



ENGENHARIA DE SOFTWARE

Professora Me. Márcia Cristina Dadalto Pascutti
Professora Esp. Janaina Aparecida de Freitas
Professora Esp. Talita Tonsic Gasparotti
Professor Esp. Victor de Marqui Pedroso

Acesse o seu livro também disponível na versão digital.

Quando identificar o ícone QR-CODE, utilize o aplicativo **Unicesumar Experience** para ter acesso aos conteúdos online. O download do aplicativo está disponível nas plataformas:



Google Play



App Store

UNICESUMAR

Av. Guedner, 1610 - Jardim Aclimação
Cep 87050-900 - MARINGÁ - PARANÁ
unicesumar.edu.br
44 3027.6360

UNICESUMAR EDUCAÇÃO A DISTÂNCIA

NEAD - Núcleo de Educação a Distância
Bloco 4 - MARINGÁ - PARANÁ
unicesumar.edu.br
0800 600 6360

as imagens utilizadas neste
livro foram obtidas a partir
do site SHUTTERSTOCK.COM

FICHA CATALOGRÁFICA

C397 **CENTRO UNIVERSITÁRIO DE MARINGÁ.** Núcleo de Educação a Distância; **PASCUTTI**, Márcia Cristina Dadalto; **FREITAS**, Janaina Aparecida de; **GASPAROTTI**, Talita Tonsic; **PEDROSO**, Victor de Marqui.

Engenharia de Software. Márcia Cristina Dadalto Pascutti; Janaina Aparecida de Freitas; Talita Tonsic Gasparotti; Victor de Marqui Pedroso.

Maringá-Pr.: Unicesumar, 2019.

232 p.

"Graduação - EaD".

1. Engenharia. 2. Software. 3. EaD. I. Título.

ISBN 978-85-459-1969-8

CDD - 22 ed. 005.1

CIP - NBR 12899 - AACR/2

Reitor

Wilson de Matos Silva

Vice-Reitor

Wilson de Matos Silva Filho

Pró-Reitor Executivo de EAD

William Victor Kendrick de Matos Silva

Pró-Reitor de Ensino de EAD

Janes Fidélis Tomelin

Presidente da Mantenedora

Cláudio Ferdinandi

NEAD - Núcleo de Educação a Distância**Diretoria Executiva**

Chrystiano Mincoff

James Prestes

Tiago Stachon

Diretoria de Graduação e Pós-graduação

Kátia Coelho

Diretoria de Permanência

Leonardo Spaine

Diretoria de Design Educacional

Débora Leite

Head de Produção de Conteúdos

Celso Luiz Braga de Souza Filho

Head de Curadoria e Inovação

Tania Cristiane Yoshie Fukushima

Gerência de Produção de Conteúdo

Diogo Ribeiro Garcia

Gerência de Projetos Especiais

Daniel Fuverki Hey

Gerência de Processos Acadêmicos

Taessa Penha Shiraishi Vieira

Gerência de Curadoria

Carolina Abdalla Normann de Freitas

Supervisão de Produção de Conteúdo

Nádila Toledo

Coordenador de Conteúdo

Danillo Xavier Saes

Designer Educacional

Rossana Costa Giani

Projeto Gráfico

Jaime de Marchi Junior

José Jhanny Coelho

Arte Capa

Arthur Cantareli Silva

Ilustração Capa

Bruno Pardinho

Editoração

Sabrina Maria Pereira de Novae

Qualidade Textual

Cindy Mayumi Okamoto Luca

Ilustração

Rodrigo Barbosa



Professor
Wilson de Matos Silva
Reitor

Em um mundo global e dinâmico, nós trabalhamos com princípios éticos e profissionalismo, não só para oferecer uma educação de qualidade, mas, acima de tudo, para gerar uma conversão integral das pessoas ao conhecimento. Baseamo-nos em 4 pilares: intelectual, profissional, emocional e espiritual.

Iniciamos a Unicesumar em 1990, com dois cursos de graduação e 180 alunos. Hoje, temos mais de 100 mil estudantes espalhados em todo o Brasil: nos quatro campi presenciais (Maringá, Curitiba, Ponta Grossa e Londrina) e em mais de 300 polos EAD no país, com dezenas de cursos de graduação e pós-graduação. Produzimos e revisamos 500 livros e distribuímos mais de 500 mil exemplares por ano. Somos reconhecidos pelo MEC como uma instituição de excelência, com IGC 4 em 7 anos consecutivos. Estamos entre os 10 maiores grupos educacionais do Brasil.

A rapidez do mundo moderno exige dos educadores soluções inteligentes para as necessidades de todos. Para continuar relevante, a instituição de educação precisa ter pelo menos três virtudes: inovação, coragem e compromisso com a qualidade. Por isso, desenvolvemos, para os cursos de Engenharia, metodologias ativas, as quais visam reunir o melhor do ensino presencial e a distância.

Tudo isso para honrarmos a nossa missão que é promover a educação de qualidade nas diferentes áreas do conhecimento, formando profissionais cidadãos que contribuam para o desenvolvimento de uma sociedade justa e solidária.

Vamos juntos!



Janes Fidélis Tomelin

Pró-Reitor de Ensino de EaD

Kátia Solange Coelho

Diretoria de Graduação e Pós

Débora do Nascimento Leite

Diretoria de Design Educacional

Leonardo Spaine

Diretoria de Permanência

Seja bem-vindo(a), caro(a) acadêmico(a)! Você está iniciando um processo de transformação, pois quando investimos em nossa formação, seja ela pessoal ou profissional, nos transformamos e, consequentemente, transformamos também a sociedade na qual estamos inseridos. De que forma o fazemos? Criando oportunidades e/ou estabelecendo mudanças capazes de alcançar um nível de desenvolvimento compatível com os desafios que surgem no mundo contemporâneo.

O Centro Universitário Cesumar mediante o Núcleo de Educação a Distância, o(a) acompanhará durante todo este processo, pois conforme Freire (1996): "Os homens se educam juntos, na transformação do mundo".

Os materiais produzidos oferecem linguagem dialógica e encontram-se integrados à proposta pedagógica, contribuindo no processo educacional, complementando sua formação profissional, desenvolvendo competências e habilidades, e aplicando conceitos teóricos em situação de realidade, de maneira a inseri-lo no mercado de trabalho. Ou seja, estes materiais têm como principal objetivo "provocar uma aproximação entre você e o conteúdo", desta forma possibilita o desenvolvimento da autonomia em busca dos conhecimentos necessários para a sua formação pessoal e profissional.

Portanto, nossa distância nesse processo de crescimento e construção do conhecimento deve ser apenas geográfica. Utilize os diversos recursos pedagógicos que o Centro Universitário Cesumar lhe possibilita. Ou seja, acesse regularmente o Studeo, que é o seu Ambiente Virtual de Aprendizagem, interaja nos fóruns e enquetes, assista às aulas ao vivo e participe das discussões. Além disso, lembre-se que existe uma equipe de professores e tutores que se encontra disponível para sanar suas dúvidas e auxiliá-lo(a) em seu processo de aprendizagem, possibilitando-lhe trilhar com tranquilidade e segurança sua trajetória acadêmica.

Professora Me. Márcia Cristina Dadalto Pascutti

Possui graduação em Processamento de Dados pela Universidade Estadual de Maringá (1989) e mestrado em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (2002). Atualmente, é professora no Instituto Federal do Paraná – Campus Umuarama – e está cursando seu doutorado na PUC-PR. Atua principalmente nos seguintes temas: arquitetura de computadores, engenharia de software, processamento de imagens, reconhecimento de padrões, visão computacional.

Link: <http://lattes.cnpq.br/0297217008123332>

Professora Esp. Janaina Aparecida de Freitas

Possui graduação em Informática pela Universidade Estadual de Maringá (2010) e especialização em MBA em Teste de Software pela UNICEUMA (2012). Atualmente, cursa o Programa de Mestrado em Ciência da Computação na Universidade Estadual de Maringá (UEM) e é graduanda de Letras - Português/Inglês na Unicesumar. Atua como professora mediadora, professora conteudista em gravação de aulas ao vivo e gravação de aulas conceituais nos cursos do NEAD – Núcleo de Educação a Distância – da Unicesumar, para os cursos de graduação de Sistemas para Internet, Análise e Desenvolvimento de Sistemas, Gestão da Tecnologia da Informação e Engenharia de Software, nas disciplinas de Engenharia de Software, Design Gráfico, Tópicos Especiais, Gerenciamento de Software, Design de Interação Humano-Computador, Projeto Implementação e Teste de Software, há três anos. Além disso, tem experiência em iniciativa privada na área de análise de sistemas e testes de software.

Link: <http://lattes.cnpq.br/4906244382612830>

Professora Esp. Talita Tonsic Gasparotti

Graduada em Tecnologia em Processamento de Dado pela Unicesumar (2006) e especialista em Gestão e Coordenação Escolar pela Faculdade Eficaz (2014). Atualmente, cursa especialização em Educação a Distância e Tecnologias Educacionais na Unicesumar.

Link: <http://lattes.cnpq.br/5030590784964019>

Professora Esp. Victor de Marqui Pedroso

Possui pós-graduação em Banco de Dados Oracle e DB2 pelo Centro Universitário de Maringá (2009) e graduação em Tecnologia em Processamento de Dados pelo Centro Universitário de Maringá (2003). Tem experiência como analista de sistemas, documentador, homologador, programador de software, bem como em desenvolvimento utilizando a ferramenta Delphi. Trabalhou como professor mediador e, atualmente, é professor formador dos cursos de Análise e Desenvolvimento de Sistemas e Sistemas para Internet, ministrando as disciplinas de Banco de Dados e Design de Interação.

Link: <http://lattes.cnpq.br/8611697826182780>

ENGENHARIA DE SOFTWARE

SEJA BEM-VINDO(A)!

Prezado(a) acadêmico(a), é com muito prazer que lhe apresentamos o livro de engenharia de software. A engenharia de software possui uma gama imensa de tópicos, os quais não seria possível abranger em cinco unidades, assim como este livro está organizado. Então, abordaremos os assuntos iniciais, essenciais para todo o contexto da disciplina. Se gostar da matéria e quiser se aprofundar, você poderá consultar os livros citados durante as unidades. Neles, com certeza, você aprenderá muitos outros assuntos e poderá ter a certeza de que essa disciplina realmente é muito importante quando tratamos de software como um todo.

Este livro está organizado em cinco unidades e todas estão estreitamente relacionadas. Na Unidade I, apresentaremos alguns conceitos referentes à disciplina. Você notará, durante a leitura das outras unidades, que esses conceitos são utilizados com frequência.

A engenharia de software surgiu mediante a necessidade de tornar o desenvolvimento de software confiável, com etapas bem definidas e custo e cronograma previsíveis, o que até 1968, quando o termo engenharia de software foi proposto, não acontecia. Além disso, gostaríamos de ressaltar que software compreende, além dos programas, toda a documentação referente a ele, e a engenharia de software é a disciplina que trata dessa documentação.

Já a Unidade II é bastante específica e trata apenas dos conceitos relacionados aos requisitos de software, já que, para o desenvolvimento de um software de forma esperada pelo cliente, é fundamental que todos os requisitos tenham sido claramente definidos. Ainda, nessa unidade, para que você possa ter uma ideia de como é um documento de requisitos, mostraremos o de uma locadora de filmes. O exemplo é bem simples, mas contém detalhes suficientes para a continuidade do processo de desenvolvimento de software.

Então, de posse do documento de requisitos, começaremos a estudar, na Unidade III de nosso livro, a respeito da modelagem do sistema. Nessa unidade, utilizaremos os conceitos de orientação a objetos e de UML. A modelagem é a parte fundamental de todas as atividades relacionadas ao processo de software, dado que os modelos que são construídos nessa etapa servem para comunicar a estrutura e o comportamento desejados do sistema, a fim de que possamos melhor compreender o sistema que está sendo elaborado.

Na Unidade IV, estudaremos alguns diagramas, a saber: diagrama de casos de uso; diagrama de classes; diagrama de sequência; diagrama de estados; e diagrama de atividades. Para auxiliá-lo no entendimento de cada um deles, preparamos uma espécie de tutorial, o qual explica a sua elaboração passo a passo, uma vez que esses diagramas são os mais importantes e os mais utilizados da UML, servindo de base para os demais.

Finalmente, chegamos à última unidade do nosso material. Essa unidade é o fechamento das etapas do processo de software, visto que tratará das etapas de projeto, implementação, teste e manutenção de software. Assim, você compreenderá todo o processo de software, bem como as etapas que o englobam.

APRESENTAÇÃO

À medida que o sistema vai sendo desenvolvido, cada programa vai sendo validado pelo desenvolvedor, mas isso só não basta. É muito importante que a etapa de validação seja cuidadosamente realizada pela equipe de desenvolvimento, pois é preciso assegurar que o sistema funcionará corretamente.

Depois que o sistema é colocado em funcionamento, ele precisa evoluir para continuar atendendo as necessidades dos usuários. Em todas estas etapas, é importante a aplicação das técnicas da engenharia de software.

Esperamos, dessa forma, que sua leitura seja agradável e que esse conteúdo possa contribuir para seu crescimento pessoal, acadêmico e profissional.

Prof.^a Márcia.

Prof.^a Janaína.

Prof.^a Talita.

Prof. Victor.

SUMÁRIO

UNIDADE I

INTRODUÇÃO A ENGENHARIA DE SOFTWARE

15	Introdução
16	Conceitos Básicos e Aplicações de Software
18	História da Engenharia de Software
20	Tipos de Aplicações de Software
26	Processos de Software
40	Metodologias Ágeis
52	Considerações Finais
58	Referências
60	Gabarito

UNIDADE II

REQUISITOS DE SOFTWARE

63	Introdução
64	Requisitos de Software
71	Documento de Requisitos
77	Engenharia de Requisitos
88	Considerações Finais
93	Referências
94	Gabarito



SUMÁRIO

UNIDADE III

MODELAGEM DE SISTEMAS

97	Introdução
98	Introdução à UML
101	Ferramentas CASE
104	Modelagem de Sistemas
107	Diagrama de Casos de Uso
122	Conceitos Básicos de Orientação a Objetos
128	Considerações Finais
134	Referências
135	Gabarito

UNIDADE IV

DIAGRAMA DE SISTEMAS

139	Introdução
140	Diagrama de Classes
153	Diagrama de Sequência
161	Diagrama de Máquina de Estados
165	Diagrama de Atividades
171	Considerações Finais
178	Referências
179	Gabarito



SUMÁRIO

UNIDADE V

GERENCIAMENTO DE SOFTWARE

183 Introdução

184 Introdução à Qualidade de Software

197 Teste de Software

205 Evolução de Software

210 Configuração de Software

216 Considerações Finais

222 Referências

224 Gabarito

225 CONCLUSÃO



Prof. Me. Márcia Cristina Dadalto Pascutti

Prof. Esp. Talita Tonsic Gasparotti

Prof. Esp. Janaina Aparecida de Freitas

INTRODUÇÃO A ENGENHARIA DE SOFTWARE

UNIDADE

I

Objetivos de Aprendizagem

- Entender o que é Engenharia de Software e qual é a sua importância.
- Conhecer a história da Engenharia de Software.
- Conhecer os tipos de aplicações de software.
- Compreender os processos de software e seus modelos.
- Apresentar os conceitos básicos sobre as metodologias ágeis.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Conceitos básicos e aplicações de software
- História da Engenharia de Software
- Tipos de aplicações de software
- Processos de software e modelos de processos de software
- Metodologias ágeis

INTRODUÇÃO

Caro(a) aluno(a), nesta primeira unidade, trataremos sobre alguns conceitos relacionados à Engenharia de Software como um todo e que serão fundamentais para o entendimento de todas as unidades.

O termo engenharia de software foi proposto, inicialmente, em 1968, em uma conferência organizada para discutir o que era chamado de crise de software. Essa crise foi originada em função do hardware, que, nessa época, teve seu poder de processamento e memória aumentados, o que gerou a necessidade de que o software acompanhasse esse avanço. Assim, o que se notou, na época, é que o desenvolvimento de grandes sistemas de maneira informal, sem seguir regras ou etapas pré-definidas, não era suficiente.

O software desenvolvido até então não era confiável e não era elaborado dentro do tempo e custos previstos inicialmente. Além disso, seu desempenho era insatisfatório e era difícil de se realizar a sua manutenção. Os custos em relação ao hardware estavam caindo, já que a sua produção passou a ser em série, porém o custo do software não acompanhava essa queda e, muitas vezes, era aumentado.

A ideia inicial da engenharia de software era a de tornar o desenvolvimento de software em um processo sistematizado, no qual seriam aplicadas técnicas e métodos necessários para controlar a complexidade inerente aos grandes sistemas.

Assim, neste livro, abordaremos alguns tópicos acerca da engenharia. Com certeza, precisaríamos de dezenas de livros como esse para esgotar esse assunto, portanto, aqui, estudaremos somente uma pequena parte desse conteúdo que é tão abrangente.

Vamos, então, aos conceitos. Boa leitura!



CONCEITOS BÁSICOS E APLICAÇÕES DE SOFTWARE

Caro(a) aluno(a), o software é um segmento de instruções que serão analisadas e processadas pelo computador: ele é quem dará significado a elas, com o objetivo de executar tarefas específicas. Além disso, são os softwares que comandam o funcionamento de qualquer computador, uma vez que são a parte lógica que fornece explicações para o hardware do computador.

Sobre a temática, Pressman (2011, p. 30) sustenta que o “software de computadores continua a ser uma tecnologia única e mais importante no cenário mundial. É também um ótimo exemplo da lei das consequências não intencionais”.

Contudo, podemos observar, também, uma outra percepção do que é o software, de acordo com Pressman e Maxim (2016, p. 1):

Software de computador é o produto que profissionais de software desenvolvem e ao qual dão suporte no longo prazo. Abrange programas executáveis em um computador de qualquer porte ou arquitetura, conteúdos (apresentados à medida que os programas são executados), informações descritivas tanto na forma impressa (*hard copy*) como na virtual.

Além disso, segundo Sommerville (2011), o software é composto não somente pelos programas, mas também pela documentação associada a esses programas.

Para Pressman e Maxim (2016), o software possui, pelo menos, três características que o diferenciam do hardware, a saber:

1. O software é desenvolvido ou passa por um processo de engenharia, não sendo fabricado no sentido clássico.

Apesar de existir semelhanças entre o desenvolvimento de software e a fabricação de hardware, essas atividades são muito diferentes. Os custos de software se concentram no processo de engenharia e, por consequência disso, os projetos de software não podem ser conduzidos como se fossem projetos de fabricação.

2. Software não se desgasta.

Normalmente, o hardware apresenta taxas de defeitos mais altas no início de sua vida, mas que são corrigidas ao longo de seu uso. No entanto, com o uso, o hardware pode se desgastar devido à poeira, má utilização, temperaturas extremas e outros fatores. Já com o software é diferente, dado que ele não está sujeito aos problemas ambientais, assim como o hardware está. Em relação aos problemas iniciais, com o software também acontece, pois alguns defeitos ainda podem passar pela etapa de validação.

Quando um componente do hardware se desgasta, normalmente, ele é trocado por um novo e o hardware volta a funcionar. Entretanto, com o software isso não acontece: não há como, simplesmente, trocar o componente, pois quando um erro acontece no software, pode ser de projeto ou de requisito mal definido, o que leva o software a sofrer manutenção, a qual pode ser considerada mais complexa do que a do hardware.

3. Embora a indústria caminhe para a construção com base em componentes, a maioria dos softwares continua a ser construída de forma personalizada (sob encomenda).

Os componentes reutilizáveis de software são criados para que o desenvolvedor possa se concentrar nas partes do projeto que representam algo novo. Esses componentes encapsulam dados e o processamento aplicado a eles, possibilitando a criação de novas aplicações a partir de partes reutilizáveis.



SAIBA MAIS

Software é tanto um produto quanto um veículo que distribui um produto.

Fonte: Pressman e Maxim (2016, p. 3).



Reprodução proibida. Art. 184 do Código Penal e Lei 9.510 de 19 de fevereiro de 1998.

HISTÓRIA DA ENGENHARIA DE SOFTWARE

A engenharia do software se define como “o estabelecimento e o emprego de sólidos princípios de engenharia de modo a obter software de maneira econômica, que seja confiável e funcione de forma eficiente em máquinas reais” (BAUER, 1969 apud PRESSMAN, 2011, p. 39). Dessa forma, a principal característica da engenharia do software são métodos e técnicas que são utilizadas para o desenvolvimento do software.

Segundo Tsui e Karam (2013), o termo engenharia de software foi mencionado pela primeira vez em uma conferência da Organização do Tratado do Atlântico Norte (OTAN), conduzida na Alemanha em 1968. Seu objetivo era o de aplicar técnicas e utilizar ferramentas na área da computação, para o desenvolvimento e a produção de software. Portanto, para que haja essa produção, é preciso planejar, a curto e a longo prazo, a equipe que será escalada, bem como a qualidade do produto e do seu processo. Tais ações são presentes em nosso dia a dia e, dessa forma, as pessoas não percebem, mas estamos ligados a engenharia de software em nosso cotidiano.

Além disso, de acordo com Sommerville (2011), a engenharia de software é uma disciplina cujo foco está no desenvolvimento de sistemas de software de alta qualidade por um custo acessível. Não só, mas é preciso ter em mente que “o software é abstrato e intangível, e que está engenharia preocupa-se com todos os aspectos da produção de software, desde os estágios iniciais de especificação de um sistema até a manutenção do sistema após ter sido posto em uso” (TSUI; KARAM, 2013, p. 35).

Essa disciplina ou área de conhecimento se formou em decorrência da preocupação em contornar a crise que estava se abatendo no software e lhe atribuir um tratamento de engenharia. Assim, a engenharia do software surgiu em 1970 e seu objetivo era o de implementar sistemas mais complexos.

Segundo Sommerville (2011, p. 4), “quando falamos de engenharia de software, não se trata apenas do programa em si, mas de toda a documentação associada e dados de configurações necessários para fazer este programa operar corretamente”. Além disso:

Para ser considerada como prática de engenharia profissional deve incluir o código e os regulamentos que seus membros devem seguir. Portanto, a engenharia do software também inclui diretivas para a aquisição de conhecimento por parte de seus membros e diretivas para as práticas comportamentais de seus membros (TSUI; KARAM, 2013, p. 36).

Assim, as pessoas que trabalham diretamente com a engenharia do software estão envolvidas e dirigem seus conhecimentos para o desenvolvimento do software, atentas a manutenção e adequação, bem como aos seus diferentes processos produtivos.

A focalização da engenharia de software se dá em todos os âmbitos da produção de um software, abrangendo “desde os estágios iniciais de especificação do sistema até a sua manutenção” (SOMMERVILLE, 2011, p. 5). Além disso, de acordo com Sommerville (2011), ela é importante por dois motivos:

1. A sociedade, cada vez mais, depende de sistemas de software avançados. Portanto, é preciso que os engenheiros de software sejam capazes de produzir sistemas confiáveis, de maneira econômica e rápida.
2. A longo prazo, normalmente, é mais barato usar métodos e técnicas propostos pela engenharia de software, ao invés de somente escrever os programas como se fossem algum projeto pessoal. Para a maioria dos sistemas, o maior custo está em sua manutenção, ou seja, nas alterações que são realizadas depois que o sistema é implantado.



TIPOS DE APLICAÇÕES DE SOFTWARE

Atualmente, pelo fato de que o software é utilizado em praticamente todas as atividades exercidas pelas pessoas, ele é dividido em sete grandes categorias, a saber:

- **Software de sistema:** de acordo com Pressman e Maxim (2016), são os programas desenvolvidos para atender a outros. São exemplos: editores

de texto, compiladores, utilitários para gerenciamento de arquivos, sistemas operacionais e softwares de rede.

- **Software de aplicação:** são programas independentes, desenvolvidos para solucionar uma necessidade específica de negócio, ao processarem dados comerciais ou técnicos, de forma que ajude nas operações comerciais ou nas tomadas de decisão pelos administradores da empresa. Um exemplo é o software ERP (*Enterprise Resource Planning*), que visa a integração de todos os dados e processos de uma organização em um sistema único.
- **Software de engenharia/científico:** são aplicações com uma extensa variedade de programas, os quais vão da astronomia à vulcanologia, da biologia molecular à fabricação automatizada. Normalmente, utilizam algoritmos para o processamento numérico pesado.
- **Software embarcado:** são residentes em um produto ou sistema e são utilizados para controlar ou gerenciar características ou funções para o próprio sistema ou usuário. Como exemplo, temos: aparelhos celulares, painel do micro-ondas, controle do sistema de freios de um veículo etc.
- **Software para linha de produtos:** conjunto, grupo de softwares ou, ainda, uma família de sistemas que compartilham características ou especificações comuns que satisfazem as estratégias de mercado, um domínio de aplicação ou missão. Em outras palavras, são softwares similares, ao invés de um único software individual (CÂMARA, 2011).
- **Aplicações Web/aplicativos móveis:** categoria de softwares que contempla uma variedade de aplicações voltadas para navegadores e softwares residentes em dispositivos móveis.
- **Software de inteligência artificial:** utilizam algoritmos não numéricos para solucionar problemas complexos que não poderiam ser solucionados pela computação ou análise direta. São exemplos: sistemas especialistas, robótica, redes neurais artificiais etc.

SOFTWARE LEGADO

Segundo Pressman e Maxim (2016), os softwares legados são mais antigos e têm uma contínua atenção por parte dos engenheiros de software desde os anos de 1960. Foram desenvolvidos há décadas atrás, mas ainda continuam sendo usados enquanto suporte para as funções de negócio vitais e consideradas indispensáveis.

Esses softwares podem apresentar projetos não expansíveis, código intrincado, documentação inexistente ou escassa e testes que nunca foram arquivados. Surge-nos, então, a pergunta: o que devemos fazer? Nada. Isso mesmo: nada, até que o software legado se submeta a modificações significativas.

Vale lembrarmos que, caso o software legado atenda às necessidades de seus usuários, possibilitando confiança e vitalidade, ele não precisa ser “consertado”. Entretanto, esses sistemas evoluem com o passar do tempo, por uma ou mais das seguintes razões:

- O software deve ser adaptado para atender às necessidades de novos ambientes ou de novas tecnologias computacionais.
- O software deve ser aperfeiçoado para implementar novos requisitos de negócio.
- O software deve ser expandido para torná-lo capaz de funcionar com outros bancos de dados ou com sistemas mais modernos.
- O software deve ser rearquitetado para torná-lo viável dentro de um ambiente computacional em evolução (PRESSMAN; MAXIM, 2016, p. 8).



O objetivo da engenharia de software moderna é o de “elaborar metodologias baseadas na evolução” (PRESSMAN; MAXIM, 2016, p. 8). Em outras palavras, os softwares modificam-se continuamente e novos softwares são construídos a partir dos antigos.

A NATUREZA MUTANTE DO SOFTWARE

Quatro categorias de software mais amplas dominam a evolução do setor, são elas:

WebApps

Os sites, por volta de 1990 e 1995, eram formados por um conjunto de arquivos hipertexto linkados, que exibiam texto e gráficos limitados. Contudo, com o crescimento da linguagem HTML, foi possível oferecer uma alta capacidade computacional e, com isso, nasciam os sistemas e aplicações baseadas na Web. Pressman e Maxim (2016) referem-se coletivamente a eles como WebApps.

Aplicativos Móveis

Segundo Saes *et al.* (2018, p. 86), “os aplicativos móveis são softwares popularmente conhecidos como ‘Apps’, uma abreviação do termo ‘aplicação de software’”. Eles podem ser baixados em lojas on-line, tais como Google Play, App Store ou Windows Phone Store. Muitos dos aplicativos que estão disponíveis são gratuitos, mas outros são pagos. Possivelmente, você já deve ter baixado muitos apps para o seu iPhone, smartphone ou PDA. O WhatsApp, o Facebook e o Uber são exemplos de aplicativos que podem ser usados.

Os aplicativos móveis podem ser classificados em: (i) ferramentas de suporte para a produtividade, tais como editores, planilhas, gravadores e entre outros; e (ii) ferramentas de recuperação de informação, que inclui calendário, agenda telefônica, correio eletrônico, informações climáticas e entre outros (SAES *et al.*, 2018).



SAIBA MAIS

Qual a diferença entre uma WebApp e um aplicativo móvel?

É importante reconhecer que existe uma diferença sutil entre aplicações web móveis e aplicativos móveis. Uma aplicação web móvel (WebApp) permite que um dispositivo móvel tenha acesso a conteúdo baseado na web por meio de um navegador especificamente projetado para se adaptar aos pontos fortes e fracos da plataforma móvel. Um aplicativo móvel pode acessar diretamente as características do hardware do dispositivo (por exemplo, acelerômetro ou localização por GPS) e, então, fornecer os recursos de processamento e armazenamento local. Com o passar do tempo, essa diferença entre WebApps móveis e aplicativos móveis se tornará indistinta, à medida que os navegadores móveis se tornarem mais sofisticados e ganharem acesso ao hardware e as informações em nível de dispositivo.

Fonte: Pressman e Maxim (2016, p. 10).

Reprodução proibida. Art. 184 do Código Penal e Lei 9.610 de 19 de fevereiro de 1998.

Computação em Nuvem

A computação em nuvem é definida, por Pressman e Maxim (2016, p. 10), como uma “infraestrutura ou ecossistema que permite a qualquer usuário em qualquer lugar, utilizar um dispositivo de computação para compartilhar recursos computacionais em grande escala”. Na Figura 1, é mostrado como a arquitetura lógica da computação em nuvem é representada:

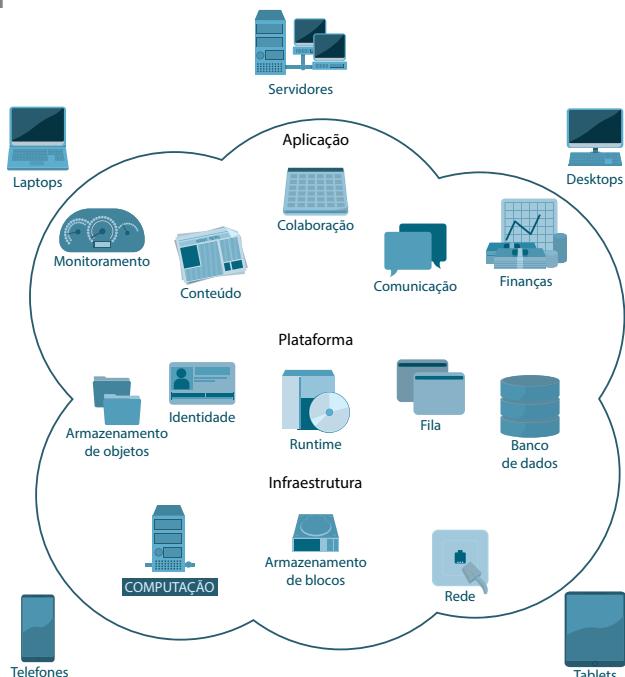


Figura 1 – Arquitetura lógica da computação em nuvem

Fonte: Pressman e Maxim (2016, p. 10).

Analisando a figura, temos os dispositivos de computação que estão fora da nuvem (servidores, desktops, tablets, telefones e laptops), os quais possuem acesso a uma grande diversidade de recursos dentro dela. Por sua vez, dentro da nuvem, temos os recursos que abrangem as aplicações, plataformas e infraestrutura. Assim, um dispositivo de computação externa acessa a nuvem por meio de um navegador Web; diante disso, a nuvem fornece o acesso aos dados armazenados nos bancos de dados ou em estruturas de dados, como fila.

Software para Linha de Produto

A Linha de Produtos de Software (*Software Product Line*) é um conjunto, grupo de softwares ou, ainda, uma família de sistemas que compartilham características ou especificações comuns, as quais satisfazem as estratégias de mercado, um domínio de aplicação ou missão, assim como podemos visualizar na Figura 2. Em outras palavras, são softwares similares, ao invés de serem um único software individual (CÂMARA, 2011):

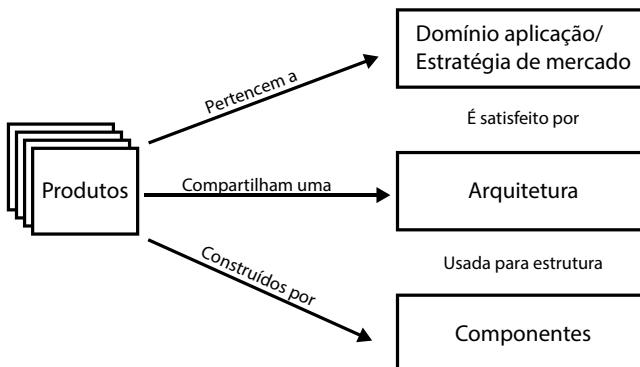
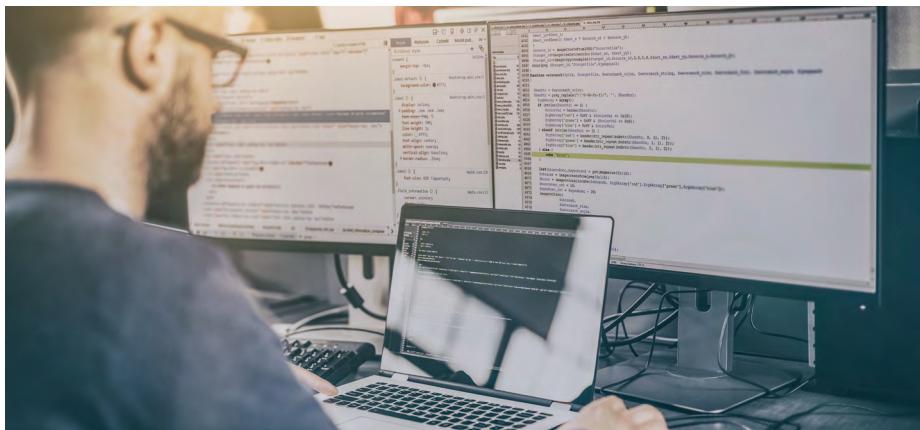


Figura 2 - Linha de Produtos de Software
Fonte: adaptada de Nunes (2013, p. 08).

De acordo com Fernandes e Teixeira (2011, p. 97), a “linha de produto é um novo emergente paradigma para o desenvolvimento de software”. A principal característica desses softwares é o fato serem desenvolvidos a partir de um conjunto de artefatos, reutilizando componentes, especificações de requisitos, casos de teste e outros, os quais podem ser utilizados para o desenvolvimento de novas aplicações que possuem um mesmo domínio (FERREIRA, 2012).



PROCESSOS DE SOFTWARE

Para que um software seja produzido, são necessárias diversas etapas, as quais são compostas por uma série de tarefas. Para esse conjunto de etapas, dá-se o nome de processo de software, o qual pode envolver o desenvolvimento de software a partir do zero, em uma determinada linguagem de programação (por exemplo, o Java ou C) ou, então, a ampliação e a modificação de sistemas já em utilização pelos usuários.

Segundo Sommerville (2011), existem muitos processos de software diferentes. No entanto, todos devem incluir quatro atividades fundamentais para a engenharia de software:

- 1. Especificação de software:** é necessário que o cliente defina as funcionalidades do software que será desenvolvido, bem como todas as suas restrições operacionais.
- 2. Projeto e implementação de software:** o software deve ser confeccionado mediante as especificações definidas anteriormente.
- 3. Validação de software:** o software precisa ser validado, a fim de garantir que ele faça o que o cliente deseja, ou seja, que atenda às especificações de funcionalidade.
- 4. Evolução de software:** as funcionalidades definidas pelo cliente durante o desenvolvimento do software podem mudar. Consequentemente, o software precisará evoluir para atender a essas mudanças.

Vamos estudar detalhadamente cada uma das atividades mencionadas durante a nossa disciplina. Utilizaremos, inclusive, ferramentas automatizadas (ferramentas CASE) para nos auxiliar na elaboração dos diversos documentos que serão necessários.

De acordo com Sommerville (2011, p. 19):

Os processos de software são complexos e, como todos os processos intelectuais e criativos, dependem de pessoas para tomar decisões e fazer julgamentos. Não existe um processo ideal, a maioria das organizações desenvolvem os próprios processos de desenvolvimento de software.

No entanto, o que acontece é que nem sempre as empresas aproveitam as boas técnicas da engenharia de software em seu desenvolvimento de software. Assim, normalmente, o software não atende aos requisitos do usuário e demora mais do que o previsto para ser desenvolvido, aumentando, assim, o valor do custo do software.

As atividades mencionadas por Sommerville (2011) podem ser organizadas pelas empresas, da forma como elas acharem melhor. Contudo, é importante ressaltar que todas essas atividades são de extrema importância e o processo de software adotado pela empresa não deve deixar de considerar nenhuma das etapas.

Além disso, é importante ressaltar que até mesmo as empresas que possuem um processo de software bem definido e documentado, para determinados softwares que elas desenvolvem, pode ser utilizado outro processo de software. Em outras palavras, dependendo do software a ser desenvolvido, pode ser utilizado um determinado processo de software. No próximo tópico, estudaremos alguns modelos de processo de software e constataremos que é possível que a empresa adote um processo de software próprio e que atenda suas necessidades.

MODELOS DE PROCESSOS DE SOFTWARE

Assim como já fora discutido, um processo de software é composto por um conjunto de etapas que são necessárias para que ele seja produzido. Sommerville (2011) afirma que um modelo de processo de software é uma representação abstrata, simplificada de um processo de software. Os modelos de processo incluem as atividades que fazem parte do processo de software (ou seja, as atividades descritas no item anterior), os artefatos de software que devem ser produzidos em

cada uma das atividades (documentos) e os papéis das pessoas envolvidas na engenharia de software. Além disso, cada modelo de processo representa um processo a partir de uma perspectiva particular, de uma maneira que proporciona apenas informações parciais sobre o processo.

Na literatura, existem diversos modelos de processo de software. Em nossa disciplina, trabalharemos somente três desses modelos e, em seguida, mostraremos as atividades básicas que estão presentes em, praticamente, todos os modelos de processos de software.

Os modelos de processo que abordaremos foram retirados de Sommerville (2011). São eles:

- 1. Modelo em Cascata (ciclo de vida clássico):** esse modelo considera as atividades fundamentais do processo de especificação, desenvolvimento, validação e evolução, e representa cada uma delas como fases separadas, como: especificação de requisitos, projeto de software, implementação, testes e assim por diante.
- 2. Modelo de Desenvolvimento Incremental:** esse modelo intercala as atividades de especificação, desenvolvimento e validação. É um sistema inicial rapidamente desenvolvido a partir de especificações abstratas, que são refinadas com informações do cliente, para produzir um sistema que satisfaça as suas necessidades, produzindo várias versões do software.
- 3. Engenharia de Software Orientada a Reuso:** esse modelo parte do princípio de que existem muitos componentes que podem ser reutilizáveis. O processo de desenvolvimento do sistema concentra-se em combinar vários desses componentes em um sistema, em vez de proceder ao desenvolvimento a partir do zero, com o objetivo de reduzir o tempo de desenvolvimento.



SAIBA MAIS

A finalidade dos modelos de processo é tentar reduzir o caos presente no desenvolvimento de novos produtos de software.

Fonte: Pressman e Maxim (2016, p. 41).

O MODELO EM CASCATA

O modelo em cascata ou ciclo de vida clássico, considerado o paradigma mais antigo da engenharia de software, sugere uma abordagem sequencial e sistemática para o desenvolvimento de software. Assim, inicia-se com a definição dos requisitos por parte do cliente e, depois, avança-se para as atividades de projeto e implementação de software, culminando no suporte contínuo do software concluído por meio de testes.

Segundo Sommerville (2011), os principais estágios do modelo em cascata demonstram as atividades fundamentais do desenvolvimento. São eles:

- 1. Análise e definição de requisitos:** as funções, as restrições e os objetivos do sistema são estabelecidos por meio da consulta aos usuários do sistema. Em seguida, são definidos em detalhes e servem como uma especificação do sistema.
- 2. Projeto de sistemas e de software:** o processo de projeto de sistemas agrupa os requisitos em sistemas de hardware ou de software. Ele estabelece uma arquitetura do sistema geral. O projeto de software envolve a identificação e a descrição das abstrações fundamentais do sistema de software e suas relações.
- 3. Implementação e teste de unidades:** durante esse estágio, o projeto de software é compreendido como um conjunto de programas ou unidades de programa. O teste de unidades envolve a verificação de que cada unidade atende a sua especificação.
- 4. Integração e teste de sistemas:** as unidades de programa ou programas individuais são integrados e testados como um sistema completo, a fim de se garantir que os requisitos de software foram atendidos. Depois dos testes, o sistema de software é entregue ao cliente.
- 5. Operação e manutenção:** normalmente (embora não necessariamente), essa é a fase mais longa do ciclo de vida. O sistema é instalado e colocado em operação. A manutenção envolve corrigir erros que não foram descobertos em estágios anteriores do ciclo de vida, melhorando a implementação das unidades de sistema e aumentando, consequentemente, as funções desse sistema à medida que novos requisitos são descobertos.

Um estágio só pode ser iniciado depois que o anterior tenha sido concluído. Entretanto, Sommerville (2011, p. 21) afirma que, “na prática, esses estágios se sobrepõem e alimentam uns aos outros de informações”. Por exemplo, durante o projeto, os problemas com os requisitos são identificados. O que acontece é que um processo de software não é um modelo linear simples, sequencial, mas envolve interações entre as fases. Os artefatos de software que são produzidos em cada uma dessas fases podem ser modificados, a fim de refletirem todas as alterações realizadas em cada um deles.

Pressman (2011) aponta alguns problemas que podem ser encontrados quando o modelo em cascata é aplicado:

1. Os projetos que acontecem nas empresas dificilmente seguem o fluxo sequencial proposto pelo modelo. Alguma iteração sempre ocorre e traz problemas na aplicação do paradigma.
2. Na maioria das vezes, o cliente não consegue definir claramente todas as suas necessidades e o modelo em cascata requer essa definição no início das atividades. Portanto, esse modelo tem dificuldade em acomodar a incerteza natural que existe no começo de muitos projetos.
3. Uma versão operacional do sistema somente estará disponível no final do projeto, ou seja, o cliente não terá nenhum contato com o sistema durante o seu desenvolvimento. Isso leva a crer que, se algum erro grave não for detectado durante o desenvolvimento, o sistema não atenderá de forma plena as necessidades do cliente.

Segundo Sommerville (2011) e Pressman (2011), o modelo em cascata deve ser utilizado somente quando os requisitos são fixos e com pouca probabilidade de alteração durante o desenvolvimento do sistema ou se o trabalho deve ser realizado de forma linear até a sua finalização.

MODELO DE DESENVOLVIMENTO INCREMENTAL

O desenvolvimento incremental, segundo Sommerville (2011, p. 21), tem como base a ideia de “desenvolver uma implementação inicial, expô-la aos comentários dos usuários e continuar por meio da criação de várias versões até que um sistema adequado seja desenvolvido”. Tal afirmação pode ser melhor compreendida por meio da figura a seguir:

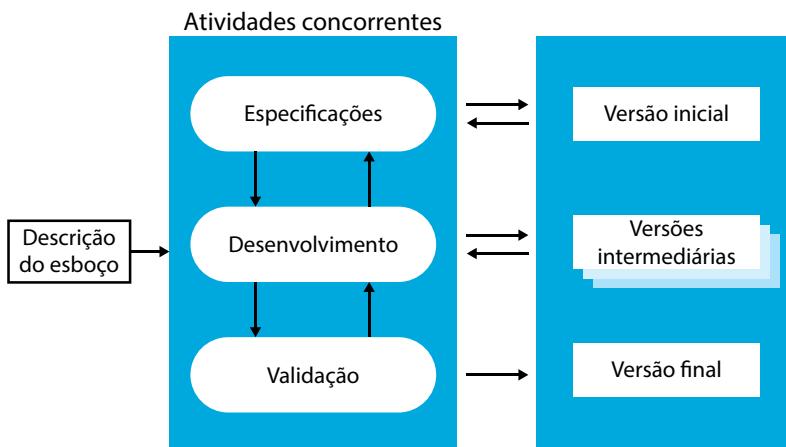


Figura 3 – Atividades concorrentes

Fonte: Sommerville (2011, p. 22).

Assim, as atividades de especificação, desenvolvimento e validação são realizadas concorrentemente com um rápido feedback entre todas as atividades. Em cada nova versão, o sistema incorpora novos requisitos definidos pelo cliente.

Para Pressman (2011), inicialmente, é necessário desenvolver um projeto rápido, o qual deve se concentrar em realizar uma representação daqueles aspectos do software que serão visíveis aos usuários finais, como o layout da interface com o usuário, por exemplo.

Além disso, desenvolvimento incremental apresenta algumas vantagens importantes em relação ao modelo em cascata. Sommerville (2011) aponta três delas, a saber:

- Se o cliente mudar seus requisitos, o custo será reduzido, pois a quantidade de análise e a documentação a ser feita serão menores do que no modelo em cascata.
- É mais fácil obter um retorno dos clientes sobre o desenvolvimento que foi feito, pois a clientela acompanha o desenvolvimento do software à medida que novas versões lhes são apresentadas.
- Os clientes podem começar a utilizar o software logo que as versões iniciais forem disponibilizadas, o que não acontece com o modelo em cascata.

Entretanto, a partir de uma perspectiva de engenharia e de gerenciamento, existem alguns problemas:

- 1. O processo não é visível:** os gerentes necessitam que o desenvolvimento seja regular, para que possam medir o progresso. Se os sistemas são desenvolvidos rapidamente, não é viável produzir documentos que refletem cada versão do sistema.
- 2. Os sistemas frequentemente são mal estruturados:** a mudança constante tende a corromper a estrutura do software. Assim, incorporar modificações no software torna-se cada vez mais difícil e oneroso.
- 3. Podem ser exigidas ferramentas e técnicas especiais:** elas possibilitam rápido desenvolvimento, mas podem ser incompatíveis com outras ferramentas ou técnicas. Além disso, relativamente, poucas pessoas podem ter a habilitação necessária para utilizá-las.

Para sistemas pequenos (com menos de 100 mil linhas de código) ou para sistemas de porte médio (com até 500 mil linhas de código), com tempo de vida razoavelmente curto, a abordagem incremental de desenvolvimento talvez seja a melhor opção. Contudo, para sistemas de grande porte, os quais possuem longo tempo de vida e várias equipes desenvolvem diferentes partes do sistema, os problemas de se utilizar o desenvolvimento incremental tornam-se particularmente graves. Para esses sistemas, é recomendado um processo misto, o qual incorpore as melhores características do modelo de desenvolvimento em cascata e do incremental, ou, ainda, algum outro modelo disponível na literatura.

ENGENHARIA DE SOFTWARE ORIENTADA A REUSO

Na maioria dos projetos de software, ocorre algum reuso de software, pois, normalmente, a equipe que trabalha no projeto conhece projetos ou códigos análogos ao que está sendo desenvolvido. Assim, essa equipe busca esses códigos, faz as modificações conforme a necessidade do cliente e as incorporam em seus sistemas.

Independentemente do processo de software que está sendo utilizado, pode ocorrer esse reuso informal. No entanto, nos últimos anos, uma abordagem para o desenvolvimento de software, com foco no reuso de software existente, tem emergido e se tornado cada vez mais utilizada. A abordagem orientada a reuso conta com um grande número de componentes de software reutilizáveis que podem ser acessados e com um *framework* de integração para esses componentes. Às vezes, esses componentes são sistemas propriamente ditos (sistemas COTS – *Commercial Off-The-Shelf* – ou Sistemas Comerciais de Prateleira, em português) e podem ser utilizados para proporcionar funcionalidade específica, como formatação de textos e cálculo numérico, por exemplo (SOMMERRVILLE, 2011).

O modelo genérico de processo baseado em reuso é mostrado na Figura 4. Note que, embora as etapas de especificação de requisitos e de validação sejam comparáveis com outros processos, as etapas intermediárias, em um processo orientado a reuso, são diferentes:

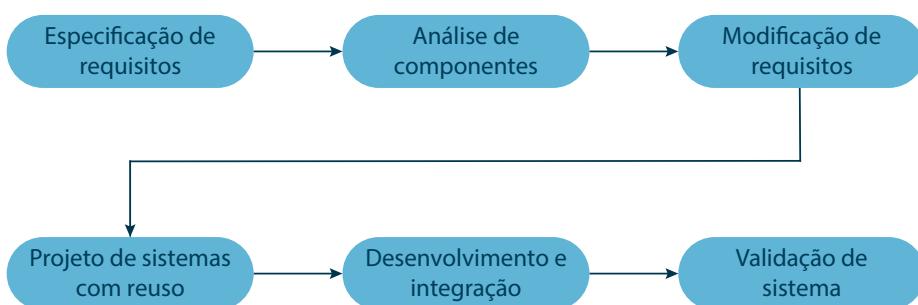


Figura 4 – Modelo genérico de processo baseado em reuso
Fonte: adaptado de Sommerville (2011).

Sommerville (2011) sustenta que essas etapas são:

1. **Análise de componentes:** considerando a especificação de requisitos, é feita uma busca de componentes para implementar essa especificação. Pode ser que não sejam encontrados componentes que atendam toda a especificação de requisitos, ou seja, pode ser fornecida somente parte da funcionalidade requerida.
2. **Modificação de requisitos:** no decorrer dessa etapa, os requisitos são analisados, levando-se em consideração as informações sobre os componentes que foram encontrados na etapa anterior. Se for possível, os requisitos são, então, modificados, para refletirem os componentes disponíveis. Quando isso não for possível, ou seja, quando as modificações forem impossíveis, a etapa de análise de componentes deverá ser refeita, a fim de buscar outras soluções.
3. **Projeto do sistema com reuso:** durante essa etapa, o *framework* do sistema é projetado ou, então, alguma infraestrutura existente é reutilizada. Os projetistas levam em consideração os componentes que são reusados e organizam o *framework* para tratar desse aspecto. Se os componentes reusáveis não estiverem disponíveis, pode ser necessário que um novo software deva ser projetado.
4. **Desenvolvimento e integração:** nessa etapa, o software que não puder ser comprado deverá ser desenvolvido e os componentes e sistemas COTS serão integrados, a fim de se criar um novo sistema. A integração de sistemas, nessa abordagem, pode ser parte do processo de desenvolvimento, em vez de uma atividade separada.

Deve-se tomar muito cuidado ao utilizar essa abordagem, pois não há como evitar as alterações nos requisitos dos usuários e isso pode acabar levando ao desenvolvimento de um sistema que não atenda às suas reais necessidades. Há, também, a possibilidade de que o controle da evolução do sistema fique comprometido, pois as novas versões dos componentes reusáveis não estão sob o controle da organização que as está utilizando.

De qualquer forma, a abordagem baseada em reuso tem a vantagem de proporcionar uma entrega mais rápida do software, pois reduz sensivelmente a quantidade de software que a empresa deve desenvolver. Assim, consequentemente, há uma diminuição nos custos de desenvolvimento, bem como em seus riscos.

ATIVIDADES BÁSICAS DO PROCESSO DE SOFTWARE

Caro(a) aluno(a), estudando os modelos de processo de software apresentados, é possível notar que algumas atividades estão presentes em todos eles. Somente algumas vezes que essas atividades estão organizadas de forma diferente, dependendo do processo de software que está sendo considerado. Sommerville (2011) afirma que, no modelo em cascata, essas atividades são organizadas de forma sequencial, ao passo que, no desenvolvimento incremental, são intercaladas. Contudo, a maneira como essas atividades serão realizadas depende do tipo de software, das pessoas e da organização envolvida.

As atividades básicas do processo de software são quatro: a especificação, o projeto e a implementação, a validação e a evolução. A partir de agora, iremos detalhar, de forma genérica, sem considerar um processo de software em específico, cada uma dessas atividades.

ESPECIFICAÇÃO DE SOFTWARE

A especificação de software ou de engenharia de requisitos é a primeira atividade básica de um processo de software e tem, como objetivo, definir quais funções são requeridas pelo sistema e identificar as restrições sobre a operação e o desenvolvimento desse sistema. Essa atividade é muito importante e crítica, pois, se a definição dos requisitos não for bem realizada, com certeza, problemas posteriores no projeto e na implementação do sistema irão acontecer.

Segundo Sommerville (2011, p. 24), “o processo de engenharia de requisitos tem como objetivo produzir um documento de requisitos acordados que especifica um sistema que satisfaz os requisitos dos *stakeholders*”. Além disso, é composto por quatro fases, conforme descreve Sommerville (2011):

- 1. Estudo de viabilidade:** uma avaliação é realizada para verificar se as necessidades dos usuários podem ser atendidas com base nas atuais tecnologias de software e hardware disponíveis na empresa. Esse estudo deve indicar se o sistema proposto será viável, do ponto de vista comercial, e se poderá ser desenvolvido considerando as restrições orçamentárias, caso existam. Um estudo de viabilidade não deve ser caro e demorado, pois

é a partir do seu resultado que a decisão de prosseguir com uma análise mais detalhada deve ser tomada.

2. **Levantamento e análise de requisitos:** nesta fase, é necessário levantar os requisitos do sistema, mediante a observação de sistemas já existentes, pela conversa com usuários e compradores em potencial, pela análise de tarefas etc. Além disso, essa fase pode envolver o desenvolvimento de um ou mais diferentes modelos e protótipos de sistema, a fim de ajudar a equipe de desenvolvimento a compreender melhor o sistema a ser especificado.
3. **Especificação de requisitos:** é a atividade de traduzir as informações coletadas durante a fase anterior em um documento que define um conjunto de requisitos. Tanto os requisitos dos usuários quanto os requisitos de sistema podem ser incluídos nesse documento. De acordo com Sommerville (2011, p. 25), “os requisitos dos usuários são declarações abstratas dos requisitos do sistema para o cliente e usuário final do sistema; requisitos de sistema são uma descrição mais detalhada da funcionalidade a ser provida”.
4. **Validação de requisitos:** essa atividade verifica os requisitos quanto a sua pertinência, consistência e integralidade. Durante o processo de validação, os requisitos que apresentarem problemas devem ser corrigidos, para que a documentação de requisitos fique correta.

As atividades de análise, definição e especificação de requisitos são intercaladas. Em outras palavras, elas não são realizadas seguindo uma sequência rigorosa, pois, com certeza, novos requisitos surgem ao longo do processo.

PROJETO E IMPLEMENTAÇÃO DE SOFTWARE

Segundo Sommerville (2011, p. 25), “o estágio de implementação do desenvolvimento de software e o processo de conversão de uma especificação do sistema em um sistema executável”. Essa etapa sempre envolve processos de projeto e programação de software, porém, se uma abordagem incremental de desenvolvimento for utilizada, poderá, também, envolver o aperfeiçoamento da especificação de software em que os requisitos foram definidos.

Para Pressman (2011, p. 205), o projeto de software “cria uma representação ou modelo do software, [...] fornecendo detalhes sobre a arquitetura do software, estruturas de dados, interfaces e componentes fundamentais para implementar

o sistema”. Diante disso, o projeto de software é aplicado independentemente do modelo de processo de software que está sendo utilizado, ou seja, se está sendo utilizado o modelo em cascata ou a abordagem incremental.

O início do projeto ocorre assim que os requisitos tiverem sido analisados e modelados, ou seja, no momento em que a modelagem do sistema tiver sido realizada. Com base no documento de requisitos, o engenheiro de software, na fase de modelagem do sistema, deverá elaborar os diagramas da UML – *Unified Modeling Language* –, como o diagrama de caso de uso e o diagrama de classes, por exemplo.

Na fase de projeto do sistema, o engenheiro de software deverá: i) definir o diagrama geral do sistema; ii) elaborar as interfaces com o usuário (telas e relatórios); e iii) desenvolver um conjunto de especificações de casos de uso. É necessário reforçar que essas especificações devem conter detalhes suficientes para permitir a sua codificação.

Geralmente, durante o projeto, o analista de sistemas deve definir cada componente do sistema ao nível de detalhamento que se fizer necessário para a sua implementação em uma determinada linguagem de programação.

A programação, a qual também é chamada de fase de implementação de um projeto típico, normalmente, começa quando termina a atividade de projeto. Ela envolve a escrita de instruções em Java, C++, C# ou em alguma outra linguagem de programação, a fim de implementar o que o analista de sistemas modelou na etapa de projeto. Sendo uma atividade pessoal, não existe um processo geral que seja normalmente seguido durante a programação do sistema.

Alguns programadores começam com os componentes que eles compreendem melhor, passando, depois, para os mais complexos. Outros preferem deixar os componentes mais fáceis para o fim, porque sabem como desenvolvê-los. Já alguns desenvolvedores gostam de definir os dados no início do processo e, então, utilizam esses dados para dirigir o desenvolvimento do programa; outros deixam os dados sem especificação, enquanto for possível.

De acordo com Sommerville (2011), normalmente, os programadores testam os códigos fontes que eles mesmos desenvolveram, a fim de descobrirem defeitos que devem ser removidos. Esse processo é chamado de depuração. O teste e

a depuração dos defeitos são processos diferentes. O teste estabelece a existência de defeitos, enquanto a depuração se ocupa em localizar e corrigir esses defeitos.

Em um processo de depuração, os defeitos no código devem ser localizados e o código precisa ser corrigido, a fim de se cumprir os requisitos. Para que se possa garantir que a mudança foi realizada corretamente, os testes deverão ser repetidos. Portanto, o processo de depuração é parte tanto do desenvolvimento quanto do teste do software.

VALIDAÇÃO DE SOFTWARE

A validação de software se dedica em mostrar que um sistema atende tanto as especificações relacionadas no documento de requisitos quanto às expectativas dos seus usuários. A principal técnica de validação, segundo Sommerville (2011), é o teste de programa, em que o sistema é executado com dados de testes simulados. Os testes somente podem ser realizados enquanto unidade isolada se o sistema for pequeno. Caso contrário, se o sistema for grande e constituído a partir de subsistemas, os quais são construídos partindo-se de módulos, o processo de testes deve evoluir em estágios, ou seja, devem ser realizados de forma incremental, iterativa.

Sommerville (2011) propõe um processo de teste em três estágios. O primeiro é o teste de componente. Em seguida, tem-se o sistema integrado e testado e, por fim, o sistema é testado com dados reais, ou seja, com dados do próprio cliente. Idealmente, os defeitos de componentes são descobertos no início do processo, enquanto os problemas de interface são encontrados quando o sistema é integrado.

Os estágios do processo de testes, de acordo com Sommerville (2011), são:

- 1. Testes de desenvolvimento:** para garantir que os componentes individuais estão operando corretamente, é necessário testá-los de forma independente dos outros componentes do sistema.
- 2. Testes de sistema:** os componentes são integrados para constituírem o sistema. Esse processo se dedica a encontrar erros que resultem de interações não previstas entre os componentes e de problemas com a interface

do componente. O teste de sistema também é utilizado para validar que o sistema atende aos requisitos funcionais e não funcionais definidos no documento de requisitos.

3. **Teste de aceitação:** nesse estágio, o sistema é testado com dados reais fornecidos pelo cliente. Aqui, há a possibilidade de se mostrar falhas na definição de requisitos, pois os dados reais podem exercitar o sistema de modo diferente dos dados de teste.

EVOLUÇÃO DE SOFTWARE

Depois que um software é colocado em funcionamento, ou seja, depois que é implantado, com certeza, ocorrerão mudanças que levarão à alteração desse software. Essas mudanças podem ser, de acordo com Pressman (2011), para correção de erros não detectados durante a etapa de validação do software, quando há adaptação a um novo ambiente, quando o cliente solicita novas características ou funções, ou ainda, quando a aplicação passa por um processo de reengenharia, a fim de proporcionar benefício em um contexto moderno.

Sommerville (2011) sustenta que, historicamente, sempre houve uma fronteira entre o processo de desenvolvimento de software e o seu processo de evolução (manutenção de software). O desenvolvimento de software é visto como uma atividade criativa, em que o software é desenvolvido a partir de um conceito inicial, até chegar ao sistema em operação.

Depois que esse sistema entra em operação, inicia-se a manutenção de software, no qual ele é modificado. Normalmente, os custos de manutenção são maiores do que os custos de desenvolvimento inicial, mas os processos de manutenção são considerados menos desafiadores do que o desenvolvimento de software original, ainda que tenha um custo mais elevado.

Todavia, atualmente, os estágios de desenvolvimento e de manutenção têm sido considerados integrados e contínuos, em vez de dois processos separados. Assim, tem sido mais realista pensar na engenharia de software enquanto um processo evolucionário, em que o software é sempre mudado ao longo de seu período de vida, em resposta aos requisitos que estão em constante modificação e às necessidades do cliente.



METODOLOGIAS ÁGEIS

As metodologias ágeis, segundo Sbrocco e Macedo (2012), têm sido muito utilizadas como uma alternativa para as abordagens tradicionais. Isso acontece, pelo fato de terem menos regras, menos burocracias e serem mais flexíveis, por permitirem ajustes durante o desenvolvimento do software.

O mercado de desenvolvimento de software está cada vez mais dinâmico e os negócios estão ficando cada vez mais competitivos. Assim, quem possui maior acesso à informação de qualidade vai possuir vantagens sobre os concorrentes (SILVA; SOUZA; CAMARGO, 2013).

Entretanto, as empresas enfrentam algumas dificuldades na hora de escolher apenas uma metodologia dentre as várias disponíveis no mercado para o gerenciamento de projetos de software. De acordo com Silva, Souza e Camargo (2013, p. 41), o tipo de empresa pode influenciar, visto que:

As empresas de grande porte e entidades governamentais tem como prática o detalhamento de vários processos por relatórios, planilhas e gráficos, sendo comumente utilizada a metodologia tradicional para projeto de software. Por outro lado as empresas pequenas e médias encontram dificuldades em utilizar a metodologia tradicional, por ser muito caro a manutenção e controle, e o longo prazo para entrega. A metodologia ágil com custo baixo e entregas rápidas pode contribuir no desenvolvimento de projeto de software para qualquer tipo de empresa e ser agregada a metodologia tradicional.

Segundo Sbrocco e Macedo (2012, p. 183), “as metodologias tradicionais são mais ‘pesadas’ e devem ser utilizadas em situações em que os requisitos do sistema são estáveis”. No entanto, se os requisitos forem passíveis de mudanças e a equipe for pequena, é interessante utilizar as metodologias ágeis.

As metodologias tradicionais baseiam-se em um conjunto de atividades predefinidas durante o processo de desenvolvimento, que se inicia com o levantamento de requisitos, segue para o projeto e modelagem, vai para a implementação, teste, validação e chega até à manutenção. Em todas as atividades que estão ligadas ao processo de desenvolvimento, temos uma forte documentação que deve ser considerada.

Essas documentações são consideradas como características limitadoras aos desenvolvedores, pois geram muitos processos burocráticos. Essa burocracia, segundo Sbrocco e Macedo (2012, p. 183), “é uma das principais razões que levam organizações de pequeno porte a não usar nenhum tipo de processo”. Contudo, ao optarem por não adotar nenhuma metodologia, essas empresas podem ter resultados indesejáveis na qualidade do software e, com isso, dificulta-se a entrega dentro dos prazos estabelecidos.

EXTREME PROGRAMMING (XP)

A *Extreme Programming* (Programação Extrema) usa uma abordagem orientada a objetos enquanto paradigma de desenvolvimento de software. A XP é considerada uma metodologia ágil, em que os projetos são conduzidos com base em requisitos que se modificam rapidamente (SBROCCO; MACEDO, 2012).



Assim como ocorre em muitas metodologias ágeis, a maioria das regras da XP causa polêmica à primeira vista e muitas não fazem sentido se aplicadas isoladamente, o que a torna praticável. Além disso, a sinergia de seu conjunto, a qual sustenta o sucesso da XP, encabeçou uma verdadeira revolução no conceito de desenvolvimento de software. A ideia básica é a de enfatizar o desenvolvimento rápido do projeto, visando garantir a satisfação do cliente, além de favorecer o cumprimento das estimativas (SBROCCO; MACEDO, 2012).

A metodologia XP tem, como meta, atender as necessidades dos clientes com mais qualidade, mais rapidez e de forma simples. Dessa forma, a XP busca desenvolver os projetos da forma mais rápida, entregando-os dentro do prazo estipulado, mesmo que os requisitos se alterem com frequência. De acordo com Martins (2007, p. 144), “a XP dá preferência ao desenvolvimento orientado a objetos e permite trabalhar com pequenas equipes de até 12 desenvolvedores (programadores)”.

Além do mais, essa metodologia utiliza um modelo incremental que, conforme o sistema é utilizado, novas melhorias ou mudanças podem ser implementadas. Com isso, o cliente sempre tem um sistema que pode utilizar, testar e avaliar se o que foi proposto foi desenvolvido corretamente. A XP é uma metodologia flexível e adaptativa, por isso, pode ser usada em vários projetos, desde que o projeto não tenha a necessidade de um maior formalismo e retrabalho ao longo do seu desenvolvimento (MARTINS, 2007).

SCRUM

As metodologias consideradas ágeis, assim como *Scrum*, são fortemente influenciadas pelas práticas da indústria japonesa, como as adotadas pelas empresas Toyota e Honda, por exemplo (SBROCCO; MACEDO, 2012). A origem do termo *scrum* surgiu de um famoso artigo escrito em 1986 por Takeuchi e Nonaka, intitulado *The new product development game* (O novo jogo do desenvolvimento de produtos, em português), em que foi sustentada a ideia de que equipes de projeto pequenas e multifuncionais produzem resultados melhores (PHAM; PHAM, 2011).

O uso dessa metodologia, segundo Pressman e Maxim (2016), vem de uma formação que ocorre durante uma partida de *rugby*. Essa formação ocorre após

uma parada ou quando a bola sai de campo e o técnico reúne a equipe de jogadores. Assim, a metodologia *Scrum* foi desenvolvida em 1990 por Jeff Sutherland e Ken Schwabe. Segundo esses estudiosos, *Scrum* é um *framework* para desenvolver e manter produtos complexos. Em outras palavras, é uma metodologia ágil de software que concentra as suas atenções no produto final, com um rápido desenvolvimento, e nas interações dos indivíduos.

Além do mais, essa metodologia segue os princípios do Manifesto Ágil, os quais, conforme Pressman e Maxim (2016, p. 78) asseveram, “são usados para orientar as atividades de desenvolvimento dentro de um processo que incorpora as seguintes atividades metodológicas: requisitos, análise, projeto, evolução e entrega”.

O *Scrum* possui algumas características consideradas importantes. São elas: flexibilidade de resultados e prazos, trabalho com equipes pequenas, uso de revisões frequentes, colaboração dos interessados e se baseia na orientação a objetos (SBROCCO, MACEDO, 2012).

De acordo com Sbrocco e Macedo (2012, p. 161), o *Scrum* é “conhecido pelo fato de estruturar seu funcionamento por ciclos, chamados de *sprints* que representam iterações de trabalho com duração variável”. As tarefas desenvolvidas dentro de um *sprint* são adaptadas ao problema pela equipe *scrum* e são realizadas em um período curto de tempo durante o desenvolvimento do projeto. Os estudiosos ainda sustentam que não pode ocorrer nenhuma mudança durante uma *sprint*, pois o produto é projetado, codificado e testado nesse momento.

Além disso, o *Scrum* estipula um conjunto de práticas e regras que devem ser seguidas pela equipe. Nele, também temos três **papéis** importantes: *Product Owner*, *Scrummaster* e *Team* (BROD, 2013), bem como ocorrem as **cerimônias**, as quais são divididas em quatro, a saber: *Daily Meeting* ou *Daily Scrum*, a *Sprint Review*, o *Sprint Planning* e o *Sprint Retrospective*.

Segundo Sbrocco e Macedo (2012, p. 165), *Sprint Planning* ou Planejamento da *Sprint* “é a primeira reunião do projeto e todos precisam participar; deve ter uma duração de no máximo 8 horas. Essa é a reunião em que o *Product Owner* planeja e elabora a lista de prioridades”. Na *Daily Meeting* ou *Daily Scrum*, ou, ainda, Reunião Diária, “cada membro do time deve responder sobre o que já fez, sobre o que pretende fazer e se há algum impedimento para a conclusão da(s)

tarefa(s)” (SBROCCO; MACEDO, 2012, p. 166). É uma reunião que ocorre diariamente e deve ser rápida, com duração de, no máximo, 15 minutos, tendo participantes do time e o *Scrum Master*.

Na reunião *Sprint Review* ou Revisão da *Sprint*, é feito um balanço sobre tudo o que foi feito durante uma *sprint*. Nela, o time deve mostrar apenas os resultados concluídos da *sprint* para o *Product Owner* e seus convidados. Já a reunião *Sprint Retrospective* ou Retrospectiva da *Sprint* tem, por objetivo, verificar o que foi bom e o que pode ser melhorado na *sprint*. Para Brod (2013), é uma reunião em que “se lava a roupa suja”, ou seja, procura-se entender o que deu errado e o que pode ser ajustado nas *sprints*, a fim de melhorá-las.

A partir dessas cerimônias, são gerados os **artefatos**, que são os documentos *Product Backlog*, *Sprint Backlog* e o *Burndown Chart* (SBROCCO; MACEDO, 2012).

O *Scrum* reconhece, enquanto princípio básico, o fato de que os clientes podem mudar de ideia durante o desenvolvimento do projeto e, por isso, adota uma abordagem empírica. Em outras palavras, aceita-se que o problema do cliente pode não ser totalmente entendido (SBROCCO; MACEDO, 2012).

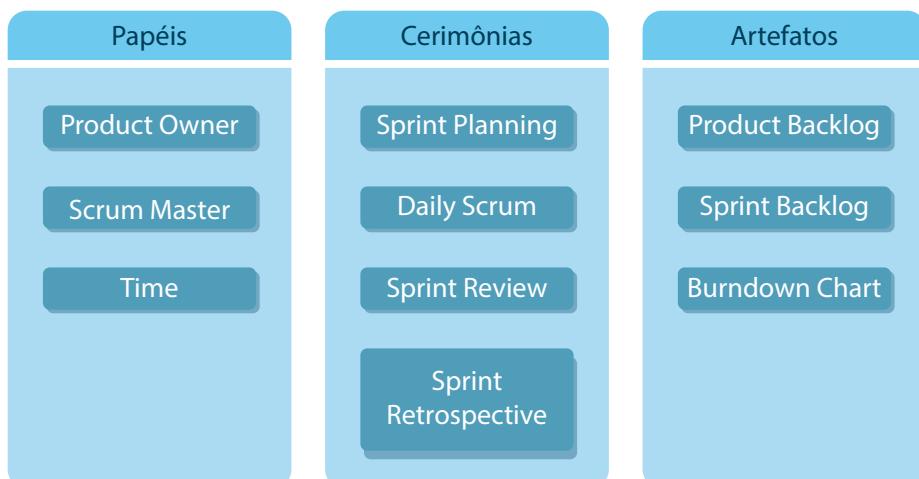


Figura 5 – Fundamentos básicos do Scrum

Fonte: a autora.

Para entender o funcionamento da metodologia *Scrum*, precisamos conhecer o fluxo geral do processo, o qual será apresentado na Figura 6, em concomitância com seus conjuntos de atividades de desenvolvimento.

Tudo se inicia com o ***Product Backlog*** (*Backlog* do Produto), que é uma lista de prioridades dos requisitos ou de funcionalidades para o projeto. O termo *backlog* é um histórico (*log*) de trabalhos realizados em um determinado período de tempo. Também são estimados os custos e riscos, bem como são definidas as ferramentas de desenvolvimento, a equipe e as datas de entrega para os resultados (SBROCCO; MACEDO, 2012).

Após o *product backlog* ser definido, segundo Sbrocco e Macedo (2012), a equipe passa a realizar a ***Sprint Backlog***. Ela é uma lista de funcionalidades que serão realizadas no próximo *sprint*, onde também são definidas as funções de cada membro da equipe.

Quando falamos de *sprint*, pensamos em planejamento e em itens do *product backlog*, que são os requisitos do projeto e serão transformados em funcionalidade dentro da próxima *Sprint*, para entrega ao cliente. Contudo, antes disso, é necessário se preparar para a *sprint*, pois para se trabalhar nos itens é preciso fazer o levantamento junto ao cliente e depois, analisá-los e procurar entendê-los, além de fazer o detalhamento desses itens.

Após o final de cada *sprint*, um incremento do produto é apresentado ao cliente e, caso surjam defeitos, eles devem ser incluídos ao *backlog* do produto. Conforme o ciclo de desenvolvimento ocorre, são adotados os mecanismos de controle do *Scrum*. São exemplos: controle de funcionalidades que não foram entregues, mudanças que podem ocorrer devido a defeitos corrigidos, problemas técnicos e controle de riscos (SBROCCO; MACEDO, 2012).

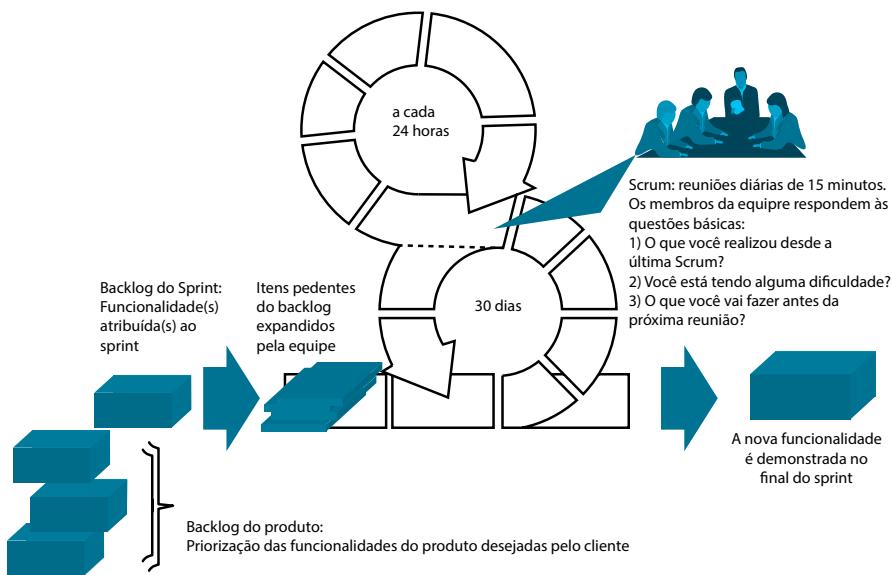


Figura 6 - Fluxo do Processo Scrum

Fonte: Pressman e Maxim (2016, p. 78).

Exemplo:

Para explicar melhor o funcionamento do Scrum, vamos apresentar um exemplo de estudo de caso. Imagine uma empresa chamada TempoSite, a qual tem um cliente chamado Sr. Dobro, que quer lançar um site para a sua empresa. Assim, o gerente de projetos (Product Owner) faz um *brieffing* e levanta, junto ao Sr. Dobro, todas as necessidades que ele precisa para o site, por meio da criação de *Users Stories* – estórias de usuário –, uma curta e simples descrição das necessidades do usuário do produto.

O gerente se reúne (Reunião de Planejamento) com a equipe de desenvolvimento (*designers, front-end e back-end*), para que, juntos, definam todas as tarefas necessárias para produzir o projeto e formalizar tudo em uma lista de tarefas (*Product Backlog* - exemplificado na Tabela 1).

Imagine que você é o *Scrum Master*, o qual é encarregado de facilitar e organizar as reuniões, a fim de garantir que todos entendam as atividades a serem desempenhadas. Para cumprir as metas estabelecidas, a equipe decide que vai entregar a página inicial do site pronta (objetivo da reunião) até o final da semana (*sprint*) (GUERRATO, 2013, on-line)¹.

Após isso, a equipe passa a dividir o objetivo do que será desenvolvido em tarefas menores (*Sprint Backlog* – exemplificado na Tabela 2) para criar o layout, como desenvolver os códigos e implantar um sistema de administração de conteúdo. Depois dessas ações, testa-se tudo em diversos *browsers*, para que não haja erros.

Assim, cada um da equipe escolhe as tarefas que irá realizar, definindo um prazo de duração em horas para cada tarefa. Diariamente, a equipe se reúne durante 15 minutos para acompanhar o andamento do projeto (*Daily Meeting ou Daily Scrum* ou Reunião Diária).

No fim da semana, a equipe apresenta o resultado esperado (*Sprint Review* ou Revisão da Sprint). Com base nos resultados apresentados, todos passam a refletir sobre as dificuldades encontradas e quais são as possíveis soluções que podem ser feitas para a próxima semana (*Sprint Retrospective* ou Retrospectiva da Sprint), desenvolvendo o Burndown Chart (GUERRATO, 2013, on-line)¹.

Ao final, o mais importante é que seja entregue ao cliente uma funcionalidade ou produto parcial (incremento) funcional. Dessa forma, a partir do feedback do cliente, a equipe pode decidir a próxima tarefa a ser executada na próxima semana (GUERRATO, 2013, on-line)¹, recomeçando, assim, o ciclo do *Scrum*.

Tabela 1 - Exemplo de *Product Backlog*

PRODUTO BACKLOG

PRODUTO	Fábrica de Software	SM	<Nome do Scrum Master>	CONCLUÍDO		
PO	Nome do Product	ATUALIZADO EM	<Data>	EM ANDAMENTO		
				CANCELADO		
ID	PRIORIDADE	HISTÓRIA DE USUÁRIO/ REQUISITO/ ITEM	TIPO	QUEM	ESTIMATIVA	STATUS
100	Alta	Captação de recursos para o projeto e contratação da equipe	Funcionalidade	CB	3	CONCLUÍDO
101	Alta	Reunião de início do projeto e definição do primeiro sprint backlog	Funcionalidade	CB	2	EM ANDAMENTO
102	Média	Definição de layout, levantamento de necessidade e ferramentas adotadas	Funcionalidade	CB	2	EM ANDAMENTO
103	Alta	Aquisição de equipamento "host", instalação, registro de domínio, implantação do CL	Correção	TBD	1	EM ANDAMENTO
104	Baixa	Criação do sistema do projeto	Funcionalidade	DEVEL	1	EM ANDAMENTO

Fonte: adaptado de Brod (2013, p. 51).

Tabela 2 - Exemplo de *Sprint Backlog*

COLABORADOR	TAREFAS	SEMANA 1 (ESTIMADA)	SEMANA 1 (REALIZADA)	TAREFAS	SEMANA 2 (ESTIMADA)	SEMANA 2 (REALIZADA)
Vinicius de Moraes	Desenho de interface	40	50	Aplicação do layout à interface	40	35
Tom Jobim	Orientação dos elementos que compõem o layout e adequação aos formatos padrões	20	30	Criação do paper prototype para estar com o cliente	40	50
Baden Powel	Montagem de infraestrutura	20	20	Criação da estrutura	10	15
TOTAL		80	100		90	100

Fonte: adaptado de Brod (2013, p. 52).

Já para uma *sprint* de quatro semanas, com 160 horas que são consumidas, observe a Tabela 3:

Tabela 3 - Consumo do tempo durante uma *sprint*

TAREFAS	HORAS RESTANTES	HORAS ESTIMADAS
0	160	160
1	90	100
2	55	60
3	20	25
4	0	0

Fonte: Brod (2013, p. 60).

No exemplo apresentado, foram consideradas as horas totais das pessoas envolvidas no projeto, conforme a Tabela 3. Começamos a semana com 160 horas e foi previsto o consumo de 40 horas na primeira semana, mas foram consumidas apenas 50 horas. O gráfico da Figura 7, a seguir, mostra o quanto a estimativa inicial foi desviada. Contudo, ao término, o que havia sido prometido foi cumprido dentro do prazo estimado no *sprint*.

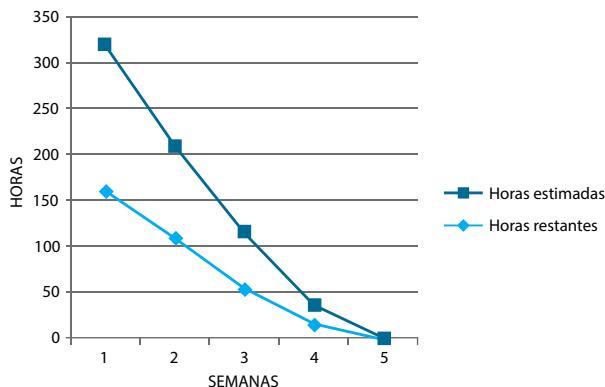


Figura 7 - Exemplo de *Burndown Chart*

Fonte: Brod (2013, p. 60).

Na figura 7, temos: i) a coluna vertical, que mostra a quantidade de esforços/ horas; ii) a coluna horizontal, a qual mostra as semanas de uma *sprint*; e iii) a linha azul, que mostra o fluxo ideal de trabalho. Assim, quando o gráfico está acima da linha azul, indica que o *Team* está longe da meta. Já quando o gráfico está em cima da linha azul, indica que o *Team* está no fluxo ideal de trabalho e, por sua vez, quando está abaixo da linha, indica que o time está acima das expectativas.

SAIBA MAIS

Durante o planejamento da sprint podemos utilizar uma técnica que estima o tamanho do trabalho a ser realizado, aplicada a uma dinâmica de grupo conhecida como “pôquer do planejamento”. Essa técnica dá a impressão de que os jogadores (time de desenvolvimento) estão jogando cartas, como num jogo de pôquer. Como vimos, no SCRUM não pensamos em tempo, ou seja, a duração das tarefas, mas no tamanho delas. Então, para que possamos definir esse tamanho, a equipe senta ao redor de uma mesa e um dos integrantes apresenta para todos uma “história”, que está relacionada a uma funcionalidade que deve ser desenvolvida. Após o time de desenvolvimento entender claramente o objetivo da história, cada membro do time escolhe uma carta, que foi previamente distribuída a todos, e coloca na mesa virada para baixo. As cartas apresentam valores numéricos relacionados com o grau de complexidade da funcionalidade a ser desenvolvida, sob o ponto de vista de cada membro da equipe. O objetivo é obter um consenso relacionado ao tamanho do desenvolvimento da funcionalidade.

Fonte: Sbrocco e Macedo (2012).

Entretanto, qual das metodologias é a melhor? Depende do contexto do software a ser desenvolvido. Mesmo considerando as vantagens das metodologias ágeis no desenvolvimento de software, sempre devemos pensar “no que” e “para que” será desenvolvido (SBROCCO; MACEDO, 2012). Para os softwares que são críticos, cuja precisão, risco e confiabilidade são decisivos, não podemos dispensar as recomendações que são impostas pelas regras e normas das metodologias tradicionais, principalmente em relação à documentação.

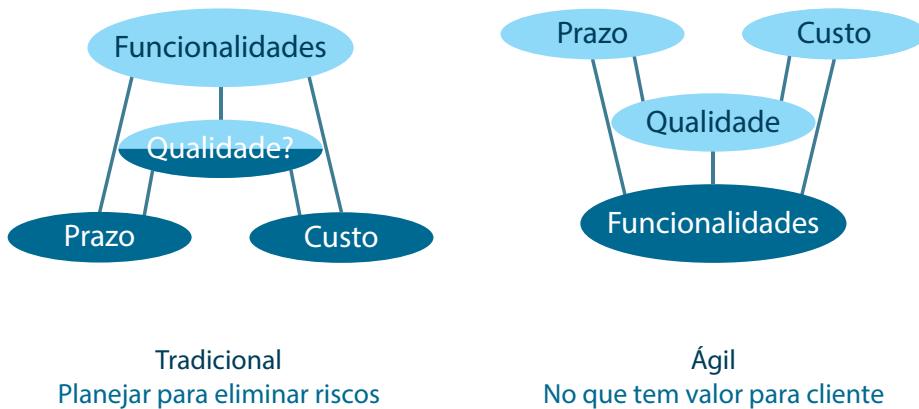


Figura 8 – Metodologia Tradicional X Metodologia Ágil

Fonte: a autora.

Para ficar mais fácil de entender as diferenças entre os aspectos das metodologias, vejamos o Quadro 1, o qual estabelece uma visão geral acerca das principais características entre as metodologias ágeis e as tradicionais.

Quadro 1 - Comparação entre as características das Metodologias Tradicionais e Metodologias Ágeis

ASPECTO	METODOLOGIA TRADICIONAL	METODOLOGIA ÁGIL
Objetivo	Orientado por atividades e foco nos processos.	Orientado por produto e foco nas pessoas.
Projeto	Estável e inflexível a mudança	Adaptável a mudanças.
Tamanho	Qualquer tamanho, efetivo em projetos de longa duração.	Pequeno, mas pode ser usado em projetos de maior porte.
Gerente de projeto	Controle total do projeto.	Papel de facilitador ou coordenador.
Equipe do Projeto	Atuação com papéis claros e bem definidos em todas as atividades.	Atuação colaborativa em todas as atividades.
Cliente	Participa nas fases iniciais do levantamento de requisitos.	Parte integrante da equipe do projeto.
Planejamento	Detalhado e os envolvidos não participam.	Curto e com a participação de todos os envolvidos.

Fonte: a autora.

Independentemente de qual metodologia a empresa escolha, o importante, em qualquer projeto de desenvolvimento de software, é a qualidade final dos produtos e a satisfação do cliente.



CONSIDERAÇÕES FINAIS

Nesta primeira unidade, foram apresentados alguns conceitos básicos sobre a engenharia de software que serão utilizados no decorrer de todo o livro. Por isso, é muito importante que esses conceitos fiquem bem claros.

A engenharia de software foi proposta como uma tentativa de levar a precisão da engenharia para o desenvolvimento de software, pois, antes do seu surgimento, desenvolver um software era algo que não podia ser mensurado, nem em relação ao tempo, nem em relação ao custo, levando-se, normalmente, muito mais tempo do que o previsto. O que acontecia era o fato de que não se tinha uma regra, ou seja, uma sequência de atividades para o seu desenvolvimento.

Assim, você vai perceber, na próxima unidade, que para tentar solucionar esse problema, os estudiosos da engenharia de software propuseram vários modelos de processos de software e a empresa pode escolher o que melhor se adequa a ela. Tudo isso tem ajudado muito o desenvolvimento de software.

Aprendemos, também, o que é um processo de software e conhecemos alguns modelos. Um processo de software é um conjunto de atividades com resultados (artefatos) associados a cada uma delas, o que leva a produção de um software. Todo software deve ser especificado, projetado, implementado e validado.

Além disso, após o uso do software feito pelo usuário, ele passa por evoluções. Todas essas etapas são muito importantes, mas constatamos que a especificação do software é uma etapa imprescindível nesse conjunto, pois, se os requisitos não forem esclarecidos e bem especificados no início do desenvolvimento, há uma grande chance de que o software não atenda às necessidades do cliente.

Esta primeira unidade foi somente um aperitivo. Agora, vamos passar a estudar os assuntos específicos da disciplina e você vai perceber que a engenharia de software é muito importante para o desenvolvimento de um software.

ATIVIDADES



1. Segundo Pressman (2011), julgue as afirmativas a seguir, as quais diferenciam o software do hardware:

- I. Software não é fabricado no sentido clássico.
- II. Software se desgasta.
- III. Software possui componentes reutilizáveis.
- IV. Software não possui seguimento de instruções.
- V. Software é a parte lógica.

É correto o que se afirma em:

- a) I, apenas.
- b) I, II e IV, apenas.
- c) I, III e V, apenas.
- d) II, III e V, apenas.
- e) I, II, III, IV e V, apenas.

2. Quando falamos de engenharia de software, não se trata apenas do _____ em si, mas de toda a _____ associada e dados de configurações necessários para fazer esse programa operar corretamente.

Diante do exposto, assinale a alternativa que apresenta os termos que completam corretamente as lacunas:

- a) Projeto, documentação.
- b) Programa, atividade.
- c) Projeto, atividade.
- d) Programa, documentação.
- e) Resultado, atividade.

3. Sabemos que um processo de software é composto por um conjunto de etapas necessárias para que um software seja produzido. Dentre elas, estão:

- I. Modelo Cascata: considera as atividades de especificação, desenvolvimento, validação e evolução, as quais são fundamentais ao processo.
- II. Desenvolvimento Incremental: esse modelo intercala as atividades de especificação, desenvolvimento e validação.
- III. Engenharia de software orientada a reuso: parte do princípio de que não existem componentes que podem ser reutilizáveis.

ATIVIDADES



É correto o que se afirma em:

- a) I, apenas.
- b) I e II, apenas.
- c) I, II e III.
- d) II e III apenas.
- e) I e III apenas.

4. Para que um software seja produzido, é necessário concluir diversas etapas, as quais podem envolver o desenvolvimento de software a partir do zero. Sendo assim, quais são as atividades fundamentais para a engenharia de software, segundo Sommerville (2011)?
- a) Especificação, projeto e implementação, validação e evolução.
 - b) Projeto, implementação e validação.
 - c) Validação, evolução e projeto.
 - d) Integração, análise, projeto, evolução.
 - e) Análise, implementação, validação e evolução.
5. Sabemos que o modelo cascata possui uma abordagem sequencial e sistemática para o desenvolvimento de software. Dessa forma, alguns estágios são fundamentais para que haja esse desenvolvimento. Sobre a temática, assinale a alternativa incorreta:
- a) Análise e definição de requisitos: as funções, as restrições e os objetivos do sistema são estabelecidos por meio da consulta aos usuários do sistema.
 - b) Processo de sistemas e de software: o processo de projeto de sistemas não agrupa os requisitos em sistemas de hardware ou de software.
 - c) Implementação e teste de unidades: o teste de unidades envolve a verificação de que cada unidade atenda sua especificação.
 - d) Integração e teste de sistema: são integrados e testados como um sistema completo, a fim de garantir que os requisitos de software foram atendidos.
 - e) Operação e manutenção: a manutenção envolve corrigir os erros que não foram descobertos em estágios anteriores ao do ciclo de vida.



UMA PROPOSTA PARA O DESENVOLVIMENTO ÁGIL DE AMBIENTES VIRTUAIS

Processos ágeis de desenvolvimento têm como característica uma abordagem iterativa e incremental dos princípios de Engenharia de Software. Tal abordagem revela-se extremamente adequada aos projetos de Realidade Virtual e proporciona, por sua natureza evolutiva, grandes benefícios associados à gestão de riscos em projetos de software.

A palavra ágil foi relacionada pela primeira vez à Engenharia de Software em 2001, por um consórcio de especialistas em métodos de desenvolvimento, que elaboraram na ocasião o "Manifesto Ágil". Tal manifesto destaca alguns dos princípios compartilhados por diferentes metodologias de desenvolvimento, a partir de então denominados processos (ou métodos) ágeis:

- Indivíduos e interações são mais importantes que processos e ferramentas;
- Software funcionando é mais importante do que documentação detalhada;
- Colaboração dos clientes é mais importante de que negociação de contratos;
- Adaptação às mudanças é mais importante do que planejamento extensivo.

Por outro lado, o desenvolvimento de aplicações em Realidade Virtual requer pleno conhecimento e entendimento de diversas disciplinas, tais como computação gráfica, modelagem geométrica, interação multimodal, entre outras. Algumas características destes sistemas revelam a necessidade de um aprimoramento contínuo em seu processo de desenvolvimento. Dentre essas características, destacam-se:

- Rápida evolução da tecnologia relacionada à visualização e à capacidade gráfica dos computadores;
- Indecisão e mudanças de opinião por parte dos clientes, problema agravado em sistemas compostos por equipamentos de alto custo;
- Necessidade da implementação contínua de protótipos, visando a avaliação da viabilidade do sistema pelo cliente.

O desenvolvimento evolucionário, o planejamento adaptativo e o acolhimento de alterações nos requisitos apresentam-se como possíveis melhorias associadas ao desenvolvimento ágil de ambientes virtuais.

Desenvolvimento ágil de software

O desenvolvimento ágil de software é uma abordagem de desenvolvimento caracterizada pela adaptabilidade, ou seja, pela capacidade de absorver mudanças, sejam elas nas forças de mercado, nos requisitos do sistema, na tecnologia de implementação ou nas equipes de projeto.





Desenvolvimento de Sistemas de Realidade Virtual

O desenvolvimento de sistemas de Realidade Virtual, assim como o desenvolvimento de qualquer sistema de software, demanda um processo, uma metodologia de desenvolvimento. No entanto, tal processo deve se adequar à rápida evolução da tecnologia associada a estes sistemas.

Um processo de desenvolvimento que agrupa características de prototipagem, desenvolvimento iterativo e desenvolvimento evolucionário de sistemas de software baseia-se em conceitos e modelos da Engenharia de Software, adaptados às peculiaridades dos Sistemas de Realidade Virtual (SRV). Este processo compõe-se de cinco etapas, realizadas iterativamente: Análise de requisitos, Projeto, Implementação, Avaliação e Implantação.

Desenvolvimento ágil de ambientes virtuais

Nenhum processo de desenvolvimento pode garantir, simplesmente por sua aplicação, o sucesso do produto de software ao qual foi aplicado. No entanto, é possível destacar algumas características comuns a alguns processos bem-sucedidos software ao qual foi aplicado:

- Desenvolvimento iterativo: projetos maiores, com vários componentes, são mais propensos a problemas de integração. Um planejamento adequado de iterações reduz os problemas de integração e facilita o acompanhamento do processo de desenvolvimento pelos gerentes de projeto;
- Avaliação contínua do processo: nenhum processo de desenvolvimento de software pode garantir a total imunidade do projeto face a problemas como mudanças na equipe de desenvolvimento ou nos requisitos do usuário. A avaliação (e consequente adaptação) do processo de desenvolvimento é de fundamental importância ao longo do ciclo de vida do projeto;
- Boas práticas (“Best practices”): as melhorias associadas à utilização de boas práticas de desenvolvimento ou de padrões de projeto não devem ser negligenciadas em projetos de software.

A abordagem sugerida para a adaptação de um processo existente a determinado contexto consiste em personalizar um processo existente, testando e refinando-o de maneira iterativa, de acordo com as características de cada projeto.

Fonte: adaptado de Mattioli *et al.* (2009).

MATERIAL COMPLEMENTAR



LIVRO

Fundamentos de Engenharia de Software (2013)

Frank Tsui e Orlando Karam

Editora: LTC

Sinopse: em sua segunda edição, "Fundamentos da Engenharia de Software" apresenta uma introdução abrangente de temas e metodologias essenciais sobre o desenvolvimento de software. Ideal para estudantes universitários e para profissionais experientes à procura de uma nova carreira no setor de tecnologia da informação – ou áreas afins –, esta obra apresenta o ciclo de vida completo de um sistema de software – da concepção até a liberação e o suporte ao usuário. Este livro constitui um texto excepcional para aqueles que estão ingressando no estimulante mundo do desenvolvimento de software.



LIVRO

Métodos Ágeis para Desenvolvimento de Software

Rafael Prikladnicki, Renato Willi e Fabiano Milani

Editora: Bookman

Sinopse: este livro traz informações completas sobre conceitos e práticas ágeis. Reúne a visão de 23 profissionais de todo o país, agregando, em uma única obra, conhecimentos e experiências sobre o tema, de projetos pequenos a grandes, dos simples aos complexos, de empresas públicas às privadas.



NA WEB

Metodologia ágil FDD

Artigo que trata sobre o FDD (Feature-Driven Development), que significa desenvolvimento guiado a funcionalidades, assim como são o XP, ASD, Scrum e AUP. Ele faz parte das metodologias ágeis originais, sendo um modelo incremental e iterativo do processo de desenvolvimento de software, o qual tem, como lema, resultados frequentes, tangíveis e funcionais.

Web: <http://www.leandromtr.com/gestao/metodologia-agil-fdd/>

REFERÊNCIAS

- BROD, C. **Scrum**: Guia Prático para Projetos Ágeis. São Paulo: Novatec, 2013.
- CÂMARA, H. M. de A. **Uma abordagem sistemática para implementação, gerenciamento e customização de testes de linhas de produto de software**. 2011. Dissertação (Mestrado Ciência da Computação), Universidade Federal do Rio Grande do Norte, Natal.
- FERNANDES, A. A.; TEIXEIRA, D. de S. **Fábrica de Software**: implantação e gestão de operações. São Paulo: Atlas, 2011.
- FERREIRA, J. M. **Teste de Linha de Produto de Software baseado em Mutação do Diagrama de Características**. 2012. Dissertação (Programa de Pós-Graduação em Informática), Setor de Ciências Exatas, Universidade Federal do Paraná, Curitiba.
- MARTINS, J. C. C. **Técnicas para gerenciamento de projetos de software**. Rio de Janeiro: Brasport, 2007.
- MATTIOLI, F. E. R.; LAMOUNIER JÚNIOR, E. A.; CARDOSO, A.; ALVES, N. M. M. Uma Proposta para o Desenvolvimento Ágil de Ambientes Virtuais. In: WORKSHOP DE REALIDADE VIRTUAL E AUMENTADA, 6., 2009, Santos. **Anais** [...]. Santos: WRVA, 2009.
- NUNES, I. O. de. **Linha de Produtos de Software** – Projeto de Sistemas de Software. Rio de Janeiro: PUC-Rio, 2013.
- PRESSMAN, R. S. **Engenharia de Software**. Uma abordagem profissional. 7. ed. Porto Alegre: AMGH, 2011.
- PRESSMAN, R. S.; MAXIM, B. **Engenharia de Software**. Uma abordagem profissional. 8. ed. Porto Alegre: McGraw Hill Brasil, 2016.
- PHAM, A.; PHAM, P. **Scrum em Ação**: Gerenciamento e Desenvolvimento Ágil de Projeto de Software. São Paulo: Novatec, 2011.
- SAES, D. X.; FREITAS, J. A. de; FLORINDO, R. A.; VANZO, R. M. da; GASPAROTTI, T. T. **Tópicos Especiais**. Maringá: Unicesumar, 2018.
- SBROCCO, J. H. T. C.; MACEDO, P. C. de. **Metodologias Ágeis**: Engenharia de Software sob medida. São Paulo: Érica, 2012.
- SILVA, D. E. dos S.; SOUZA, I. T. de.; CAMARGO, T. Metodologias ágeis para o desenvolvimento de software: aplicação e o uso da Metodologia SCRUM em contraste ao modelo tradicional de gerenciamento de projetos. **Revista Computação Aplicada**, Rio de Janeiro, v. 2, n. 1, 2013.
- SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.
- TSUI, F.; KARAM, O. **Fundamentos de engenharia de software**. 2. ed. Rio de Janeiro: LTC, 2013.



REFERÊNCIAS

REFERÊNCIA ON-LINE:

- 1 Em: <https://tableless.com.br/desenvolvimento-agil-utilizando-scrum/>. Acesso em: 13 jun. 2019.



GABARITO

1. A alternativa correta é a C.
2. A alternativa correta é a D.
3. A alternativa correta é a B.
4. A alternativa correta é a A.
5. A alternativa correta é a B.



REQUISITOS DE SOFTWARE

UNIDADE



Objetivos de Aprendizagem

- Entender os diversos tipos de requisitos relacionados ao desenvolvimento de software.
- Expor a importância do documento de requisitos.
- Compreender o processo de engenharia de requisitos.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Requisitos de Software
- Documento de Requisitos
- Engenharia de Requisitos

INTRODUÇÃO

Caro(a) aluno(a), na primeira unidade, você aprendeu os conceitos relacionados ao processo de software e constatou que ele é composto por quatro atividades fundamentais: especificação de software, projeto e implementação, validação e, finalmente, evolução.

Esta unidade vai tratar especificamente sobre os requisitos de software e, em seu término, você vai compreender os motivos pelos quais os requisitos são importantes e devem ser muito bem definidos para que o software desenvolvido alcance seus objetivos.

Uma das tarefas mais difíceis que os desenvolvedores de software enfrentam é a de entenderem os requisitos de um problema. Os requisitos definirão o que o sistema deve fazer, suas propriedades emergentes desejáveis e essenciais, bem como as restrições quanto à operação do sistema. Essa definição de requisitos somente é possível com a comunicação entre os clientes, os usuários e os desenvolvedores de software.

Nesta unidade, você aprenderá a diferença entre os vários tipos de requisitos e trataremos, principalmente, dos requisitos funcionais e não funcionais. Os requisitos funcionais representam as descrições das diversas funções que clientes e usuários querem ou precisam que o software ofereça. Um exemplo de requisito funcional é a possibilidade de cadastramento dos dados pessoais dos pacientes. Já os requisitos não funcionais declaram as restrições ou atributos de qualidade para um software, como precisão, manutenibilidade, usabilidade e entre outros, por exemplo.

Também estudaremos, nesta unidade, os requisitos de qualidade que são definidos pela Norma ISO/IEC 9126 e devem ser considerados quando um software está sendo projetado. Por fim, certificaremos que a engenharia de requisitos é um processo que envolve quatro atividades genéricas, a saber: i) avaliar se o sistema que está sendo projetado será útil para a empresa (estudo de viabilidade); ii) obter e analisar os requisitos (levantamento e análise); iii) especificar esses requisitos, convertendo-os em um documento de requisitos (especificação de requisitos); e iv) verificar se os requisitos realmente definem o sistema que o cliente deseja (validação).



REQUISITOS DE SOFTWARE

Normalmente, os problemas que os desenvolvedores de software têm para solucionar são imensamente complexos e, se o sistema for novo, entender a natureza desses problemas pode ser muito mais difícil ainda. As descrições das funções e das restrições são os requisitos para o sistema, enquanto o processo de descobrir, analisar, documentar e verificar essas funções e restrições é chamado de engenharia de requisitos.

De acordo com Sommerville (2011), a indústria de software não utiliza o termo requisito de modo consistente. Muitas vezes, o requisito é visto como uma declaração abstrata, em alto nível, de uma função que o sistema deve fornecer ou de uma restrição do sistema. Em outras, ele é uma definição detalhada e formal de uma função do sistema.

Dentre alguns problemas que surgem durante a especificação de requisitos, estão as falhas por não se fazer uma separação clara entre os diferentes níveis de descrição e os dos requisitos. Por isso, Sommerville (2011) propõe uma distinção entre esses níveis por meio do uso do termo “requisitos de usuário”, para expressar os requisitos abstratos de alto nível, e “requisitos de sistema”, para expressar a descrição detalhada que o sistema deve fazer.

Dessa forma, os requisitos de usuário deverão fornecer, em forma de declarações, quais serviços o sistema deverá oferecer e as restrições com as quais deve operar. Já os requisitos de sistema são as descrições mais detalhadas sobre as funções, serviços e restrições operacionais do sistema.

Caro(a) aluno(a), se a sua empresa deseja estabelecer um contrato para o desenvolvimento de um grande sistema, ela deve definir todas as necessidades/requisitos de maneira suficientemente abstrata, para que uma solução não seja pré-definida. Em outras palavras, essas necessidades devem ser redigidas, de modo que os diversos fornecedores possam apresentar propostas, oferecendo, talvez, diferentes maneiras de atender às necessidades organizacionais da sua empresa.

Uma vez estabelecido um contrato, o fornecedor precisa preparar uma definição de sistema para o cliente, com mais detalhes, de modo que a clientela compreenda e possa validar o que o software fará. Esses dois documentos podem ser chamados de documentos de requisitos do sistema. Veremos, mais adiante, o documento de requisitos com mais detalhes.

Contudo, o que pode acontecer se os requisitos não forem definidos corretamente, ou seja, se ficarem errados? Se isso acontecer, o sistema pode não ser entregue no prazo combinado e com o custo acima do esperado no início do projeto. Por conseguinte, o usuário final e o cliente não ficarão satisfeitos com o sistema e isso pode até implicar em seu descarte. Portanto, o ideal é que essa etapa seja muito bem elaborada.



REFLITA

A parte mais difícil ao construir um sistema de software é decidir o que construir. Nenhuma parte do trabalho afeta tanto o sistema resultante se for feita a coisa errada. Nenhuma outra parte é mais difícil de consertar depois.

(Frederick Phillips Brooks Jr.)

REQUISITOS FUNCIONAIS E NÃO FUNCIONAIS

Primeiramente, vamos definir o que é requisito, independentemente da área de informática. Um requisito é uma condição imprescindível para a aquisição ou preenchimento de determinado objetivo. Na abordagem da engenharia de software, segundo Sommerville (2011, p. 57), “os requisitos de um sistema são as descrições do que o sistema deve fazer, os serviços que oferece e as restrições a seu funcionamento”.

Esses requisitos dizem respeito as necessidades dos usuários para um sistema que deve atender um determinado objetivo, como cadastrar um pedido de venda ou emitir um relatório, por exemplo. Assim, a engenharia de requisitos é um processo que engloba as atividades que são necessárias para criar e manter um documento de requisitos de sistema. Essas atividades são: estudo de viabilidade, levantamento e análise de requisitos, especificação de requisitos e, finalmente, a validação desses requisitos.

De acordo com Sommerville (2011), os requisitos de software são, normalmente, classificados em funcionais ou não funcionais:

- 1. Requisitos funcionais:** definem as funções que o sistema deve fornecer, sobre como o sistema deve reagir a entradas específicas e sobre como se comportar em determinadas situações. Em alguns casos, os requisitos funcionais podem, também, explicitamente, declarar o que o sistema não deve fazer. Exemplos de requisitos funcionais: o software deve possibilitar o cálculo das comissões dos vendedores de acordo com os produtos vendidos; o software deve emitir relatórios de compras e vendas por período; o sistema deve mostrar, para cada aluno, as disciplinas em que foi aprovado ou reprovado.
- 2. Requisitos não funcionais:** são os requisitos relacionados com a utilização do software em termos de desempenho, confiabilidade, segurança, usabilidade e portabilidade, entre outros. Exemplos de requisitos não funcionais: o sistema deve ser protegido para acesso apenas de usuários autorizados; o tempo de resposta do sistema não deve ultrapassar 20 segundos; o tempo de desenvolvimento não deve ultrapassar doze meses.

Contudo, a diferenciação entre esses dois tipos de requisitos não é tão clara como sugerem as definições apresentadas. Um requisito referente à proteção pode parecer um requisito não funcional, mas, quando desenvolvido com mais detalhes, pode levar a outros requisitos que são claramente funcionais, como a necessidade de incluir recursos de autorização de usuários no sistema, por exemplo (SOMMERVILLE, 2011).

Portanto, embora seja interessante dividir os requisitos em funcionais e não funcionais, devemos lembrar que essa é, na verdade, uma distinção artificial. O mais importante é que os requisitos, sejam eles funcionais ou não funcionais, sejam claramente definidos.

REQUISITOS FUNCIONAIS

Os requisitos funcionais devem descrever detalhadamente os serviços e a funcionalidade que deve ser fornecida pelo sistema, indicando suas entradas e saídas, exceções etc. Esses requisitos podem ser expressos de diversas maneiras, com diferentes níveis de detalhes. A imprecisão na especificação de requisitos é uma das causas de muitos problemas da engenharia de software (SOMMERVILLE, 2011).

Há a possibilidade de que um desenvolvedor de sistemas interprete um requisito ambíguo para simplificar sua implementação, mas nem sempre é isso o que o cliente quer. Quando isso acontece, pode ser que novos requisitos devam ser estabelecidos, tornando necessário a realização de mudanças no sistema, o que pode atrasar a entrega final do sistema e, consequentemente, gerar aumento de custos.

De acordo com Sommerville (2011, p. 60), “em princípio, a especificação de requisitos funcionais de um sistema deve ser completa e consistente”. A completeza denota que todas as funções requeridas pelo usuário devem estar definidas, enquanto a consistência denota que os requisitos não devem ter definições contraditórias. Na prática, para grandes sistemas, atingir a consistência e a completeza dos requisitos é bastante difícil, por causa da complexidade inerente ao sistema e, em parte, porque diferentes pontos de vista apresentam necessidades inconsistentes.

REQUISITOS NÃO FUNCIONAIS

Os requisitos não funcionais são aqueles que não dizem respeito diretamente às funções específicas oferecidas pelo sistema. Eles podem estar relacionados a propriedades, tais como confiabilidade, tempo de resposta e espaço em disco, por exemplo. Como alternativa, eles também podem definir restrições para o sistema, como a capacidade dos dispositivos de E/S (entrada/saída) e as representações de dados utilizadas nas interfaces de sistema (SOMMERVILLE, 2011).

Os requisitos não funcionais surgem conforme a necessidade dos usuários, em razão de restrições de orçamento, de políticas organizacionais, pela necessidade de interoperabilidade com outros sistemas de software ou hardware ou devido a fatores externos, como regulamentos de segurança e legislação sobre privacidade, por exemplo.

Sommerville (2011) faz uma classificação dos requisitos não funcionais em: i) requisitos de produto; ii) requisitos organizacionais; e iii) requisitos externos. Os requisitos de produto são aqueles que especificam o comportamento do produto e podem ser subdivididos em requisitos de usabilidade, de eficiência, de confiança e de proteção. Já os requisitos organizacionais são os derivados das políticas e dos procedimentos da organização do cliente e do desenvolvedor. Além disso, são subdivididos em requisitos ambientais, operacionais e de desenvolvimento. Finalmente, os requisitos externos abrangem todos os requisitos que procedem de fatores externos ao sistema e seu processo de desenvolvimento, e são subdivididos em requisitos reguladores, éticos e legais.

Os requisitos funcionais e não funcionais deveriam ser diferenciados em um documento de requisitos, mas, na prática, não é fácil fazer essa distinção. Em nossos documentos de requisitos, preocuparemos-nos mais com os requisitos funcionais do sistema. Se os requisitos não funcionais foram definidos separadamente dos requisitos funcionais, pode ser difícil enxergar a relação existente entre eles. Se eles forem definidos com os requisitos funcionais, poderá ser difícil separar as considerações funcionais das não funcionais e identificar os requisitos que correspondem ao sistema como um todo. Assim, é preciso encontrar um equilíbrio adequado que dependerá do tipo de sistema que está sendo modelado. Contudo, requisitos claramente relacionados com as propriedades emergentes

do sistema devem ser explicitamente destacados. Isso pode ser feito colocando-os em uma seção separada do documento de requisito ou diferenciando-os, de alguma maneira, dos outros requisitos de sistema.

REQUISITOS DE USUÁRIO

De acordo com Sommerville (2011), os requisitos de usuários para um sistema devem descrever os requisitos funcionais e não funcionais, de forma que os usuários do sistema que não tenham conhecimentos técnicos detalhados consigam entender. Eles devem especificar somente o comportamento externo do sistema, evitando, sempre que possível, as características do projeto de sistema. Portanto, os requisitos não devem ser definidos mediante a utilização de um modelo de implementação, mas escritos com o uso de linguagem natural, formulários e diagramas intuitivos simples.

REQUISITOS DE SISTEMA

Os requisitos de sistema são as descrições mais detalhadas dos requisitos do usuário, as quais servem como base para um contrato destinado à implementação do sistema e, portanto, devem ser uma especificação completa e consistente de todo o sistema (SOMMERVILLE, 2011). Eles são utilizados pelos engenheiros de software como um ponto de partida para o projeto de sistema.

Antes de qualquer coisa, os requisitos de sistema deveriam definir o que o sistema deveria fazer, e não como ele teria de ser implementado. Todavia, no que se refere aos detalhes exigidos para especificar o sistema completamente, é quase impossível excluir todas as informações de projeto. Há, pelo menos, duas razões para isso:

1. Uma arquitetura inicial do sistema pode ser definida para ajudar a estruturar a especificação de requisitos.

2. Na maioria dos casos, os sistemas devem interoperar com outros sistemas existentes, restringindo, assim, o projeto em desenvolvimento e, muitas vezes, essas restrições geram requisitos para o novo sistema.

De acordo com Sommerville (2011), os requisitos devem ser escritos em níveis diferentes de detalhamento, para que diferentes leitores possam usá-los de formas distintas. Os possíveis leitores para os requisitos de usuário são: os gerentes clientes, os usuários finais do sistema, os engenheiros clientes, os gerentes contratantes e os arquitetos de software. Esses leitores não têm a preocupação com a forma como o sistema será implementado. Já para os requisitos de sistema, podem haver os seguintes leitores: os usuários finais do sistema, os engenheiros clientes, os arquitetos de sistema e os desenvolvedores de software. Esses leitores precisam saber com mais detalhes o que o sistema fará, principalmente os desenvolvedores que estarão envolvidos no projeto e na implementação do sistema.



REFLITA

As sementes das principais catástrofes de software são normalmente semeadas nos três primeiros meses do projeto de software.

(Capers Jones)



DOCUMENTO DE REQUISITOS

O documento de requisitos de software ou especificação de requisitos de software é a declaração oficial do que é exigido dos desenvolvedores de sistema. Ele deve incluir os requisitos de usuários para um sistema e uma especificação detalhada dos requisitos de sistema. Em alguns casos, os requisitos de usuário e de sistema podem ser integrados em uma única descrição. Em outros, os requisitos de usuário são definidos em uma introdução para a especificação dos requisitos de sistema. Se houver um grande número de requisitos, os requisitos detalhados de sistema poderão ser apresentados como documentos separados.

O documento de requisitos serve como um termo de consenso entre a equipe técnica (desenvolvedores) e o cliente, bem como constitui a base para as atividades subsequentes do desenvolvimento do sistema, fornecendo um ponto de referência para qualquer validação futura do software construído. Além disso, estabelece o escopo (o que faz parte e o que não faz) do sistema, abrangendo um conjunto diversificado de usuários, o qual vai desde a alta gerência da organização, que está pagando pelo sistema, até os engenheiros responsáveis pelo desenvolvimento do software.

A Tabela 1 mostra uma possível organização de um documento de requisitos, a qual foi definida por Sommerville (2011), com base em uma norma IEEE (*Institute of Electrical and Electronics Engineers*) para documentos de requisitos:

Tabela 1 – A estrutura de um documento de requisitos

CAPÍTULO	DESCRIÇÃO
Prefácio	Deve definir os possíveis leitores do documento e descrever seu histórico de versões, incluindo uma justificativa para a criação de uma nova versão e um resumo das mudanças feitas em cada versão.
Introdução	Deve descrever a necessidade para o sistema. Deve descrever brevemente as funções do sistema e explicar como ele vai funcionar com outros sistemas. Também deve descrever como o sistema atende aos objetivos globais de negócio ou estratégicos da organização que encomendou o software.
Glossário	Deve definir os termos técnicos usados no documentos. Não se deve fazer suposições sobre a experiência ou o conhecimento do leitor.
Definição de requisitos de usuário	Deve descrever os serviços fornecidos ao usuário. Os requisitos não funcionais de sistema também devem ser descritos nessa seção. Essa descrição pode usar a linguagem natural, diagramas ou outras notações compreensíveis para os clientes. Normas de produto e processos a serem seguidos devem ser especificados.
Arquitetura do sistema	Deve apresentar uma visão geral em alto nível da arquitetura do sistema previsto, mostrando a distribuição de funções entre os módulos do sistema. Componentes de arquitetura que são reusados devem ser destacados.
Especificação de requisitos do sistema	Deve descrever em detalhes os requisitos funcionais e não funcionais. Se necessário, também podem ser adicionados mais detalhes aos requisitos não funcionais. Interface com outros sistemas podem ser definidas.
Modelos do sistema	Pode incluir modelos gráficos do sistema que mostram os relacionamentos entre os componentes do sistema, o sistema e seu ambiente. Exemplos de possíveis modelos são modelos de objetos, modelos de fluxo de dados ou modelo semânticos de dados.
Evolução do sistema	Deve descrever os pressupostos fundamentais em que o sistema se baseia, bem como quaisquer mudanças previstas, em decorrência da evolução do hardware, de mudanças nas necessidades do usuário etc. Essa seção é útil para projetistas de sistema, pois pode ajudá-los a evitar decisões capazes de restringir possíveis mudanças futuras no sistema.
Apêndices	Deve fornecer informações detalhadas e específicas relacionadas à aplicação em desenvolvimento, além de descrições de hardware e banco de dados, por exemplo. Os requisitos de hardware definem as configurações mínimas ideais para o sistema. Requisitos de banco de dados definem a organização lógica dos dados usados pelo sistema e os relacionamentos entre esses dados.
Índice	Vários índices podem ser incluídos no documento. Pode haver, além de um índice alfabético normal, um índice de diagramas, de funções, entre outros pertinentes.

Fonte: Sommerville (2011, p. 64).

Para o desenvolvimento da nossa disciplina, serão utilizados modelos de documento de requisitos mais simplificados do que o apresentado na tabela. O documento de requisitos trará detalhes de como o sistema funciona atualmente e quais funcionalidades o usuário deseja para o novo sistema. A seguir, é exposto um modelo de documento de requisitos para uma locadora de filmes. Lembre-se que deve ser a partir do documento de requisitos que faremos a modelagem do sistema, a qual será detalhada na próxima unidade.

EXEMPLO DE DOCUMENTO DE REQUISITOS – LOCADORA DE FILMES

Uma determinada locadora possui muitos títulos em seu acervo e não consegue controlar, de maneira eficaz, as locações, devoluções e reservas dos filmes. Portanto, ela deseja ter um sistema informatizado, o qual controle todos esses itens de maneira eficiente, a fim de aperfeiçoar o seu atendimento com o cliente e seu controle interno.

Atualmente, a locadora possui uma ficha para o cadastro de clientes com os seguintes dados: nome do cliente, fone residencial, fone celular, sexo, RG, CPF, endereço completo, data de nascimento, estado civil, nomes de cinco dependentes e o grau de parentesco de cada um (o dependente pode locar filmes em nome do cliente).

Assim, o sistema informatizado deve:

1. Manter o cadastro de filmes: neste cadastro, deverão constar os seguintes dados: nome do filme, duração, sinopse, classificação, gênero, diretor e elenco. Além disso, para cada cópia do filme, é necessário saber o fornecedor, a data da compra, o valor pago e o tipo (VHS ou DVD).

2. Controlar locações:

- A locação é feita mediante a verificação de cadastro do cliente. Se o cliente for cadastrado, então, efetua-se a locação. Caso não seja, é feito o cadastro do cliente.
- Caso a locação seja efetuada pelo dependente do cliente, é necessário deixar registrado qual é o dependente e qual é o cliente.
- É verificado se o filme está disponível e se o cliente possui pendências financeiras ou atraso de devolução. Caso uma das alternativas seja

afirmativa, bloqueia-se a operação, sendo liberada somente após a devida regularização.

- Deve-se emitir comprovante de locação com a data prevista para a devolução de cada filme, discriminação dos filmes e se o pagamento foi, ou não, efetuado.
- A data prevista para devolução deve ser calculada desconsiderando domingos e feriados. Cada categoria pode ter um prazo diferente para que o cliente possa ficar com o filme. Por exemplo: a categoria lançamento permite que o cliente fique com o filme por dois dias.

3. Controlar devoluções:

- É necessário verificar se a devolução está no prazo correto e se o pagamento foi efetuado. Caso o prazo esteja vencido, é preciso calcular a multa incidente. Efetuado o pagamento, emite-se o recibo de devolução.
- Não é permitido esquecer que não pode ser cobrada multa, caso seja domingo ou feriado.

4. Controlar reservas:

- Verificar se o cliente já está cadastrado. Caso contrário, o sistema permitirá o cadastro do cliente no momento da reserva. Também, é verificado se o filme desejado está disponível para reserva.
- Reservar somente para clientes sem pendências financeiras e devoluções vencidas.

5. Consultar filmes locados por cliente: o sistema deve apresentar uma consulta em que seja exposto um determinado cliente e sejam mostrados todos os filmes já locados por esse sujeito, bem como a data em que cada filme foi locado.

6. Consultar reservas por filme: o sistema deve ter uma consulta que informe um determinado filme e sejam mostradas todas as reservas efetuadas para ele no período informado.

7. Emitir os seguintes relatórios:

- **Relatório geral de clientes**, em que sejam expostos o código, nome, endereço, telefone e dependentes do cliente.

- **Etiquetas com códigos de barras**, para a identificação das cópias no processo de locação e devolução.
- **Relatório de filmes por gênero**, em que conste o código e o nome do filme, o nome do diretor e dos atores, o total de cópias, bem como o total de cópias locadas e disponíveis.
- **O relatório deve ser agrupado por gênero**, mostrando, também, o código e a descrição do gênero.
- **Relatório de filmes locados por cliente por período**. Para cada cliente, devem ser emitidas todas as cópias que estão locadas para ele. Além disso, são apresentados no relatório: o código e o nome do cliente, o código e o nome do filme, o código da cópia (exemplar), a data e o valor da locação. O relatório necessita ser agrupado por cliente e devem sair somente as cópias locadas e não devolvidas.
- **Relatório de cópias não devolvidas**, em que conste o código e o nome do filme, o código da fita, o nome e o telefone do cliente, a data de locação, a data prevista para devolução e o número de dias em atraso.
- **Relatório dos filmes mais locados**, em que se exponha o código e o nome do filme, a descrição do gênero e o número total de locações. O relatório deve ser agrupado por mês/ano, ou seja, para um determinado mês/ano, devem ser emitidos os dez filmes mais locados.
- **Relatório de reservas por período**, em que se evidencie o código, o nome e o telefone do cliente, o código e o nome do filme reservado, bem como a data em que foi feita a reserva (data em que o cliente telefonou para a locadora dizendo que queria fazer a reserva).
- **Relatório dos valores das locações mensais**, o qual mostre os valores das locações de determinado mês separados por data e somatória de valores de cada dia, somando-se, assim, ao final, uma totalidade de locações. Nele, deve-se conter a data e a soma das locações feitas nesse dia.

Todos os relatórios servirão para o processo de tomadas de decisões, no qual os administradores poderão obter informações sobre o andamento da locadora.

REQUISITOS DE QUALIDADE

Quanto mais rígidos forem os requisitos de qualidade e mais complexo for o software a ser desenvolvido, aumenta-se a necessidade de se aplicar teorias e ferramentas que garantam que esses requisitos sejam satisfeitos. A Norma ISO (*The International Organization for Standardization*) / IEC (*The International Electrotechnical Commission*) 9126 define seis características de qualidade de software que devem ser avaliadas:

- **Funcionalidade:** capacidade de um software em fornecer funcionalidades que atendam às necessidades explícitas e implícitas dos usuários, dentro de um determinado contexto de uso.
- **Usabilidade:** conjunto de atributos que evidenciam o esforço necessário para a utilização do software.
- **Confiabilidade:** indica a capacidade do software em manter seu nível de desempenho sob determinadas condições durante um período de tempo estabelecido.
- **Eficiência:** indica que o tempo de execução e os recursos envolvidos são compatíveis com o nível de desempenho do software.
- **Manutenibilidade:** conjunto de atributos que evidenciam o esforço necessário para fazer modificações especificadas no software, incluindo tanto as melhorias/extensões de funcionalidades quanto as correções de defeitos, falhas ou erros.
- **Portabilidade:** indica a capacidade do software em ser transferido de um ambiente para outro.

A ISO1 e a IEC2 formam o sistema especializado para padronização mais conhecido no mundo.



ENGENHARIA DE REQUISITOS

Caro(a) aluno(a), assim como foi exposto na unidade anterior, a engenharia de requisitos é um processo que envolve todas as atividades necessárias para a criação e manutenção de um documento de requisitos de software. Para tanto, existem quatro atividades genéricas do processo de engenharia de requisitos que são de alto nível: (i) o estudo de viabilidade do sistema; (ii) o levantamento e análise de requisitos; (iii) a especificação de requisitos e sua documentação; e (iv) a validação desses requisitos.

A seguir, abordaremos todas as atividades, com exceção da especificação de requisitos, que já foi discutida nesta unidade. A figura seguinte ilustra a relação entre essas atividades e mostra, também, os documentos produzidos em cada estágio do processo de engenharia de requisitos, de acordo com Sommerville (2011):

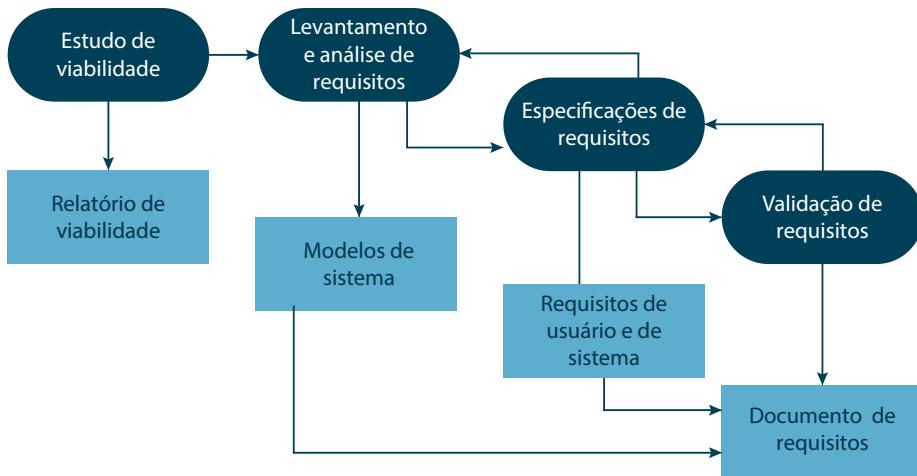


Figura 1 – Processo de engenharia de requisitos

Fonte: Sommerville (2011, p. 24).

As atividades de engenharia de requisitos mostradas na Figura 1 dizem respeito ao levantamento, documentação e verificação dos requisitos. Entretanto, é necessário deixar claro que, em praticamente em todos os sistemas: i) os requisitos se modificam; ii) as pessoas interessadas desenvolvem uma melhor compreensão do que elas querem que o software faça; iii) a organização compradora do sistema sofre modificações; e iv) são feitas alterações no hardware, no software e no ambiente organizacional do sistema (SOMMERVILLE, 2011).



Requisito não é sinônimo de arquitetura. Requisito não é sinônimo de projeto nem de interface do usuário. Requisito é sinônimo de necessidade.

(Andrew Hunt e David Thomas)

ESTUDO DE VIABILIDADE

Segundo Sommerville (2011), para todos os sistemas novos, o processo de engenharia de requisitos de sistemas deve ser iniciado com um estudo de viabilidade ou licitação de requisitos. Tal estudo se principia com uma descrição geral do sistema e de como ele será utilizado dentro de uma organização. O resultado desse estudo deve ser um relatório que recomende a viabilidade, ou não, do processo de realização do processo de engenharia de requisitos e, consequentemente, do processo de desenvolvimento de sistemas.

Um estudo de viabilidade é rápido, direcionado e destinado a responder a algumas perguntas, a saber:

1. O sistema contribui para os objetivos gerais da organização?
2. O sistema pode ser implementado com a utilização da tecnologia atual dentro das restrições de custo e de prazo?
3. O sistema pode ser integrado com os outros sistemas já em operação?

Questionar-se se o sistema contribui, ou não, para os objetivos da empresa é fundamental, pois, se um sistema não for compatível com esses objetivos, ele não terá nenhum valor real. Embora isso possa parecer óbvio, muitas organizações desenvolvem sistemas que não contribuem para suas metas, seja porque não existe uma declaração clara acerca desses objetivos ou porque outros fatores políticos ou organizacionais influenciam na aquisição do sistema (SOMMERVILLE, 2011).

Preparar um estudo de viabilidade envolve avaliar e coletar informações para redigir relatórios. A fase de avaliação identifica as informações exigidas para responder as três perguntas apresentadas. Uma vez identificadas as informações, é preciso questionar as fontes de informação, a fim de se encontrar as respostas para essas perguntas. Eis alguns exemplos das possíveis perguntas que devem ser feitas:

1. Como a organização se comportaria se esse sistema não fosse implementado?
2. Quais são os problemas com os processos atuais e como um novo sistema ajudaria a diminuir esses problemas?
3. Que contribuição direta o sistema trará para os objetivos da empresa?

4. As informações podem ser transferidas para outros sistemas organizacionais e podem ser recebidas a partir deles?
5. O sistema requer alguma tecnologia que já não tenha sido utilizada na organização?
6. O que precisa e o que não precisa ser compatível com o sistema?

Entre as fontes de informação, estão os gerentes de departamentos em que o sistema será utilizado, os engenheiros de software que estão familiarizados com o tipo de sistema proposto, os peritos em tecnologia, os usuários finais de sistema, entre outros. Eles devem ser entrevistados durante o estudo de viabilidade, a fim de se coletar as informações exigidas.

O relatório do estudo de viabilidade deverá ser elaborado com base nas informações mencionadas e deve recomendar ao desenvolvimento do sistema se este deve ou não continuar. Ademais, se este deve ou não continuar no orçamento e no cronograma, além de sugerir outros requisitos de alto nível para o sistema.

LEVANTAMENTO E ANÁLISE DE REQUISITOS

De acordo com Sommerville (2011), após os estudos iniciais de viabilidade, a próxima atividade do processo de engenharia de requisitos é o levantamento e a análise de requisitos. Nessa atividade, os membros da equipe técnica de desenvolvimento de software trabalham com o cliente e os usuários finais do sistema para descobrir mais informações sobre o domínio da aplicação, quais serviços o sistema deve fornecer, o desempenho exigido do sistema, as restrições de hardware e assim por diante.

O levantamento e a análise de requisitos podem envolver diferentes tipos de pessoas em uma organização. O termo **stakeholder** é utilizado para se referir a qualquer pessoa que terá alguma influência, seja direta ou indireta, sobre os requisitos do sistema. Dentre os *stakeholders*, destacam-se os usuários finais que interagiram com o sistema e todo o pessoal, em uma organização, que venha a ser afetado por ele. Os engenheiros que estão desenvolvendo o sistema ou fazendo a manutenção de outros sistemas relacionados, os gerentes de negócios,

os especialistas nesse domínio, os representantes de sindicato e entre outros, podem ser, também, os *stakeholders* do sistema.

O levantamento e a análise de requisitos compõem um processo difícil por diversas razões:

1. Exceto em termos gerais, os *stakeholders* costumam não saber o que querem de um sistema computacional; eles podem achar difícil articular o que querem que o sistema faça, e como não sabem o que é viável e o que não é, podem fazer exigências inviáveis.
2. Naturalmente, os *stakeholders* expressam requisitos em seus próprios termos e com o conhecimento implícito de seu próprio trabalho. Engenheiros de requisitos, sem experiência no domínio do cliente, podem não entender esses requisitos.
3. Diferentes *stakeholders* têm requisitos diferentes e podem expressar essas diferenças de várias maneiras. Engenheiros de requisitos precisam descobrir todas as potenciais fontes de requisitos e descobrir as semelhanças e conflitos.
4. Fatores políticos podem influenciar os requisitos de um sistema. Os gerentes podem exigir requisitos específicos, porque estes lhes permitem aumentar sua influência na organização.
5. O ambiente econômico e empresarial no qual a análise ocorre é dinâmico. É inevitável que ocorram mudanças durante o processo de análise. A importância dos requisitos específicos pode mudar e novos requisitos podem surgir a partir de novos *stakeholders* que não foram inicialmente consultados (SOMMERVILLE, 2011, p. 71).

SAIBA MAIS



O artigo “Introdução à Engenharia de Requisitos”, de Ávila e Spínola, é muito interessante, pois trata sobre a engenharia de requisitos, a qual é um dos mais importantes conjuntos de atividades a serem realizados em projetos de desenvolvimento de software. Embora não garanta a qualidade dos produtos gerados, é um pré-requisito básico para que obtenhamos sucesso no desenvolvimento do projeto. Para saber mais, acesse o link disponível em: <https://ebrito.com.br/profa-elaine/requisitos.pdf>.

Fonte: o autor.

ENTREVISTA

As entrevistas formais ou informais com os *stakeholders* do sistema fazem parte da maioria dos processos de engenharia de requisitos. Elas são a fonte mais importante para o levantamento dos requisitos, desde que o entrevistado confie no entrevistador. A sumarização durante e no final da entrevista é necessária, primeiro, para garantir que toda informação apresentada foi anotada e, segundo, que foi corretamente entendida.

Antes de tentar uma entrevista, o engenheiro de software deve prepará-la. A seguir, são apresentadas algumas dicas a serem seguidas para que haja tal preparação:

- a. **Comece por definir os objetivos:** verifique a documentação formal e desenvolva um esquema do sistema existente ou proposto. Identifique questões, partes omitidas e ambíguas. Esses fatos ou componentes desconhecidos representam um esboço inicial dos objetivos. Pode ser necessária a entrevista de várias pessoas para atingir o objetivo.
- b. **Selecionar a pessoa ou grupo a ser entrevistado:** é claro que você quer encontrar a pessoa que melhor possa responder sobre o assunto. Você pode encontrá-la utilizando o organograma, uma análise do fluxo de trabalho ou uma lista de distribuição de relatórios. Comece pelo organograma e pelo gerente que parece ser o melhor para responder às questões. Além disso, as pessoas ficam menos hesitantes se souberem que a entrevista foi autorizada pelo chefe.
- c. **Ler a documentação relevante:** conheça a posição e as responsabilidades do entrevistado, bem como os procedimentos e os documentos relevantes.
- d. **Preparar questões específicas:** selecione questões específicas que podem ser respondidas. Desenvolva uma lista de perguntas a serem seguidas, caso a entrevista comece a se desviar do ponto-chave.

A entrevista deve ser marcada com antecedência, o horário deve ser combinado e as questões devem ser preparadas. Além disso, ela é composta por três partes: a abertura, o corpo e o fechamento:

1. **Abertura:** o objetivo-chave é estabelecer harmonia (concordância). Comece se identificando, apresentando o tópico que pretende discutir

e o propósito (objetivo) da entrevista. Se houver necessidade, “quebre o gelo” com conversas informais, mas não caia na “perda de tempo”.

2. **Corpo:** pode ser iniciado com uma questão relativamente aberta e, gradualmente, seguir para questões específicas. É um exemplo de pergunta aberta: “Quando eu li a documentação para esse sistema, tive trabalho com (anuncie a parte ou seção). Você poderia me explicar?”.

Além disso, nesta fase, o engenheiro de software deve:

- a. Mostrar que conhece as responsabilidades e deveres do trabalho do entrevistado. Exemplo: “isso é o que eu entendo do seu trabalho (apresentar uma breve descrição). Está correto?”.
 - b. Procurar saber as decisões que o entrevistado toma (quais são e como ele toma as decisões; quais são as informações necessárias e se, da forma como são apresentadas, são satisfatórias; qual é o tempo necessário para que se possa tomar as decisões).
 - c. Procurar respostas quantitativas. Exemplo: “quantos telefones, você, funcionário, tem no departamento?”.
 - d. Evitar falar palavras sem sentido, falar baixo ou fazer generalizações.
 - e. Ouvir as respostas. Dê um tempo para o entrevistado responder, não saia com respostas antecipadas. Não se concentre na próxima questão (é um erro comum dos iniciantes). A lista de questões preparadas é apenas um guia. Tenha certeza de que as questões são relevantes e evite perguntas complexas e desnecessárias.
 - f. Pedir explicações para as questões que ficarem obscuras.
 - g. Pedir ideias e sugestões e descobrir se o entrevistado quer que sejam consideradas. Exemplo: “você tem alguma sugestão ou recomendações relativas ao método para calcular o orçamento? Você gostaria que os seus superiores ou os demais ficassem sabendo de suas sugestões?”.
3. **Encerramento:** se a entrevista tiver consumido mais tempo do que o previsto, peça para continuar e ofereça uma reprogramação. Quando tiver todas as informações necessárias, agradeça e faça um sumário de todos os pontos principais. Avise, caso seja necessária outra sessão de entrevista com a mesma pessoa.

Muitas vezes, algumas expressões corporais podem substituir ou comunicar mais informações do que as próprias palavras. Esse tipo de comunicação pode ajudar o engenheiro de software a:

- a. Interpretar as palavras do entrevistado.
- b. Determinar a atitude geral do entrevistado para as questões que estão sendo discutidas.
- c. Avaliar a confidência de que o entrevistado demonstrou tanto ao seu redor quanto no tratamento da área de abrangência do sistema.

Além disso, vários pontos devem ser aprendidos e esclarecidos na entrevista:

- a. Organização da empresa (ambiente de trabalho): como o administrador organiza o seu pessoal? Como essa organização se relaciona às funções maiores que a empresa executa?
- b. Os objetivos e exigências do sistema (declarados nos manuais de procedimentos) devem ser reafirmados e esclarecidos na entrevista: muitas vezes, os objetivos e exigências declarados nos manuais não são os mesmos que os representantes veem. Quando existe uma discrepância, é possível que as metas representadas nos documentos possam ser irreais com o atual potencial humano. Isso se deve, pois o tempo e o crescimento podem ter alterado a meta declarada.
- c. Fluxo funcional: para cada função importante, é necessário determinar as etapas exigidas e descrever o significado delas.
- d. Exigência de recursos: determinar quais são os recursos aplicados pela organização para executar o trabalho. Além disso, é preciso saber quais são as exigências com:
 - Recursos humanos (treinamento especializado, experiência exigida).
 - Equipamento e material necessário para apoiar na execução do trabalho.
- e. Relação de tempo: como o trabalho executado se relaciona com períodos específicos do ano ou outros ciclos comerciais. Existe pico? Qual é o atual volume de trabalho?
- f. Formulários, procedimentos e relatórios: quais são utilizados? Inclua um exemplo de cada formulário, relatório e procedimento. Para tanto:
 - Verifique se o material tem origem no escritório, se é modificado pelo escritório e/ou transmitido para outro setor.
 - Faça comparações que determinam se é inutilizado, duplicado ou incompleto.
 - Verifique a satisfação dos usuários com esses documentos.

- g. Funções desejáveis e não existentes: registre a opinião das pessoas sobre o sistema, como ele existe e como poderia ser. Atribua atenção para as opiniões mais subjetivas do que as objetivas.
- h. Flexibilidade dos procedimentos: o sistema atual é tão rígido e inflexível que a menor modificação requer o maior remendo?
- i. Capacidade: o sistema atual consegue manipular volumes maiores do que aqueles que resultam do crescimento normal?



REFLITA

Coloque três interessados em uma sala e pergunte a eles que tipo de sistema desejam. Provavelmente você obterá quatro ou mais opiniões diferentes.

ESPECIFICAÇÃO DE REQUISITOS

Durante o levantamento de requisitos (levantamento de dados), a equipe de desenvolvimento tenta entender o domínio (contexto/problema) que deve ser automatizado. O produto do levantamento de requisitos é o documento de requisitos ou especificação de requisitos, o qual declara os diversos tipos de requisitos do sistema (requisitos funcionais, requisitos não funcionais, de usuário e de sistema). Já tratamos desse tópico nesta unidade.

VALIDAÇÃO DE REQUISITOS

A validação de requisitos tem, como objetivo, mostrar que os requisitos realmente definem o sistema que o cliente deseja. Ela tem muito em comum com a análise de requisitos, uma vez que se preocupa em descobrir problemas nos requisitos. Contudo, esses são processos distintos, já que a validação deve se ocupar com

a elaboração de um esboço completo do documento de requisitos, enquanto a análise envolve o trabalho com requisitos incompletos (SOMMERVILLE, 2011).

A validação de requisitos é importante, porque a ocorrência de erros em um documento de requisitos pode levar a grandes custos relacionados ao retribalho, quando descobertos durante o desenvolvimento ou depois que o sistema estiver em operação. O custo de fazer uma alteração no sistema, resultante de um problema de requisito, é muito maior do que reparar erros de projeto e de codificação, pois, em geral, significa que o projeto do sistema e a implementação também devem ser modificados e que o sistema tem de ser novamente testado.

Durante a etapa de validação de requisitos, Sommerville (2011) propõe que diferentes tipos de verificação devam ser realizados sobre os requisitos no documento de requisitos. Dentre as verificações, destacam-se:

- 1. Verificações de validade:** um usuário pode considerar que um sistema é necessário para realizar certas funções. No entanto, mais estudos e análises podem identificar funções adicionais ou diferentes as quais são exigidas. Isso se deve, pois os sistemas têm diversos usuários com necessidades diferentes e qualquer conjunto de requisitos é, inevitavelmente, uma solução conciliatória da comunidade de usuários.
- 2. Verificações de consistência:** os requisitos, em um documento, não devem ser conflitantes, ou seja, não se pode existir restrições contraditórias ou descrições diferentes para uma mesma função do sistema.
- 3. Verificações de completeza:** o documento de requisitos deve incluir os requisitos que definam todas as funções e restrições exigidas pelos usuários do sistema.
- 4. Verificações de realismo:** utilizando o conhecimento acerca da tecnologia existente, os requisitos devem ser verificados, a fim de assegurar que eles realmente podem ser implementados, levando-se em consideração o orçamento e os prazos para o desenvolvimento do sistema.
- 5. Facilidade de verificação:** para diminuir as possíveis divergências entre cliente e fornecedor, os requisitos do sistema devem sempre ser escritos de modo que possam ser verificados. Isso implica na definição de um conjunto de verificações para mostrar que o sistema entregue cumpre com esses requisitos.

Algumas técnicas de validação de requisitos podem ser utilizadas em conjunto ou individualmente. A seguir, são mostradas algumas delas:

- 1. Revisões de requisitos:** os requisitos são analisados sistematicamente por uma equipe de revisores, a fim de eliminar erros e inconsistências.
- 2. Prototipação:** nessa abordagem de validação, um modelo executável do sistema é mostrado aos usuários finais e clientes, possibilitando que eles experimentem o modelo para verificar se atende às necessidades da empresa.
- 3. Geração de casos de teste:** como modelo ideal, os requisitos deveriam ser testáveis. Se os testes para os requisitos são criados como parte do processo de validação revelam problemas com os requisitos. Se um teste é difícil ou impossível de ser projetado, significa, muitas vezes, que os requisitos serão de difícil implementação e devem ser reconsiderados. O desenvolvimento de testes a partir dos requisitos de usuário, antes da implementação do sistema, é uma parte integrante da *Extreme Programming*.

As dificuldades da validação de requisitos não devem ser subestimadas, pois é muito difícil demonstrar que um conjunto de requisitos atende às necessidades de um usuário. Os usuários devem pensar no sistema em operação e imaginar como esse sistema se adequaria ao seu trabalho. Não é fácil para profissionais habilitados em computação conseguirem realizar esse tipo de análise abstrata, imagine o quanto difícil é para os usuários de sistema. Dessa forma, a validação de requisitos não consegue descobrir todos os problemas com os requisitos, implicando em modificações para corrigir essas omissões e falhas de compreensão depois que o documento de requisitos foi aceito (SOMMERVILLE, 2011).



REFLITA

Gastamos um bom tempo – a maior parte do esforço de um projeto – não implementando ou testando, mas sim tentando decidir o que construir.

(Brian Michael Lawrence)

CONSIDERAÇÕES FINAIS

Caro(a) aluno(a), chegamos ao fim da segunda unidade, na qual estudamos os requisitos de software. Nesta unidade, foi evidenciado que os requisitos para um software deve estabelecer o que o sistema deve fazer e definir as restrições sobre o seu funcionamento e implementação.

Os requisitos de software podem ser classificados em requisitos funcionais, que são os serviços que o sistema deve fornecer, e em requisitos não funcionais, os quais estão, frequentemente, relacionados com as propriedades emergentes do sistema, aplicando-se ao sistema como um todo.

Todos os requisitos, sejam eles funcionais ou não funcionais, devem ser definidos da forma mais clara possível, a fim de que não haja problemas em sua interpretação. Isso se deve a partir da definição desses requisitos que o sistema será modelado, projetado, implementado, testado e, por fim, colocado em funcionamento.

A primeira atividade servirá para avaliar se o sistema será útil para a empresa. Ela é feita por meio do estudo de viabilidade e, ao seu término, um relatório de viabilidade deve ser elaborado e recomendando se vale a pena, ou não, realizar o processo de engenharia de requisitos.

A segunda atividade do processo de engenharia de requisitos é a conversão dos requisitos levantados em uma forma-padrão, ou seja, em um documento de requisitos de software. Já a terceira e última atividade é a verificação de que os requisitos realmente definem o sistema que o cliente quer, ou seja, a validação dos requisitos anotados no documento de requisitos.

Na próxima unidade, vamos conhecer como se dá o processo de modelagem de um sistema e vamos usar os conceitos de orientação a objetos apresentados na primeira unidade. Isso se deve, pois a UML – *Unified Modeling Language* – que utilizaremos para realizar a modelagem é baseada no paradigma da orientação a objetos.

ATIVIDADES



1. Descreva como os requisitos de software são classificados.
2. Quanto mais rígidos forem os requisitos de qualidade e mais complexo for o software a ser desenvolvido, aumenta-se a necessidade de se aplicar teorias e ferramentas que garantam que esses requisitos sejam satisfatórios. Descreva as seis características de qualidade de software que devem ser avaliadas.
3. Defina e explique os requisitos de usuário.
4. Segundo Sommerville (2011, p. 57), "os requisitos de um sistema são as descrições do que o sistema deve fazer, os serviços que oferece e as restrições a seu funcionamento". É importante enfatizar que esses requisitos dizem respeito às necessidades dos usuários para um sistema que deva atender um determinado objetivo. Baseado no conceito de requisitos, defina e exemplifique os requisitos funcionais.
5. O levantamento e a análise de requisitos podem envolver diferentes tipos de pessoas em uma organização. Baseado nessa afirmação, assinale a alternativa que conte com o significado do termo *stakeholder*:
 - a) O termo *stakeholder* se refere a qualquer pessoa que terá alguma influência, seja direta ou indireta, sobre os requisitos do sistema.
 - b) O termo *stakeholder* se refere a linguagem de desenvolvimento de software.
 - c) O termo *stakeholder* se refere aos engenheiros de software que estão analisando o sistema.
 - d) O termo *stakeholder* se refere ao programador do software.
 - e) As alternativas anteriores não estão corretas.



Os Processos da Engenharia de Requisitos

A Engenharia de Requisitos é a parte da Engenharia de Software que engloba “as atividades envolvidas com a descoberta, documentação e manutenção de um conjunto de requisitos para um sistema informatizado” (KOTONYA; SOMMERVILLE, 1998, p. 8). Embora, usualmente, haja algumas variações na nomenclatura utilizada, os processos da Engenharia de Requisitos usualmente são referenciados (KOTONYA; SOMMERVILLE, 1998), (PRESSMAN, 2000) como levantamento, análise, especificação, validação e gerenciamento de requisitos.

Levantamento é o processo executado para identificar, junto aos *stakeholders*, seus desejos em relação ao produto de software (o problema a ser resolvido, o desempenho desejado do produto de software, restrições de equipamentos etc.). Para executar esse processo, tipicamente, um técnico de software aplica uma técnica de levantamento (entrevista, brainstorming, observação entre outras) e identifica o que os usuários esperam do software.

Análise é o processo em que o técnico de software realiza considerações técnicas e determina quais requisitos são necessários para atender à solicitação apresentada durante o levantamento. Na análise, ocorrem a categorização e a organização dos requisitos encontrados, explorando o relacionamento de cada requisito com os demais, examinando inconsistências, omissões e ambiguidades. Tipicamente, ocorre, ainda, a priorização dos requisitos, com base nas necessidades apontadas pelos solicitantes. De fato, é durante a análise que os requisitos de software são determinados, no sentido que as solicitações dos usuários são abordadas tecnicamente.

Especificação é o processo em que a compreensão/interpretação/racionalização (resultante do processo de análise) da solicitação dos usuários (realizada via levantamento) é formalizada na forma de requisitos por um profissional de software. Além disso, a especificação:

Consiste do detalhamento, estruturação e documentação das características que foram acordadas em relação ao sistema a ser desenvolvido [...] [através de um] documento de requisitos (DR). [...] Uma grande quantidade de técnicas pode ser utilizada para suportar o processo de especificação e documentação, incluindo modelos de linguagem natural estruturada, notações diagramáticas e especificações formais (VAN LAMSWEERDE, 2009, p. 33).

Em relação ao conteúdo especificado, vale destacar que, em uma situação ideal, cada requisito deveria ser “entendido exatamente da mesma forma por qualquer pessoa que o lesse. [...] Para especificar um requisito de modo que ele seja entendido de uma única forma, você deve definir os termos que está usando e o significado desses termos” (ROBERTSON; ROBERTSON, 2007, p. 267). Nesse sentido, parece relevante destacar que “é mais fácil escrever requisitos, e mais conveniente, se os analistas de requisitos têm um guia para redigi-los” (ROBERTSON; ROBERTSON, 2007, p. 11).





Validação é o processo no qual a compreensão da solicitação e a formalização técnica elaborada são submetidas à aprovação do solicitante.

Gerenciamento é um processo executado ao longo dos demais, em que são controladas as alterações solicitadas e são rastreadas as várias abstrações criadas para os requisitos. As alterações devem ser gerenciadas para que elas tenham sentido econômico e contribuam para as necessidades de negócio da organização na qual o software é (ou será) utilizado.

Fonte: adaptado de Marquioni (2011).

MATERIAL COMPLEMENTAR



LIVRO

Engenharia de Software - Análise e Projeto de Sistemas

Sérgio Luiz Tonsig

Editora: Moderna

Sinopse: é indicado para quem deseja aprender sobre o planejamento, análise e projeto de software para sistemas de informação. Este livro ensina, passo a passo, todas as etapas envolvidas no planejamento, análise e projeto de softwares, com demonstrações práticas dos conceitos apresentados. O livro relata com clareza as circunstâncias atuais do desenvolvimento de software no ambiente empresarial, a preocupação do alinhamento dos recursos da tecnologia da informação com as necessidades do negócio da empresa e como essa relação pode atrapalhar ou ajudar na construção de softwares.



NA WEB

O vídeo mostra uma entrevista com Sérgio Ayres, consultor com vasta experiência em Gestão e Governança Corporativa, abordando a Engenharia de Requisitos.

Web: <http://www.youtube.com/watch?v=P4ixBvRF4NY&feature=related>.



REFERÊNCIAS

MARQUIONI, C. E. Especificação de requisitos no desenvolvimento de software para TV Digital Interativa no Brasil: Reflexões e Relato de Experiência. **Prisma.com**, n. 15, p. 107-147, 2011.

PRESSMAN, R. **Engenharia de Software**. 7. ed. Porto Alegre: AMGH, 2011.

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.

GABARITO

1. Os requisitos funcionais definem as funções que o sistema deve fornecer, de como o sistema deve reagir a entradas específicas e de como deve se comportar em determinadas situações. Em alguns casos, os requisitos funcionais podem, também, declarar o que o sistema não deve fazer. São exemplos de requisitos funcionais: o software deve possibilitar o cálculo das comissões dos vendedores, de acordo com os produtos vendidos; o software deve emitir relatórios de compras e vendas por período; o sistema deve mostrar, para cada aluno, as disciplinas em que foi aprovado ou reprovado. Já os requisitos não funcionais são relacionados com a utilização do software em termos de desempenho, confiabilidade, segurança, usabilidade e portabilidade etc. São exemplos de requisitos não funcionais: o sistema deve ser protegido para acesso apenas de usuários autorizados; o tempo de resposta do sistema não deve ultrapassar 20 segundos; o tempo de desenvolvimento não deve ultrapassar doze meses.
2. Funcionalidade, usabilidade, confiabilidade, eficiência, manutenibilidade e portabilidade.
3. Os requisitos de usuário para um sistema devem descrever os requisitos funcionais e não funcionais, de forma que usuários do sistema que não tenham conhecimentos técnicos detalhados consigam entender. Eles devem especificar somente o comportamento externo do sistema, evitando, sempre que possível, as características do projeto de sistema. Portanto, não devem ser definidos com base em um modelo de implementação, e sim escritos com o uso de linguagem natural, formulários e diagramas intuitivos simples.
4. Os requisitos funcionais dizem respeito às funções que o sistema deve fornecer, de como o sistema deve reagir a entradas específicas e de como deve se comportar em determinadas situações. São exemplos de requisitos funcionais: o software deve possibilitar o cálculo das comissões dos vendedores de acordo com os produtos vendidos; o software deve emitir relatórios de compras e vendas por período; o sistema deve mostrar, para cada aluno, as disciplinas em que foi aprovado ou reprovado.
5. A alternativa correta é a A.



MODELAGEM DE SISTEMAS

UNIDADE



Objetivos de Aprendizagem

- Entender a introdução à UML.
- Exibir um estudo de caso no qual são construídos os diagramas de casos de uso.
- Expor a importância da modelagem de sistemas.
- Trabalhar os conceitos relacionados aos diagramas de casos de uso.
- Apresentar os conceitos básicos de orientação a objetos.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Introdução à UML
- Ferramentas CASE
- Modelagem de Sistemas
- Diagrama de Casos de Uso
- Conceitos Básicos de Orientação a Objetos

INTRODUÇÃO

Caro(a) aluno(a), na segunda unidade, estudamos os requisitos de um sistema. Além disso, foi bastante destacada a importância de um documento de requisitos. Já nesta unidade, você perceberá que, a partir do documento de requisitos, realizaremos a modelagem de um sistema.

A modelagem de sistema é o processo de elaboração de modelos abstratos de um sistema. Normalmente, é representada por meio de um diagrama, em que cada um desses modelos apresenta uma visão ou perspectiva diferente do sistema (SOMMERVILLE, 2011). Esses modelos, normalmente, são elaborados utilizando uma notação gráfica, a qual, em nosso caso, será a UML.

Da mesma forma que os arquitetos elaboram plantas e projetos para que haja a construção de um edifício, os engenheiros de software criam os diagramas UML para auxiliarem os desenvolvedores de software a construírem o software.

A UML define, em sua versão 2.0, treze tipos de diagramas para uso na modelagem de software. Nesta unidade, veremos somente o diagrama de casos de uso. Se você quiser conhecer os demais, teremos mais alguns demonstrados na unidade IV e você também pode consultar alguns livros relacionados nas referências.

O diagrama de casos de uso, o qual será apresentado nesta unidade, ajuda a determinar a funcionalidade e as características do sistema sob o ponto de vista do usuário, além de ser um dos diagramas mais gerais e informais da UML. Para que haja a sua elaboração, deve ser utilizada uma linguagem simples e de fácil compreensão, a fim de que os usuários possam ter uma ideia geral de como o sistema irá se comportar (GUEDES, 2007).

Esse diagrama é muito importante e um dos mais utilizados da UML, pois serve de apoio para a construção de grande parte de outros diagramas. Bons estudos!



INTRODUÇÃO À UML

Segundo Booch, Rumbaugh e Jacobson (2006, p. 13), “a UML (Unified Modeling Language ou Linguagem de Modelagem Unificada) é uma linguagem-padrão para a elaboração da estrutura de projetos de software”, a qual pode ser utilizada para visualização, especificação, construção e documentação de artefatos de software, por meio do paradigma de orientação a objetos. Além disso, a UML tem sido a linguagem-padrão de modelagem de software adotada internacionalmente pela indústria de engenharia de software (GUEDES, 2007).

A UML não é uma linguagem de programação, mas uma linguagem de modelagem que tem, como meta, auxiliar os engenheiros de software a definirem as características do software, tais como seus requisitos, seu comportamento, sua estrutura lógica, a dinâmica de seus processos e até mesmo suas necessidades físicas em relação ao equipamento em que o sistema deverá ser implantado.

Todas essas características são definidas por meio da UML antes do início do desenvolvimento do software (GUEDES, 2007).

De acordo com Booch, Rumbaugh e Jacobson (2006), a UML é apenas uma linguagem de modelagem e é independente de processo de software, visto que pode ser utilizada em modelo cascata, desenvolvimento evolucionário ou em qualquer outro processo que esteja sendo utilizado para o desenvolvimento do software.

A notação UML utiliza diversos símbolos gráficos, existindo uma semântica bem definida para cada um deles, o que permite elaborar diversos modelos. Além disso, a UML tem sido empregada de maneira efetiva em sistemas cujos domínios abrangem os sistemas de informações corporativos, os serviços bancários e os financeiros, os transportes, os serviços distribuídos baseados na web e entre outros. No entanto, ela não se limita à modelagem de software, dado que pode modelar sistemas, tais como o fluxo de trabalho no sistema legal, a estrutura e o comportamento de sistemas de saúde e o projeto de hardware (BOOCH; RUMBAUGH; JACOBSON, 2006).

FERRAMENTAS CASE BASEADAS NA LINGUAGEM UML

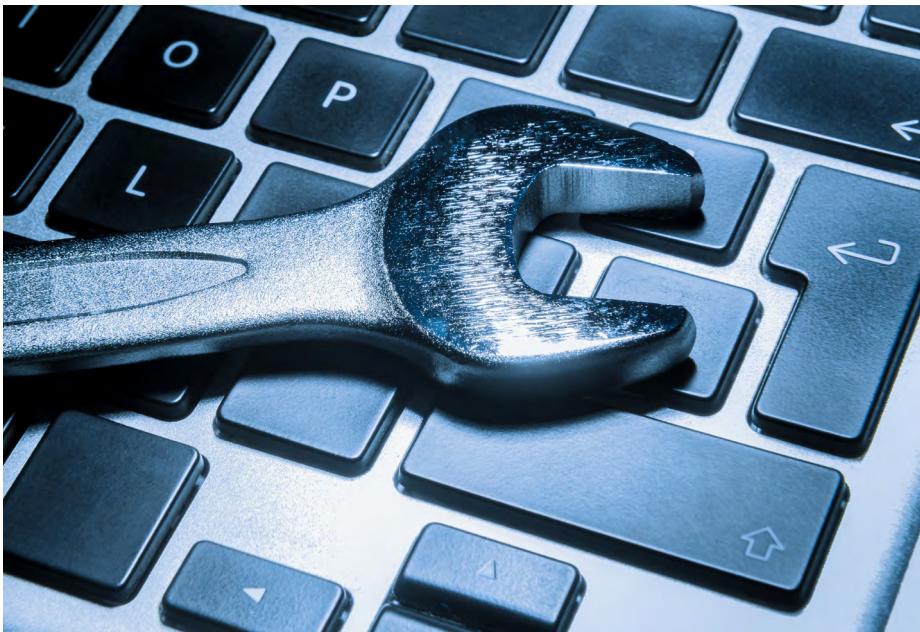
Nesta unidade, já constatamos que as ferramentas CASE (*Computer-Aided Software Engineering* – Engenharia de Software Auxiliada por Computador, em português) são softwares que, de alguma forma, colaboram para a realização de uma ou de mais atividades realizadas durante o processo de desenvolvimento de software. Agora, vamos ver alguns exemplos de ferramentas CASE que suportam a UML, a qual é, em geral, sua principal característica. Existem diversas ferramentas no mercado, dentre as quais podemos destacar:

- **Rational Rose:** esta ferramenta foi desenvolvida pela *Rational Software Corporation*, empresa que estimulou a criação da UML, sendo a primeira ferramenta CASE baseada na linguagem UML. Atualmente, essa ferramenta é bastante usada pelas empresas desenvolvedoras de software. Ela permite a modelagem dos diversos diagramas da UML e a construção de modelos de dados com possibilidade de exportação para construção da base de dados ou de realização de engenharia reversa de uma base de

dados existente. Em 20 de fevereiro de 2003, a empresa *Rational* foi adquirida pela IBM e a ferramenta foi renomeada como *IBM Rational Architect*.

- **Astah Professional:** é uma ferramenta para criação de diagramas UML e possui uma versão gratuita, o *Astah Community*, bem como outras versões pagas. A versão gratuita possui algumas restrições de funções, porém, para as que precisaremos nesta unidade, ela será suficiente. Portanto, será essa a ferramenta que utilizaremos para modelar os diagramas da UML que aprenderemos. Anteriormente, essa ferramenta era conhecida por Jude.
- **Visual Paradigm for UML ou VP-UML:** esta ferramenta oferece uma versão que pode ser baixada gratuitamente, a *Community Edition*, porém ela não suporta todos os serviços e opções disponíveis nas versões *standard* ou *professional* da ferramenta. Para quem deseja praticar a UML, a versão gratuita é uma boa alternativa, apresentando um ambiente amigável de fácil compreensão.
- **Enterprise Architect:** esta ferramenta não possui uma edição gratuita, assim como as anteriores, mas é uma das ferramentas que mais oferecem recursos compatíveis com a UML 2.

O importante, em nossos estudos, é que você consiga entender como se inicia a modelagem do seu sistema. Para tanto, buscamos que a informação não fique apenas anotada em um documento, mas que você possa facilitar o processo de desenvolvimento utilizando os recursos de modelagem e de construção de diagramas.



FERRAMENTAS CASE

Uma ferramenta CASE é um software que pode ser utilizado para apoiar as atividades do processo de software, como a engenharia de requisitos, o projeto, o desenvolvimento de programa e os testes. As ferramentas CASE podem incluir editores de projeto, dicionários de dados, compiladores, depuradores, ferramentas de construção de sistemas e entre outros (SOMMERVILLE, 2011).

Sommerville (2011) expõe alguns exemplos de atividades que podem ser automatizadas utilizando a CASE. Entre elas, estão:

1. O desenvolvimento de modelos gráficos de sistemas enquanto parte das especificações de requisitos ou do projeto de software.
2. A compreensão de um projeto utilizando um dicionário de dados que carrega informações sobre as entidades e sua relação em um projeto.
3. A geração de interfaces com usuários, a partir de uma descrição gráfica da interface, que é criada interativamente pelo usuário.
4. A depuração de programas pelo fornecimento de informações sobre um programa em execução.

5. A tradução automatizada de programas a partir de uma antiga versão de uma linguagem de programação, como Cobol, para uma versão mais recente.

A tecnologia CASE está, atualmente, disponível para a maioria das atividades de rotina no processo de software, proporcionando, assim, algumas melhorias na qualidade e na produtividade de software.

Existem, basicamente, duas formas de classificação geral para as ferramentas CASE. A primeira delas divide-se em: *Upper CASE*, *Lower CASE*, *Integrated-CASE* e *Best in Class*:

- **Upper CASE ou U-CASE ou Front-End:** são as ferramentas que estão voltadas para as primeiras fases do processo de desenvolvimento de sistemas, como a análise de requisitos, projeto lógico e documentação. São utilizadas por analistas e pessoas mais interessadas na solução do problema do que na implementação.
- **Lower CASE ou L-CASE ou Back-End:** são, praticamente, o oposto das ferramentas *Upper CASE*. Elas oferecem suporte nas últimas fases do desenvolvimento, como a codificação, os testes e a manutenção.
- **Integrated CASE ou I-CASE:** são ferramentas que, além de englobar características das *Upper* e *Lower CASE's*, ainda oferecem outras características, como controle de versão, por exemplo. Entretanto, somente são utilizadas em projetos de desenvolvimento muito grandes, por serem bastante caras e difíceis de operar.
- **Best in Class ou Kit de Ferramenta:** semelhante às I-CASE, os Kits de Ferramenta também acompanham todo o ciclo de desenvolvimento. Entretanto, possuem a propriedade de conjugar sua capacidade a outras ferramentas externas complementares, as quais variam de acordo com as necessidades do usuário. Para um melhor entendimento, podemos compará-las a um computador sem o kit multimídia: as *Best in Class* seriam o computador que funciona normalmente, enquanto as outras ferramentas fariam o papel do kit multimídia, promovendo um maior potencial de trabalho para a máquina. São mais usadas para desenvolvimento corporativo, apresentando controle de acesso, versão, repositórios de dados, entre outras características.

A segunda forma divide as ferramentas CASE em orientadas a função e orientadas a atividade:

- **Orientadas a função:** seriam as *Upper CASE* e *Lower CASE*. Baseiam-se na funcionalidade das ferramentas, ou seja, são as que têm funções diferentes no ciclo de vida de um projeto, como representar apenas o Diagrama de Entidades e Relacionamentos (DER) ou o Diagrama de Fluxo de Dados (DFD).
- **Orientadas a atividade:** seriam as *Best in Class* e as I-CASE, as quais processam atividades, tais como especificações, modelagem e implementação.



REFLITA

As diferenças não são insignificantes – são bem semelhantes às diferenças entre Salieri e Mozart. Estudos após estudos demonstram que os melhores projetistas produzem estruturas mais rápidas, menores, mais simples, mais claras e com menos esforço.

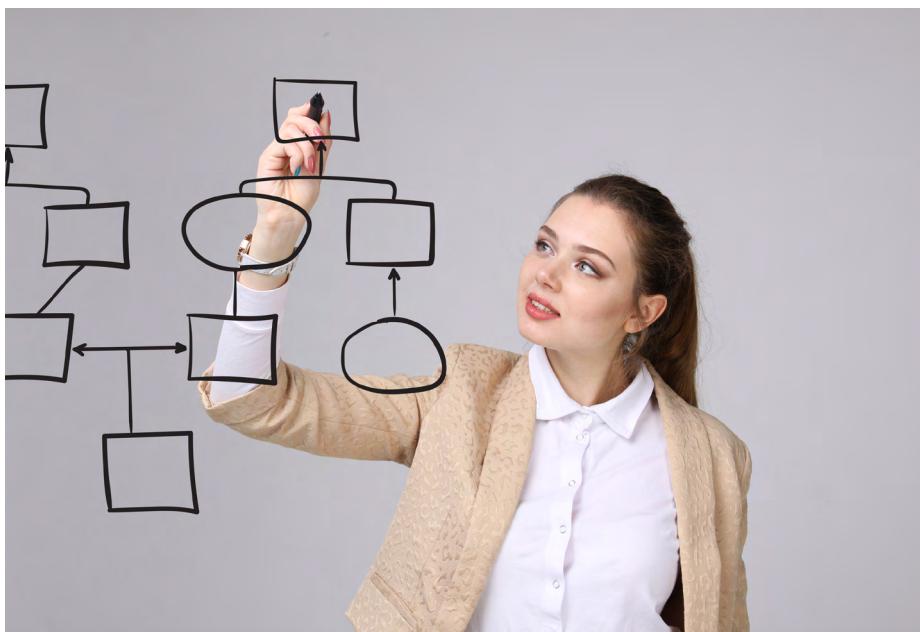
(Frederick Phillips Brooks Jr.)



SAIBA MAIS

Conheça, no artigo escrito por Leandro Ribeiro, um estudo prático sobre UML e uma introdução acerca de um dos seus principais diagramas: o de Casos de Uso. São explicados os fundamentos de uma linguagem muito importante não só para desenvolvedores, mas para todos os profissionais que se envolvem em projetos de desenvolvimento de sistemas e clientes. Para saber mais, acesse o link disponível em: <https://www.devmedia.com.br/o-que-e-uml-e-diagramas-de-caso-de-uso-introducao-pratica-a-uml/23408#ixzz3zyz3sLwB>.

Fonte: Os autores.



MODELAGEM DE SISTEMAS

Caro(a) aluno(a), a necessidade de planejamento no desenvolvimento de sistemas de informação leva ao conceito de modelagem de software, ou seja, antes do software ser concebido, deve-se criar um modelo para ele. Um modelo pode ser entendido como uma representação idealizada de um sistema a ser construído. São exemplos: maquetes de edifício, plantas de casa, fluxogramas etc.

A modelagem de sistemas de software baseia-se na utilização de notações gráficas e textuais com o objetivo de construir modelos que representem as partes essenciais de um sistema. São várias as razões para se utilizar modelos na construção de sistemas:

1. No desenvolvimento do software, usamos desenhos gráficos denominados diagramas, a fim de representar o comportamento do sistema. Esses diagramas seguem um padrão lógico e possuem uma série de elementos gráficos que carregam um significado pré-definido.
2. Apesar de um diagrama conseguir expressar diversas informações de forma gráfica, em diversos momentos, há a necessidade de se adicionar

informações em forma de texto, com o objetivo de explicar ou definir certas partes. A modelagem de um sistema em forma de diagrama, em conjunto com a informação textual, forma a documentação de um sistema de software.

3. O rápido crescimento da capacidade computacional das máquinas resultou na demanda por sistemas de software cada vez mais complexos, que tirassem proveito de tal capacidade. Por sua vez, o surgimento desses sistemas mais complexos gerou a necessidade de reavaliação na forma de desenvolver sistemas. Desde o aparecimento do primeiro computador até os dias de hoje, as técnicas para a construção de sistemas computacionais têm evoluído para suprir as necessidades do desenvolvimento de software. A seguir, há um parâmetro histórico acerca dessa evolução:
 - Década de 50/60: os sistemas de software eram bastante simples e, dessa forma, as técnicas de modelagem também. Era a época dos fluxogramas e diagramas de módulos.
 - Década de 70: nessa época, houve uma grande expansão do mercado computacional. Sistemas complexos começavam a surgir e, por consequência, modelos mais robustos foram propostos. Além disso, surge a programação estruturada e, no final da década, a análise e o projeto estruturado.
 - Década de 80: surge a necessidade de se ter interfaces homem-máquina mais sofisticadas, o que originou a produção de sistemas de software mais complexos. A análise estruturada consolidou-se na primeira metade da década e, em 1989, Edward Yourdon lançou o livro *Análise Estruturada Moderna*, tornando-se uma referência no assunto.
 - Década de 90: inicia-se um novo paradigma de modelagem, a análise orientada a objetos, enquanto resposta para as dificuldades encontradas na aplicação da análise estruturada em certos domínios de aplicação.
 - Final da década de 90 e momento atual: o paradigma da orientação a objetos atinge sua maturidade. Os conceitos de padrões de projetos (*design patterns*), frameworks de desenvolvimento, componentes e padrões de qualidade começam a ganhar espaço. Também surge a Linguagem de Modelagem Unificada (UML), que é a ferramenta de modelagem utilizada no desenvolvimento atual de sistemas.

O processo de desenvolvimento de software é uma atividade bastante complexa. Isso se reflete no alto número de projetos de software que não chegam ao fim ou que extrapolam recursos de tempo e de dinheiro alocados. Em um estudo clássico sobre projetos de desenvolvimento de software, o qual foi realizado em 1994, constatou-se que:

- Porcentagem de projetos que terminam dentro do prazo estimado: 10%.
- Porcentagem de projetos que são descontinuados antes de chegarem ao fim: 25%.
- Porcentagem de projetos acima do custo esperado: 60%.
- Atraso médio nos projetos: um ano.

Os modelos construídos na fase de análise devem ser cuidadosamente validados e verificados. O objetivo da validação é o de assegurar que as necessidades do cliente estão sendo atendidas. Nessa atividade, os analistas apresentam os modelos criados para representar o sistema aos futuros usuários, a fim de que esses modelos sejam validados.

Já a verificação objetiva analisar se os modelos construídos estão em conformidade com os requisitos definidos. Na verificação dos modelos, são analisadas a exatidão de cada modelo, separadamente, bem como a consistência entre os modelos. A verificação é uma etapa típica da fase de projeto e é a próxima etapa do desenvolvimento de software.

Caro(a) aluno(a), utilizaremos a UML para realizar a modelagem de sistemas. Na primeira unidade, estudamos alguns conceitos relacionados com a orientação de objetos e fizemos uma introdução à linguagem UML. Lembre-se de que os artefatos de software produzidos durante a modelagem servirão de base para a fase seguinte do ciclo de desenvolvimento de sistemas, ou seja, a fase projeto.

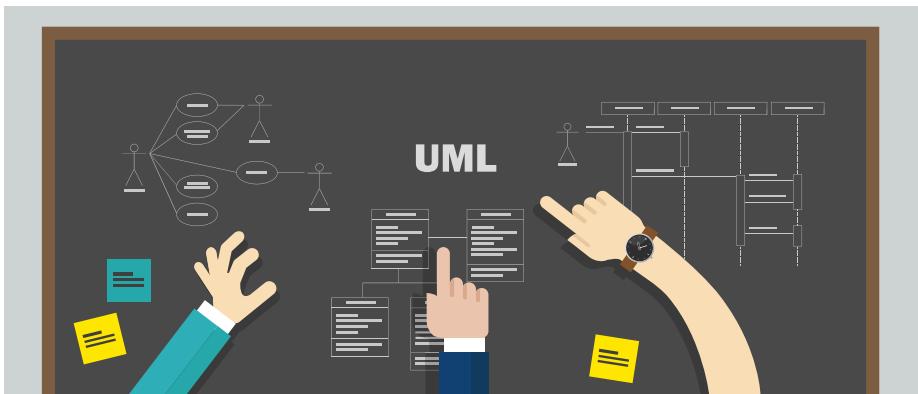


DIAGRAMA DE CASOS DE USO

O diagrama de casos de uso (*use case diagram*) é, dentre todos os diagramas da UML, o mais abstrato, flexível e informal. Ele é utilizado, principalmente, no início da modelagem do sistema, a partir do documento de requisitos, e pode ser consultado e possivelmente modificado durante todo o processo de engenharia. Além disso, serve de base para a modelagem de outros diagramas (GUEDES, 2007).

O principal objetivo desse diagrama é o de modelar as funcionalidades e serviços oferecidos pelo sistema, buscando, por meio de uma linguagem simples, demonstrar o comportamento externo do sistema a partir da perspectiva do usuário.

De acordo com Silva (2007), o diagrama de caso de uso incorpora o conjunto de requisitos funcionais estabelecidos para o software que está sendo modelado. Esses requisitos devem estar descritos no documento de requisitos, assim como já explicamos na unidade anterior. Além disso, há uma correspondência entre os requisitos funcionais previstos para o software e os casos de uso. Os requisitos não funcionais não aparecem no diagrama de casos de uso, pois não constituem o foco da modelagem que estamos realizando.

O diagrama de casos de uso é composto por atores, casos de uso e seus relacionamentos. A seguir, descreveremos cada um desses elementos.

ATORES

Um ator representa um papel que um ser humano, um dispositivo de hardware ou até mesmo outro sistema desempenha com o sistema (BOOCH; RUMBAUGH; JACOBSON, 2006). Assim, um ator pode ser qualquer elemento externo que interage com o software. O nome do ator identifica qual é o papel assumido por ele dentro do diagrama (GUEDES, 2007).

Um caso de uso é sempre iniciado por um estímulo de um ator e, ocasionalmente, outros atores também podem participar. A Figura 1 apresenta alguns exemplos de atores:



Figura 1 – Exemplos de atores

Fonte: o autor.

CASOS DE USO

De acordo com Booch, Rumbaugh e Jacobson (2006), um caso de uso especifica o comportamento de um sistema ou de parte de um, referindo-se a serviços, tarefas ou funções apresentadas, como cadastrar funcionário ou emitir relatório de produtos, por exemplo.

No diagrama de caso de uso, não é possível documentar os casos de uso. Nem mesmo a UML oferece um recurso para que isso seja feito, mas é indicado que cada caso de uso seja documentado, demonstrando qual é o comportamento pretendido para o caso de uso em questão e quais funções ele executará quando for solicitado. Essa documentação deverá, também, ser elaborada de acordo com o documento de requisitos e poderá auxiliar o desenvolvedor na elaboração dos demais diagramas da UML. A Figura 2 apresenta alguns exemplos de casos de uso:



Figura 2 – Exemplos de casos de uso

Fonte: o autor.

Normalmente, os nomes de casos de uso são breves expressões verbais ativas, nomeando algum comportamento encontrado no vocabulário do sistema cuja modelagem está sendo realizada a partir do documento de requisitos (BOOCH; RUMBAUGH; JACOBSON, 2006). São verbos que podem ser usados para nomear os casos de uso: efetuar, cadastrar, consultar, emitir, registrar, realizar, manter, verificar, entre outros.

RELACIONAMENTO ENTRE CASOS DE USO E ATORES

Segundo Melo (2004), os casos de uso representam conjuntos bem definidos de funcionalidades do sistema, os quais precisam se relacionar com outros casos de uso e com atores que enviaram e receberam mensagens deles. Podemos ter os relacionamentos de associação, generalização, extensão e inclusão da seguinte forma:

- Para relacionamentos entre atores e casos de uso: somente associação.
- Para relacionamentos de atores entre si: somente generalização.
- Para relacionamentos de casos de uso entre si: generalização, extensão e inclusão.

ASSOCIAÇÃO

A associação é o único relacionamento possível entre ator e caso de uso e sempre é binária, ou seja, envolve apenas dois elementos. Melo (2004, p. 60) afirma que ela “representa a interação do ator com o caso de uso, ou seja, a comunicação entre atores e casos de uso, por meio do envio e recebimento de mensagens”.

Um relacionamento de associação demonstra que o ator utiliza a funcionalidade representada pelo caso de uso. Esse tipo de relacionamento é representado por uma reta ligando o ator ao caso de uso. Pode ser que essa reta possua, em sua extremidade, uma seta, que indica a navegabilidade dessa associação, ou seja, se as informações são fornecidas pelo ator em caso de uso (nesse caso, a seta aponta para o caso de uso), se são transmitidas pelo caso de uso ao ator (nesse caso, a seta aponta para o ator) ou ambos (a reta não possui setas) (GUEDES, 2007).

A Figura 3 representa uma associação entre um ator e um caso de uso, em que o ator fornece uma informação para o caso de uso:

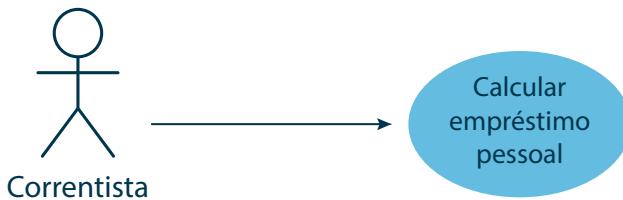


Figura 3 – Associação entre um ator e um caso de uso

Fonte: o autor.

Agora, veja a Figura 4: ela mostra o ator gerente administrativo recebendo o relatório de vendas por período (note que ele não solicita a emissão do relatório, somente o recebe):

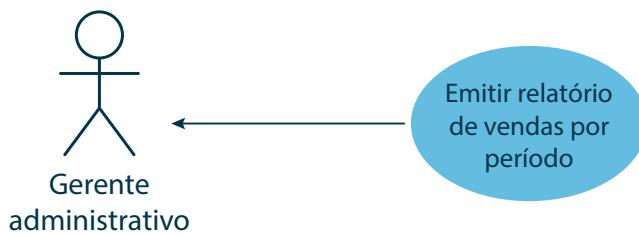


Figura 4 – Associação entre um ator e um caso de uso

Fonte: o autor.

Na figura 5, outro exemplo, percebemos que o ator denominado submissor utiliza, de alguma forma, o serviço realizar submissão e a informação referente a esse processo trafega nas duas direções:

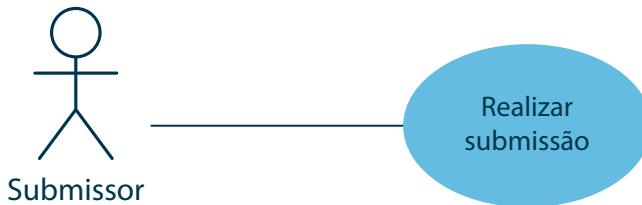


Figura 5 – Associação entre um ator e um caso de uso

Fonte: o autor.

GENERALIZAÇÃO

Já explicamos o conceito de generalização/especialização quando tratamos sobre herança. Aqui, no diagrama de casos de uso, também aplicamos esse conceito, ou seja, o relacionamento de generalização entre casos de uso pode ocorrer quando existirem dois ou mais casos de usos com características semelhantes, apresentando pequenas diferenças entre si.

Quando isso acontece, define-se um caso de uso geral, o qual deverá possuir as características compartilhadas por todos os casos de uso em questão. Então, relacionamos esse caso de uso geral com os casos de uso específicos, os quais devem conter somente a documentação das características específicas de cada um deles. Dessa forma, evita-se reescrever toda a documentação para todos os casos de uso envolvidos, porque toda a estrutura de um caso de uso generalizado é herdada pelos casos de uso especializados, incluindo quaisquer possíveis associações que o caso de uso generalizado possua.

A associação é representada por uma reta com uma seta mais grossa, partindo dos casos de uso especializados até atingir o caso de uso geral, assim como ilustra a Figura 6:

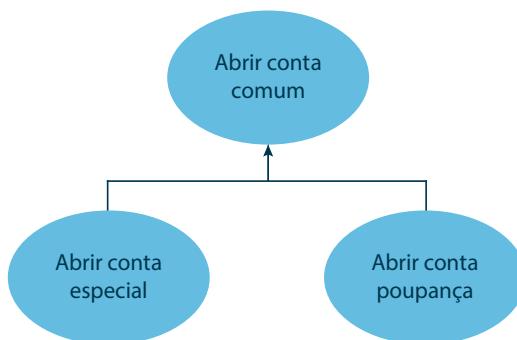


Figura 6 – Exemplo de generalização entre casos de uso

Fonte: o autor.

Agora, vamos entender o que esse exemplo está representando. Em um banco, existem três opções de abertura de contas: abertura de conta comum, de conta especial e de conta poupança, cada uma representada por um caso de uso diferente.

As aberturas de conta especial e de conta poupança possuem todas as características e requisitos da abertura de conta comum, mas cada uma delas possui algumas características e requisitos próprios, o que justifica elencá-las como especializações do caso de uso abertura de conta comum, detalhando-se as particularidades de cada caso de uso especializado em sua própria documentação (GUEDES, 2007).

O relacionamento de generalização/especialização também pode ser aplicado entre atores. A Figura 7 apresenta um exemplo em que existe um ator geral chamado cliente e dois atores especializados chamados pessoa física e pessoa jurídica:

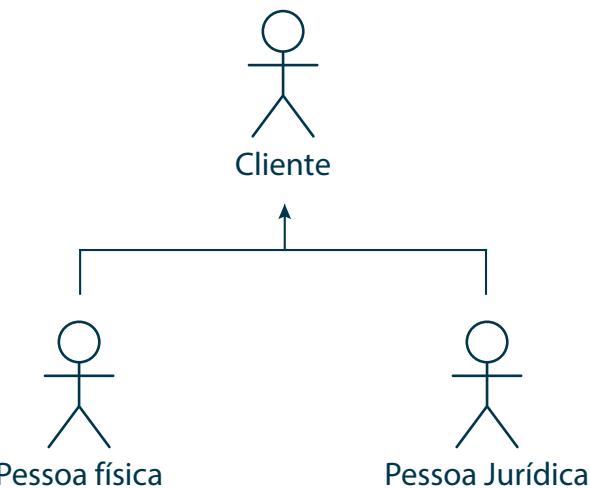


Figura 7 - Generalização entre atores

Fonte: o autor.

INCLUSÃO

Esse tipo de relacionamento é possível somente entre casos de uso e é utilizado quando existem ações comuns a mais de um caso de uso. Quando isso ocorre, a documentação dessas ações é colocada em um caso de uso específico, permitindo que outros casos se utilizem dessas ações, evitando a descrição de uma mesma sequência de passos em vários casos de uso (GUEDES, 2007).

Um relacionamento de inclusão entre casos de uso significa que o caso de uso base incorpora explicitamente o comportamento de outro. O caso de uso incluído nunca permanece isolado, mas é apenas instanciado enquanto parte de alguma base maior que o inclui (BOOCH; RUMBAUGH; JACOBSON, 2006).

O relacionamento de inclusão pode ser comparado com a chamada sub-rotina. Portanto, indica uma obrigatoriedade, ou seja, quando um caso de uso base possui um relacionamento de inclusão com outro caso de uso, a execução do primeiro obriga, também, a execução do segundo (GUEDES, 2007).

A representação do relacionamento de inclusão é feita por meio de uma reta tracejada que contém uma seta em uma de suas extremidades que é rotulada com a palavra-chave <<include>>. A seta deve sempre apontar para o caso de uso a ser incluído.

Veja um exemplo na Figura 8 a seguir:
Nesse exemplo, sempre que um saque ou depósito ocorrer, ele deve ser registrado para fins de histórico bancário. Como as rotinas para registro de um saque ou um

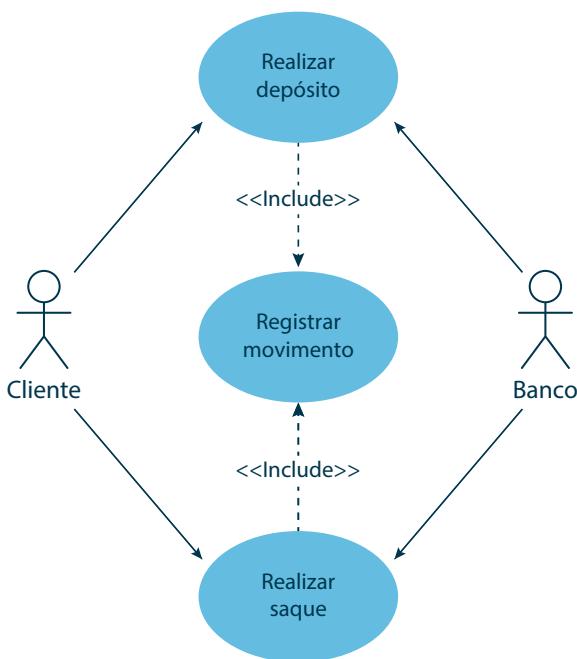


Figura 8 – Exemplo de inclusão de caso de uso
Fonte: Guedes (2007, p. 43).

depósito são extremamente semelhantes, colocou-se a rotina de registro em um caso de uso à parte, chamado registro movimento, que será executado, obrigatoriamente, sempre que os casos de uso como realizar depósito ou saque forem executados. Assim, só é preciso descrever os passos para registrar um movimento no caso de uso incluído (GUEDES, 2007). Além disso, temos dois casos de uso base e um caso de uso a ser incluído.

Agora, veja outro exemplo na Figura 9. Aqui, está sendo apresentada a seguinte situação: em algum ponto dos casos de uso de realizar a matrícula do aluno (caso de uso base), cadastrar pagamento da sua mensalidade (caso de uso base) e emitir seu histórico escolar (caso de uso base). É necessário validar a matrícula (caso de uso a ser incluído) do aluno. Assim, nesses pontos, o caso de uso da validação da matrícula será internamente copiado.

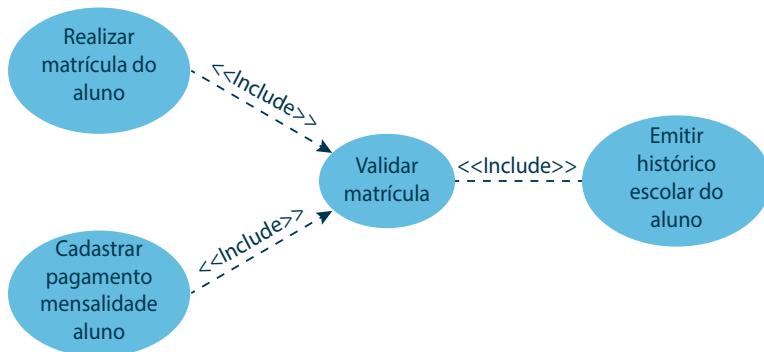


Figura 9 – Outro exemplo de inclusão de caso de uso

Fonte: o autor.

EXTENSÃO

O relacionamento de extensão também é possível somente entre casos de uso e é utilizado para modelar as rotinas opcionais de um sistema que ocorrerão somente se uma determinada condição for satisfeita. A extensão separa um comportamento obrigatório de um opcional.

As associações de extensão possuem uma representação muito semelhante à das associações de inclusão, as quais também são representadas por uma reta

tracejada, diferenciando-se por possuir um estereótipo com o texto “<<extend>>” e pelo fato de que a seta aponta para o caso de uso que estende, ou seja, nesse caso, a seta aponta para o caso de uso base (GUEDES, 2007). Veja, na próxima figura, um exemplo de relacionamento de extensão:

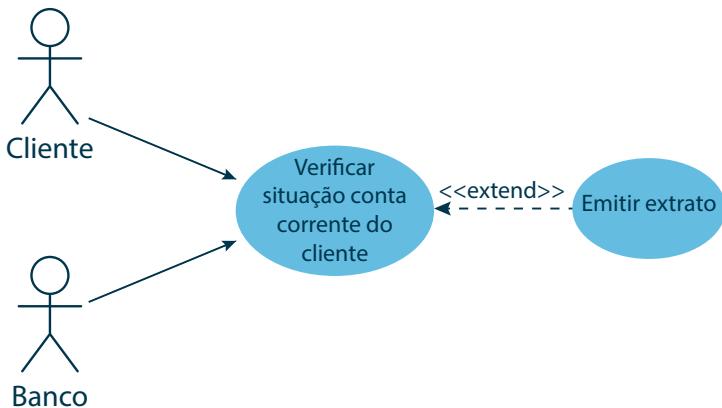


Figura 10 – Exemplo de extensão de caso de uso

Fonte: o autor.

Na Figura 10, mostra-se que tanto o cliente, quanto o banco podem verificar a situação da conta corrente do cliente e, se for o caso, pode-se emitir um extrato. Contudo, note que o extrato somente será emitido se alguma condição do caso de uso base – verificar a situação da conta corrente do cliente – for satisfeita. Caso contrário, o extrato nunca será emitido.

Agora, vamos mostrar um exemplo bastante comum do que acontece quando utilizamos sistemas via Internet. Para utilizar os serviços, o cliente deve logar no sistema e, caso seja a primeira vez, realizar o seu cadastro pessoal:



Figura 11 – Outro exemplo de extensão de caso de uso

Fonte: o autor.

No exemplo da Figura 11, o caso de base é o realizar login no site, enquanto o realizar cadastro de dados pessoais é o caso de uso a ser estendido.

Para facilitar o entendimento do relacionamento de extensão, vamos mostrar mais um exemplo de uso. Na figura 12, apresenta-se a seguinte situação: durante a execução do caso de uso efetuar venda, a venda pode ser realizada para um cliente especial. Nesse caso, é necessário, em um ponto predefinido, incluir uma complementação da rotina que será responsável por calcular o desconto para um cliente especial. Da mesma forma, durante o pagamento, pode haver algum tipo de falha na autorização do cartão. Veja que esse não é um procedimento normal, pois é possível ter pagamentos em dinheiro, cheque etc. Assim, havendo falha na autorização, o caso de uso pertinente deverá dar um tratamento para a situação:

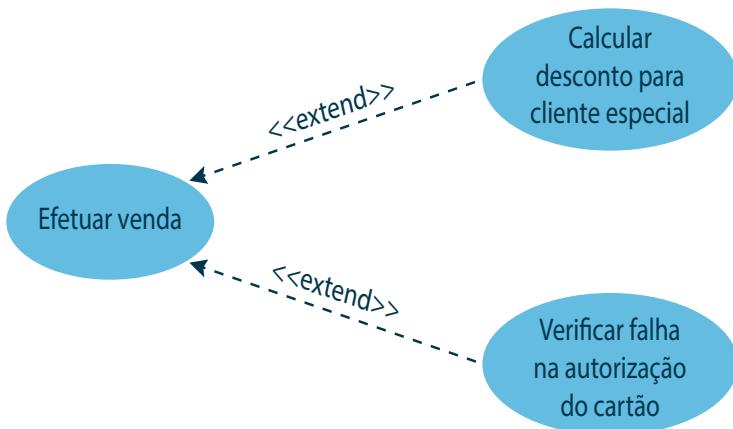


Figura 12 – Representação de um Relacionamento Extend

Fonte: o autor.

RESUMO DE RELACIONAMENTOS ENTRE CASOS DE USO E ATORES

A partir do contexto abordado nesta unidade, relativo ao diagrama de caso de uso, demonstraremos, no quadro a seguir, as possibilidades de relacionamentos que podem ocorrer entre os casos de uso e os atores descritos em um sistema:

Quadro 1 – Relacionamentos entre casos de uso e atores

	ASSOCIAÇÃO	ESPECIALIZAÇÃO/ GENERALIZAÇÃO	INCLUSÃO	EXTENSÃO
CASO DE USO E CASO DE USO	-----	OK	OK	OK
ATOR E ATOR	-----	OK	-----	-----
ATOR E CASO DE USO	OK	-----	-----	-----

Fonte: o autor.

ESTUDO DE CASO

Neste estudo de caso, será apresentado um novo documento de requisitos, agora, de um laboratório de análises clínicas. Com base nesse documento de requisitos, vamos elaborar o diagrama de casos de uso e o diagrama de classes, mas, dessa vez, faremos tudo isso passo a passo.

Antes de começarmos a ler o documento de requisitos, gostaria que você imaginasse que foi a um médico, por exemplo, um clínico geral. Para te dar um diagnóstico preciso e correto, esse médico pediu que você fizesse uma série de exames, por exemplo: hemograma, glicemia, creatinina, triglicerídeos e urocultura. Ele anotou essa lista de exames em um pedido de exames e você, de posse desse pedido, foi até um laboratório de análises clínicas chamado São João, a fim de fazer os exames. É nesse momento que começa o nosso documento de requisitos.

Documento de requisitos – laboratório de análises clínicas

O laboratório de análises clínicas São João deseja informatizar suas atividades, pois, hoje, não há controle sobre o histórico de cada paciente (dos exames que ele já realizou) e gasta-se muito tempo com atividades que poderiam ser feitas por um sistema informatizado.

Hoje, o laboratório funciona da seguinte forma:

- O paciente (você) chega ao laboratório com o pedido dos exames (preenchido pelo médico, o clínico geral ou qual você acabou de consultar).
- Se for a primeira vez que o paciente vai fazer exames, é preenchida uma ficha de cadastro com os seguintes dados: nome, endereço, cidade, UF, CEP, telefone, data de nascimento, RG e CPF.
- A recepcionista (a moça que te atendeu quando você chegou ao laboratório) preenche uma ficha com o nome do paciente, nome do convênio, os nomes dos exames que o paciente irá fazer e a data e horário da realização de cada um. No final da ficha, é anotada a data em que os exames estarão prontos. A primeira via dessa ficha é entregue ao paciente, ou seja, você).
- O resultado do exame é colocado no envelope para posterior retirada pelo paciente.

O laboratório deseja que o novo sistema possa fornecer informações rápidas, precisas e seguras, a fim de melhorar suas atividades administrativas e o atendimento aos seus pacientes. Dessa forma, você vai permanecer bem menos no laboratório, pois os processos estarão automatizados. Para tanto, o novo sistema deve:

- Permitir o cadastro dos pacientes do laboratório, com todos os dados preenchidos na ficha de cadastro. Esse cadastro será realizado pelas recepcionistas.
- Permitir o cadastro dos exames que o laboratório pode realizar. Cada exame pertence a um grupo. Por exemplo, o exame **hemograma**, pertence ao grupo **sangue**. Além disso, para cada exame, é preciso saber o seu código, descrição, valor e procedimentos para a sua realização. Por exemplo, para o hemograma, o paciente deve estar em jejum. Esse cadastro será realizado pelos bioquímicos.
- Permitir o cadastro dos pedidos de exames dos pacientes. É necessário saber qual é o nome do paciente, o nome do médico que está solicitando os exames, o nome do convênio que o paciente irá utilizar para esse pedido, data e horário dos exames. Atenção: cada exame pode ser realizado em datas e horários diferentes. Além disso, é preciso saber os nomes dos exames a serem feitos, bem como a data, o horário em que cada exame ficará

pronto (cada exame pode ficar pronto em uma data e horário diferente) e o valor de cada um deles. Lembre-se de que o médico pode solicitar mais de um exame em cada pedido (no seu caso, o médico solicitou cinco exames). Esse cadastro será realizado pelas recepcionistas.

- Emitir a ficha do paciente, a qual contém todos os dados cadastrados. Esse relatório será solicitado e recebido tanto pelas recepcionistas quanto pelo departamento administrativo do laboratório.
- Emitir relatório com todos os exames que o laboratório realiza, com o código, a descrição, os procedimentos e o valor de cada exame, agrupados por grupo de exame, devendo ser impressos, nesse relatório, o código e a descrição do grupo. O relatório será solicitado e recebido pelas recepcionistas.
- Emitir o pedido do exame em três vias, com todos os dados do pedido do exame. O relatório será emitido pela recepcionista e a 1^a via será entregue ao paciente (comprovante da entrega do exame), a 2^a para o departamento de faturamento (para a cobrança dos exames dos convênios) e a 3^a via para os bioquímicos (para a realização dos exames).
- Emitir relatório com os resultados dos exames por pedido, contendo o nome do paciente, data e horário do exame (da sua realização), nome do médico que solicitou o procedimento, nome do convênio e o resultado de cada exame realizado, caso tenha sido mais de um. O relatório será solicitado pela recepcionista e entregue ao paciente (não é necessário que a recepcionista fique com esse relatório).

Passo a passo para a elaboração do diagrama de casos de uso

1. Leia novamente o documento de requisitos apresentado e verifique quais são os usuários do sistema. Essas pessoas serão os atores. Assim, faça uma lista com os atores:
 - a. Recepção.
 - b. Bioquímicos.
 - c. Departamento Administrativo.
 - d. Departamento de Faturamento.
 - e. Paciente.

2. Agora, faça uma lista com as funcionalidades do sistema. Algumas aparecem descritas claramente no documento de requisitos. Cada relatório que é mencionado no documento será uma funcionalidade. Então, já sabemos que teremos as seguintes funcionalidades:
 - a. Emitir ficha do paciente.
 - b. Emitir relatório de exames.
 - c. Emitir pedido de exame.
 - d. Emitir resultados dos exames.
3. É importante observar que, além dos relatórios, um sistema precisa ter os cadastros para que o usuário consiga inserir os dados no sistema, a fim de ser possível a emissão dos relatórios. Então, com base nos relatórios mencionados, teremos os seguintes cadastros:
 - a. Cadastrar pacientes.
 - b. Cadastrar exames.
 - c. Cadastrar pedidos de exame.
 - d. Cadastrar resultados dos exames.
4. Cada uma das funcionalidades relacionadas será um caso de uso em nosso diagrama. Então, agora, você precisa verificar qual(is) ator(es) estará(ão) envolvido(s) em cada caso de uso. Você descobrirá tal informação ao fazer uma nova leitura do documento de requisitos.
5. Além das funcionalidades já relacionadas, é importante pensar que algumas informações podem ser transformadas em classes, facilitando o uso e manutenção do sistema. Por exemplo: o sistema pode ter, além dos cadastros relacionados, um cadastro de médico, evitando que a recepcionista precisasse digitar o nome completo do médico toda vez que um pedido de exame daquele profissional fosse lançado no sistema. Seguindo esse mesmo raciocínio, alguns cadastros seriam importantes para o sistema, a saber:
 - a. Cadastro de médicos.
 - b. Cadastro de convênios.
 - c. Cadastro de cidades.
 - d. Cadastro de UF's.
 - e. Cadastro de grupos de exames (nesse caso, esse cadastro é de suma importância, pois o relatório de exames deve ser agrupado por grupo de exames).

Com a criação de um cadastro, evita-se que o usuário cadastre o mesmo grupo várias vezes).

6. Agora, é só você utilizar a ferramenta Astah e desenhar o seu diagrama. Depois que você tiver terminado de desenhar, veja se ficou parecido com o meu, exposto na Figura 13. Note que o nome do caso de uso sempre começa com um verbo:

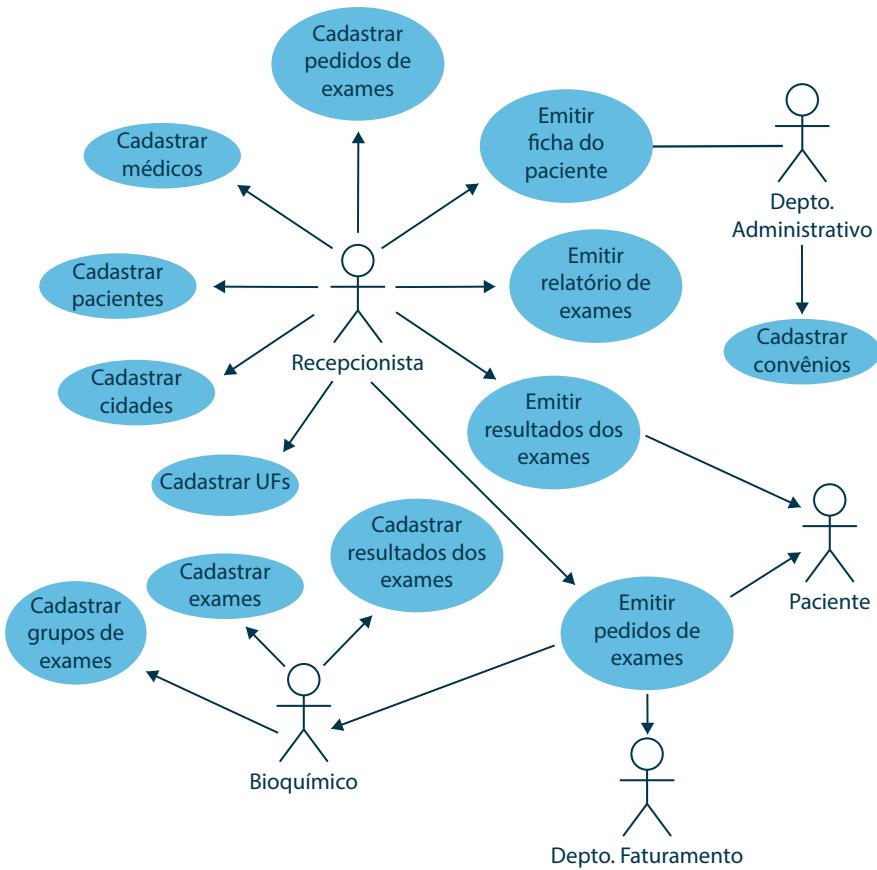


Figura 13 – Diagrama de casos de uso para o laboratório de análises clínicas

Fonte: o autor.



CONCEITOS BÁSICOS DE ORIENTAÇÃO A OBJETOS

A UML é totalmente baseada no paradigma de Orientação a Objetos (OO). Para compreendê-la corretamente, precisamos, antes, estudar alguns dos conceitos relacionados a orientação a objetos.

OBJETOS

Segundo Melo (2004), um objeto é qualquer coisa, em forma concreta ou abstrata, que existe física ou apenas conceitualmente no mundo real. São exemplos de objetos: cliente, professor, carteira, caneta, carro, disciplina, curso, caixa de diálogo. Além disso, os objetos possuem características e comportamentos.

CLASSE

Uma classe é uma abstração de um conjunto de objetos que possuem os mesmos tipos de características e comportamentos, sendo representada por um retângulo que pode ter até três divisões. A primeira divisão armazena o nome da classe; a segunda, os atributos (características), enquanto a terceira carrega os métodos.

Geralmente, uma classe possui atributos e métodos, mas é possível encontrar classes que contenham apenas uma dessas características ou até mesmo nenhuma, quando se trata de classes abstratas. A Figura 14 apresenta um exemplo de classe:

De acordo com Melo (2004), o momento inicial para a identificação de classes em um documento de requisitos é assinalar os substantivos. Note que esses substantivos podem levar a objetos físicos (cliente, livro, computador) ou objetos conceituais (reserva, cronograma, norma). Em seguida, é preciso identificar somente os objetos que possuem importância para o sistema, verificando o que realmente consiste em ser objeto e o que consiste em ser atributo. Ainda, é preciso fazer uma nova análise dos requisitos para identificar as classes implícitas no contexto trabalhado, excluir classes parecidas ou juntar duas classes em uma única classe. Vale a pena ressaltar que esse processo será iterativo e não será possível definir todas as classes de uma só vez.

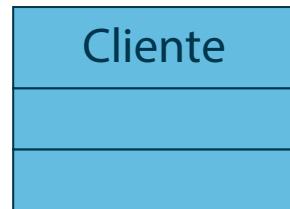


Figura 14 – Exemplo de classe
Fonte: o autor.

ATRIBUTOS

Uma classe, normalmente, possui atributos que representam as suas características, as quais costumam variar de objeto para objeto, como o nome de um objeto da classe cliente ou a cor em um objeto da classe carro, por exemplo. Tais aspectos são os responsáveis por diferenciar um objeto de outro da mesma classe.

De acordo com Guedes (2007), na segunda divisão da classe, aparecem os atributos. Cada atributo deve possuir um nome e, eventualmente, o tipo de dado que armazena, por exemplo, *integer*, *float* ou *char*. Assim, a classe especifica a

estrutura de um objeto sem informar quais serão os seus valores. Em relação ao objeto, este corresponde a uma instância de uma classe em um determinado momento. Vou apresentar um exemplo para facilitar o seu entendimento:

Uma classe pessoa possui os atributos nome, sexo e data de nascimento. Essa classe pode ter um objeto ou instância com os seguintes valores para cada atributo, respectivamente: Márcia, feminino e 22/03/1969. Dessa forma, em um sistema, devemos trabalhar com as instâncias (objetos) de uma classe, pois os valores de cada atributo são carregados nas instâncias (MELO, 2004). A figura seguinte mostra um exemplo de classe com atributos:

Cidade
-nome: string
-sexo: char
-data_nascimento: date

Figura 15 – Exemplo de classe com atributos

Fonte: o autor.

MÉTODOS

Métodos são processos ou serviços realizados por uma classe e disponibilizados para uso de outras classes em um sistema. Além disso, devem ficar armazenados na terceira divisão da classe. Os métodos são as atividades que uma instância de uma classe pode executar (GUEDES, 2007).

Um método pode receber, ou não, parâmetros (valores que são utilizados durante a execução do método), bem como pode, também, retornar valores, os quais podem

representar o resultado da operação executada ou somente indicar se o processo foi concluído com sucesso. Assim, um método representa um conjunto de instruções que são executadas quando o método é chamado. Um objeto da classe cliente, por exemplo, pode executar a atividade de incluir novo cliente. A Figura 16 apresenta um exemplo de uma classe com métodos:

Cliente
-nome: string
-sexo: char
-data_nascimento: date
+IncluirNovoCliente(): void
+AtualizarCliente(): void

Figura 16 – Exemplo de classe com métodos

Fonte: o autor.

VISIBILIDADE

De acordo com Guedes (2007), a visibilidade é um símbolo que antecede um atributo ou método e é utilizada para indicar o seu nível de acessibilidade. Existem, basicamente, três modos de visibilidade: o público, o protegido e o privado. Saiba mais sobre cada um nos tópicos a seguir:

- O símbolo de mais (+) indica visibilidade pública, ou seja, significa que o atributo ou método pode ser utilizado por objetos de qualquer classe.
- O símbolo de sustenido (#) indica que a visibilidade é protegida, ou seja, determina que apenas objetos da classe possuidora do atributo ou método ou de suas subclasses podem acessá-lo.
- O símbolo de menos (-) indica que a visibilidade é privada, ou seja, somente os objetos da classe possuidora do atributo ou método poderão utilizá-lo.

Note que, no exemplo da classe cliente, tanto os atributos quanto os métodos apresentam sua visibilidade representada à esquerda de seus nomes. Assim, os atributos nome, sexo e data_nascimento possuem visibilidade privada, pois apresentam o símbolo de menos (-) à esquerda da sua descrição. Já os métodos IncluirNovoCliente() e AtualizarCliente() apresentam visibilidade pública, assim como indica o símbolo +, acrescentado à esquerda de sua descrição.

HERANÇA (GENERALIZAÇÃO/ESPECIALIZAÇÃO)

A herança permite que as classes de um sistema compartilhem atributos e métodos, otimizando, assim, o tempo de desenvolvimento, com a diminuição de linhas de código, facilitando futuras manutenções (GUEDES, 2007). A herança (ou generalização/especialização) acontece entre classes gerais (chamadas de superclasses ou classes-mãe) e classes específicas (chamadas de subclasses ou classes-filha) (BOOCH; RUMBAUGH; JACOBSON, 2006).

A herança significa que os objetos da subclasse podem ser utilizados em qualquer local em que a superclasse ocorra, mas não vice-versa. A subclasse herda

as propriedades da mãe, ou seja, seus atributos e métodos, bem como pode possuir atributos e métodos próprios, além dos herdados.

De acordo com Guedes (2007), a vantagem do uso da herança é que, quando uma classe é declarada com seus atributos e métodos específicos e, após isso, uma subclasse é derivada, não é necessário redeclarar os atributos e métodos já definidos. Em outras palavras, por meio da herança, a subclasse herda tais atributos e métodos automaticamente, permitindo a reutilização do código já pronto.

Assim, é preciso somente declarar os atributos ou métodos restritos da subclasse, o que torna o processo de desenvolvimento mais ágil, além de facilitar as manutenções futuras, pois, em caso de uma alteração, será necessário somente alterar o método da superclasse, para que todas as subclasses estejam também atualizadas. A Figura 17 apresenta um exemplo de herança, explicitando os papéis de superclasse e subclasse, bem como apresenta o símbolo de herança da UML,

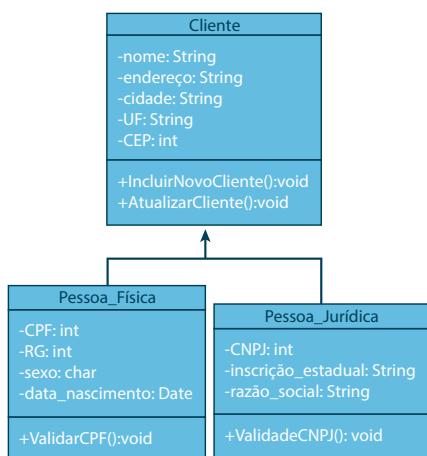


Figura 17 – Exemplo de herança

Fonte: o autor.

uma linha que liga as classes com um triângulo tocando a superclasse:

A figura apresentada mostra a superclasse cliente com os atributos nome, endereço, cidade, UF e CEP, bem como os métodos IncluirNovoCliente() e AtualizarCliente(). A subclasse Pessoa_Física herda esses atributos e métodos, além de possuir os atributos CPF, RG, sexo e data_nascimento e o método ValidarCPF(). A seta que aponta para a superclasse cliente indica a herança. A subclasse Pessoa_Jurídica também herda

todos os atributos e métodos da superclasse cliente, além de possuir os atributos CNPJ, inscrição_estadual e razão_social e o método ValidarCNPJ().

Quando olhamos para a Figura 17, notamos que a classe cliente é a mais genérica e as classes Pessoa_Física e Pessoa_Jurídica são as mais especializadas. Então, podemos afirmar que generalizamos quando partimos das subclasses para a superclasse e especializamos quando fazemos o contrário, ou seja, quando partimos superclasse para as subclasses.

POLIMORFISMO

O conceito de polimorfismo está associado com a herança, pois trabalha com a redeclaração de métodos previamente herdados por uma classe. Esses métodos, embora parecidos, diferem-se, de alguma forma, da implementação utilizada na superclasse, sendo preciso implementá-los novamente na subclasse. Todavia, a fim de não ter que alterar o código-fonte é acrescentada uma chamada a um método com um nome diferente, redeclara-se o método com o mesmo nome declarado na superclasse (GUEDES, 2007).

De acordo com Lima (2009), polimorfismo é o princípio em que classes derivadas (as subclasses) e uma mesma superclasse podem chamar métodos que têm o mesmo nome (ou a mesma assinatura), mas que possuem comportamentos diferentes em cada subclasse, produzindo resultados diferentes, dependendo de como cada objeto implementa o método.

Em outras palavras, podem existir dois ou mais métodos com a mesma nomenclatura, diferenciando-se na maneira como foram implementados. O sistema é o responsável por verificar se a classe da instância em questão possui o método declarado nella própria ou se o herda de uma superclasse (GUEDES, 2007).

Por ser um exemplo bastante claro, para ilustrar o polimorfismo, apresentamos a figura seguinte:

No exemplo apresentado, há uma classe geral chamada Conta_Comum (que, nesse caso, é a superclasse), a qual possui um atributo chamado saldo, que contém o valor total depositado em uma determinada instância da classe, e um método chamado saque. Esse método somente diminui o valor a ser debitado do saldo da conta se ele possuir o valor suficiente. Caso contrário, a operação não poderá ser realizada, ou seja, o saque deve ser recusado pelo sistema (GUEDES, 2007).

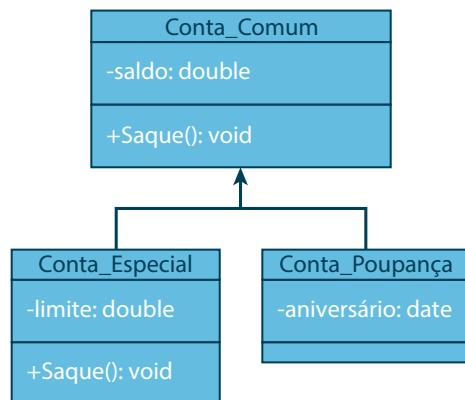


Figura 18 – Exemplo de herança
Fonte: o autor.

CONSIDERAÇÕES FINAIS

No decorrer desta unidade, estudamos a importância da modelagem de um sistema a partir do documento de requisitos. A modelagem é a parte fundamental de todas as atividades, dentro de um processo de software, as quais levam à implantação de um bom software. Esses modelos, normalmente, são representados por meio de diagramas em que se é utilizada uma notação gráfica, que, em nosso caso, foi a UML.

A UML tem muitos tipos de diagramas, o que apoia a criação de diferentes modelos de sistema. No entanto, conhecemos, nesta unidade, somente o diagrama de casos de uso e o diagrama de classes.

Além disso, a UML não estabelece uma ordem pré-definida para a elaboração dos seus diversos diagramas. Todavia, em consequência do fato de que o diagrama de casos de uso é um dos mais gerais e informais da UML, normalmente, a modelagem do sistema se inicia com a elaboração desse diagrama.

O diagrama de casos de uso mostra as interações entre um sistema e seu ambiente externo, determinando as funcionalidades e as características do sistema sob o ponto de vista do usuário. Além do mais, conhecemos, nesta unidade, os conceitos necessários para a elaboração desse diagrama: atores, casos de uso e os possíveis relacionamentos entre esses elementos.

Depois que o diagrama de casos de uso é elaborado, fica bem mais fácil elaborar o diagrama de classes, que é o mais importante e o mais utilizado da UML. Ele é quem define a estrutura das classes identificadas para o sistema, mostrando os atributos e os métodos possuídos por cada uma das classes, bem como os seus relacionamentos.

Para auxiliar no entendimento desses dois diagramas, foi apresentada a elaboração passo a passo de cada um deles. Para verificar se você realmente entendeu todos os conceitos, proponho mais um documento de requisitos nas atividades de estudo. Então, mãos à obra!

ATIVIDADES



1. Sobre o relacionamento entre casos de uso e atores:
 - a) Explique o relacionamento de associação (*association*).
 - b) Explique o relacionamento de generalização entre casos de uso (*generalization*).
2. Com base no documento de requisitos, a seguir, elabore o diagrama de casos de uso:

Sistema Controle de Cinema

Um cinema pode ter muitas salas, o que exige, portanto, o registro de informações a respeito de cada uma, tais como a sua capacidade, ou seja, o número de assentos disponíveis.

Além disso, o cinema apresenta muitos filmes. Um filme, por sua vez, tem informações como título e duração. Assim, sempre que um filme for apresentado, ele também deve ser registrado.

Ademais, um mesmo filme pode ser apresentado em diferentes salas e em horários diferentes. Cada apresentação em uma determinada sala e horário é chamado sessão. Um filme, quando apresentado em uma sessão, tem um conjunto máximo de ingressos, o qual é determinado pela capacidade da sala.

Os clientes do cinema podem comprar, ou não, ingressos para assistir à sessão. Assim, o funcionário deve intermediar a compra do ingresso. Um ingresso deve conter informações, como o tipo de ingresso (meio ingresso ou ingresso inteiro), por exemplo. Além disso, um cliente só pode comprar ingressos para sessões ainda não encerradas.

3. Um caso de uso especifica o comportamento de um sistema ou de parte de um, referindo-se a serviços, tarefas ou funções apresentadas pelo sistema. Baseado no conceito apresentado, analise os casos de uso a seguir e desenhe o diagrama de caso de uso correspondente:

- A secretaria emite o relatório de consultas realizadas e o envia para o médico.
- A secretaria emite o relatório de consultas agendadas e o envia para o médico.

ATIVIDADES



4. Dado um sistema de laboratório de análises clínicas, percebemos, em nossa análise, que ele precisa ter os cadastros para que a secretaria consiga inserir os dados no sistema, a fim de ser possível emitir os relatórios. Baseado nessas informações, elabore um diagrama de casos de uso em que a personagem secretária irá realizar os seguintes casos:

- a) Cadastrar pacientes.
- b) Cadastrar exames.
- c) Cadastrar pedidos de exame.
- d) Cadastrar resultados dos exames.
- e) Cadastrar médicos.
- f) Cadastrar convênios.
- g) Cadastrar cidades.
- h) Cadastrar UF's.

5. Os símbolos menos (-) e mais (+), na frente dos atributos e métodos, representam a sua visibilidade. A visibilidade, por sua vez, é utilizada para indicar o nível de acessibilidade de um determinado atributo ou método, sendo sempre representada à esquerda.

A partir da informação apresentada, descreva os principais os três modos de visibilidade utilizados na UML.



Casos de uso são definidos para satisfazer aos objetivos dos atores principais. Assim, o procedimento básico é:

1. Escolher a fronteira do sistema. Ele é somente uma aplicação de software, é o hardware e a aplicação como uma unidade, é isso e mais uma pessoa usando o sistema ou é toda uma organização?
2. Identificar os atores principais – aqueles que têm objetivos satisfeitos por meio do uso dos serviços do sistema.
3. Identificar os objetivos para cada ator principal.
4. Definir casos de uso que satisfaçam os objetivos dos usuários; nomeie-os de acordo com o objetivo. Geralmente, os casos de uso no nível de objetivo do usuário estarão em uma relação de um-para-um com os objetivos dos usuários, mas existe pelo menos uma exceção que será examinada.

Certamente, em um desenvolvimento iterativo e evolutivo, nem todos os objetivos ou casos de uso vão estar total ou corretamente identificados logo no início. Trata-se de uma descoberta evolutiva.

Passo 1: escolher a fronteira do sistema

Se não estiver claro, a definição da fronteira do sistema que está sendo projetado pode ser esclarecida definindo-se o que está de fora – os atores principais e os atores de suporte externos. Uma vez que os atores externos tenham sido identificados, a fronteira se torna mais clara. Por exemplo, a responsabilidade completa pela autorização de pagamento está dentro da fronteira do sistema? Não, existe um ator que é um serviço externo de autorização de pagamento.

Passos 2 e 3: encontrar atores principais e objetivos

É artificial linearizar estritamente a identificação dos atores principais antes dos objetivos de usuário; em um workshop sobre requisitos, as pessoas fazem reuniões especulativas (brainstorms) e geram uma mistura de ambos. Às vezes, os objetivos revelam os atores ou vice-versa.

Diretriz: enfatize reuniões especulativas primeiro, para encontrar os atores principais, pois isso estabelece a estrutura para investigações adicionais.



Existem questões para ajudar a encontrar atores e objetivos?

Além dos atores principais e objetivos óbvios, as seguintes perguntas ajudam a identificar outros que podem ter passado despercebidos:

- Quem ativa e para o sistema?
- Quem faz a administração de usuários e da segurança? Existe um processo de monitoração que reinicia o sistema em caso de falha?
- Como são tratadas as atualizações de software? As atualizações são do tipo forçadas ou negociadas?
- Além dos atores principais humanos, existe algum sistema externo de software ou robótica que solicita os serviços do sistema?
- Quem faz a administração do sistema?
- O “tempo” é um ator porque o sistema faz algo em resposta a um evento temporal? Quem avalia a atividade ou desempenho do sistema?
- Quem avalia os registros? Eles são recuperados remotamente?
- Quem é notificado quando há erros ou falhas?

Como organizar os atores e objetivos?

Existem pelo menos duas abordagens:

1. À medida que você descobre os resultados, desenhe-os em um diagrama de caso de uso, nomeando os objetivos como casos de uso.
2. Escreva primeiro uma lista de atores/objetivos, reveja-a e refine-a, e depois deseñe o diagrama de casos de uso.

Se você criar uma lista de atores/objetivos, então, em termos de artefatos do PU, ela pode ser uma seção no artefato Visão.

Fonte: Larman (2007, p. 109-110).

MATERIAL COMPLEMENTAR



LIVRO

UML – Guia do Usuário

Grady Booch, James Rumbaugh e Ivar Jacobson

Editora: Campus

Sinopse: Há quase dez anos, a Unified Modeling Language (UML) é o padrão para visualizar, especificar, construir e documentar os artefatos de um sistema de software. A UML 2.0 vem para assegurar a contínua popularidade e viabilidade da linguagem. Sua simplicidade e expressividade permitem que os usuários modelem tudo, desde sistemas empresariais de informação, passando por aplicações distribuídas baseadas na Web, até sistemas embutidos de tempo real. Nesta nova edição totalmente revista, os criadores da linguagem fornecem um tutorial dos aspectos centrais da UML. Começando com um modelo conceitual da UML, o livro aplica progressivamente a linguagem a uma série de problemas de modelagem cada vez mais complexos usando diversos domínios de aplicação. A abordagem em tópicos e recheada de exemplos que fez da primeira edição um recurso indispensável continua a mesma. Entretanto, o conteúdo foi totalmente revisto e reflete a mudança na notação e no uso exigido pela UML 2.0.



NA WEB

Vídeo que mostra uma introdução aos conceitos de orientação a objetos.

Web: <http://www.youtube.com/watch?v=MnJLeYAno4o&feature=relmfu>.



NA WEB

Vídeo que mostra a importância da modelagem de sistemas, bem como trata da elaboração do diagrama de casos de uso.

Web: <http://www.youtube.com/watch?v=hfN6n5fJfLc&feature=relmfu>.

REFERÊNCIAS

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **UML**: Guia do Usuário. 2. ed. São Paulo: Campus, 2006.

GUEDES, G. T. A. **UML 2**: guia prático. São Paulo: Novatec, 2007.

LARMAN, C. **Utilizando UML e padrões**: uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo. 3. ed. Porto Alegre: Bookman, 2007.

LIMA, A. S. **UML 2.0**: do requisito à solução. São Paulo: Érica, 2009.

MELO, A. C. **Desenvolvendo aplicações com UML 2.0**: do conceitual à implementação. Rio de Janeiro: Brasport, 2004.

SILVA, R. P. e. **UML 2 em modelagem orientada a objetos**. Florianópolis: Visual Books, 2007.

SOMMERRVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.

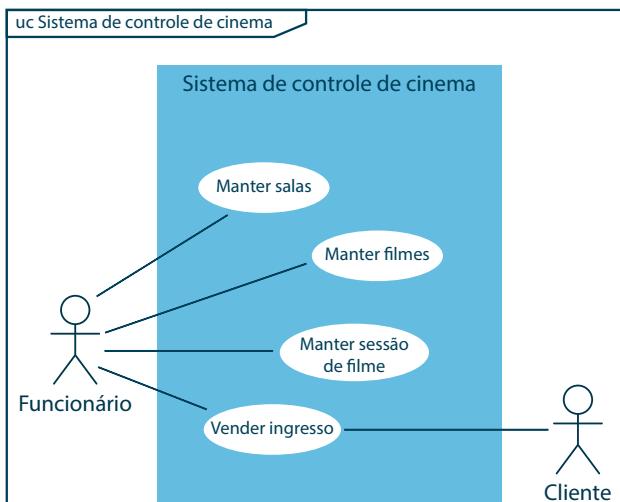


GABARITO

1.

- a) É um relacionamento que conecta duas ou mais classes, demonstrando a colaboração entre as instâncias de classe. Pode-se, também, ter um relacionamento de uma classe com ela mesma; neste caso, há uma associação unária ou reflexiva.
- b) Relacionamento que ocorre quando dois ou mais casos de uso possuem características semelhantes.

2. Sistema de controle de cinema:



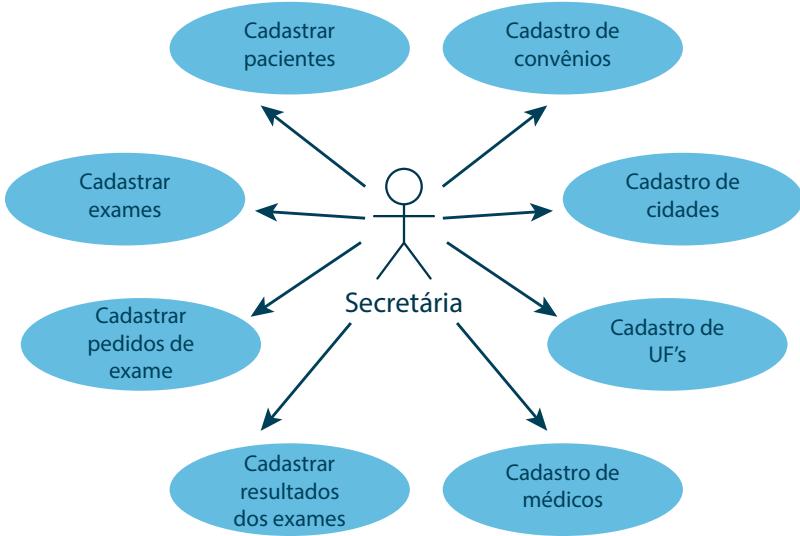
3. Resposta:

Este diagrama deverá ter como atores: secretária e o médico. Além disso, deverá apresentar, enquanto casos de uso: relatório de consultas realizadas e relatório de consultas agendadas. Pelo fato de que a secretária os emite e os envia para o médico, esse envio deve ser simbolizado pelas flechas. A seguir, segue um modelo do diagrama esperado:



GABARITO

4.



5. Os três modos de visibilidade são: o público, o protegido e o privado.

- O símbolo de mais (+) indica visibilidade pública, ou seja, significa que o atributo ou método pode ser utilizado por objetos de qualquer classe.
- O símbolo de sustenido (#) indica que a visibilidade é protegida, ou seja, determina que apenas objetos da classe possuidora do atributo ou método ou de suas subclasses podem acessá-lo.
- O símbolo de menos (-) indica que a visibilidade é privada, ou seja, somente os objetos da classe possuidora do atributo ou método poderão utilizá-lo.



DIAGRAMA DE SISTEMAS

Objetivos de Aprendizagem

- Entender a estrutura do sistema por meio de elementos, tais como classes, atributos e associações entre os objetos.
- Compreender a sequência de processos que estão inseridos em um software.
- Entender os estados e as situações que um objeto pode passar durante a execução dos processos em um sistema.
- Mostrar o diagrama de atividades e representar do fluxo de controle entre as atividades descritas no sistema.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Diagrama de classes
- Diagrama de sequência
- Diagrama de máquina de estados
- Diagrama de atividades

INTRODUÇÃO

Caro(a) aluno(a), na terceira unidade, estudaremos os conceitos básicos de UML e a respeito do diagrama de casos de uso. Agora, chegou a hora de aplicarmos os reflexos dos conhecimentos adquiridos.

A modelagem de sistema é o processo de elaboração de modelos abstratos de um sistema, normalmente representado por meio de um diagrama. No diagrama, cada um desses modelos apresenta uma visão ou perspectiva diferente do sistema (SOMMERRVILLE, 2011).

Da mesma forma que os arquitetos elaboram plantas e projetos para serem usados para a construção de um edifício, os engenheiros de software criam os diagramas UML para auxiliarem os desenvolvedores de software a construir o software.

Nesta unidade, veremos os diagramas de classes, de sequência, de máquina de estados e de atividades. Primeiramente, será apresentado, caro(a) aluno(a), o diagrama de classe, o qual é muito importante e o mais utilizado da UML, pois serve de apoio para a maioria dos demais. O diagrama de classes define a estrutura das classes identificadas para o sistema, mostrando os atributos e métodos possuídos por cada uma das classes, além de estabelecer como as classes se relacionam e trocam informações entre si. A seguir, estudaremos o diagrama de sequência, o qual atua no comportamento do sistema, a fim de evidenciar a sequência de eventos que ocorrem em um processo (GUEDES, 2011).

O próximo diagrama a ser estudado é o de máquina de estados. Nosso intuito é o de aprendermos a respeito de como é possível modelar o comportamento dos elementos por meio de transições de estados. Por último, trataremos sobre o diagrama de atividades, que, por sua vez, tem o objetivo de demonstrar o algoritmo (condições) da UML.

Mesmo abordando todos esses diagramas, ainda não conseguiremos englobar todos os existentes. Contudo, se você quiser conhecer os demais, consulte alguns livros relacionados, os quais são apresentados nas referências, ao final desta unidade.

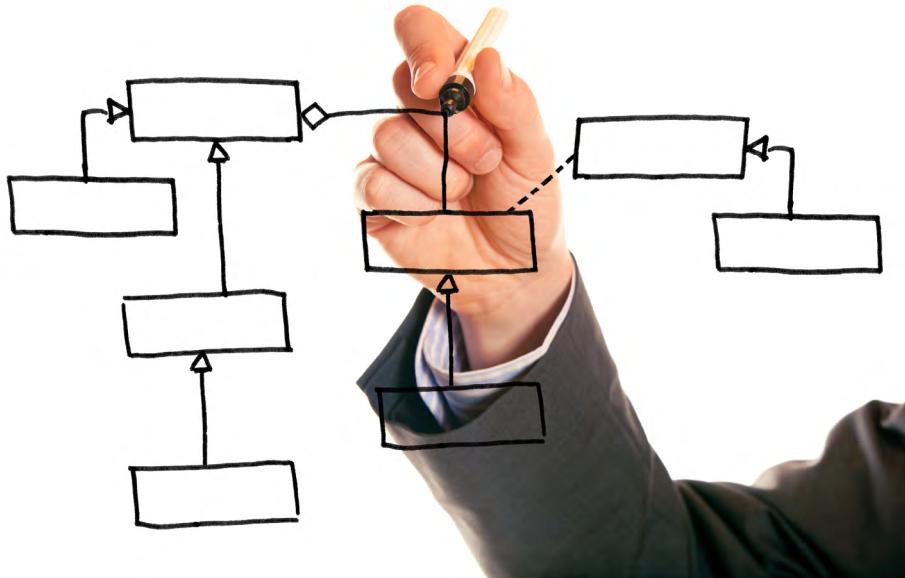


DIAGRAMA DE CLASSES

O diagrama de classes tem, como objetivo, permitir a visualização das classes utilizadas pelo sistema e como elas se relacionam, apresentando uma visão estática de como estão organizadas, preocupando-se apenas em definir sua estrutura lógica (GUEDES, 2007). Ainda, de acordo com o estudioso, um diagrama de classes pode ser utilizado para modelar o modelo lógico de um banco de dados, parecendo-se, nesse caso, com o diagrama de entidade-relacionamento (o modelo entidade-relacionamento o qual você estuda na disciplina de banco de dados). No entanto, deve ficar bem claro que o diagrama de classes não é utilizado unicamente para essa finalidade e que uma classe não necessariamente corresponde a uma tabela.

Uma classe pode representar o repositório lógico dos atributos de uma tabela, mas a classe não é a tabela. Isso se deve, uma vez que os atributos de seus objetos são armazenados em memória, enquanto uma tabela armazena seus registros fisicamente em disco. Além disso, uma classe possui métodos que não existem em uma tabela.

RELACIONAMENTOS

As classes não trabalham sozinhas. Pelo contrário, elas colaboram umas com as outras, por meio de relacionamentos (MELO, 2004). No diagrama de classes, temos alguns tipos de relacionamentos, como o de associação (que pode ser unária ou binária), generalização ou de agregação, por exemplo. Na sequência, detalharemos a respeito desses e de outros tipos de relacionamentos.

ASSOCIAÇÃO

De acordo com Melo (2004), a associação é um relacionamento que conecta duas ou mais classes, demonstrando a colaboração entre as instâncias de classe. Pode-se, também, haver um relacionamento de uma classe com ela mesma e, nesse caso, tem-se uma associação unária ou reflexiva. A seguir, apresentamos um exemplo de uma associação entre classes:



Figura 1 – Exemplo de associação

Fonte: o autor.

ASSOCIAÇÃO UNÁRIA OU REFLEXIVA

Segundo Guedes (2007), a associação unária ocorre quando existe um relacionamento de uma instância de uma classe com instâncias da mesma classe, conforme ilustra a Figura 2, a seguir:

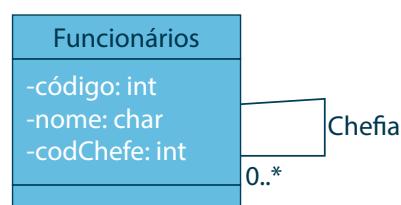


Figura 2 – Exemplo de associação unária

Fonte: o autor.

No exemplo apresentado, temos a classe funcionário, cujos atributos são código, nome e o código do possível chefe do funcionário. Pelo fato de que esse chefe também é um funcionário, a associação chamada chefia indica uma possível relação entre uma ou mais instâncias da classe funcionário com outras instâncias da mesma classe, ou seja, tal associação determina que um funcionário pode ou não chefiar outros funcionários. Além disso, essa associação faz com que a classe possua o atributo codChefe, para armazenar o código do funcionário, que é o responsável pela instância do funcionário em questão. Desse modo, após consultar uma instância da classe funcionário, pode-se utilizar o atributo codChefe da instância consultada para pesquisar por outra instância da classe (GUEDES, 2007).

ASSOCIAÇÃO BINÁRIA

A associação binária conecta duas classes por meio de uma reta que liga uma classe a outra. A Figura 3 demonstra um exemplo de associação binária:

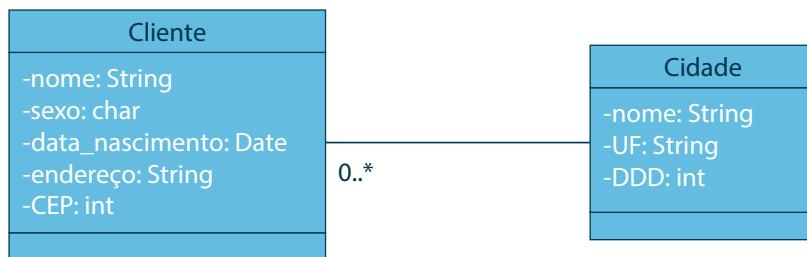


Figura 3 – Exemplo de associação binária
Fonte: o autor.

No exemplo, um objeto da classe cidade pode se relacionar, ou não, com instâncias da classe cliente, conforme demonstra a multiplicidade **0..***. Entretanto, se existir um objeto da classe cliente, ele terá que se relacionar com um objeto da classe cidade, pois, pelo fato de que não foi definida a multiplicidade na extremidade da classe cidade, isso significa que ela é **1..1**. Após termos explicado os relacionamentos em um diagrama de classes, explanaremos o conceito de multiplicidade.

AGREGAÇÃO

Uma agregação pode ocorrer somente entre duas classes. De acordo com as regras da UML, esse tipo de relacionamento ou associação é identificável a partir da notação de uma linha sólida entre duas classes: a classe denominada todo-agregado recebe um diamante vazio, enquanto a outra ponta da linha é ligada à classe denominada parte-constituinte. Ambas as classes podem “viver” de forma independente, ou seja, não existe uma ligação forte entre as duas. Objetos da parte constituinte ou da parte agregado são independentes em termos de vida, mas ambas são partes de um único todo. Essa análise dependerá do domínio do problema em estudo.

Para sabermos se um relacionamento é de agregação, basta perguntarmos se, em primeiro lugar, comporta a estrutura todo-parte. Em seguida, questionamo-nos se o objeto constituinte-parte “vive” sem o objeto agregado. Se a resposta for sim para ambas as perguntas, teremos uma agregação. A figura seguinte apresenta a notação para essa semântica. Para um determinado domínio de problema que trata de apartamentos e condomínios, observamos que um apartamento tem depósitos. Quando nos referimos a um depósito, podemos perguntar a qual apartamento aquele depósito pertence. Por outro lado, um determinado proprietário pode ter decidido sobre a venda de um depósito para alguém que sequer mora no prédio. Assim, teremos os dois objetos, neste exemplo, vivendo separadamente, sem incômodo algum.

Poderíamos ter a situação em que um apartamento é interditado ou passa por uma grande reforma. Durante algum tempo, não teremos movimentação naquele apartamento, mas o depósito que faz parte daquele jogo apartamento-depósito continua sendo usado livremente. Assim, o mesmo exemplo valeria para garagens de um apartamento:

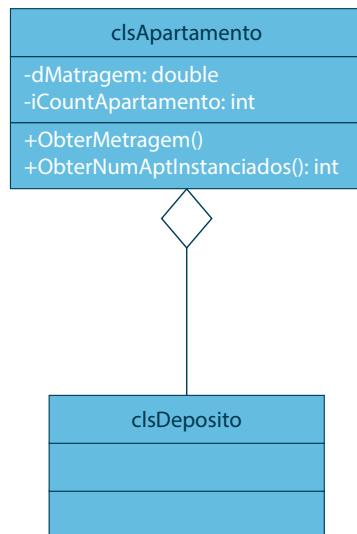


Figura 4 – Uma agregação entre duas classes
Fonte: o autor.

COMPOSIÇÃO

Uma composição ocorre quando temos uma situação semelhante à da agregação entre duas classes, mas os objetos da classe parte não podem viver quando o todo é destruído. Esse tipo de relacionamento é identificável pela notação de uma linha sólida entre duas classes: a classe denominada todo-composta, a qual recebe um diamante preenchido, enquanto a outra ponta da linha é ligada à classe denominada parte-componente.

Ambas as classes “vivem” unidas de forma dependente, ou seja, existe uma ligação forte entre as duas. Os objetos-parte não podem ser destruídos por um objeto diferente do objeto-todo ao qual estão relacionados (GUEDES, 2011). Essa análise dependerá do domínio do problema em estudo. Para sabermos se um relacionamento é de composição, basta perguntarmos se, em primeiro lugar, cabe a estrutura todo-part. Em seguida, questionamos se o objeto parte “vive” sem o objeto todo. Se a resposta for sim para a primeira pergunta e não para segunda, teremos uma composição.

A Figura 5 apresenta uma visão da notação para esse tipo de relacionamento. Para esse domínio de problema, precisamos pensar no prédio como um todo. Isso parece plausível, pois, nesse domínio de problema, necessariamente, um prédio deveria existir para que haja um apartamento. Caso não pudesse qualificar um prédio, nada poderia ser feito em relação àquele apartamento. Não se pode fazer nada com um apartamento, vender ou alugar, enquanto ele não possuir um endereço. O endereço é do prédio, o apartamento possui apenas complemento. Assim, para os fins desse software, não se conseguiria instanciar um objeto `clsApartamento`, caso não fosse designado um objeto `clsPredio`:

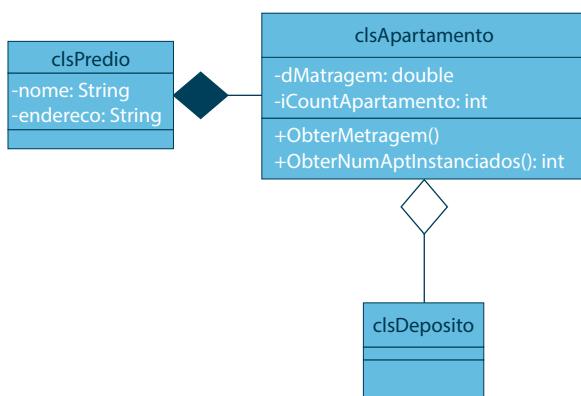
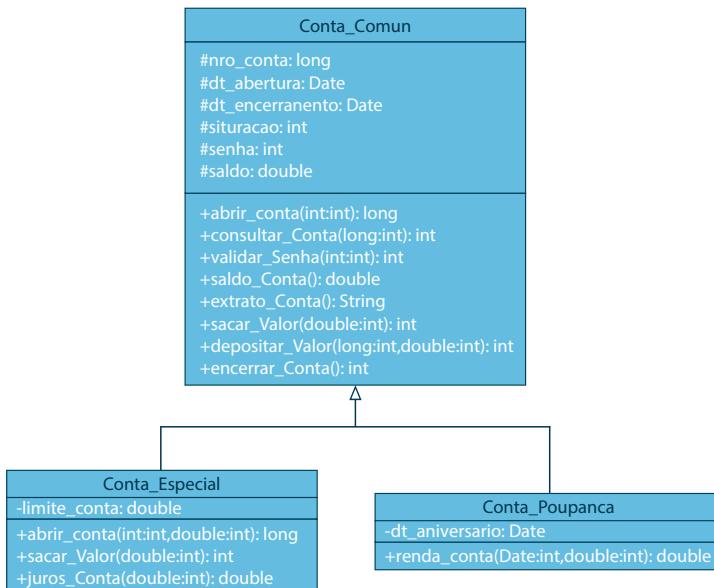


Figura 5 - Composição entre duas classes
Fonte: o autor.

GENERALIZAÇÃO/ESPECIALIZAÇÃO

Essa associação identifica as classes-mãe (superclasse), as quais são chamadas gerais, e as classes-filhas (subclasses), as chamadas especializadas, demonstrando a ocorrência de herança e, possivelmente, de métodos polimórficos nas classes especializadas. A seguir, segue um exemplo entre classes, o qual demonstra o uso de generalização/especialização:



Reprodução proibida. Art. 184 do Código Penal e Lei 9.610 de 19 de fevereiro de 1998.

Figura 6 - Composição entre duas classes

Fonte: Guedes (2011, p. 114).

MULTIPLICIDADE

De acordo com Guedes (2007), a multiplicidade determina o número mínimo e máximo de instâncias envolvidas em cada uma das extremidades da associação, permitindo, também, especificar o nível de dependência de um objeto com os outros. Quando não existe multiplicidade explícita, entende-se que a multiplicidade é “1..1”, o que significa que somente uma instância dessa extremidade

da associação relaciona-se com as instâncias da outra extremidade. A tabela a seguir demonstra alguns dos diversos valores de multiplicidade que podem ser utilizados em uma associação:

Tabela 1 — Exemplos de multiplicidade

MULTIPLICIDADE	SIGNIFICADO
0..1	No mínimo zero (nenhum) e no máximo um. Indica que os objetos das classes associadas não precisam obrigatoriamente estar relacionadas, mas se houver relacionamento indica que apenas uma instância da classe se relaciona com as instâncias da outra classe.
1..1	Um e somente um. Indica que apenas um objeto da classe se relaciona com os objetos da outra classe.
0..*	No mínimo nenhum e no máximo muitos. Indica que pode ou não haver instância da classe participando do relacionamento.
*	Muitos. Indica que muitos objetos da classe estão envolvidos no relacionamento
1..*	No mínimo 1 e no máximo muitos. Indica que há pelo menos um objeto envolvido no relacionamento, podendo haver muitos objetos envolvidos.
2..5	No mínimo 2 e no máximo 5. Indica que existe pelo menos 2 instâncias envolvidas no relacionamento e que pode ser 3, 4 ou 5 as instâncias envolvidas, mas não mais do que isso.

Fonte: Guedes (2011, p. 108).

As associações que possuem multiplicidade do tipo muitos (*), em qualquer de suas extremidades, forçam a transmissão do atributo-chave contido na classe da outra extremidade da associação. Entretanto, esse atributo-chave não aparece desenhado na classe.

CLASSE ASSOCIATIVA

Quando um relacionamento possui multiplicidade, ou seja, muitos (*) em suas duas extremidades, é necessário criar uma classe associativa para guardar os objetos envolvidos nessa associação. Na classe associativa, são definidos o conjunto de

atributos que participam da associação, e não as classes que participam do relacionamento. Nesse caso, pelo menos os atributos-chave devem fazer parte da classe associativa, mas a classe associativa também pode possuir atributos próprios.

Uma classe associativa é representada por uma reta tracejada que parte do meio da associação e atinge uma classe. A Figura 7 apresenta um exemplo de classe associativa que possui atributos próprios, além dos atributos-chave das classes que participam da associação. Contudo, é necessário reforçar que, nesse caso, esses atributos-chave não são representados no diagrama:

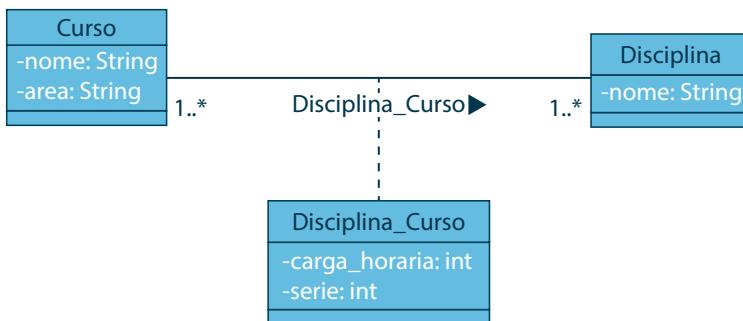


Figura 7 – Exemplo de classe associativa

Fonte: o autor.

No exemplo apresentado, uma instância da classe curso pode se relacionar com muitas instâncias da classe disciplina, bem como uma instância da classe disciplina pode se associar com muitas instâncias da classe curso. Como existe a multiplicidade nas extremidades de ambas as classes da associação, não há como reservar atributos para armazenar as informações decorrentes da associação, já que não é possível determinar um limite para a quantidade de disciplinas que um curso pode ter e nem como saber em quantos cursos uma disciplina pode pertencer. Assim, é preciso criar uma classe para guardar essa informação, além das informações próprias da classe, que são a carga horária da disciplina no curso e a qual série essa disciplina estará vinculada. Os atributos-chave das classes curso e disciplina são transmitidos de forma transparente pela associação, não sendo necessário, portanto, escrevê-los como atributos da classe associativa.

Segundo Guedes (2007), as classes associativas podem ser substituídas por classes normais, que, nesse caso, são chamadas de classes intermediárias, mas que desempenham exatamente a mesma função das classes associativas. A figura

seguinte mostra o mesmo exemplo da Figura 7, porém com a classe intermediária ao invés da classe associativa:

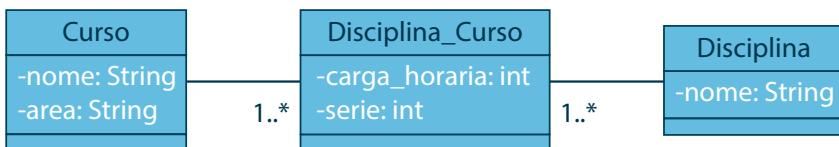


Figura 8 – Exemplo de classe intermediária

Fonte: o autor.

PASSO A PASSO PARA A ELABORAÇÃO DO DIAGRAMA DE CLASSES

1. Como você já fez o diagrama de casos de uso na unidade III, vai ficar mais fácil elaborar o diagrama de classes. Com base nos casos de uso que você definiu, você vai fazer uma lista das possíveis classes do sistema. Lembre-se que os relatórios não serão a classe por si só, mas, para emitir cada relatório, utilizaremos diversas classes. Uma dica: se o diagrama possui um caso de uso de cadastro, certamente, precisará de uma classe para armazenar os dados que serão cadastrados nesse caso de uso. Seguindo esse raciocínio, teríamos as seguintes classes:
 - a. Paciente.
 - b. Exame.
 - c. Pedido de Exame.
 - d. Resultado de Exame.
 - e. Médicos.
 - f. Convênios.
 - g. Cidades.
 - h. UF's.
 - i. Grupos de Exames.

2. Agora, para cada uma das classes listadas, relate os possíveis atributos de cada uma delas. A maioria desses atributos já aparece descrita no documento de requisitos. Nunca se esqueça de voltar ao documento de requisitos sempre que tiver dúvidas.
 - a. Paciente: código, nome, endereço, CEP, cidade, UF, telefone, data de nascimento, RG e CPF.
 - b. Exame: código, descrição, valor, procedimentos e grupo ao qual pertence o exame.
 - c. Pedido de exame: código, nome do paciente, nome do médico, nome do convênio, nomes dos exames que serão realizados, data e hora da realização de cada exame, data e hora em que cada exame ficará pronto, valor de cada exame.
 - d. Resultado de exame: descrição do resultado (para cada exame do pedido, o resultado deverá ser cadastrado).
 - e. Médicos: CRM, nome (como o documento de requisitos não menciona nada sobre os dados dos médicos, coloquei somente os atributos que interessam para o pedido de exame).
 - f. Convênios: código, nome (como o documento de requisitos não menciona nada sobre os dados dos convênios, coloquei somente os atributos que interessam para o pedido de exame).
 - g. Cidades: código, nome, DDD (o documento de requisitos não mencionou nada sobre, mas esses atributos devem constar em qualquer classe de cidades).
 - h. UF's: sigla, nome.
 - i. Grupos de exames: código, descrição.
 - j. Desenhar as classes relacionadas com seus respectivos atributos no diagrama de classes. Faremos o desenho do diagrama utilizando a ferramenta Astah e no mesmo arquivo em que já desenhamos o diagrama de casos de uso. Assim, ficaremos com os dois diagramas em um único arquivo.

3. Para cada classe desenhada no diagrama, estabeleça o seu relacionamento com as demais. Lembre-se dos tipos de relacionamentos que estudamos: associação (únaria e binária), generalização/especialização, agregação. Releia, também, a explicação sobre classes associativas.
4. Depois disso, estabeleça a multiplicidade de cada relacionamento, lembrando-se de eliminar os atributos que podem ser obtidos por meio do relacionamento.
5. Primeiro, desenhe o seu diagrama de classes e, depois, compare-o com o meu. Veja como ficou o meu diagrama:

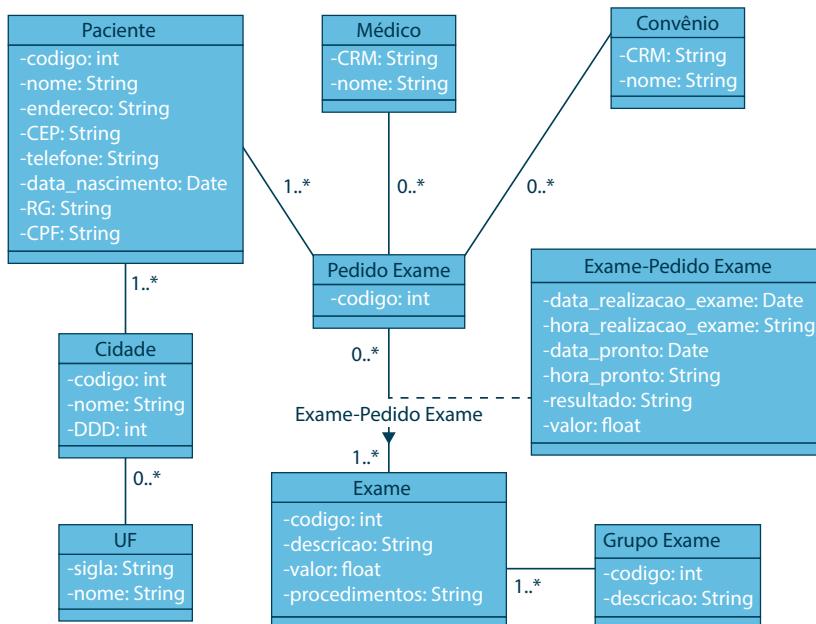


Figura 9 – Diagrama de classes do laboratório de análises clínicas

Fonte: o autor.

Seguem alguns esclarecimentos sobre o diagrama representado na Figura 9:

1. Um pedido de exame pode estar relacionado somente um paciente, um médico e um convênio (no diagrama, não aparece a multiplicidade 1, por ser o valor *default* de um relacionamento).

2. Um pedido de exame pode estar relacionado a um ou a vários exames (no caso desse documento de requisitos, a cinco exames).
3. Note que os atributos `data_realizacao_exame`, `hora_realizacao_exame`, `data_pronto`, `hora_pronto`, `resultado` e `valor` estão armazenados na classe associativa que foi originada do relacionamento muitos para muitos entre pedido de exame e exame. Isso se deve ao fato de que, para cada exame, esses atributos podem ser diferentes. Por exemplo: se o atributo `data_pronto` tivesse sido armazenado na classe `pedido_exame`, seria possível cadastrar somente uma data em que os exames ficariam prontos. Todavia, na realidade, não é isso o que acontece, ou seja, em uma lista de exames que o paciente precisa realizar, pode-se ter exames que ficam prontos em dois dias e os que ficam prontos em cinco.
4. Veja que não foi criada uma classe `resultado_exame`, pois como é somente uma descrição, decidiu-se armazená-la na classe associativa `exame-pedido_exame`.
5. Note, também, que, na classe `pedido_exame`, não aparece o nome do paciente, assim como relacionamos no item 2 desse passo a passo. Isso, porque o nome será obtido por meio do relacionamento de `pedido_exame` com `paciente`. Não desenhamos o atributo-chave de paciente na classe `pedido_exame`, mas ele está lá, ou seja, por meio dele é que buscaremos o nome do paciente na classe `paciente` quando precisarmos.
6. Ao invés de termos utilizado o recurso da classe associativa, poderíamos ter usado o relacionamento de agregação. Veja como ficaria (serão mostradas somente algumas classes; as demais não foram mostradas, pois ficariam iguais ao diagrama já apresentado):

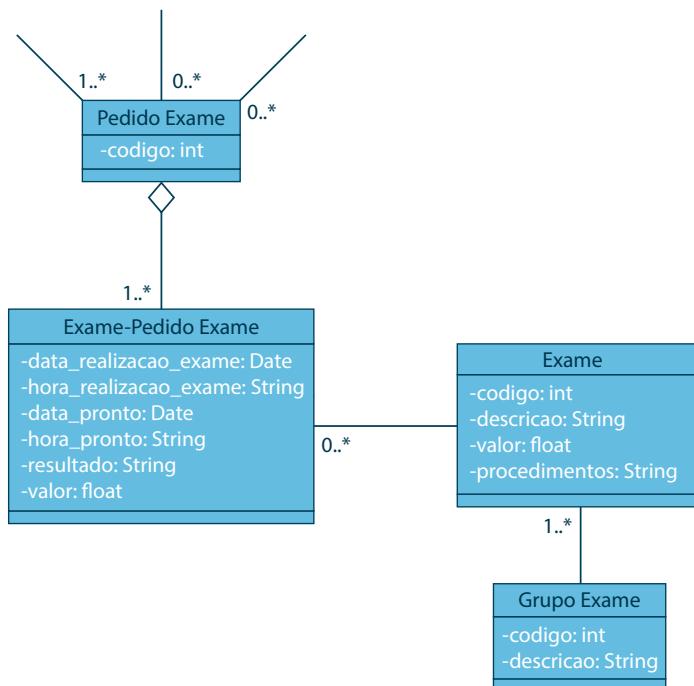


Figura 10 – Diagrama de classes associativas

Fonte: o autor.



SAIBA MAIS

Por que a construção de software não apresenta a mesma constância que outras áreas?

Quando você pensa em construir uma casa, um prédio, um navio, enfim, qualquer obra de engenharia civil, naval ou eletrônica, inicia-se com uma planta. Engenheiros, arquitetos e, eventualmente, mestres de obras coloam à sua disposição anos de trabalho e consequente experiência. Nada se inicia, em termos de construção, antes que a concepção do projeto esteja terminada. Por anos, tentamos construir software tendo como termos de comparação a construção civil. O problema é que os requisitos de um software sofrem mudanças. Elas ocorrem porque o interessado no software passa pelas mudanças diárias, não importando o tamanho do software.

Fonte: Medeiros (2004, p. 2).

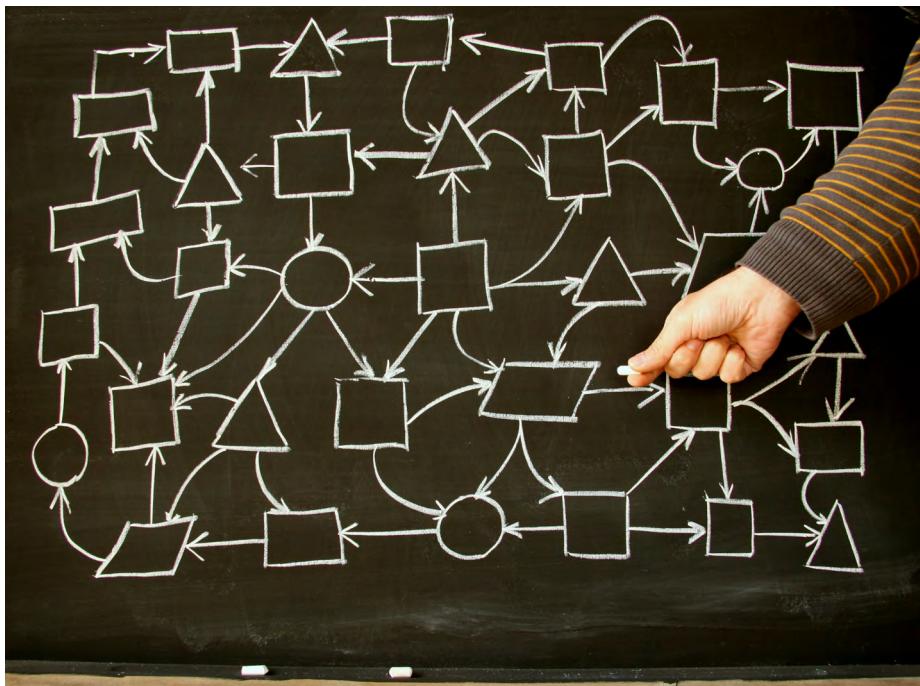


DIAGRAMA DE SEQUÊNCIA

Segundo Guedes (2011, p. 192), “o diagrama de sequência procura determinar a sequência de eventos que ocorrem em um determinado processo, identificando quais mensagens devem ser disparadas entre os elementos envolvidos e em que ordem”.

A criação do diagrama de sequência pode ser baseada no diagrama de caso de uso e no diagrama de classes. Em relação ao diagrama de caso de uso, é importante saber que cada caso especificado refere-se a um processo disparado por um ator, o que permite a documentação do referido caso. Já em relação ao uso do diagrama de classe, as classes contidas nele se tornam objetos no diagrama de sequência.

A partir do conceito abordado, é importante conhecermos os elementos que podem compor o diagrama de sequência.

COMPOSIÇÃO DO DIAGRAMA DE SEQUÊNCIA

Atores

Os atores são usuários ou pessoas do meio externo que participam, tendo algum papel dentro do sistema. Dentro do diagrama de sequência, um ator pode ser aproveitado com base no diagrama de casos de uso, gerando, assim, eventos iniciais dos processos (GUEDES, 2011). A representação de um ator se dá por meio de bonecos idênticos aos do diagrama de casos de uso, mas contendo uma linha de vida logo aabaixo. A seguir, temos a Figura 11, que representa o ator atendente:



Figura 11 – Exemplo de atendente
Fonte: o autor.

Linha de vida (*lifelines*)

Trata-se de uma linha pontilhada que representa o tempo que um objeto existe durante um processo (GUEDES, 2011). A linha de vida pode estar ligada a objetos, atores e entre outros. É um exemplo:



Figura 12 – Exemplo de linha de vida
Fonte: o autor.

A linha de vida é composta por uma instância de uma classe que participa de uma interação. Em geral, uma linha de vida é constituída por um objeto, o qual pode ser um ator, uma classe etc., seguido de uma linha de vida (GUEDES, 2011), assim como se pode visualizar no exemplo a seguir:

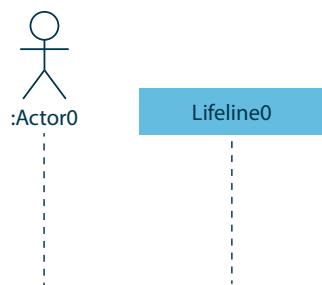


Figura 13 – Exemplo de *lifeline* ator e classe
Fonte: o autor.

Quando é necessário destruir um objeto, a linha da vida é interrompida por um X. A seguir, temos um exemplo:



Figura 14 – Exemplo de *lifeline* interrompida
Fonte: o autor.

Foco de controle ou ativação

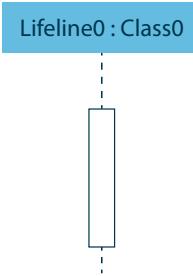


Figura 15 – Exemplo de foco de controle
Fonte: o autor.

Para representar os momentos ativos em que um objeto está executando um ou mais métodos, é utilizado o foco de controle. Para representar o momento, é utilizada uma linha mais grossa sobre a linha pontilhada (GUEDES, 2011). Na Figura 15, segue a sua representação:

Mensagens

As mensagens, dentro do diagrama de sequência, têm o objetivo de mostrar a ocorrência de eventos. Elas normalmente forçam a chamada de um método entre os objetos envolvidos no processo (GUEDES, 2011). Partindo desse conceito, temos, a seguir, alguns exemplos de ocorrência de mensagens entre objetos:

MENSAGEM	EXEMPLO
Um Ator e outro Ator	<p>Figura 16 – Mensagens simples entre atores</p>

MENSAGEM	EXEMPLO
Um Ator e um Objeto	<pre> sequenceDiagram participant Atendente as :Atendente participant Pessoa as Pessoa Atendente->>Pessoa: 1: Consulta Cliente: ConsCpf(long) </pre> <p>Figura 17 – Mensagens simples entre ator e objeto</p>
Um Objeto e outro Objeto	<pre> sequenceDiagram participant Nota_Fiscal as Nota_Fiscal participant Estoque as Estoque Note_Fiscal->>Estoque: 1: Após emissão da NF, ocorre a baixa no Estoque() </pre> <p>Figura 18 – Mensagens simples entre ator e objeto</p>
Um Objeto e um Ator	<pre> sequenceDiagram participant Nota_Fiscal as Nota_Fiscal participant Cliente as :Cliente Nota_Fiscal->>Cliente: 1: Envio da NFe para o Cliente após emissão() </pre> <p>Figura 19 – Mensagens simples entre ator e objeto</p>

Fonte: o autor.

Mensagens de retorno

Após entendermos a respeito das mensagens e suas respectivas funções, é importante sabermos sobre a mensagem de retorno. A função da mensagem de retorno é simbolizar o retorno das informações do método que a solicitou. Assim, esse retorno pode simbolizar os valores ou se o método foi executado com sucesso ou não (GUEDES, 2011). Conforme pode ser visualizado na Figura 20, a representação da mensagem de retorno se dá por meio de uma linha tracejada a qual contém uma ponta fina que aponta para o objeto que recebe o resultado:

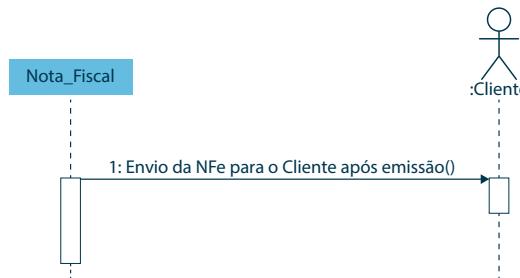


Figura 20 – Mensagens simples entre ator e objeto

Fonte: o autor.

Tipos de mensagens

Mensagens síncronas

No momento em que um objeto (chamador) realizar o envio de uma mensagem síncrona, o objeto deverá aguardar que ela seja finalizada para que seja continuado o fluxo. Um bom exemplo seria a chamada de uma sub-rotina (FOWLER, 2005). A representação de que uma mensagem é do tipo síncrona é uma flecha com o desenho da ponta preenchido, conforme demonstrado na Figura 21:

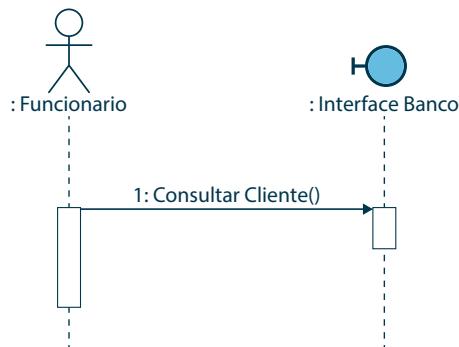


Figura 21 – Exemplo do uso de uma mensagem síncrona

Fonte: Guedes (2011, p. 196).

Mensagens assíncronas

As mensagens assíncronas são utilizadas para indicar que a execução ocorre em paralelo aos outros processos e que ela pode ter um processamento contínuo, o que retira a necessidade de aguardar por uma resposta para que o processo continue.

A representação de que uma mensagem é do tipo assíncrona, na versão da UML 2.0, é a ponta de seta tipo “pé de galinha”, ou seja, uma flecha com o desenho da ponta vazado (FOWLER, 2005). A seguir, na Figura 22, temos a sua representação:

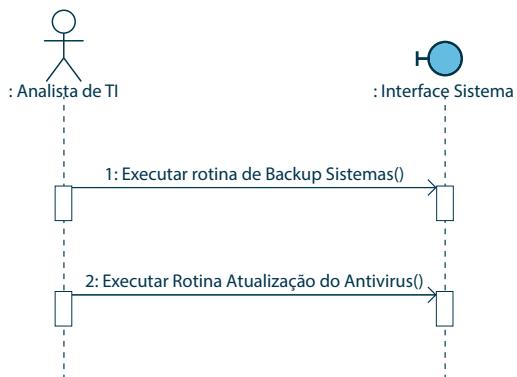


Figura 22 – Exemplo do uso de uma mensagem assíncrona

Fonte: o autor.

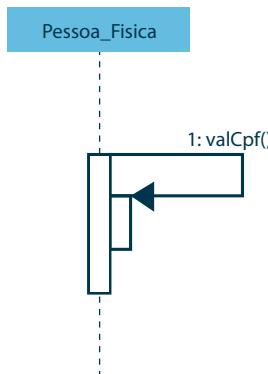


Figura 23 – Autochamadas

Fonte: Guedes (2011, p. 199).

Autochamadas ou autodelegações

Autochamadas são mensagens que o objeto envia para ele mesmo. Nelas, a seta parte da linha da vida e retorna ao próprio objeto (GUEDES, 2011). Na Figura 23, há a demonstração desse tipo de mensagem:

Estereótipo <<boundary>>

Também conhecido como estereótipo de fronteira, o estereótipo <<boundary>> é utilizado para identificação de uma classe que comunica os atores externos e o próprio sistema. O uso de classes <<boundary>> é feito quando se quer definir uma interface para o sistema (GUEDES, 2011).

A seguir, apresentamos sua representação:

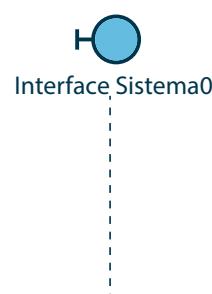


Figura 24 – Exemplo do uso do estereótipo <<boundary>>

Fonte: o autor.

Estereótipo <<control>>

O estereótipo <<control>> trata-se de uma classe intermediária entre as classes <<boundary>> e as demais. Seu objetivo é o de realizar uma interpretação de eventos realizados pelo objeto <<boundary>>, como as ações do mouse e a execução de botões, por exemplo, o que permite retransmitir ações aos demais objetos do sistema (GUEDES, 2011):

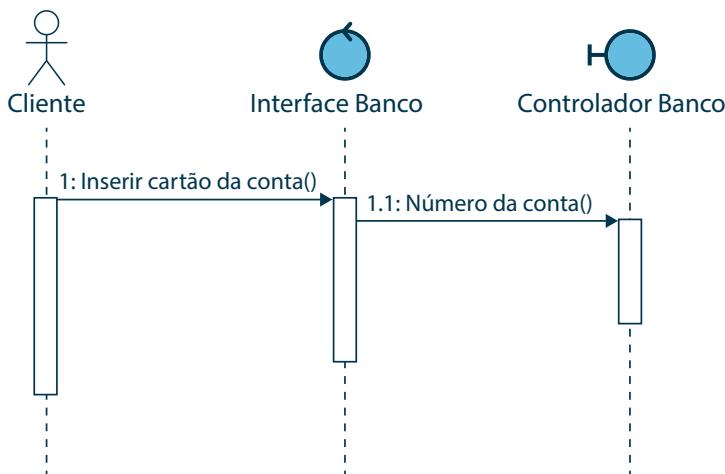


Figura 25 – Exemplo do uso do estereótipo <<boundary>> e estereótipo <<>>
Fonte: Guedes (2011, p. 203).

EXEMPLO DE DIAGRAMA DE SEQUÊNCIA

Após descrevermos alguns elementos que podem estar contidos em um diagrama de sequência, é importante exemplificarmos, de maneira que você o compreenda melhor. A seguir, na Figura 26, há um exemplo de um diagrama de sequência que engloba o processo de emissão de saldo:

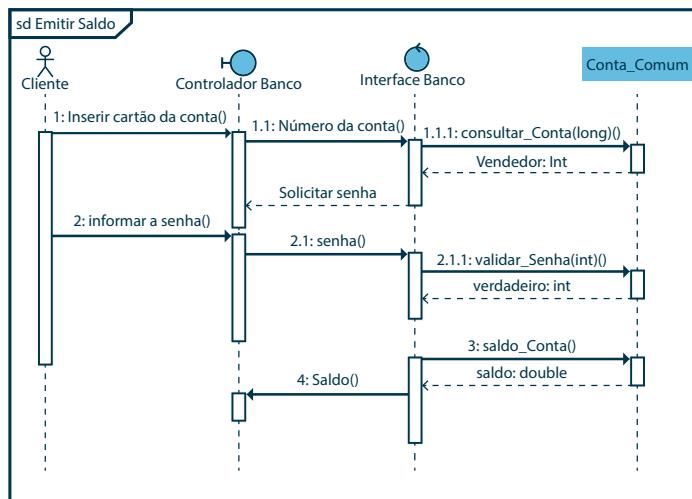


Figura 26 – Exemplo de diagrama de sequência: processo de emissão de saldo

Fonte: Guedes (2011, p. 203).

Conforme foi exposto na Figura 26, o processo de emissão de saldo é composto, inicialmente, pelo cliente, o qual insere o cartão da conta. Posteriormente, é passado o número da conta para o controlador, o qual dispara a consulta da conta por meio do método `consultar_Conta`. Assim, pode-se realizar a consulta na classe `conta_Comum`, se ela existir. Além disso, se o método `consultar_Conta` retornar enquanto `verdadeiro`, pode-se afirmar que a conta existe, permitindo a solicitação da senha ao cliente por meio da interface do sistema. Ao informar a senha, ela é validada junto a classe `conta_Comum` e, se for verdadeira, é, então, acionada a consulta do saldo e o seu retorno em tela para o ator cliente.



DIAGRAMA DE MÁQUINA DE ESTADOS

Segundo Guedes (2011, p. 242), “o diagrama de máquina de estados demonstra o comportamento de um elemento por meio de um conjunto finito de transições de estado, ou seja, uma máquina de estados”.

O diagrama de máquina de estados é um diagrama de comportamentos. Ele pode ser usado para especificar o comportamento de vários elementos, seja uma instância de uma classe ou um diagrama de caso de uso, por exemplo.

Para entendermos melhor a respeito do diagrama de máquina de estados, é importante compreendermos alguns de seus componentes. Para tanto, na sequência, abordaremos alguns deles.

ESTADO

Sobre um estado, é importante sabermos que ele representa, dentro do diagrama, os momentos em que um componente (objeto) está ou pode vir a estar. Dentro de um processo, podemos ter um ou vários estados ocorrendo de forma simultânea (GUEDES, 2011). Para representarmos um estado, utilizamos um retângulo, assim como o da Figura 27:

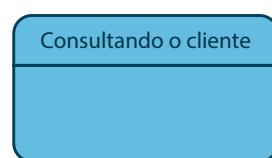


Figura 27 – Exemplo de estado: consultando o cliente

Fonte: o autor.

Atividades internas

Aprofundando um pouco mais os nossos estudos em relação ao objeto estado, é importante salientarmos que, na parte inferior, pode-se (não é obrigatório) descrever as atividades internas relativas ao estado descrito: Dentro das atividades internas, pode-se haver as seguintes variações:

- **Entry:** determina que a atividade descrita será executada no momento em que o objeto entra em um estado.
- **Exit:** determina que a atividade descrita será executada no momento em que o objeto sai de um estado.
- **Do:** determina que a atividade descrita será executada no período em que o estado for executado.



Figura 28 – Exemplo de atividades internas: consultando o cliente

Fonte: o autor.

TRANSIÇÕES

As transições ocorrem para evidenciar uma alteração no estado entre um objeto e outro, a fim de permitir a geração de um novo estado (GUEDES, 2011). A representação de uma transição se dá por meio de uma flecha, que pode, ou não, conter uma descrição a respeito. A seguir, na Figura 29, temos um exemplo de transição:



Figura 29 – Exemplo de transição: CPF ou CNPJ informado

Fonte: o autor.

ESTADO INICIAL

Para simbolizar que o processo está começando a ser modelado, usa-se o estado inicial, o qual é constituído por um círculo preenchido. A Figura 30 o representa:



Figura 30 – Exemplo do estado inicial

Fonte: o autor.

ESTADO FINAL

Para simbolizar que a modelagem do processo chegou ao fim, usa-se o símbolo do estado final. De acordo com a Figura 31, percebe-se que ele é demonstrado por meio de um círculo preto preenchido o qual está envolto de um círculo não preenchido:



Figura 31 – Exemplo do estado final

Fonte: o autor.

EXEMPLO DO DIAGRAMA DE MÁQUINA DE ESTADOS

Para entendermos melhor a respeito do diagrama de máquina de estados, demonstraremos uma rotina chamada emitir saldo:

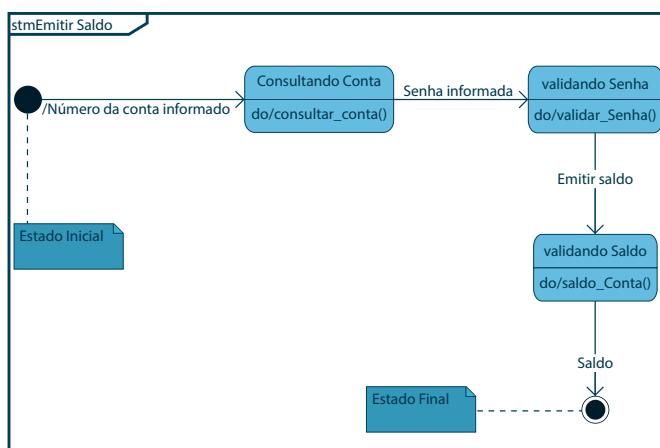


Figura 32 – Exemplo de diagrama de máquina de estados: processo de emissão de saldo
Fonte: Guedes (2011, p. 245).

No diagrama da Figura 32, temos, a partir do Estado Inicial, o número da conta a ser informado, o qual, por sua vez, será a transição para a ocorrência do próximo estado em que a consulta da conta será solicitada. Após a consulta da conta ocorrer por meio do método Consultar_conta, se o retorno for válido, acontecerá a próxima transição, na qual o usuário deverá informar a senha. Assim, haverá a mudança para o próximo estado, que será o de validar a senha. Estando correto os estados anteriores, temos a transição Emitir Saldo, que acionará o estado de consulta de saldo, o qual, por fim, retornará ao estado final com o saldo solicitado.

PSEUDOESTADO DE ESCOLHA

No diagrama de máquina de estados, utiliza-se o símbolo de losango para representar uma tomada de decisão, ou seja, uma condição que decidirá o próximo estado. A seguir, temos um exemplo de uso de pseudoestado de escolha:

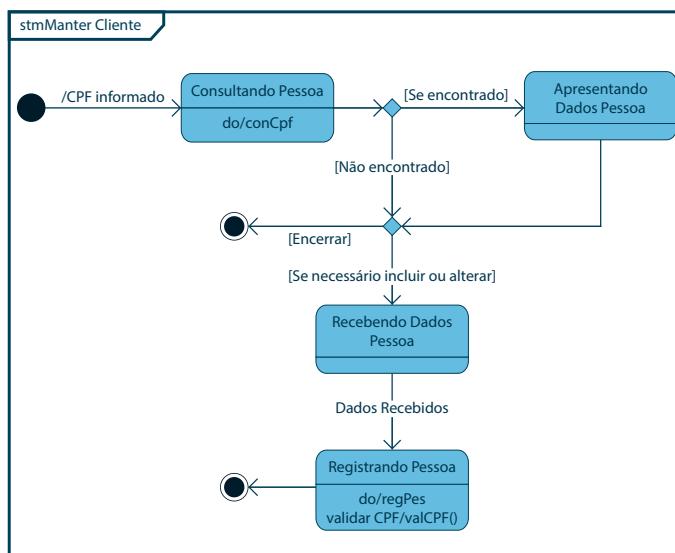


Figura 33 – Exemplo de uso de pseudoestado de escolha
Fonte: Guedes (2011, p. 250).

No diagrama da Figura 33, temos, como transição inicial, a informação do CPF, a qual levará ao primeiro estado, que é a consulta da pessoa a partir do dado

passado. Nesse momento, temos um pseudoestado de escolha, no qual o processo poderá seguir. Contudo, se, no estado de consulta, for retornado para pessoa, o processo irá para o próximo estado, que é a apresentação dos dados da pessoa. O outro momento se dá mediante a condição de que, se não for encontrado o cadastro a partir do CPF passado, a consulta poderá ser encerrada ou é possível ser feito o cadastro e registro, encerrando, assim, o processo.

SAIBA MAIS



A ideia inicial da construção do software existe em um formato nebuloso dentro da mente do interessado e a forma de resolver a essa necessidade não tem um formato definido.

Fonte: Medeiros (2004, p. 2).



DIAGRAMA DE ATIVIDADES

Segundo Guedes (2011, p. 277), “o diagrama de atividades ajuda na modelagem de atividades que podem ser um método, ou um algoritmo ou mesmo um processo completo”. Ainda, segundo o autor, o uso do diagrama de atividades está ligado, também, a descrição de computação procedural, a modelagem organizacional para engenharia de processos e ao *workflow*.

A partir desse conceito, temos alguns componentes que são importantes para o nosso conhecimento, a fim de que possamos realizar a criação desse tipo de diagrama. Para tanto, na sequência, listaremos alguns deles.

ATIVIDADE

Sendo representada por um retângulo com bordas arredondadas, uma atividade pode receber vários tipos de comportamentos, tais como ocorrências de funções aritméticas, comportamento de atividades, ações de comunicação, bem como a leitura e gravação de atributos ou até mesmo a instanciação, por exemplo (GUEDES, 2011).

Emitir NFe

Figura 34 – Exemplo de atividade
Fonte: o autor.

NÓ DE AÇÃO

Segundo Guedes (2011, p. 278), “um nó de ação representa um passo, uma etapa que deve ser executada em uma atividade. Um nó de ação é atômico, não podendo ser decomposto”. A representação é parecida com a da atividade, mas o retângulo é um pouco menor.

Receber o número do CPF ou CNPJ

Figura 35 – Exemplo de nó de ação
Fonte: o autor.

FLUXO DE CONTROLE

É representado por uma seta que realiza a ligação entre dois nós, local onde passam sinais de controle do nó antigo apontando para o novo. No fluxo de controle, pode-se, também, descrever a condição ou restrição, segundo Guedes (2011).

Receber o número do CPF ou CNPJ

Consultar Cliente

Figura 36 – Exemplo de fluxo de controle
Fonte: o autor.

NÓ INICIAL

Representado por um círculo preenchido, o nó inicial visa demonstrar o início do fluxo da atividade invocada. Assim como é demonstrado na Figura 37, temos o nó inicial acoplado a um fluxo de um sistema:



Figura 37 – Exemplo de nó inicial

Fonte: o autor.

NÓ DE FINAL DE ATIVIDADE

Sendo do tipo nó de controle e representado por um círculo preenchido por outro, o nó de final de atividade é utilizado para representar que o fluxo de uma atividade acabou. Na Figura 38, ele está sendo representado pela atividade ligada ao nó final:



Figura 38 – Exemplo de nó final de atividade

Fonte: o autor.

NÓ DE DECISÃO

A grande maioria dos fluxos de um sistema oferece possibilidades de percurso ou, assim como já conhecemos, normalmente, condições. Dentro do diagrama de atividades, utilizamos o símbolo de

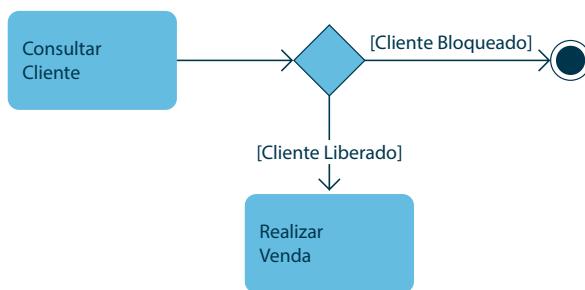


Figura 39 – Exemplo de nó de decisão

Fonte: o autor.

losango para simbolizar as possíveis condições que aquele ponto do sistema nos oferece. Na Figura 39, demonstramos que, no momento da consulta de um cliente, podemos ter duas consequências: uma delas é, se o cliente estiver bloqueado, ele não poderá realizar a venda e o fluxo termina. Em contrapartida, caso o seu cadastro esteja em condições normais (cliente liberado), ele poderá realizar a venda.

PARTIÇÃO DE ATIVIDADE

Em alguns casos, faz-se necessário representar por onde passa o fluxo de um processo. Para isso, utilizamos o desenho de retângulos com o nome do setor, departamento ou até mesmo de um processo que pode ser manipulado entre atores. Vale lembrar que esses retângulos podem ser na horizontal ou vertical, mas ambos devem ter o mesmo objetivo. Conforme pode ser visualizado na Figura 40, temos o início do fluxo representado, que é a responsabilidade do vendedor em emitir o pedido de venda. Após essa atividade, passa-se ao faturista a responsabilidade de emitir a NFe (Nota Fiscal Eletrônica). Por último, cabe ao departamento de estoque entregar a mercadoria no momento da apresentação da NFe, finalizando, assim, o fluxo do processo. Observe:

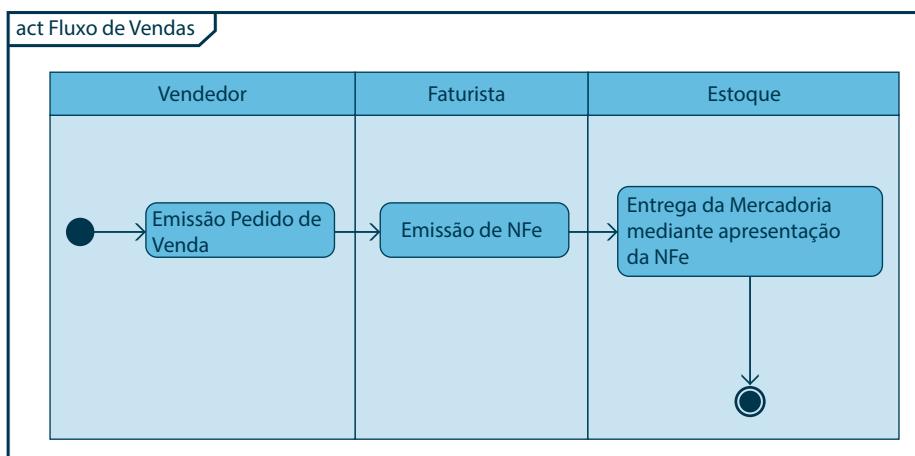


Figura 40 – Exemplo de partição de atividade: fluxo de vendas

Fonte: o autor.

EXEMPLO DE DIAGRAMA DE ATIVIDADES

Para ajudar na compreensão do diagrama de atividades da Figura 40, apresentaremos um fluxo simples do funcionamento de emissão de um pedido até o seu fechamento:

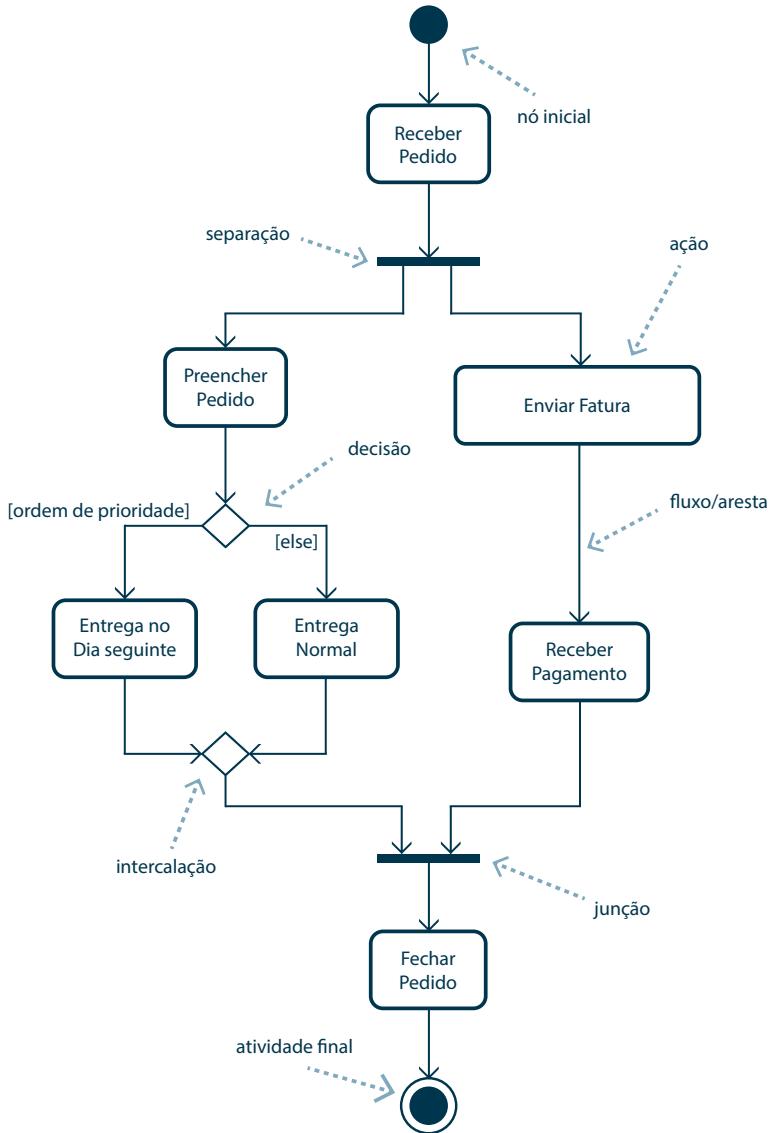


Figura 41 – Exemplo de um diagrama de atividades
Fonte: Fowler (2005, p. 119).

No diagrama de atividade apresentado, temos, de forma clara, a representação de um processo de pedido. Após a apresentação do nó inicial, simboliza-se a primeira ação, que é a de receber o pedido. Na sequência, observamos as primeiras duas ações que podem acontecer: a de preencher o pedido e a de enviar a fatura. Se for escolhida a de preencher o pedido, temos, como decisão, a escolha pela entrega no dia seguinte ou a entrega normal, resultando no fechamento do pedido. Já se a escolha for enviar a fatura, o fluxo remeterá ao recebimento do pedido e, assim, causará, na sequência, o fechamento do pedido, atividade final do fluxo apresentado.

Finalizando nossa unidade, constatamos que a modelagem é a parte fundamental de todas as atividades dentro de um processo de software as quais levam à implantação de um bom software. Esses modelos, normalmente, são representados por meio de diagramas em que é utilizada uma notação gráfica, a qual, em nosso caso, foi a UML.

CONSIDERAÇÕES FINAIS

No decorrer desta unidade, estudamos a modelagem de um sistema por meio de diagramas e dos elementos que compõem cada um. Em cada diagrama apresentado, os exemplos foram baseados em casos reais, possibilitando, assim, uma compreensão do conteúdo de forma mais simplificada.

A modelagem é a parte fundamental de todas as atividades dentro de um processo de software que levam à implantação de um bom software. Esses modelos, normalmente, são representados por meio de diagramas em que é utilizada uma notação gráfica, a qual, em nosso caso, foi a UML.

A UML tem muitos tipos de diagramas, apoiando a criação de diferentes modelos de sistema. Nesta unidade, conhecemos, o diagrama de classes, o diagrama de sequência, o diagrama de máquina de estados e o diagrama de atividades.

Mesmo cada diagrama tendo sua particularidade, todos os que foram estudados nesta unidade têm, como objetivo comum, auxiliar no processo de desenvolvimento. Isso se deve, pois, por intermédio do diagrama, a visão do processo fica mais real, mais clara e poderá ser melhor compreendida pelo desenvolvedor. Com isso, diminui-se a possibilidade de falhas no momento do desenvolvimento do sistema completo ou na criação de uma rotina específica.

Enfim, conhecer os diagramas e coloca-los em prática no momento de análise e desenvolvimento do sistema pode ajudá-lo a não ter surpresas desagradáveis no momento da entrega do software ao seu cliente. Contudo, vale lembrar que a utilização de diagramas não irá resolver todos seus problemas, embora possa te ajudar a evitar vários deles.

ATIVIDADES



1. Uma classe é um conjunto de objetos que possuem os mesmos tipos de características e comportamentos, sendo representada por um retângulo que pode possuir até três divisões. Partindo desse conceito, elabore um diagrama de classes duas classes que contenham as características citadas a seguir. Lembre-se que classes não trabalham sozinhas, portanto, construa, entre elas, um relacionamento 1..* (no mínimo 1 e, no máximo, muitos):

Nome da Classe:		Descrição:		
FUNCIONÁRIO		- Contém dados cadastrais dos funcionários.		
CAMPO	TIPO	VISIBILIDADE	CHAVE	DESCRIÇÃO
Matricula	Integer	Pública	PK	Matrícula
CpfCnpj	String	Pública		CPF ou CNPJ
NomeFunc	String	Pública	---	Nome do Funcionário
DtAdmissao	Date	Pública	---	Data da Admissão

Métodos:

- +IncluirNovoFunc
- +BuscarFunc
- +ExcluirFunc

Nome da Classe:		Descrição:		
DEPENDENTE		- Contém dados cadastrais do dependente		
CAMPO	TIPO	VISIBILIDADE	CHAVE	DESCRIÇÃO
IdDep	Integer	Privada	PK	Número do Dependente
NomeDep	String	Privada	---	Nome do Dependente
DtNascimento	Date	Privada	---	Data de Nascimento
IdFunc	Integer	Privada	FK	Código do Funcionário

Métodos:

- +IncluirNovoDep
- +ExcluirDep

ATIVIDADES



2. Desenvolva um diagrama de sequência para o processo de login do sistema com base nos seguintes requisitos:
 - 1: O usuário deverá inserir o login na interface do sistema.
 - 1.1: A interface do sistema irá enviar o login do sistema ao controlador do sistema.
 - 1.1.1: O controlador do sistema irá executar o método ConsultarLogin, que irá consultar na classe de usuários. Caso o retorno seja verdadeiro, ele retornará para a interface do sistema, a solicitação de senha.
 - 2.: Após o retorno da senha como verdadeira, o usuário deverá informá-la.
 - 2.1: A interface do sistema irá repassar ao controlador do sistema a senha.
 - 2.1.1: O controlador do sistema irá executar o método ValidarSenha na classe usuários. Caso o retorno seja verdadeiro, o controlador do sistema irá permitir o acesso a tela inicial do sistema (3).
3. Desenvolva um diagrama de máquina de estados para a inserção de um novo cliente no sistema:
 - a) No sistema, será informado o CPF ou CNPJ. O próximo estado será a consulta do cliente por meio do método ConsultarCliente.
 - b) Caso o cliente exista, o próximo estado será o de consultar pendências, no qual será executado o método ConsultaPendencias.
 - c) Caso o cliente não exista, o sistema pode ser encerrado ou pode-se passar para o próximo estado, que é o de inserir o novo cliente. Nele, será executado o método InserirCliente ou pode-se encerrar o sistema.
 - d) Após o cadastro de um novo cliente, o próximo estado deverá ser o de consultar pendências, no qual será executado o método ConsultaPendencias.
 - e) Após o estado de consulta de pendências, caso o cliente possua pendências, o sistema será encerrado. Em contrapartida, caso ele não possua pendências, o estado será o de mostrar os dados do cliente, podendo, assim, ser encerrado o sistema.

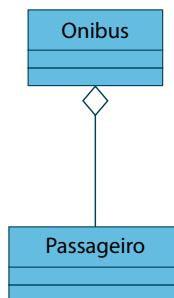
ATIVIDADES



4. Quando se fala em diagrama, a primeira imagem que nos vem à cabeça é um desenho composto de linhas e retângulos. Esses “desenhos”, porém, podem representar linhas e linhas de informações escritas. Os diagramas servem para sintetizar ideias que serão transferidas para o papel ou para o código fonte de uma aplicação. Com eles, é possível entender o funcionamento da aplicação como um todo, passando pela diagramação do banco, dos processos e até das classes utilizadas na programação. Uma série deles podem ser utilizados na documentação dos projetos, a fim de auxiliar os envolvidos a entender melhor toda a estrutura analisada e desenvolvida (AVANT SOLUÇÕES, 2016, on-line)¹.

A partir do contexto abordado, descreva a função e quando devemos utilizar o diagrama de atividades.

5. O diagrama de classe é uma representação da estrutura e das relações das classes que servem de modelo para objetos. Partindo disso, analise o diagrama de classe a seguir e assinale a alternativa correta:



- a) Entre as entidades Onibus e Passageiro, temos um relacionamento de composição.
- b) Entre as entidades Onibus e Passageiro, temos um relacionamento de agregação.
- c) Entre as entidades Onibus e Passageiro, temos um relacionamento de generalização/especialização (herança).
- d) Entre as entidades Onibus e Passageiro, temos um relacionamento unário.
- e) Entre as entidades Onibus e Passageiro, temos um relacionamento de realização.



Como qualquer outro fluxo de trabalho do processo de desenvolvimento de software, problemas e armadilhas em potencial estão associados ao fluxo de trabalho de levantamento de necessidades. Em primeiro lugar, é essencial ter a cooperação dedicada de possíveis usuários do produto-alvo desde o início do processo. Normalmente, as pessoas sentem-se ameaçadas pela informatização, com receio de que os computadores tomarão seu trabalho. Há certa razão para esse receio. Ao longo dos últimos 30 anos, o impacto da informatização tem sido o de reduzir a necessidade de trabalhadores não especializados, mas também o de gerar postos de trabalho para trabalhadores especializados. Afinal, o número de oportunidades de emprego bem remuneradas, criadas como consequência direta da informatização, ultrapassou, de longe, o número de trabalhos sem necessidade de especialização que se tornaram redundantes, conforme evidenciado tanto por taxas de desemprego menores, quanto de remuneração média maiores. Mas o crescimento econômico sem paralelo de tantos países ao redor do mundo, como consequência direta ou indireta da assim chamada "era da informática", de forma alguma é capaz de compensar o impacto negativo sofrido por aqueles indivíduos que perderam seus empregos como consequência da informatização.

É essencial que cada membro da equipe de levantamento de necessidades esteja atento, a todo momento, que os funcionários da organização-cliente com os quais eles interagem certamente estão muito preocupados com o possível impacto do produto de software a ser desenvolvido em seus empregos. No pior caso, funcionários podem deliberadamente fornecer informações incorretas ou que podem induzir a erros, na tentativa de garantir que o produto não atenda às necessidades do cliente e, assim, preservem seus próprios empregos. Porém, mesmo sem nenhuma sabotagem desse tipo, alguns funcionários da empresa-cliente podem ser menos úteis, simplesmente por terem um vago pressentimento de estarem sendo ameaçados pela informatização da empresa. Outro desafio do fluxo de trabalho de levantamento de necessidades é a habilidade de negociar. Por exemplo, muitas vezes, é essencial reduzir o que o cliente deseja. Não é surpresa alguma que todo cliente adoraria ter um produto de software que pudesse fazer tudo o que fosse conceitivamente necessário. Um produto desses levaria um tempo inaceitavelmente longo para ser construído e custaria muito mais do que o cliente consideraria razoável. Portanto, muitas vezes, é necessário persuadi-lo a aceitar menos (algumas vezes, bem menos) do que ele desejaría. Calcular os custos e benefícios de cada necessidade em questão pode ajudar nesse aspecto.

Outro exemplo da capacidade de negociação necessária é a habilidade de chegar a um acordo entre os gerentes sobre a funcionalidade do produto desejado. Por exemplo, um gerente astuto poderia tentar estender seu poder incluindo uma necessidade que pode ser implementada apenas por meio da incorporação nas áreas de sua responsabilidade de certas funções de negócio, atualmente sob responsabilidade de outro gerente. De forma não surpreendente, o outro gerente colocará fortes objeções ao descobrir o que está acontecendo. A equipe de levantamento de necessidades deve se sentar com ambos os gerentes e resolver a questão.



Um terceiro desafio do fluxo de trabalho de levantamento de necessidades é que, em muitas organizações, as pessoas que possuem informações que a equipe de levantamento de necessidades precisa revelar simplesmente não tem tempo para longas reuniões. Quando isso acontece, a equipe deve informar o cliente de que deve decidir o que é mais importante para eles: as atuais atribuições do cargo dessas pessoas ou o produto de software a ser construído. E, se o cliente não insistir que o produto de software vem em primeiro lugar, talvez os desenvolvedores de software não tenham outra alternativa a não ser abandonar um projeto que está fadado ao fracasso.

Finalmente, flexibilidade e objetividade são essenciais para o levantamento de necessidades. É vital que os membros da equipe de levantamento de necessidades abordem cada entrevista sem ideias preconcebidas. Particularmente, um entrevistador jamais deve fazer pressuposições sobre as necessidades em consequência das primeiras entrevistas e, depois conduzir as entrevistas posteriores dentro da estrutura dessas suposições. Ao contrário, um entrevistador deve suprimir, de forma consciente, quaisquer informações coletadas em entrevistas anteriores e conduzir cada uma delas de forma imparcial. Fazer hipóteses prematuras referentes às necessidades é perigoso; fazer quaisquer pressuposições durante o fluxo de trabalho de levantamento de necessidades referentes ao produto de software a ser construído pode ser desastroso. O capítulo encerra-se com a Figura 42, que sintetiza as etapas necessárias do fluxo de trabalho de levantamento de necessidades.

Como realizar o fluxo de trabalho de levantamento de necessidades

- Iterar

- Ter um entendimento do campo de aplicação.

- Elaborar o modelo de negócio.

- Estabelecer as necessidades.

- Até as necessidades serem satisfatórias.

Figura 42 – Como realizar o fluxo de trabalho de Levantamento de Necessidades

Fonte: Schach (2010, p. 326).

Fonte: Schach (2010, p. 325-326).

MATERIAL COMPLEMENTAR



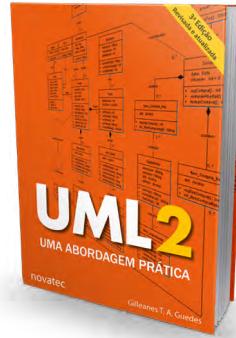
LIVRO

UML 2. Uma Abordagem Prática

Gilleanes Thorwald Araújo Guedes

Editora: Novatec

Sinopse: a UML – Unified Modeling Language ou Linguagem de Modelagem Unificada – é uma linguagem utilizada para modelar softwares baseados no paradigma de orientação a objetos, aplicada principalmente durante as fases de análise de requisitos e projeto de software. Essa linguagem consagrou-se como a linguagem-padrão de modelagem adotada internacionalmente pela indústria de Engenharia de Software, havendo um amplo mercado para profissionais que a dominem. Este livro procura ensinar ao leitor, por meio de exemplos práticos, como modelar softwares por meio da UML. A linguagem é ensinada mediante a apresentação de seus muitos diagramas, detalhando o propósito e a aplicação de cada um deles, bem como os elementos que os compõem, suas funções e como podem ser aplicados. A obra enfatiza, ainda, a importância da UML para a Engenharia de Software, além de abordar o paradigma de orientação a objetos, um conceito imprescindível para a compreensão correta da linguagem. Além disso, o livro demonstra como mapear classes em tabelas de banco de dados relacionais, enfocando a questão de persistência. A obra contém diversos estudos de caso modelados, bem como exemplos ao longo dos capítulos, além de um estudo de caso maior ao término, em que um sistema é analisado e modelado, com a ilustração completa de todos os diagramas referentes ao software. O livro apresenta, também, vários exercícios para avaliar e consolidar os conhecimentos adquiridos pelo leitor, com as respectivas soluções ao final do capítulo onde foram propostos. A obra pode ser utilizada tanto por professores e alunos universitários de cursos da área de computação quanto por profissionais da área de engenharia e desenvolvimento de software.



REFERÊNCIAS

FOWLER, M. **UML essencial**: um breve guia para a linguagem-padrão de modelagem de objetos. 3. ed. Porto Alegre: Bookman, 2005.

GUEDES, G. T. A. **UML 2**: guia prático. São Paulo: Novatec, 2007.

_____. **UML 2**: uma abordagem prática. 2. ed. São Paulo: Novatec, 2011.

MEDEIROS, E. S. de. **Desenvolvendo software com UML 2.0**: definitivo. São Paulo: Pearson Makron Books, 2004.

MELO, A. C. **Desenvolvendo Aplicações com UML 2.0**: do conceitual à implementação. Rio de Janeiro: Brasport, 2004.

SCHACH, S. R. **Engenharia de software**: os paradigmas clássico e orientado a objetos. 7. ed. Porto Alegre: AMGH, 2010.

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.

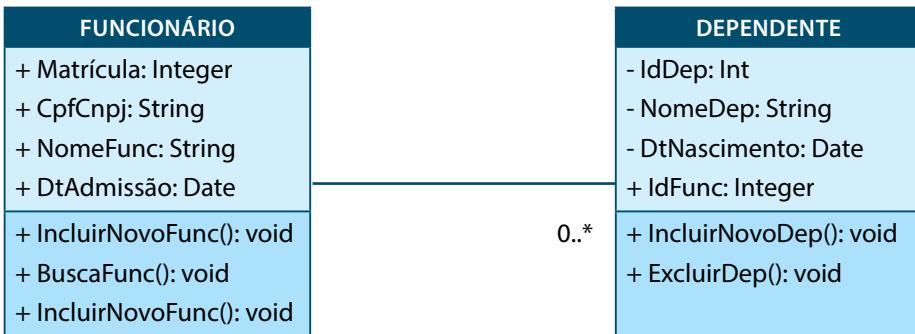
REFERÊNCIA ON-LINE:

- 1 Em: <http://www.avantsolucoes.com.br/importancia-dos-diagramas-na-documentacao-de-projetos/>. Acesso em: 01 jul. 2019.

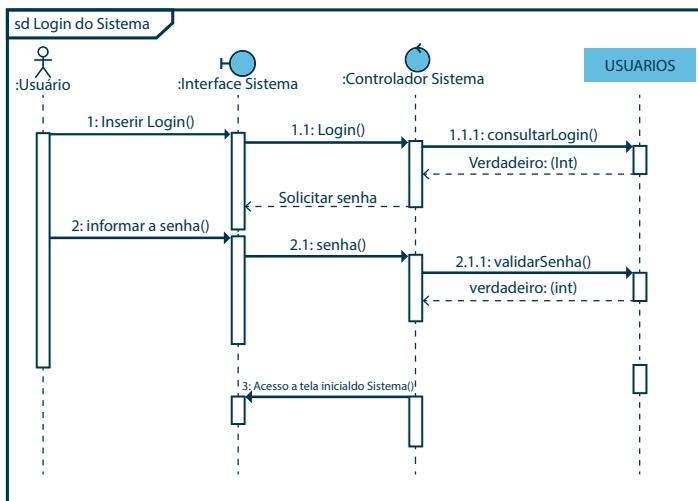


GABARITO

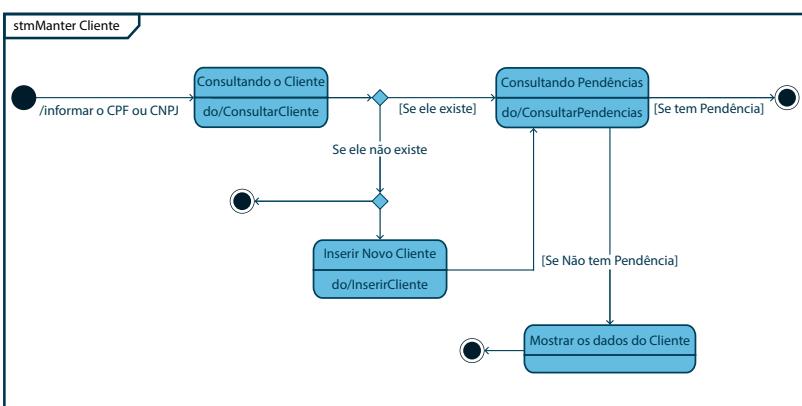
1.



2.



3.



GABARITO

4. O diagrama de atividades ajuda na modelagem de atividades, as quais podem ser um método, um algoritmo ou até mesmo um processo completo. O uso do diagrama de atividades está ligado, também, com a descrição de computação procedural, modelagem organizacional para engenharia de processos e workflow.
5. A alternativa correta é a D.



GERENCIAMENTO DE SOFTWARE

UNIDADE



Objetivos de Aprendizagem

- Conceituar qualidade de software, seus fatores e elementos de garantia de qualidade.
- Definir teste de software, seus objetivos e conceitos.
- Entender a evolução e a manutenção de software.
- Apresentar os conceitos da configuração de software.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Qualidade de software
- Teste de software
- Evolução de software
- Configuração de software

INTRODUÇÃO

Olá, aluno(a)! Na quarta unidade, estudamos a modelagem do sistema e aprendemos o diagrama de classes, diagrama de sequência, diagrama de máquinas de estados e diagrama de atividades. Agora, vamos dar sequência para as próximas etapas do processo que envolve o desenvolvimento de software.

Para tanto, trataremos sobre a qualidade, teste, evolução e configuração do software, com o objetivo de compreender a importância desses conceitos enquanto partes do processo de desenvolvimento do software.

Nesta unidade, iniciaremos nossos estudos com a qualidade de software, bem como seus fatores influenciadores. Quando pensamos em qualidade, geramos muitas preocupações relacionadas com a gestão da qualidade que podem ser resumidas em elementos que contribuem para organizá-la, assim como os padrões e normas de qualidade e os testes de software, também estudaremos os conceitos dos padrões e as normas de qualidade CMMI e MPSBr.

Segundo, abordaremos a etapa de testes, na qual passamos a validar o sistema que foi implementado. Nessa etapa, são testados os códigos, a fim de se procurar defeitos ou problemas que podem ocorrer na interface, bem como os que podem surgir quando o sistema for integrado.

Além disso, estudaremos a evolução e manutenção de software, visto que, independentemente do domínio de aplicação, tamanho ou complexidade, o software continuará a evoluir com o tempo. Em outras palavras, podem ocorrer alterações quando são corrigidos os erros, quando há adaptação de um novo componente ou quando o cliente pede novas funcionalidades.

Assim, nosso objetivo, nesta unidade, é o de entender os conceitos das etapas que envolvem o processo de software, bem como elas se englobam. Vamos, portanto, aos nossos estudos! Boa leitura!



INTRODUÇÃO À QUALIDADE DE SOFTWARE

Para que o software possa ser funcional e prático, é preciso ter qualidade em sua produção e manutenção. A qualidade, em software, refere-se às características dos produtos que atendem os requisitos de funcionalidade. Esses requisitos estão ligados diretamente com a área de utilização do programa e, para cada um, temos um campo de abrangência, uma especificidade. Assim, tais requisitos são: funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade, portabilidade, estabilidade.

De acordo com Vasconcelos *et al.* (2006, p. 47):

Esses requisitos fazem parte da qualidade do software de forma que cada produto, cada programa, possa atender as necessidades tecnológicas, econômicas, sociais, intelectuais e práticas do setor ao qual se destinam, observando as especificidades de cada área onde o produto é utilizado. Assim deve se levar em consideração a utilização do produto. Em áreas econômicas certos requisitos serão mais necessários do que na área educacional, onde os requisitos de um programa atendem as necessidades daquele público em particular.

Assim, a qualidade se torna uma parte importante em todo o processo de produção de um software ao se considerar não apenas os aspectos tecnológicos,

econômicos ou sociais, mas a sua praticidade, as funções que desempenha, a facilidade de uso, a capacidade de ser reciclado, manutenção, entre outros. A qualidade visa apresentar um produto completo, que atenda todas as necessidades do setor ao qual se destina.

O termo qualidade possui várias definições, as quais variam de acordo com a abordagem utilizada. A seguir, apresentamos algumas definições utilizadas na literatura:

- Conformidade com as especificações, de Philip B. Crosby: sugere que o gerenciamento da qualidade deve ser feito desde o início do desenvolvimento, para tentar evitar defeitos e diminuir o retrabalho. No desenvolvimento de software, esse conceito significa que devemos nos preocupar com a qualidade desde o início do processo (levantamento de requisitos e a parte de modelagem), para reduzir os problemas nas fases finais (codificação e testes) (DEVMEDIA, 2009, on-line)¹.
- Adequação ao uso, de Joseph M. Duran: significa que as expectativas do cliente são atendidas. Assim, temos dois fatores a considerar: a qualidade obrigatória, na qual o produto cumpre o que devia fazer, e a qualidade atrativa, na qual o produto vai além do que ele devia fazer, ou seja, oferece recursos adicionais ao cliente (DEVMEDIA, 2009, on-line)¹.
- NBR ISO 8402: “qualidade é a totalidade das características de uma entidade que lhe confere a capacidade de satisfazer às necessidades explícitas e implícitas” (ABNT, 1994, p. 01).

Esse último conceito exige alguns complementos, principalmente para definir o que são as entidades, as necessidades explícitas e as implícitas.

A **entidade** é o produto do qual estamos falando, que pode ser um bem ou um serviço. Já as **necessidades explícitas** são expressas na definição de requisitos que definem as condições em que o produto deve ser utilizado, seus objetivos, funções e qual vai ser o seu desempenho esperado. Por sua vez, as **necessidades implícitas** não são expressas em documentos, mas são necessárias para o usuário durante o manuseio do produto no seu dia a dia.

Para uma empresa, o que é mais importante no que diz respeito à qualidade é que não basta que ela exista, mas que ela seja reconhecida pela satisfação do cliente com o produto que ele adquiriu. Isso nos leva a perceber que, para iniciarmos o desenvolvimento de um produto, devemos começar com as necessidades

dos clientes em relação a esse item (SOMMERVILLE, 2011).

No início, certificava-se a qualidade do produto a um custo muito elevado. No final dos anos 70, os autores Boehm e McCall propuseram uma classificação para os fatores que afetam a qualidade de um produto de software. Assim, os padrões para a garantia de qualidade surgiram nos anos 70. As normas ISO também utilizaram os conceitos destes autores como base para as normas de qualidade de produto de software (SOMMERVILLE, 2011).

Dessa forma, programas de melhorias de qualidade bem-sucedidos proporcionam um aumento de produtividade, redução no número de defeitos no software, maior previsão e visibilidade dos processos definidos, além do cumprimento das metas de custo, prazo, funcionalidade e aumento na motivação da equipe desenvolvedora. A decisão de qual modelo ou norma que deve ser aplicado em cada empresa será feita com base em suas estratégias de negócio e de mercado, levando-se em consideração os benefícios e retorno sobre o investimento realizado.



REFLITA

Como posso definir qualidade de software da melhor maneira possível?

(Roger S. Pressman e Bruce R. Maxim)

FATORES DE QUALIDADE

Segundo Pressman e Maxim (2016, p. 416) “McCall, Richards e Walters criaram uma proposta de categorização dos fatores que afetam a qualidade de software”. Os fatores de qualidade apresentados na Figura 1 e descritos no Quadro 1 se concentram em três importantes aspectos: revisão, a transição e a operação do produto de software:

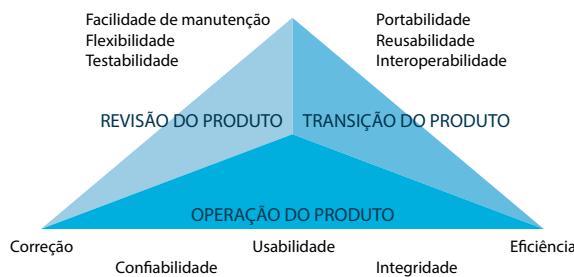


Figura 1 – Fatores de qualidade de software

Fonte: Pressman e Maxim (2016, p. 417).

O Quadro 1 mostra os fatores de qualidade de software citados na Figura 1, bem como suas descrições:

Quadro 1 - Fatores de qualidade de software e suas descrições

CORREÇÃO	O quanto um programa satisfaz a sua especificação e atende aos objetivos da missão do cliente.
CONFIABILIDADE	O quanto se pode esperar que um programa realize a função pretendida com a precisão exigida.
EFICIÊNCIA	A quantidade de recursos computacionais e código exigidos por um programa para desempenhar sua função.
INTEGRIDADE	O quanto o acesso ao software ou dados por pessoas não autorizadas pode ser controlado.
USABILIDADE	Esforço necessário para aprender, operar, preparar a entrada de dados e interpretar a saída de um programa.
FACILIDADE DE MANUTENÇÃO	Esforço necessário para localizar e corrigir um erro em um programa.
FLEXIBILIDADE	Esforço necessário para modificar um programa em operação.
TESTABILIDADE	Esforço necessário para testar um programa, de modo a garantir que ele desempenhe a função destinada.
PORATIBILIDADE	Esforço necessário para transferir o programa de um ambiente de hardware e/ou software para outro.
REUSABILIDADE	O quanto um programa [ou partes de um] pode ser reutilizado em outras aplicações relacionadas ao empacotamento e o escopo das funções que o programa executa.
INTEROPERABILIDADE	Esforço necessário para integrar um sistema a outro.

Fonte: adaptado de Pressman e Maxim (2016, p. 417).



SAIBA MAIS

Está comprovado que quanto mais cedo for detectado e corrigido um defeito, menor será o impacto e a propagação nas demais fases do ciclo de vida. Dessa forma, devemos procurar identificar os defeitos o mais breve dentro do processo de desenvolvimento, ou seja, já a partir da identificação dos requisitos do sistema. O pior resultado obtido por um sistema de informação é quando seus requisitos não correspondem àqueles esperados por seus usuários. Se a análise de requisitos for “pobre”, todo o restante do processo estará comprometido e o risco de insucesso do projeto será bastante alto. Para auxiliar na identificação de defeitos durante a etapa de análise de requisitos, podemos recorrer às inspeções. Inspeções são uma das poucas técnicas de revisão disponíveis para aplicação em artefatos de software que não o código-fonte, podendo ser aplicadas a qualquer tipo de documento escrito, sejam elas especificações de requisitos, documentos ou modelos de projeto.

Fonte: Costa Júnior e Melo (2003, p. 45).

ELEMENTOS DE GARANTIA DA QUALIDADE DE SOFTWARE

Quando pensamos em garantir a qualidade de software, temos muitas preocupações e atividades relacionadas com a gestão da qualidade, as quais podem ser resumidas assim como expõe o Quadro 2:

Quadro 2 – Elementos de garantia da qualidade de software

PADRÓES	O IEEE, a ISO e outras organizações de padronização produziram uma ampla gama de padrões para engenharia de software e seus respectivos documentos. Os padrões podem ser adotados voluntariamente por uma organização de engenharia de software ou impostos pelo cliente ou outros interessados.
REVISÕES E AUDITORIAS	As revisões técnicas são uma atividade de controle de qualidade realizada por engenheiros de software. Seu intuito é o de revelar erros.

TESTES	Os testes de software são uma função de controle de qualidade com um objetivo principal: o de descobrir erros.
COLETA E ANÁLISE DE ERROS/DEFEITOS	A única forma de melhorar é medir o nosso desempenho. Assim, deve-se reunir e analisar dados de erros e defeitos para melhor compreender como os erros são introduzidos e quais atividades de engenharia de software melhor se adequam para sua eliminação.
ADMINISTRAÇÃO DA SEGURANÇA	Com o aumento dos crimes cibernéticos e novas regulamentações governamentais referentes à privacidade, toda organização de software deve instituir políticas que protejam os dados em todos os níveis, estabelecer proteção por meio de <i>firewalls</i> para as aplicações da Internet (<i>WebApps</i>) e garantir que o software não tenha sido alterado internamente, sem autorização.
PROTEÇÃO	O fato de o software ser quase sempre um componente fundamental de sistemas que envolvem vidas humanas (por exemplo, aplicações na indústria automotiva ou aeronáutica), o impacto de defeitos ocultos pode ser catastrófico. Devemos avaliar o impacto de falhas de software e por iniciar as etapas necessárias para redução de riscos.

Reprodução proibida. Art. 184 do Código Penal e Lei 9.610 de 19 de fevereiro de 1998.

Fonte: adaptado de Pressman e Maxim (2016, p. 450).

A seguir, vamos explanar dois dos elementos de garantia da qualidade de software: os padrões de qualidade de software e os testes de software.

NORMAS E MODELOS DE PADRÕES DE QUALIDADE DE SOFTWARE

As normas e modelos de padrões de qualidade de software são a principal chave para a garantia da qualidade. São elas quem definem as características que todos os componentes de software devem possuir e como o processo de software deve ser

conduzido, de forma a assegurar a qualidade do produto de software (REZENDE, 2005). Na literatura, encontramos várias normas e modelos. Entretanto, na disciplina, vamos estudar o CMMI e MPSBr.

CMMI

O CMMI (*Capability Maturity Model Integration* ou Modelo Integrado de Maturidade e Capacitação, em português), o qual foi desenvolvido pelo SEI (*Software Engineering Institute*), é um modelo desenvolvido para a melhoria da maturidade dos processos de desenvolvimento de software. Segundo Rezende (2005), o CMMI é um modelo para avaliar e melhorar a capacitação das empresas que desenvolvem software, propondo etapas que levam a instituição a se aprimorar continuamente em busca de soluções para o seu crescimento com qualidade.

O CMMI é uma evolução do modelo CMM, que era voltado para o desenvolvimento do software. Uma das modificações realizadas se dá na inclusão de um novo conceito no que se refere à representação contínua, a qual oferece maior flexibilidade para a empresa.

Para Rezende (2005), os objetivos principais do CMMI envolvem:

- Auxiliar as empresas a se conhecerem. A partir disso, espera-se que elas melhorem seus processos de desenvolvimento e a manutenção do software.
- Fornecer, às empresas, um controle de seus processos por meio de uma estrutura conceitual e, com isso, obter a melhoria contínua de seus produtos de software.

O modelo não diz como implementar determinadas práticas, ele apenas indica o que deve ser feito. Isso é comprovado pela definição dada por Machado (2016, p. 42), o qual afirma que o “CMMI é formado por um pacote de produtos que reúne múltiplos modelos de processo, associados a seus manuais, material de treinamento e de avaliação”.

Representações do CMMI

Existem duas representações, uma contínua e outra em estágios, nas quais os modelos CMMI podem variar de acordo com a representação e o corpo de conhecimento escolhido pela empresa. A seguir, explicaremos ambas representações:

Representação contínua: é caracterizada pelos níveis de capacidade. O modelo descreve um caminho de melhoria de maturidade por meio de cinco níveis distintos, cada um com características individuais, as quais determinam qual é a capacitação do processo. Quanto maior o nível, maior é a maturidade dos processos de desenvolvimento de software de uma empresa. Os níveis do modelo CMMI são representados assim como ilustra a Figura 2 a seguir:



Figura 2 – Níveis do modelo CMMI
Fonte: Machado (2016, p. 52).

Representação em estágio: foca nas melhores práticas que uma empresa pode utilizar. A maturidade é medida por um conjunto de processos (MACHADO, 2016). A seguir, são mostradas, no Quadro 3, as áreas de processos existentes por cada nível de maturidade:

Quadro 3 – Áreas de processo por nível de maturidade

NÍVEL DE MATURIDADE	ÁREAS DE PROCESSO
Nível 2 – Gerenciado	Gestão de requisitos Planejamento de projeto Monitorização e controle de projeto Gestão de acordo com o fornecedor Mediação e análise Garantia da qualidade de processo e do produto Gestão de configurações
Nível 3 – Definido	Desenvolvimento de requisitos Solução técnica Integração do produto Verificação Validação Definição do processo organizacional Formação organizacional Gestão de riscos Integração de equipes Gestão integrada de fornecedores Ambiente organizacional para integração Análise das decisões e resoluções
Nível 4 – Quantitativamente Gerenciado	Performance do processo organizacional Gestão quantitativa do projeto
Nível 5 - Otimizado	Inovação e desenvolvimento organizacional Análise e resolução causal

Fonte: adaptado de Machado (2016, p. 45).

O Quadro 3 nos mostra que cada nível de maturidade é composto por várias áreas-chaves de processo, as quais possuem objetivos específicos e genéricos. Cada objetivo específico é alcançado por meio de algumas práticas bem definidas. Por sua vez, cada objetivo genérico especifica várias funcionalidades comuns, as quais estão ligadas com as práticas genéricas. Essas práticas são os detalhes operacionais que devem ser abordados pelo processo ou pela metodologia de desenvolvimento usado na empresa.

O modelo CMMI fornece orientação para melhorias nos processos e nas habilidades organizacionais. Ele também inclui o ciclo de vida de produtos e serviços de um software, o qual abrange as fases de: concepção, desenvolvimento, aquisição, entrega e manutenção (RETAMAL, 2005).

O modelo CMMI tornou-se rapidamente conhecido pela riqueza de detalhes e pela sua completa descrição do processo de software. Além disso, influenciou outros modelos, como o SPICE – ISO/IEC 15504, o PSP e MPSBr. A seguir, trataremos sobre o modelo MPSBr.

MPSBR

O MPSBr é um modelo de qualidade de processos de software voltado para as características das empresas brasileiras. Ele auxilia as empresas a implantarem seus processos de software em conformidade com os principais padrões e modelos internacionais de qualidade de software (SOFTEX, 2016).

Além disso, o MPSBr é baseado no padrão CMMI, nas normas ISO/IEC 12207 e SPICE, e principalmente, na realidade do mercado brasileiro. Uma das principais vantagens do modelo é seu custo reduzido de certificação, sendo ideal para micro, pequenas e médias empresas. Ele está dividido em três componentes:

- Modelo de Referência (MR-MPS): para serviço e gestão de pessoas.
- Método de Avaliação (MA-MPS).
- Modelo de Negócio (MN-MPS).

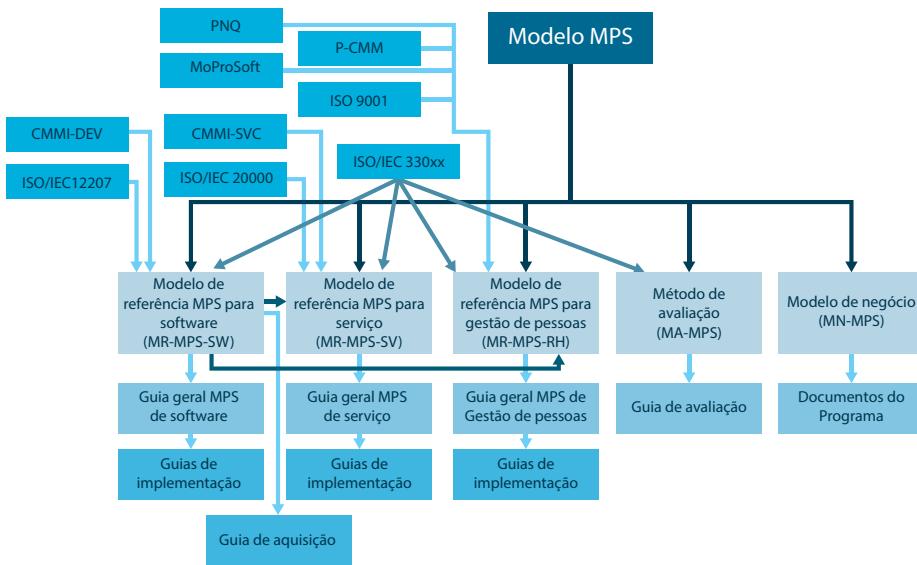


Figura 3 – Componentes do modelo MPSBr

Fonte: Softex (2016, p. 12).

O MR-MPS é um modelo de referência para a melhoria do processo de software e apresenta sete níveis de maturidade, que são:



De acordo com o *Guia Geral MPS de Software* elaborado por Softex (2016), cada nível de maturidade possui suas áreas de processo, nas quais são analisados os:

- **Processos fundamentais:** aquisição, gerência de requisitos, desenvolvimento de requisitos, solução técnica, integração, instalação e liberação do produto.
- **Processos organizacionais:** gerência de projeto, adaptação do processo para gerência de projeto, análise de decisão e resolução, gerência de riscos, avaliação e melhoria do processo organizacional, definição do processo

organizacional, desempenho do processo organizacional, gerência quantitativa do projeto, análise e resolução de causas, inovação e implantação na organização.

- **Processos de apoio:** garantia de qualidade, gerência de configuração, validação, medição, verificação e treinamento.

Em seguida, apresentam-se os níveis de capacidade, nos quais são obtidos os resultados dos processos analisados. Neles, cada nível de maturação possui um número definido de capacidades, que são:

- AP 1.1 - O processo é executado.
- AP 1.2 - O processo é gerenciado.
- AP 2.2 - Os produtos de trabalho do processo são gerenciados.
- AP 3.1 - O processo é definido.
- AP 3.2 - O processo está implementado.

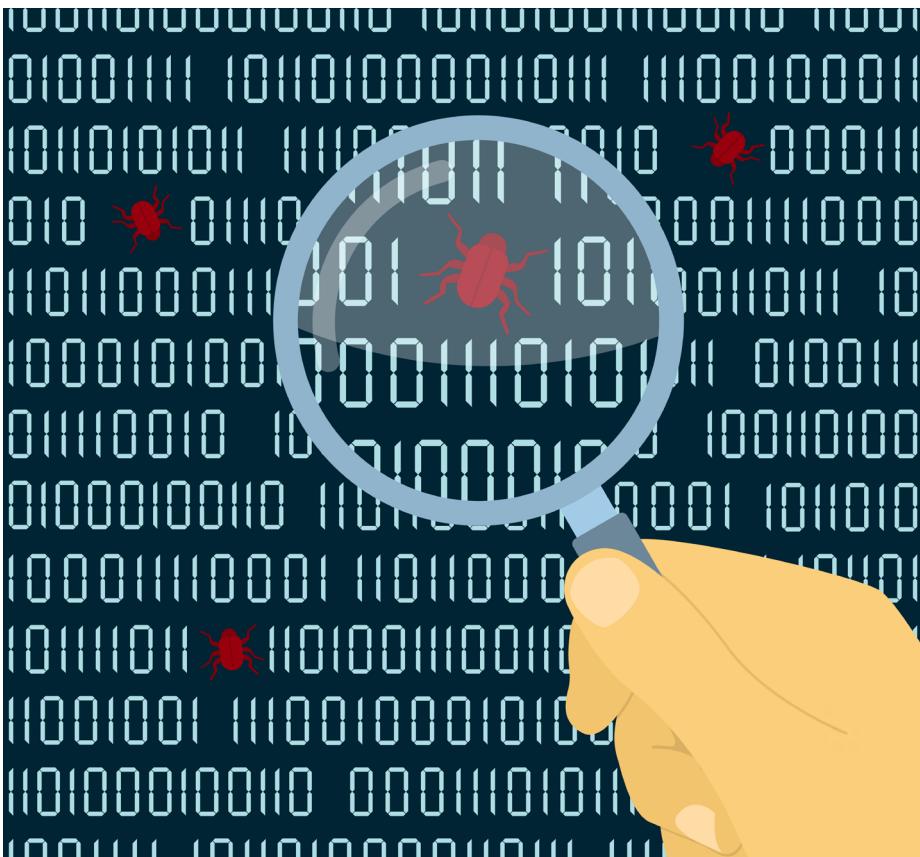
O MA-MPS é um método de avaliação para melhoria do processo de software e o seu objetivo é o de orientar a realização de avaliações, em conformidade com a norma ISO/IEC 15504, em empresas que já implementaram o MR-MPS. Já o MN-MPS é um modelo de negócio para melhoria do processo de software (SOFTEX, 2016).



SAIBA MAIS

Os 3 modelos estão em conformidade com os requisitos para da Norma Internacional ISO/IEC 33020 [ISO/IEC, 2015]. Os modelos possuem guias que tem como objetivo facilitar a utilização dos modelos. Eles descrevem sugestões de implementação para cada nível dos modelos MPS ou particularidade de uma determinada área de negócio para a qual um dos modelos pode ser utilizado ou mapeamentos entre os modelos e outras normas/modelos existentes no mercado. As sugestões de implementação presentes nos Guias de Implementação não constituem requisitos do modelo e devem ser consideradas apenas de caráter informativo.

Fonte: Softex (2016, p. 13).



Reprodução proibida. Art. 184 do Código Penal e Lei 9.610 de 19 de fevereiro de 1998.

TESTE DE SOFTWARE

Segundo Pressman e Maxim (2016), a qualidade de software é determinada pela qualidade dos processos que são usados durante a fase de desenvolvimento do software.

De acordo com Molinari (2003), todo software que se destina ao público e/ou ao mercado deve sofrer um nível mínimo de teste. Assim, quanto maior o nível de complexidade do software, mais testes e técnicas se tornam necessários para a obtenção de sua qualidade.

Isso se deve, visto que, sem os testes, não se é possível garantir que o software irá se comportar conforme o esperado ou de acordo com a solicitação do cliente, o que pode ser negativo para a empresa que o desenvolveu. Entretanto, caso a empresa não possua uma análise de requisitos ou uma documentação do

software detalhada, a equipe de desenvolvimento e a equipe de teste não sabem se o que está sendo construído é o que o cliente espera.

Molinari (2003) sustenta que há alguns axiomas e conceitos que podem ser usados no processo de teste e que, em muitos casos, são considerados verdades no mundo dos testes. São eles:

- Não é possível testar um programa completamente.
- Teste de software é um exercício baseado em risco.
- Teste não mostra que *bugs* não existem, mas sim, o contrário.
- Quanto mais *bugs* são encontrados, mais *bugs* poderão aparecer.

O teste de software tem, como objetivo, mostrar que um sistema está de acordo com as especificações descritas no documento de requisitos e que atende as expectativas do cliente comprador do sistema. Esse processo envolve a verificação dos processos em cada estágio do processo de software, desde a definição dos requisitos dos usuários até o desenvolvimento de cada um dos programas que compõem o sistema. Todavia, a maior parte dos custos de validação é observada depois da implementação, quando o sistema é testado.

Para Tsui e Karam (2013), exceto os programas mais simples, os testes de software não podem provar que um produto funciona, mas apenas encontrar defeitos. Além disso, podem afirmar que o produto funciona para as situações em que foi testado, mas não garantem outras situações que não foram testadas. O autor também afirma que os testes de software, geralmente, possuem dois objetivos principais:

- Encontrar defeitos no software, para que eles possam ser corrigidos ou minimizados.
- Fornecer uma avaliação geral de qualidade e uma estimativa das possíveis falhas.

O processo de testes pode ocupar grande parte do cronograma de desenvolvimento de sistema, dependendo do cuidado com que tenham sido executadas as atividades iniciais de especificação, projeto e implementação.

Agora, vamos conhecer algumas definições sobre teste de software que constam na literatura:

- Testar é verificar se o software está fazendo o que deveria fazer, de acordo com seus requisitos, e não está fazendo o que não deveria fazer (RIOS; MOREIRA FILHO, 2013).
- Testar é o processo de executar um programa ou sistema com a intenção de encontrar defeitos (teste negativo) (MYERS, 1979).
- Testar é qualquer atividade que, a partir da avaliação de um atributo ou capacidade de um programa ou sistema, seja possível determinar se ele alcança os resultados desejados (HETZEL, 1988).



SAIBA MAIS

O teste de software é uma das atividades que buscam contribuir para a melhoria da qualidade do software. O teste revela a presença de defeitos no software e atende as exigências de qualidade de software. O objetivo do presente trabalho foi esclarecer como o teste de software influencia a qualidade do software no mercado. A metodologia utilizada para desenvolvimento do trabalho foi baseada em revisão bibliográfica, em materiais publicados em livros, revistas, jornais, rede eletrônica, periódicos especializados, monografias e dissertações como fonte de coleta de dados. Diante da incessante competitividade do mercado atual, empresas procuram garantir a qualidade de seus produtos como fator que represente sua capacidade em desenvolver sistemas com qualidade. No mercado existem várias técnicas de teste de software que auxiliam essa garantia.

Fonte: Barbosa e Torres (2011, p. 49).

CONCEITOS BÁSICOS DE TESTE DE SOFTWARE

Para facilitar o entendimento do processo de teste de software, alguns conceitos devem ser vistos. Entre tantos, está a diferença entre erro e defeito. Você sabe a diferença entre eles?

De acordo com a terminologia padrão para a Engenharia de Software do *Institute of Electrical and Electronics Engineers* (IEEE), defeitos são considerados parte do universo físico, são provocados por pessoas e podem ocasionar erros em um software, fazendo que ele fique diferente do que foi especificado. Para se esclarecer ainda mais, segundo Bastos *et al.* (2007), um defeito pode ocorrer em função de desvios do que foi levantado na análise de requisitos. Assim, segundo os autores, existem:

- Defeitos decorrentes da falta de concordância com a especificação do produto.
- Defeitos decorrentes de situações inesperadas, mas não definidas nas especificações do produto.

Para Pressman e Maxim (2016, p. 468) o teste de software “é um elemento de um tópico mais amplo, muitas vezes conhecido como verificação e validação (V&V)”. Nele, a verificação diz respeito ao conjunto de tarefas que visa garantir que o software teve suas funções específicas implementadas corretamente. Já a validação se constitui como um conjunto de tarefas que objetiva assegurar que o software foi criado e pode ser verificado com base nos requisitos do cliente.

As atividades de verificações e validações são consideradas úteis na garantia da qualidade do software, principalmente porque são fortemente recomendadas pelos modelos atuais de qualidade de software, tais como as normas ISO e o CMMI. Isso se deve, pois, assim como Bastos *et al.* (2007, p. 30) afirmam, a “verificação realiza inspeções e revisões sobre os produtos gerados pelas diversas etapas do processo de teste”, enquanto a validação “avalia se o sistema atende aos requisitos do projeto (usuário)”.

Para facilitar o entendimento sobre a diferença entre verificação e validação, devemos responder a duas perguntas:

Verificação: “Estamos construindo corretamente o sistema?”

Validação: “Estamos construindo o sistema correto?”

Ainda, segundo Bastos *et al.* (2007), a primeira questão diz respeito ao que foi construído, enquanto a segunda se refere ao entendimento do que era para ser construído.

TÉCNICAS DE TESTE DE SOFTWARE E TIPOS DE TESTES

As técnicas de teste são procedimentos técnicos e gerenciais que ajudam na avaliação e nas melhorias do processo de software. Segundo Tsui e Karam (2013), temos uma grande variedade de técnicas de teste que podem ser aplicadas em diferentes cenários. Assim, elas podem ser utilizadas para classificar:

- Diferentes conceitos de testes de software.
- Técnicas que envolvem o design de testes e suas situações.
- Técnicas de execução de teste e organizações de testes de software.

Além disso, a fase de testes de software pode ser dividida em duas técnicas: funcional e estrutural.

Testes funcionais: garantem o atendimento aos requisitos do sistema, ou seja, que os requisitos estão corretamente codificados. São conhecidos como testes de caixa preta:



Figura 4 – Técnica de teste funcional
Fonte: Pressman e Maxim (2016, p. 470).

Para Bastos *et al.* (2007), os testes funcionais ou de caixa preta são projetados para garantir que os requisitos e as especificações do sistema tenham sido atendidos. Para tanto, eles utilizam as especificações do documento de análise de requisitos e de projetos para definir os testes a serem realizados. Assim, no teste funcional, o componente de software a ser testado é abordado como se fosse uma caixa preta, sem considerar o seu comportamento interno.

Além disso, os testes de caixa preta não verificam como ocorre o processamento, mas apenas os resultados produzidos. Bastos *et al.* (2007) explanam, ainda, que esse teste pode encontrar funções incorretas ou que estejam faltando, erros de interface, erros em estruturas de dados ou acesso a bases de dados externas, erros de comportamento ou de desempenho e, por último, erros de inicialização e término.

Testes estruturais: garantem que os sistemas sejam estruturalmente sólidos e que funcionem no contexto técnico em que serão instalados. São conhecidos como testes de caixa branca:

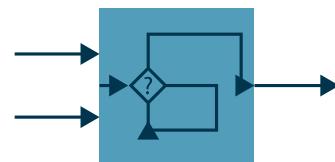


Figura 5 – Técnica de teste estrutural
Fonte: Pressman e Maxim (2016, p. 471).

Para Pressman e Maxim (2016) essa técnica não é desenhada para garantir que o sistema esteja funcionalmente correto, mas para que ele seja estruturalmente robusto. Os testes estruturais ou de caixa branca estabelecem os objetivos do teste com base em uma determinada implementação, analisando os detalhes do código fonte. Assim, todas as variações originadas por estruturas de condições/ repetições do código são testadas.

Portanto, os testes de caixa branca podem, segundo Pressman e Maxim (2016), garantir que todos os caminhos independentes de um módulo sejam executados pelo menos uma vez, exercitando todas as decisões lógicas nos seus estados verdadeiro e falso, executando todos os ciclos em seus limites e dentro de suas fronteiras operacionais e, por último, exercitando estruturas de dados internas para assegurar a sua validade.

Estas técnicas ajudam o processo de software a assegurar o funcionamento adequado de alguns aspectos do sistema ou da unidade. No Quadro 4, podemos visualizar uma lista com alguns tipos de testes que podem ser utilizados:

Quadro 4 – Alguns tipos de testes

TIPO DE TESTE	DESCRIÇÃO
Teste de Unidade	Teste em um nível de componente ou classe. É o teste cujo objetivo é um “pedaço do código”.
Teste de Integração	Garante que um ou mais componentes combinados (ou unidades) funcionam. Podemos dizer que um teste de integração é composto por diversos testes de unidade.
Teste Positivo-Negativo	Garante que a aplicação vai funcionar no “caminho feliz” de sua execução e funcionará no seu fluxo de exceção.

TIPO DE TESTE	DESCRIÇÃO
Teste de Interface	Verifica se a navegabilidade e os objetivos da tela funcionam assim como foram especificados e se atendem da melhor forma ao usuário.
Teste de Aceitação do Usuário	Testa se a solução será bem vista pelo usuário. Ex: caso exista um botão pequeno demais para executar uma função, isso deve ser criticado em fase de testes (aqui, cabem quesitos fora da interface também).
Teste de Volume	Testa a quantidade de dados envolvidos (pode ser pouca, normal, grande ou além de grande).
Testes de Configuração	Testa se a aplicação funciona corretamente em diferentes ambientes de hardware ou de software.
Testes de Instalação	Testa se a instalação da aplicação foi bem sucedida.
Teste de Sistemas	Testa a execução do sistema como um todo, a fim de validar a exatidão e perfeição na execução de suas funções.
Teste de Usabilidade	Testa e simula as condições de utilização do software sob a perspectiva do usuário final. Esses testes focalizam a facilidade de navegação entre as telas, clareza dos textos e as mensagens que são apresentadas aos usuários, dentre outros aspectos da interface do sistema.
Testes de Progressão	Testa apenas as funcionalidades (ou requisitos não funcionais) especificadas para a versão.
Teste de Fumaça	Teste o qual acontece rapidamente, executando as principais funcionalidades do sistema sem se preocupar com as condições de erro. É o mesmo que o teste do “caminho feliz”.

Fonte: adaptado de Bastos *et al.* (2011).

Na literatura, acerca dos testes de software, alguns autores apresentam uma definição de uma técnica de teste chamada de “**caixa cinza**”, a qual mescla as técnicas de caixa preta com a caixa branca:

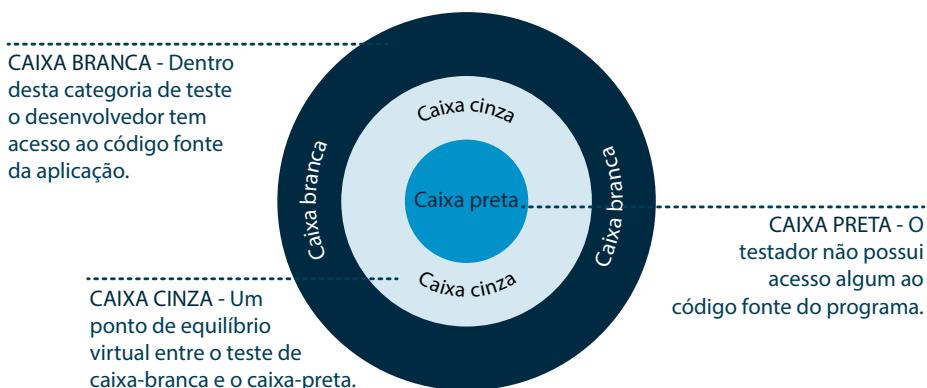


Figura 6 – Técnicas de teste de software.

Fonte: adaptado de Pressman e Maxim (2016).

Para Pressman e Maxim (2016), o teste dificilmente chega ao fim. O que acontece é uma transferência do desenvolvedor para o seu cliente, ou seja, toda vez que um cliente usa o sistema, um teste está sendo realizado.



SAIBA MAIS

O sistema de software comercial tem de passar por três estágios de teste:

1. Testes em desenvolvimento, em que o sistema é testado durante o desenvolvimento para descobrir bugs e defeitos. Projetistas de sistemas e programadores podem estar envolvidos no processo de teste.
2. Testes de release, em que uma equipe de teste independente testa uma versão completa do sistema antes que ele seja liberado para os usuários.
3. Testes de usuário, em que os usuários ou potenciais usuários de um sistema testam o sistema em seu próprio ambiente.

Para produtos de software, o “usuário” pode ser um grupo de marketing interno, que decidirá se o software pode ser comercializado, liberado e vendido. Os testes de aceitação são um tipo de teste de usuário no qual o cliente testar formalmente o sistema para decidir se ele deve ser aceito por parte do fornecedor do sistema ou se é necessário um desenvolvimento adicional.

Fonte: Sommerville (2011, p. 147).



Evolução de Software

Depois que o software é posto em funcionamento, ou seja, depois que é implantado, com certeza ocorrerão mudanças que levarão a realização de sua alteração. Essas mudanças podem ser, de acordo com Pressman e Maxim (2016), para correção de erros não detectados durante a etapa de validação do software, quando há adaptação a um novo ambiente, quando o cliente solicita novas características ou funções ou, ainda, quando a aplicação passa por um processo de reengenharia para proporcionar benefício em um contexto moderno.

Para Sommerville (2011), historicamente, sempre houve uma fronteira entre o processo de desenvolvimento de software e o processo de evolução desse mesmo software (manutenção de software). Isso se deve, pois o desenvolvimento de software é visto como uma atividade criativa, em que o software é desenvolvido a partir de um conceito inicial até chegar ao sistema em operação.

Depois que esse sistema entra em operação, inicia-se a manutenção de software, no qual ele é modificado. Normalmente, os custos de manutenção são maiores do que os custos de desenvolvimento inicial, mas, para muitas empresas,

os processos de manutenção são considerados menos desafiadores do que o desenvolvimento de um software original, ainda que tenha um custo mais elevado.

No entanto, atualmente, os estágios de desenvolvimento e manutenção têm sido considerados **integrados** e **contínuos**, em vez de dois processos separados. Assim, tem sido mais realista pensar na engenharia de software enquanto um processo evolucionário, em que o software é sempre mudado ao longo de seu ciclo de vida em resposta aos requisitos, os quais estão em constantes modificações, e às necessidades do cliente:

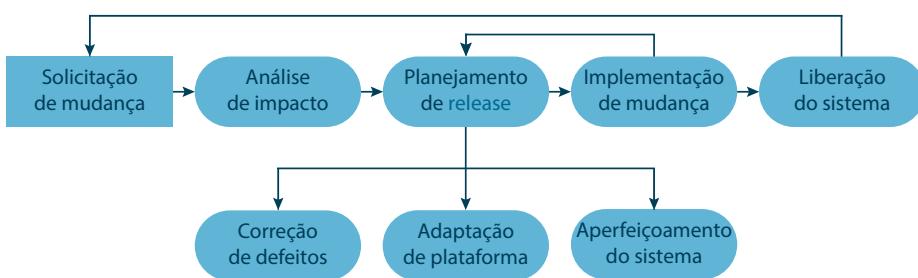


Figura 7 – Evolução do software.

Fonte: Sommerville (2011, p. 181).

MANUTENÇÃO DE SOFTWARE

Após a implantação de um sistema, é inevitável que ocorram mudanças, seja para pequenos ajustes após a implantação, para melhorias substanciais, por força da legislação, para atender novos requisitos dos usuários, ou por, finalmente, estar com erros.

De acordo com Sommerville (2011), cerca de dois terços dos custos de software estão relacionados com a evolução do software, o que requer grande parte do orçamento de Tecnologia da Informação (TI) para todas as empresas. O desafio da manutenção começa assim quando o software é colocado em funcionamento.

Normalmente, depois que os usuários começam a utilizar o software, eles percebem que outras funcionalidades (ainda inexistentes) podem ser acrescentadas. Em outras palavras, aparecem requisitos que o usuário não havia mencionado,

pois foi com o uso do sistema que ele passou a perceber que o software pode oferecer mais do que está oferecendo.

Como o sistema já está em funcionamento, essas novas funcionalidades são consideradas manutenção. Assim, a manutenção se dá para a correção de erros no sistema, pois descobrir todos os erros enquanto o software está na etapa de validação é bastante complexo, dado que todos os caminhos possíveis dentro dos programas teriam que ser testados e nem sempre isso é possível. O fato é que a manutenção sempre vai existir e consumirá bastante tempo por parte da equipe de desenvolvimento.

Uma das razões para o problema acerca da manutenção de software se dá na troca das pessoas que compõem as equipes de desenvolvimento, visto que, possivelmente, a equipe que desenvolveu o software inicialmente já não se encontre mais por perto, ou ainda, que ninguém que esteja trabalhando atualmente na empresa tenha participado da concepção do sistema. Nesse caso, se o sistema desenvolvido estiver bem documentado e, em seu desenvolvimento, tenham sido seguidos os preceitos da engenharia de software, com certeza, sua alteração se tornará mais fácil e é possível afirmar que o sistema apresenta uma alta manutenibilidade.

Para Pressman e Maxim (2011, p. 797), “no nível organizacional, a manutenção é executada por pessoal de suporte que faz parte da organização de engenharia de software”. Entretanto, você pode estar se perguntando: “por que a manutenção do software é importante?” Pense em como o mundo muda rapidamente. As demandas por tecnologia de informação que suportem essas mudanças e exigências no mercado impõem um ritmo que de enorme pressão competitiva em todas as organizações comerciais. É por essa razão que o software deve ser mantido continuamente, ou seja, deve sempre passar por manutenções.

A manutenção e o suporte de software são atividades contínuas que ocorrem por todo o ciclo de vida de um aplicativo. Durante essas atividades, defeitos são corrigidos, aplicativos são adaptados a um ambiente operacional ou de negócio em mutação, melhorias são implementadas por solicitação dos interessados e é fornecido suporte, visto que:

Softwares são produtos mutáveis, isto é, sofrem diversas mudanças e correções ao longo de sua vida útil. A evolução do software é necessária

também para manter sua utilidade/aplicabilidade, pois caso o cenário de utilização/contexto mude, ele precisa acompanhar/ser atualizado, caso contrário se tornará inútil e até mesmo perigoso para a organização (PRESSMAN; MAXIM, 2016, p. 797).

Segundo Sommerville (2011), existem três tipos diferentes de manutenção de software:

Quadro 5 – Tipos de manutenção de software

1. CORREÇÃO DE DEFEITOS	Erros de codificação são relativamente baratos para serem corrigidos e erros de projeto são mais caros, pois podem implicar na reescrita de vários componentes de programa. Erros de requisitos são os mais caros para se corrigir, devido ao extenso reprojeto de sistema que pode ser necessário.
2. ADAPTAÇÃO AMBIENTAL	Esse tipo de manutenção é necessário quando algum aspecto do ambiente do sistema, como o hardware, a plataforma do sistema operacional ou outro software de apoio, sofre uma mudança. O sistema de aplicação deve ser modificado para se adaptar a essas mudanças de ambiente.
3. ADIÇÃO DE FUNCIONALIDADE	Esse tipo de manutenção é necessário quando os requisitos de sistema mudam em resposta às mudanças organizacionais ou de negócios. A escala de mudanças necessárias para o software é, frequentemente, muito maior do que para os outros tipos de manutenção.

Fonte: adaptado de Sommerville (2011).

Pesquisas feitas em empresas de desenvolvimento mostram que a manutenção de software ocupa uma proporção maior dos orçamentos do que o desenvolvimento em si. Em outras palavras, a manutenção detém, aproximadamente, dois terços do orçamento, contra um terço para desenvolvimento do software. A pesquisa também mostrou que se gasta mais do orçamento na implementação de novos requisitos do que na correção de *bugs*.

A Figura 8 mostra, aproximadamente, a distribuição do esforço dos custos de manutenção de software. Para Sommerville (2011), as porcentagens podem variar de uma empresa para outra, mas, universalmente, a correção de defeitos

não é a atividade de manutenção mais cara. Evoluir o software para lidar com novos ambientes de tecnologias e novos ou alterados requisitos consomem mais esforços de manutenção:

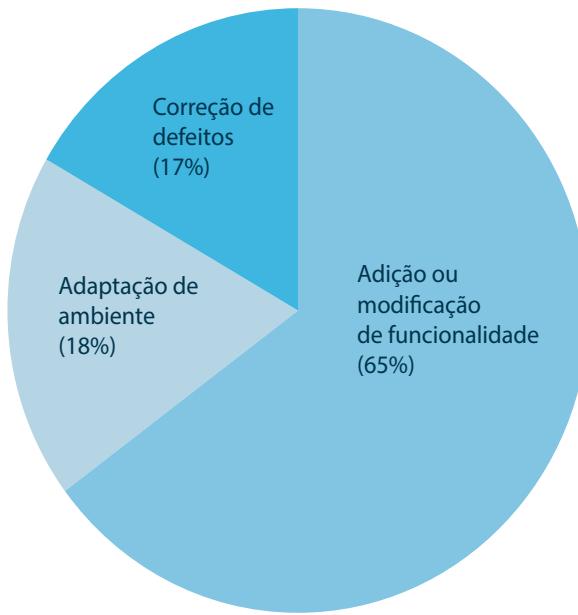
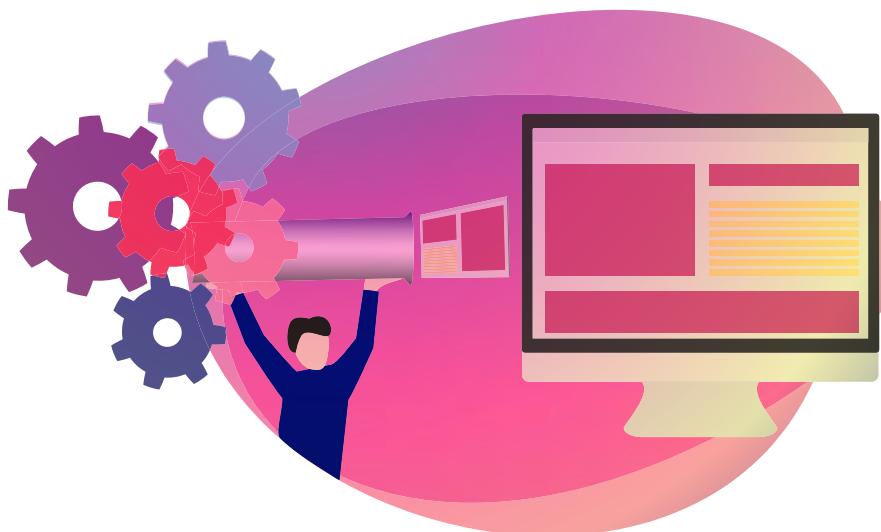


Figura 8 – Distribuição do esforço de manutenção

Fonte: Sommerville (2011, p. 185).

A manutenção de software é considerada um processo de mudanças de um sistema, o qual está em uso pelo cliente e, geralmente, é aplicado em softwares que são desenvolvidos sob encomenda. As mudanças, segundo Sommerville (2011), podem ser simples, a fim de se corrigir erros de codificação, mais extensas, para corrigir erros que ocorreram no projeto, ou, ainda, apenas melhorias para corrigir erros de especificações ou para adaptar novos requisitos que o cliente tenha solicitado.



CONFIGURAÇÃO DE SOFTWARE

O Gerenciamento de Configuração de Software (SCM – *Software Configuration Management*) ou GCS é uma atividade de apoio destinada a gerenciar as mudanças, identificando os artefatos que precisam ser alterados, as relações entre eles e o controle de versão desses artefatos, controlando essas mudanças, auditando e relatando todas as alterações feitas no software (HUZITA; FREITAS, 2019).

A SCM é uma atividade do tipo “guarda-chuva”, aplicada no decorrer de toda a gestão de qualidade. As mudanças podem ocorrer em qualquer instante, por isso, as atividades de SCM são desenvolvidas para:

- Identificar a alteração feita.
- Controlar a alteração que está sendo feita.
- Assegurar que a alteração esteja sendo implementada corretamente no software.
- Relatar as alterações aos membros da equipe.

Como as mudanças podem ocorrer a qualquer momento em um software, o SCM envolve quatro atividades, conforme descrito no Quadro 6:

Quadro 6 - Atividades de Gerenciamento de Configuração de Software (SCM)

1. GERENCIAMENTO DE MUDANÇAS	Envolve manter o acompanhamento das solicitações dos clientes e desenvolvedores por mudanças no software, definir os custos e o impacto de fazer tais mudanças, bem como decidir se e quando as mudanças devem ser implementadas.
2. GERENCIAMENTO DE VERSÕES	Envolve manter o acompanhamento de várias versões de componentes do sistema e assegurar que as mudanças nos componentes, realizadas por diferentes desenvolvedores, não interferem uma nas outras.
3. CONSTRUÇÃO DO SISTEMA	É o processo de montagem de componentes de programas, dados e bibliotecas e, em seguida, compilação e ligação destes, para criar um sistema executável.
4. GERENCIAMENTO DE RELEASES	Envolve a preparação de software para o release externo e manter o acompanhamento das versões de sistema que foram liberadas para uso do cliente.

Fonte: Sommerville (2011, p. 476).

Qual é a diferença entre suporte de software e gerenciamento de configuração de software? É importante saber a distinção entre eles:

Suporte é um conjunto de atividades de engenharia que ocorrem depois que o software foi fornecido ao cliente e posto em operação. Gestão de configuração é um conjunto de atividades de rastreamento e controle iniciadas quando um projeto de engenharia de software começa e termina apenas quando o software sai de operação (PRESSMAN; MAXIM, 2016, p. 623).

A Figura 9 mostra o relacionamento entre as atividades de Gerenciamento de Configuração de Software:

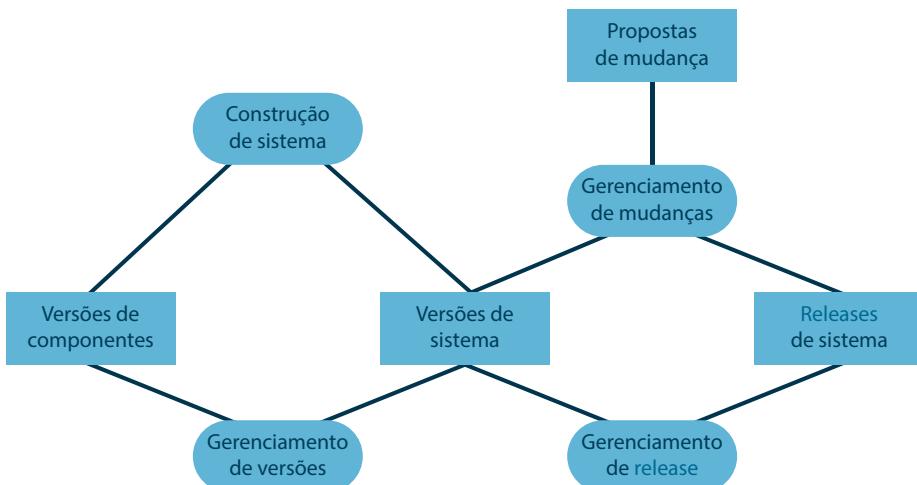


Figura 9 – Atividades de Gerenciamento de Configuração de Software
Fonte: Sommerville (2011, p. 476).



SAIBA MAIS

Muitas alterações de software são justificadas; portanto, não faz sentido reclamar delas. Em vez disso, esteja certo de ter os mecanismos prontos para cuidar delas.

Fonte: Pressman e Maxim (2016, p. 627).

A SCM possui um conjunto de atividades que são desenvolvidas para controlar as mudanças que podem ocorrer ao longo do ciclo de vida de um software. Assim, ela pode ser entendida como uma atividade de garantia de qualidade do sistema e pode ser aplicada em todo o processo do software. As informações que são resultadas do processo de software podem ser divididas em três categorias principais:

- Programas de computador (forma de código-fonte ou executável).
- Produtos que descrevem os programas de computador (focado em vários interessados do sistema – analistas, desenvolvedores, testadores, suporte etc.).
- Dados ou conteúdo (contidos nos programas ou externos a ele).

Segundo Pressman e Maxim (2016, p. 624), “os itens que compõem todas as informações produzidas como parte do processo de software são chamados coletivamente de configuração de software”. Além disso, conforme se avança no desenvolvimento do software, cria-se uma hierarquia dos itens de configuração de software, a qual corresponde a um elemento de informação que contém um nome que pode ser pequeno, como um diagrama UML, ou grande, como um documento de projeto de software completo.

Todavia, um dos problemas do gerenciamento de configurações, de acordo com Sommerville (2011, p. 476), «é que diferentes empresas falam sobre os mesmos conceitos usando termos diferentes». A seguir, apresentamos um quadro com algumas das terminologias usadas no SCM:

Quadro 7 - Terminologias usadas no Gerenciamento de Configuração de Software

TERMO	EXPLICAÇÃO
Item de Configuração ou item de configuração de software (SCI, do inglês <i>Software Configuration Item</i>)	Qualquer coisa associada a um projeto de software (projeto, código, dados de teste, documentos etc) que tenha sido colocado sob controle de configuração. Muitas vezes, existem diferentes versões de um item de configuração. Itens de configuração têm um nome único.
Controle de Configuração	O processo de garantia de que versões de sistemas e componentes sejam registradas e mantidas para que as mudanças sejam gerenciadas e todas as versões de componentes sejam identificadas e armazenadas por todo o tempo de vida do sistema.

TERMO	EXPLICAÇÃO
Versão	Uma instância de um item de configuração que difere de alguma forma, de outras instâncias deste item. As versões sempre têm um identificador único, o qual é geralmente composto pelo nome do item de configuração mais um número de versão.
Baseline	Uma <i>baseline</i> é uma coleção de versões de componentes que compõem um sistema. As <i>baselines</i> são controladas, o que significa que as versões dos componentes que constituem o sistema não podem ser alteradas. Isso significa que deveria sempre ser possível recriar uma <i>baseline</i> a partir de seus componentes.
Codeline	Uma <i>codeline</i> é um conjunto de versões de um componente de software e outros itens de configuração dos quais esse componente depende.
Mainline	Trata-se de uma sequência de <i>baselines</i> que representam diferentes versões de um sistema.
Release	Uma versão de um sistema que foi liberada para os clientes (ou outros usuários em uma organização) para uso.
Branching	Trata-se da criação de uma nova <i>codeline</i> de uma versão em uma <i>codeline</i> existente. A nova <i>codeline</i> e uma <i>codeline</i> existente podem, então, ser desenvolvidas independentemente.
Merging	Trata-se da criação de uma nova versão de um componente de software, fundindo versões separadas em diferentes <i>codelines</i> . Essas <i>codelines</i> podem ter sido criadas por um <i>branch</i> anterior de uma das <i>codelines</i> envolvidas.

Fonte: Sommerville (2011, p. 477).

SAIBA MAIS



Para os engenheiros de software, o objetivo é trabalhar eficazmente. Isso significa que os engenheiros não interferem uns com os outros de forma desnecessária na criação e teste do código e na produção de artefatos de suporte. Mas, ao mesmo tempo, eles tentam se comunicar e coordenar eficientemente. Os engenheiros usam ferramentas que ajudam a criar artefatos consistentes. Eles se comunicam e se coordenam notificando uns aos outros sobre as tarefas necessárias e as tarefas completadas. As alterações são propagadas por meio do trabalho dos outros mesclando arquivos. Existem mecanismos que asseguram que, para componentes submetidos a alterações simultâneas, há uma maneira de resolver conflitos e mesclar alterações. É mantido um histórico da evolução de todos os componentes do sistema juntamente com um registro (log) com as razões para as alterações e um registro do que realmente foi alterado. Os engenheiros têm seu próprio espaço de trabalho para criar, alterar, testar e integrar o código.

Fonte: Pressman e Maxim (2016, p. 516).

Para finalizar, é importante que tenhamos em mente que a SCM é uma atividade abrangente e aplicada em todo o processo de software. Ela identifica, controla, realiza auditoria e relata modificações que podem ocorrer invariavelmente enquanto o software está sendo desenvolvido e até mesmo depois que está sendo usado pelo cliente. Todos os artefatos criados como parte do processo de desenvolvimento tornam-se parte de uma configuração de software.

CONSIDERAÇÕES FINAIS

Nesta última unidade, conseguimos fechar todas as etapas do processo de software. Em outras palavras, já tínhamos estudado a importância de definirmos bem os requisitos do sistema e de deixar isso devidamente anotado em um documento de requisitos.

Assim, aprendemos a modelagem de software e UML, bem como estudamos a qualidade, os testes, a evolução, a manutenção e a configuração de software. Depois que o software foi implementado, é hora dos testes, ou seja, é hora de verificar se ele realmente está funcionando. Enquanto o sistema está sendo desenvolvido, ele já está sendo testado pelas pessoas que o estão desenvolvendo, mas isso só não basta.

Após a implantação e efetiva utilização do sistema pelo usuário, qualquer alteração que seja necessária será considerada uma manutenção do software. Se um software tiver uma vida longa, passará por manutenção durante esse período e, para que continue manutenível, constatamos que é necessário manter a aplicação das técnicas de engenharia de software, pois nem sempre quem desenvolve é quem vai realizar a manutenção no software.

Com isso, chegamos ao final das atividades básicas do processo de software. Espero que você, aluno(a), tenha conseguido entender os conceitos explorados nesta unidade e ao longo do livro, pois, se você entendeu, conseguirá compreender qualquer processo de software que possa vir a ser adotado pela empresa que você trabalha (ou trabalhará) como desenvolvedor(a). Boa sorte!

ATIVIDADES



1. É Importante fazer uma distinção clara entre suporte de software e gestão de configuração de software. Suporte é um conjunto de atividades de engenharia que ocorrem depois que o software é fornecido ao cliente e posto em operação. Pensando nisso, explique a atividade de gerenciamento de *releases* de sistemas.

2. De acordo com Tsui e Karam (2013), os testes de software, geralmente, possuem dois objetivos principais. Sobre a temática, julgue as afirmativas a seguir:
 - I. Os testes devem encontrar defeitos no software, para que possam ser corrigidos ou maximizados e restaurados.
 - II. Os testes fornecem uma avaliação geral de qualidade e uma estimativa das possíveis falhas.
 - III. Os testes devem encontrar defeitos no software, para que possam ser corrigidos ou minimizados.
 - IV. Os testes geram uma avaliação de cada componente e uma estimativa dos possíveis sucessos.
 - V. Os testes não encontram defeitos, mas apenas fornecem uma avaliação geral de qualidade.

É correto o que se afirma em:

 - a) I, apenas.
 - b) II e III, apenas.
 - c) I, IV e V, apenas.
 - d) II, IV e V.
 - e) I, II, III, IV e V.

3. Para Sommerville (2011), o processo de teste apresenta dois objetivos distintos. Sobre a temática, julgue as afirmativas a seguir:
 - I. Demonstrar ao desenvolvedor e ao cliente que o software atende seus requisitos.
 - II. Descobrir situações em que o software se comporta de maneira incorreta, indesejável ou de forma diferente do que foi especificado.
 - III. Demonstrar ao usuário e ao cliente que o software não atende seus requisitos.

ATIVIDADES



IV. Demonstrar ao desenvolvedor e ao cliente que o software não atende seus requisitos.

V. Descobrir situações em que o software se comporta de maneira correta, desejável ou de forma igual do que foi especificado.

É correto o que se afirma em:

- a) I, apenas.
- b) I e II, apenas.
- c) I e III, apenas.
- d) III e V, apenas.
- e) I, II, III, IV e V.

4. Axiomas e conceitos que podem ser utilizados no processo de teste, em muitos casos, são considerados verdades no mundo dos testes. Sobre os axiomas e conceitos usados, julgue as afirmativas a seguir com (V) para as Verdadeiras e (F) para as Falsas:

- () Não é possível testar um programa completamente.
- () Teste de software é um exercício baseado em risco.
- () Teste não mostra que bugs não existem, mas sim, o contrário.
- () Quanto menos bugs são encontrados, mais bugs poderão aparecer.
- () Teste de software não encontra erros, nem bugs.

A sequência correta é:

- a) V, F, V, F, V.
- b) F, F, V, V, V.
- c) V, V, F, V, V.
- d) F, F, F, V, V.
- e) V, V, V, F, F.

5. Observamos, na etapa de manutenção de software, que, segundo Sommerville (2011), existem três tipos de diferentes de manutenção. Cite e explique esses três tipos.



Testes de usuário

Teste de usuário ou de cliente é um estágio no processo de teste em que os usuários ou clientes fornecem entradas e conselhos sobre o teste de sistema. Isso pode envolver o teste formal de um sistema que foi aprovado por um fornecedor externo ou processo informal em que os usuários experimentam um produto de software novo para ver se gostam e verificar se faz o que eles precisam. O teste de usuário é essencial, mesmo em sistemas abrangentes ou quando testes de release tenham sido realizados. A razão para isso é que as influências do ambiente de trabalho do usuário têm um efeito importante sobre a confiabilidade, o desempenho, a usabilidade e a robustez de um sistema.

Na prática, existem três tipos de testes de usuário:

1. Teste alfa, em que os usuários do software trabalham com a equipe de desenvolvimento para testar o software no local do desenvolvedor.

2. Teste beta, em que um release do software é disponibilizado aos usuários para que possam experimentar e levantar os problemas que eles descobriram com os desenvolvedores do sistema.

3. Teste de aceitação, em que os clientes testam um sistema para decidir se está ou não pronto para ser aceito pelos desenvolvedores de sistemas e implantado no ambiente do cliente.

Em **testes alfa**, usuários e desenvolvedores trabalham em conjunto para testar um sistema que está sendo desenvolvido. Isso significa que os usuários podem identificar os problemas e as questões que não são aparentes para a equipe de testes de desenvolvimento. Os desenvolvedores só podem trabalhar a partir dos requisitos, mas, muitas vezes, estes não refletem outros fatores que afetam o uso prático do software.

O **teste beta** ocorre quando um release antecipado, por vezes inacabado, de um sistema de software é disponibilizado aos clientes e usuários para avaliação.

Testadores beta podem ser um grupo de clientes selecionados, os primeiros a adotarem o sistema. Como alternativa, o software pode ser disponibilizado para uso, para qualquer pessoa que esteja interessada nele. O teste beta é usado principalmente para produtos de software que são usados nos mais diversos ambientes (por oposição aos sistemas customizados, que são geralmente usados em um ambiente definido). O teste beta é essencial para descobrir justamente problemas de interação entre o software e as características do ambiente em que ele é usado.

O **teste de aceitação** é uma parte inerente ao desenvolvimento de sistemas customizados, que ocorre após o teste de release. Engloba o teste formal de um sistema pelo cliente para decidir se ele deve ou não ser aceito. A aceitação designa que o pagamento pelo sistema deve ser feito.

Existem seis estágios no processo de teste de aceitação, entre eles:

1. Definir critérios de aceitação. Esse estágio deve, idealmente, ocorrer no início do processo antes de o contrato do sistema ser assinado. Os critérios de aceitação devem





ser parte do contrato do sistema e serem acordados entre o cliente e o desenvolvedor. Na prática, porém, pode ser difícil definir critérios para o início do processo. Os requisitos detalhados podem não estar disponíveis e podem haver mudanças significativas dos requisitos durante o processo de desenvolvimento.

2. Planejar testes de aceitação. Trata-se de decidir sobre os recursos, tempo e orçamento para os testes de aceitação e estabelecer um cronograma de testes. O plano de teste de aceitação também deve discutir a cobertura dos requisitos exigidos e a ordem em que as características do sistema são testadas.

3. Derivar testes de aceitação. Uma vez que os testes de aceitação tenham sido estabelecidos, eles precisam ser projetados para verificar se existe ou não um sistema aceitável. Testes de aceitação devem ter como objetivo testar tanto as características funcionais quanto as não funcionais (por exemplo, desempenho) do sistema. Eles devem, idealmente, fornecer cobertura completa dos requisitos de sistema.

4. Executar testes de aceitação. Os testes de aceitação acordados são executados no sistema. Idealmente, isso deve ocorrer no ambiente real em que o sistema será usado, mas isso pode ser interrompido e impraticável. Portanto, um ambiente de testes de usuário pode ter de ser configurado para executar esses testes. É difícil automatizar esse processo, pois parte dos testes de aceitação pode envolver testes de interações entre os usuários finais e o sistema. Pode ser necessário algum treinamento para os usuários finais.

5. Negociar resultados de teste. É muito improvável que todos os testes de aceitação definidos passem e que não ocorra qualquer problema com o sistema. Se esse for o caso, então o teste de aceitação está completo, e o sistema pode ser entregue. O mais comum, contudo, é que alguns problemas sejam descobertos. Nesses casos, o desenvolvedor e o cliente precisam negociar para decidir se o sistema é bom o suficiente para ser colocado em uso. Eles também devem concordar sobre a resposta do desenvolvedor para os problemas identificados.

6. Rejeitar/aceitar sistema. Esse estágio envolve uma reunião entre os desenvolvedores e o cliente para decidir se o sistema deve ser aceito. Se o sistema não for bom o suficiente para o uso, então será necessário um maior desenvolvimento para corrigir os problemas identificados. Depois de concluída, a fase de testes de aceitação é repetida.

Nos métodos ágeis, como XP, o teste de aceitação tem um significado bastante diferente. Em princípio, ele compartilha a ideia de que os usuários devem decidir quando o sistema é aceitável. No entanto, no XP, o usuário é parte da equipe de desenvolvimento (isto é, ele ou ela é um testador alfa) e fornece os requisitos de sistema em termos de estórias de usuários. Ele ou ela também é responsável pela definição dos testes, que decide se o software desenvolvido apoia ou não a estória de usuário. Os testes são automatizados, e o desenvolvimento não continua até os testes de aceitação de estória passarem. Portanto, não há uma atividade de teste de aceitação em separado.

Fonte: Sommerville (2011, p. 159-161).

MATERIAL COMPLEMENTAR



LIVRO

Planeje seus testes de software e não morra na praia

Emerson Rios e Ricardo Cristalli

Editora: Imagem Art Studio

Sinopse: um gerente ou um líder de projeto, normalmente, não tem tempo para ficar pesquisando, em livros, as informações que precisa. Por outro lado, a informação, quando encontrada, é apresentada de uma forma complexa e, muitas vezes, inacessível para um acesso rápido. Um livro leve, de fácil leitura, não significa que é um documento superficial. Nós nos preocupamos em passar por todos os tópicos importantes que envolvem o planejamento nos projetos de teste de software.



LIVRO

Garantia da Qualidade de Software

Alexandre Bartie

Editora: Elsevier

Sinopse: totalmente alinhado com as mais modernas metodologias existentes no mercado, este livro te coloca diante dos conceitos mais avançados sobre como aplicar um processo de garantia da qualidade de software em sua empresa. Usando uma abordagem simplificada e de fácil de entendimento, possibilita, aos leitores, assimilar gradualmente os aspectos mais relevantes envolvidos na implantação de um processo de garantia da qualidade de software. Além disso, estabelece uma visão corporativa de qualidade de software e prepara a organização ao desafio de incorporar esses conceitos no seu dia a dia. Combinando visão acadêmica com realidade empresarial, o livro apresenta um modelo metodológico viável tanto para as organizações que nunca iniciaram um SPI (Software Process Improvement) quanto às que buscam atingir os níveis CMM 2 e 3. A busca pela viabilidade na aplicação das melhores práticas voltadas para a garantia da qualidade de software torna este livro uma peça-chave para fazer uma verdadeira revolução na sua organização.



NA WEB

Artigo que traz 3 dicas que podem ser usadas para se ter uma documentação de teste em uma equipe ágil. É excelente e foi escrito por Elias Nogueira, QA Engineer, Agile Coach, Trainer na Adaptworks, professor de pós-graduação na área e palestrante em eventos sobre teste e desenvolvimento de software.

Web: <https://imasters.com.br/agile/3-dicas-para-documentacao-de-teste-em-uma-equipe-agil>.



NA WEB

Artigo que descreve o teste ágil e explica como implementá-lo, a partir do ponto de vista das diversas perspectivas discutidas na Mesa Redonda DFTestes.

Web: <http://tmtestes.com.br/teste-agil-como-implementar/>.

REFERÊNCIAS

- ABNT. **NBR ISO 8402.** Gestão da qualidade e garantia da qualidade – Terminologia. Rio de Janeiro: ABNT, 1994.
- BARBOSA, F. B.; TORRES, I. V. O Teste de Software no Mercado de Trabalho. **Revista Tecnologias em Projeção**, v. 2, n. 1, p. 49-52, 2011.
- BASTOS, A.; RIOS, E.; CRISTALLI, R.; MOREIRA, T. **Base de Conhecimento em Teste de Software**. 2. ed. São Paulo: Martins, 2007.
- COSTA JÚNIOR, P. J. da.; MELO, W. L. **Um processo de inspeção utilizando leitura baseada em perspectiva aplicado à análise de requisitos do unified process**. [S.I.], 2003. Disponível em: http://www.geocities.ws/walcelio_melo/PBR-UP-CITS2003.PDF. Acesso em: 04 jul. 2019.
- HETZEL, W. **The Complete Guide to Software Testing**. Wellesley: QED Information Sciences, 1988.
- HUZITA, E. H. M.; FREITAS, J. A. de. **Tópicos Especiais em Engenharia de Software II**. Maringá: Unicesumar, 2019.
- MACHADO, F. N. R. **Análise e Gestão de Requisitos de Software**: onde nascem os sistemas. 3. ed. São Paulo: Érica, 2016.
- MOLINARI, L. **Testes de Software**: Produzindo Sistemas Melhores e Mais Confiáveis. São Paulo: Érica, 2003.
- MYERS, G. J. **The art of Software Testing**. New York: John Wiley & Sons, 1979.
- PRESSMAN, R. S.; MAXIM, B. **Engenharia de Software**. Uma abordagem profissional. 8. ed. Porto Alegre: McGraw Hill Brasil, 2016.
- REZENDE, D. A. **Engenharia de Software e Sistemas de Informação**. 3. ed. Rio de Janeiro: Brasport, 2005.
- RIOS, E.; MOREIRA FILHO, T. R. **Teste de Software**. 3. ed. Rio de Janeiro: Alta Books, 2013.
- SOFTEX. **Guia Geral MPS de Software**. [S.I.]: Softex, 2016. Disponível em: https://www.softex.br/wp-content/uploads/2016/04/MPS.BR_Guia_Geral_Software_2016-com-ISBN.pdf. Acesso em: 01 jul. 2019.
- SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.
- TSUI, F.; KARAM, O. **Fundamentos de Engenharia de Software**. 2. ed. Rio de Janeiro: LTC, 2013.
- VASCONCELOS, A. M. L.; ROUILLER, A. C.; MACHADO, C. A. F.; MEDEIROS, T. M. M. de. **Introdução à engenharia de software e à qualidade de software**. Lavras: UFLA/FAEPE, 2006.



REFERÊNCIAS

REFERÊNCIA ON-LINE:

- 1 Em: <https://www.devmmedia.com.br/artigo-clube-delphi-62-cmmi/13856>. Acesso em: 01 jul. 2019.



GABARITO

1. A atividade de gerenciamento de *releases* envolve a preparação de software para o *release* externo e o mantimento do acompanhamento das versões de sistema que foram liberadas para uso do cliente.
2. A alternativa correta é a B.
3. A alternativa correta é a B.
4. A alternativa correta é a E.
5. Os três tipos diferentes de manutenção de software são:
 - Manutenção para reparo de defeitos de software.
 - Manutenção para adaptar o software a um ambiente operacional diferente.
 - Manutenção para adicionar funcionalidade ao sistema ou modificá-la.



CONCLUSÃO

Neste livro, procuramos lhe mostrar a importância da disciplina Engenharia de Software e como ela pode ser aplicada durante o desenvolvimento de um sistema. A fim de possibilitar o seu entendimento, na Unidade I, foram estudados os conceitos de software e de engenharia de software. Explicamos, também, que podemos ter várias aplicações para o software, desde o software embutido, que pode estar em sua máquina de lavar roupas até o software que controla um foguete espacial. Con tudo, neste material, procuramos utilizar exemplos que fazem parte do nosso dia a dia, a fim de facilitar o entendimento do problema e sua possível solução.

Já a Unidade II foi dedicada exclusivamente para explanar o que são requisitos de software. Além disso, tratamos a respeito da importância do documento de requisitos, mostrando, inclusive, um exemplo.

Na Unidade III, observamos como, a partir do documento de requisitos, podemos realizar a modelagem de um sistema utilizando a UML. Ainda, nesta unidade, foi entendido, com detalhes, o diagrama de casos de uso, o qual pode ser utilizado como base para criação de outros.

Já na Unidade IV, estudamos a modelagem englobando o diagrama de classes, o diagrama de sequência, o diagrama de estados e o diagrama de atividades, os quais são os principais e mais importantes diagramas da UML.

Para finalizar, trabalhamos, na Unidade V, qualidade, testes e manutenção do software, permitindo que você pudesse entender todas as etapas envolvidas nos modelos de processos de software.

Esperamos ter alcançado o objetivo inicial, que era o de mostrar a importância da Engenharia de Software. Desejamos que você seja muito feliz profissionalmente utilizando os conceitos apresentados aqui e, se pudermos ajudá-lo de alguma forma, estamos à sua disposição. Desejamos muito sucesso e paz.

Prof.^a Márcia.

Prof.^a Janaína.

Prof.^a Talita.

Prof.^o Victor.



ANOTAÇÕES



ANOTAÇÕES



ANOTAÇÕES



ANOTAÇÕES



ANOTAÇÕES



ANOTAÇÕES



ANOTAÇÕES

