos nas fases finais.

Os *workflows* são orientados por modelos associados à UML (Unified Modeling Language), uma linguagem para a elaboração da estrutura de projetos de software que contribuiu para a criação do modelo RUP (SOMMERVILLE, 2011). Além das duas perspectivas apresentadas na Figura 18, o RUP também fornece um conjunto de boas práticas a serem usadas durante o processo, são elas: (i) desenvolver o software iterativamente; (ii) gerenciar requisitos; (iii) usar arquiteturas baseadas em componentes; (iv) modelar o software visualmente; (v) verificar a qualidade do software; (vii) controlar as mudanças do software (IBM, 1998). Estas são chamadas de "boas práticas" porque são comumente usadas por organizações bem-sucedidas.

Sommerville (2011) entende que as inovações mais importantes do RUP são a separação de fases e *workflows* e o reconhecimento explícito da implantação de software como uma fase do processo (transição).

SAIBA MAIS

Nesta seção, você viu que um processo de desenvolvimento de software pode envolver uma série de atividades, artefatos, pessoas e fluxos de trabalho, que por sua vez, estão relacionados com diversas áreas de conhecimento. O SWEBOK (Software Engineering Body of Knowledge) é um documento criado sob o patrocínio da IEEE (Institute of Electrical and Electronic Engineers) Computer Society, em parceria com a ACM (Association for Computing Machinery), com a finalidade de criar um consenso sobre as áreas de conhecimento dae de software e seu escopo. Pesquise e descreva quais são as 15 áreas de conhecimento da engenharia de software propostas no SWEBOK V3, publicado em 2014. Podem ser utilizados sites na Internet.

FÓRUM

Nesta unidade, você viu que diferentes modelos de processo ou paradigmas de desenvolvimento de software podem apresentar características em comum. Por exemplo, alguns modelos que estudamos nas seções anteriores apresentam a característica de serem iterativos, ou seja, suas atividades se repetem ao longo do processo de desenvolvimento. Utilize o fórum on-line da disciplina para discutir com seus colegas sobre outras características que dois ou mais modelos estudados nessta unidade apresentam em comum.

SUGESTÃO DE LIVRO

SBROCCO, José Henrique Teixeira de Carvalho; MACEDO, Paulo Cesar de. **Metodologias ágeis:** engenharia de software sob medida. 1. ed. São Paulo: Erica, 2012. ISBN 978-85-365-1941-8.

Tendo em vista o crescente interesse pelo uso das metodologias ágeis, esta obra apresenta, de maneira comparativa e completa, as características, as aplicações e os exemplos de seus principais paradigmas, tais como Iconix, SCRUM, XP, FDD, DSDM, ASD e família de metodologias Crystal. Contempla também um capítulo sobre uma nova proposta de

metodologia ágil, desenvolvida para ser utilizada em projetos acadêmicos. Destinada a estudantes e pesquisadores da área de computação e de gestão de projetos, mostra-se alinhada com as novas tendências mundiais, necessárias para atender aos requisitos da atual realidade de mercado de projetos de software.

SUGESTÃO DE FILME

Aardvark'd: 12 Weeks with Geeks (2005)

Quatro estagiários são trazidos para o escritório de Manhattan da Fog Creek Software e recebem 12 semanas para projetar, desenvolver, depurar e enviar um programa de computador que, entre outras coisas, ajudará milhões de usuários frustrados a consertar os computadores de seus parentes pela Internet. A Boondoggle Films apresenta uma jornada pelo mundo do desenvolvimento de software da perspectiva de um startup de software única, quatro estagiários peculiares e o mundo dos geeks. O documentário está disponível no YouTube com legendas em inglês e legendas com tradução automática para o português.

CONSIDERAÇÕES FINAIS

Nesta unidade, você viu que diversos modelos de processo (ou paradigmas) de desenvolvimento de software surgiram ao longo da história da engenharia de software. Esses modelos fornecem um conjunto de diretrizes gerais para o desenvolvimento de softwares, e muitas vezes precisam ser adaptados a cada novo projeto, inclusive com a combinação de características de diferentes modelos.

Essa unidade não teve o objetivo de realizar uma comparação entre os modelos de processo apresentados, no sentido de avaliar qual é o melhor e o pior modelo. É apropriado que esse tipo de avaliação seja realizado, considerando-se cada cenário de desenvolvimento.

Por exemplo, se você está desenvolvendo um software extremamente simples e sem uma equipe, o modelo em cascata pode ser o mais adequado. Por outro exemplo, se você está desenvolvendo um software cujos módulos tenham baixo acoplamento (dependência), os modelos iterativos e incrementais podem ser os mais adequados. Por último exemplo, se você está diante do desenvolvimento de um software cujos requisitos são obscuros, a prototipação pode ser uma boa estratégia para identificar e refinar os requisitos.

Continue levando a sério o estudo do caderno, pois na próxima unidade vamos falar um pouco sobre documentação, uma importante atividade para o sucesso de projetos de software.

EXERCÍCIO FINAL

1. O modelo em cascata é um paradigma de desenvolvimento de software que sugere uma abordagem sequencial e sistemática. Trata-se de um modelo inapropriado para projetos de software que têm ritmos acelerados e estão constantemente sujeitos a mudanças, porém, pode servir para projetos de curta duração, tais como pequenas manutenções e evoluções de software, e projetos cujos requisitos sejam inicialmente bem compreendidos e pouco suscetíveis a mudanças.

Em relação ao modelo em cascata, avalie as afirmações a seguir:

I. O processo inicia com a atividade de requisitos e avança pelas atividades de análise, projeto, implementação, teste e operação, culminando com a atividade de manutenção contínua do software concluído;
II. O resultado de cada atividade é a aprovação de um ou mais documentos que alimentam de informações as próximas atividades;
III. Não existe a possibilidade de realimentação entre as atividades.

É correto o que se afirma em:
A. II, apenas.
B. III, apenas.
C. I e II, apenas.
D. I e III, apenas.
E. I, II e III.

2. Na engenharia de software, o conjunto estruturado de atividades utilizado no desenvolvimento de software é referido tanto como “modelo de processo de desenvolvimento de software” como “paradigma de desenvolvimento de software”. O modelo de processo adotado para o desenvolvimento de um software é que determina quais serão as atividades do processo e como será o inter-relacionamento entre elas. Além disso, cada modelo apresenta um conjunto de características que o torna mais adequado para determinados cenários de desenvolvimento.

Em relação às características dos diferentes modelos de processo ou paradigmas de desenvolvimento de software, analise as seguintes sentenças:

I. Pode ser utilizado como um modelo de processo isolado, que evolui de uma solução simplificada até alcançar um produto acabado, porém, ele é mais utilizado no contexto de qualquer um dos modelos de processo, como uma técnica de auxílio para as atividades de requisitos e projeto;
II. Propõe o desenvolvimento de software com a aplicação de uma sequência de atividades metodológicas, que são executadas de forma iterativa para produzir versões cada vez mais refinadas do software;
III. Proporciona a redução dos custos e do tempo de execução do projeto, por meio do reuso de software existente.

As sentenças I, II e III se referem, respectivamente, a quais modelos de processo de desenvolvimento de software?

A. Prototipação, modelo espiral e desenvolvimento baseado em componentes.
B. Prototipação, modelo em cascata e desenvolvimento baseado em componentes.
C. Modelo em cascata, modelo espiral e prototipação.
D. Modelo em cascata, prototipação e desenvolvimento baseado em componentes.
E. Modelo espiral, modelo em cascata e prototipação.

3. O Processo Unificado, também chamado de Processo Unificado Racional (RUP - Rational Unified Process), é um modelo que reconhece a importância da comunicação com o cliente e propõe a implementação dos princípios do desenvolvimento ágil, aproveitando as melhores características e recursos dos modelos de processo genéricos, tais como a prototipação e a entrega iterativa e incremental (PRESSMAN, 2016).

Em relação às características do modelo RUP, avalie as afirmações a seguir:

I. Fornece uma perspectiva dinâmica, que mostra ciclos, fases e iterações do processo, e uma perspectiva estática, que mostra como o processo é descrito em termos de atividades (*workflows*);
II. Tanto uma fase como todo o conjunto de fases (ciclo) pode ser executada de forma iterativa e incremental;
III. As atividades (*workflows*) são orientadas por modelos associados à UML (Unified Modeling Language).

É correto o que se afirma em:

A. II, apenas.
B. III, apenas.
C. I e II, apenas.
D. I e III, apenas.
E. I, II e III.

REFERÊNCIAS

BOEHM, B. W. A Spiral Model of Software Development and Enhancement, **IEEE Computer**, 21(5), pp. 61-72, 1988.

BROWN, Alan; WALLNAU, Kurt**. Engineering of Component-Based Systems**. In: IEEE 2nd ICECCS'96, Montreal, Canada (1996), 414-422.

FERREIRA, Aurélio Buarque de Holanda. **Miniaurélio Século XXI**: O minidicionário da língua portuguesa. 5. ed. Rio de Janeiro: Nova Fronteira, 2001.

IBM. **Rational Unified Process**: best practices for software development teams. Rational Software White Paper, 1998. Disponível em: <https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf>. Acesso em: 19 set 2019.

LARMAN, Craig; BASILI, Victor R. **Iterative and Incremental Development**: A Brief History. IEEE Computer Society, 2003.

MCILROY, M. D. **Mass produced software components**. In Naur, P. and B. Randell, editors, Report on a Conference of the NATO Science Committee, pp 138–150, 1968.

PAULA FILHO, Wilson de Pádua. **Engenharia de software**: fundamentos, métodos e padrões. 3.ed. Rio de Janeiro: LTC, 2009. ISBN 978-85-216-1650-4.

PRESSMAN, Roger S.; MAXIM, Bruce R. **Engenharia de Software**: uma abordagem profissional. 8. ed. Porto Alegre: AMGH, 2016.

ROYCE W. W. **Managing the Development of Large Software Systems**. Proceedings of the IEEE WESCON. Los Angeles: IEEE; 1970:1–9.

SBROCCO, José Henrique Teixeira de Carvalho; MACEDO, Paulo Cesar de. **Metodologias ágeis**: engenharia de software sob medida. 1. ed. São Paulo: Erica, 2012. ISBN 978-85-365-1941-8.

SCHACH, Stephen R. **Engenharia de software**: os paradigmas clássico e orientado a objetos. 7. ed. Porto Alegre: AMGH, 2010. ISBN 978-85-63308-44-3.

SOMMERVILLE, Ian. **Engenharia de software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.

unidade 3

DOCUMENTAÇÃO DE SOFTWARE

3 INTRODUÇÃO À UNIDADE

O termo "documentação de software" é muitas vezes entendido como sendo um conjunto de documentos que são produzidos apenas nas etapas finais do processo de desenvolvimento de software. E é natural que iniciantes em engenharia de software pensem assim, pois a palavra "documento" normalmente nos remete a produções escritas que acompanham os produtos que adquirimos no dia a dia, tais como manuais de instrução, manuais de montagem e instalação; dentre outros tipos de documentos que descrevem as características e o funcionamento de algo para os seus usuários.

Na verdade, a documentação é uma atividade que está presente em todas as fases do processo de desenvolvimento de software. Cuida de documentar tanto o **processo** de desenvolvimento em si, como o **produto** (**software**) que está sendo desenvolvido. A **documentação do processo** é produzida para que o processo de desenvolvimento seja administrável, e pode incluir planos, estimativas, cronogramas, relatórios e até mesmo as mensagens eletrônicas que registram as comunicações entre as pessoas envolvidas no projeto. Esse tipo de documentação é um dos focos de estudo da disciplina de Gerência de Projetos.

Por sua vez, a **documentação do software** inclui tanto a documentação do usuário (manuais, guias de instalação, arquivos de ajuda) como a documentação do sistema, que descreve como o sistema foi implementado. Fazendo uma analogia simples com a construção civil, os documentos do sistema são como as plantas e projetos de um empreendimento, utilizados tanto na execução da obra como na sua manutenção; e os documentos dos usuários são como os manuais dos proprietários, que explicam como fazer o bom uso do imóvel.

Nesta unidade, nossos estudos estarão voltados para a **documentação do sistema**, já que a documentação do usuário varia muito em função das características de cada produto, e pode até ser desnecessária quando o produto é autoexplicativo. Ao final da leitura, você deve ser capaz de:

- Compreender a importância da documentação de software;
- Identificar documentações úteis e necessárias em diferentes cenários de desenvolvimento de software;
- Produzir documentos de software.

3.1 DOCUMENTAÇÃO DE SOFTWARE

Por que softwares devem ser documentados? A resposta para essa pergunta está na história que vou contar a partir de agora. Certo dia, uma equipe de desenvolvimento de software foi designada para executar um projeto para uma empresa de logística. Como primeira ação do projeto, o engenheiro de software se reuniu com os dirigentes da empresa para definir o que o software deveria fazer, ou seja, quais seriam as funcionalidades que deveriam estar disponíveis para os usuários. Durante a reunião, que se estendeu por várias horas, o engenheiro registrou alguns lembretes que julgou importantes no aplicativo de texto do seu laptop.

Ao chegar em seu ambiente de trabalho, o engenheiro de software se reuniu com a equipe de desenvolvimento e realizou uma breve explicação sobre o sistema que seria desenvolvido. Pressionada pelo prazo de entrega, a equipe decidiu iniciar a implementação do sistema imediatamente, os programadores começaram a "botar a mão na massa", produzindo o código fonte do sistema.

Passadas algumas semanas, dois programadores da equipe receberam uma atrativa proposta de trabalho de outra empresa, e se desligaram do projeto. Foram necessárias mais algumas semanas para que dois novos programadores pudessem ser introduzidos na equipe. Não bastasse isso, ao chegar no ambiente de trabalho, os novos programadores descobriram que milhares de linhas de código haviam sido escritas, e que nenhum documento sobre os padrões implementados havia sido produzido. Neste momento, os novos programadores se viram diante de um grande desafio: compreender o que havia sido implementado.

Apesar de todos esses fatos que atrasaram o cronograma do projeto, a equipe conseguiu com muito suor concluir a primeira versão do sistema. No entanto, durante a implantação do sistema, os dirigentes da empresa de logística alegaram que o sistema não fornecia todas as funcionalidades necessárias, e apesar de o engenheiro de software tentar convencê-los que desenvolveu o que foi solicitado, não houve um consenso sobre quais foram exatamente os requisitos definidos na reunião inicial.

O engenheiro de software estava diante do pior cenário que pode ser encontrado em um projeto de software: cronograma atrasado, custos extrapolados e um produto que não atende às demandas do cliente. E grande parte desses problemas poderia ter sido evitada se os requisitos e a implementação do software tivessem sido devidamente documentados e validados.

A esta altura, você deve estar imaginando que essa é uma história hipotética. E realmente é. No entanto, os fatos que rodeiam essa história não são raros de acontecer

em projetos de desenvolvimento de software. Eu mesmo já presenciei diversas situações semelhantes.

Agora que você já compreendeu a importância da documentação de software, vamos iniciar o estudo de alguns tipos de documentos comumente usados em projetos de desenvolvimento de software.

3.2 DOCUMENTO DE VISÃO

Na Seção 2.7, você viu que o documento de visão é um dos produtos de trabalho da fase de concepção do modelo RUP (*Rational Unified Process*). Mas o que é um documento de visão? Segundo a IBM (2019, on-line), o documento de visão "*define o escopo de alto nível e o propósito de um programa, produto ou projeto*". Ou seja, o objetivo do documento de visão é fornecer uma visão de alto nível (sem se aprofundar em detalhes) do produto, para que os interessados possam compreender o que será desenvolvido. A Figura 19 apresenta a estrutura do documento de visão da IBM. Vamos conhecê-la melhor.

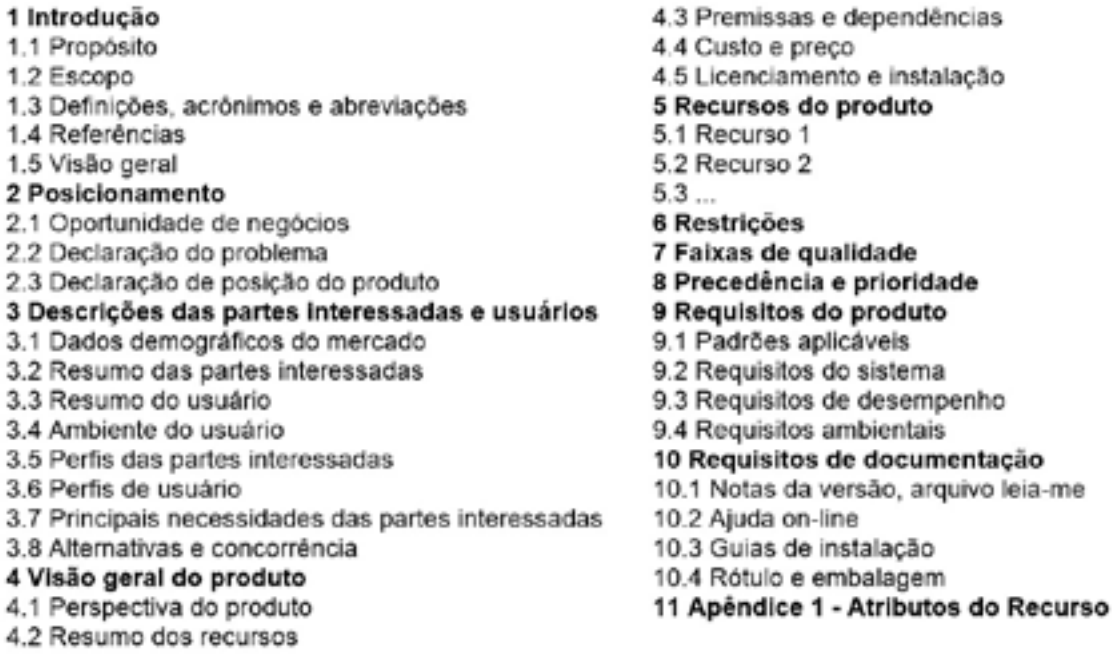


Figura 19: Estrutura do documento de visão da IBM.
Fonte: Adaptado pelo autor a partir de IBM (2019).

A **introdução** determina o propósito do documento (e não do produto em si), descreve brevemente o seu escopo (alvo), apresenta todas as definições de termos, acrônimos e

abreviações necessárias para que o documento seja interpretado corretamente e lista todos os documentos referenciados (por exemplo, um relatório). Por fim, descreve o conteúdo do documento e explica como ele está organizado (IMB, 2019).

A seção de **posicionamento** descreve brevemente a oportunidade de negócios e declara o problema que o projeto resolve, incluindo a descrição das partes afetadas pelo problema, o impacto do problema nos negócios e os principais benefícios que serão proporcionados pela solução (IBM, 2019). O posicionamento é encerrado com a declaração de posição do produto, que comunica o propósito do produto e a sua importância para todas as partes interessadas.

O documento de visão também possui uma seção dedicada à descrição das **partes interessadas e usuários**. Ela deve identificar o nome e a função de todas as partes interessadas na definição dos requisitos, bem como o nome dos usuários e sua relação com o sistema. Além disso, essa seção apresenta os dados demográficos de mercado que motivam as decisões sobre o produto e as alternativas disponíveis (por exemplo, a compra do produto de um concorrente) (IBM, 2019).

Por sua vez, a **visão geral do produto** começa fornecendo uma perspectiva do produto em relação a outros produtos e ao ambiente do usuário. Em seguida, ela resume os principais benefícios e recursos do produto e aponta as suas premissas e dependências. Pense em uma premissa como algo supostamente verdadeiro, mas ainda não confirmado. Por exemplo, uma premissa pode indicar que determinada infraestrutura de hardware e software deverá estar disponível quando o sistema for implantado. Por fim, a seção de visão geral do produto registra quais são os aspectos importantes para o custo e a precificação do produto e os requisitos de licenciamento e instalação que podem impactar diretamente no esforço de desenvolvimento (IBM, 2019).

Os **recursos do produto** são as suas capacidades. No documento de visão, as funcionalidades do sistema de software devem ser descritas em alto nível, pois existem documentos específicos para a especificação e o detalhamento de requisitos de software, conforme será visto nas Seções 3.3 e 3.4. Por exemplo, “disponibilizar ao médico as informações de cada paciente” e “permitir que o paciente envie resultados de exames ao médico” podem ser consideradas descrições de alto nível das capacidades de um sistema para a área da saúde.

A estrutura do documento de visão da IBM também fornece seções específicas para a comunicação das **restrições** (por exemplo, “serão utilizados apenas softwares livres para o desenvolvimento do sistema”) e **faixas de qualidade** do produto, que definem o desempenho, a tolerância a falhas, a usabilidade e outras características não descritas pela seção de recursos do produto (IBM, 2019).

Na segunda unidade, você viu que os projetos de software que utilizam modelos de

processo incrementais frequentemente entregam o produto essencial (mais urgente ou importante) já nos primeiros incrementos. Assim, é importante que o documento de visão defina também a **precedência e prioridade** dos recursos do software. Quais funcionalidades são mais importantes para o cliente? Quais funcionalidades devem ser implementadas primeiro?

A seção que trata dos **outros requisitos do produto** começa fornecendo uma lista com todos os padrões de conformidade do produto. A lista pode incluir padrões jurídicos e regulamentares, de comunicação (por exemplo, TCP/IP), de plataforma (por exemplo, UNIX) e de segurança e qualidade (por exemplo, normas ISO) (IBM, 2019). Em seguida, são definidos os requisitos de sistema (por exemplo, plataformas de rede e memória) e os requisitos de desempenho, que podem estar relacionados com fatores como largura de banda, exatidão e tempo de resposta. O documento de visão inclui também uma seção de requisitos relacionados a problemas ambientais como umidade, radiação, temperatura, condições de uso e problemas de manutenção (IBM, 2019).

Por fim, o documento de visão descreve a documentação necessária para a implementação bem-sucedida do sistema (**requisitos de documentação**) e um apêndice, que adiciona atributos aos recursos do produto para que eles possam ser avaliados, controlados, priorizados e gerenciados. Trata-se de em uma espécie de plano de gerenciamento de requisitos (IBM, 2019).

SAIBA MAIS

De acordo com o plano de gerenciamento de requisitos fornecido no Apêndice 1 do modelo de documento de visão da IBM (2019), um recurso pode ser considerado "Crítico", "Importante" ou "Útil", ao mesmo tempo que pode assumir os estados "Proposta", "Aprovado" ou "Incorporado". Pesquise e descreva cada um desses possíveis estados de um recurso. Uma versão em português do modelo documento de visão da IMB pode ser encontrada no próprio site da empresa por meio da *string* de busca "Documento de Visão para um Projeto de Requisitos".

Nessa seção, você conheceu um modelo estrutural para a elaboração de um documento de visão. Esse tipo de documento é usado nas fases preliminares do processo de desenvolvimento de software para proporcionar uma visão de alto nível e fornecer diversas informações importantes tanto para a tomada de decisão pelos gestores do

projeto quanto para o desenvolvimento do produto em si.

No decorrer das próximas subseções, vamos explorar documentos usados para proporcionar visões mais detalhadas de um software; ou seja, para fornecer informações mais voltadas para o processo de desenvolvimento em si, mas que, apesar disso, não deixam de ser importantes também para a gestão de projetos de software.

3.3 ESPECIFICAÇÃO DE REQUISITOS

Na primeira unidade, vimos que as atividades da engenharia de software podem estar voltadas tanto para o desenvolvimento de novos softwares como para a modificação ou evolução de softwares existentes. Por exemplo, uma instituição de ensino pode estar interessada em (i) adquirir um novo sistema de gestão acadêmica, (ii) adquirir um módulo adicional (por exemplo, processo seletivo) para um sistema existente ou (iii) ampliar os dados de cadastro dos alunos, a fim de que sejam registradas informações como contatos de emergência, tipo sanguíneo e alergias.

Seja qual for o caso, é necessário descobrir, analisar e documentar o que será desenvolvido para que se tenham critérios de qualidade e aceitação do produto pelo cliente. Em uma analogia simples, um apartamento novo com sacada integrada ao living pode ser de má qualidade para um cliente que deseja comprar um apartamento com espaço ao ar livre; ao mesmo tempo que um apartamento antigo pode ser de boa qualidade para um cliente que necessita de um apartamento com peças amplas e dependência de empregado.

O entendimento sobre a qualidade de software segue o mesmo raciocínio. Um software de qualidade é aquele que, dentre outros aspectos, está em conformidade com as necessidades do cliente. As descrições do que um software deve fazer, dos serviços que ele deve oferecer e das restrições ao seu funcionamento são chamadas de **requisitos** (PFLEEGER, 2004; MARINHO, 2016). Em outras palavras, um requisito é uma característica que o sistema deve possuir para atingir seus objetivos, ou seja, atender às necessidades dos processos de negócio específicos do cliente.

A **engenharia de requisitos** é uma subárea da engenharia de software que faz o uso sistemático e repetitivo de técnicas para elicitar (obter informações junto aos clientes), analisar, documentar, validar e manter os requisitos de um software de maneira completa, clara e precisa (PAULA FILHO, 2009; VAZQUEZ; SIMÕES, 2016).

A Figura 20 mostra que a engenharia de requisitos pode ser vista como uma disciplina que aborda diferentes “camadas” de requisitos. A partir da definição dos requisitos

(ou necessidades) de negócio, são realizados sucessivos refinamentos até se chegar à definição dos requisitos do software. Com isso, a definição dos requisitos de uma camada é usada como fundamento para a definição dos requisitos da próxima camada.

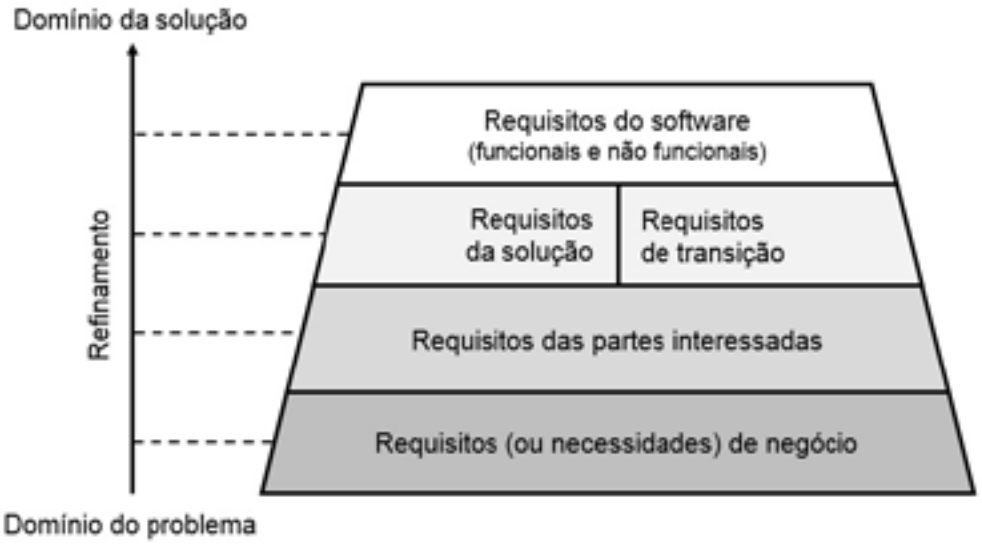


Figura 20: Tipos de requisitos abordados pela engenharia de requisitos.
Fonte: Adaptado pelo autor a partir de Vazquez e Simões (2016).

Segundo Vazquez e Simões (2016), os requisitos de negócio são declarações que descrevem a motivação do projeto, suas metas, bem como as métricas que serão utilizadas para aferir o seu sucesso. Eles refletem as necessidades ou objetivos da organização como um todo, sem considerar as necessidades particulares de grupos ou pessoas. A segunda camada de requisitos cuida de descrever as necessidades de informação para que as partes interessadas possam desempenhar suas tarefas. Considera-se parte interessada qualquer pessoa que possa afetar ou ser afetada pela solução que está sendo desenvolvida (VAZQUEZ; SIMÕES, 2016).

Os requisitos da solução são as descrições estruturadas a serem usadas no desenvolvimento do software. Essa camada de requisitos resolve os conflitos das partes interessadas, preenche as lacunas de informação e aproveita as oportunidades de racionalização do projeto (VAZQUEZ; SIMÕES, 2016). Em alguns casos, pode ser necessário declarar necessidades para a fase de transição da solução que está sendo desenvolvida. Por exemplo, pode-se declarar que o sistema legado deverá funcionar em paralelo nos primeiros meses de operação do novo sistema. Essas declarações são chamadas de requisitos de transição.

De acordo com Vazquez e Simões (2016), a soma dos requisitos da solução com os requisitos de transição (se houver) resulta nos requisitos de software, que são compostos

por requisitos funcionais e requisitos não funcionais. Enquanto os requisitos funcionais descrevem o software sob a perspectiva das interações dos seus usuários (por exemplo, cadastrar clientes e cadastrar produtos), os requisitos não funcionais cuidam de descrever as propriedades do software, tais como usabilidade, desempenho, plataformas, linguagens de programação e sistemas de gerenciamento de banco de dados.

Sommerville (2011) (Figura 21) fornece uma visão em espiral do processo de engenharia de requisitos que nos mostra as macroatividades (elicitação, especificação e validação) que se repetem a cada ciclo do processo de requisitos.

Figura 21: Visão em espiral do processo de engenharia de requisitos.
Fonte: Adaptado pelo autor a partir de Sommerville (2011).

A elicitação de requisitos é uma atividade especialmente desafiadora. Devemos trabalhar com os clientes, usuários e até mesmo especialistas da área de aplicação para determinar os limites da solução que será desenvolvida. Para tal, podemos (e quase sempre devemos) utilizar uma variedade de técnicas, tais como mapeamento de processos, questionários e entrevistas que realizam as mesmas perguntas de várias maneiras.

A especificação de requisitos é a atividade de analisar e escrever os requisitos em um documento de requisitos, que serve como um "contrato" entre o cliente e a equipe de

desenvolvimento. Pfleeger (2004) explica que o formato da documentação resultante da especificação de requisitos depende das características e necessidades do projeto e do modelo de processo utilizado. Por exemplo, as histórias do usuário (*user stories*) (Figura 22) são documentos usados para descrever requisitos quando se utiliza metodologias de desenvolvimento ágil de software, como SCRUM e Extreme Programming.

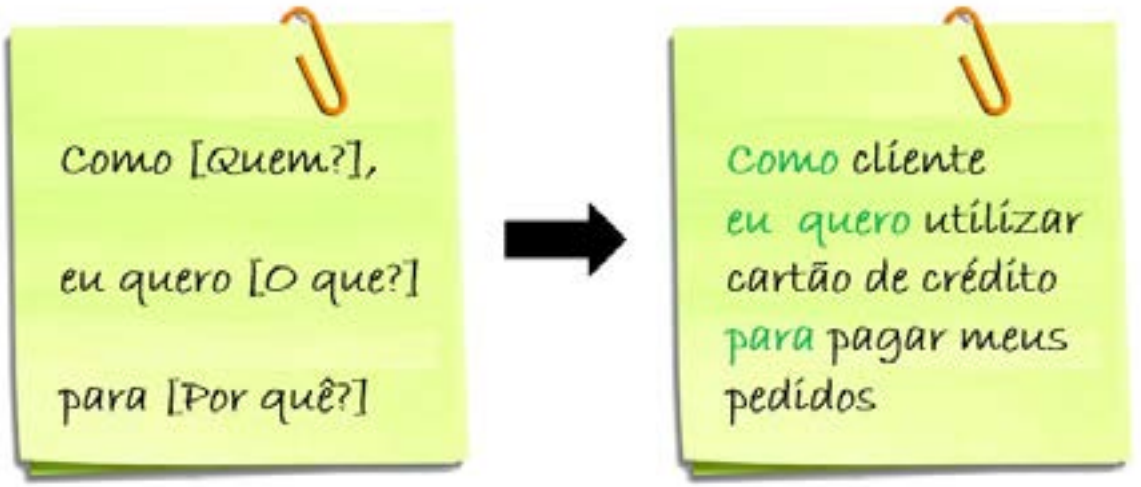


Figura 22: Histórias de usuários.
Fonte: Elaborado pelo autor (2019).

Em outros casos, listas de requisitos cumprem esse papel. O Quadro 1 mostra um exemplo de lista de requisitos funcionais, requisitos não funcionais e regras de negócio para o desenvolvimento de um sistema de gestão de biblioteca. Perceba que os requisitos funcionais descrevem o software sob a perspectiva das interações dos seus usuários (cadastrar livros, consultar livros, ...). Por sua vez, os requisitos não funcionais descrevem as propriedades do software, tais como compatibilidade, desempenho e sistema de gerenciamento de banco de dados.

Por fim, as regras de negócio completam a lista de requisitos, descrevendo as políticas, condições ou restrições que devem ser consideradas na execução dos processos existentes na biblioteca.

Requisitos funcionais
RF01. O sistema deverá permitir ao bibliotecário cadastrar livros. **RF02**. O sistema deverá permitir ao bibliotecário e aos alunos consultar livros. **RF03**. O sistema deverá permitir ao bibliotecário efetuar empréstimo de livros. **RF04**. O sistema deverá permitir ao bibliotecário efetuar devolução de livros. **RF05**. O sistema deverá notificar o aluno dos empréstimos expirados. **RF06**. O sistema deverá permitir ao bibliotecário gerar multas das devoluções em atraso e enviá-las ao sistema financeiro da instituição. RF07...
Requisitos não funcionais.
RNF01. O sistema dever ser compatível com os navegadores Google Chrome e Mozilla Firefox. **RNF02**. A consulta de livros não deve levar mais de 10 segundos para fornecer resultados. **RNF03**. O banco de dados deve ser desenvolvido no MySQL. RNF04...
Regras de negócio
RN01. Um aluno pode emprestar até três livros simultaneamente. **RN02**. Um aluno somente pode emprestar livros se não tiver multas a pagar. **RN03**. O valor da multa por atraso na devolução de livros é de R$ 2,00 por dia de atraso. RN04...

Quadro 1 –Lista de requisitos funcionais, requisitos não funcionais e regras de negócio para um sistema de gestão de biblioteca.
Fonte: Elaborado pelo autor (2019)

Segundo Paula Filho (2009), um documento de requisitos pode apresentar problemas decorrentes da própria linguagem natural que ele utiliza. Assim, há que se ter muito cuidado na elaboração de um documento de requisitos. Uma especificação de requisitos de qualidade deve ser correta, completa, clara, consistente, modificável, priorizada, verificável e rastreável (VAZQUEZ; SIMÕES, 2016).

Por exemplo, "A consulta deve ser rápida" não é um requisito verificável, pois "rápida" é um termo subjetivo que pode ser interpretado de muitas formas. Por outro exemplo, se você utilizar os termos "funcionário da biblioteca" e "bibliotecário" para se referir ao mesmo tipo de usuário em diferentes partes do documento, você estará redigindo uma especificação inconsistente, que provavelmente causará confusão nos leitores, tais como: existem outros funcionários na biblioteca além do bibliotecário? O bibliotecário possui permissões que os demais funcionários não possuem?

Além disso, é importante você ter em mente que um documento de requisitos pode servir a três propósitos: (i) proporcionar aos desenvolvedores o entendimento de como os clientes querem que o sistema funcione; (ii) informar aos projetistas que funcionalidades e características o sistema deve ter; e (iii) informar à equipe de testes o que demonstrar para convencer os clientes de que o sistema está sendo entregue de acordo com o que foi

solicitado (PFLEEGER, 2004).

A falha em não detalhar de maneira suficiente as informações na especificação de requisitos pode levar a interpretações equivocadas. Por exemplo, a sonda espacial norte-americna Mars Climate Orbiter (MCO) foi destruída ao entrar na órbita de Marte devido a um erro de cálculo. Uma investigação apurou que os desenvolvedores do software utilizaram o sistema métrico britânico para enviar coordenadas, quando deveriam ter utilizado o sistema métrico universal. Por outro lado, uma especificação muito detalhada pode ser onerosa e levar o projeto ao atraso. O desafio é encontrar o nível de detalhe que melhor consiga promover a comunicação entre as partes (VAZQUEZ; SIMÕES, 2016).

A última macroatividade do processo de requisitos (Figura 21) é a validação de requisitos. Nessa fase, o cliente deve ser capaz de compreender a especificação, fornecer *feedback* e aprovar o documento de forma consciente, para que o projeto possa seguir com poucos riscos de entregar um produto que não satisfaça o cliente (VAZQUEZ; SIMÕES, 2016). No entanto, ter um documento de requisitos aprovado pelo cliente nem sempre significa ter requisitos permanentes. De acordo com Paula Filho (2009), a instabilidade de requisitos é um problema comum no desenvolvimento de software. Clientes podem solicitar novos requisitos ou alterações de requisitos com o projeto em andamento, e isso gera novos custos e retrabalho.

Além disso, a engenharia de requisitos é um processo que pode encontrar diversas dificuldades, tais como a falta de clareza na comunicação; a falta de acesso (ou acesso limitado) às partes interessadas, indefinições e imprecisões do cliente, resistência a mudanças, conflitos e desejos pessoais, dentre outros desafios inerentes a qualquer atividade que exija lidar com diferentes perfis de pessoas (VAZQUEZ; SIMÕES, 2016).

Nesta seção, você viu que a especificação de requisitos é um documento que cuida de descrever os requisitos funcionais, requisitos não funcionais e regras negócio de um software usando uma linguagem natural, que pode ser compreendida tanto pelo cliente como pela equipe de desenvolvimento. Agora chegou a hora de você praticar um pouco. Leia atentamente o enunciado a seguir, destacando as informações que você julgar importantes, e elabore um documento de especificação de requisitos com base no modelo apresentado no Quadro 1.

EXERCÍCIO

Uma loja de perfumes deseja lançar um sistema de comércio eletrônico para vender seus produtos. A loja trabalha com perfumes de diversas categorias e aceita cartão de crédito e boleto bancário como formas de pagamento. Cada cliente deverá se cadastrar no sistema antes de realizar uma compra. O valor do frete será calculado por meio

de integração com o calculador remoto de preços e prazos dos Correios, e será gratuito para encomendas acima de R$ 500,00. O proprietário da loja deseja que o sistema forneça relatórios de estoque e de vendas. Para a implementação do sistema, deverá ser utilizada a linguagem de programação PHP e o sistema de gerenciamento de banco de dados MySQL. Supondo que você seja contratado para desenvolver esse sistema, elabore um documento de especificação de requisitos, fornecendo a lista dos requisitos funcionais, requisitos não funcionais e regras de negócio.

3.4 DETALHAMENTO DE CASOS DE USO

Se você analisar bem a especificação de requisitos apresentada no Quadro 1 da seção anterior, vai perceber que as listas de requisitos não funcionais e regras de negócio contêm descrições suficientemente explicativas. Por exemplo, "*O sistema dever ser compatível com os navegadores Google Chrome e Mozilla Firefox*" (RNF01) e "*Um aluno pode emprestar até três livros simultaneamente*" (RN01) são descrições que dificilmente causariam dúvidas ao leitor, seja um cliente ou desenvolvedor.

No entanto, a lista de requisitos funcionais não fornece informações suficientes para descrever o funcionamento do software. Ela descreve quais são as interações dos usuários com o software (cadastrar livros, consultar livros, ...), mas não revela os detalhes de como essas iterações ocorrem, dificultando o progresso para atividades de engenharia de software mais técnicas (PRESSMAN, 2016).

Assim, durante o processo de engenharia de requisitos, é comum que a atividade de especificação seja apoiada por um documento responsável por detalhar as funcionalidades propostas para o sistema. Esse documento contém um ou mais diagramas de casos de uso, que fornecem a visão geral das interações dos usuários com o software e o detalhamento de cada caso por meio de descrições estruturadas em tabelas e/ou outros diagramas (MARINHO, 2016).

Os diagramas usados no detalhamento de casos de uso fazem parte da Linguagem de Modelagem Unificada ou UML (*Unified Modeling Language*), utilizada para modelar o comportamento e a estrutura de sistemas de software. Estudaremos a UML mais detalhadamente na próxima seção. Agora vamos nos concentrar na Figura 23, que apresenta o diagrama de casos de uso do sistema de gestão de biblioteca, cujos requisitos foram definidos no Quadro 1 da seção anterior.

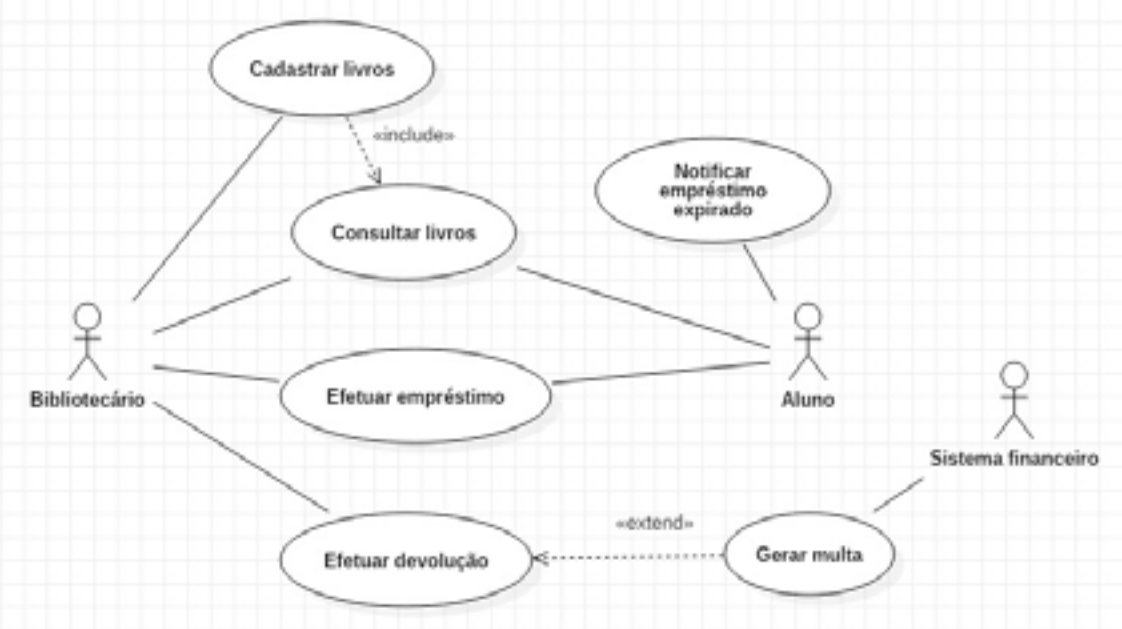


Figura 23: Diagrama de casos de uso do sistema de gestão de biblioteca.
Fonte: Elaborado pelo autor com o uso da ferramenta StarUML (2019).

As elipses representam os **casos de uso** do sistema. Perceba que os casos de uso que compõem o diagrama refletem a lista de requisitos funcionais do sistema (Quadro 1). Os bonecos de palito indicam os **atores**. Um ator representa um papel desempenhado por um usuário ou qualquer outra entidade externa, tal como um hardware ou outro sistema. No caso do sistema de gestão de biblioteca, além dos atores "bibliotecário" e "aluno", existe um ator denominado "sistema financeiro", que representa o sistema responsável pela gestão financeira da instituição de ensino. Ele interage trocando dados com o sistema de gestão da biblioteca quando o caso de uso "Gerar multa" é executado.

As linhas contínuas representam as **associações**. A execução de um caso de uso pode envolver a participação de um ou mais atores. Perceba que somente o ator "bibliotecário" participa do caso de uso "Cadastrar livros". Já o caso de uso "Efetuar Empréstimo" envolve tanto a participação do ator "bibliotecário" como do ator "Aluno", que deve fornecer suas credenciais no terminal de teclado para que o bibliotecário possa registrar o empréstimo no sistema.

Um diagrama de casos de uso também pode conter relações de **inclusão** (include) e **extensão** (extend) entre casos de uso. A relação de inclusão é utilizada quando a execução de um caso de uso inclui a execução de outro caso de uso como subcaso. No diagrama da Figura 23, o caso de uso "Consultar livros", que pode ser executado tanto pelo "bibliotecário" como pelo "aluno", é também um subcaso do caso de uso "Cadastrar livro", pois ao realizar o cadastro de um novo livro, o bibliotecário primeiro deve realizar uma consulta para verificar se o livro já está cadastrado.

A notação de extensão é usada para indicar um comportamento excepcional disparado por alguma condição. No exemplo da biblioteca, quando uma devolução é efetuada, o bibliotecário deve verificar se o prazo de devolução está expirado e, caso afirmativo, gerar uma multa que será enviada para o sistema financeiro da instituição.

Como você pode perceber, o diagrama de casos de uso é uma representação visual das funcionalidades do sistema. Ele facilita a compreensão da lista de requisitos do sistema e, consequentemente, a comunicação entre desenvolvedores e clientes. No entanto, o diagrama, por si só, ainda não fornece informações suficientes sobre o funcionamento do sistema. É necessário detalhar cada caso de uso por meio de descrições estruturadas, conforme o exemplo apresentado no Quadro 2.

CU01 – Cadastrar livros	
Descrição	Permite ao bibliotecário cadastrar novos livros no sistema de gestão da biblioteca.
Atores	Bibliotecário.
Requisitos	**RF01**. O sistema deverá permitir ao bibliotecário cadastrar livros.
Pré-condições	O bibliotecário deve estar autenticado no sistema.
Pós-condições	Um novo livro foi cadastrado.
Fluxo principal	
1. O bibliotecário executa o caso de uso "Consultar livros" (CU02) para verificar se o livro já está cadastrado. 2. O bibliotecário seleciona a opção "Cadastro de livros". 3. O bibliotecário seleciona a opção "Novo livro". 4. O bibliotecário informa os dados do livro (título, autores, edição, cidade, editora e ano) e sua classificação. 5. O bibliotecário seleciona a opção "Salvar".	
Fluxo alternativos	
FA01 - Livro já cadastrado: no passo 1, caso o livro já esteja cadastrado no sistema: 1.1 O bibliotecário seleciona a opção "Adicionar exemplar". 1.2 O bibliotecário seleciona a opção "Salvar" e o caso de uso é encerrado.	
Exceções	
E01 - Campos obrigatórios não informados: no passo 4, caso não sejam informados todos os dados obrigatórios, o sistema apresenta mensagem "Preencha todos os campos obrigatórios" e o bibliotecário repete o passo 4. E02 – Dados inválidos: no passo 4, caso algum dado esteja no formato inválido, o sistema apresenta mensagem "Dados inválidos" e o bibliotecário repete o passo 4. E03 – Livro já cadastrado: no passo 5, caso o livro já esteja cadastrado, o sistema apresenta mensagem "Livro já cadastrado" e o bibliotecário executa o fluxo alternativo FA01.	

Quadro 2 – Detalhamento do caso de uso "Cadastrar livros".
Fonte: Elaborado pelo autor (2019)

Inicialmente é fornecida uma breve descrição do caso de uso. Em seguida, são apresentados os atores, os requisitos, as pré-condições e as pós-condições do caso de uso. Caso haja alguma regra de negócio relacionada com a execução do caso de uso, também deve ser apresentada no campo "Requisitos".

Todo caso de uso possui um fluxo principal, que envolve as interações entre os atores e o sistema no cenário típico de execução, e pode possuir um ou mais fluxos alternativos, que envolvem interações opcionais ou interações que ocorrem quando determinadas condições são satisfeitas. As exceções descrevem o que acontece quando algo inesperado ocorre na interação entre um ator e o sistema.

O detalhamento de caso de uso pode conter um diagrama de sequência ou diagrama de atividades para facilitar a compreensão e/ou revelar mais detalhes sobre a execução do caso de uso, conforme o exemplo apresentado na Figura 24.

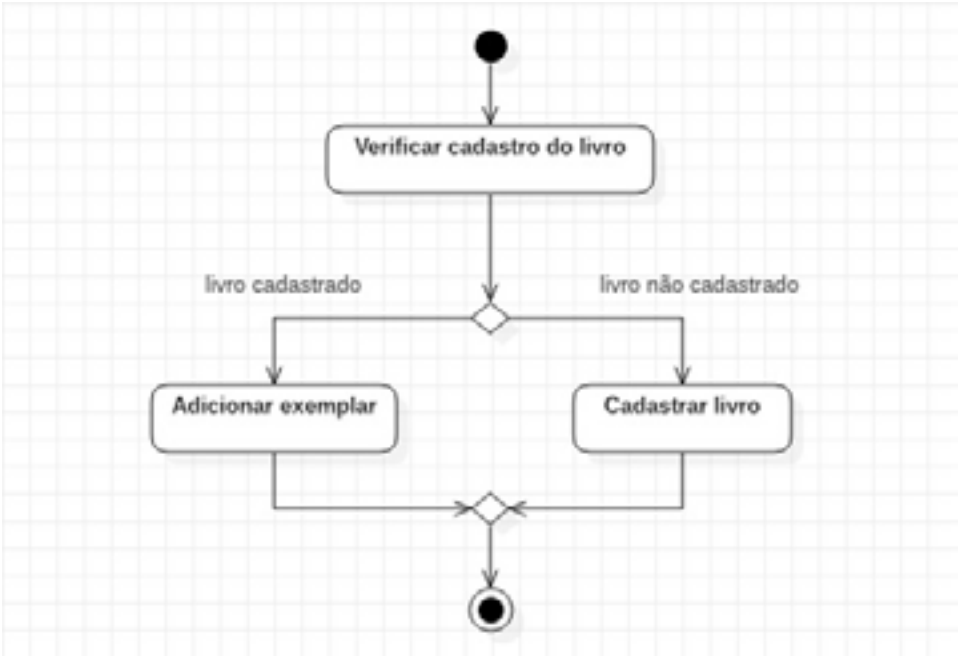


Figura 24: Diagrama de atividades "Cadastrar livro".
Fonte: Elaborado pelo autor (2019).

O diagrama de atividades da UML (*Unified Modeling Language*) é usado para modelar o comportamento de processos. Os retângulos arredondados representam as atividades que compõem o processo. As setas são usadas para indicar o fluxo de atividades, que inicia no ponto indicado por um círculo preenchido e termina no ponto indicado por um duplo círculo.

O losango é o símbolo usado para indicar tanto uma decisão como uma fusão. O que determina se o símbolo é uma decisão ou fusão é a quantidade de entradas e saídas. Enquanto uma decisão contém uma entrada e duas saídas, uma fusão pode conter duas

ou mais entradas e apenas uma saída.

No diagrama da Figura 24, após o bibliotecário verificar o cadastro do livro, é tomada a decisão por adicionar um novo exemplar (caso o livro já esteja cadastrado) ou cadastrar um novo livro (caso o livro ainda não esteja cadastrado). Seja qual for a atividade realizada após a decisão, o fluxo do processo é direcionado para um único caminho por meio da junção das saídas das atividades.

O detalhamento de um caso de uso também pode conter protótipos de interface de usuário para que os interessados possam vivenciar o sistema antes de ele ser desenvolvido, conforme mostra a Figura 25. Com a prototipação, os usuários e clientes podem identificar pontos fortes e fracos do software e obter novas ideias, bem como os desenvolvedores podem eliminar dúvidas quanto ao modo de projetar o software.

Figura 25: Protótipo da tela "Cadastro de livros".
Fonte: Elaborado pelo autor com o uso da ferramenta Lucidchart (2019).

Por meio de pesquisas na Web com os termos "Ferramenta de prototipagem", "Ferramenta de mockup" ou "Ferramenta de wireframe", você encontrará diversas ferramentas para a construção de protótipos de interface.

PARA REFLETIR

Nesta seção, você viu que o detalhamento de casos de uso pode conter protótipos de interface e alguns tipos de diagramas UML. No entanto, se todos os casos de uso de um software forem descritos com esse nível de detalhamento, o processo de engenharia de requisitos pode se tornar oneroso e levar o projeto ao atraso. O desafio está em decidir em que situações os protótipos de interface e/ou diagramas UML devem estar presentes em um caso de uso. Reflita sobre quais os fatores que podem (ou devem) influenciar nessa decisão.

EXERCÍCIO

Com base no modelo apresentado no Quadro 2, elabore um detalhamento para o caso de uso "Efetuar empréstimo" da Figura 23. Não esqueça que o aluno deve fornecer suas credenciais no terminal de teclado para que o bibliotecário possa registrar o empréstimo no sistema.

3.5 DIAGRAMAS UML

Imagine uma pessoa que deseja construir uma casa. Primeiro, ela se reúne com um arquiteto para definir como será a casa, discutindo aspectos como tamanho e distribuição dos ambientes, quantidade de vagas de garagem, tipo de acabamento e outras características que julgar importantes. Após essa fase, o arquiteto cuida de projetar a casa, elaborando plantas para que os profissionais da construção civil possam executar a obra.

Agora vamos transportar esse cenário para a engenharia de software. Quando um cliente deseja "construir" um software, primeiro ele se reúne com o engenheiro de software para definir o que o software deve fazer, quais os serviços que ele deve oferecer e quais as restrições ao seu funcionamento; ou seja, são definidos os requisitos do software. Após essa fase, o engenheiro cuida de projetar o software, elaborando as "plantas" para que os programadores possam "construir" o software.

Assim como os arquitetos, os engenheiros de software também fazem o uso de padrões de notação para a elaboração de "plantas". A diferença é que, no caso da engenharia de software, essas plantas são chamadas de **diagramas**, e o padrão de notação usado é a Linguagem de Modelagem Unificada ou **UML** (*Unified Modeling Language*). A UML é

uma linguagem visual (e não uma metodologia) utilizada para modelar as características estruturais e comportamentais de softwares (GUEDES, 2014). A Figura 26 apresenta o conjunto de diagramas da UML verão 2.5.1. São sete diagramas estruturais (sinalizados em amarelo) e sete diagramas de comportamento (sinalizados em verde).

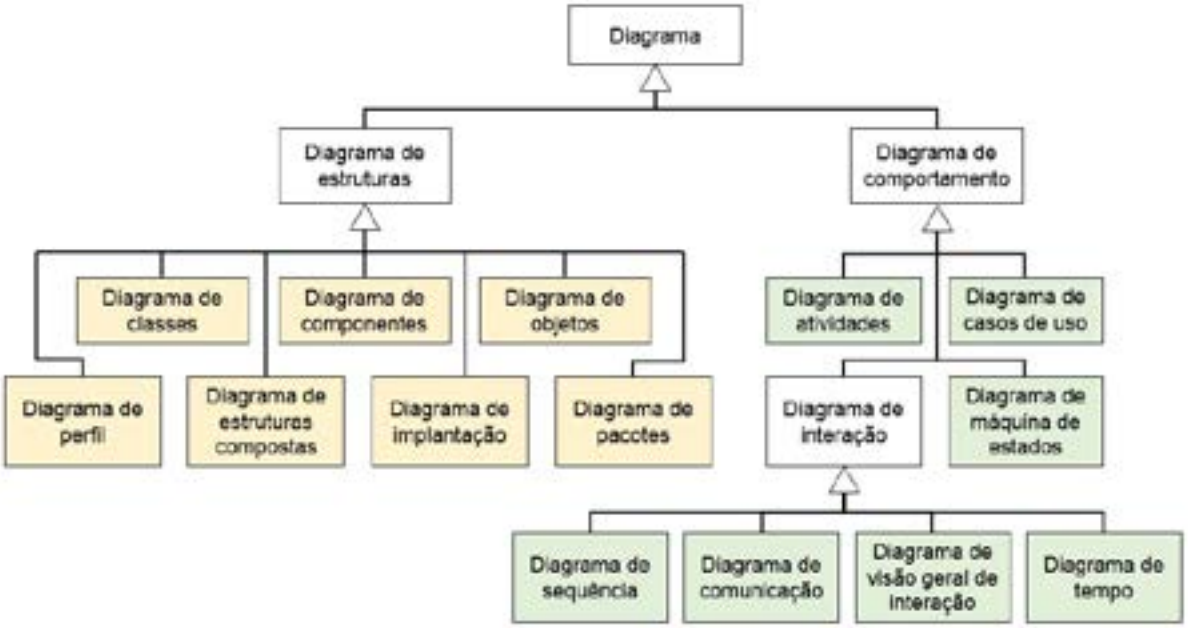


Figura 26: Taxonomia dos diagramas UML versão 2.5.1.
Fonte: Adaptado pelo autor a partir de OMG (2017).

Os diagramas estruturais são usados para descrever os elementos do sistema e seus relacionamentos (SILVA, 2009). Por exemplo, no projeto do sistema de gestão de biblioteca que estudamos nas seções anteriores, um diagrama de classes poderia ser usado para descrever que um objeto da classe livro possui os atributos título, edição, cidade, editora e ano, e que um livro pode se relacionar com um ou muitos objetos da classe autor. Também poderia ser usado um diagrama de implantação para descrever a interoperabilidade entre o sistema da biblioteca e o sistema financeiro, que ocorre quando uma multa por atraso na devolução de um livro é gerada.

Por sua vez, os diagramas de comportamento são usados para descrever como os elementos do sistema se comportam e interagem em tempo de execução (SILVA, 2009). Por exemplo, poderíamos utilizar um diagrama de máquina de estados para descrever os possíveis estados para um empréstimo de livro: (i) aguardando devolução; (ii) finalizado sem multa; (iii) não devolvido; (iv) finalizado com multa. Esse mesmo diagrama poderia ser usado para determinar que no estado "não devolvido", um empréstimo somente pode fazer a transição para o estado "finalizado com multa".

Como você deve ter percebido, alguns dos diagramas UML de comportamento podem

ser usados para apoiar o processo de engenharia de requisitos. Na Seção 3.3, foram usados os diagramas de casos de uso e de atividades para o detalhamento do caso de uso "Cadastrar livro". Vale lembrar que o diagrama de sequência também pode ser usado no detalhamento de casos de uso.

Um projetista não precisa fazer o uso dos 14 diagramas UML para descrever um software. A decisão sobre quantos e quais diagramas usar deve considerar a complexidade e as características de cada projeto, bem como as visões necessárias para que os desenvolvedores possam implementar o sistema. De acordo com Silva (2009), uma modelagem pode ser considerada completa se ela proporcionar as quatro visões apresentadas na Figura 27.

Figura 27: As quatro visões proporcionadas por uma modelagem de software completa.
Fonte: Adaptado pelo autor a partir de Silva (2009).

Essas quatro visões permitem que os desenvolvedores enxerguem o sistema sob quatro diferentes pontos de vista. O ponto de vista estrutural mostra a organização estrutural dos elementos do sistema e a estrutura interna de cada classe, enquanto o ponto de vista dinâmico mostra o comportamento de cada classe e do sistema como um todo. Sob outra perspectiva, o ponto de vista do sistema mostra a estrutura e o comportamento do sistema como um tudo, enquanto o ponto de vista da classe mostra a estrutura e comportamento de cada classe.

Obviamente, o projetista deve usar diagramas estruturais para fornecer visões estruturais, e diagramas de comportamento para fornecer visões dinâmicas. O Quadro 3 mostra quais diagramas podem ser utilizados para fornecer cada uma das quatros visões apresentadas na Figura 27.

Visão	Foco específico	Diagramas
Estrutural de sistema		De classes, de objetos, de pacotes, de estrutura composta, de componentes, de implantação, de perfil.
Estrutural de classe		De classes, de objetos.
Dinâmica de sistema	Funcionalidades do sistema	De casos de uso, de atividades, de máquina de estados, de visão geral de interação.
	Interação de objetos	De sequência, de comunicação, de tempo.
Dinâmica de classe	Modelagem de estados	De máquina de estados.
	Algoritmos de métodos	De atividades, de máquina de estados.

Quadro 3 – Classificação dos diagramas UML de acordo com as visões que eles fornecem.
Fonte: Adaptado pelo autor a partir de Silva (2009)

Com base na classificação apresentada, vamos analisar quão completa está a modelagem do sistema de gestão de biblioteca. Na Seção 3.4 (detalhamento de casos de uso), utilizamos os diagramas de casos de uso e de atividades para proporcionar uma visão dinâmica do sistema com foco específico nas funcionalidades do sistema. Para enriquecer o ponto de vista dinâmico, falta incluir um diagrama que proporcione uma visão dinâmica de classe. Considerando que o sistema não possui nenhum algoritmo ou método complexo, vamos focar na análise e modelagem dos estados das classes.

Os usuários podem estar com o acesso ao sistema bloqueado ou não. Da mesma forma, um livro pode estar reservado ou não. Até então, estamos falando de entidades que podem assumir apenas dois estados. É provável que grande parte dos projetistas (incluindo eu) dispensaria o uso de diagramas para descrever somente dois estados. São situações simples, que podem ser descritas nos próprios requisitos do sistema.

A entidade que mais pode gerar dúvidas quanto aos seus estados e transições é o empréstimo. Quando um empréstimo é efetuado, assume-se um estado que vamos chamar de "Aguardada devolução". Nesse estado, três eventos podem ocorrer: (i) a renovação do livro; (ii) a devolução do livro; e (iii) a expiração do prazo de devolução do livro. O que fazer em cada uma dessas situações?

O diagrama apresentado na Figura 28 responde a essa pergunta: (i) quando ocorre a renovação, o empréstimo permanece no mesmo estado; (ii) quando o livro é devolvido, o empréstimo passa para o estado "Finalizado sem multa"; (iii) quando o prazo de devolução do livro expira, o empréstimo passa para o estado "Não devolvido" e, a partir desse estado, só pode passar para o estado "Finalizado com multa".

Figura 28: Diagrama de máquina de estados do empréstimo de livro.
Fonte: Elaborado pelo autor com o uso da ferramenta StarUML (2019).

Agora que já temos diagramas que fornecem visões dinâmicas tanto do sistema como da classe, vamos analisar novamente o Quadro 3. Perceba que a modelagem do nosso sistema ainda não possui diagramas que forneçam visões estruturais do sistema. Perceba também que o diagrama de classes é capaz de fornecer tanto a visão estrutural de sistema como a visão estrutural de classe, ou seja, se adicionarmos esse diagrama, finalmente teremos uma modelagem completa.

Para elaborar um diagrama de classes, é necessário pensar no sistema como um conjunto de objetos que se relacionam entre si. Um objeto computacional pode ser tanto uma abstração (simplificação) de um objeto real (por exemplo, um livro) como a representação de algum conceito ou processo (por exemplo, um empréstimo). As classes fornecem as "plantas" para a construção desses objetos. Por exemplo, dizer que existem diversos objetos da classe "livro" em um sistema, significa dizer que todos esses objetos possuem a mesma "planta"; ou seja, o mesmo conjunto de atributos e o mesmo comportamento (ações que o objeto pode realizar).

A Figura 29 apresenta um diagrama de classes simplificado para o sistema de gestão de biblioteca. Antes de analisarmos a estrutura do diagrama, perceba que as palavras não foram acentuadas. Isso porque o diagrama de classes é usado pelos programadores na geração do código do sistema, e como você já deve saber, as linguagens de programação não permitem o uso de acentos.

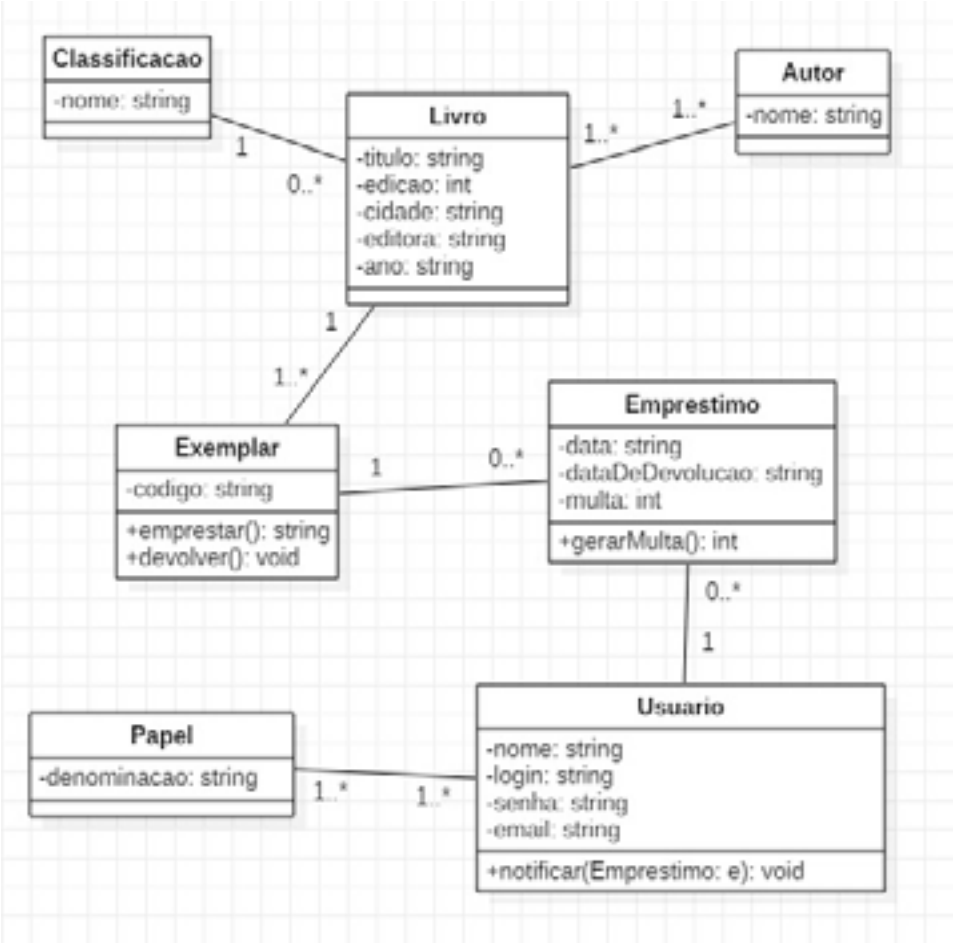


Figura 29: Diagrama de classes do sistema de gestão de biblioteca.
Fonte: Elaborado pelo autor com o uso da ferramenta StarUML (2019).

A primeira pergunta que você pode fazer ao analisar o diagrama de classes é: por que existe uma classe “Exemplar” e uma classe “Livro”? A resposta é simples: porque se o sistema for projetado para possuir apenas objetos do tipo “Livro”, sempre que o bibliotecário receber um livro que já existe na biblioteca, o sistema terá que armazenar novamente os dados do livro (título, edição, ...).

Com a inclusão da classe “Exemplar” para representar o livro físico em si, a classe “Livro” passa a ser uma representação dos dados de cadastro de um conjunto de exemplares iguais. O mesmo raciocínio vale para as classes “Papel”, “Classificacao” e “Ator”. Por exemplo, quando um novo usuário é cadastrado no sistema, se o seu papel (bibliotecário ou aluno) já estiver cadastrado no sistema, não é necessário repetir essas informações.

A descrição de uma classe é feita dentro de um retângulo com três áreas. A área superior apresenta o identificador (nome) da classe. A área central é utilizada para a descrição dos atributos da classe. A área inferior mostra quais são os métodos da classe, ou seja, quais são os comportamentos que os objetos da classe podem apresentar. Por

exemplo, os objetos da classe “Exemplar” possuem os métodos “emprestar” e “devolver”, enquanto os objetos da classe “Empréstimo” possuem o método “gerarMulta”, para ser utilizado quando a devolução ocorrer fora do prazo.

As linhas contínuas e as notações a elas associadas descrevem como as classes estão relacionadas. Por exemplo, um usuário pode efetuar zero ou muitos (0.*) empréstimos, ao mesmo tempo que um empréstimo pode ser efetuado por apenas um (1) usuário. Por outro exemplo, um exemplar pode estar associado a um (1) cadastro de livro, ao mesmo tempo que um cadastro de livro pode estar associado a um ou muitos (1.*) exemplares.

Ufa! Terminamos a modelagem do sistema de gestão de biblioteca. Espero que você tenha curtido. A UML é um assunto suficiente para escrever um livro extenso. Por isso, ao longo da terceira unidade, busquei proporcionar a você uma visão geral da UML com o foco voltado para os diagramas mais utilizados em projetos de softwares corporativos.

Procure praticar um pouco. Existem diversas ferramentas para a construção de diagramas UML. Astah (<http://astah.net/download>), Star UML (<http://staruml.io/>) e Enterprise Architect (<https://sparxsystems.com/products/ea/>) são apenas alguns exemplos.

FÓRUM

Nesta unidade, você viu que diferentes tipos de documentos e diagramas UML podem ser usados ao longo do processo de desenvolvimento de um software. Utilize o fórum on-line da disciplina para propor uma discussão mais aprofundada sobre o diagrama de classes da UML. Pesquise e fale sobre os relacionamentos de herança, agregação e composição. Forneça exemplos de uso dessas notações.

SUGESTÃO DE LIVRO

SOMMERVILLE, Ian. Engenharia de software. 9. ed. São Paulo: Pearson Prentice Hall, 2011.

O livro “Engenharia de Software”, de Ian Sommerville, é uma obra clássica para os profissionais de engenharia de software e que fornece uma leitura enriquecedora para quem deseja se aprofundar nos temas explorados neste caderno de estudos. No livro, há um capítulo dedicado ao estudo da Engenharia de Requisitos.

SUGESTÃO DE VÍDEO

O vídeo "A importância da documentação e da tecnologia da informação", disponível no canal do IBGE (Instituto Brasileiro de Geografia e Estatística) do YouTube, mostra que desde os primórdios, os homens procuravam meios para documentar seus achados. O vídeo desperta a reflexão sobre a importância da documentação para que novos membros de uma equipe de desenvolvimento de software possam se inteirar de todos os acontecimentos e contribuir com eficiência nas atividades do projeto.

CONSIDERAÇÕES FINAIS

Nesta unidade, você viu que a documentação é uma atividade que apoia diversas fases do processo de desenvolvimento de um software. Na fase inicial, ela cuida de descrever as funcionalidades e as características do software para que o cliente esteja ciente do produto que será entregue e os projetistas possam ter informações suficientes para elaborar os documentos técnicos (diagramas UML) que, por sua vez, servem de base para que os desenvolvedores possam implementar o sistema.

A documentação de um software deve estar sempre completa e atualizada. Por exemplo, se durante a execução de um projeto surgirem novos requisitos, todos os documentos (inclusive diagramas) afetados pela inclusão desses requisitos devem ser atualizados. Por outro exemplo, se durante a fase de implementação o programador verificar a necessidade de uma classe que não está prevista no projeto, os diagramas estruturais devem ser atualizados.

Uma boa razão para garantir que a documentação de software esteja sempre atualizada é a rotatividade de pessoas a que uma equipe de desenvolvimento está sujeita. Você ainda lembra do caso que eu contei no início dessa unidade? Além disso, é praticamente impossível avançar para uma nova fase de desenvolvimento se os documentos da fase anterior não estiverem completos e atualizados.

A atividade de documentação não encerra com a fase de implementação. Na próxima unidade, você verá que a atividade de teste também produz documentos, sem falar que também faz o uso dos documentos estudados nessa unidade para que seja possível atestar que o software está se comportando como deveria.

EXERCÍCIO FINAL

1. A engenharia de requisitos é uma subárea da engenharia de software que faz o uso de diversas técnicas para elicitar, analisar, documentar, validar e manter requisitos de software de maneira completa, clara e precisa. O processo de engenharia de requisitos começa com a definição dos requisitos de negócio e realiza sucessivos refinamentos até alcançar a definição dos requisitos de sistema. Assim, a engenharia de requisitos pode ser vista como uma disciplina que aborda diferentes tipos ou "camadas" de requisitos.

Em relação aos tipos de requisitos abordados pela engenharia de software, avalie as afirmações a seguir:

I. Os requisitos de negócio descrevem as regras de negócio do software;
II. Os requisitos de usuários descrevem as necessidades de informação para que os usuários possam desempenhar suas tarefas;
III. Os requisitos de sistema são compostos por requisitos funcionais e requisitos não funcionais.

É correto o que se afirma em:

A. II, apenas.
B. III, apenas.
C. I e II, apenas.
D. I e III, apenas.
E. I, II e III.

2. Durante o processo de engenharia de requisitos, é comum que a atividade de especificação seja apoiada por um documento responsável por detalhar os casos de uso do sistema. Esse documento contém um ou mais diagramas de casos de uso e o detalhamento de cada caso por meio de descrições estruturadas em tabelas e/ou outros diagramas.

Em relação ao detalhamento de casos de uso, avalie as afirmações a seguir:

I. A descrição dos cenários de um caso de uso inclui o fluxo principal, os fluxos alternativos e as exceções;
II. Diagramas de atividades podem ser usados para detalhar casos de uso;
III. O detalhamento de um caso de uso pode conter protótipos de interface de usuário.

É correto o que se afirma em:

A. II, apenas.
B. III, apenas.
C. I e II, apenas.
D. I e III, apenas.
E. I, II e III.

3. A UML (*Unified Modeling Language*) ou Linguagem de Modelagem Unificada é uma notação utilizada para modelar sistemas computacionais por meio do paradigma de orientação a objetos. A versão 2.5.1 da UML fornece um conjunto de 14 diagramas para modelagem das características de um software, sendo 7 deles voltados para modelagem estrutural e 7 para modelagem de comportamento.

Em relação ao conjunto de diagramas da UML 2.5.1, analise as seguintes afirmações:

I. Os diagramas estruturais são usados para descrever os elementos (classes, objetos, componentes) do sistema e seus relacionamentos;
II. Os diagramas de comportamento são usados para descrever como os elementos do sistema se comportam e interagem em tempo de execução;
III. O diagrama de classes é um diagrama estrutural, e o diagrama de casos de uso é um diagrama de comportamento.

É correto o que se afirma em:

A. II, apenas.
B. III, apenas.
C. I e II, apenas.
D. I e III, apenas.
E. I, II e III.

REFERÊNCIAS

GUEDES, G. T. A. **UML 2**: Guia prático. 2. ed. São Paulo: Novatec, 2014.

IBM - INTERNATIONAL BUSINESS MACHINES. **Documento de Visão**. 2019. Disponível em: <https://www.ibm.com/support/knowledgecenter/pt-br/SSWMEQ_4.0.6/com.ibm.rational.rrm.help.doc/topics/r_vision_doc.html>. Acesso em: 23 set 2019.

MARINHO, Antonio Lopes (org.). **Análise e modelagem de sistemas**. São Paulo: Pearson Education do Brasil, 2016.

OMG - OBJECT MANAGEMENT GROUP. **Unified Modeling Language version 2.5.1**. 2017. Disponível em: <https://www.omg.org/spec/UML/2.5.1/PDF>. Acesso em: 23 set 2019.

PAULA FILHO, Wilson de Pádua. **Engenharia de software**: fundamentos, métodos e padrões. 3.ed. Rio de Janeiro: LTC, 2009. ISBN 978-85-216-1650-4.

PFLEEGER, S. L. **Engenharia de software**: teoria e prática. 2. ed. São Paulo: Prentice Hall, 2004.

PRESSMAN, Roger S.; MAXIM, Bruce R. **Engenharia de Software**: uma abordagem profissional. 8. ed. Porto Alegre: AMGH, 2016.

SILVA, R. P e. **Como modelar com UML**. Florianópolis: Visual Books, 2009.

SOMMERVILLE, Ian. **Engenharia de software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.

VAZQUEZ, C. E.; SIMÕES, G. S. **Engenharia de requisitos**: software orientado ao negócio. Rio de Janeiro: Brasport, 2016.

unidade 4

TESTE DE SOFTWARE

4 INTRODUÇÃO À UNIDADE

Na primeira unidade, você viu que a engenharia de software é uma disciplina que tem como pedra fundamental o comprometimento com a qualidade. A garantia da qualidade de software engloba uma ampla gama de preocupações e atividades, tais como a adoção de padrões ISO (*International Organization for Standardization*) e IEEE (Instituto de Engenheiros Eletricistas e Eletrônicos), a realização de revisões e auditorias, a administração de riscos e segurança, o gerenciamento de mudanças e fornecedores, a educação da equipe de desenvolvimento e o **teste de software** (PRESSMAN; MAXIM, 2016).

Assim, o teste de software é uma das atividades (e não a única) que contribui para a entrega de softwares de qualidade. Mas o que é teste de software? Quais os seus objetivos específicos? É necessário planejar e documentar testes de software? Quando, como e o que devemos testar? Existem ferramentas para auxiliar o testador no processo de teste? Essas e muitas outras questões serão respondidas ao longo da quarta unidade deste caderno de estudos. Ao final da leitura, você deve ser capaz de:

- Compreender a importância dos testes de software para o desenvolvimento e a qualidade de software;
- Identificar tipos e estratégias de testes adequados em diferentes cenários de desenvolvimento de software;
- Projetar testes de softwares.

4.1 TESTE DE SOFTWARE

Quando eu iniciei minhas atividades de programador, na década de 1980, uma das minhas preocupações era fazer constantes revisões no código fonte dos sistemas em desenvolvimento para encontrar erros de lógica no encadeamento das instruções. Eu estava ciente que a entrega de um sistema com anomalias poderia gerar uma série de inconvenientes, tais como a falha do sistema durante um processo de negócio importante para a empresa (por exemplo, uma venda), a insatisfação do cliente e a demanda por retrabalho.

É natural que programadores demonstrem uma significativa preocupação com o funcionamento correto dos sistemas que eles implementam, afinal, ver uma "obra" de autoria própria ganhando vida é uma das principais fontes de motivação para qualquer pessoa que trabalha com desenvolvimento de software. Mas então por que lemos e

ouvimos tantos relatos de "bugs" (defeitos) de software? Por que os softwares que usamos do dia a dia algumas vezes falham?

A demanda por softwares está cada vez mais exigente, tanto em relação às tecnologias que os softwares devem usar quanto às funcionalidades que eles devem fornecer. Atualmente, até mesmo o desenvolvimento de softwares considerados "simples" pode demandar a produção de dezenas de milhares de linhas de código. O principal desafio para garantir o funcionamento correto desses grandes volumes de código não está em realizar a revisão gramatical do código (isto, as ferramentas de desenvolvimento fazem), mas em identificar e testar todos os possíveis caminhos que o sistema poderá percorrer durante a sua execução, e esta é uma tarefa difícil até mesmo para programadores mais experientes.

Nesse contexto, a disciplina de teste de software desempenha um importante papel na garantia da qualidade de software. Ela fornece um arcabouço de métodos, técnicas e ferramentas que permitem às equipes de desenvolvimento de software planejar, executar e documentar testes de software de maneira eficaz. Mas, afinal, o que é exatamente um teste de software?

De acordo com Sommerville (2011, p. 144), o teste é destinado a "*mostrar que um programa faz o que é proposto a fazer e para descobrir os defeitos do programa antes do uso*". Por sua vez, Paula Filho (2009, p. 349) define teste como sendo "uma atividade na qual um produto, sistema ou componente é executado sob condições especificadas, com observação e registro dos resultados e avaliação de um ou mais aspectos".

Existem três aspectos importantes nessas definições. O primeiro (e mais óbvio) é que o objetivo de um teste de software é descobrir defeitos. Por um lado, apesar de o teste de software ser um indicador de qualidade, não tem a capacidade de garantir que um software é livre de defeitos (SOMMERVILLE, 2011). Por outro lado, a identificação de um elevado número de defeitos em um software indica a necessidade de revisão do projeto, pois quanto maior o número de defeitos, maior é a chance de haver defeitos não detectados (PAULA FILHO, 2009).

O segundo aspecto é que testes devem ser executados sob condições especificadas. Em termos mais detalhados, toda atividade de teste se baseia em uma especificação, que deve: (i) indicar a parte do sistema a ser testada; (ii) descrever os procedimentos a serem realizados (o roteiro do teste); (iii) indicar os dados a serem fornecidos para o sistema; e (iii) indicar as saídas esperadas, ou seja, as saídas que devem fornecidas pelo sistema caso ele esteja funcionando corretamente (PAULA FILHO, 2009; SOMMERVILLE, 2011).

O terceiro aspecto é que o teste de software deve ser realizado antes de o software entrar em operação no ambiente de produção. Existem vários tipos de testes que podem ser realizados ao longo do processo de desenvolvimento de software, mas antes de

estudar o funcionamento de cada um deles, vamos conhecer um pouco da terminologia utilizada na área de teste de software.

4.2 ERRO, DEFEITO E FALHA

Erro, defeito e falha são termos usados para indicar problemas na qualidade de software. Existem diversos entendimentos sobre o significado desses termos, pois a distinção entre eles permanece confusa até mesmo quando consultamos o bom e velho Dicionário Aurélio (FERREIRA, 2001, p. 336), que define o termo "falha" como sendo um defeito.

Para fazer a distinção entre erro e defeito, Perssman (2016) utiliza o critério temporal "entrega do software". Segundo o autor, os erros são problemas encontrados antes de o software ser liberado aos usuários finais. Após esse evento, os problemas encontrados passam a ser considerados defeitos. Por sua vez, Schach (2010) entende que defeito é um termo genérico, que pode ser referir a um erro, uma falha ou uma imperfeição. E, finalmente, Gonçalves et al. (2019) e Braga (2016) fazem a distinção entre os termos erro, defeito e falha conforme mostra a Figura 30.

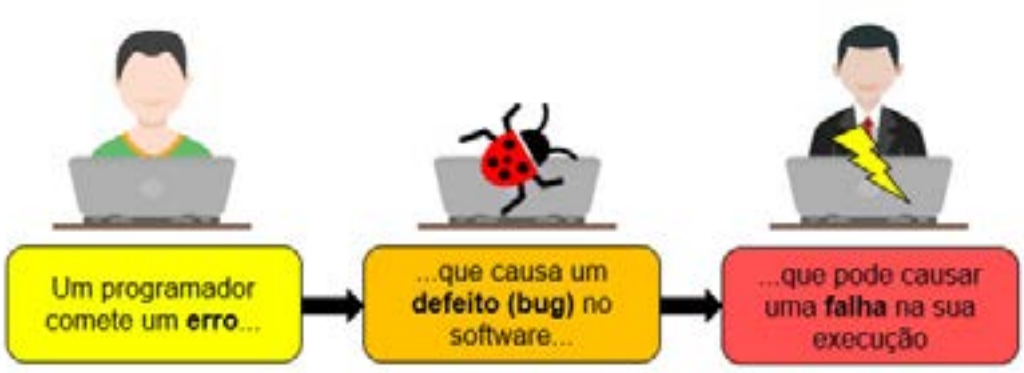


Figura 30: Erro, defeito e falha.
Fonte: Adaptado pelo autor a partir de Braga (2016), Gonçalves et al. (2019) e imagens de https://elements.envato.com/50-professionals-flat-multicolor-icons-SXSHQQ. Acesso em 13 setembro de 2019.

Um programador comete um erro, que causa um defeito (também referido como bug) no software, que por sua vez pode apresentar uma falha na execução. Em outras

palavras, um erro decorre da ação humana, tal como uma imperfeição na escrita de um código-fonte cometida por um programador. Essa instrução ou comando incorreto causa um defeito que somente é percebido pelo usuário final quando for gerada uma falha na execução do software.

Seja qual for a maneira escolhida para interpretar os termos erro, defeito e falha, o que realmente importa para uma equipe de desenvolvimento de software é encontrar essas imperfeições antes que os usuários finais o façam (PRESSMAN; MAXIM, 2016). Existem diversas situações em que falhas de software podem ser embaraçosas, caras e até mesmo destrutivas. Para você ter uma ideia, no ano de 1962, um pequeno erro de programação no foguete que deveria levar a sonda Mariner 1 para Vênus fez como que o veículo se desviasse do seu curso, causando um prejuízo de muitos milhões de dólares.

SAIBA MAIS

Nesta seção, você viu que um erro de programação causou uma falha no lançamento da sonda Mariner 1. Pesquise mais duas falhas de software que marcaram a história e descreva o cenário, a causa e o custo de cada uma delas. Podem ser utilizados sites na Internet.

4.3 VERIFICAÇÃO E VALIDAÇÃO

Na introdução desta unidade, você viu que a garantia da qualidade de software engloba uma ampla gama de atividades, tais como revisões, auditorias e testes. Esse conjunto de atividades que cuida da garantia da qualidade de software ou SQA (*Software Quality Assurance*) é denominado Verificação e Validação (V&V).

Verificação e validação são coisas distintas. Enquanto a verificação cuida de garantir que o software está sendo desenvolvido da maneira correta (conforme a sua especificação), a validação cuida de garantir que o software em desenvolvimento é o software certo, conforme ilustrado na Figura 31.

Figura 31: Verificação e validação.
Fonte: Adaptado pelo autor a partir de Sommerville (2011) e imagens de https://elements.envato.com/50-professionals-flat-multicolor-icons-SXSHQQ. Acesso em 13 setembro de 2019.

Em termos mais específicos, a verificação é a atividade que cuida de garantir que o fluxo de trabalho que implementa os requisitos funcionais e não funcionais do software foi executado corretamente; ou seja, verifica se a implementação está consistente com a especificação de requisitos que estudamos na Seção 3.2 (PEZZÈ; YOUNG, 2008; PRESSMAN; MAXIM, 2016; SCHACH, 2010; SOMMERVILLE, 2011).

Por sua vez, a validação cuida de garantir que a especificação de requisitos atende às reais necessidades do cliente, ou seja, busca demonstrar que o software faz o que o cliente espera que ele faça (PEZZÈ; YOUNG, 2008; SOMMERVILLE, 2011). A validação é uma atividade essencial para a qualidade de software, pois as especificações são escritas por pessoas e, consequentemente, podem conter erros e/ou não refletir as reais necessidades ou desejos dos clientes e usuários (PEZZÈ; YOUNG, 2008; SOMMERVILLE, 2011).

Fazendo analogia com a construção de uma casa, a atividade de verificação cuida de garantir que o mestre de obras está executando a construção conforme as plantas e projetos que foram fornecidos, enquanto a atividade de validação cuida de garantir que o arquiteto desenvolveu as plantas e projetos conforme os desejos das pessoas que residirão na casa. Em resumo, o objetivo final da V&V é garantir que o software está pronto para uso. Embora o teste de software tenha um papel extremamente importante em V&V, muitas atividades são necessárias para garantir a qualidade de software (PRESSMAN; MAXIM, 2016).

PARA REFLETIR

Nesta seção, você viu que o teste de software é uma das atividades que integram a Verificação e Validação (V&V) de software. Com base no que estudamos até agora, reflita sobre as contribuições que o teste de software pode fornecer para o processo de V&V.

4.4 DIMENSÕES DE TESTE

A definição de um teste de software passa pela decisão sobre qual aspecto do software será testado (o que testar), qual técnica de teste será usada (como testar) e em que nível ou fase do desenvolvimento o teste será realizado (quando testar). Essas três variáveis formam as dimensões do teste de software apresentadas na Figura 32.

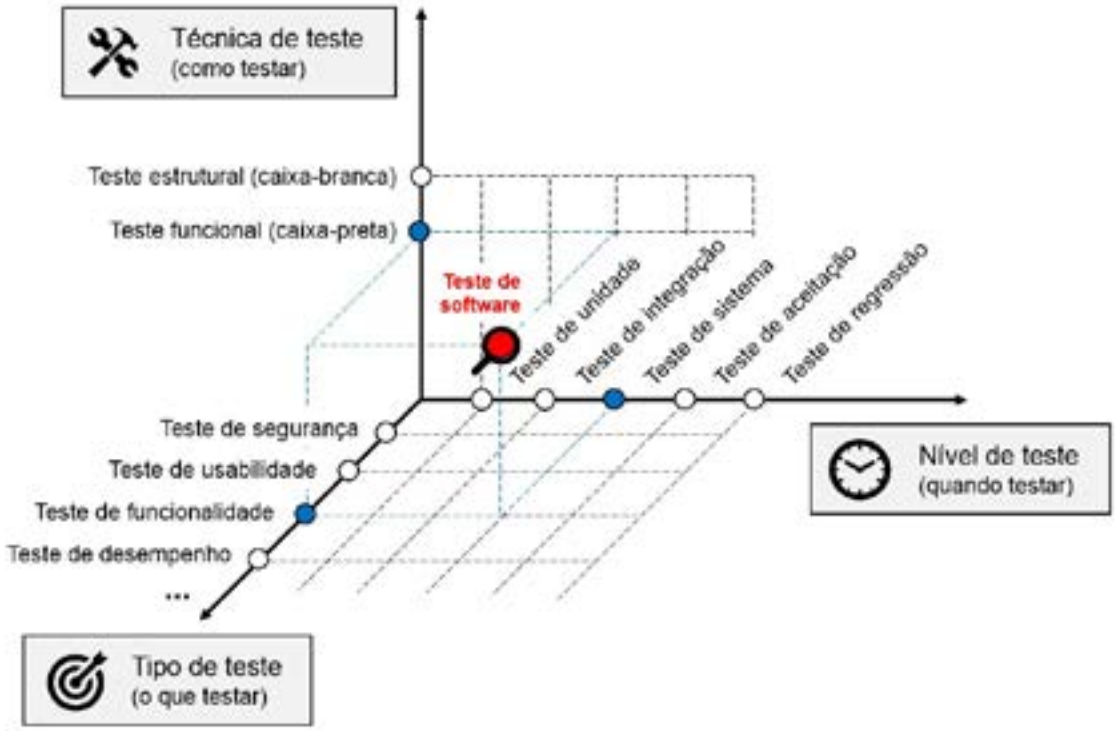


Figura 32: Dimensões do teste de software.
Fonte: Adaptado pelo autor a partir de Braga (2016).

Nas subseções a seguir, estudaremos cada uma dessas dimensões, a começar pelo nível de teste.

4.4.1 Níveis de teste

Se você retornar à segunda unidade deste caderno de estudos e observar as imagens dos modelos de processo de desenvolvimento de software apresentadas ao longo das seções (Figura 10, Figura 13 e Figura 15), verá que o teste de software é uma atividade que normalmente está posicionada após a atividade de implementação e antes da atividade de operação de manutenção de software.

A mera observação desses diagramas de fluxo pode levar ao entendimento errado de que testes de software devem ser realizados somente quando o software está pronto. Mas, pense bem: é razoável que um programador tenha que finalizar todo o código-fonte de um sistema para só então testar as estruturas de dados, estruturas de controle e sub-rotinas que implementou? Não faz sentido, não é mesmo?

Na verdade, o que se quer expressar pelas representações gráficas dos modelos de processo de desenvolvimento de software não é que o teste de software é uma etapa de uma sequência encadeada, mas sim uma atividade que normalmente está mais ativa nas fases finais do processo. E, realmente, a realização de grande parte dos testes de software (por exemplo, teste de aceitação) requer que o sistema esteja pronto.

O modelo de Processo Unificado (Figura 18) apresentado na Seção 2.6 deixa esse entendimento mais claro ao apresentar uma perspectiva dinâmica do processo de desenvolvimento de software. Ele mostra que o *workflow* (fluxo de trabalho) de teste está presente em todas as fases do processo de desenvolvimento de software, porém, com maior atividade entre as fases de construção e transição.

Assim, conforme um software "cresce", diferentes níveis de teste podem ser realizados. Essa perspectiva temporal da atividade de teste é apresentada na Figura 33. No início do processo de desenvolvimento de software, a atividade de teste está mais concentrada nas unidades (classes, funções, métodos) do sistema. Em seguida, cuida de testar a integração dessas unidades para verificar se estão se comunicando corretamente. Após a fase de implementação, o sistema é testado como um todo e submetido à aceitação do cliente e dos usuários. Caso haja alguma alteração ou incremento no software após a entrega, é realizado um teste de regressão para verificar se as novas implementações afetaram o funcionamento do software.

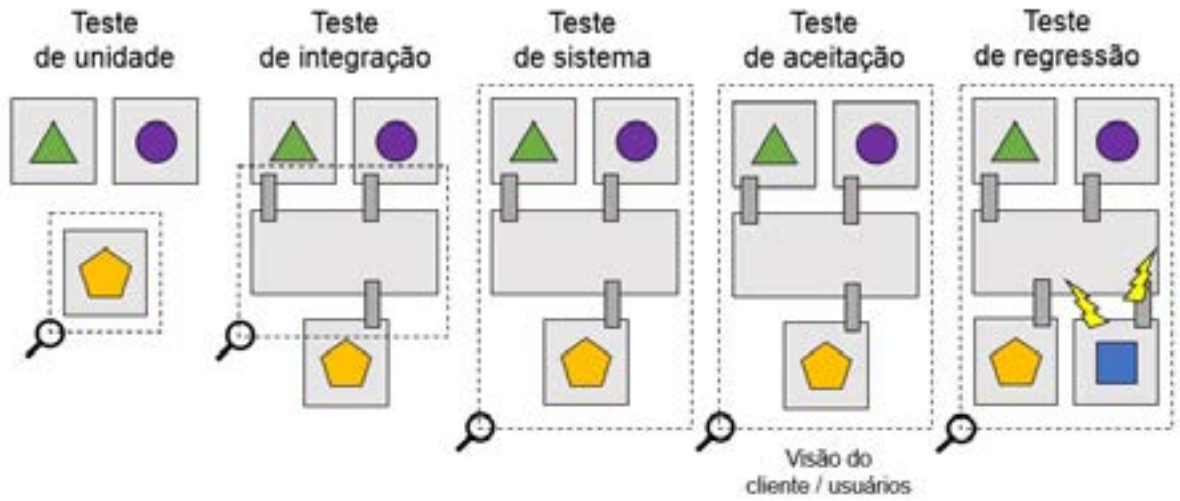


Figura 33: Níveis de teste de software.
Fonte: Elaborado pelo autor (2019).

O **teste de unidade,** ou teste unitário, cuida de testar as menores partes em que um software pode de ser dividido, tais como chamadas de funções e procedimentos (GONÇALVES et al., 2019). Quando se trabalha com o paradigma de orientação a objetos, o teste de unidade deve testar todos os atributos e métodos do objeto e colocá-lo em todos os possíveis estados (SOMMERVILLE, 2011).

Se tomarmos como exemplo o sistema de gestão de biblioteca que projetamos na Seção 3.4 (Figura 29), o teste de unidade da classe "Empréstimo" deve testar se a implementação garante que o valor do atributo "data" (do empréstimo) nunca seja maior que o valor do atributo "dataDeDevolução". O teste de unidade também deve testar se a implementação do método "gerarMulta" garante que a multa somente seja gerada se o empréstimo estiver no estado "Não devolvido" (Figura 28).

O teste de unidade é normalmente realizado pelos próprios programadores, sem que seja necessário documentar de maneira formal os problemas encontrados e as correções realizadas (GONÇALVES et al., 2019). Por sua vez, o **teste de integração** cuida de testar se as unidades testadas no nível anterior têm a capacidade de funcionar em conjunto para formar um sistema. No caso do sistema de gestão de biblioteca, por exemplo, o teste de integração deve testar se, quando o método "emprestar" do objeto "Exemplar" é executado, o sistema cria um novo objeto "Empréstimo" com o valor do atributo "data" igual a data atual. O teste de integração também deve testar se, ao executar o método "devolver" do objeto "Exemplar ", o atributo "dataDeDevolução" do objeto "Empréstimo" é atualizado com o valor da data atual.

Parece óbvio que se as unidades de um sistema funcionam corretamente de maneira isolada funcionarão corretamente em conjunto. No entanto, os testes de unidade podem não ser capazes de identificar situações como a incompatibilidade de tipos de dados entre

objetos, a imprecisão numérica (que somente é percebida quando propagada por diversas sub-rotinas), dentre outros problemas que se tornam mais evidentes quando as unidades de um sistema são colocadas para funcionar em conjunto (PRESSMAN; MAXIM, 2016). É importante que a realização de testes de integração envolva tanto programadores como analistas de testes.

No **teste de sistema**, as unidades também são testadas em conjunto. Porém, esse nível de teste considera a visão do usuário final, ou seja, os casos de uso do sistema. Por isso, as atividades de teste de sistema devem ser realizadas em cenários semelhantes àqueles em que o sistema funciona, levando em consideração as interfaces, as tecnologias, as massas de dados e qualquer outro aspecto relacionado com a operação do sistema no dia a dia do negócio (GONÇALVES et al., 2019). De acordo com Pressman e Maxim (2016), o teste de sistema é uma série de diferentes testes cuja finalidade principal é exercitar totalmente o sistema. Por isso, é natural que o nível de teste de sistema possa ser realizado somente quando o sistema ou incremento estiver totalmente implementado.

Para verificar se o software está pronto para entrar em produção, é realizado o **teste de aceitação**. Normalmente, esse teste é conduzido por um testador indicado pelo cliente e conta com a participação de um grupo formado por representantes dos usuários finais, que são responsáveis por simular as operações que o software deve executar de maneira rotineira (GONÇALVES et al., 2019). Os usuários devem verificar se o comportamento do software está de acordo com os requisitos levantados. Assim, o principal objetivo do teste de aceitação não é identificar defeitos, mas determinar se o software é ou não aceito da forma como ele está sendo entregue (GONÇALVES et al., 2019).

Por fim, o **teste de regressão** é um nível de teste realizado quando uma nova versão do software é desenvolvida. O objetivo é verificar se alterações ou incrementos realizados no código-fonte do software não afetam as partes do software que já estavam em funcionamento (GONÇALVES et al., 2019).

4.4.2 Técnicas de teste

As técnicas de testes definem como o software é testado, ou seja, a maneira como os testadores manipularam o software para encontrar defeitos. Neste sentido, quando se fala em testar um software, a primeira ideia que provavelmente vem à mente da maioria das pessoas leigas em desenvolvimento é a de executar as funcionalidades que o software fornece e verificar se funcionam corretamente.

Essa intuição natural sobre como testar um software é o que chamamos de **teste**

funcional ou **teste de caixa-preta**. A Figura 34 mostra o funcionamento da abordagem de teste de caixa-preta. O testador deve selecionar a funcionalidade que será testada, realizar os procedimentos definidos do roteiro do caso de teste e verificar se o resultado fornecido pelo sistema é o resultado esperado. Se o resultado fornecido for o esperado, entende-se que o teste foi bem-sucedido.

Por exemplo, vamos imaginar que estamos definindo um caso de teste para o fluxo principal do caso de uso "Cadastrar livros" (Seção 3.4, Quadro 2). Como procedimentos, definimos que o testador deve (1) entrar na tela "Cadastro de livros", (2) selecionar a opção "Novo livro", (3) informar os dados de cadastro do livro (título, autores, edição, cidade, editora e ano) e (4) clicar no botão "Salvar". Como resultado esperado, definimos o sistema deve incluir um novo livro na listagem de livros.

Casos de teste			
Funcionalidade	Cenário	Procedimentos (roteiro)	Resultado esperado
Cadastrar livro	Fluxo principal	1. Entrar na tela...2...	O sistema deve incluir...
Cadastrar livro	Livro já cadastrado	1. Selecionar a opção...2...	O sistema deve atualizar...
Cadastrar livro	Dados inválidos	1. Clicar no botão...2...	O sistema deve negar...

Figura 34: Teste de caixa-preta.
Fonte: Elaborado pelo autor (2019).

Como você pode perceber, o teste funcional não está preocupado em saber como o código-fonte foi escrito (por isto, o nome caixa-preta), mas em determinar se todos os requisitos do software funcionam da maneira como foram elicitados e documentados (GONÇALVES et al., 2019). De acordo com Pressman e Maxim (2016), essa técnica de teste é aplicada para encontrar funcionalidades faltantes ou incorretas, erros na interface, erros de estruturação de dados, erros de acesso a dados externos, erros de comportamento e desempenho, bem como problemas com o início e o fim da execução das tarefas.

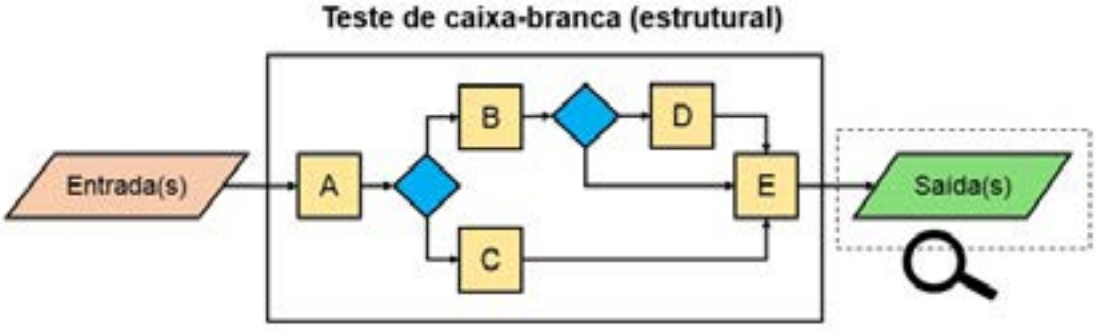
O teste caixa-preta tende a ser aplicado em níveis mais avançados do processo de

desenvolvimento de software. Sua principal vantagem é que a equipe de testadores não precisa conhecer as linguagens de programação usadas na implementação do código-fonte, precisa apenas verificar se o que foi produzido está de acordo com o que foi prometido aos usuários (GONÇALVES et al., 2019).

Agora, imagine que o teste do fluxo principal da funcionalidade "Cadastrar livro" (Figura 34) tenha sido bem-sucedido, e que alguns meses após a entrega do software a mesma funcionalidade tenha apresentado uma falha durante o cadastro de um novo livro. Quais seriam as suas hipóteses acerca do motivo da falha? Bom, é possível que o software tenha realizado algum caminho lógico diferente daquele realizado durante o teste de caixa-preta, em função do novo conjunto de dados (título, autores, edição, cidade, editora e ano) fornecidos pelo bibliotecário.

Se for esse mesmo o motivo da falha, poderia ter sido identificado antes da entrega do software por meio de um **teste estrutural** ou **teste de caixa-branca**. O teste estrutural é uma técnica que busca verificar se todos os possíveis caminhos (chamadas de funções, laços de repetição, decisões lógicas, cálculos e demais tipos de processamento) que podem ser realizados pelo software estão em perfeito funcionamento.

A Figura 35 ilustra o funcionamento do teste estrutural. As letras representam os processos que a unidade ou sistema implementa, e os losangos em azul as decisões que determinam quais processos devem ser realizados. São fornecidos diferentes conjuntos de dados de entrada, de tal forma que todos os possíveis caminhos que possam ser realizados sejam testados.



Casos de teste		
Dado(s) de entrada	Caminho	Saída(s) esperada(s)
{1,5,6}	A → B → D → E	{4,7}
{4,1,6}	A → B → E	{1,6}
{4,5,1}	A → C → E	{5,6}

Figura 35: Teste de caixa-branca.
Fonte: Elaborado pelo autor (2019).

A massa de dados de entrada também deve incluir valores inválidos, para que seja possível testar se o sistema está validando as entradas e tratando as exceções. Por exemplo, ao testar uma função que gera uma multa por atraso, o teste de caixa-branca deve fornecer um valor negativo para o parâmetro "dias em atraso". Se a função retornar um valor de multa negativo, significa que a implementação não cuidou de validar as entradas na função.

Em geral, a técnica de teste estrutural consegue identificar se um software foi implementado de maneira 100% correta, pois todos os caminhos lógicos possíveis são testados de maneira exaustiva e detalhada (GONÇALVES et al., 2019). No entanto, testar todos os possíveis caminhos de um sistema envolve o uso de grandes massas de dados. Por isso, testes estruturais normalmente são realizados com o auxílio de ferramentas que automatizam o processo de teste. Estudaremos sobre o funcionamento de testes automatizados na Seção 4.6.

Existe ainda na literatura uma técnica chamada de **teste de caixa-cinza**, na qual algumas partes da estrutura interna do software são observadas ao mesmo tempo que suas funcionalidades são testadas. Por exemplo, ao testar a funcionalidade "Cadastrar livro" (Figura 34), o testador poderia executar o roteiro do caso de teste e em seguida acessar a base de dados por meio de uma consulta SQL (Structured Query Language) para verificar se os dados do livro foram registrados corretamente.

4.4.3 Tipos de teste

Pode-se dizer que um software livre de erros de programação é um software de qualidade? A conclusão não é tão simples, pois a qualidade de um software quase sempre está associada a diversos critérios, que podem variar a cada projeto. Para uma instituição financeira, por exemplo, é importante que um software seja confiável, seguro e capaz de lidar com um grande volume de transações e usuários simultâneos. Por outro exemplo, usuários com pouca afinidade com informática esperam que um software seja de fácil uso.

Esses são alguns exemplos de critérios que determinam a qualidade de um software e, consequentemente, os tipos de teste a serem realizados ao longo do seu desenvolvimento, conforme ilustra a Figura 36.

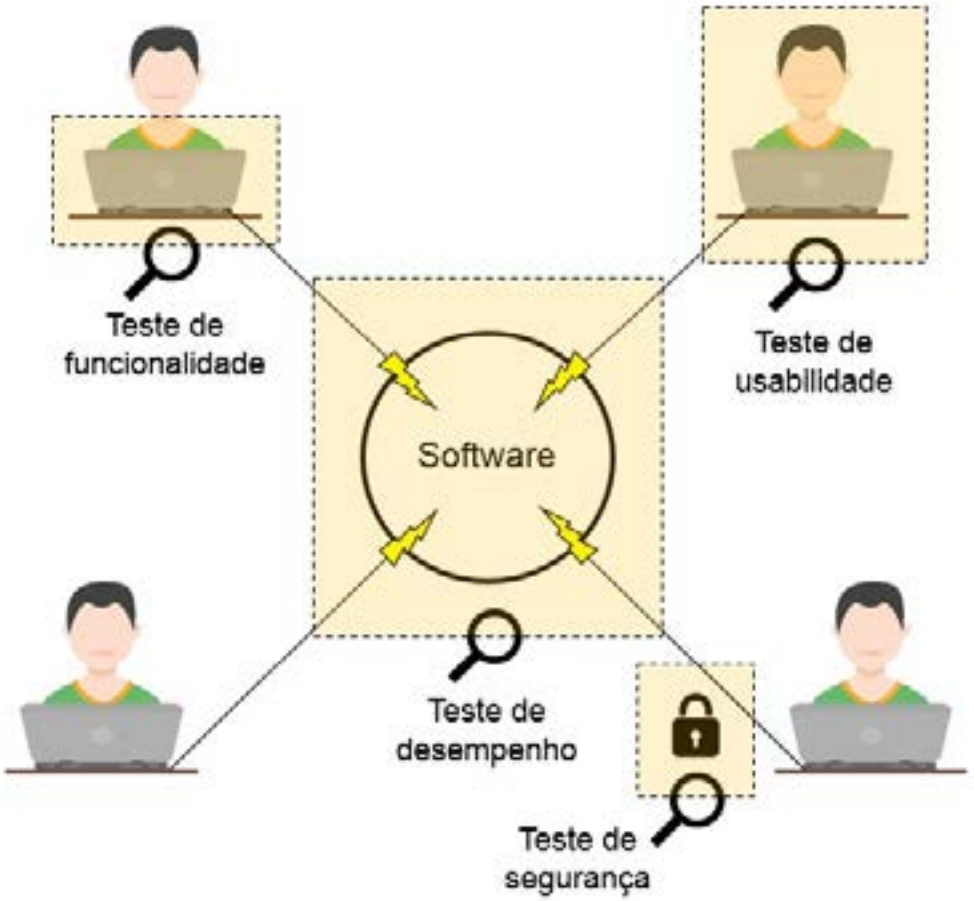


Figura 36: Tipos de teste de software.
Fonte: Elaborado pelo autor (2019) com imagens de https://elements.envato.com/50-professionals-flat--multicolor-icons-SXSHQQ. Acesso em 13 setembro de 2019.

Como exemplo prático, vamos analisar que tipos de teste são necessários para o sistema de gestão de biblioteca estudado na terceira unidade, tendo como critérios de qualidade os requisitos apresentados no Quadro 2. Certamente, o principal teste a ser realizado é o teste de funcionalidade, pois será responsável por garantir que o software atenda aos seus requisitos funcionais.

Também será necessário realizar um teste de desempenho, pois o segundo requisito não funcional (RNF02) determina que a consulta de livros não deve levar mais de 10 segundos para fornecer resultados. Logicamente, para que esse teste seja válido, é necessário que a base de dados seja alimentada com uma massa de dados simulados capaz de exigir da consulta um custo computacional semelhante ao que será exigido no dia a dia de operação da biblioteca.

Como teste de segurança, é necessário verificar se o controle de permissões do sistema

garante que somente um usuário com login e senha válidos pode acessar o sistema, e que um usuário autenticado somente pode executar as funcionalidades atribuídas ao seu papel dentro do sistema (por exemplo, somente o bibliotecário pode cadastrar livros). Testes de segurança também podem ser realizados para avaliar a vulnerabilidade do software frente a diferentes tipos de ataques de segurança.

Poderíamos ainda pensar em realizar um teste de usabilidade, para avaliar a experiência dos usuários, e um teste de configuração, para verificar o comportamento do sistema nos diferentes navegadores (Google Chrome e Mozilla Firefox) para os quais ele foi projetado.

Enfim, a relação de tipos de teste apresentada na Figura 36 não pretende ser exaustiva. Além disso, dentro de uma dimensão de teste existem testes específicos. Por exemplo, um teste de desempenho pode estar interessado em avaliar como o software se comporta em situações normais de uso (teste de carga), como ele se comporta com um grande número de acessos simultâneos (teste de stress) ou como comporta ao longo do tempo (teste de estabilidade).

EXERCÍCIO

A ISO/IEC 25010 é uma norma ISO que define as características de qualidade que os softwares devem possuir para alcançar um alto nível de qualidade. Pesquise e descreva quais são essas características. Podem ser utilizadas imagens e sites da Internet.

4.5 O PROCESSO DE TESTE

No início da quarta unidade, você viu que um teste de software pode ser definido como "*uma atividade na qual um produto, sistema ou componente é executado sob condições especificadas, com observação e registro dos resultados e avaliação de um ou mais aspectos*" (PAULA FILHO, 2009, p. 349). Note que essa definição envolve dois aspectos importantes: (i) um teste de software requer uma especificação que descreva as condições para a realização do teste; (ii) um teste de software deve ter seus resultados registrados e avaliados.

Esses aspectos permitem enxergar um teste de software como um processo que envolve um conjunto de atividades que utiliza e produz um conjunto de artefatos (dados e documentos), conforme mostra a Figura 37. O documento que descreve as

instruções para a realização de um teste é chamado de **caso de teste**. Ele descreve os procedimentos a serem realizados, os dados a serem fornecidos e as saídas que são esperadas após a execução dos procedimentos, além da declaração do que está sendo testado (SOMMERVILLE, 2011).

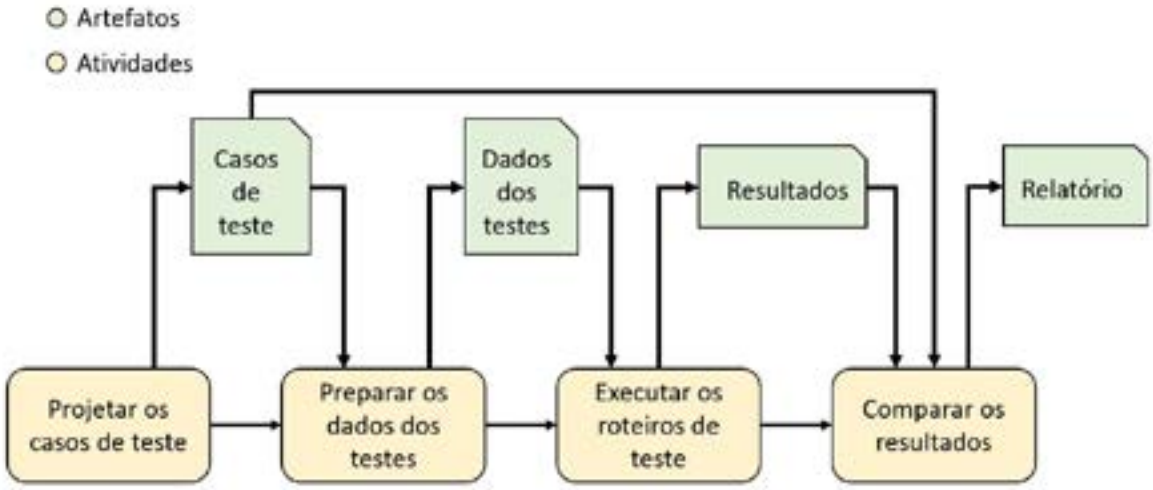


Figura 37: Modelo de processo de teste de software.
Fonte: Adaptado pelo autor a partir de Sommerville (2011).

No exemplo apresentado na Figura 34 (Seção 4.4.2), é possível perceber que um processo de teste pode envolver a execução de um ou vários casos de teste. Isso porque para testar uma funcionalidade do software, por exemplo, é necessário testar cada um dos fluxos e exceções previstos no documento de detalhamento de casos de uso que estudamos na Seção 3.4. A geração de casos de teste a partir de casos de uso UML é uma estratégia muito usada em testes de software.

Segundo Gonçalves et al. (2019), um documento de caso de teste normalmente é estruturado com os seguintes itens: (i) resumo; (ii) pré-condições; (iii) entradas; (iv) procedimentos; (v) resultados esperados, e; (vi) pós-condições. O resumo descreve a parte do software que será testada e a finalidade do teste. Uma pré-condição é uma declaração de um estado necessário para que o teste possa ser realizado, por exemplo, o testador estar autenticado no sistema.

As entradas são os valores (válidos e inválidos) que devem ser informados durante a execução do caso de teste. Os procedimentos descrevem as ações que devem ser realizadas para que o teste possa atingir os seus objetivos. Por fim, o caso de teste deve indicar os resultados esperados (por exemplo, o sistema deve apresentar a mensagem "OK" após a realização de um cadastro) e as pós-condições (por exemplo, existe um novo registro no cadastro do sistema). O Quadro 4 apresenta um exemplo de um típico documento de caso de teste.

CT001 – Login	
Resumo	Testa o procedimento de login no sistema com um usuário de teste.
Pré-condições	O usuário de teste está cadastrado no sistema. Não há usuários com sessão ativa no ambiente de teste.
Entradas	Nome: admin Senha: 123456
Procedimentos	1. Acessar a tela de login; 2. Inserir o dado "Nome" no formulário; 3. Inserir o dado "Senha" no formulário; 4. Clicar no botão "Login".
Resultados esperados	O sistema apresenta a tela "Dashboard".
Pós-condições	Existe um novo log de acesso ao sistema.

Quadro 4 – Exemplo de caso de teste.
Fonte: Elaborado pelo autor (2019)

Um documento de caso de teste também pode apresentar informações sobre o ambiente em que o teste deve ser realizado, exigências especiais para a realização do teste, relações de dependência com outros casos de teste e sobre a elaboração do próprio documento (nome do autor, data e versão).

Antes de iniciar um teste, é necessário criar a massa de dados do teste, com dados válidos e inválidos. Por exemplo, ao testar um cadastro de pessoa, a massa de dados deve incluir as entradas "Fulano de Tal" (entrada válida) e apenas "Beltrano" (entrada inválida), para que seja possível verificar se o campo de formulário "Nome completo" aceita apenas entradas com duas ou mais palavras.

Em seguida, o testador executa o roteiro de teste (procedimentos) e registra o resultado. Um caso de teste pode apresentar dois resultados: (i) teste positivo ou bem-sucedido, que demonstra que a funcionalidade está executando conforme o esperado e, consequentemente, o requisito foi atendido; (ii) teste negativo ou malsucedido, que demonstra que a funcionalidade não está executando ou está produzindo resultados inesperados (PAULA FILHO, 2009; GONÇALVES et al., 2019).

A última atividade do processo de teste consiste em tabular e documentar os resultados dos testes (quantidade de erros, defeitos e falhas encontrados) em forma de relatório, que será utilizado por outros profissionais posteriormente (GONÇALVES et al., 2019).

EXERCÍCIO

Com base no modelo apresentado no Quadro 4, elabore um caso de teste para a exceção 03 (livro já cadastrado) do caso de uso "Cadastrar livro", apresentado no Quadro 2.

4.6 TESTES AUTOMATIZADOS

Na Seção 4.4.2, vimos que o teste de caixa-branca ou teste estrutural é uma técnica de teste que busca verificar se todos os possíveis caminhos que podem ser realizados pelo software estão em perfeito funcionamento. Imagine a aplicação dessa técnica de teste em um software com dezenas de milhares de linhas de código. Quanto tempo levaria para testar manualmente todos esses possíveis caminhos? E no caso de haver alterações no software após a finalização dos testes, todo o trabalho teria que ser refeito?

Para nosso alívio, existem diversas ferramentas para a automatização de testes de software. Ao desenvolver um sistema com a linguagem de programação PHP, por exemplo, você pode usar o PHPUnit (https://phpunit.de/) para automatizar os testes de unidade. Ela fornece notações que permitem implementar os procedimentos e as massas de dados dos casos de teste, que por sua vez podem ser executados sempre que necessário com um comando simples.

No entanto, existem alguns tipos de teste de software que são mais difíceis de serem automatizados tendo em vista a natureza subjetiva do que está sendo testado. Os testes de usabilidade e aceitação, por exemplo, buscam verificar como o software será percebido pelo cliente e usuários sob diferentes aspectos. Uma ferramenta de automatização até pode substituir uma pessoa no papel de executar ações na interface de um software, mas dificilmente pode substituir uma pessoa no papel de avaliar se o software é "bom" (teste bem-sucedido) ou "ruim" (teste malsucedido).

A Figura 38 mostra a capacidade de automatização de diferentes tipos de teste de software. O termo "teste de propriedade" (ainda não visto nesta disciplina) é usado para se referir aos testes que avaliam os requisitos não funcionais do software, tais como os testes de desempenho (carga, estresse e estabilidade) e segurança.

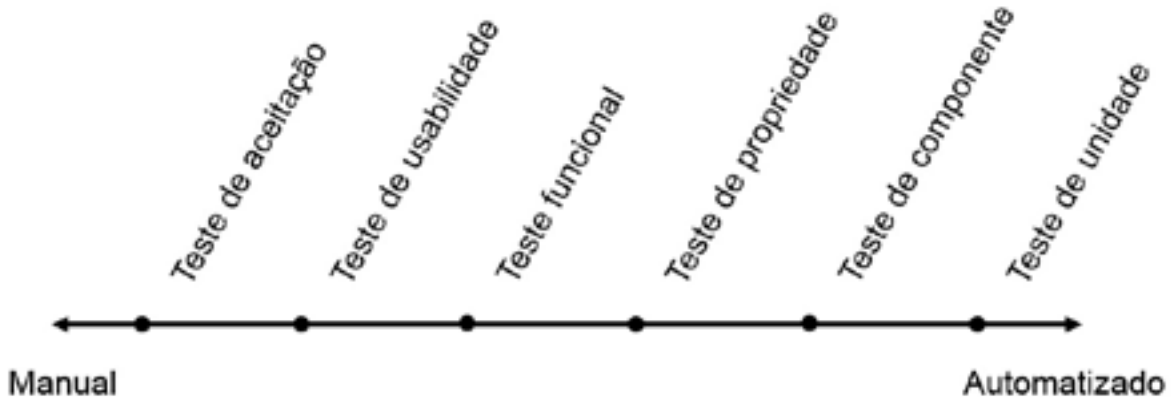


Figura 38: Capacidade de automatização de diferentes tipos de teste de software.
Fonte: Adaptado pelo autor a partir de Poppendieck e Poppendieck (2011).

De acordo com as classificações de Poppendieck e Poppendieck (2011), os tipos de testes com maior capacidade de automatização são aqueles que testam o software sob a perspectiva da tecnologia e envolvem a verificação de aspectos de programação. Por sua vez, os testes com menor capacidade de automatização são aqueles que testam o software sob a perspectiva do negócio e envolvem a avaliação do software como um produto.

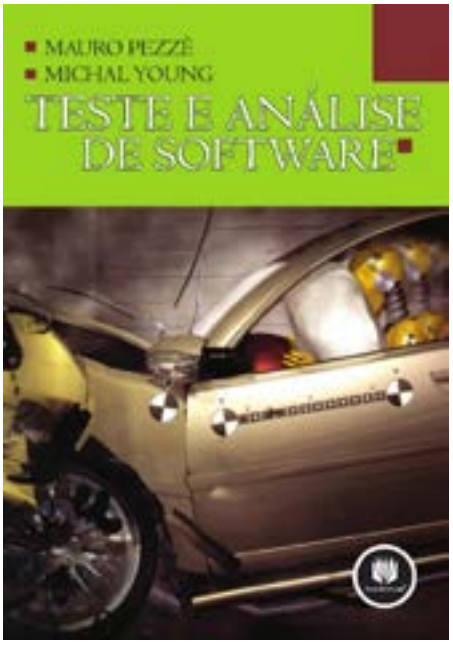
QUIZ

Descreva os testes relacionados no quadro a seguir com base nos estudos realizados na unidade 4 deste caderno de estudos.

Teste	Descrição
Teste de unidade	
Teste de integração	
Teste de sistema	
Teste de aceitação	
Teste de regressão	
Teste de caixa-branca	
Teste de caixa-preta	
Teste de desempenho	
Teste de usabilidade	

Teste de segurança	
Teste de funcionalidade	

SUGESTÃO DE LIVRO



PEZZÈ, Mauro. YOUNG, Michal. **Teste e análise de software:** processos, princípios e técnicas. Porto Alegre: Bookman, 2008.

Um software de qualidade não pode ser construído sem a utilização de técnicas de teste e análise de software. O livro "Teste e análise de software: processos, princípios e técnicas", de Mauro Pezzè e Michal Young, apresenta um conjunto de técnicas de teste e análise em uma prática moderna. Escrito em linguagem acessível, abrange tópicos bem aprofundados e oferece uma visão geral sobre o assunto.

SUGESTÃO DE VÍDEO

That Time the World Almost Ended (2018)

O vídeo "That Time the World Almost Ended", disponível no canal The Science Elf do YouTube, aborda a história bug do milênio (também conhecido como bug Y2K), um problema previsto para ocorrer em sistemas informatizados na passagem do ano de 1999 para o ano de 2000. O vídeo desperta a reflexão sobre a importância de testar sistemas com diferentes combinações de entradas e está disponível com legendas em inglês e legendas com tradução automática para o português.

CONSIDERAÇÕES FINAIS

Nesta unidade, você viu que o teste de software desempenha um importante papel na garantia da qualidade de software ou SQA (Software Quality Assurance). O teste de software é uma atividade essencial para qualquer projeto de desenvolvimento de software, pois é ele que permite demonstrar se o trabalho de desenvolvimento está sendo feito corretamente.

Na Seção 4.4, vimos que existem diferentes níveis, técnicas e tipos de teste que podem ser realizados ao longo do processo de desenvolvimento de um software. No entanto, não é difícil encontrarmos cenários de desenvolvimento de software em que o arcabouço de conhecimento fornecido pela disciplina de teste de software é pouco explorado; dessa forma, o processo de teste acaba se resumindo a realização de testes de unidade durante a codificação.

O ideal é que um grupo de testadores seja responsável por planejar, executar e relatar as atividades de teste, bem como manter os gestores e desenvolvedores do projeto informados sobre o que precisa ser corrigido ou melhorado para que o produto de software atinja seus objetivos. Mas, infelizmente, isso nem sempre é possível, pois uma abordagem ampla de testes envolve a dedicação de pessoas e pode implicar em custos extremamente significativos para o projeto.

Uma alternativa comumente adotada é a de deixar um analista responsável pelos testes e selecionar um conjunto de testes capaz de garantir a qualidade das propriedades do software (segurança, desempenho, usabilidade, etc.) mais importantes para o cliente e usuários finais.

EXERCÍCIO FINAL

1. A garantia da qualidade de software engloba uma ampla gama de atividades, tais como revisões, auditorias e testes. Esse conjunto de atividades que cuida da garantia da qualidade de software ou SQA (Software Quality Assurance) é denominado Verificação e Validação (V&V). O conjunto de atividades V&V, como um todo, envolve tanto a garantia de que o software está sendo desenvolvido da maneira correta como a garantia de que o software em desenvolvimento é o software certo.

Em relação às atividades de Verificação e Validação (V&V), avalie as afirmações a seguir:

I. A verificação busca demonstrar que a implementação está consistente com a especificação de requisitos;
II. A validação busca demonstrar que o software faz o que o cliente espera que ele faça;
III. Verificação e validação não são coisas distintas.

É correto o que se afirma em:
A. II, apenas.
B. III, apenas.
C. I e II, apenas.
D. I e III, apenas.
E. I, II e III.

2. A definição de um teste de software passa pela decisão sobre qual aspecto do software será testado (o que testar), qual técnica de teste será usada (como testar) e em que nível ou fase do desenvolvimento o teste será realizado (quando testar). Na dimensão "como testar", existe uma técnica de teste que busca verificar se todos os possíveis caminhos (chamadas de funções, laços de repetição, decisões lógicas, cálculos e demais tipos de processamento) que podem ser realizados pelo software estão em perfeito funcionamento.

A técnica de teste à qual o texto se refere é chamada de:
A. Teste de unidade.
B. Teste de caixa-preta.
C. Teste funcional.
D. Teste de caixa-branca.
E. Teste de desempenho.

3. O documento de caso de teste é um importante artefato para a realização de testes de software. É ele que fornece as informações e instruções para que o testador possa executar um teste de software, incluindo os procedimentos a serem realizados, os dados a serem fornecidos, as saídas esperadas e outras informações necessárias para a realização do teste.

Em relação às informações e instruções que compõem um típico documento de caso de teste, avalie as afirmações a seguir:

I. Uma pré-condição é uma declaração de um estado necessário para que o teste possa ser realizado;
II. As entradas são os valores (válidos e inválidos) que devem ser informados durante a execução do teste;
III. Os procedimentos descrevem as ações a serem realizadas pelo testador.

É correto o que se afirma em:

A. II, apenas.
B. III, apenas.
C. I e II, apenas.
D. I e III, apenas.
E. I, II e III.

REFERÊNCIAS

BRAGA, Pedro Henrique Cacique (Org.). **Teste de software**. São Paulo: Pearson Education do Brasil, 2016.

FERREIRA, Aurélio Buarque de Holanda. **Miniaurélio Século XXI**: O minidicionário da língua portuguesa. 5. ed. Rio de Janeiro: Nova Fronteira, 2001.

GONÇALVES, Priscila de Fátima; et al. **Testes de software e gerência de configuração**. [Recurso eletrônico]. Porto Alegre: SAGAH, 2019.

PAULA FILHO, Wilson de Pádua. **Engenharia de software**: fundamentos, métodos e padrões. [recurso eletrônico]. 3.ed. Rio de Janeiro: LTC, 2009. ISBN 978-85-216-1650-4.

PEZZÈ, Mauro. YOUNG, Michal. **Teste e análise de software**: processos, princípios e técnicas. Porto Alegre: Bookman, 2008.

POPPENDIECK, Mary; POPPENDIECK, Tom. **Implementando o desenvolvimento lean de software**: do conceito ao dinheiro. [recurso eletrônico]. Porto Alegre: Bookman, 2011.

PRESSMAN, Roger S; MAXIM, Bruce R. **Engenharia de Software**: uma abordagem profissional. 8. ed. Porto Alegre: AMGH, 2016. ISBN 978-0-07-802212-8.

SCHACH, Stephen R. **Engenharia de software**: os paradigmas clássico e orientado a objetos. [recurso eletrônico]. 7. ed. Porto Alegre: AMGH, 2010. ISBN 978-85-63308-44-3.

SOMMERVILLE, Ian. **Engenharia de software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.

uniavan.edu.br