



MODELAGEM DE SOFTWARE

Professor Me. Erinaldo Sanches Nascimento
Professora Me. Vanessa Ravazzi Perseguine

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; **PERSEGUINE**, Vanessa Ravazzi; **NASCIMENTO**, Erinaldo Sanches.

Modelagem de Software. Vanessa Ravazzi Perseguine. e Erinaldo Sanches Nascimento
Maringá-Pr.: Unicesumar, 2021.
216 p.
"Graduação - EaD".

1. Modelagem. 2. Software. EaD. I. Título.

ISBN 978-65-5615-613-2

CDD - 22 ed. 005
CIP - NBR 12899 - AACR/2

Ficha catalográfica elaborada pelo bibliotecário
João Vivaldo de Souza - CRB-8 - 6828

Impresso por:

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

Kátia Coelho

Diretoria de Pós-graduação

Bruno do Val Jorge

Diretoria de Permanência

Leonardo Spaine

Diretoria de Design Educacional

Débora Leite

Head de Curadoria e Inovação

Tania Cristiane Yoshiie Fukushima

Gerência de Processos Acadêmicos

Taessa Penha Shiraishi Vieira

Gerência de Curadoria

Carolina Abdalla Normann de Freitas

Gerência de Contratos e Operações

Jislaine Cristina da Silva

Gerência de Produção de Conteúdo

Diogo Ribeiro Garcia

Gerência de Projetos Especiais

Daniel Faverki Hey

Coordenador de Conteúdo

Fabiana de Lima

Designer Educacional

Paulo Victor Souza e Silva

Rossana Costa Giani

Projeto Gráfico

Jaime de Marchi Junior

José Jhonne Coelho

Arte Capa

Arthur Cantareli Silva

Ilustração Capa

Bruno Pardinho

Editoração

Thomas Hudson Costa

Juliana Duenha

Qualidade Textual

Keren Pardini

Anna Clara Gobbi

Ilustração

André Luís Onishi

Bruno Pardinho

Andre Azevedo



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. Vanessa Ravazzi Perseguine

Mestre em Ciências da Computação. Especialista em Gestão e Coordenação Escolar. MBA em Gestão de Projetos. Graduada em Ciências da Computação. Experiência profissional na Gestão de TI, coordenação de cursos acadêmicos e gestão de projetos. Experiência acadêmica na área de Ciências da Computação e Administração, atuando nas áreas: Engenharia de Software e Gerência de Projetos, Sistemas de Informação.

Lattes: <http://lattes.cnpq.br/5951082062477259>

Professor Me. Erinaldo Sanches Nascimento

Mestre em Bioinformática. Especialista em Administração e Desenvolvimento em Banco de Dados. Graduado em Ciência da Computação. Professor na educação superior nos cursos de Tecnologia da Informação. Atuação como professor e coordenador na educação profissional pública do Paraná. Incentivador de programas de letramento digital na educação básica. Experiência profissional no desenvolvimento de projetos computacionais nas áreas de Engenharia de Software e Sistemas de Informação.

Lattes: <http://lattes.cnpq.br/2744461340274680>

MODELAGEM DE SOFTWARE

SEJA BEM-VINDO(A)!

Olá! Bem-vindo(a) à disciplina Modelagem de Software. Nesta disciplina, iremos abordar os conceitos e principais técnicas de modelagem de software.

Iniciaremos nossos estudos na unidade I, com o objetivo de compreender a importância da modelagem ao perceber que, ao desenvolver um projeto, oferecemos condições de personalização total de acordo com todas as exigências do cliente, quanto às características especiais do software, e possibilitamos uma visualização completa da arquitetura antes dela ser construída, assim, todos temos a certeza de que será desenvolvido exatamente o que é pretendido. Para se projetar, torna-se essencial a modelagem de processos. Os modelos oferecem uma visão simplificada da realidade complexa do software, permitindo uma melhor compreensão dessa realidade. Modelar é um meio utilizado para analisar e projetar sistemas de software. O modelo pode representar o software como um todo e partes dele. Essas partes representam as várias visões que podem ser abstraidas do sistema. Os detalhes diferem de acordo com a perspectiva do profissional que cria o modelo e sua necessidade.

Na unidade II, estudaremos os vários tipos de modelos possíveis de serem criados de acordo com cada perspectiva. Estudaremos que é necessário utilizar uma linguagem para a notação dos modelos. Com esse objetivo, utilizaremos a UML® (*Unified Modeling Language*), padrão de aceitação internacional de linguagem para modelagem utilizada no desenvolvimento de software. A utilização de modelos UML® tende à padronização e estruturação, o que facilita a comunicação entre equipe de desenvolvimento, entre os membros e também com o cliente.

Na unidade III, abordaremos a UML®, suas especificações e diagramas.

Na unidade IV, abordaremos as diferentes arquiteturas de software, necessárias para o projeto de software, com as especificações e diagramas da UML®. Para concluir, a unidade V trará a você um estudo de caso que irá te auxiliar a entender na prática a modelagem de software.

Vamos começar nossos estudos?

SUMÁRIO

UNIDADE I

OBJETIVO DA MODELAGEM DE SOFTWARE

15 INTRODUÇÃO

16 CONCEITOS E OBJETIVOS DA MODELAGEM DE SOFTWARE

21 MODELAGEM DE SOFTWARE E O PROCESSO DE DESENVOLVIMENTO

24 IMPORTÂNCIA DA MODELAGEM DE SOFTWARE

26 MODELOS E O QUE ELES MODELAM

27 CONSIDERAÇÕES FINAIS

UNIDADE II

TIPOS DE MODELOS

33 INTRODUÇÃO

34 MODELOS DE CONTEXTO

41 MODELOS DE INTERAÇÃO

49 MODELOS ESTRUTURAIS

56 MODELOS COMPORTAMENTAIS



SUMÁRIO

UNIDADE III

UML®

69 INTRODUÇÃO

70 PILARES DA ORIENTAÇÃO A OBJETOS

82 LINGUAGEM DE MODELAGEM UNIFICADA - UML®

94 DIAGRAMAS UML®

112 CONSIDERAÇÕES FINAIS

UNIDADE IV

PROJETO DE ARQUITETURA DE SOFTWARE

121 INTRODUÇÃO

122 GÊNEROS E ESTILOS DE ARQUITETURA

132 ARQUITETURA DE SOFTWARE ORIENTADO A OBJETOS

141 ARQUITETURA DE SOFTWARE CLIENTE-SERVIDOR

147 ARQUITETURA ORIENTADA A SERVIÇO

154 ARQUITETURA DE SOFTWARE CONCORRENTE E EM TEMPO REAL

162 CONSIDERAÇÕES FINAIS



SUMÁRIO

UNIDADE V

MODELAGEM E ARQUITETURA ORIENTADA A OBJETOS

171 INTRODUÇÃO

172 PROCESSOS DE SOFTWARE

176 ENGENHARIA DE REQUISITOS

183 MODELAGEM DO SISTEMA

190 MODELAGEM ARQUITETÔNICA

196 PROJETO, IMPLEMENTAÇÃO, TESTE E EVOLUÇÃO DO SOFTWARE

204 CONSIDERAÇÕES FINAIS

210 REFERÊNCIAS

216 GABARITO



OBJETIVO DA MODELAGEM DE SOFTWARE

Objetivos de Aprendizagem

- Conceituar modelagem de software.
- Identificar a importância da modelagem de software..

Plano de Estudo

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

- Conceitos e objetivos da modelagem de software
- Modelagem de software e o processo de desenvolvimento
- Importância da modelagem de software
- Modelos e o que eles modelam

INTRODUÇÃO

Olá, caro(a) aluno(a)! Começamos nossos estudos apresentando a contextualização da modelagem e estabelecendo sua importância dentro do processo de desenvolvimento de software.

O sucesso de um projeto de software pode ser determinado por aspectos técnicos e pelo cumprimento das especificações dadas pelo planejamento, entretanto a satisfação geral das partes interessadas é o medidor principal, uma vez que, mesmo cumprindo prazos e custos, se não for entregue um software que resolva os problemas dos interessados, esse projeto pode ser considerado um fracasso. A modelagem de software entra como um processo importante que oferece um canal de comunicação, por meio dos diagramas, promovendo a interação entre os engenheiros de software e o cliente, garantindo a correta interpretação entre o que se pede e o que será desenvolvido.

A modelagem pode ser desenvolvida a qualquer momento dentro do processo de desenvolvimento de software, ela auxilia na interpretação e no levantamento de requisitos e pode atuar como documentação final para a manutenção e rastreabilidade dos requisitos após a implantação.

Estudaremos, nesta unidade, que a modelagem de software tem como objetivo principal dividir o software em partes, utilizando os requisitos como se fossem peças de um quebra-cabeça, para obter visões específicas e direcionadas para diferentes contextos de análise.

Vamos lá!?

CONCEITOS E OBJETIVOS DA MODELAGEM DE SOFTWARE

Ao pesquisar sobre modelagem de software é comum encontrar como definição algo do tipo: um modelo é uma simplificação da realidade. Eu não concordo por um único motivo, um modelo de software busca retratar uma ação ou procedimento que foi descrito e definido por um requisito, geralmente aquele de difícil interpretação, com o objetivo de visualizar todas as ações decorrentes dele. Um modelo não simplifica a realidade, ele a traduz, e realidade, nesse contexto, pode ser encarada como “problema” a ser resolvido, ou melhor, como a função a ser desempenhada; e a representa de uma forma mais lúdica, abstraindo alguns detalhes e focando outros, dependendo da perspectiva de visão e entendimento desejada.

Nunes e O’Neill (2000) apoiam minha teoria quando dizem que a modelagem se traduz como a representação dos aspectos estruturais e das possíveis funções de um programa computacional, por meio de símbolos padronizados com significados específicos, ou seja, a modelagem de software tem como objetivo principal dividir o sistema em partes menores, usando os requisitos como se fossem peças de um quebra-cabeça, para obter uma “imagem” da estrutura e comportamento daquele pedaço específico.

Para promover uma base para a conceituação e entendermos melhor o objetivo da modelagem de software, recorro ao método de associar o significado da palavra a suas tarefas. Entendendo o significado da palavra, fica mais fácil associá-la a sua função no contexto em que é aplicada. O dicionário Michaelis online apresenta a definição a seguir para o verbo modelar:

modelar¹mo.de.lar¹**(modelo+ar²) vtd 1** Fazer o modelo ou o molde de: **Modelar uma estátua.****Modelou a imagem em cera. vtd 2 Pint** Imitar com muita exatidão o relevo ou os contornos de. **vtd 3** Amoldar: **Modelar o bronze. vtd 4** Ajustar-se a, cobrir ou envolver, unir-se bem a, deixando ver a forma do conteúdo: **Uma blusa modelava-lhe o corpo. vtd e vpr 5** Tomar como modelo: **Modelai os vossos atos pelos ensinos do Divino Mestre. Modelava-se pelo exemplo dos grandes homens. vtd 6** Delinear, regular, traçar intelectualmente: **Modelou poemas segundo padrões clássicos.****modelar²**mo.de.lar²**adj (modelo+ar³)** Que serve de modelo; exemplar.

Fonte: Modelar – Michaelis (online)

O Dicionário Online de Português apresenta a seguinte definição:

v.t. Fazer o modelo ou o molde de uma peça.

Denunciar ou insinuar as formas.

Fig. Formar de acordo com um modelo.

Fig. Formar, criar, produzir, dar forma e contornos a.

Fonte: Modelar – Dicionário Online de Português (online).

As definições formais para o verbo modelar nos ensartam ao modelo. Modelar significa fazer um modelo, formar algo. Modelar software é o processo de criação de modelos que representam um software existente ou um que será desenvolvido.



SAIBA MAIS

O não sucesso de projetos de software tem relação com aspectos únicos e específicos de cada projeto, mas todos os projetos de sucesso são semelhantes em vários aspectos. Existem vários elementos que contribuem para um projeto bem-sucedido; um desses componentes é a utilização de modelagem. A escolha dos modelos a serem criados tem profunda influência sobre a maneira como um determinado problema é atacado e como uma solução é definida.

Aprofunde seu conhecimento com a leitura do estudo “Um guia para a criação de modelos de software no Praxys Synergia”, disponível em: <<https://www.dcc.ufmg.br/pos/cursos/defesas/871M.PDF>>. Acesso em: 29 ago. 2015.

Fonte: Rodrigues (2007).

Os objetivos gerais de um modelo de software são relacionados por Pressman (2010, p. 145):

1. Descrever o que o cliente exige;
2. Estabelecer a base para a criação de um projeto de software;
3. Definir um conjunto de requisitos que possam ser validados quando o software for construído.

Sommerville (2011, p. 83), por sua vez, classifica os modelos em função de sua utilização:

1. Como forma de facilitar a discussão sobre um sistema existente ou proposto;
2. Como forma de documentar um sistema existente;
3. Como uma descrição detalhada de um sistema, que pode ser usada para gerar uma implementação do sistema.

Podemos observar que ambos os autores, de sua maneira, apresentam classificações semelhantes para os objetivos da modelagem de software. Quando Pressman sugere que o objetivo do modelo é descrever o que o cliente deseja, coincide com a classificação de Sommerville, que diz que o modelo é uma forma de facilitar a discussão. Ambas garantem que o objetivo é promover a interação entre os engenheiros de software e o cliente, garantindo a correta interpretação entre o que se pede e o que será desenvolvido. O objetivo de Pressman, estabelecer a base para a criação de um projeto de software, é semelhante ao de Sommerville, que afirma que o modelo pode ser usado para gerar uma implementação do sistema. Significa que a partir do modelo é possível ser gerado o código-fonte do sistema, quando ele for desenvolvido com os critérios e detalhes necessários para isso.

Outra palavra bastante associada à modelagem de software e importante de ser conceituada é o termo **abstração**. Abstrair na informática corresponde ao isolamento, à desconsideração de determinados aspectos ou características, com o objetivo de simplificar sua avaliação ou sua interpretação.

Os modelos de software podem ser desenvolvidos em vários níveis de abstração, por exemplo, os engenheiros podem desenvolver um modelo específico para o ponto de vista dos usuários, abstraindo informações técnicas e focando somente cenários de leitura rápida e fácil. A modelagem pode ser baseada em classes, definindo objetos, atributos e relacionamentos, já operando em um nível mais técnico, ou ainda, pode-se desenhar um modelo que representa o fluxo de dados, indicando como os dados são transformados durante a execução das funções do sistema.

A abstração é o aspecto mais importante de um modelo, afirma Sommerville (2011, p. 83). Um modelo deixa de fora alguns detalhes do sistema, portanto ele é uma abstração do sistema. Um modelo seleciona as características mais relevantes e a representa em uma linguagem geralmente gráfica que possibilita a visão do software de diferentes perspectivas, por exemplo:

1. Perspectiva externa: modelagem do ambiente do sistema.
2. Perspectiva de interação: modelagem das interações entre o sistema e o ambiente, ou entre os seus componentes.
3. Perspectiva estrutural: modelagem considerando a estrutura de dados processados pelo sistema.

4. Perspectiva comportamental: modelagem que considera o comportamento dinâmico do sistema, sua reação aos eventos.

Espíndola (online) apresenta esse conceito de uma forma bastante simples em seu artigo “A Importância da Modelagem de Objetos no Desenvolvimento de Sistemas” para o site Linha de Código. Ele diz que o software pode ser analisado sob diferentes aspectos, de acordo com o interesse e necessidade de quem realiza a análise. Por exemplo, o profissional responsável pela criação e manutenção do banco de dados direciona seus modelos para os relacionamentos entre as entidades e tabelas, dando atenção especial aos processos de armazenamento e aos eventos que os iniciam. Já o foco será para os algoritmos, e o fluxo de dados de um evento, para o outro, se a modelagem for designada para as análises de um profissional responsável pela estrutura interna do software, que tem preocupações em manter um código simples e eficiente. Já se o sistema for orientado a objetos, uma modelagem necessária seria em relação à interação entre as classes e o funcionamento delas em conjunto.

Na unidade II, estudaremos os principais tipos de modelos nos aprofundando um pouco mais nesse assunto.



REFLITA

A palavra escrita é um veículo magnífico de comunicação, mas não é necessariamente o melhor modo de representar os requisitos de software para computador.

Fonte: Pressman (2010, p. 144).

MODELAGEM DE SOFTWARE E O PROCESSO DE DESENVOLVIMENTO

O modelo de processo de desenvolvimento de software tradicional recomenda que a modelagem deve ser feita a partir da primeira versão aprovada do documento de requisitos. Na academia, dividimos as etapas desse processo em disciplinas distintas e as ensinamos em sequência para facilitar a didática; já, na prática, a modelagem de software pode ocorrer a qualquer momento, por exemplo, durante o levantamento de requisitos, o engenheiro, que não domina a regra de negócio, precisa de mecanismos e ferramentas que o auxiliem na interpretação de uma determinada rotina, junto com o especialista, ou o cliente, ele representa o fluxo de informações que demandam daquele requisito por meio de desenhos, isto é, um modelo.

Outra situação, quando apresentada, por exemplo, é uma solicitação de mudança ou quando é solicitada uma inclusão de rotina em softwares já em operação; para analisar o impacto dessas alterações por todo o sistema e para encontrar a melhor proposta de código e banco de dados, o analista de sistemas irá observar todas as tabelas e suas ligações, uma forma prática de ser fazer isso é por meio de representações gráficas, que são os modelos. O ponto aonde quero chegar é que, mesmo não utilizando uma linguagem de modelagem formal para desenvolver um software, sempre é feito algum tipo de modelo, entretanto esses modelos informais nem sempre apresentam uma linguagem comprehensível por leitores que não participam do processo.

Sommerville (2011, p. 82) diz que os modelos são usados durante o processo de engenharia de requisitos auxiliando no levantamento, são usados também durante a fase de projeto, auxiliando na interpretação dos requisitos, e também como documentação, registrando a estrutura e a operação do software - a ser desenvolvido ou de um já existente. Pressman (2010, p. 145) acrescenta que, a partir dos requisitos, o modelador (engenheiro ou analista) pode construir modelos que descrevam cenários de usuários, atividades funcionais, classes de problemas e seus relacionamentos, comportamento do sistema e das classes, e fluxo dos dados, à medida que são transformados. Nesse sentido, o modelo de software fica entre uma descrição em nível de sistema e o projeto de software.

A Figura 1 apresenta esse relacionamento.

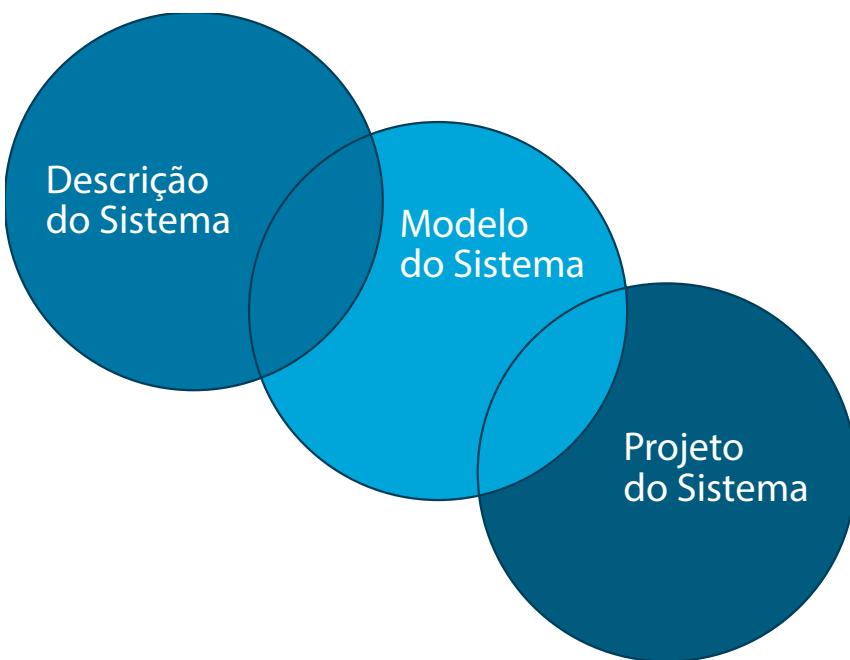


Figura 1: Modelo de Software como uma ponte entre a descrição e o projeto do sistema
Fonte: adaptado de Pressman (2010, p. 146).

Existem no mercado muitas propostas, com vários formatos, roteiros e artefatos, para o processo de desenvolvimento de softwares; cada profissional ou cada empresa vai decidir pelo processo que melhor se adéque a sua realidade e necessidade. Por exemplo, Engholm (2010, p. 66) sugere um roteiro para o desenvolvimento de softwares orientados a objeto, contendo os seguintes processos: elicitação de requisitos, análise de requisitos, arquitetura, design, implementação e testes, propõe que o desenvolvimento de software é a transformação do modelo mental dos envolvidos no projeto (*stakeholders*) para o código e, utilizando a linguagem padronizada UML® (Estudaremos a UML® na unidade V), usa modelos em todas as fases do seu processo.

Morais (2011) observa que os modelos são bem aceitos dentro das metodologias ágeis de desenvolvimento de software. Ele diz que a modelagem ágil procura focar o desenvolvimento de artefatos apenas quando agregarem valor

real ao projeto, diferente da tradicional, que os utiliza. Ela não é prescritiva, fornece propostas para ganhar efetividade. A modelagem ágil procura o equilíbrio entre a criação dos artefatos e o custo de manutenção desses artefatos e, conforme Moraes (2011) afirma, toma por base dois princípios: (1) O objetivo primário de um projeto de software é o próprio software, e não um grande conjunto de documentos sobre ele; (2) um artefato é criado primordialmente para permitir a comunicação e a troca de informações entre a equipe e permitir a discussão e refinamento do modelo do sistema.

Deboni (2003, p. 26) considera a modelagem necessária em todas as etapas do processo de desenvolvimento e nos diz que o processo de desenvolvimento de software é realizado com base na evolução de uma visão que os engenheiros e desenvolvedores constroem em conjunto com o cliente sobre o problema a ser resolvido. Essa visão é traduzida pelos modelos e deve evoluir juntamente com a evolução do processo de desenvolvimento. Complementa que uma única visão não é suficiente para traduzir todas as necessidades e a complexidade do problema, para isso, é necessário um conjunto coerente de modelos em escala diferentes e criadas de diferentes pontos de vista. Ele propõe três tipos de modelo, a fim de garantir a introdução da complexidade nos modelos aos poucos, na medida em que a análise evolui:

1. **Modelo de contexto:** inicia com a definição do problema, e é construído em alto nível de abstração, onde a maioria dos problemas não é considerada. Utilizado para definir as fronteiras do sistema e contextualizar o ambiente.
2. **Modelo conceitual:** esse modelo reúne todos os conceitos dos problemas existentes, construído em um nível menor de abstração que o de contexto. O objetivo nesse ponto da análise é a construção de um modelo simplificado de classes que contenha os principais componentes e suas relações. Esse modelo contém a proposta de solução do problema, mas abstrai os detalhes de implementação.
3. **Modelo detalhado:** esse modelo representa todos os detalhes de uma versão projetada do software, e pode possuir uma equivalência ao código. Desenvolvido em sistemas automatizados de manutenção de modelagem, pode sofrer alterações quando o código for alterado.



IMPORTÂNCIA DA MODELAGEM DE SOFTWARE

A modelagem de software comumente representa o software por meio de algum tipo de notação gráfica. A notação mais comumente utilizada como base para a modelagem é a UML®, que é apresentada por Cardoso (2003, p. 3) como um conjunto de diagramas com suas finalidades, porém, sem nenhuma ligação ou sequência definida pela linguagem, o que não orienta o processo de desenvolvimento, isto é, a UML® oferece suporte gráfico para o entendimento do sistema no ponto em que ele é mais necessário.

Para promover o entendimento sobre a importância dos modelos no desenvolvimento de software, recorro à teoria da imagem para estabelecer um referencial e reforçar meu ponto de vista. Os primeiros vestígios de comunicação humana que se tem conhecimento foram feitas por meio de imagens, as pinturas rupestres (antigas representações pictóricas, datadas do período Paleolítico (100.000-10.000 a.C.).



SAIBA MAIS

A modelagem, do ponto de vista ágil, é um método eficiente que tem como objetivo tornar mais produtivos os esforços da tarefa de modelar, tão comum nos projetos de software. Os valores, princípios e práticas da Modelagem Ágil podem auxiliar as equipes na definição de componentes técnicos de alto e baixo nível que farão parte do desenvolvimento de software. Artefatos sofisticados elaborados por ferramentas de alto custo nem sempre são os melhores para ajudar no desenvolvimento do software. Uma boa prática é modelar o software em grupo e com a participação dos usuários, utilizando rascunhos.

Aprofunde seu conhecimento lendo este artigo na íntegra, disponível em: <<http://www.devmedia.com.br/modelagem-em-uma-visao-agil-engenharia-de-software-32/19006>>. Acesso em: 20 out. 2015.

Fonte: Moraes (2011, online).

Silva (2008) apresenta, em seu estudo sobre a importância da imagem no ensino-aprendizagem, que quando se compara as definições de imagem ao longo

da história é possível perceber que a imagem é conceituada como sendo algo capaz de representar visualmente outra coisa, que pode estar ausente. Seguindo essa linha de raciocínio e considerando a complexidade de se definirem rotinas, algoritmos, cardinalidade, interações por meio da linguagem escrita, a utilização de gráficos e diagramas para a compreensão das funções de um software assume grande importância, como facilitadores na transmissão de informação.

Outro fator que pode ser usado para mensurar a importância dos modelos no processo de desenvolvimento de software é que, mesmo para aqueles leitores não especialistas, a mensagem vinculada na forma de imagem possibilita um entendimento mais fácil, mais direto. Um projeto de software não é composto apenas por elementos tecnológicos, um importante e decisivo elemento que compõe o projeto de software é o social. Os elementos tecnológicos são responsáveis pela construção do software enquanto os elementos sociais, pelo relacionamento entre os desenvolvedores e os técnicos, e devemos sempre considerar que o software é desenvolvido por pessoas para ser utilizado por outras pessoas (DEBONI, 2003, p. 19). A comunicação facilitada pelos modelos pode garantir o sucesso do projeto de software.

Aproveitando ainda o trabalho de Silva (2008), constatamos que a leitura visual sempre foi importante para o homem, desde os primórdios da humanidade, uma vez que as imagens significam mais do que representações de objetos, elas despertam considerações que excedem a percepção visual. Isso quer dizer que o processo de assimilação e retenção da informação transmitida por uma imagem ocorre de forma emocional e subliminar e, por isso, é bem mais fácil de interpretar que o de uma redação ou palavra.

Projetar software é construir um modelo. Um modelo que representa de uma forma mais simples o que se pretende construir. A engenharia de software procura trazer para a ciência da computação o que já é tão utilizado nas outras disciplinas de engenharia, como a civil, onde a figura clássica de um engenheiro civil é uma pessoa envolvida por plantas e diagramas enquanto comanda uma construção (DEBONI, 2003, p. 18).

Concluo essa discussão com as palavras de Confúcio: “uma imagem vale mais que mil palavras”¹, sendo assim, a modelagem de software é a representação gráfica, é a imagem do que será codificado.

¹ Disponível em: <<http://pensador.uol.com.br/frase/NTcxMjMz/>>. Acesso em: 03 set. 2015.



MODELOS E O QUE ELES MODELAM

É óbvio o que eu vou afirmar, mas um modelo é sempre um modelo de alguma coisa. A “coisa” que está sendo modelada, genericamente falando, pode ser considerada um elemento dentro de algum domínio qualquer. O modelo fará declarações sobre esse elemento abstraindo detalhes a partir de certo ponto de vista e para um determinado fim. Não se preocupe, estudaremos todos esses conceitos nas próximas unidades.

Para um sistema existente, o modelo pode representar uma análise das propriedades e do comportamento daquele sistema. Para um sistema em planejamento, o modelo pode representar uma especificação de como o sistema será construído e como deverá se comportar.

Um método de modelagem deve considerar três categorias nas suas propostas de visões (OMG, 2015, p. 12):

- Objetos: suporte para a descrição de um conjunto de objetos que possuem identidade e relacionamentos com outros objetos.
- Eventos: um evento descreve um conjunto de possíveis ocorrências.
- Comportamentos: um comportamento descreve um conjunto de possíveis execuções.

Para modelar, precisamos adotar uma notação padrão: uma ferramenta e uma linguagem que forneça os recursos necessários para a representação de toda a complexidade envolvida no desenvolvimento de um software.

CONSIDERAÇÕES FINAIS

Nesta unidade, estudamos os principais conceitos referentes à atividade de modelagem de software. Foi possível observar que a modelagem deve estar relacionada com o processo de engenharia de requisitos, garantindo uma ponte entre as etapas de definição do sistema e projeto, que representam as atividades mais importantes do processo de desenvolvimento de software.

Sabemos que a perfeita interpretação dos requisitos é essencial para o sucesso do desenvolvimento do software. Os modelos se mostram, nesse contexto, ótimos aliados para promoção da discussão entre a equipe técnica e usuários, pois promovem uma especificação clara e precisa. Dentro do processo de desenvolvimento de software o modelo completa o processo de desenvolvimento garantindo como resultado uma metodologia de desenvolvimento. Outro fator que pode ser usado para mensurar a importância dos modelos no processo de desenvolvimento de software é que, mesmo para aqueles leitores não especialistas, a mensagem vinculada na forma de imagem possibilita um entendimento mais fácil, mais direto. Sistemas de informação estão em constante mudança. Essas mudanças ocorrem por vários fatores, como, por exemplo: os clientes solicitam modificações constantes ou por exigência do mercado que está em mudança constante. Assim, um sistema precisa de documentação detalhada, atualizada e que seja fácil de ser mantida. A modelagem se apresenta também como uma forma bastante eficiente de documentação.

Vimos também que a modelagem de software comumente representa o software por meio de algum tipo de notação gráfica. A notação mais comumente utilizada como base para a modelagem é a UML® e será estudada com mais critério na unidade V. Finalmente estudamos que os modelos de software podem ser desenvolvidos em vários níveis de abstração, que possibilitam a visão do software de diferentes perspectivas, por exemplo: perspectiva externa, perspectiva de interação, perspectiva estrutural e perspectiva comportamental.

ATIVIDADES



1. Defina, com suas palavras, o termo Modelagem de Software.
2. Quais são os objetivos gerais de um modelo de software?
3. O que significa abstração no contexto da modelagem de software?



POR QUE OS PROJETOS DE TI FRACASSAM?

O sucesso de um projeto de TI pode ser determinado pela satisfação geral das partes interessadas (NELSON, 2005). A satisfação das partes interessadas pode ser mensurada a partir da combinação de 6 critérios: critérios de processo – tempo (cumprimento de cronograma), custo (execução dentro do orçamento) e produto (atendimento aos requisitos com qualidade aceitável) – e critérios de resultado – uso (produto efetivamente usado), aprendizado (extração de lições aprendidas relevantes para a organização) e valor (melhoria efetiva do negócio) (NELSON, 2005). É responsabilidade do gerente de projetos estabelecer junto às partes interessadas as métricas de avaliação dos critérios de sucesso. No entanto, a partir de entrevista realizada a um número significativo de grupos de partes interessadas de diferentes projetos, conclui-se que os critérios mais relevantes para o sucesso são produto, uso e valor (NELSON, 2005).

Por outro lado, o fracasso de um projeto de TI pode ser definido como o abandono total do projeto antes ou logo depois de o produto ter sido entregue (CHARETTE, 2005).

Diversos são os fatores que levam projetos de TI ao fracasso. Tais fatores têm impacto direto em um ou mais critérios de satisfação das partes interessadas. Planos de projeto medíocres, que não incluem um planejamento de riscos adequado, *business cases* que não evidenciam a conexão entre as necessidades do projeto e do negócio, baixo suporte e envolvimento da alta gestão e usuários, imaturidade das tecnologias adotadas e indefinição quanto ao responsável pelo negócio estão entre os fatores críticos que levam projetos de TI ao fracasso (WHITTAKER, 1999; YARDLEY, 2002).

Fonte: Atalla (online).

MATERIAL COMPLEMENTAR



LIVRO

Engenharia de Software: Uma Abordagem Profissional

Roger S. Pressman

Editora: Bookman

Sinopse: Versão concebida tanto para alunos de graduação quanto para profissionais da área. Oferece uma abordagem contemporânea sobre produção do software, gestão da qualidade, gerenciamento de projetos, com didática eficiente e exercícios de grande aplicação prática.



TIPOS DE MODELOS

Objetivos de Aprendizagem

- Compreender por que diferentes tipos de modelos são necessários.
- Apresentar as perspectivas fundamentais de modelagem de software

Plano de Estudo

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

- Modelos de contexto
- Modelos de interação
- Modelos estruturais
- Modelos comportamentais

INTRODUÇÃO

Olá! Vamos para mais uma unidade de estudos? Na unidade I, estudamos sobre o auxílio que os modelos oferecem na interpretação e na visualização de um software em projeto, um pedaço dele ou ainda um existente, oferecendo uma ilustração de diferentes perspectivas e níveis de abstração. Quando estamos iniciando na área de engenharia de software, às vezes é complicado entender o que significa “diferentes perspectivas” ou “níveis de abstração”, para facilitar, vamos considerar que estamos na fase de levantamento de requisitos, da engenharia de requisitos, e vamos seguir a classificação feita por Sommerville (2011) para a modelagem de sistemas.

Ele organiza a atividade de modelagem de dados partindo do nível mais alto de abstração, que é o contexto até o nível mais interno, que ele chama de comportamental. Nesta unidade, estudaremos sobre os tipos de modelagem possíveis de desenvolvimento, considerando as perspectivas, ou visões, relacionadas por Sommerville (2011, p. 83): perspectiva externa, perspectiva de interação, perspectiva estrutural, perspectiva comportamental.

Os modelos de contexto representam a perspectiva externa do software, isto é, o seu relacionamento com os elementos exteriores a ele que irão impactar e sofrer impactos pelo seu funcionamento, ou seja, define os limites externos do sistema.

Os modelos de interação são voltados para a perspectiva de interação; oferecem uma visão de como é feito o diálogo entre o sistema e o usuário. Serão apresentados o diagrama de casos de uso e o de sequência da UML®.

Os modelos estruturais representantes da visão estrutural do desenvolvimento de software registram a estrutura interna do software, seus componentes, elementos de banco de dados e os relacionamentos entre eles. Será apresentado o diagrama de classes da UML®. Por último, estudaremos os modelos comportamentais representantes dos procedimentos dinâmicos do software.

MODELOS DE CONTEXTO

Qualquer discussão sobre contexto deve começar focando o termo contexto. Contexto dentro da informática pode assumir diferentes posturas, dependendo do ambiente em que é empregado. Por exemplo, em sistemas computacionais, contexto é um instrumento de apoio à comunicação entre os sistemas e seus usuários, onde, a partir da compreensão do contexto, o sistema pode mudar a sequência de ação, o tipo de informação oferecida ao usuário. As áreas da Computação Ubíqua e Inteligência Artificial foram as pioneiras nos estudos e na utilização do conceito de contexto (VIEIRA; TEDESCO; SALGADO, online). Ainda falando de sistemas adaptativos, os modelos de contexto representam características atuais do usuário e do ambiente (MACHADO; OLIVEIRA, 2013). Para o nosso estudo, contexto também representa o ambiente, mas como ferramenta limitadora.

Os modelos de contexto representam as perspectivas externas de onde será modelado o ambiente do sistema, os seus limites e suas relações técnicas e também as relações não técnicas. Por exemplo, um limite do sistema pode ser estabelecido de forma que o processo da análise ocorra em um site, ou pode ser definido de forma que um gerente, particularmente difícil, não precise ser consultado (SOMMERVILLE, 2011, p. 84).

Nem sempre as barreiras entre um software e seu ambiente são claras, por isso é muito importante que a definição do contexto do sistema seja feita juntamente com os *stakeholders* do cliente. Nesse sentido, os modelos de contexto podem ser desenvolvidos juntamente com as duas primeiras tarefas da engenharia de requisitos, Levantamento e Negociação - estudados na disciplina de Engenharia de Requisitos. Outras atividades da engenharia de requisitos podem ser usadas para identificar interessados, definir escopo do problema, especificar os objetivos operacionais e descrever os objetos que serão manipulados pelo sistema (PRESSMAN, 2010, p. 153).

REFLITA



"Tornou-se chocantemente óbvio que a nossa tecnologia excedeu a nossa humanidade."

Fonte: Albert Einstein.

O modelo de contexto, então, pode ser utilizado como um instrumento da análise funcional que irá identificar os seguintes subsídios:

1. Os elementos externos que irão interagir com o sistema.
2. O fluxo de informação existente entre o sistema e o ambiente externo.
3. Os limites do sistema.
4. Os eventos do ambiente externo que impactam (provocam alterações) o sistema provocando respostas.

Um modelo de contexto pode ser montado utilizando dois componentes, um diagrama de contexto e uma lista de eventos. O diagrama de contexto é uma representação gráfica do sistema com seu ambiente, e a lista de eventos relaciona os episódios do contexto externo que o sistema deve obrigatoriamente considerar.

A Figura 2 representa um diagrama de contexto genérico.



Figura 2: Diagrama de Contexto Genérico

Fonte: a autora.

A Tabela 1, Lista de Eventos Genérica, sugere como uma lista de eventos pode ser construída.

ESTÍMULO	RESPOSTA	EVENTO

Tabela 1: Lista de Eventos Genérica

Fonte: a autora.

Ações para a construção dos artefatos:

1. Identificar os eventos externos.
2. Identificar os fluxos.
3. Produzir a tabela de eventos.
4. Analisar se todos os eventos foram considerados.

Para facilitar a compreensão, vamos analisar alguns exemplos práticos, primeiro o desenvolvido por Sommerville (2011, p. 84), que modela o contexto do projeto de um Sistema de Gerenciamento da Saúde Mental de Pacientes (MHC-PMS).

O sistema destina-se a gerenciar informações sobre os pacientes que procuram clínicas de saúde mental e os tratamentos prescritos. Ao desenvolver a especificação para esse sistema, é preciso decidir se ele deve se concentrar exclusivamente em coletar informações sobre as consultas (usando outros sistemas para coletar informações pessoais sobre os pacientes) ou se deve também coletar informações pessoais dos pacientes.

A Figura 3 mostra o contexto do sistema de forma simples que mostra como o MHC-PMS irá se relacionar com os demais sistemas (ambiente externo ao MHC-PMS).

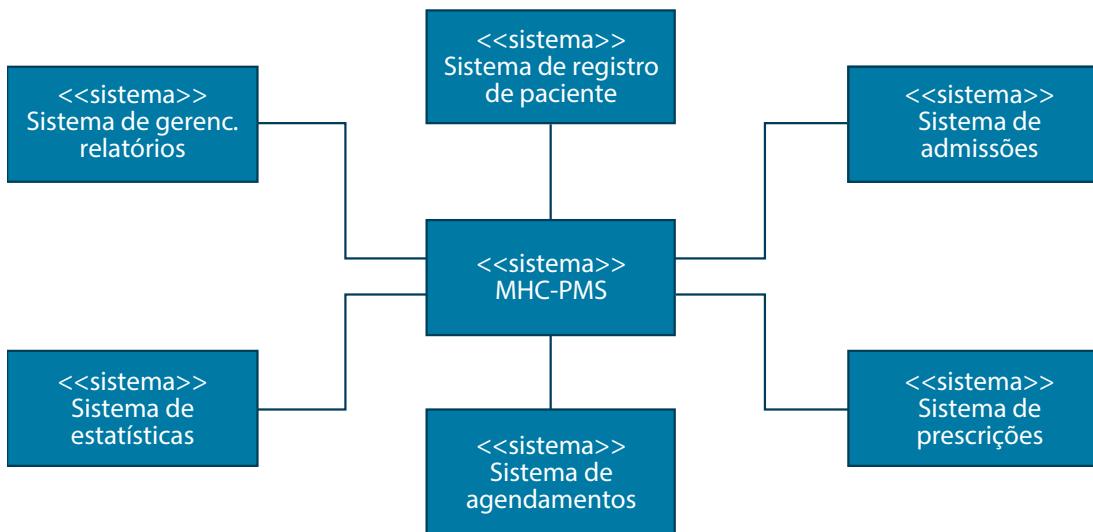


Figura 3: Contexto de um sistema MHC-PMS

Fonte: adaptada de Sommerville (2011, p. 84).



Os modelos de desenvolvimento de software orientado ao planejamento, ágil e livre, têm características em comum e diferenças significativas, mas nenhum deles é efetivo para todos os projetos. Apesar da maioria das propostas existentes para a conciliação de processos de desenvolvimento de software envolver uma combinação rígida das práticas dos diferentes modelos, esse trabalho de pesquisa utiliza a gestão de contexto para obter o equilíbrio entre a colaboração e a disciplina dos processos de desenvolvimento. Para esse balanceamento, é necessário entender e caracterizar o contexto que envolve as pessoas e o ambiente em que o desenvolvimento de software ocorre.

Saiba mais sobre a influência do contexto acessando a pesquisa no endereço disponível em: <http://reuse.cos.ufrj.br/files/publicacoes/relatorioTecnico/RT_VanessaAndrea_ModelagemContexto.pdf>. Acesso em: 20 out. 2015.

Fonte: Nunes, Magdaleno e Werner (2010, online).

Observe na Figura 3 que o MHC-PMS se conecta com os outros sistemas: sistema de agendamento, sistema de registro de pacientes, sistema de admissões, sistema para gerenciamento de relatórios, sistema de estatística e sistema de prescrições. Esses sistemas definem os limites do MHC-PMS e interagem com ele, inserindo e recolhendo informações, portanto o MHC-PMS sofre e promove influência desses outros sistemas. Nesse contexto, os sistemas são o ambiente externo do MHC-PMS, pois todos os fluxos de dados destinados a ele virão dos sistemas. Nas próximas unidades, esse processo será mais detalhado.

Agora vamos utilizar, como exemplo, a especificação de um software comercial. Considere um sistema de controle de inadimplência simplificado, que deverá executar as rotinas de: registrar pagamentos de clientes; emitir comprovantes de pagamento para os clientes que efetuarem o pagamento; emitir extrato para os clientes quando solicitado; emitir relatório de inadimplentes quando solicitado. A Figura 4 representa o modelo do contexto do software de controle de inadimplência.

Observe o seguinte:

- O cliente é fornecedor e receptor de informações do sistema.
- O sistema gera informações para algum responsável pelo relatório de inadimplência.
- O sistema recebe informações do cliente quando efetivado um pagamento.
- O sistema fornece recibo e extrato para o cliente

O Cliente e o Responsável pelo relatório de inadimplência são entidades externas, fontes e receptores das informações geradas pelo e para o sistema. O pagamento do Cliente é um fluxo de dados que o sistema interpreta – podemos considerar como um fluxo de entrada do sistema. O Recibo, o Extrato e o Relatório de Inadimplência são fluxos de dados que o sistema emite – fluxo de saída do sistema.



Figura 4: Diagrama de Contexto do Sistema de Controle de Inadimplência

Fonte: a autora.

Com base na Figura 4, vamos analisar o Diagrama de Contexto em relação ao que foi registrado como ambiente externo. Consideramos como ambiente externo o Cliente e o Responsável pela cobrança, pois não fazem parte do sistema, mas interagem com ele, isto é, geram impacto e provocam respostas do sistema.

Nesse exemplo, é possível incrementar no sistema rotinas que executem a cobrança, incorporando essa responsabilidade no escopo do sistema. Se assim for decidido, a entidade responsável pela cobrança deixaria de ser ambiente externo e não comporia o diagrama de contexto. Essas são as decisões que devem ser tomadas no momento da concepção do sistema e durante a engenharia de requisitos, onde o modelo auxilia na compreensão dos elementos do sistema e facilita a comunicação entre o engenheiro e o cliente.

Para a elaboração do Diagrama de contexto, considere sempre que os fluxos são dados em movimento e podem representar dados de entrada (fluxos de entrada), mas também podem representar dados de saída (fluxos de saída). As entidades externas são as geradoras desses fluxos, elas afetam ou são afetadas pelo sistema, isto é, elas podem fornecer ou receber dados do sistema.

Vamos agora construir uma Lista de Eventos. A Lista de Eventos é composta por três elementos: os Eventos, os Estímulos e as Respostas. Os eventos são ações que estimulam uma reação do sistema. Eles podem ser externos e temporais. Os externos, como a própria denominação insinua, ocorrem fora do sistema, por exemplo, quando o cliente paga a conta. Você deve estar se perguntando, mas o registro do pagamento da conta não é feito no sistema, isso não é interno? Sim, mas o processo vem de um elemento externo; o cliente precisa efetivamente

efetuar o pagamento, que então gera a reação do sistema, que é processar essa ação. As características de um evento externo são: ocorre fora do sistema, provoca uma reação no sistema, gera uma resposta do sistema. Os eventos temporais não estão atrelados a ações externas ao sistema, eles são eventos programados e relacionados ao tempo, por exemplo, um sistema de controle de tráfego que precisa ter acesso a um satélite de hora em hora para acessar fotos e atualizar suas informações.

Os estímulos são os fluxos de dados encarregados de comunicar ao sistema que um evento externo ocorreu. Esse fluxo carrega todos os dados da ação executada pelo evento externo e provoca a reação de resposta no sistema. Um estímulo está sempre ligado a um evento externo. Um evento temporal nunca está ligado a um estímulo.

As respostas são os fluxos de dados encarregados de carregar as informações de saída do sistema, o resultado do processamento. São geradas respostas tanto na ocorrência de estímulos externos quanto na dos temporais.

Ao compor a Lista de Eventos, nomeie os eventos utilizando uma frase simples que indique a ação que será executada fora do sistema, por exemplo, Cliente realiza pagamento. Para relacionar os estímulos, considere a ação executada em si. Para compor as respostas na lista, verifique qual foi a resposta do sistema para aquele estímulo. A Tabela 2 ilustra as ideias apresentadas.

ESTÍMULO	RESPOSTA	EVENTO
Cliente executa pagamento	Pagamento	Recibo de pagamento
Cliente emite extrato	Solicitação	Extrato de pagamento
Responsável emite relatório de inadimplentes	Solicitação	Relatório de inadimplentes

Tabela 2: Lista de Eventos do Sistema de Controle de Inadimplência
Fonte: a autora.

Claro que essa é uma versão muito simplificada de um sistema de cobrança, muitos outros eventos externos podem e devem acontecer. Esse foi um exemplo para auxiliar a compreensão do conteúdo.

A lista de eventos pode ser substituída ou aprimorada para uma especificação de requisitos e pode evoluir conforme evoluí a interpretação das funções e limites do sistema.

Podemos considerar os modelos de contexto como resultado de um primeiro contato entre o cliente e o engenheiro de software e a equipe de desenvolvimento; depois de refinados e levantados todos os eventos, seus estímulos e suas respostas, é possível ter uma boa noção do que o cliente deseja do sistema e, então, é possível definir o documento preliminar de requisitos apoiado no Diagrama de Contexto e seguir para a próxima etapa, que é definir como será a interação entre o sistema e os usuários.



REFLITA

"Acho que vírus de computador deve contar como vida. Creio que dizem algo sobre a natureza humana que a única forma de vida que criamos até agora é puramente destrutiva. Nós criamos vida à nossa própria imagem."

Fonte: Stephen Hawking.

MODELOS DE INTERAÇÃO

As interações são os aspectos dinâmicos do sistema, isto é, são os fluxos de dados – também denominados na literatura por mensagens – trocados entre os elementos que compõem o sistema com o objetivo de realizar alguma ação. No exemplo do sistema de controle de inadimplência que usamos no tópico anterior, o fluxo de dados entre o cliente e o sistema (efetua pagamento) é uma interação entre o usuário e o sistema; no exemplo do MHC-PMS, as interações representadas lá ocorrem entre sistemas. Vamos entender melhor isso?

Sommerville (2011, p. 86) diz que todos os sistemas envolvem algum tipo de interação, que podem ser do usuário, do tipo entrada e saída (fluxos de entradas e fluxos de saída), entre o software que está sendo desenvolvido e outros softwares, ou ainda interações entre os próprios componentes do software.



SAIBA MAIS

A interação entre máquinas e humanos faz parte da história da humanidade. Essa necessidade impulsiona o desenvolvimento de interfaces sempre mais elaboradas e sempre mais simples de utilização.

Interface, até pouco tempo, era referência de softwares que tinham como objetivo a função de interpretar comandos humanos e reproduzi-los digitalmente e vice-versa. Atualmente, essa abordagem agrega aspectos como processamento perceptual, motor, viso-motor e cognitivo do usuário. A IHC é uma subárea da Engenharia de Software (ES) que propõe modelos de processos, métodos, técnicas e ferramentas para o desenvolvimento de sistemas interativos.

Leia mais sobre o artigo Interação Humano-Computador no link disponível em: <<http://www.devmedia.com.br/artigo-engenharia-de-software-16-interacao-humano-computador/14192#ixzz3mBrKY3H0>>. Acesso em: 20 out. 2015.

Fonte: Moreira (online).

Os diagramas de interação representam a ação interna do software para que o usuário alcance a resposta esperada. A modelagem de um sistema geralmente demanda vários diagramas de interação, o conjunto de todos esses diagramas, segundo Bezerra (2014), constitui o seu modelo de interações.

Os principais objetivos de um diagrama de interação são: (1) levantar informações adicionais para complementar os modelos estruturais e comportamentais; (2) prover aos programadores uma visão detalhada dos objetos e das mensagens envolvidas na realização de uma determinada função. Nesse sentido, a autoridade máxima da interação é a **mensagem**, uma vez que as funcionalidades somente serão realizadas após a interação entre os elementos envolvidos pela troca de mensagens.

Os Casos de Uso e outros diagramas, como os diagramas de sequência e o diagrama de comunicação, são utilizados para modelar as interações de um sistema em projeto. Os diagramas de caso de uso e o diagrama de sequência representam as interações em níveis diferentes de detalhamento e, usados juntos, se complementam mutuamente. O diagrama de comunicação também representa as interações, mas é uma alternativa ao diagrama de sequência, tanto que algumas ferramentas Case de Modelagem de Dados (estudaremos as ferramentas de modelagem na unidade IV) geram um diagrama de comunicação a partir de um diagrama de sequência.

O Guia para Modelagem de Interações Metodologia CELEPAR (2009) reforça essa informação explicando que a modelagem de cada interação pode ser feita de duas formas: dando-se ênfase à ordem temporal (diagrama de sequência) ou dando-se ênfase à sequência das mensagens, conforme elas ocorrem no contexto da estrutura de composição dos objetos (diagrama de comunicação). Portanto, como ambos, o diagrama de sequência e o diagrama de comunicação, são derivados das mesmas informações, são semanticamente equivalentes e podem ser convertidos um no outro sem prejuízo de informações.

É importante ressaltar que eles oferecem abstrações distintas do cenário que estão representando, isto é, são pontos de vista diferentes e a escolha entre um e outro vai depender do que se pretende visualizar do sistema em um determinado momento do projeto ou da análise. Não trataremos, em nossos estudos, de exemplos utilizando o diagrama de comunicação, na unidade III, quando tratarmos da UML, ele será apresentado e conceituado.

CASO DE USO

O caso de uso é uma técnica de descoberta de requisitos, como já citado, criada por Jacobson em 1987; em sua forma mais simples, identifica os atores envolvidos em uma interação e nomeia essa interação. O conjunto de casos de uso é registrado em um Diagrama de Casos de Uso que representa todas as possíveis interações que serão consideradas pelo documento de requisitos do sistema. Esse diagrama tem como objetivo descrever um modelo funcional de alto nível do software em análise.

Segundo Deboni (2003, p. 80), os casos de uso são utilizados em todas as fases do desenvolvimento de um sistema: na fase inicial, no levantamento dos requisitos, durante a fase de *design*, quando são usados para auxiliar na criação de novas visões, além da funcionalidade do sistema; durante a codificação, quando os casos podem e devem ser explorados para promover a validação de cada nova funcionalidade implementada, verificando se está de acordo com o especificado pelo usuário.

Portanto, um caso de uso descreve um cenário de uso específico, utilizando uma linguagem promovida direto do ponto de vista de um ator definido. Mas como traduzir um requisito em casos de uso? Para iniciar o desenvolvimento de um conjunto de casos de uso, que mostrem valor como ferramenta de modelagem, as funções necessárias para o sistema devem estar listadas (PRESSMAN, 2010, p. 153). Vamos retomar o exemplo da modelagem do sistema MHC-PMS proposto por Sommerville (2011, p. 75). A descrição a seguir é representada pelo Caso de Uso Agendar Consulta representado pela Figura 5.

Agendar a consulta permite que dois ou mais médicos de consultórios diferentes possam ler o mesmo registro ao mesmo tempo. Um médico deve escolher, em um menu de lista de médicos on-line, as pessoas envolvidas. O prontuário do paciente é então exibido em suas telas, mas apenas o primeiro médico pode editar o registro. Além disso, uma janela de mensagem de texto é criada para ajudar a coordenar as ações. Supõe-se que uma conferência telefônica para comunicação por voz será estabelecida separadamente.

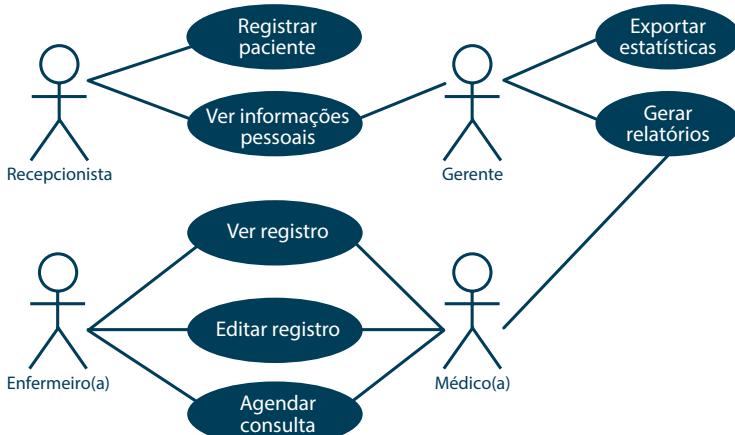


Figura 5: Casos de Uso Agendar Consulta.

Fonte: Sommerville (2011, p. 75).

Observe que temos quatro atores e sete casos de uso nessa visão e conseguimos, de maneira fácil, interpretar as mensagens que estão efetuando com o sistema. A Figura 6, Caso de Uso Transferência de dados, representa o exemplo de um caso específico de troca de mensagem, considerando especificação feita para o exemplo apresentado pelo contexto demonstrado pela Figura 3 – Diagrama de Contexto MHC-PMS.

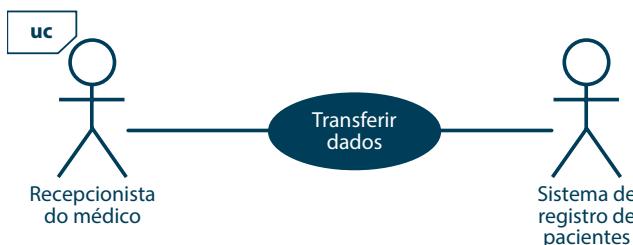


Figura 6: Use Case Transferência de Dados

Fonte: Sommerville (2011, p. 86).

Nesse caso de uso, um dos atores é outro sistema. Os diagramas de caso de uso representam a interação por meio de linhas sem setas, pois na UML as setas representam a direção do fluxo de dados (direção da troca de mensagens). Um diagrama de caso de uso abstrai esse detalhe.

Ficou claro que um caso de uso garante uma visão bastante simplificada da interação, portanto é necessária alguma complementação, conforme a necessidade de detalhes. Essa complementação pode ser uma simples descrição textual, mediante uma tabela, ou outro diagrama, como o de sequência. A decisão deve ser adequada à realidade e necessidade do projeto.

Para encerrar nossa discussão sobre o caso de uso, a Figura 7, Caso de Uso Efetua Pagamento, mostra o caso de uso para a modelagem do sistema de controle de inadimplência, mostrado pelo diagrama de contexto representado pela Figura 4.

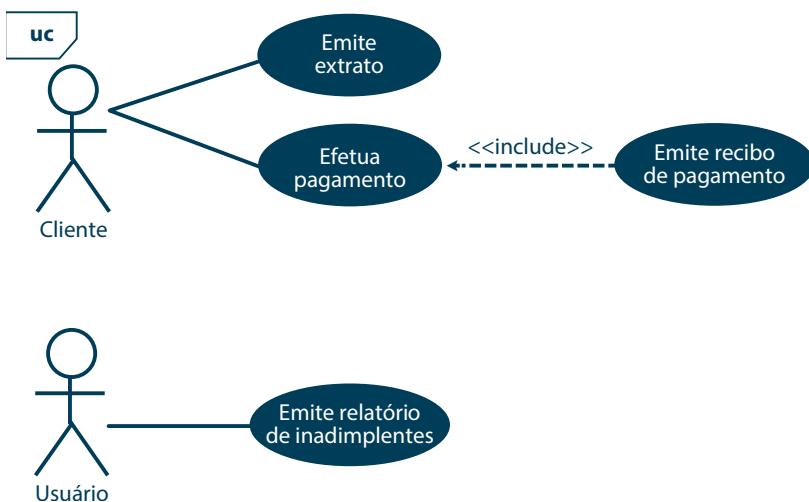


Figura 7: Caso de Uso Efetuar Pagamento

Fonte: Sommerville (2011, p. 86).

Observe que esse caso de uso apresenta uma notação diferente, o <<include>>. O include representa um caso de uso que será executado sempre que um outro for, isso quer dizer que o caso de uso Emite recibo de pagamento é essencial para o comportamento do caso Efetua pagamento. A UML permite dizer que Emite recibo de pagamento_é_parte_de_Efetua_pagamento. Observe também que lá no contexto a emissão de relatório de inadimplentes estava relacionada a um “responsável pela cobrança”, aqui foi substituído por um usuário, porque estamos considerando que essa pessoa responsável pela cobrança possa ser um ou vários usuários do sistema que forem designados para isso, então generalizamos, da mesma maneira que generalizamos cliente.

DIAGRAMA DE SEQUÊNCIA

Os diagramas de sequência na UML® são utilizados para representar as interações – troca de mensagens – entre os atores (atores que representam os elementos externos, ou os que representam outros sistemas) e os objetos, e também entre os próprios objetos. Quando falo objetos, estou me referindo aos elementos que compõem o sistema internamente.

Um diagrama de sequência é montado considerando duas dimensões: uma horizontal, que representa os objetos, e outra vertical, que representa o tempo. Como o nome indica, descreve a sequência e as interações que representam o comportamento dos objetos do sistema, que se relacionam pela troca de mensagens, sequencializando-as no tempo. Cada diagrama representa uma visão (um cenário) das mensagens organizadas por objetivos, ou seja, por funcionalidade do sistema.

Os objetos e atores considerados são listados na parte superior do diagrama, uma linha tracejada na vertical a partir deles representa o tempo. O fluxo de dados – a troca de mensagens – é representado por uma seta, que indica o sentido. Sommerville (2011, p. 87) complementa a descrição acrescentando que o retângulo na linha tracejada indica a linha da vida do objeto. A leitura deve ser feita de cima para baixo. As anotações sobre as setas indicam as chamadas para os objetos, seus parâmetros e os valores de retorno. A Figura 8, Diagrama de Sequência Transferir Dados, é um exemplo de Diagrama de Sequência que representa a troca de mensagens que ocorre para o Caso de Uso Transferir Dados, mostrado na Figura 6.

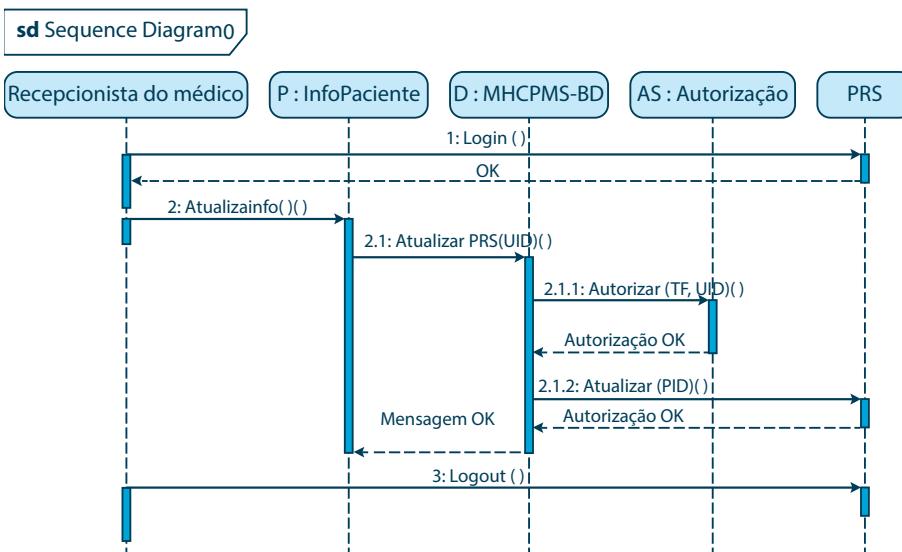


Figura 8: Diagrama de Sequência Transferir Dados
Fonte: adaptada de Sommerville (2011, p. 89).

A leitura desse diagrama pode ser feita da seguinte forma:

1. O Ator Recepção do Médico inicia uma sessão fazendo o login no sistema PRS.
2. A permissão desse Ator é verificada.
3. As informações pessoais do paciente são transferidas para o sistema banco de dados do sistema PRS.
4. Após a conclusão da transferência, uma mensagem de status é enviada pelo PRS para o Ator e a sessão de login é encerrada.

A Figura 9, Diagrama de Sequência Efetuar Pagamento, representa o diagrama de sequência para o Caso de Uso Emitir Pagamento, representado pela Figura 7. Observe que esse diagrama de sequência representa duas possibilidades de interação com o sistema pelo cliente, ele pode efetuar o pagamento, como também pode solicitar a emissão do extrato do cliente. Para atender à solicitação de emissão de extrato, perceba que os objetos trocam mensagens entre si internamente, para então enviar uma resposta ao cliente.

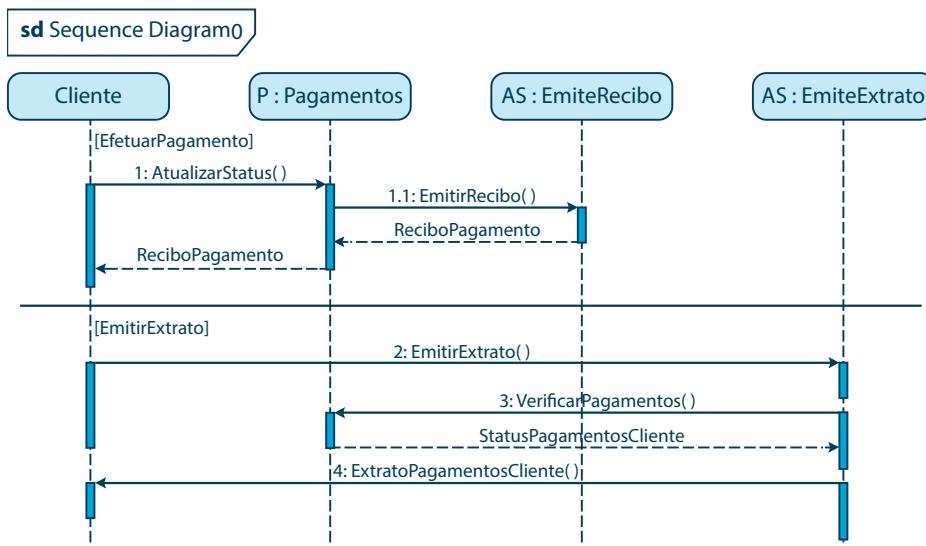


Figura 9: Diagrama de Sequência Efetuar Pagamento
Fonte: a autora.

A leitura desse diagrama pode ser feita da seguinte forma:

Na interação Efetuar pagamento:

1. O Ator Cliente efetua o pagamento e provoca uma alteração de status, antes devedor, para pago.
2. A alteração do status do cliente dispara uma solicitação para a emissão do recibo de pagamento.
3. O recibo de pagamento do cliente é emitido.

Na interação Emite extrato:

1. O Ator Cliente solicita a emissão de extrato.
2. O objeto EmiteExtrato busca as informações de pagamento do cliente e recebe a resposta do objeto Pagamentos.
3. O Extrato é emitido para o cliente.

O nível de detalhamento utilizado no diagrama de sequência depende de como ele será utilizado, se ele será utilizado como base de codificação ou como documentação; todas as interações e todos os parâmetros e mensagens devem ser considerados. Se for utilizado para apoio na engenharia de requisitos, o nível de abstração pode ser mais alto.

MODELOS ESTRUTURAIS

Passando para a perspectiva estrutural, os modelos estruturais representam a organização, a disposição e ordem dos elementos essenciais que compõem o sistema. Os modelos estruturais podem ser estáticos, que mostram a estrutura do projeto do sistema, ou dinâmicos, que mostram a organização do sistema quando está em execução (Sommerville 2011 pag. 89). Nós estudaremos nesta disciplina a modelagem da estrutura estática dos objetos em um software.



SAIBA MAIS

A engenharia dirigida a modelos (MDE, do inglês model-based-engineering) é uma abordagem do desenvolvimento de software segundo a qual os modelos, em vez de programas, são as saídas principais do processo de desenvolvimento. Os programas executados em um hardware/software são, então, gerados automaticamente a partir dos modelos. Os defensores da MDE argumentam que esta aumenta o nível de abstração na engenharia de software, e, dessa forma, os engenheiros não precisam mais se preocupar com detalhes da linguagem de programação ou com as especificidades das plataformas de execução. A engenharia dirigida a modelos tem suas raízes na arquitetura dirigida a modelos (MDA, do inglês model-driven-architecture), proposta pelo Object Management Group (OMG), em 2001, como novo paradigma de desenvolvimento de software. A engenharia e a arquitetura dirigidas a modelos são, frequentemente, vistas como a mesma coisa. No entanto, penso que a MDE tem um alcance maior que a MDA.

Saiba mais sobre a engenharia dirigida a modelos lendo os capítulos 6, 18 e 19 de Engenharia de Software, de Ian Sommerville.

Fonte: Sommerville (2011, p. 96).

DIAGRAMA DE CLASSES

A partir deste ponto, nossos objetos passam a se chamar “classes”. Em uma definição bastante generalizada, as classes representam um conjunto de objetos com as mesmas características, são matrizes de objetos, identificam grupos de elementos do sistema que compartilham as mesmas propriedades. O diagrama de classes é a representação fundamental da modelagem orientada a objeto e evolui de uma visão conceitual para uma visão detalhada durante a evolução do projeto (DEBONI, 2003).

No primeiro estágio, chamado de conceitual, o foco está em identificar os objetos e classificá-los em classes específicas, considerando suas características semelhantes. Simula o domínio em estudo, em uma perspectiva destinada ao cliente. Nesse estágio de modelagem, o Diagrama de Classes é chamado de Conceitual.

A Figura 10 traz o Diagrama de Classes Conceitual do MHC-PMS apresentado por Sommerville (2011). Observe que a figura representa as classes e as associações entre elas são representadas por uma linha. A associação apresenta o vínculo que incide, geralmente, em duas classes (binária), mas pode acontecer de uma classe se associar com ela mesma (unária) ou de se associar com mais de uma classe (n-ária).

O diagrama especifica também a multiplicidade que determina qual das classes envolvidas em uma associação fornece informações para as outras. Estudaremos mais sobre a multiplicidade na unidade III, quando analisaremos mais detalhadamente os diagramas da UML®. Outro exemplo do Diagrama de Classes no nível conceitual é representado pela Figura 11, Diagrama de Classes Conceitual Caixa Eletrônico.

A evolução do nível conceitual do diagrama de classes, chamado de especificação, tem foco nas principais interfaces da arquitetura e nos principais métodos, mas ainda não tem preocupação em como eles serão implementados, em uma perspectiva destinada àqueles envolvidos que não precisam dos detalhes da codificação, como o gerente de projeto ou o patrocinador. Para exemplificar a evolução, a Figura 12, Diagrama de Classes Especificação Caixa Eletrônico, mostra como fica o diagrama acrescido das principais interfaces, identificados por <<interfaces>>, e os principais métodos, relacionados na última partição da classe.

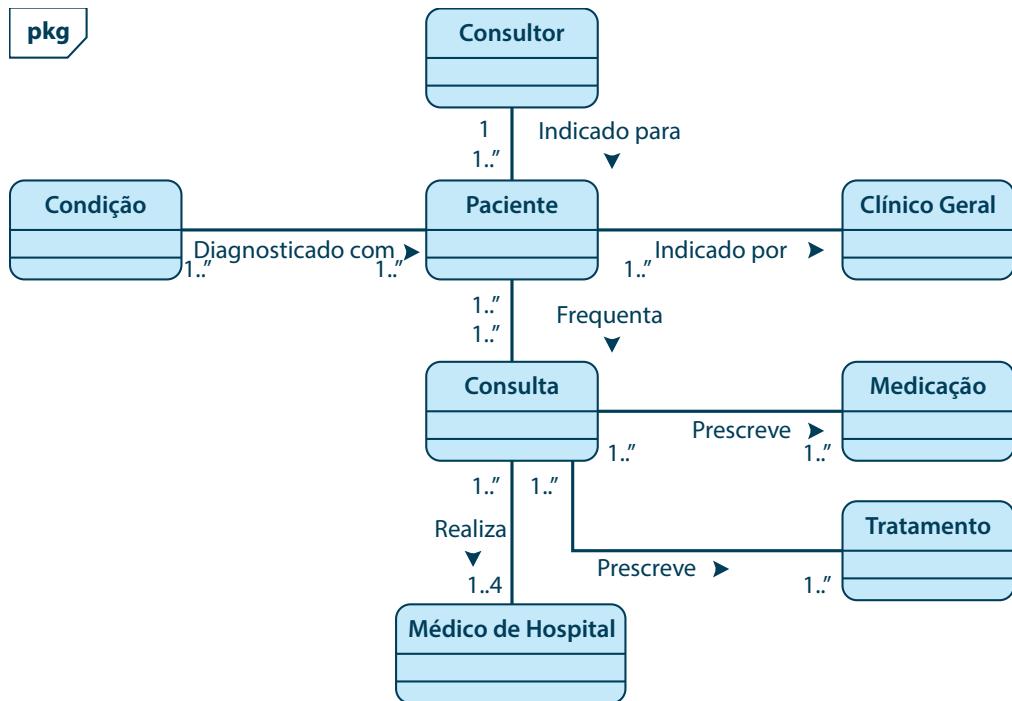


Figura 10: Diagrama de Classes Conceitual do MHC-PMS

Fonte: adaptada de Sommerville (2011, p. 91).

O último nível de detalhamento do diagrama de classes considera os requisitos em nível de implementação, como naveabilidade, tipo dos atributos, métodos etc. É uma perspectiva designada à equipe de programação. A Figura 13, Diagrama de Classes Implementação Caixa Eletrônico, mostra todos os detalhes necessários para a equipe de programação ser capaz de gerar o código do sistema a partir dele.

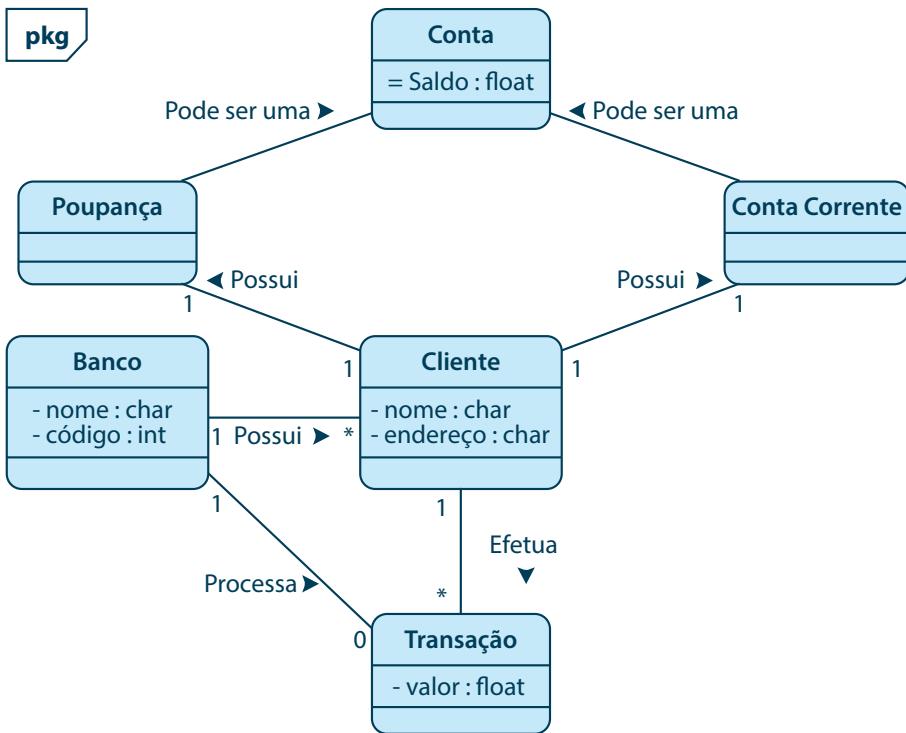


Figura 11: Diagrama de Classes Conceitual Caixa Eletrônico

Fonte: adaptada de UFCG (online).

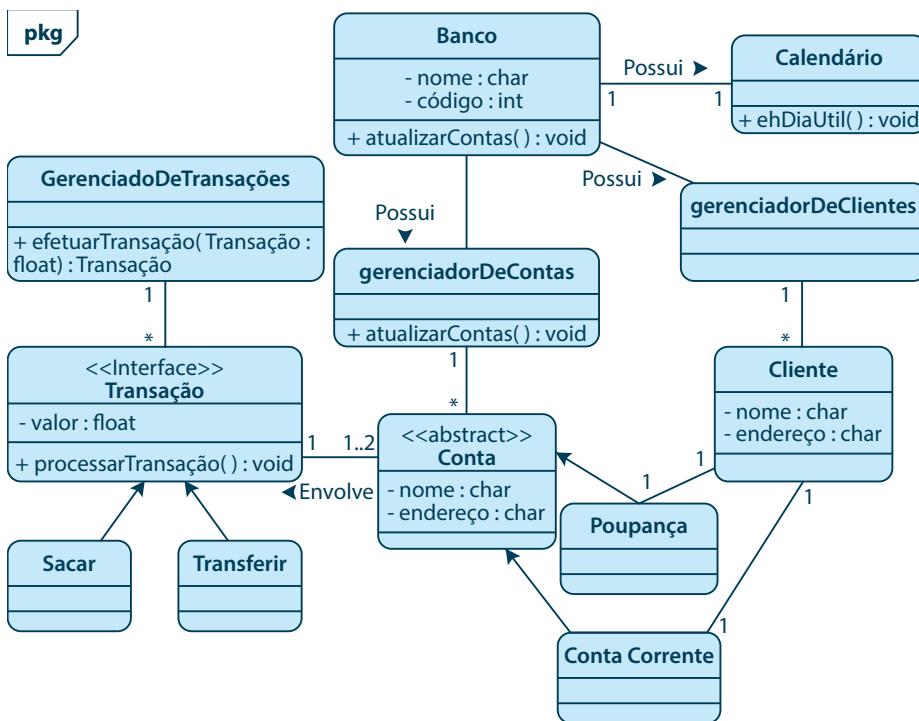


Figura 12: Diagrama de Classes Especificação Caixa Eletrônico
Fonte: adaptada de UFCG (online).

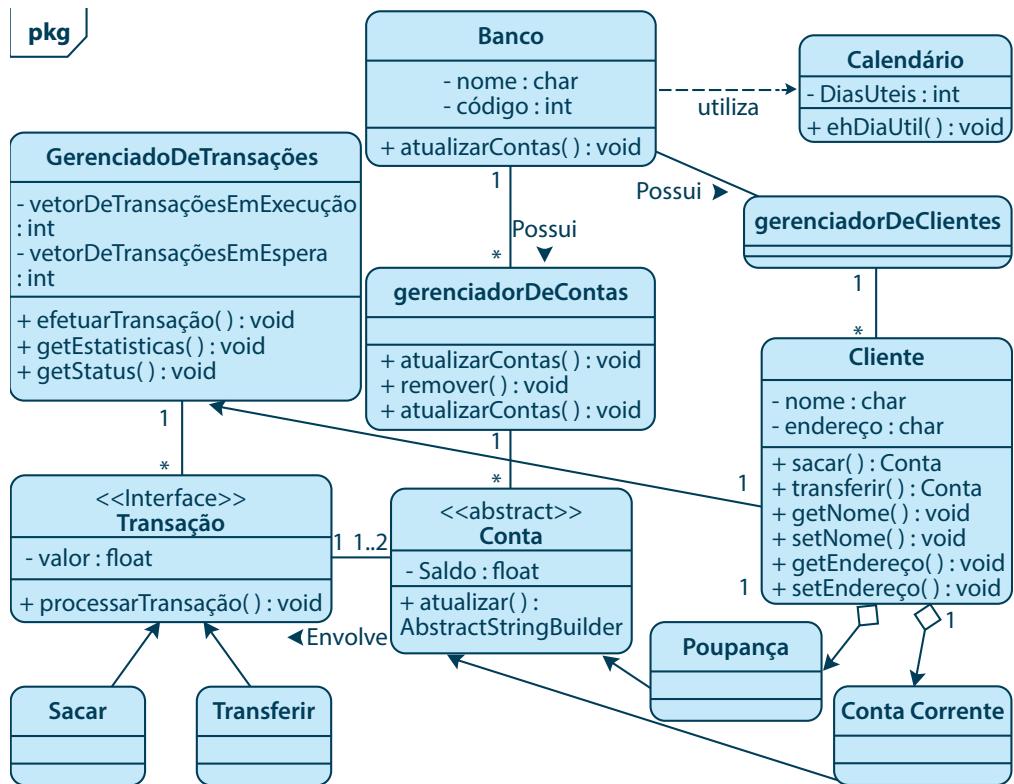


Figura 13: Diagrama de Classes Implementação Caixa Eletrônico

Fonte: adaptada de UFCG (online).

Reforço novamente que os Diagramas apresentados possuem um nível de complexidade baixo, desconsiderando várias funções possíveis para os sistemas sugeridos, tendo um objetivo didático e não comercial. Para o desenvolvimento de sistemas comerciais, todas as funcionalidades devem ser consideradas.

MODELOS COMPORTAMENTAIS

Os modelos estruturais discutidos no tópico anterior representam elementos estáticos, chegou a hora de progredir para o comportamento dinâmico do sistema. Os modelos comportamentais representam o comportamento dinâmico do sistema, o que acontece ou o que deve acontecer quando o sistema responde a um estímulo. Representa a perspectiva – visão – do comportamento dos dados mediante a ação das funções. A modelagem comportamental irá retratar os estados e os eventos que transformam esses estados.

A maioria dos sistemas computacionais é orientada a dados, isto é, são controlados pela entrada de dados com carga relativamente baixa de processamento de eventos. Sommerville (2011, p. 93) nos orienta a considerar dois tipos de estímulos:

- a. Dados: aqueles que chegam e precisam ser processados (transformados).
- b. Eventos: aqueles que geram o processamento do sistema, mas nem sempre estão acompanhados de dados.

A orientação por dados é fácil, estudamos isso desde o início da nossa vida acadêmica, uma função do sistema é ativada quando recebe um dado para ser processado, que então é transformado em novos dados de saída, isto é, são sistemas direcionados por um fluxo de controle padronizado. Agora, e o processamento de eventos? Ao estudarmos os Modelos de Contexto, vimos que eventos estavam relacionados com a interação entre o ambiente externo, ou de outro sistema, e o sistema. Essa definição vale aqui também, mas um evento, em sistemas orientados a eventos, não precisa ser necessariamente de usuário ou de outro sistema, pode ser de hardware, de *socket*, de *timers*, ou ainda de qualquer outro objeto da programação. A programação orientada a eventos é um dos paradigmas de programação, isto é, é uma maneira básica de se programar, na qual a execução do programa é direcionada por eventos, normalmente sensível a sensores. É bastante aplicado no desenvolvimento de software de interface com o usuário. Para exemplificar uma aplicação, apresento a seguir um trecho do artigo de Nagao (2009, online):

Esta forma de se programar é, na minha humilde opinião, a base de todos os sistemas de UI (User Interface) sofisticados. Por exemplo, aquela barrinha do MAC OSX que todos adoram e suas similares: [...] ela sim-

plesmente possui um sensor que avisa “Olha, programa, detectei que o mouse entrou na posição (x,y) da barra”, ao receber esta mensagem o programa faz o zoom e mostra o label dos ícones na posição. Este parece um exemplo bobo, mas acho que ilustra bem a diferença entre a programação em lote e a programação orientada a eventos.

Nesse sentido, a modelagem deve respeitar o comportamento interno do programa, no caso, se orientado a dados ou se orientado a eventos.

MODELAGEM ORIENTADA A DADOS

Para falar de modelagem orientada a objetos, temos que dar um pulinho no passado. A modelagem de dados foi uma das propostas pioneiras como solução para a crise de software que aconteceu em meados da década de setenta. A análise estruturada de sistemas passa a ser, então, um conjunto de técnicas e de ferramentas, tendo como objetivo apoiar a análise e a definição dos sistemas computacionais. Sua base conceitual é a construção de um modelo do sistema utilizando técnicas gráficas: Diagrama de Fluxo de Dados (DFD) e o Dicionário de Dados (DD). A UML® não oferece suporte ao DFD, porque sua proposta é modelar o processamento de dados, e, portanto, foca somente as funções do sistema e não reconhece os objetos do sistema.

Para contornar essa situação, uma vez que os sistemas orientados a dados são a maioria no mundo dos negócios, a UML® desenvolveu o Diagrama de Atividades – muito parecido com o DFD – em que é possível modelar as etapas do processamento (atividades) e os dados (objetos) navegando entre as etapas. A Figura 14, Diagrama de Atividade Caixa Eletrônico – Cartão de Crédito, mostra o fluxo de atividades em um único processo, ele registra como uma atividade depende de outra.

Para associar um objeto ao modelo, o diagrama contém espaços chamados *swimlanes*. Dentro de cada *swimlane* estão registradas as atividades relativas ao objeto daquela região. No Exemplo, as *swimlanes* são o cliente, o caixa eletrônico e o banco. Outro detalhe do diagrama é que as atividades são conectadas por setas (transições ou *threads*), essas setas representam as dependências entre elas. O exemplo representa, de forma simplificada, a função de retirar dinheiro de um caixa eletrônico (utilizando cartão de crédito).

act Activity Diagramo

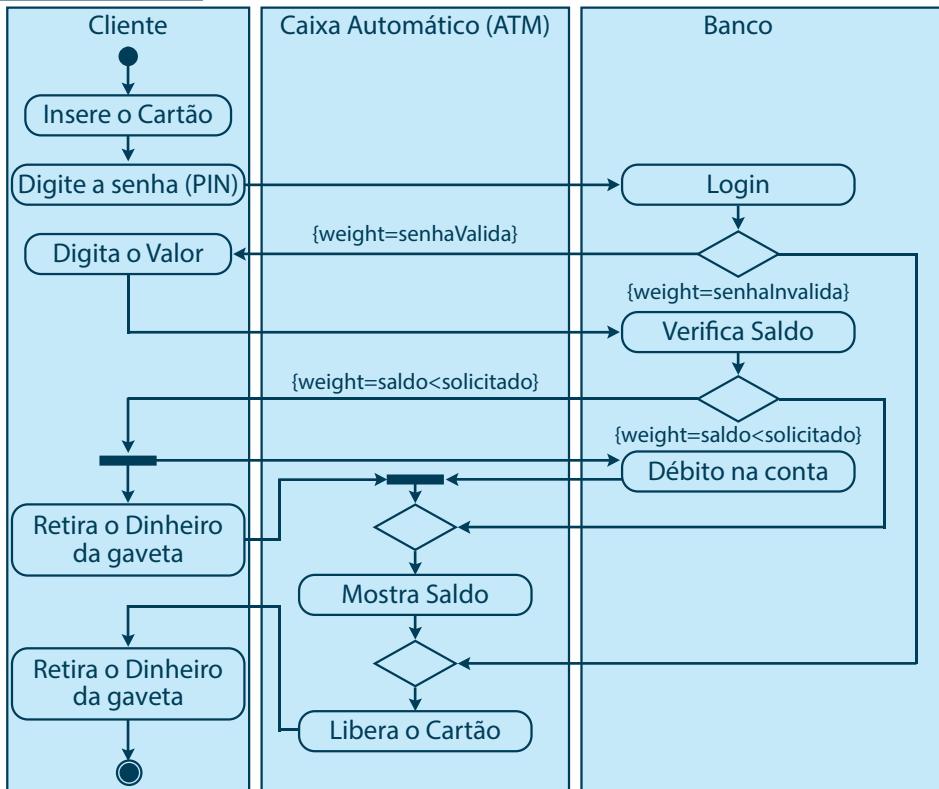


Figura 14: Diagrama de Atividade Caixa Eletrônico – Cartão de Crédito.

Fonte: adaptada de UFCG – Diagrama de Atividades (online).

O diagrama de sequência da UML® também pode ser utilizado para modelar o fluxo de dados de um sistema.

MODELAGEM ORIENTADA A EVENTOS

Se um sistema orientado a dados reage a dados, então um sistema orientado a eventos irá reagir a eventos. Como vimos, eventos podem partir de usuários, de hardware, de sensores, de *timers* etc. O que difere aqui, Sommerville (2011, p. 94) diz, é que a modelagem dirigida a eventos supõe que um software tem um número finito de estados e que os eventos causam uma transição entre esses estados.

O sistema que controla o *airbag* dos nossos carros pode traduzir essa afirmação de uma forma muito simples – e dramática – quando o estado do *airbag* é desarmado (estado 1) e quando o estímulo (batida) é recebido pelo sensor, ele passa para o estado armado. Podemos considerar também a situação de uma válvula de pressão, que está desligada, mas no momento que receber o estímulo, que pode vir do operador ou do próprio software, passa para ligada. São estados finitos.

A UML® usa o Diagrama de Estados para a modelagem de sistemas orientados a eventos (PRESSMAN, 2010, p. 177). Para criar um modelo de comportamento orientado a eventos, o engenheiro precisa executar os seguintes passos:

Avaliar todos os casos de uso para entender plenamente a sequência de interação dentro do sistema.

1. Avaliar todos os casos de uso para entender plenamente a sequência de interação dentro do sistema.
2. Identificar os eventos que dirigem a sequência de interação e entender como esses eventos se relacionam a classes específicas.
3. Criar uma sequência para cada caso de uso.
4. Construir um diagrama de estado para o sistema.
5. Revisar o modelo comportamental para verificar a precisão e a consistência.

A Figura 15, Diagrama de Estado da Conta Bancária, mostra de maneira simplificada os estados que a conta de um usuário bancário pode assumir, complementando os exemplos de modelagem para caixa eletrônico, apresentados até agora.

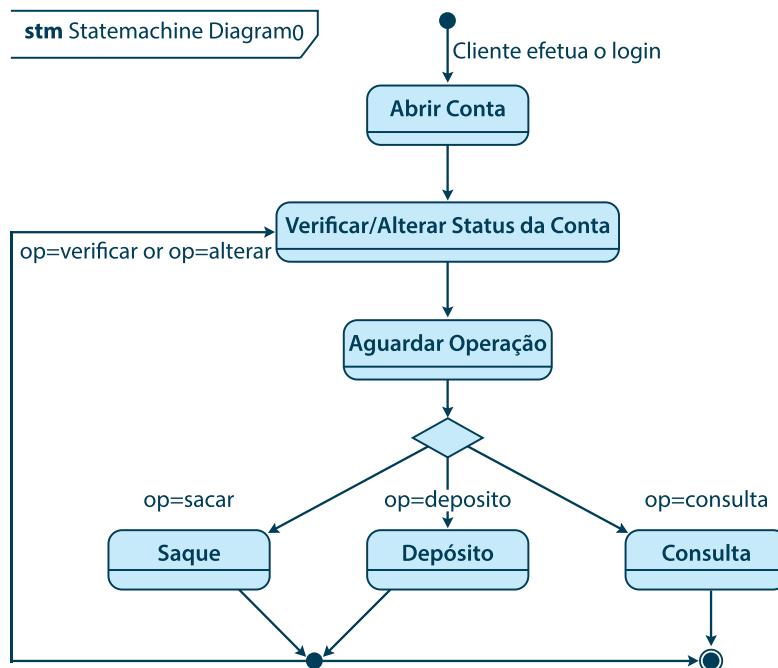


Figura 15: Diagrama de Estado da Conta Bancária
Fonte: adaptada de UFMA – Estudo de caso (online).

Para complementar e empregar outra abordagem de eventos, vamos analisar o exemplo oferecido por Sommerville. Ele mostra o diagrama de estados de um software simplificado para o controle de um forno micro-ondas representado na Figura 16, Diagrama de Estados Forno Micro-ondas.

Forno de micro-ondas reais são muito mais complexos, [...] esse micro-ondas simples tem um interruptor para selecionar a potência total ou meia, um teclado numérico para inserir o tempo de cozimento, um botão iniciar/cancelar e um display alfanumérico. Vamos assumir que a sequência de ações no uso do micro-ondas seja: (1) Selecionar o nível de potência (ou meia potência ou potência total). (2) Introduzir o tempo de cozimento usando um teclado numérico. (3) Pressionar ‘iniciar’, e os alimentos são cozidos no tempo selecionado. [...] o forno não opera com a porta aberta, e uma campainha é acionada no final do cozimento (SOMMERVILLE, 2011, p. 96).

Observe que o sistema parte do estado de espera e responde ao botão de potência total ou meia potência. Após a seleção, é possível mudar de ideia e pressionar outro botão. O tempo é definido, se a porta estiver fechada, o botão iniciar é habilitado. Ao ser pressionado o botão iniciar, a operação começa e o cozimento ocorrerá durante o tempo definido, finalizando assim esse ciclo, e o sistema volta para o estado de espera (SOMMERVILLE, 2011, p. 95).

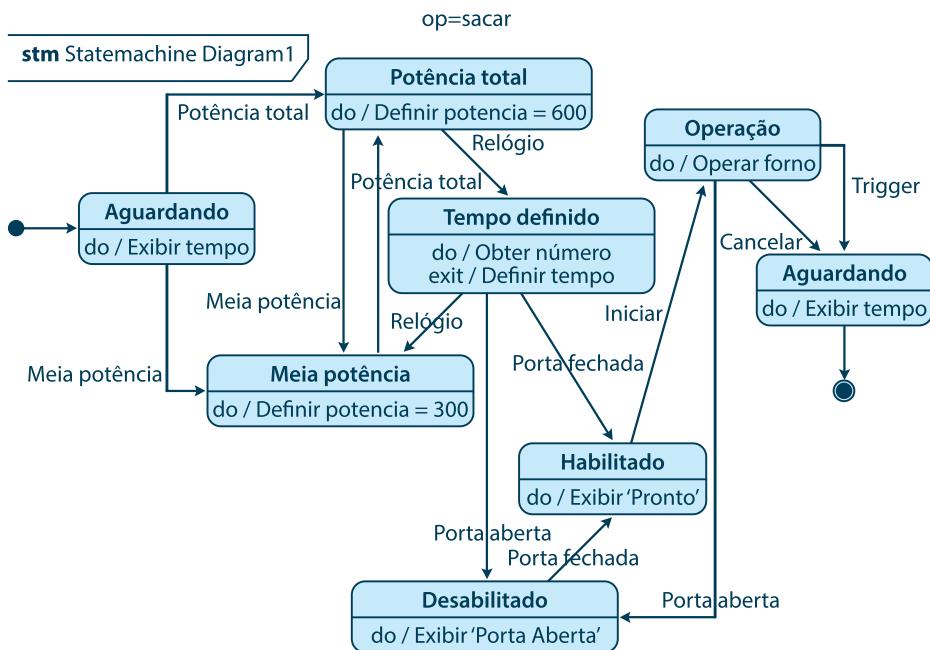


Figura 16: Diagrama de Estados Forno Micro-ondas
Fonte: adaptada de Sommerville (2011, p. 96).

CONSIDERAÇÕES FINAIS

Nesta unidade, analisamos os principais tipos de modelos utilizados nos projetos de desenvolvimento de software e percebemos que diferentes tipos de modelos são necessários, porque existem várias formas de se observar o problema em desenvolvimento. O conjunto de visões tem o objetivo de propiciar formas diferentes de se observar a mesma coisa, sempre considerando a ótica do interessado ou da necessidade naquele momento.

Os modelos de contexto, por exemplo, fornecem visões que interessam aos *stakeholders* e aos engenheiros analistas responsáveis por definir ou identificar as funções essenciais do software, sem se preocupar com as especificidades da estrutura ou implementação. Uma visão mais lógica, do mesmo pedaço do software, seria mais interessante para os programadores, assim podem definir as melhores estruturas para atender às necessidades de forma eficiente, e assim acontece para todas as etapas envolvidas no projeto de desenvolvimento de software.

Estudamos, para isso, as perspectivas fundamentais de modelagem de softwares: perspectiva externa, que mostra como um sistema que está sendo modelado está posicionado, referente a outros sistemas ou processos, auxiliando na definição dos seus limites operacionais; perspectiva de interação, na qual vimos que casos de uso e diagramas de sequência ilustram por simbologias lógicas a interação entre o sistema e os atores externos, no caso de uso, ou entre os objetos do sistema, no diagrama de sequências, também que os modelos estruturais representam a organização e arquitetura do software, e que os diagramas de classe são utilizados para representar a estrutura estática das classes e seus relacionamentos; por fim, a perspectiva comportamental, que traz que os diagramas de estado são utilizados para modelar o comportamento de um software em resposta a eventos promovidos por elementos internos ou externos.

ATIVIDADES



1. Qual é a importância da participação do cliente no processo de desenvolvimento de software?
2. Exemplifique dois softwares que podem ser modelados por meio de várias visões. Faça um breve descritivo dessas visões.
3. Os diagramas de caso de uso da UML® são importantes para a modelagem do contexto de um sistema. Qual seria outra utilidade dos diagramas de caso de uso?
 - a) Testes de sistemas executáveis.
 - b) Definição dos limites do sistema.
 - c) Gerenciamento de versões.
 - d) Modelagem da visão estática da implantação de um sistema.
4. No desenvolvimento de softwares, diversas atividades estão envolvidas, uma delas pode ser definida como: criação de modelos que permitam ao engenheiro, programador e cliente entenderem mais claramente os requisitos do software e o projeto que vai atender a esses requisitos. Essa atividade é conhecida como:
 - a) Modelagem
 - b) Construção
 - c) Comunicação
 - d) Planejamento
 - e) Análise
5. Em relação à modelagem de software, assinale a opção correta.
 - a) Um modelo é uma abstração elaborada para entender um problema antes de implementar uma solução. As abstrações são subconjuntos da realidade, selecionados para determinada finalidade.
 - b) Modelos de contexto são usados para mostrar como os dados ocorrem por uma sequência de etapas de processamento.
 - c) A capacidade de reproduzir fielmente a complexidade do problema é uma das principais motivações para a realização da modelagem.
 - d) Uma forma comum de modelagem de programas procedurais é por meio de fluxogramas de objeto.



METODOLOGIAS DE DESENVOLVIMENTO DE SOFTWARE: UMA ANÁLISE NO DESENVOLVIMENTO DE SISTEMAS NA WEB

O desenvolvimento de software objetiva a criação de sistemas de software que correspondam às necessidades de clientes e usuários (VASCONCELOS, 2006). Dessa forma, se torna fundamental que se realize uma correta especificação dos requisitos do software para se obter o sucesso do processo. Assim, é cada vez mais utilizado dentro das organizações o analista de requisitos, desempenhando um papel de crucial importância. Com isso, surgem as novas Metodologias de Desenvolvimento de Software (MDS), que dividem o processo de desenvolvimento de software, a fim de organizá-lo e facilitar seu entendimento. Assim, segundo Souza Neto (2004) e Soares (2004), divide-se em duas áreas de atuação:

Desenvolvimento “tradicional”, o qual se fundamenta na análise e no projeto, que conserva tudo em documentação, no entanto, não é vantajoso para mudanças.

Desenvolvimento ágil, baseado em código, inteiramente adaptável a mudanças nos requisitos, mas deficiente na esfera contratual e de documentação.

Maia (2007) define a engenharia de software como um processo para a produção organizada que utiliza uma coleção de técnicas predefinidas e convenções de notação. Para Sommerville (2008), a engenharia de software é uma área da engenharia que se ocupa de todos os aspectos produtivos do software, desde os estágios iniciais de

especificação e entendimento do sistema até sua manutenção depois que ele entrou em operação.

Podemos verificar que a engenharia de software se preocupa com a implantação de métodos, ferramentas e técnicas no processo produtivo de sistemas, buscando a eficácia e eficiência dos recursos, melhorando a qualidade do produto final, bem como reduzindo o prazo e custo de produção.

Essa pesquisa demonstra que o processo utilizado não é responsável direto pelo sucesso do projeto, e sim sua adequação ao ambiente onde foi implantado, atendendo à demanda do cliente da melhor maneira possível, seja com a entrega de um produto apenas ao final do cronograma ou com entregas sucessivas ao longo do projeto. Verifica-se que os clientes não têm interesse no processo de produção da aplicação, descartando ou minimizando a importância dos artefatos e valorizando apenas o software. Esse motivo dá-se principalmente pelo fato de que a documentação e o processo de desenvolvimento não agregam valor ao negócio, e sim a aplicação que objetiva resolver seus problemas negociais.

Essa visão tem causado problemas em relação à aderência do produto às especificações e necessidades do cliente, bem como custo com retrabalho, atrasos e insucessos no projeto. Como forma de ampliar os conhecimentos acerca do assunto, sug-



re-se que se realize um levantamento dos processos utilizados para o desenvolvimento de software nas áreas pública e privada, relacionando com o tamanho dos projetos e sua taxa de sucesso.

Essa pesquisa pode levantar um quadro atual do desenvolvimento no mercado brasileiro e sugerir novas soluções para atender às novas necessidades.

Fonte: Oliveira e Seabra (2015, online).

MATERIAL COMPLEMENTAR



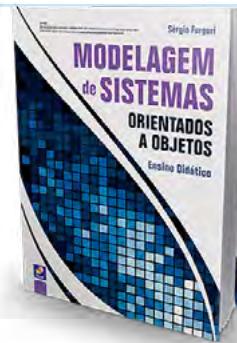
LIVRO

Modelagem de Sistemas: Orientados a Objetos - Ensino Didático

Sergio Furgeri

Editora: Érica

Sinopse: A modelagem de sistemas tem se tornado uma forte aliada na comunicação entre membros de um time de software. Nos últimos anos, os processos de desenvolvimento, a UML e as ferramentas CASE têm ajudado a equipe a ser mais eficiente, participativa e flexível, reduzindo o tempo de produção e, consequentemente, o custo do projeto como um todo. O livro apresenta uma proposta de ensino diferente, com a qual o leitor aprende os elementos mais importantes do processo de modelagem, começando com a captura de requisitos e seu impacto no código de programação, fornecendo exemplos na linguagem Java. Em todo o livro se estabelecem relações entre o mundo conceitual e o prático, abordando os principais diagramas da UML e Java, uma maneira de mapear o mundo gráfico com a programação Java. Analogias também são bastante usadas para facilitar o entendimento. Contempla estudos de caso e exemplos que envolvem os diagramas da UML, o diagrama de entidades e relacionamentos para modelagem do banco de dados, ambos usando o Visual Paradigm for UML 10.0, uma das principais ferramentas CASE do mercado com suporte à modelagem de sistemas. Além disso, dois apêndices demonstram a utilização do Visual Paradigm for UML 10.0 e a IDE NetBeans versão 4 para criação de alguns exemplos em Java presentes no livro. Destina-se aos profissionais da área de informática e estudantes de ensino técnico, tecnológico e universitário. Independente do nível escolar, o aprendizado da UML e da modelagem de sistemas é fundamental. O conteúdo do livro é útil também às empresas que selecionam candidatos para a área e concursos públicos. O estudo da modelagem e da orientação a objetos é aprimorado por diversos exemplos e exercícios, inclusive resolvidos. As respostas dos exercícios, o código-fonte dos exemplos em Java e os diagramas feitos com o Visual Paradigm estão disponíveis em: <www.editoraerica.com.br> para download.





Objetivos de Aprendizagem

- Revisar principais conceitos de orientação a objeto.
- Compreender os objetivos dos principais diagramas da linguagem de modelagem UML®.

Plano de Estudo

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

- Pilares da orientação a objeto
- Linguagem de modelagem unificada UML®
- Diagramas UML®

INTRODUÇÃO

Muito bem! Já entendemos que os modelos são importantes, estudamos também que diferentes aspectos do software são representados por diferentes modelos, a fim de atingir diferentes objetivos. O próximo passo agora é compreendermos a linguagem utilizada nos modelos.

A UML® começou a ser desenhada quando Jim Rumbaugh e Grady Booch resolveram combinar dois métodos populares de modelagem orientada a objeto: Booch e OMT (*Object Modeling Language*). Ivar Jacobson, o criador do método Objectory, uniu-se aos dois para a concepção da primeira versão da linguagem UML (*Unified Modeling Language*). A UML® foi adotada em 1997 pela OMG (*Object Management Group*), que a mantém até hoje. A UML® acrônimo de *Unified Modeling Language*, é uma linguagem para modelagem de dados orientado a objetos, que tem por principal objetivo apoiar e incentivar as boas práticas para os projetos de desenvolvimento de software.

Estudaremos nesta unidade os diagramas propostos pela UML® e suas estruturas, com o objetivo de identificar sua aplicabilidade nas diferentes etapas do processo de desenvolvimento de software. Por ser orientada a objetos, passaremos brevemente por uma revisão dos principais conceitos relacionados a esse paradigma.

Perceberemos, por fim, que, apesar de recomendar diagramas específicos para cada estágio do processo de desenvolvimento de software, organizando-os em duas grandes categorias – Diagramas Estruturais e Diagramas Comportamentais – a especificação UML® não restringe a mistura de diferentes tipos de diagramas, por exemplo, combinar elementos estruturais e comportamentais para mostrar uma máquina de estado aninhado em um caso de uso. Portanto, não existem fronteiras entre a utilização de um ou outro modelo pelo seu objetivo principal possibilitando a utilização de acordo com o necessário para o projeto e as necessidades em questão.

Venha comigo nesta aventura, vamos conhecer os principais diagramas UML®!

PILARES DA ORIENTAÇÃO A OBJETOS

Para entender os conceitos da linguagem UML®, é necessário revisar alguns da orientação a objetos. A linguagem UML é baseada nos princípios da orientação a objetos e trata da representação gráfica parcial de um sistema na sua fase de projeto, implementação ou de sistemas existentes.

Antes de iniciarmos nossa discussão sobre a orientação a objetos e a UML®, convido-o(a) a relembrar dois conceitos importantes necessários para “costurar” a modelagem enquanto análise e a programação. O primeiro deles é o termo paradigma. Provavelmente, você já se deparou com a expressão paradigma de software ou paradigma de linguagem de programação ou ainda paradigmas de programação. Você sabe exatamente o que isso significa? Vamos lá!

O termo paradigma tem origem no grego “*paradeigma*”, que significa modelo ou padrão. Vamos observar o que o dicionário nos apresenta no quadro a seguir:

paradigma

pa.ra.dig.ma

sm (gr *parádeigma*) 1 Modelo, padrão, protótipo. 2 **Ling** Conjunto de unidades suscetíveis de aparecerem num mesmo contexto, sendo, portanto, comutáveis e mutuamente exclusivas. No paradigma, as unidades têm, pelo menos, um traço em comum (a forma, o valor ou ambos) que as relaciona, formando conjuntos abertos ou fechados, segundo a natureza das unidades. No primeiro caso temos os paradigmas lexicais e, no segundo, gramaticais.

Exemplo de paradigma lexical:

A **bela casa/alta/grande/verde**. Exemplo de paradigma gramatical: **and-a/and-as/and-a/and-amos**.

Fonte: Paradigma (online).

A linguagem de programação é uma forma padronizada de repassar comandos para um computador. É um conjunto de regras de sintaxes e de semântica que possibilita a interação entre o homem e a máquina. Cada linguagem de programação é diferente da outra, contendo suas próprias “palavras” de código. Sobrepondo as definições, chegamos à conclusão de que um paradigma de linguagem de programação é um modelo, um conjunto de unidades capaz de estar em um mesmo contexto.

Um paradigma é o que determina o ponto de vista da realidade e como se atua sobre ela, os quais são classificados quanto ao seu conceito de base, podendo ser: Imperativo, funcional, lógico, orientado a objetos e estruturado. Cada qual determina uma forma particular de abordar os problemas e de formular respectivas soluções. Além disso, uma linguagem de programação pode combinar dois ou mais paradigmas para potencializar as análises e soluções. Deste modo, cabe ao programador escolher o paradigma mais adequado para analisar e resolver cada problema (JUNGTHON; GOULART, s/d, p. 1).

Um desses paradigmas é a orientação a objetos. O paradigma de programação orientada a objetos fundamenta-se na utilização de objetos, que colaboram entre si, para a construção do software. A colaboração entre os objetos é feita por meio da troca de mensagens. O paradigma orientado a objeto possibilitou a criação de várias soluções, entre outras, Engholm (2010) cita: programação OO, modelagem OO, banco de dados OO, interfaces OO, metodologias de desenvolvimento de software OO. Por exemplo, esse padrão de programação é seguido – incorporado – por linguagens de programação como *Smalltalk* (a pioneira), *Python*, *Ruby*, *C++*, *Object Pascal*, *Java*, *C#*, *Oberon*, *Ada*, *Eiffel*, .NET e *Simula*.

A orientação a objetos está sustentada nos seguintes pilares: abstração, encapsulamento, herança e polimorfismo. Conceitos que estudaremos a seguir.

SAIBA MAIS



Sobre os paradigmas de desenvolvimento no paradigma estruturado, temos procedimentos (ou funções) que são aplicados globalmente em nossa aplicação. No caso da orientação a objetos, temos métodos que são aplicados aos dados de cada objeto. Essencialmente, os procedimentos e métodos são iguais, sendo diferenciados apenas pelo seu escopo.

A programação estruturada, quando bem feita, possui um desempenho superior ao que vemos na programação orientada a objetos. Isso ocorre pelo fato de ser um paradigma sequencial, em que cada linha de código é executada após a outra, sem muitos desvios, como vemos na POO. Entretanto, a programação orientada a objetos traz outros pontos que acabam sendo mais interessantes no contexto de aplicações modernas. Essa difusão se dá muito pela questão da reutilização de código e pela capacidade de representação do sistema muito mais perto do que veríamos no mundo real. Entenda quais são as vantagens e desvantagens de cada um dos paradigmas de programação. Leia mais no link disponível em: <<http://www.devmedia.com.br/programacao-orientada-a-objetos-versus-programacao-estruturada/32813>>. Acesso em: 21 out. 2015.

ABSTRAÇÃO

Novamente o termo abstração é apresentado. Enquanto para a modelagem abstração significa níveis de detalhamento para cada visão específica, para a orientação a objetos, a abstração representa uma entidade do mundo real. Ela representa as características essenciais de um objeto que o diferenciam de outros (ENGHOLM, 2010). Mas ambas as definições remetem o conceito de abstração ao isolamento de propriedades que se deseja representar fora do seu ambiente complexo. Em orientação a objetos, isolamos um objeto do ambiente real e representamos somente as características interessantes para o problema em questão. A abstração está fortemente relacionada com outro conceito da Orientação a Objeto conhecido como Acoplamento. Quanto mais abstrações utilizadas estratégicamente, mais baixo é o nível de Acoplamento¹.

A abstração de um objeto dentro da orientação a objetos deve possuir três características: Identidade, Propriedades e Métodos.

¹ Acoplamento em orientação a objeto significa o grau de dependência entre dois objetos. Por exemplo, Uma classe com alto nível de acoplamento é uma classe dependente de várias outras para ser executada.

- Identidade - a identidade de uma abstração é o seu nome, sua identificação dentro do código e do sistema como um todo. Aqui cabe passear pelo conceito da padronização. Um código bem escrito é fácil de entender, portanto, fácil de manter. Existem regras, padrões e convenções disponíveis para algumas linguagens, por exemplo, a linguagem de programação Java, por convenção, diz que toda classe deve começar com uma letra maiúscula e, de preferência, não pode conter letras não ASCII. A linguagem de programação Delphi preconiza que a classe deve ser iniciada pela letra T maiúscula acompanhada pelo nome seguindo o padrão infix caps. Por exemplo, Pessoa. Essa pode ser a identidade de um objeto que irá representar pessoas do mundo real.
- Propriedade - as propriedades são as características do objeto em abstração (também podem ser encontradas na literatura, definidas como atributos). Tendo em mente que um objeto representa algo do mundo real e, se no mundo real esse objeto possui características que o definem, quando o abstraímos, trazemos para a orientação a objeto as características que nos interessam. Por exemplo, o objeto Pessoa, que definimos acima. Para o programa em desenvolvimento, foi analisada a necessidade de se registrar as seguintes características/propriedades: nome, RG, CPF, data de nascimento, endereço e telefone.
- Métodos - os métodos representam o que esse objeto pode fazer, ou seja, as ações que ele deverá executar, seu comportamento. Por exemplo, quais são as ações possíveis de se realizar para o objeto Pessoa? Inserir(), Alterar(), Excluir(), Listar().

A modelagem para a abstração do objeto Pessoa é representada pela Figura 18 – Classe Pessoa.

É importante considerar nesse momento os termos Objeto e Classe. Ficou fácil perceber que o Objeto representa uma abstração do mundo real envolvido no problema a ser resolvido pelo software. Um objeto é essencialmente uma coleção de atributos e métodos válidos para manipular outros objetos (ENGHOLM, 2010, p. 119). Uma Classe representa um conjunto de objetos que possuem as mesmas propriedades. Considere o objeto Pessoa que foi utilizado como exemplo para a abstração. Esse objeto pode ser dividido (especializado) em Pessoas diferentes, como Clientes, Fornecedores, Alunos, Professores, Médico, Paciente, Enfermeiro etc., mas suas propriedades e métodos permanecem iguais. Portanto,

objetos são organizados em classes, e objetos da mesma classe possuem as mesmas propriedades e os mesmos métodos. Aproveitando a Figura 17, para ilustrar, observe que a Classe é Pessoa, e os objetos poderiam ser: Maria -> Atributos: 21-01-1998, Rua das Flores, Maria dos Anjos, 5.749.846-89, 000.145.568-89, 44 3222 2015. Métodos: Alterar(), Excluir(), Inserir(), Listar(). João 10-10-1960, Avenida Roxa, Pedro do Céu, 1.111.222-11, 000.256.145-10, 44 3221 2112. Métodos Alterar(), Excluir(), Inserir(), Listar(), pois, apesar de possuírem valores diferentes, compartilham os mesmos atributos e métodos.

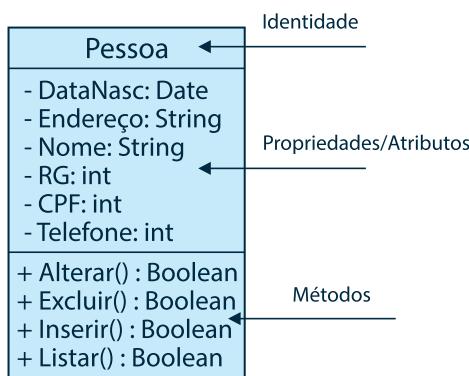


Figura 17: Classe Pessoa

Fonte: a autora.

ENCAPSULAMENTO

“As pessoas que usam os objetos não precisam se preocupar em saber como eles são constituídos internamente acelerando o tempo de desenvolvimento” (CORREIA; TAFNER, 2006, p. 13). Esse é o grande propósito do encapsulamento. Na orientação a objetos, os objetos possuem tanto dados quanto métodos. Como vimos na definição de métodos, eles são pedaços de códigos, são funções que dizem o que aquele objeto pode fazer. O encapsulamento oculta essa codificação e o trata como um “componente”, que permite que você use suas propriedades e métodos, mostrando como as partes do objeto se relacionam com o exterior, e você não tem que se preocupar com dados e códigos que implementam o comportamento dos objetos da classe. Muitos autores utilizam o sinônimo “caixa preta”

para o encapsulamento, onde se vê o exterior, mas não precisa se preocupar com o que acontece lá dentro.

O encapsulamento, na maioria das linguagens de programação orientada a objetos, é praticado utilizando métodos especiais conhecidos por *getters* e *setters*, onde o método *setter* é responsável por alterar ou inserir valores nos dados do objeto, e o método *getter* é utilizado para recuperar valores encapsulados no objeto e pode ser implementado em três níveis de acesso: Privado, Público e Protegido.

- Público (public): os atributos e métodos da classe são visíveis dentro da classe e para classes externas.
- Privado (private): os atributos e métodos da classe não são visíveis a nenhuma classe externa. São somente visíveis dentro da classe.
- Protegido (protect): os atributos e métodos da classe não são visíveis a nenhuma classe externa. São visíveis dentro da classe e para as classes herdeiras.

Observe na Figura 18, Exemplo de Encapsulamento, a modelagem de um objeto em que todos os atributos são Privados e os métodos são públicos. Essa configuração impede, por exemplo, uma quantidade de meses para a aplicação financeira diferente da estabelecida pela regra de negócio, codificada (ELGHOLM, 2010, p. 125).

AplicaçãoFinanceira
<pre> - idAplicaçãoFinanceira : int - descrição : String - taxasDeJuros : double - quantidadeDeMeses : int - valorAplicação : double </pre>
<pre> + setIdOperacaoFinanceira() : void + getIdOperacaoFinanceira() : void + setDescrição(descrição : String) : String + getDescrição() : String + setTaxaDeJuros(taxaDeJuros : double) : Double + getTaxaDeJuros() : double + setQuantidadeDeMeses(quantidadeDeMeses : int) : int + getQuantidadeDeMeses() : int + setValorAplicacao() : double + getValorFuturoAplicaco() : double </pre>

Figura 18: Exemplo de Encapsulamento

Fonte: adaptado de Elgholm (2010, p. 125).



REFLITA

“Trocava toda minha tecnologia por uma tarde com Sócrates.”

Fonte: Steve Jobs.

HERANÇA

O terceiro pilar da orientação a objetos é a possibilidade de compartilhamento de atributos, métodos entre as classes que compõem o sistema. Esse mecanismo, denominado Herança, permite criar novas classes a partir de classes já existentes (ELGHOLM, 2010, p. 128).

Na modelagem, essa característica é representada como **especialização** e funciona da seguinte maneira: considere a modelagem de um sistema financeiro, que deverá administrar tanto funcionários quanto os clientes de um banco. Durante o processo de análise, o engenheiro de software identifica que Clientes e Funcionários possuem várias características similares e também que várias ações deverão ser aplicadas a ambos os objetos, sobrando somente características exclusivas dos objetos que os diferenciam. A Tabela 3, Propriedades e Métodos dos Objetos Clientes e Funcionários, destaca essas características em comum.

PROPRIEDADES DOS OBJETOS

Funcionário	Cliente
Nome	Nome
Endereço	Endereço
dataNascimento	dataNascimento
CPF	CPF
RG	RG
telefone	telefone
cargo	dataCadastroCliente
salário	contaCorrente
dataAdmissao	aplicacoesFinanceiras
	emprestimosBancarios

MÉTODOS DOS OBJETOS	
Funcionário	Cliente
setNome()	setNome()
getNome()	getNome()
setEndereco()	setEndereco()
getEndereço()	getEndereço()
setDataNascimento()	setDataNascimento()
getDataNascimento()	getDataNascimento()
setCPF()	setCPF()
getCPF()	getCPF()
setRG()	setRG()
getRG()	getRG()
setTelefone()	setTelefone()
getTelefone()	getTelefone()
setCargo()	insereEmprestimo()
getCargo()	insereContaCorrente()
setSalario()	insereAplicacaoFinanceira()
getSalario()	retornaEmprestimosBancarios()
setDataAdmissao()	retornaContasCorrentes()
getDataAdmissao()	getContaCorrente()
	getEmprestimosBancarios()
	retornaAplicacoesFinanceiras()
	getAplicacoesFinanceiras()
	setDataCadastroCliente()
	getDataCadastroCliente()

Tabela 3: Propriedades e Métodos dos Objetos Clientes e Funcionários

Fonte: a autora.

Para aperfeiçoar o código e não ser necessário repetir os atributos e os métodos comuns para as duas classes, entra o mecanismo da Herança. Cria-se uma entidade em que se incluirá o que é comum aos dois objetos e, a partir dela, derivar especializações.

A Figura 19, Especialização da Classe Pessoa, representa a modelagem da classe Pessoa.

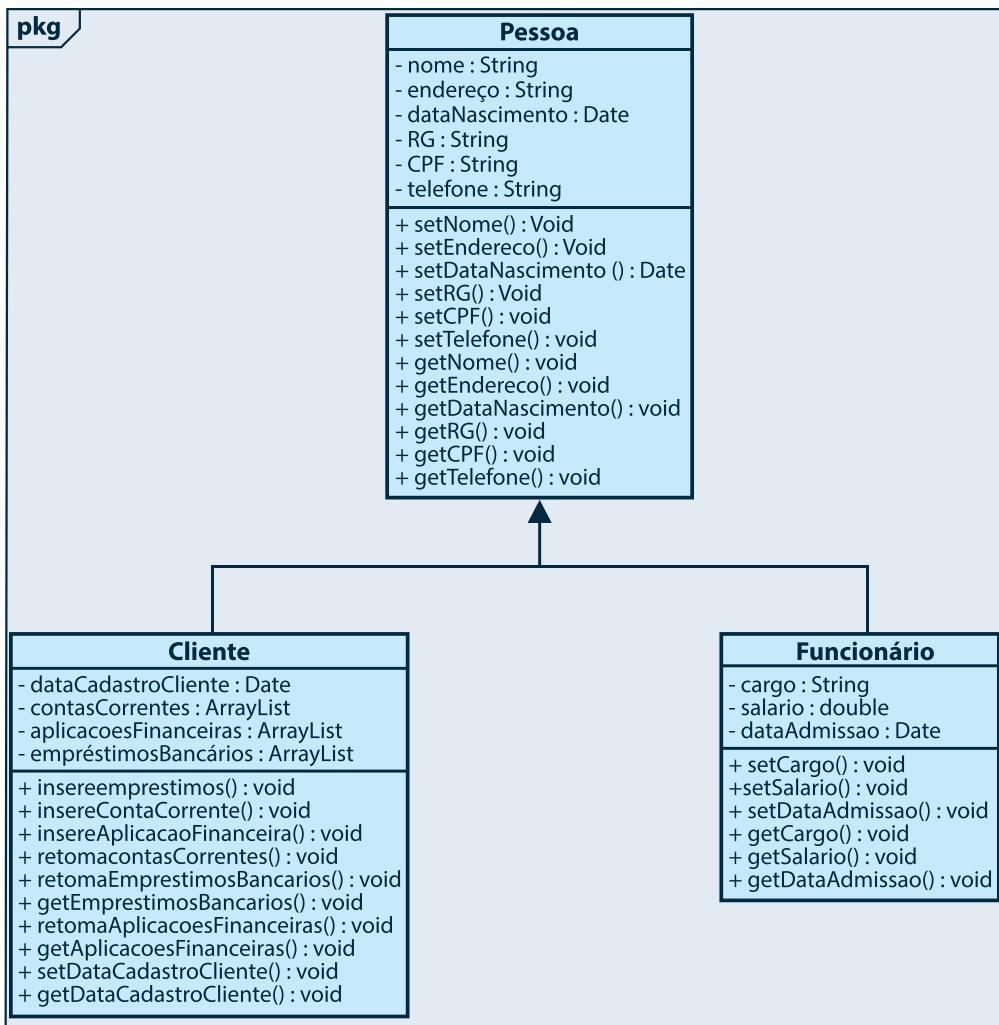


Figura 19: Especialização da Classe Pessoa

Fonte: a autora.

Na engenharia de software, a classe de onde as outras são especializadas é chamada **superclasse**, e as classes especialistas são chamadas de **subclasses**. Ao construir uma classe a partir de outra, a nova classe pode herdar os estados e os

comportamentos da superclasse, e pode usá-las além das próprias (ENGHOLM, 2010, p. 128). Na Figura 19, a classe Pessoa é a superclasse (generalização) e as classes Funcionário e Cliente são subclasses (especializações de Pessoas).



REFLITA

"Meus filhos terão computadores, sim, mas antes terão livros. Sem livros, sem leitura, os nossos filhos serão incapazes de escrever - inclusive a sua própria história."

Fonte: Bill Gates.

POLIMORFISMO

Para explicar o conceito de polimorfismo, apelo novamente ao meu sistema de vincular o significado da palavra à sua função no contexto da engenharia de software. Polimorfismo é a qualidade ou o estado de ser capaz de assumir diferentes formas. Na orientação a objeto, ele está extremamente vinculado ao mecanismo de herança, pois ele possibilita que métodos herdados de uma superclasse sejam reescritos, isto é, uma mesma mensagem pode ser interpretada de maneiras diferentes. Em outras palavras, o polimorfismo consiste na alteração do funcionamento interno de um método herdado de um objeto pai (GASPAROTTO, 2014).

Para exemplificar, vamos utilizar o problema proposto pelo Professor José Carlos Macoratti, no seu artigo Programação Orientada a Objetos em 10 lições práticas – Parte 07 (2014, online).

Problema: Você deseja controlar sua conta bancária pessoal registrando os saques, depósitos e controlando o saldo da conta.

Análise: Classe Conta responsável por definir os comportamentos e atributos de qualquer Conta. Propriedades: tipoConta. Métodos básicos: Sacar(); Depositar(). Superclasse: Conta; Subclasses: ContaPoupança; ContaInvestimento. Como mostra a Figura 20, Classes para Polimorfismo.

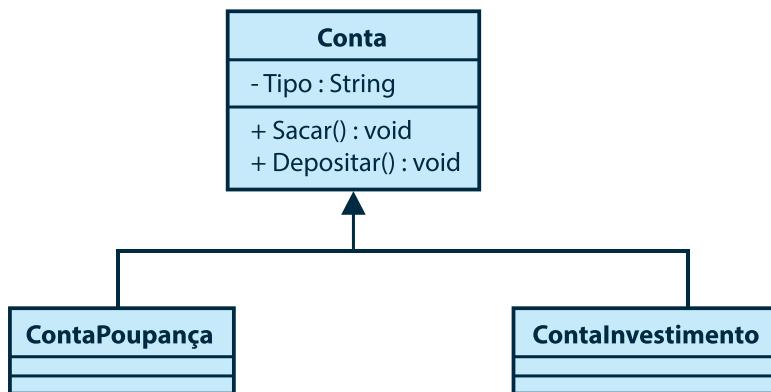


Figura 20: Classes para Polimorfismo
Fonte: adaptada de Macoratti (2014).

Até aqui, sem novidades, modelamos uma herança. Não temos como modelar o polimorfismo, por isso vou apresentar o código do Professor Macoratti na linguagem VB .NET. O polimorfismo será promovido usando a herança e reescrevendo os métodos. Observe os trechos de código nas Figuras 21, Código para Polimorfismo em VB.NET, e Figura 22, Código para Polimorfismo em VB.NET 2.

```

1  Public Class ContaPoupança
2      Inherits Conta
3      Public Sub New(tipoConta As String)
4          MyBase.New(tipoConta)
5      End Sub
6      Public Overrides Sub Sacar()
7          Console.WriteLine("Sacando da conta de poupança")
8      End Sub
9
10     Public Overrides Sub Depositar()
11         Console.WriteLine("Depositando na conta de poupança")
12     End Sub
13 End Class
  
```

Figura 21: Código para Polimorfismo em VB.NET.
Fonte: adaptada de Macoretti (2014).

```

1 Module Module1
2 Sub Main()
3     Dim conta As Conta() = New Conta(1) {}
4     conta(0) = New ContaPoupanca("Poupança do Macoratti")
5     conta(1) = New ContaInvestimento("Conta de Investimento do Macoratti")
6     MovimentarConta(conta(0))
7     MovimentarConta(conta(1))
8     Console.ReadKey()
9 End Sub
10 Public Sub MovimentarConta(_conta As Conta)
11     Console.WriteLine(_conta.Tipo)
12     _conta.Sacar()
13     _conta.Depositar()
14 End Sub
15 End Module

```

file:///C:/vbn/OOP_Polimorfismo1/OOP_Polimorfismo1.vb

Poupança do Macoratti
Sacando da conta de poupança.
Depositando na conta de poupança.
Conta de Investimento do Macoratti
Sacando da conta de investimento.
Depositando na conta de investimento.

Figura 22: Código para Polimorfismo em VB.NET 2

Fonte: Macoratti (2014).

O Professor Macorati (2014, online) explica,

no método MovimentarConta estamos usando o método Sacar() e Depositar() para cada tipo de conta que foi instanciada e cada objeto sabe realizar o comportamento em resposta à mesma chamada do método. Aqui estamos usando o conceito de polimorfismo, pois o método Sacar() e o método Depositar() são executados e a decisão de qual comportamento será usado ocorre em tempo de execução. Para usar o polimorfismo os objetos precisam executar as mesmas ações (métodos) mesmo que possuam comportamentos diferentes.

Concluindo essa rápida revisão sobre o paradigma da programação orientada a objetos, generalizando, temos a orientação a objetos como sendo um conjunto de encapsulamento com abstração e polimorfismo (ENGHOLM, 2010).

Muitas linguagens de programação modernas suportam o conceito de abstração de dados, porém, o uso de abstração juntamente com polimorfismo e herança, como suportado em orientação a objetos, é um mecanismo muito mais poderoso.



SAIBA MAIS

Coesão e Acoplamento em Sistemas Orientados a Objetos

Muitos desenvolvedores de software percorreram caminhos que passaram pelo desenvolvimento procedural antes de chegarem à orientação a objetos.

Diferentemente do paradigma orientado a objetos, no qual a subdivisão de sistemas é baseada no mapeamento de objetos do domínio do problema para o domínio da solução, o desenvolvimento procedural não possui uma semântica forte que oriente a subdivisão de sistemas. Nesse contexto, muitos conceitos e métricas foram definidos para avaliar e auxiliar a subdivisão de sistemas. Dois desses conceitos são especialmente importantes por terem grande influência na qualidade dos sistemas desenvolvidos: coesão e acoplamento.

Leia mais em: Artigo Java Magazine 77 - Coesão e Acoplamento em Sistemas Orientados a Objetos, disponível em: <<http://www.devmedia.com.br/artigo-javamagazine-77-coesao-e-acoplamento-em-sistemas-orientados-a-objetos/16167#ixzz3lqLLQdz9>>. Acesso em: 21 out. 2015.

Fonte: Luque (online).

LINGUAGEM DE MODELAGEM UNIFICADA - UML®

Antes de iniciarmos nossa discussão sobre a UML®, gostaria de deixar claro que não é objetivo desse conteúdo a descrição completa de cada conceito. Pretendo abordá-lo como complemento para nosso estudo sobre a modelagem de software.

É comum você encontrar na literatura a UML® definida como um método, ela até pode ser encarada como parte dos procedimentos e técnicas necessários para o desenvolvimento de um software, mas, por concepção, a UML® é uma linguagem.

Ela disponibiliza um meio sistemático, um conjunto de símbolos com o intuito de comunicar ideias. O objetivo da UML® é fornecer aos profissionais de software ferramentas para análise, projeto e implementação de sistemas. As versões iniciais da UML® surgiram de três principais métodos orientados a objeto

(Booch, OMT e OOSE) e incorporaram vários artefatos baseados nas melhores práticas de modelagem de *designer*, programação orientada a objetos e modelagem de arquiteturas.

A UML® nos possibilita cinco visões. Observe a Figura 23, Visões UML®.



Figura 23: Visões UML®

Fonte: UML (online).

A visão do caso de uso envolve os casos de uso que descrevem o comportamento do sistema pelo ponto de vista de seus usuários. Não representa a organização interna do software, mas determina, de forma preliminar, sua arquitetura. Os aspectos estáticos dessa visão na UML® são representados pelos diagramas de caso de uso; os aspectos dinâmicos são representados pelos diagramas de interação, diagrama de gráfico de estados e diagrama de atividades.

A visão do projeto representa as classes e suas colaborações. Nessa visão, as funcionalidades que o sistema irá desempenhar são modeladas. A UML® utiliza, para captar os aspectos estáticos dessa visão, os diagramas de classes e de objetos. E, para os aspectos dinâmicos, os diagramas de interações, os diagramas de estados e os diagramas de atividades.

A visão do processo envolve os elementos relacionados ao desempenho do sistema. Mecanismos de concorrência, processos etc. são representados na UML® por meio dos mesmos diagramas utilizados para a visão do projeto, mas com o foco nas classes ativas e na representação de seus processos e concorrência.

A visão da implementação modela os componentes e arquivos que, reunidos, produzem o sistema executável. A visão da implantação representa como os componentes e arquivos que compõem o sistema serão organizados e distribuídos para a sua instalação. Os diagramas UML® que modelam os aspectos estáticos dessa visão são os diagramas de implantação, e os aspectos dinâmicos podem ser representados nos diagramas de interações, diagramas de atividades e gráficos de estados.

Essas visões são representadas por diagramas, que estão organizados em duas grandes categorias: diagramas que modelam a estrutura do sistema (estáticos) e diagramas que modelam o comportamento do sistema (dinâmicos).

- Diagramas estáticos: diagrama de classes, diagrama de objetos, diagrama de componentes e diagrama de distribuição.
- Diagramas dinâmicos: diagrama de casos de uso, diagramas de interação – sequência e colaboração – diagrama de atividades e diagrama de estados.

Cada visão é descrita em um número de diagramas que contém informação enfatizando um aspecto particular do sistema. Analisando-se o sistema por meio de visões diferentes é possível se concentrar em um aspecto de cada vez.

ESPECIFICAÇÃO UML®

A especificação da UML® é composta por duas partes:

- Semântica - especifica a sintaxe abstrata dos conceitos de modelagem estática e dinâmica de objetos.
- Notação - especifica a notação gráfica para a representação visual da semântica, composta pelos itens, relações e diagramas.

Os itens podem ser estruturais, representando a parte mais estática do modelo: classes, interface, colaborações, caso de uso, classes ativas, componentes, nós. Podem ser comportamentais, que representam um comportamento no tempo e no espaço: interação, máquina de estado. De agrupamento: pacote, que representam um grupo de outros itens, que podem ser estruturais ou comportamentais; por último, itens de notação são comentários utilizados para descrever, esclarecer sobre quaisquer elementos do modelo.

São quatro os tipos de relacionamentos contidos na UML®: Dependência, Associação, Generalização e Realização.

Os Diagramas, como estudamos acima, estão organizados em duas grandes categorias: diagramas que modelam a estrutura do sistema (estáticos) e diagramas que modelam o comportamento do sistema (dinâmicos).

ITENS ESTRUTURAIS

Classe

Estudamos no tópico Pilares da Orientação a Objetos que uma Classe pode ser vista como um *template* que representa um conjunto de objetos que compartilham os mesmos atributos, métodos e relacionamentos (ENGHOLM, 2010, p. 121). A representação de uma classe é apresentada pela Figura 17 – Classe Pessoa.

Na UML®, a representação de um atributo tem que conter no mínimo uma identificação e pode ser complementada por outras propriedades, como, por exemplo, o seu tipo ou um valor inicial.

A notação para a visibilidade dos elementos que compõem as classes (atributos, e métodos) na linguagem UML® é:

- + para atributos públicos
- - para atributos privados
- # para atributos protegidos

Interface

Em modelagem UML®, interfaces são as estruturas que definem o conjunto de operações que outras estruturas do modelo, como classes ou componentes, devem implementar. Cada interface especifica um conjunto de operações bem definido que possuem visibilidade pública. É possível especificar os seguintes tipos de interfaces (IBM Knowledge Center):

- Interfaces fornecidas: representa o que a classe faz.
- Interfaces requeridas: como uma classe faz as tarefas.

Na prática, lemos a interface da seguinte maneira: “quem desejar ser autenticável precisa saber autenticar dado um inteiro e retornando um booleano”. Ela é um contrato no qual quem assina se responsabiliza por implementar esses métodos (INTERFACES, online). A Figura 24, Interface UML®, representa o símbolo para a modelagem da Interface pela UML®.



Figura 24: Interface UML

Fonte: a autora.

Colaborações

As colaborações definem as interações. Representam os elementos que funcionam em conjunto a fim de garantir um comportamento cooperativo maior que a soma de todos os elementos, isto é, a implementação do padrão que forma o sistema.

Caso de uso

Um caso de uso representa um conjunto de atividades que irão produzir um resultado para o ator relacionado. Eles modelam as funções que o sistema deve realizar a partir das interações entre os usuários e o sistema. O caso de uso se preocupa somente em dizer “o que” o sistema deve fazer e não “como” ele deve fazer. A Figura 25, Caso de uso UML®, representa o símbolo para a modelagem do caso de uso pela UML®.



Figura 25: Caso de uso UML®

Fonte: a autora.

Classes ativas

Uma classe ativa se difere da Classe, pois seus objetos representam elementos que são concorrentes com outros elementos. A Figura 26, Classe Ativa UML®, representa o símbolo para a modelagem da classe ativa pela UML®.

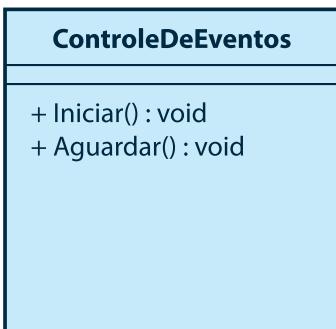


Figura 26: Classe Ativa UML®

Fonte: a autora.

Componentes

Um componente é uma parte física de um sistema, sua função é garantir a realização de um conjunto de interfaces; representa um empacotamento físico de elementos relacionados logicamente (normalmente classes). Por exemplo, executáveis, bibliotecas, tabelas, banco de dados etc. A Figura 27, Componente UML®, representa o símbolo para a modelagem do componente pela UML®.

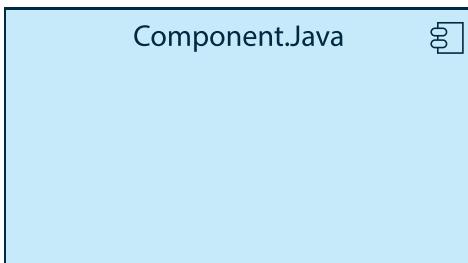


Figura 27: Componente UML®

Fonte: a autora.

Nó

Um nó representa um elemento computacional físico, com alguma capacidade de processamento e memória. Por exemplo, um computador, uma rede etc. A Figura 28, Nó UML®, representa o símbolo para a modelagem do nó pela UML®.

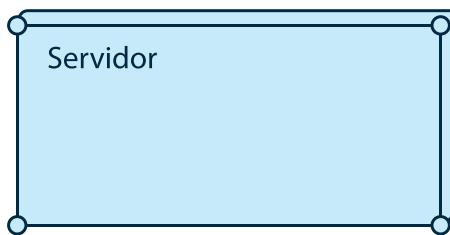


Figura 28: Nó UML®

Fonte: a autora.

ITENS COMPORTAMENTAIS

Interação

Interação representa as mensagens trocadas entre os objetos. É representada por uma seta.

Máquina de estado

Representa a sequência de estados que os objetos assumem no decorrer do tempo em resposta aos eventos aos quais são submetidos. A Figura 29, Máquina de

Estado UML®, representa o símbolo para a modelagem da máquina de estado pela UML®.

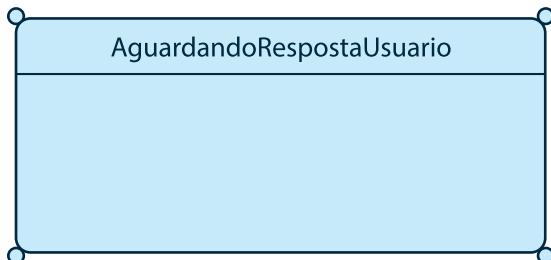


Figura 29: Máquina de Estado UML®

Fonte: a autora.

ITENS DE AGRUPAMENTOS

Chamados Pacotes, os itens de agrupamento têm o propósito de organizar os elementos em grupos. Um Pacote é puramente conceitual, não representa um comportamento de execução. A Figura 30, Pacote UML®, representa o símbolo para a modelagem do pacote pela UML®.

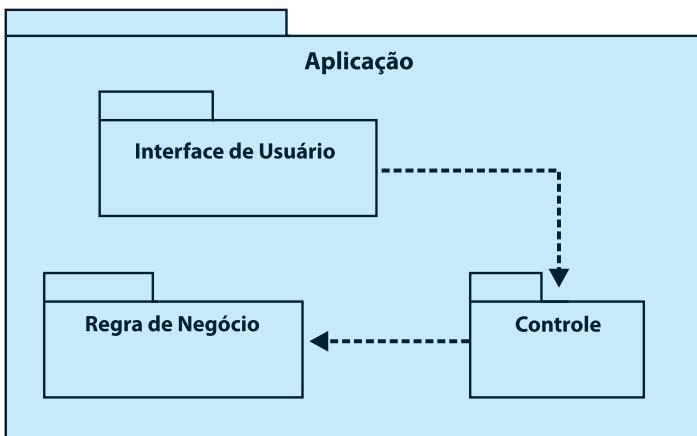


Figura 30: Pacote UML®

Fonte: adaptada de UML (online).

ITENS ANOTACIONAIS

São os símbolos utilizados para fazer anotações, descrever, esclarecer sobre qualquer elemento do modelo. A UML® os define como Nota. A Figura 31, Nota UML®, representa o símbolo para a modelagem da nota pela UML®.

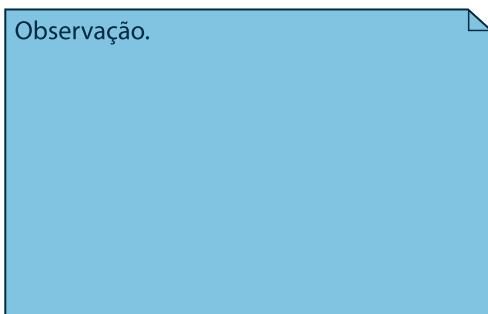


Figura 31: Nota UML®

Fonte: a autora.

RELACIONAMENTOS

Os relacionamentos conectam os objetos entre si, criando relações lógicas entre eles.

Dependência

Na UML®, um relacionamento de dependência é um relacionamento no qual um elemento usa ou depende de outro elemento. Indica que qualquer alteração no objeto, que chamaremos aqui de “pai”, causa uma alteração no objeto dependente. Um relacionamento de dependência também pode ser utilizado para representar precedência, em que um elemento deve preceder outro (IBM Knowledge Center).

Geralmente, os relacionamentos de dependência não são identificados. A Figura 32, Dependência UML®, ilustra uma dependência, representada por uma linha tracejada com uma seta aberta que aponta do objeto “pai” para o objeto dependente. Resumindo de uma forma bastante objetiva, relacionamento de

dependência ocorre quando a existência de uma classe depende da existência de outra. Por exemplo, em um sistema comercial que controla Pedidos de Clientes, só existirá um Item de Pedido se existir um Pedido.



Figura 32: Dependência UML®

Fonte: a autora.

Associação

Uma associação é um relacionamento entre dois objetos como classes ou casos de uso que tem por objetivo representar os motivos e as regras que conduzem os objetos ao relacionamento. As associações registram as propriedades dos objetos. Por exemplo, você pode utilizar um recurso de navegabilidade de uma associação para mostrar como um objeto de uma classe obtém acesso a um objeto de outra classe ou, em uma associação reflexiva, para um objeto da mesma classe. Cada extremidade de um relacionamento de associação possui propriedades que especificam sua função, multiplicidade, visibilidade, navegabilidade e restrições (IBM Knowledge Center).

Quanto à classificação, uma associação pode ser:

- **Normal/Simples:** representa somente a colaboração entre os objetos de elementos diferentes; a navegabilidade pode ser unidirecional ou bidirecional e é representada por um traço simples.
- **Qualificada:** um qualificador de associação é um atributo do elemento-alvo, que identifica uma instância entre as demais.
- **Recursiva:** acontece quando um objeto do elemento se conecta a outro contínuo nele próprio.
- **Agregação:** indica que um elemento é parte ou está contido em outro elemento, mas são independentes entre si (parte existe sem todo).
- **Composição:** onde um elemento está contido em outro, e a execução de um depende do outro (todo controla a execução da parte).

Quanto à multiplicidade, nos extremos do traço que representa a associação, é definido o número de instâncias permitidas para aqueles objetos. Uma multiplicidade do final da associação pode ter um dos seguintes valores: (1) um, (0) zero, (*) muitos.

O nome de uma associação deve descrever, de forma nítida, a natureza do relacionamento e deve ser um verbo ou frase simples. Uma associação é representada como uma linha sólida entre dois elementos. A boa prática recomenda sempre incluir o nome da associação ou um papel que a identifique; a definição do seu papel é muito útil na geração do código quando utilizadas ferramentas CASE. A Figura 33, Associação UML®, ilustra uma associação simples entre classes. A Figura 34 ilustra os relacionamentos do tipo agregação e composição.

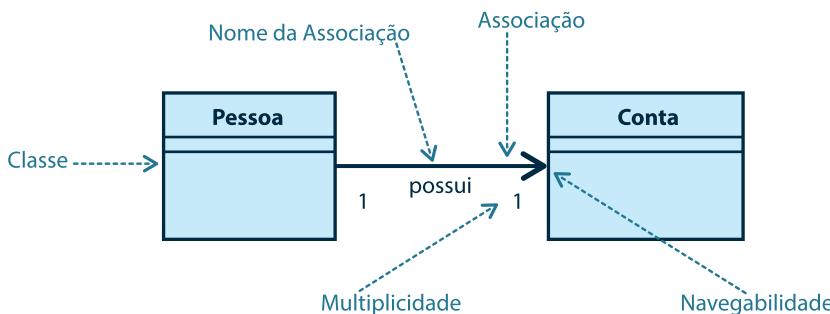


Figura 33: Associação UML®

Fonte: Barcelar (online).

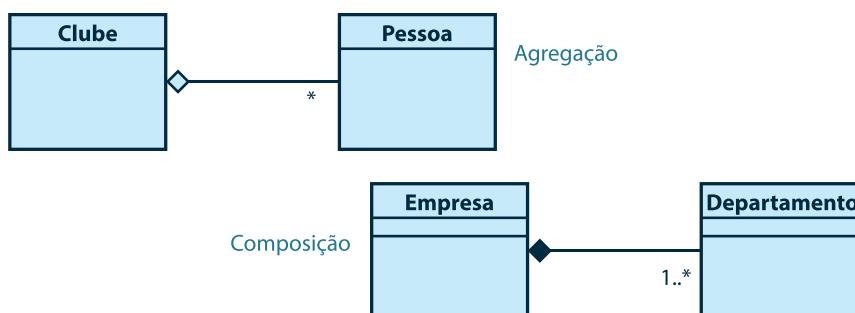


Figura 34: Agregação e Composição

Fonte: Barcelar (online).

Generalização

O relacionamento Generalização está fortemente relacionado com a característica Herança da orientação a objetos. Esse relacionamento representa a especialização de um objeto a partir de um outro com atributos mais gerais. Por exemplo, a especialização do objeto Pessoa para o objeto Funcionário, como vimos no exemplo representado pela Figura 19, Especialização da Classe Pessoa, quando estudamos orientação a objeto no tópico anterior.

Realização

Na modelagem UML®, um relacionamento de realização é aquele entre dois elementos, no qual um deles realiza o comportamento que o outro elemento especificou. É possível utilizar os relacionamentos de realização nos diagramas de classe e diagramas de componentes. Esse relacionamento é representado por uma seta tracejada com a cabeça de seta fechada e vazia que aponta do realizador para o especificador. A Figura 35, Realização UML®, ilustra uma realização.



Figura 35: Realização UML

Fonte: a autora.

DIAGRAMAS UML®

DIAGRAMAS ESTRUTURAIS

Diagramas de estrutura representam a estrutura estática do sistema e de partes do sistema em diferentes níveis de abstração e de implementação e também mostram como essas partes estão relacionadas umas com as outras. Os elementos em um diagrama de estrutura representam os conceitos significativos de um sistema e podem incluir abstrato, mundo real ou conceitos de implementação. Diagramas de estrutura utilizam conceitos de tempo, não mostram os detalhes de comportamento dinâmico.

Diagrama de Classe

O objetivo do diagrama de classe é representar a estrutura de um sistema em fase de projeto, de um subsistema ou ainda de um componente, como, por exemplo, as classes e interfaces relacionadas; ele inclui na denotação as características, limitações e relacionamentos (associações, generalizações, dependências etc.). Os Elementos que compõem o diagrama de classe são os seguintes: classe, relacionamentos e interfaces.

A estrutura de uma classe divide-se em três compartimentos, no primeiro fica registrado o nome da classe, no segundo, as propriedades (atributos), no terceiro, as operações (métodos). A UML® permite representar as classes com mais ou menos detalhes. Para representar, de forma plena, as características dos objetos, as classes são dotadas de várias simbologias. Observe a Figura 36, ela identifica todos os componentes discricionais da classe.

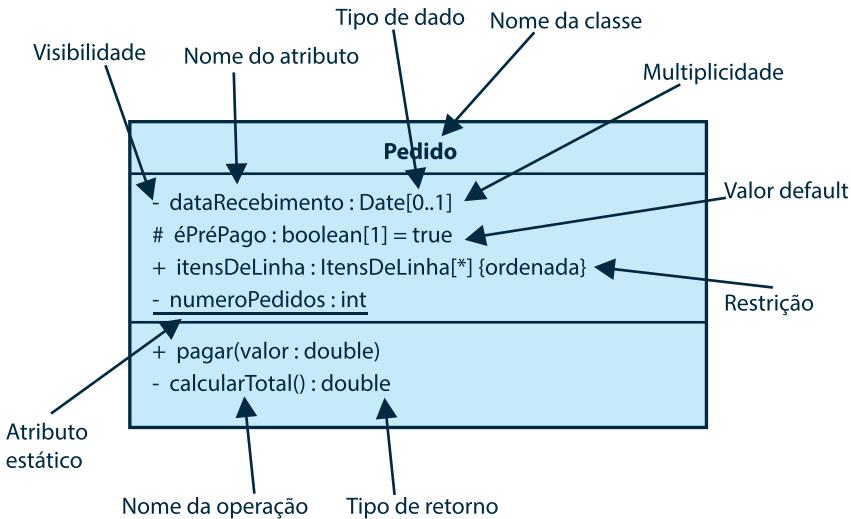


Figura 36: Estrutura da classe

Fonte: Barcelar (online).

Os relacionamentos ligam as classes entre elas, criando relações lógicas, como vimos, são classificados como: Associação (simples, agregação e composição), Generalização, Dependência e Realização.

A Figura 37 ilustra um diagrama de classes e seus componentes.

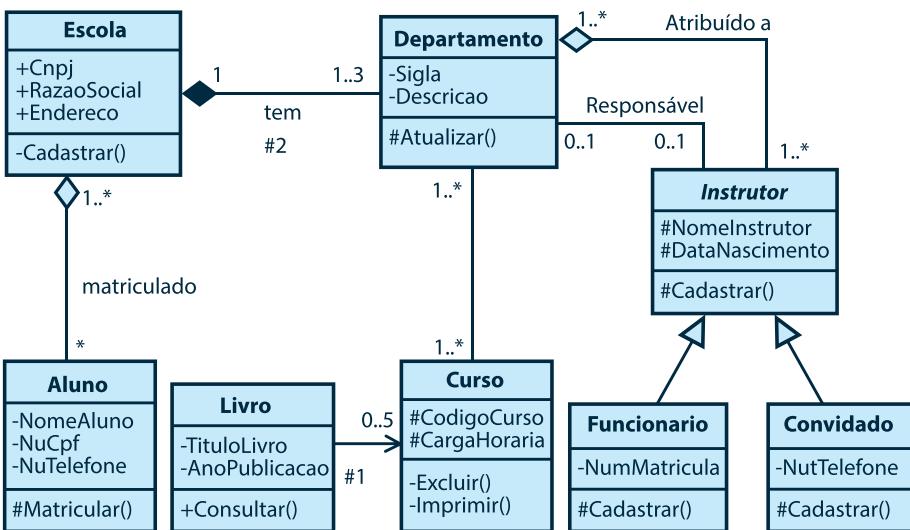


Figura 37: Diagrama de Classes

Fonte: Barcelar (online).

Diagrama de objetos

O diagrama de objetos é uma variação do diagrama de classes, ele representa uma instância do objeto, isto é, representa o sistema em um determinado momento de sua execução. Sua definição oficial está presente somente na UML® versão 1.4.2, as versões mais novas não contemplam essa definição.

um gráfico de instâncias, incluindo objetos e valores de dados, diagrama de objeto estático é uma instância de um diagrama de classes, com o objetivo de detalhar um estado do sistema em um ponto no tempo².

A mesma notação do diagrama de classes é utilizada no diagrama de objetos com duas exceções: a primeira, em que os objetos são escritos com seus nomes sublinhados, e na segunda, todas as instâncias em um relacionamento são mostradas.

Ambos os diagramas de classes e de objetos são úteis para exemplificar objetos e comportamentos complexos de uma classe diagramas complexos de classes auxiliando na sua compreensão. A Figura 38 ilustra um diagrama de objetos.

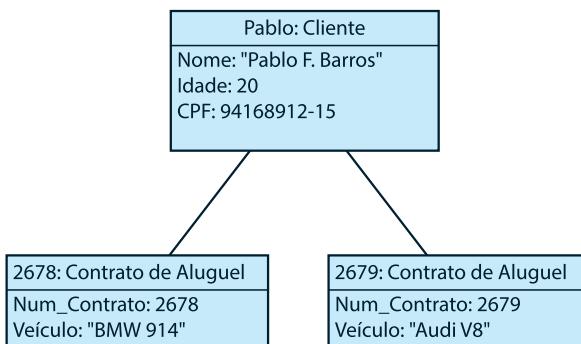


Figura 38: Diagrama de Objetos

Fonte: Linguagem de Modelagem Unificada (online).

Diagrama de pacotes

Um pacote é um conjunto de elementos agrupados. Esses elementos podem ser classes, diagramas, ou até mesmo outros pacotes. Ele é muito utilizado para ilustrar a arquitetura de um sistema, representando os pedaços do sistema repartidos em agrupamentos lógicos e suas dependências (ENGHOLM, 2010, p. 225).

² Disponível em: <<http://www.omg.org/spec/UML/2.5/Beta1/PDF/>>. Arquivo em PDF para download.

Os elementos que compõem esse diagrama são pacotes, dependências, pacotes de elementos, elementos importados, pacotes importados, pacotes mesclados. As Figuras 39, Diagrama de pacotes, e 40, Diagrama de pacotes (estrutura), ilustram formas diferentes de modelagem.

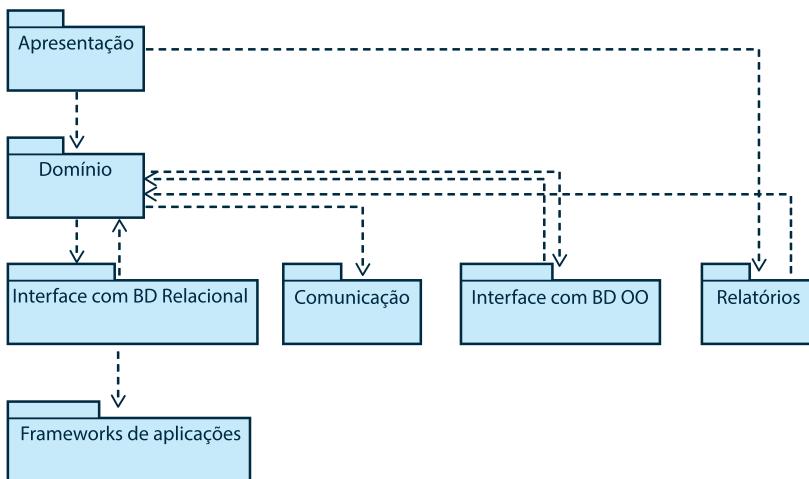


Figura 39: Diagrama de Pacotes

Fonte: adaptada de Diagrama de Pacotes (online).

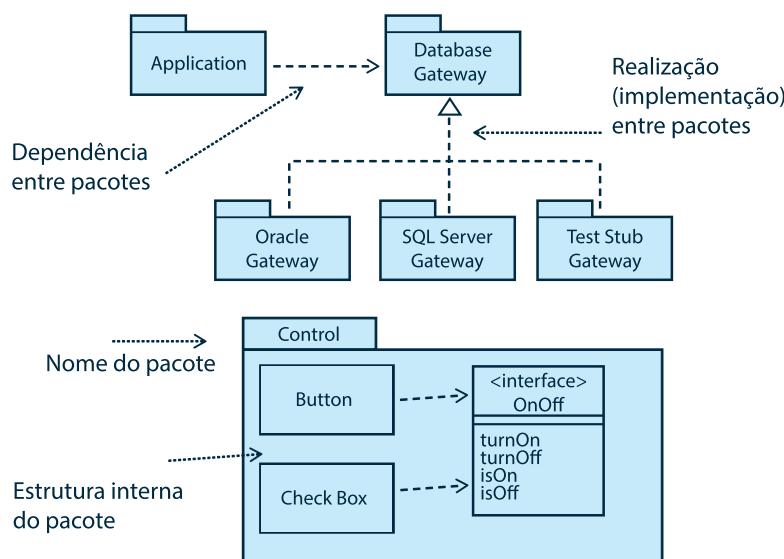


Figura 40: Diagrama de pacotes (estrutura)

Fonte: Barcelar (online).

Diagrama de componentes

Componentes de software podem ser definidos como pedaços de código que contêm um conjunto de interfaces comuns, por exemplo, os executáveis, as bibliotecas, as tabelas do banco de dados, *JavaBeans* etc. O diagrama de componentes permite decompor o sistema em subsistemas que detalham o funcionamento interno.

Os componentes representam a implementação na arquitetura física (linguagem de programação) dos conceitos e das funcionalidades definidas na arquitetura lógica (classes, objetos, relacionamentos). O Diagrama de componentes auxilia no processo de engenharia reversa pela organização do sistema e seus relacionamentos (ENGHOLM, 2010, p. 226). Os elementos que compõem um diagrama de componentes são: componentes, dependências, interface, interface provida, interface requerida, classes, portas, conectores, artefatos, componentes de realização.

Esse tipo de diagrama é muito utilizado para o Desenvolvimento Baseado em Componentes (CBD) e para descrever sistemas com *Service Oriented Architecture* (SOA). A Figura 41 exemplifica um diagrama de componentes.

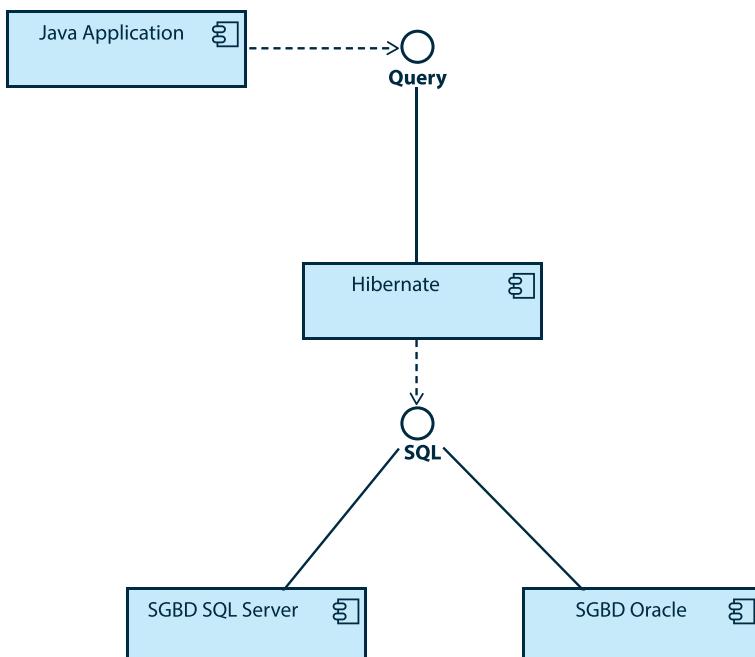


Figura 41: Diagrama de Componentes
Fonte: adaptada de Barcelar (online).

Diagrama de implantação

Mostra a arquitetura do sistema como implantação (distribuição) de artefatos de software para destinos de implementação. Representa a distribuição dos pacotes de sistema em execução nos equipamentos (hardwares). Pode ser utilizado, por exemplo, para mostrar diferenças em implementações de ambientes de desenvolvimento, teste ou de produção com os nomes de compilação específica ou servidores ou dispositivos de implantação (UML-diagrams).

Os elementos que compõem esses diagramas são:

- Nós, representando dispositivos ou ambientes de execução.
- Artefatos, representando código fonte, código binário, executáveis etc.

A Figura 42 ilustra a modelagem de um diagrama de implantação.

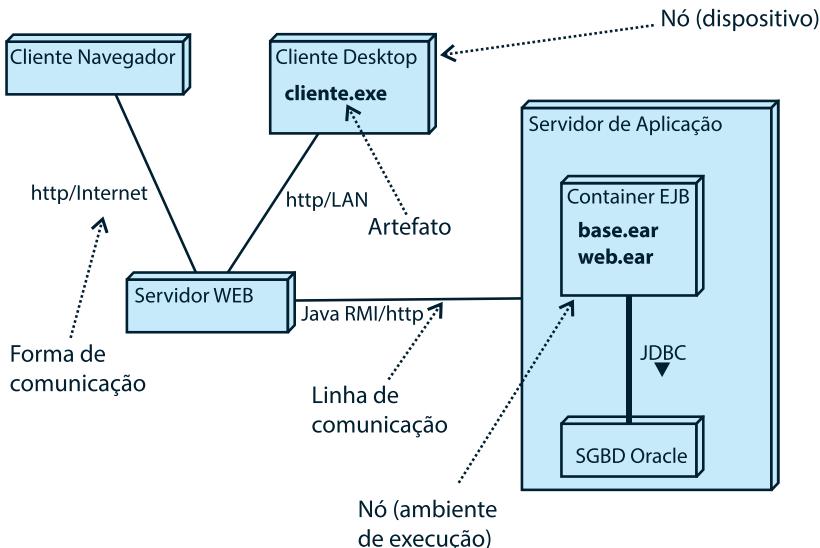


Figura 42: Diagrama de Implantação
Fonte: Barcelar (online).

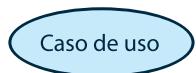
Diagramas comportamentais

Diagramas de comportamento mostram o comportamento dinâmico dos objetos em um sistema, comportamentos que podem ser descritos como uma série de mudanças no sistema ao longo do tempo (UML-diagrams).

Diagrama Caso de Uso

Os diagramas de casos de uso descrevem as funções principais de um sistema e identificam as interações entre o sistema e seu ambiente externo, representado por Atores. Esses Atores podem ser pessoas, organizações, máquinas ou outros sistemas externos (IBM Knowledge Center).

O diagrama é composto por Atores, Casos de uso e relacionamentos:

- 
- Representação do Ator. Um ator é um usuário do sistema, que pode ser pessoas, organizações, máquinas ou outros sistemas.
- 
- Representação do caso de uso. Um caso de uso define uma função do sistema.

Os relacionamentos no diagrama de caso de uso podem acontecer entre os atores, entre os casos de uso e entre um ator e um caso de uso. Pode ser do tipo associação, generalização, realização e entre casos de uso, do tipo *Include* e *Extend*.

- <<include>> representa que um caso de uso é essencial para o comportamento de outro.
- <<extend>> representa que um caso de uso pode ser acrescentado para descrever o comportamento de outro, mas não é essencial para isso.

A Figura 43, Diagrama de caso de uso Realizar Pedido Online, ilustra todos os elementos que o compõem.

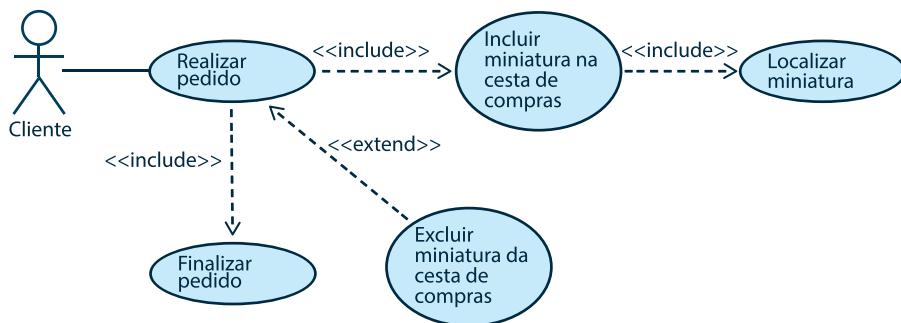


Figura 43: Diagrama de caso de uso Realizar Pedido Online

Fonte: adaptada de Engholm (2010, p. 217).

Um retângulo pode ser utilizado para delimitar os limites do sistema, isto é, até onde as operações são internas e a partir de onde existe interação de atores externos ao sistema. Observe a Figura 44, Diagrama de caso de uso Clínica médica, que representa os limites do sistema.

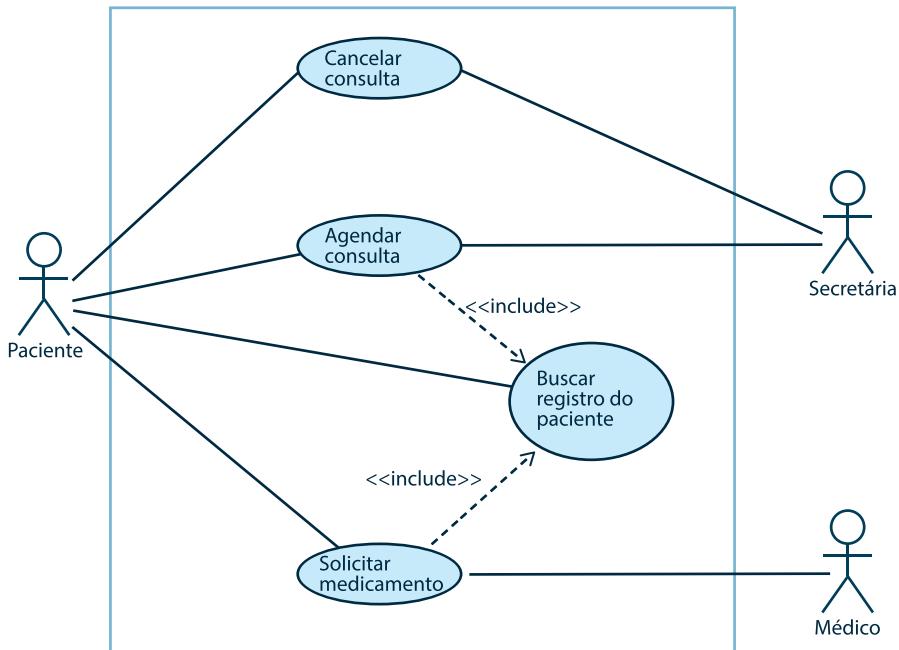


Figura 44: Diagrama de caso de uso Clínica médica
Fonte: adaptado de UFCG – Casos de Uso (online).

Diagrama de atividade

Um diagrama de atividades modela as atividades dos métodos, algoritmos ou, ainda, processos completos. Até a versão 2.0 da UML® esse diagrama era tratado como um caso especial do diagrama de estado, por isso grande parte dos elementos que compõem um diagrama de atividades foi herdada dos diagramas de estados.

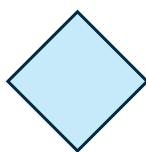
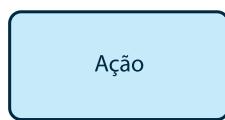
Um diagrama de atividade tem uma grande similaridade com os fluxogramas, ele é essencialmente um gráfico de fluxos. Enquanto os diagramas de sequência focam o fluxo de um objeto para outro, os diagramas de atividades enfatizam o fluxo de uma atividade para outra.

Faço aqui uma pequena pausa na discussão sobre o diagrama para lembrar que, didaticamente, a organização de apresentação dos diagramas tem que seguir uma sequência linear, mas, na prática, os diagramas podem ser utilizados concomitantemente, um em complemento do outro e em várias fases do projeto, por exemplo, os diagramas de atividades são úteis nas seguintes fases:

- Antes de iniciar um projeto, para modelar os fluxos de trabalho mais importantes.
- Durante a fase de requisitos, para ilustrar o fluxo de eventos descritos nos casos de uso.
- Durante as fases de análise e projeto, para auxiliar a definir o comportamento das operações.

Uma atividade na UML® é um elemento que descreve o nível mais alto do comportamento. O diagrama ilustra diversos nós de atividade e linhas de atividade que representam a sequência de tarefas em um fluxo de trabalho, resultando em um comportamento (IBM Knowledge Center).

Os elementos que compõem o diagrama são relacionados a seguir:



- Estado que representa um passo, uma atividade dentro de um fluxo de controle; não pode ser decomposto; não possui ações internas; não pode ser interrompido.
- Representa um ponto de decisão; realização de um teste para seguir um fluxo em detrimento de outro. Conforme a notação sugere, textos de identificação das condições devem ser envolvidos por colchetes.
- *Forks e Joins*. Representa um comportamento condicional. A diferença aqui, para o diagrama de atividades, é que uma bifurcação ou Fork representa uma atividade subdividida em outras; Join significa que as atividades sendo executadas em paralelo chegaram a um ponto de junção para então gerar uma nova atividade.

A construção do diagrama de atividades oferece elementos que promovem a representação da complexidade do software, como, por exemplo, as Raias (*Swimlanes*); a marcação da região de atividade interrompível, isto é, que existe a possibilidade de ser cancelada, interrompida; Zona de expansão; subatividades etc.

A Figura 45, Diagrama de Atividades Pedido, representa um diagrama de atividades com raias, representando os setores físicos de uma empresa que estão envolvidos no processo. A Figura 46, Diagrama Atividades Utilizando *Forks* e *Joins*, ilustra a utilização dos elementos representativos de comportamento condicional. No exemplo da Figura 46, é possível observar que, após a atividade de Receber Pedido, um *Fork* “reparte” as atividades, dando origem a duas outras (Preencher Pedido e Enviar Fatura), e um *Join* ilustra a condição de que somente com a finalização das atividades (Enviar Fatura e Receber Pagamento) é que se executará a próxima (Realizar Entrega).

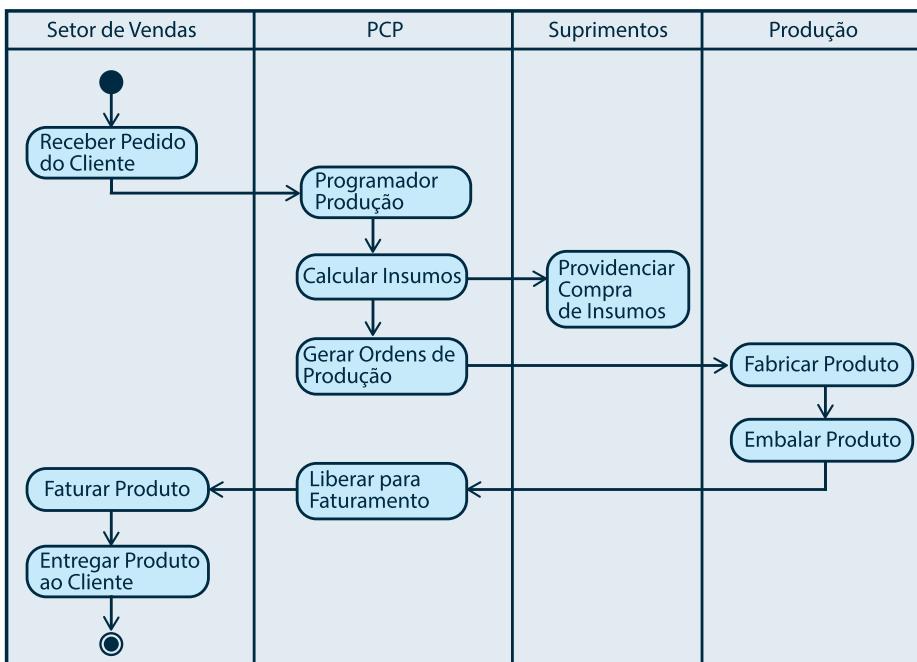


Figura 45: Diagrama de Atividades Pedido

Fonte: Artigo... (online).

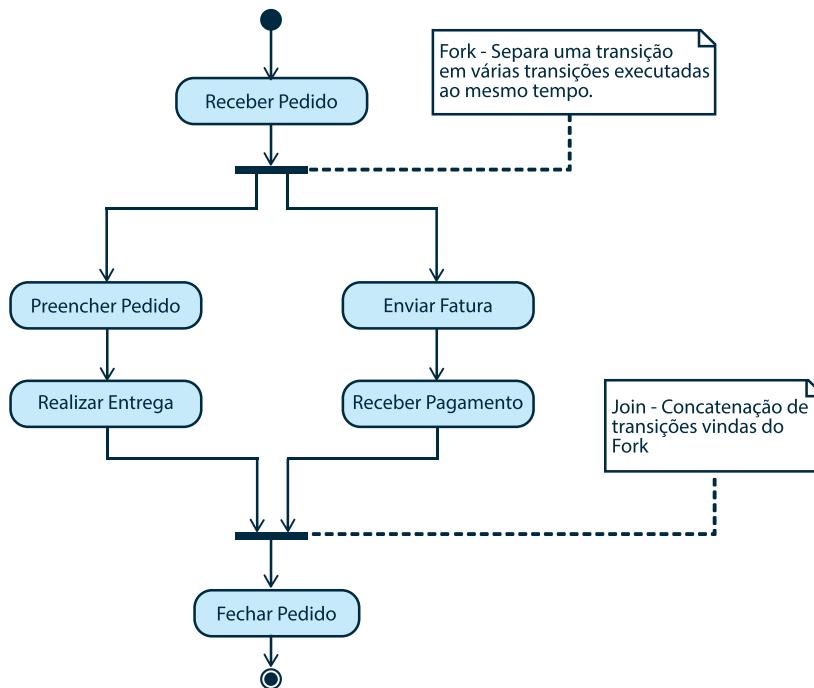


Figura 46: Diagrama Atividades Utilizando Forks e Joins
Fonte: Artigo... (online).

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

A Figura 47 ilustra a estrutura do Diagrama de Atividade.

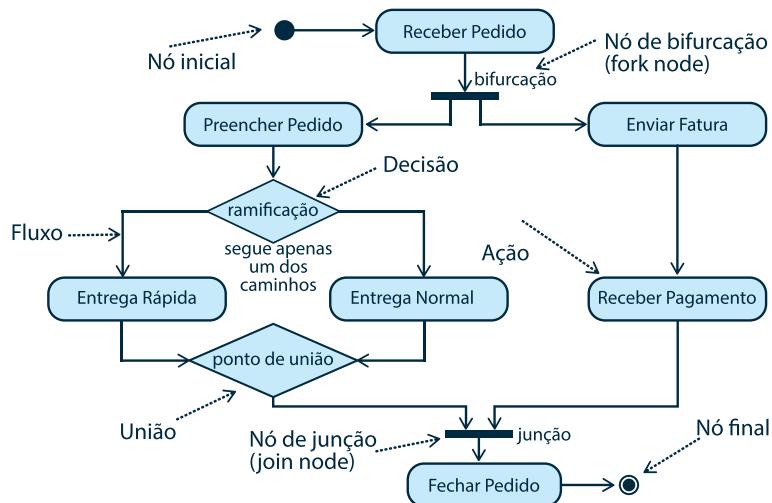


Figura 47: Diagrama de Atividade (estrutura)
Fonte: Barcelar (online).

Diagrama de estados

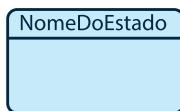
É uma representação gráfica da sequência de estados de um objeto e dos eventos que causam a transição de um estado para outro e também das ações resultantes da alteração de um estado (IBM Knowledge Center).

É usado para modelar o comportamento discreto por transições de estados finitos. Além de expressar o comportamento de uma parte do sistema, máquinas de estado também podem ser utilizadas para expressar o protocolo de utilização de parte de um sistema. Esses dois tipos de máquinas de estado são referidos como máquinas de estados comportamentais e máquinas de estado de protocolo (UML-diagrams).

Não é necessário representar o estado de todos os objetos, esse diagrama é recomendado para aqueles que possuem um comportamento mais complexo.

É muito aplicado na modelagem para o desenvolvimento de sistemas de tempo real ou dirigidos por eventos porque mostram o comportamento dinâmico dos objetos.

Os elementos que compõem um diagrama de estados estão relacionados a seguir:



- Um retângulo com as bordas arredondadas representa o estado do objeto.



- Representa um tipo especial de estado, o Estado inicial do objeto. É especial por que nenhum evento será capaz de devolvê-lo a este estado depois de iniciado o processo.



- Representa outro tipo especial de estado, o Estado final, mantendo o contexto; é um estado especial por que nenhum evento será capaz de devolvê-lo a ele depois de encerrado o processo.



- Transições, representadas por setas simples abertas.



- Forks e Joins, apresentados por um traço largo, representam um comportamento condicional. Fork (bifurcação), ou divisão dos estados, e Join (agrupamento) representam a junção de dois estados para proporcionar a transição para um novo.



SAIBA MAIS

O desenvolvimento de sistemas automatizados de informações, que apoiam as atividades de projeto e manufatura de produtos, deve seguir um modelo como referência para permitir uma melhor compatibilidade e portabilidade de tais sistemas, principalmente quando inseridos em um ambiente integrado de engenharia concorrente. Este artigo demonstra como a Linguagem de Modelagem Unificada (UML) pode ser aplicada em conjunto com o Modelo de Referência para Processamento Distribuído Aberto (ISO/RM-ODP), para o apoio ao desenvolvimento de sistemas de informações orientados a objetos. Enquanto o RM-ODP oferece um padrão para representação de diferentes pontos de vistas de tais sistemas, a UML é utilizada como notação para representação de cada uma dessas vistas. Um processo baseado em Use Cases é empregado para apoiar a evolução da representação das informações dentro desse modelo de referência. O ambiente de projeto de moldes de injeção é utilizado como exemplo para ilustração dos diagramas da UML.

Fonte: Costa (2001, online).

O estado em sua notação pode conter três repartições, no primeiro, como apresentado acima, mostra o nome do estado, os dois outros são opcionais e representam na sequência a variável do estado relacionando seus atributos (de acordo com o listado na classe) e a atividade (lista de eventos e ações). Um objeto passará por vários estados, por exemplo: momento em que foi criado, momento em que foi iniciado, momento em que executou uma solicitação, momento do seu encerramento etc.

Um estado tem várias propriedades: Nome, um rótulo para sua identificação e distinção de um estado para outro; Ações de entrada e saída, ações executadas ao entrar ou ao sair do estado; Transições internas, operações realizadas internamente e que não causam mudança de estado; Subestados e Eventos adiados, uma lista de eventos não manipulados, estado atual do objeto, que aguardam para serem manipulados pelo objeto em outro estado. Uma transição é um relacionamento entre dois estados que indica que após a execução das ações o objeto passará para um novo estado. O acionamento da transição acontece na mudança de estado, que delimita para o objeto o seu estado de origem e seu estado de

destino. Uma transição tem várias propriedades, não é nosso objetivo entrar nesse nível de detalhes. Um evento representa um valor, mensagem ou notificação que habilita a transição de estado. Por exemplo, uma interrupção do sistema operacional, uma função feita por outro objeto. Ele pode ser expresso por argumentos (valores recebidos junto com o evento), condição (expressão lógica, uma transição só ocorre se o evento acontecer e se a condição associada for verdadeira) e ação (cálculo, atribuição, envio de mensagem, etc.).

A Figura 48, Diagrama de Estado Registrar Pedido, ilustra um diagrama de estado.

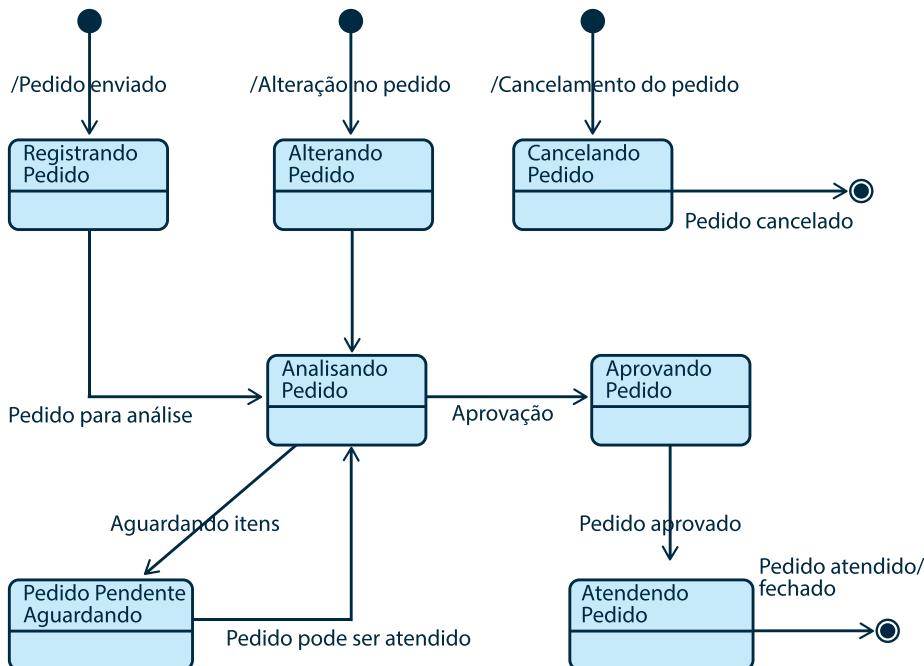


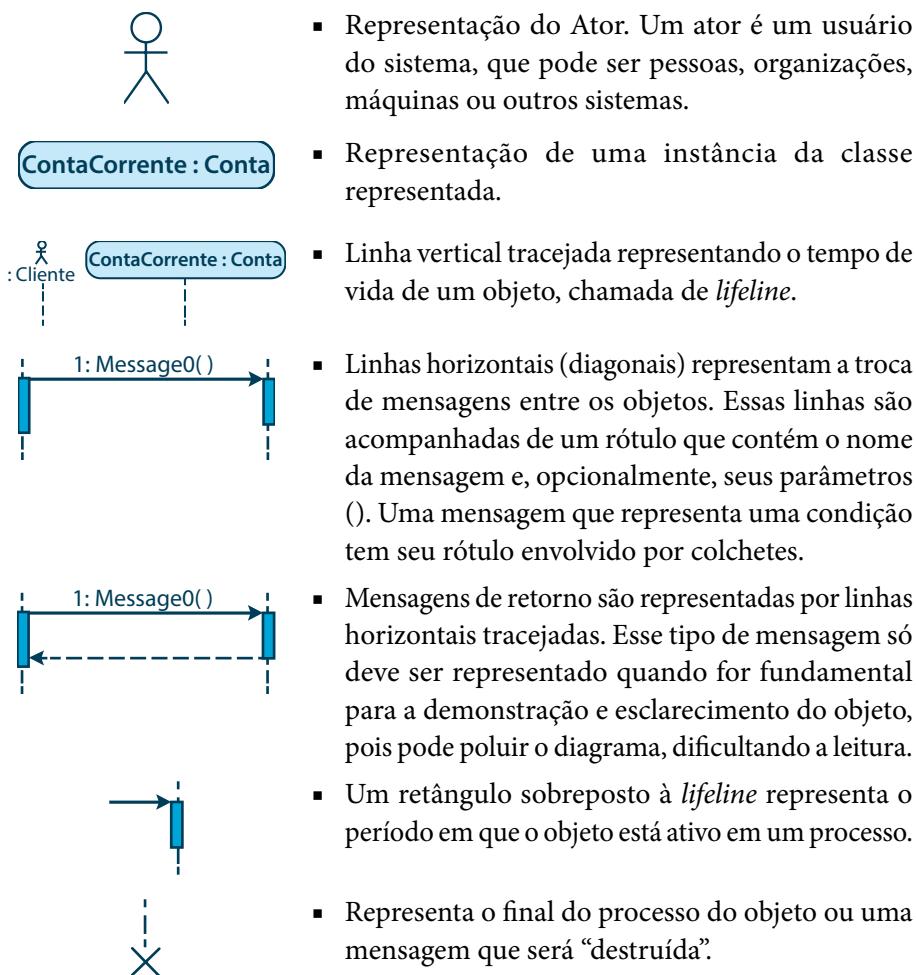
Figura 48: Diagrama de Estado Registrar Pedido
Fonte: adaptada de Diagrama de Estado (online).

Diagramas de Interação

Os diagramas de interação representam a colaboração dos objetos para um determinado comportamento do sistema, eles capturam o comportamento de um objeto dentro de um único caso.

DIAGRAMA DE SEQUÊNCIA

Em modelos UML, uma interação é um comportamento que representa a comunicação entre um ou mais elementos do diagrama (IBM Knowledge Center). Ilustram a sequência de acontecimentos dos eventos de um processo. Utilizado para visualizar e entender: quais condições devem ser satisfeitas, quais métodos devem ser disparados, qual a ordem em que os métodos são disparados. Um diagrama de sequência é composto pelos seguintes elementos:



As mensagens podem ser: síncronas (*wait*), assíncronas (*no wait*), fluxo de controle e de retorno. As trocas podem acontecer de Ator para Ator, de Ator para Objeto, de Objeto para Objeto, de Objeto para Ator.

Na unidade II, quando estudamos sobre os tipos de modelagem, conversamos sobre o diagrama de sequência, a Figura 8 - Diagrama de Sequência Transferir Dados é um exemplo de Diagrama de Sequência. A Figura 49, Diagrama de Sequência Conta Bancária, ilustra um diagrama de sequência.

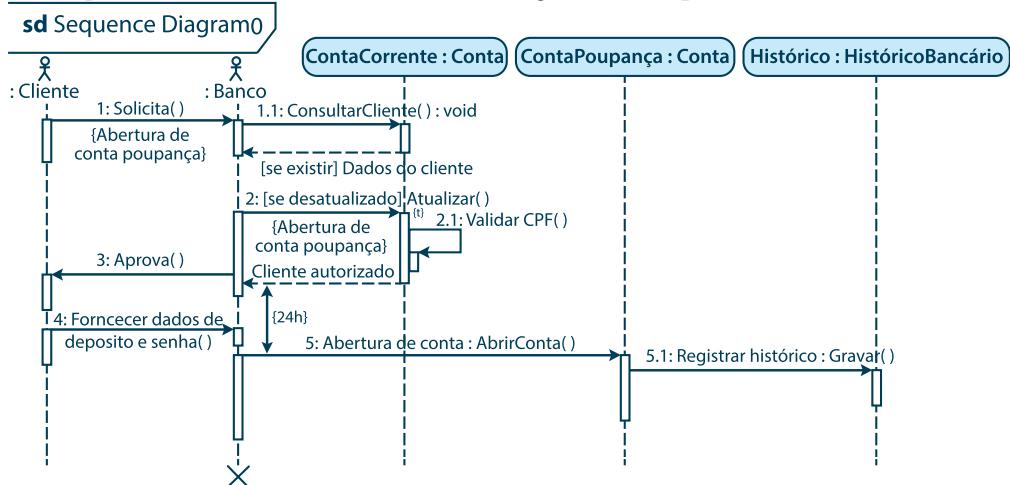


Figura 49: Diagrama de Sequência Conta Bancária

Fonte: a autora.

DIAGRAMA DE COMUNICAÇÃO

Esse diagrama substituiu o diagrama de colaboração das versões anteriores da UML®. Um diagrama de comunicação mostra as interações entre os objetos relacionados com linhas de vida e mensagens transmitidas entre linhas de vida.

O diagrama de comunicação é um diagrama de interação fornecendo uma visualização alternativa das mesmas informações oferecidas pelo diagrama de sequência, sendo que, no de sequência, o foco é ordenar as mensagens pelo tempo, aqui, no de comunicação, o foco é a estrutura de transmissão das mensagens entre os objetos. Esses diagramas ilustram o fluxo de mensagens entre objetos e os relacionamentos implícitos entre eles (IBM Knowledge Center).

Os diagramas de comunicação identificam os seguintes aspectos de uma interação:

- Objetos participantes
- Interfaces exigidas pelos objetos participantes
- Alterações estruturais requeridas por uma interação
- Dados transmitidos entre os objetos em uma interação



REFLITA

“Estou fazendo um sistema operacional gratuito (apenas um hobby, não será grande e profissional como GNU) para 386/486 AT.”

Fonte: Linus Torvalds.

O diagrama de comunicação assemelha-se muito com o diagrama de fluxo de dados da análise estruturada, embora o DFD e informações relacionadas não sejam parte formal da UML, eles podem ser usados para complementar os diagramas UML e fornecer visão adicional dos requisitos e fluxo do sistema. O DFD tem uma visão entrada-processo-saída de um sistema, ou seja, objetos de dados entram no software, são transformados por elementos de processamento, e os objetos de dados resultantes saem do software (PRESSMANN, 2010, p. 159).

Os elementos que compõem a notação do diagrama de comunicação são os seguintes:

- **:FormularioConta** Representa o objeto (classe ou método envolvido).
- Uma linha sólida, representando um relacionamento entre os objetos e uma troca de mensagem.
- Uma seta indica a direção da mensagem.
- Números cardinais, representando a sequência das mensagens.

A Figura 50, Diagrama de Comunicação Conta Bancária, ilustra um diagrama de comunicação.

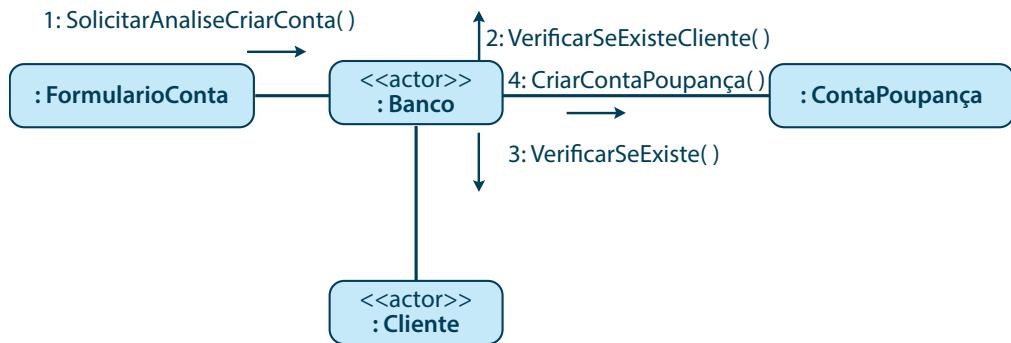


Figura 50: Diagrama de Comunicação Conta Bancária

Fonte: a autora.

Concluindo esta unidade, vimos que o conjunto de diagramas propostos pela UML® para representar a ideia do software, prover sua modelagem, são nove: diagrama de caso de uso, de classes, de objeto, de estado, de sequência, de colaboração, de atividade, de componente e de execução. Sabemos que todos os sistemas possuem uma estrutura estática e um comportamento dinâmico, então a organização lógica para a utilização dos diagramas é:

- Modelagem Estática: Diagrama de Classes; Diagrama de Objetos.
- Modelagem Dinâmica: Diagramas de Estado; Diagrama de Sequência; Diagrama de Comunicação; Diagrama de Atividade.
- Modelagem Funcional: Diagrama de caso de uso; Diagramas de Componente; Diagrama de Execução.

CONSIDERAÇÕES FINAIS

Estudamos nesta unidade os fundamentos da orientação a objetos e a linguagem de modelagem UML®. Observamos que, em um cenário orientado a objetos, a abstração é um dos aspectos mais importantes, pois permite estudar uma entidade complexa abstraindo dela somente os detalhes que interessam em um determinado momento, e, quando usada de forma estratégica, garante um código mais inteligente, mais coeso e com baixo nível de acoplamento, características que possibilitam, entre outras coisas, uma manutenção facilitada e o reuso de pacotes. Na orientação a objetos, os objetos se relacionam e fazem referência a outros objetos o tempo todo, princípio conhecido como dependência.

Na sequência dos estudos, fomos apresentados à UML®, uma linguagem visual de modelagem de dados orientada a objetos, ela é independente de linguagem de programação e também é independente de processo de desenvolvimento. Ficou claro em nossos estudos que a UML® não é uma linguagem de programação.

Um processo de desenvolvimento de software que assume a UML® como linguagem de modelagem deve promover a criação de vários documentos, tanto gráficos quanto textuais. A especificação UML® define dois principais tipos de diagramas: diagramas de estrutura e diagramas de comportamento.

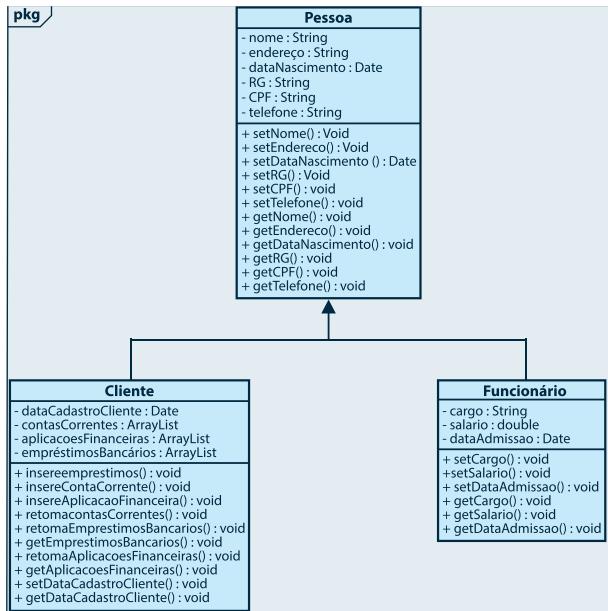
Diagramas de estrutura mostram a estrutura estática do sistema e suas partes em diferentes níveis de abstração e de implementação. Registram como esses elementos estão relacionados uns aos outros e como se afetam no contexto em que estão inseridos. Os elementos em um diagrama de estrutura representam os conceitos significativos de um sistema e podem incluir abstrações, mundo real e conceitos de implementação.

Diagramas de comportamento mostram o comportamento dinâmico dos objetos em um sistema, que podem ser descritos como uma série de mudanças no sistema ao longo do tempo.

ATIVIDADES



1. Dê um exemplo da aplicação dos conceitos de generalização e especialização.
2. Qual é o pilar da orientação a objetos, automaticamente da modelagem orientada a objeto, que se preocupa em decompor um sistema complexo em suas partes fundamentais a fim de descrevê-las em uma linguagem simples e precisa?
 - a) Modularidade
 - b) Encapsulamento
 - c) Abstração
 - d) Herança
3. Dentro do paradigma orientação a objetos existe um mecanismo utilizado para impedir o acesso direto ao estado de um objeto. Assinale a alternativa que apresenta o nome desse mecanismo.
 - a) Mensagem
 - b) Herança
 - c) Polimorfismo
 - d) Encapsulamento
4. Observe a figura abaixo e identifique a alternativa correta a respeito do diagrama de classes.
 - a) Representa um diagrama de colaboração e representa a interação entre as classes Cliente, Funcionário e Pessoa.
 - b) A classe Cliente herda os atributos nome e CPF da classe Pessoa.
 - c) A classe Pessoa herda os atributos cargo, salário e dataAdmissão da classe Funcionário.
 - d) A classe Cliente herda os atributos +getNome() e +SetNome() de Pessoa.



ATIVIDADES



5. Assinale “F” para falso ou “V” para verdadeiro e marque a alternativa corrente:

- () A UML pode ser aplicada somente para modelar projetos relacionados ao desenvolvimento de software.
- () A UML é uma linguagem para especificação, documentação e visualização de softwares orientados a objetos.
- () Ao se modelar um sistema utilizando a UML, segundo normas do *Object Management Group* – OMG – seu mantenedor, deve-se obrigatoriamente utilizar no mínimo quatro de seus diagramas.
- () A UML é um método, isto é, ela diz o que fazer e como modelar um software.

A sequência está correta em:

- a) F, V, F, F.
- b) V, V, F, F.
- c) F, F, V, F.
- d) F, F, V, V.

6. São diagramas UML, EXCETO:

- a) Diagrama de Estado.
- b) Diagrama de Classe.
- c) Diagrama de Conformidade.
- d) Diagrama de Colaboração.

7. Analise a definição a seguir:

“... tem uma grande similaridade com os fluxogramas, ele é essencialmente um gráfico de fluxos. Enquanto os diagramas de sequência focam o fluxo de um objeto para outro, estes enfatizam o fluxo de uma atividade para outra”.

O diagrama descrito é o diagrama de:

- a) Sequência.
- b) Atividades.
- c) Casos de uso.
- d) Comunicação.



PROJETO DE SOFTWARE USANDO A UML

O desenvolvimento orientado a objetos começou em 1967 com a linguagem Simula-67 e desde então surgiram linguagens orientadas a objetos como Smalltalk e C++ entre outras. Nos anos 80, começaram a surgir metodologias de desenvolvimento orientadas a objetos para tirar vantagens desse paradigma. Entre 1989 e 1994 surgiram quase 50 métodos de desenvolvimento orientados a objetos, fenômeno que foi chamado: a guerra dos métodos.

Entre os mais importantes, estavam: o método de G. Booch, a *Object Modeling Technique* (OMT) de J. Rumbaugh, o método de Shlaer e Mellor e o método *Objectory* de I. Jacobson. I. Jacobson introduziu a modelagem de casos de uso em 1987 e criou o primeiro processo de desenvolvimento de software que utiliza casos de uso, chamado *Objectory*.

Cada método possuía uma notação própria, o que gerou uma infinidade de tipos de diagramas e notações. Isso causava problemas de comunicação, treinamento de pessoal e portabilidade. Um esforço de unificação começou em 1994 quando J. Rumbaugh e, logo após, I. Jacobson juntaram-se à G. Booch, na empresa *Rational Software Corporation*. O primeiro grande resultado desse esforço foi a criação da *Unified Modeling Language* (UML), apresentada, na sua versão 1.0, em 1997.

Outro grande resultado desse esforço de unificação de metodologias foi a criação, pela Rational, de um processo de desenvolvimento que usa a UML em seus modelos, chamado Processo Unificado. O Processo Unificado evoluiu do processo *Rational Objectory*, sendo inicialmente chamado de *Rational Unified Process* (RUP). O Processo Unificado, apesar de não ser um padrão, é amplamente adotado, sendo considerado como um modelo de processo de desenvolvimento de software orientado a objetos.

O ciclo de vida de um sistema consiste de quatro fases, divididas em iterações:

- Concepção (define o escopo do projeto)
- Elaboração (define os requisitos e a arquitetura)
- Construção (desenvolve o sistema)
- Transição (implanta o sistema)

Cada fase é dividida em iterações e cada iteração:

- é planejada;
- realiza uma sequência de atividades (de elicitação de requisitos, análise e projeto, implementação etc.) distintas;
- geralmente resulta em uma versão executável do sistema;
- é avaliada segundo critérios de sucesso previamente definidos.





O Processo Unificado é guiado por casos de uso.

- Os casos de uso não servem apenas para definir os requisitos do sistema.
- Várias atividades do Processo Unificado são guiadas pelos casos de uso:
 - planejamento das iterações;
 - criação e validação do modelo de projeto;
 - planejamento da integração do sistema;
 - definição dos casos de teste.

O Processo Unificado é baseado na arquitetura do sistema.

- Arquitetura é a visão geral do sistema em termos dos seus subsistemas e como estes se relacionam.
- A arquitetura é prototipada e definida logo nas primeiras iterações.
- O desenvolvimento consiste em complementar a arquitetura.
- A arquitetura serve para definir a organização da equipe de desenvolvimento e identificar oportunidades de reuso.

Fluxos de atividades

- atividades;
- passos;
- entradas e saídas;
- guias (de ferramentas ou não), *templates*;
- responsáveis (papel e perfil, não pessoa);
- artefatos.

Fonte: Pimentel (2015, online).

MATERIAL COMPLEMENTAR



LIVRO

Linguagens de Programação: Princípios e Paradigmas

Allen Tucker e Robert Noonan

Editora: AMGH

Sinopse: Este texto enfatiza um tratamento prático e expandido das questões essenciais do projeto de linguagem de programação, oferecendo a professores e alunos uma mistura de experiências fundamentadas em explicações e implementações. Atualizado, cada capítulo inicia com a apresentação dos principais fundamentos, paradigmas e tópicos das linguagens, provendo tanto uma abordagem ampla quanto profunda dos princípios de projeto de linguagens, permitindo flexibilização na escolha de quais tópicos enfatizar. Inclui amplo tratamento dos quatro maiores paradigmas da programação – programação imperativa, orientada a objetos, funcional e lógica – incorporando algumas das linguagens mais atuais como Perl e Python. Tópicos especiais incluem manipulação de eventos, concorrência e ajuste.



PROJETO DE ARQUITETURA DE SOFTWARE

UNIDADE

IV

Objetivos de Aprendizagem

- Desenvolver uma abordagem sistemática para a elaboração de um projeto de arquitetura de software.
- Conhecer as principais arquiteturas de desenvolvimento de software.
- Escolher a arquitetura que melhor resolva um problema computacional.
- Esboçar a arquitetura de software.
- Utilizar os diagramas UML mais indicados para representar as arquiteturas.

Plano de Estudo

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

- Gêneros e estilos de arquitetura
- Arquitetura de software orientado a objetos
- Arquitetura de software cliente-servidor
- Arquitetura orientada a serviço
- Arquitetura de software concorrente e em tempo real

INTRODUÇÃO

Olá, caro(a) aluno(a), nesta quarta unidade do livro de Modelagem de Software, você será apresentado(a) aos conceitos de arquitetura de software. Abordaremos múltiplas visualizações da arquitetura de software e uma visão geral dos padrões. Forneceremos e discutiremos a importância da arquitetura e do uso de padrões em projetos de software.

Você aprenderá como as arquiteturas de software são projetadas considerando padrões relevantes como as arquiteturas de software orientada a objetos, cliente-servidor, orientada a serviços e de tempo real.

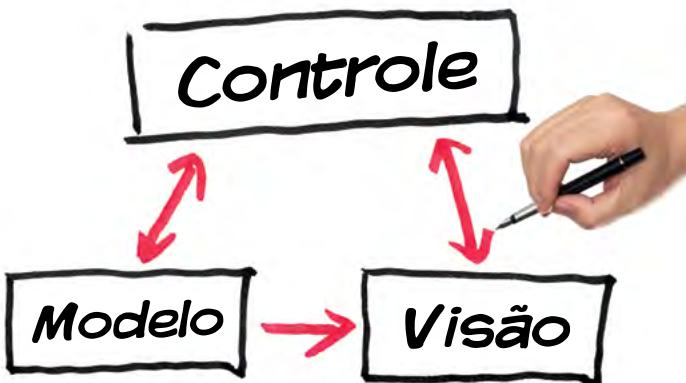
Ao descrever a arquitetura de software de um projeto orientado a objetos, você relembará os conceitos de encapsulamento, abstração de dados, classes e herança. O projeto detalhado das classes também considera as classes de máquina de estado, classes de interface gráfica do usuário e classes da lógica de negócios.

Um projeto de clientes e servidores inclui uma discussão de padrões estruturais e comportamentais, serviços sequenciais, concorrentes e mapeamento de um modelo estático para um banco de dados relacional.

A arquitetura de software orientada a serviços refere-se à reutilização de componentes de software, que podem ser acessados pela Internet e disponibilizados para vários clientes, como *web services*, padrões de registro e padrões de transação.

No caso de sistemas concorrentes e de tempo real, veremos um padrão de arquitetura utilizado em projetos de sistemas embutidos, além do processo de análise de *timing* e uma discussão sobre os sistemas operacionais de tempo real.

Concluiremos nossos estudos sobre modelagem de software na próxima unidade, quando teremos a oportunidade de modelar e construir a arquitetura de um software orientado a objetos. Então vamos lá! Aproveite a leitura.



GÊNEROS E ESTILOS DE ARQUITETURA

A modelagem de software, como você já estudou, permite a visualização do software sob quatro diferentes perspectivas, que compõem o modelo de contexto, o modelo de interação, o modelo estrutural e o modelo comportamental. Todos esses modelos podem ser apresentados com o apoio dos diferentes diagramas da UML®.

Nesse momento, a sua preocupação na condição de engenheiro de software é compreender como o sistema deve ser organizado e como deve ser a estrutura geral desse software. Para isso, você aprenderá a criar representações coerentes e bem planejadas das camadas de dados e da arquitetura do modelo de projeto.

O desafio é desenvolver uma abordagem sistemática para a elaboração de um projeto da arquitetura do software. Pois, segundo Sommerville (2011), o projeto de arquitetura é responsável por descrever como o sistema está organizado em um conjunto de componentes de comunicação. É por intermédio do projeto de arquitetura de software que o software será construído.

ARQUITETURA DE SOFTWARE

Dentro das perspectivas do software, você pode considerar que a arquitetura de software é uma perspectiva estrutural. No entanto, para entender completamente a arquitetura de software é necessário estudá-la, também, sob as perspectivas

estática e dinâmica. Considere nessa abordagem abordar a funcionalidade e a qualidade da funcionalidade fornecida.

Um projeto de arquitetura de software considera o projeto de dados e o projeto arquitetural, como ilustrado na Figura 51. O projeto de dados/classes será transformado no diagrama de classes e nas estruturas de dados necessárias para implementação do software. Enquanto que o projeto arquitetural representa a organização da solução técnica utilizada no software.

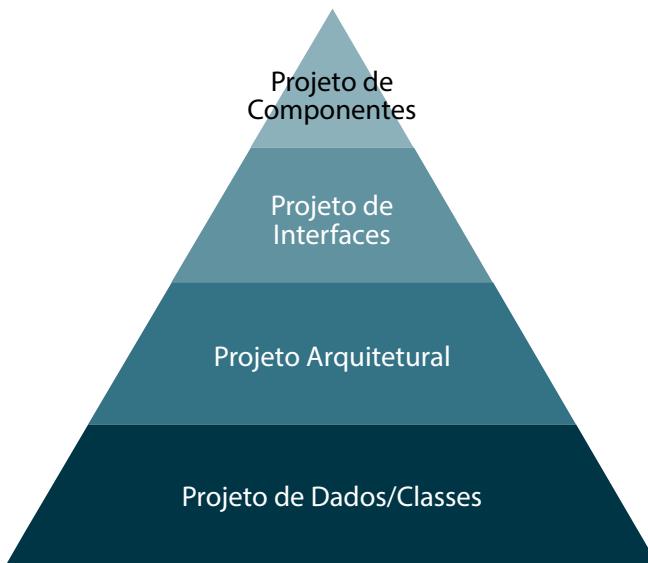


Figura 51 - Modelo de projeto / Fonte: adaptada de Pressman (2011, p. 208).

SAIBA MAIS



Vários autores pesquisam sobre a melhor forma de descrever a arquitetura de software. De modo geral, uns defendem o uso da UML enquanto outros defendem notações informais, por serem mais rápidas de se desenhar. Na disciplina de Modelagem de Software vamos eleger a UML para documentar todas as arquiteturas de software que estudaremos.

Na condição de um arquiteto de software, você deve considerar os aspectos funcionais e não-funcionais incluídos na etapa de levantamento de requisitos e, com base neles, decidir pelo projeto de arquitetura mais adequado. Mais adiante ao estudarmos diferentes arquiteturas e veremos que, apesar de cada sistema de software ser único, os sistemas de mesmo domínio tendem a utilizar arquiteturas similares.

Antes de continuarmos para o próximo tópico é bom ter em mente as quatro visões que Sommerville (2011) aponta como fundamentais de arquitetura de software:

- 1. Lógica:** compõe os requisitos funcionais do sistema.

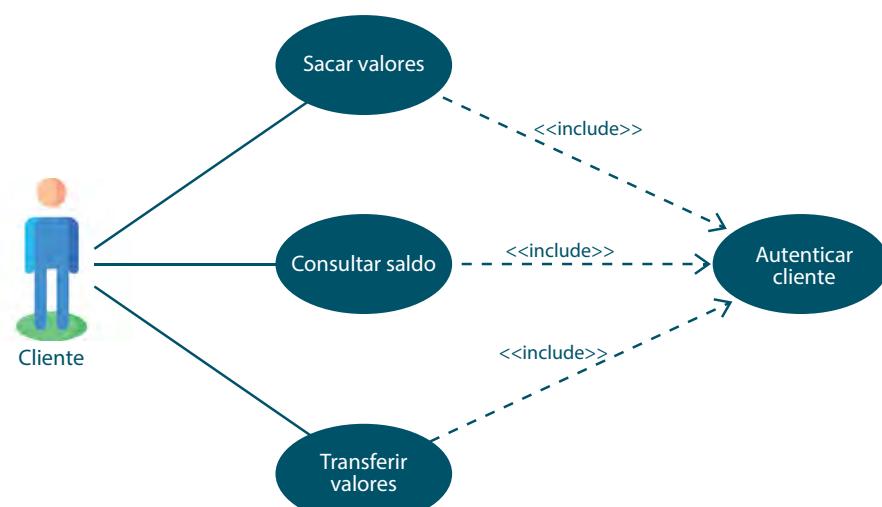


Figura 52 - Modelo de casos de uso para o problema de domínio de um sistema bancário
Fonte: adaptada de Gomaa (2011, p. 373).

- 2. Processo:** as características não funcionais do sistema, como desempenho e disponibilidade.

3. Desenvolvimento: a decomposição do software para compreensão do desenvolvedor.

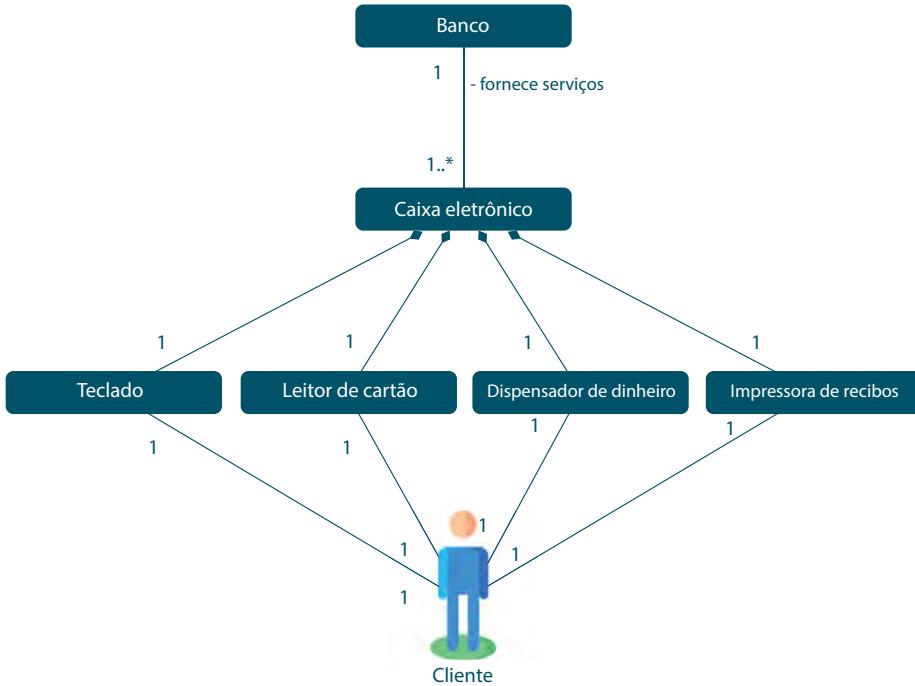


Figura 53 - Modelo estático conceitual para o problema de domínio de um sistema bancário
Fonte: adaptada de Gomaa (2011, p. 377).

4. **Física:** mostra o que é necessário para a implantação do sistema.

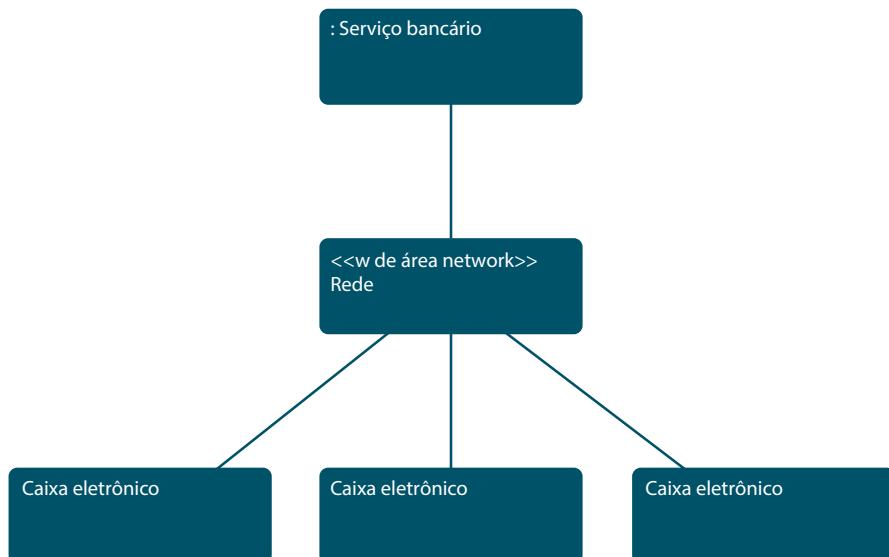


Figura 54 - Diagrama de implantação para um sistema bancário / Fonte: adaptada de Gomaa (2011, p. 417).

A partir da descrição do problema, os casos de uso devem ser criados, como ilustra a Figura 52, e descritos no modelo de caso de uso. A partir do domínio do problema e do contexto do sistema você deve continuar para a elaboração da modelagem estática, como mostra a Figura 53, por intermédio do diagrama de classes. Por fim, você deve considerar o estilo da arquitetura e representá-la com o auxílio de um diagrama de implantação, como apresentado na Figura 54.

Agora vamos tratar dos diferentes tipos de software que exigem diferentes gêneros arquitetônicos.

GÊNEROS DE ARQUITETURA

De modo geral, tenha em mente que os fundamentos de um projeto de arquitetura de software podem se aplicar a qualquer tipo de arquitetura. No entanto, cada estilo de arquitetura pode ser descrito de forma padronizada. Por isso, é

importante justificar o uso de determinada arquitetura dentro de um gênero de arquitetura de software.

Podemos encontrar uma variedade de software para diversos gêneros específicos de arquitetura. Entre eles você pode ter sistemas de gestão, voltados para a área comercial, financeira e governamental entre outras; sistemas científicos, indicados para pesquisa médica e aplicações científicas em geral; sistemas de simulação, que auxiliam os processos industriais, o controle de veículos autônomos, por exemplo, com o apoio de inteligência artificial, sensores e atuadores; sem contar com uma infinidade de outros gêneros que estão presentes nas diversas áreas do conhecimento, como a geração de conteúdo, a indústria do entretenimento e serviços.

Durante a modelagem de software, as decisões de projeto são feitas em relação às características da arquitetura de software. Vamos estudar um pouco a respeito das diferentes arquiteturas de software que você verá neste capítulo.

PADRÕES DE ARQUITETURA

A proposta de um padrão de arquitetura é apresentar, compartilhar e reusar o conhecimento sobre sistemas de software. Nesta seção vou te apresentar três padrões de arquitetura de software utilizados no desenvolvimento de sistemas.

ARQUITETURA EM CAMADAS

A estrutura de uma arquitetura de software definida em camadas separa os elementos de um sistema possibilitando independência entre eles. Segundo Sommerville (2011) um sistema organizado em camadas separa as funcionalidades e cada camada depende apenas dos recursos e serviços oferecidos pela camada imediatamente abaixo dela.

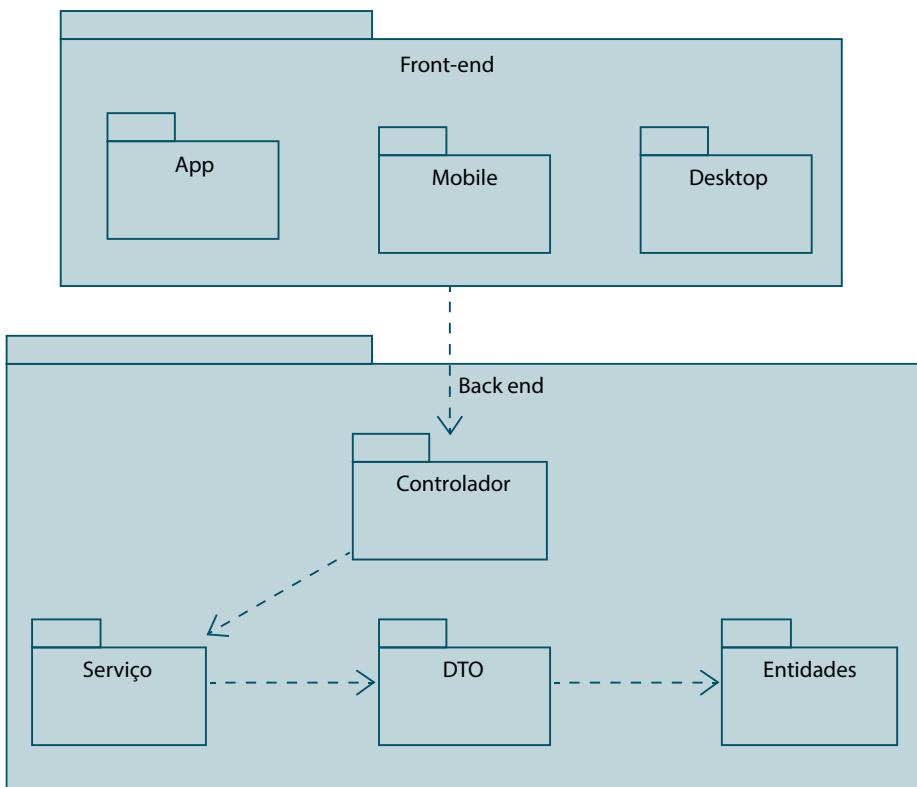


Figura 55 - Arquitetura de uma aplicação usando o padrão MVC / Fonte: o autor.

O padrão MVC (*Model-View-Controller*, modelo-visão-controlador), segundo Sommerville (2011), separa o sistema em três componentes lógicos que interagem entre si. Por intermédio da Figura 55 você consegue perceber que a camada mais externa proporciona operações de interface com o usuário, enquanto que as camadas mais internas oferecem serviços para a aplicação.

Quando você cria uma aplicação Desktop, a camada de apresentação é desenvolvida por intermédio de GUIs (*Graphical User Interface*, Interface Gráfica do Usuário). Já em projetos voltados para a Web, a camada de apresentação é uma página que é carregada em um *browser*. Por último, quando você desenvolve um projeto Mobile, a camada de apresentação utiliza componentes de visualização que podem ou não serem nativos da linguagem.

A camada de controle processa e responde a eventos, chamado de *endpoint* da aplicação. As respostas das requisições do usuário geralmente são retornadas

em um formato específico, por intermédio de uma API REST (*Representational State Transfer*, Transferência de Representação de Estado).

Por fim, a camada de modelo representa o domínio específico da informação na qual o software opera. Nessa camada são definidas as regras de acesso e manipulação dos dados, a base de dados para armazenamento e a persistência dos dados.

ARQUITETURA CENTRALIZADA EM DADOS

Neste modelo de arquitetura, diversos componentes utilizam um repositório de dados. Tais componentes são capazes de manipular os dados contidos no repositório.

Tenha em mente que o termo componente refere-se a sistemas modulares. Gomaa (2011) define um componente como um objeto autocontido com uma interface bem definida que pode ser usada em aplicativos diferentes daqueles para os quais foi originalmente projetado. Um componente vai além do conceito de operações que um objeto fornece, pois, ao ser integrado em um sistema é importante compreender as operações que o componente requer e fornece.



Figura 56 - Uma arquitetura de repositório para um IDE / Fonte: adaptada de Sommerville (2011, p. 112).

A Figura 56 apresenta uma série de ferramentas (componentes) que compartilham dados com o repositório do projeto, promovendo integração dos componentes existentes.

ARQUITETURA PIPE-FILTER

Esta arquitetura está associada a um software que utiliza-se de outros componentes ou sistemas para se obter um resultado. A saída resultante do processamento de um componente serve de entrada para o processamento de outro componente, e assim sucessivamente até se obter o resultado esperado.

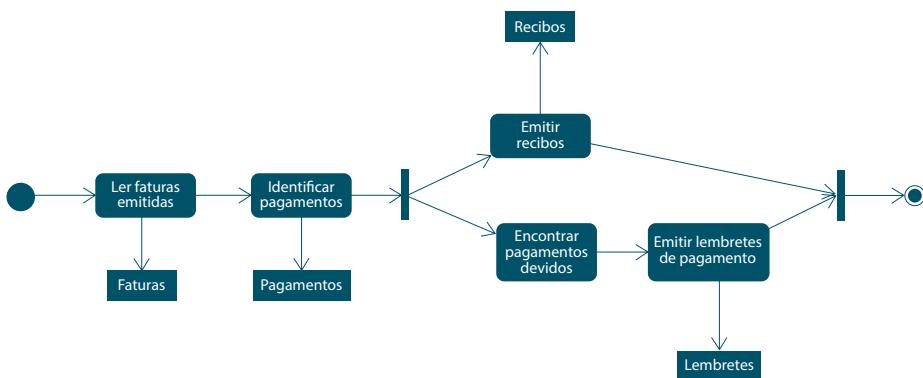


Figura 57 - Um exemplo de arquitetura pipe-filter / Fonte: adaptada de Sommerville (2011, p. 115).

O diagrama de atividades ilustrado na Figura 57 representa bem essa definição, pois cada atividade resulta em um objeto de saída, e esse objeto é a entrada para a próxima atividade, e assim sucessivamente. Apesar de ser possível alterar o fluxo de execução do *pipeline* a arquitetura culmina em uma única saída.



Esta indicação de texto traz alguns conceitos e abordagens comuns ao ambiente do desenvolvimento de software, como DevOps, que, segundo Donovan Brown, refere-se a união de pessoas, processos e produtos para permitir a entrega contínua de valor aos usuários finais.

PROJETO DA ARQUITETURA

Ao desenvolver o projeto de arquitetura do software, você deve considerar o contexto, as interfaces de interação, os elementos comportamentais e a estrutura do sistema. Segundo Pressman (2011) esse é um processo iterativo e incremental até que a arquitetura completa tenha sido alcançada.

Quando você desenvolve um sistema deve pensar nos requisitos funcionais dos sistemas, mas também nos requisitos não-funcionais, que inclui o desempenho, a proteção, a disponibilidade e a escalabilidade. A arquitetura de software influencia diretamente nesses requisitos não-funcionais.

A escolha da arquitetura a ser utilizada deve se pautar pelo tipo de aplicação e as diferentes perspectivas.

A partir de agora vamos nos dedicar no estudo de diferentes arquiteturas - orientada a objetos, cliente-servidor, orientada a serviços, concorrente e em tempo real -, todas elas utilizadas como um meio de reusar componentes em larga escala.



QR CODE



A IEEE Computer Society elaborou um documento chamado Recommended Practice for Architectural Description for Software-Intensive Systems com o objetivo de definir um vocabulário e fornecer diretrizes para a descrição de arquiteturas de software. O documento está disponível no endereço.



ARQUITETURA DE SOFTWARE ORIENTADO A OBJETOS

Um projeto de software orientado a objetos utiliza os conceitos de ocultação de informações, classes e herança, os objetos são instanciados a partir de classes e acessados por meio de operações, também chamadas de métodos.

Uma classe encapsula detalhes de uma estrutura de dados ou máquina de estado, dentre outros tipos de informações determinadas durante a fase de análise.

Nesta seção você estudará a arquitetura e os padrões utilizados em um projeto orientado a objetos, como escrever os diferentes tipos de classes para ocultação de informações, incluir ao projeto hierarquia de classes, classes abstratas e subclasses, especificar as interfaces de classe, detalhar as classes que ocultam informações, o polimorfismo e a ligação dinâmica. Tudo isso servirá de base para implementação de classes em uma linguagem de programação orientada a objetos.

TIPOS DE CLASSE

O conceito de ocultação de informações é fundamental quando se deseja que uma classe encapsule algumas informações, como uma estrutura de dados, que está oculta do resto do sistema. Outro conceito importante é separar a interface

da implementação, de forma que a interface forme um contrato entre o provedor da interface e o usuário da interface.

É importante que você tenha em mente que os conceitos de orientação a objetos também são aplicados e estendidos a outros projetos de arquiteturas de software. Para comunicação entre objetos, Pressman (2011) indica a arquitetura de chamadas e retorno como padrão de comunicação em uma arquitetura sequencial.

Na modelagem do projeto, as classes são categorizadas por estereótipo. A partir do modelo de domínio do problema as classes são categorizadas como entidade (*entity*), limite (*boundary*), controle (*control*) e lógica da aplicação.

CLASSE DE ENTIDADES

As classes de entidades são determinadas durante a fase de análise e encapsulam os dados representados com o estereótipo «entity». Objetos de entidade precisam ser armazenados em um banco de dados. Sendo assim, a classe de entidade fornece uma interface para o banco de dados.

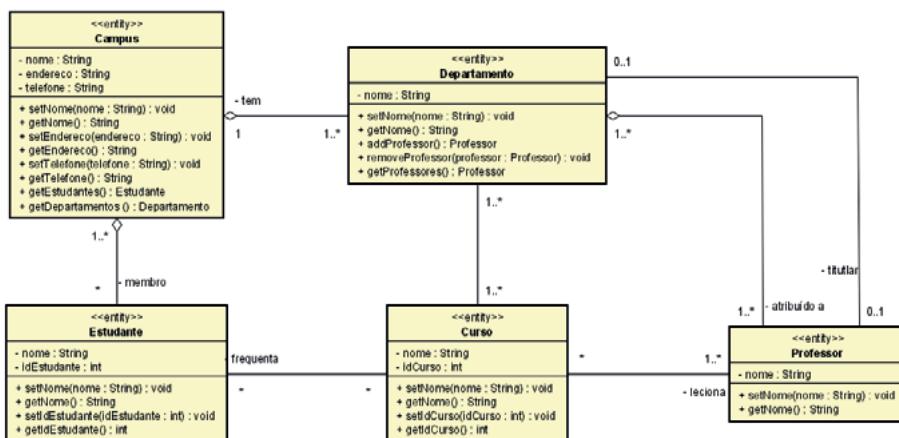
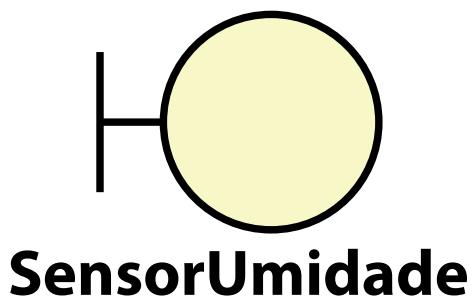


Figura 58 - Modelagem de um esquema de banco de dados
Fonte: adaptada de Booch, Rumbaugh & Jacobson (2005, p. 103).

A Figura 58 ilustra um conjunto de classes de entidades relativas a um sistema de informação para uma universidade. A figura revela os detalhes dessas classes em um nível suficiente para construir um banco de dados físico.

CLASSE LIMITES



É fundamental que um sistema interaja com o ambiente externo. As chamadas classes limite possibilitem, por exemplo, a interação gráfica do usuário, que faz interface com usuários humanos e apresenta informações a eles.

Figura 59 - Elemento estereotipado como um ícone
Fonte: adaptada de Booch, Rumbaugh & Jacobson (2005, p. 77).

Um sensor de umidade ilustrado pela Figura 59 permite ao sistema comunicar-se com o ambiente externo. Essa classe limite é uma classe ativa, ou seja, representa um processo ou thread simultâneo.

Considere que você tenha que desenvolver um sistema computacional para captação de água da chuva e irrigação de uma horta, no qual você tenha vários sensores para mensurar umidade do solo, nível da cisterna e chuva, para abrir ou fechar as comportas e registro da cisterna por intermédio de um motor acionado e iniciar a irrigação automaticamente.

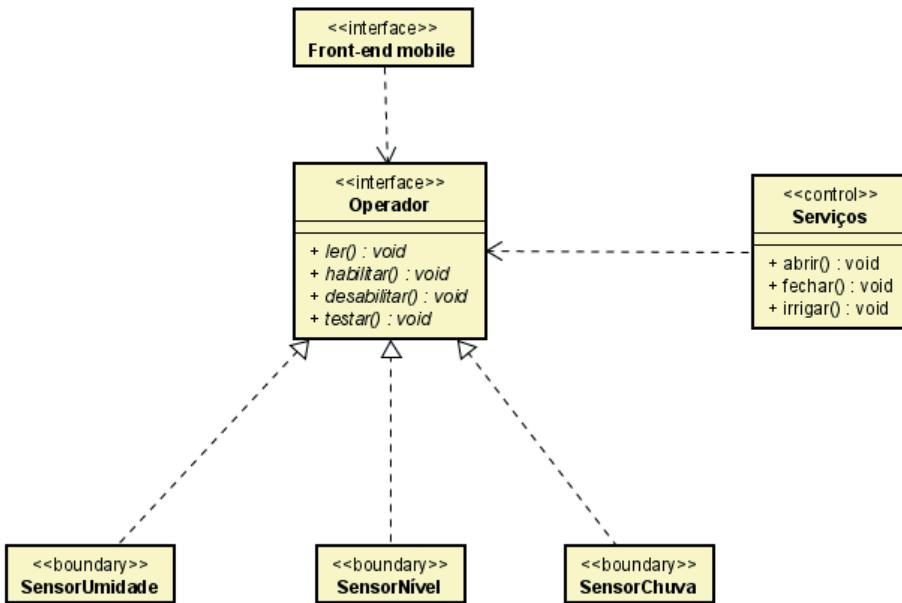


Figura 60 - Sistema automatizado de irrigação / Fonte: o autor.

A Figura 60 ilustra a arquitetura do sistema de irrigação, com as classes limite (*boundary*) representadas como um estereótipo e não como um ícone.

CLASSE DE CONTROLE

As classes de controle fornecem a coordenação geral para uma coleção de objetos.

Considerando um sistema hipotético desenhado no padrão MVC (*Model-View-Controller*), por exemplo, com todas as classes funcionando juntas, é necessário a distribuição de responsabilidades entre as classes. As classes de controle têm a responsabilidade de sincronizar mudanças no modelo e suas visualizações.

CLASSE DE LÓGICA DA APLICAÇÃO

Estas classes encapsulam a lógica e os algoritmos específicos da aplicação. Você pode encontrar outras nomenclaturas para categorizá-las, como classes de lógica

de negócios, classes de serviço ou classes de algoritmo, dependendo do padrão de arquitetura adotado.

CLASSE DE INTERAÇÃO GRÁFICA DO USUÁRIO

Todo sistema precisa ter alguma forma de interação para entrada e saída de dados. Esta interação acontece por intermédio da interface do usuário. Em uma determinada aplicação, a interface do usuário pode ser uma interface de linha de comando simples ou uma interface gráfica sofisticada. Uma classe de interação gráfica do usuário, GUI (*Graphical User Interaction*) oculta de outras classes os detalhes da interface para o usuário.

Uma interface de linha de comando normalmente é tratada por uma classe de interação do usuário. No entanto, o projeto de uma interface gráfica de usuário normalmente requer o *design* de várias classes de GUI.

Classes de GUI com elementos como janelas, menus, botões e caixas de diálogo, são normalmente armazenadas em uma biblioteca de componentes de interface do usuário.

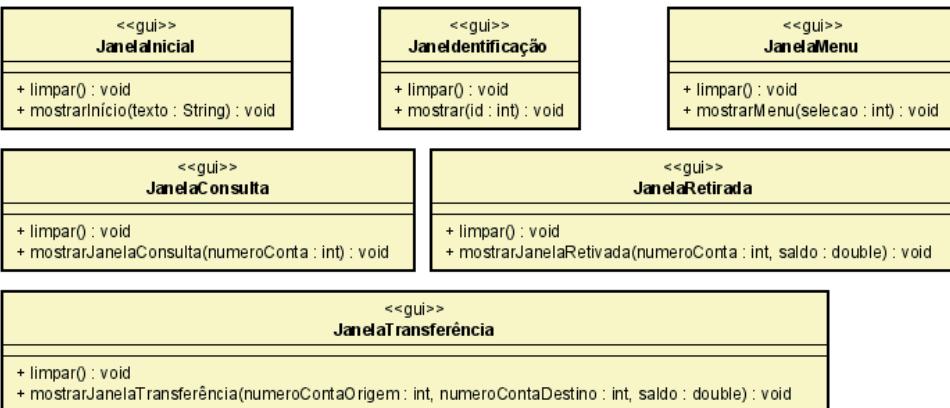


Figura 61 - Exemplo de classes de interação gráfica do usuário (GUI) de um sistema bancário
Fonte: adaptada de Gomaa (2011, p. 238).

A Figura 61 é um exemplo de classes de interação com o usuário de uma aplicação financeira. As classes utilizadas na ilustração do projeto da GUI são projetadas para cada uma das janelas usadas para interagir com o cliente do sistema.

CLASSE DA LÓGICA DO NEGÓCIO

A lógica do negócio define a tomada de decisão, que representa uma aplicação lógica específica para processar uma solicitação do cliente. O objetivo das classes contendo a lógica do negócio é encapsular as regras de negócios que podem mudar independentemente umas das outras. Normalmente, um objeto da lógica do negócio acessa vários objetos da entidade durante sua execução.

Considere, para isso, a classe de controle de transações de saque em um sistema financeiro, que possui operações para inicializar, sacar, confirmar e abortar.

A operação inicializar é chamada no instante da inicialização; a operação sacar é chamada para retirar fundos de uma conta do cliente; a operação confirmar é chamada para efetivar que a transação de saque foi concluída com sucesso; e a operação abortar é chamada se a transação não foi efetivada com sucesso.

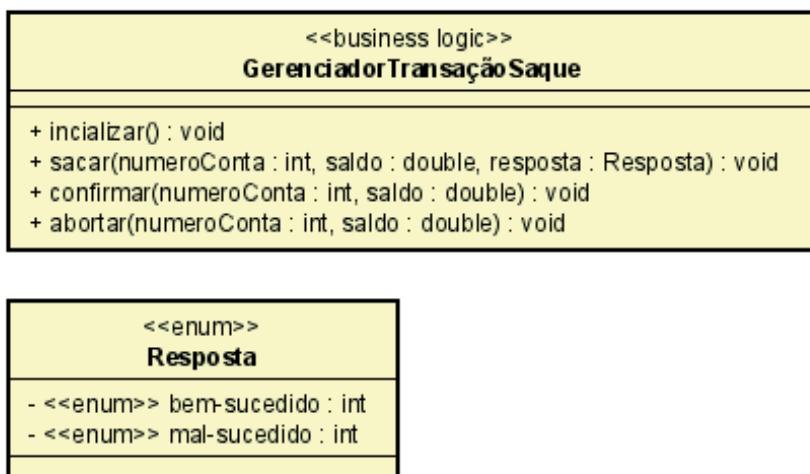


Figura 62 - Diagrama de classe do modelo de projeto / Fonte: adaptada de Gomaa (2011, p. 240).

As operações ilustradas pela Figura 62 são obtidas a partir da análise do serviço bancário e determinadas pelo diagrama de classes.



QR CODE

A Figura 62 apresenta uma classe do tipo enumeration ou, simplesmente, enum. O objetivo dessa classe é apresentar uma lista de valores fixos, onde o primeiro valor tem associado ao valor 0 e assim por diante até o último valor armazenado.

INTERFACES DE CLASSE

A interface de classe consiste nas operações fornecidas por cada classe. Cada operação pode ter ou não parâmetros de entrada e um valor de retorno. As operações de uma classe podem ser determinadas a partir do modelo estático ou do modelo dinâmico.

Vamos ver a seguir como usar os diagramas do modelo de interação entre os objetos participantes para auxiliar a encontrar as operações de cada classe.



SAIBA MAIS

O modelo estático tem a função de mostrar as operações de cada classe, certo? No entanto, dependendo da situação problema é mais fácil determinar as operações por intermédio da troca de mensagens entre os objetos, particularmente utilizando os diagramas de comunicação ou diagramas de sequência. O modelo dinâmico é responsável por mostrar a interação entre os objetos realizando as operações de chamada no objeto de destino que recebe a mensagem.

MODELO DE INTERAÇÃO

Considere uma situação problema em que um funcionário de uma loja precisa cadastrar um novo produto no banco de dados. Basicamente você precisará de uma classe limite para cada ator participante. Você também não pode se esquecer de uma classe de controle e, por último, de uma classe de entidade.

No caso da funcionalidade que precisa ser implementada, os atores são o funcionário e o banco de dados. As classes de limite serão representadas pela interface com o funcionário e o repositório do banco de dados. A classe de controle representa a ação de cadastrar o novo produto. E a classe de entidade é o produto.

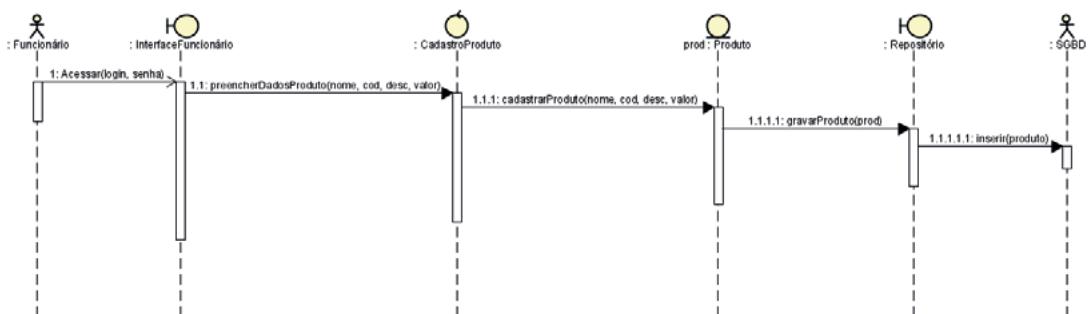


Figura 63 - Diagrama de sequência para cadastrar produtos / Fonte: o autor.

Cada elemento da *lifeline* ilustrado na Figura 63 foi representado por um ícone específico, conforme descrito anteriormente. Na arquitetura do sistema, a partir desse modelo de interação, cada elemento constitui uma classe do sistema.

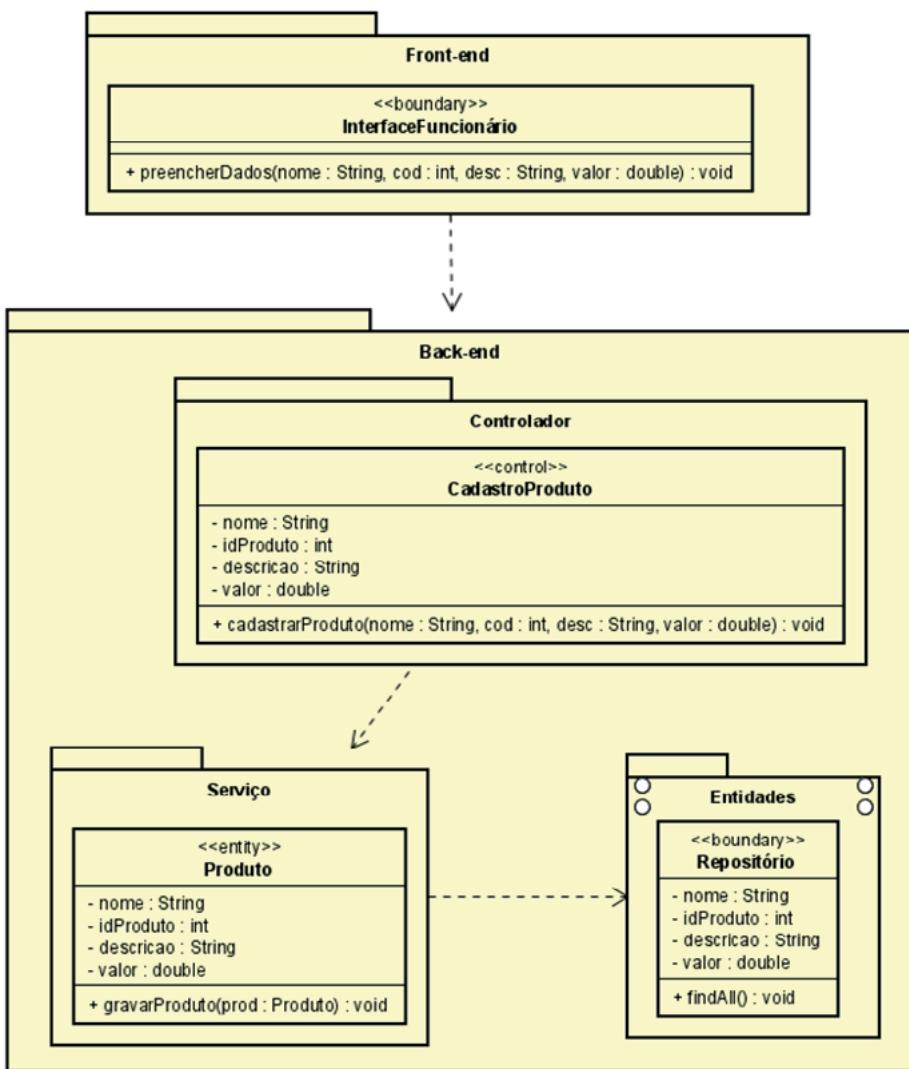


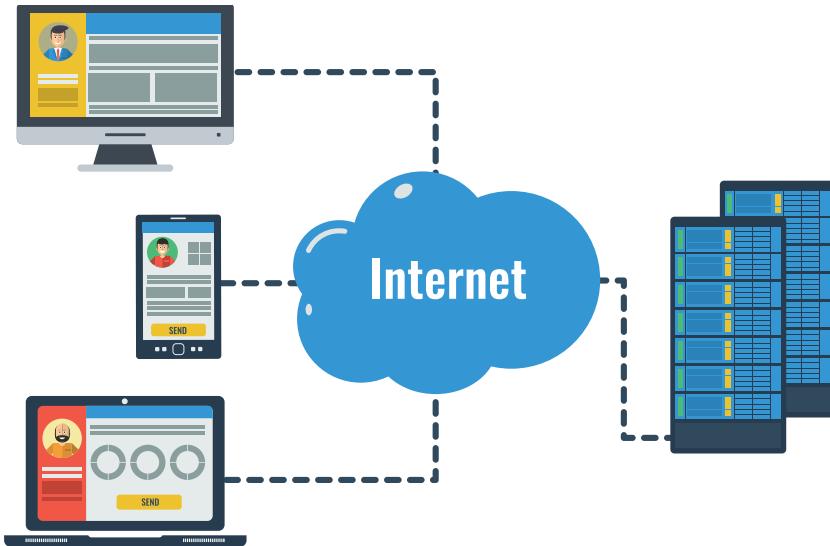
Figura 64 - Arquitetura do sistema para cadastrar produtos / Fonte: o autor.

A Figura 64 ilustra a arquitetura em camadas relacionada a funcionalidade cadastrar produtos. Você consegue observar que todas as mensagens trocadas entre os objetos na Figura 13 foram representadas como operações nas diferentes visões estruturais do modelo estático.

REFLITA



Segundo Gomaa (2011), o diagrama de classes do modelo estático, particularmente, possibilita determinar as operações padrão de uma classe - criar, recuperar, atualizar e excluir - para as classes de entidade. No entanto, muitas vezes é possível adaptar as operações às necessidades mais específicas da classe de abstração de dados específica, definindo os serviços fornecidos pela classe.



ARQUITETURA DE SOFTWARE CLIENTE-SERVIDOR

Nesta seção vamos estudar a arquitetura de um software cliente-servidor. Este tipo de sistema se caracteriza por ser constituído de vários computadores, no qual todos os componentes do sistema executam em um único computador, e a comunicação entre eles acontece por intermédio de uma rede.

A partir da década de 1990 os grandes sistemas computacionais migraram dos sistemas legados de *mainframe* para os sistemas cliente-servidor. A capacidade de rede deve ser suficiente para manter o desempenho de um sistema cliente-servidor.

Sommerville (2011) aponta algumas questões importantes, inerentes a um projeto de sistemas cliente-servidor, como:

1. **Transparência:** o usuário deve ter ciência da dependência da rede de computadores, no geral, apesar de ter uma visão do sistema como sendo único.
2. **Abertura:** o sistema deve permitir a integração com outros sistemas, utilizando padrões de *web services* ou protocolos RESTful.
3. **Escalabilidade:** tal característica do sistema envolve a possibilidade de incluir mais recursos, está distribuído geograficamente sem comprometer o desempenho e ser gerenciável independentemente da localização.
4. **Proteção:** o sistema deve conter ataques externos que possam prejudicar o sistema com relação a confidencialidade dos dados (intercepção), interrupção dos serviços, alteração dos dados e a invasão do sistema.
5. **Qualidade de serviço:** reflete a confiabilidade dos serviços oferecidos, o tempo de resposta e transferência aceitáveis para uma requisição do cliente.
6. **Gerenciamento de falhas:** o ideal é que o sistema fosse resistente a falhas, no entanto isso é praticamente inevitável, por isso o sistema deve incluir mecanismos de tolerância a falhas, com recuperação automática.

A seguir vamos estudar os padrões de arquitetura utilizados e sistemas cliente-servidor.

PADRÃO DE ARQUITETURA PARA SISTEMAS DISTRIBUÍDOS

Chegou o momento de você ter contato com os padrões de arquitetura de um software cliente-servidor. Vou te apresentar estruturas que variam desde vários clientes com um único servidor até vários clientes com vários servidores e arquitetura cliente-servidor de várias camadas.

MÚLTIPLOS CLIENTES-ÚNICO SERVIDOR

Esta é uma arquitetura de software em que vários clientes podem solicitar um serviço e um único servidor atende às solicitações do cliente. O padrão de arquitetura múltiplos clientes-único servidor pode ser representado por intermédio de um diagrama de implantação.

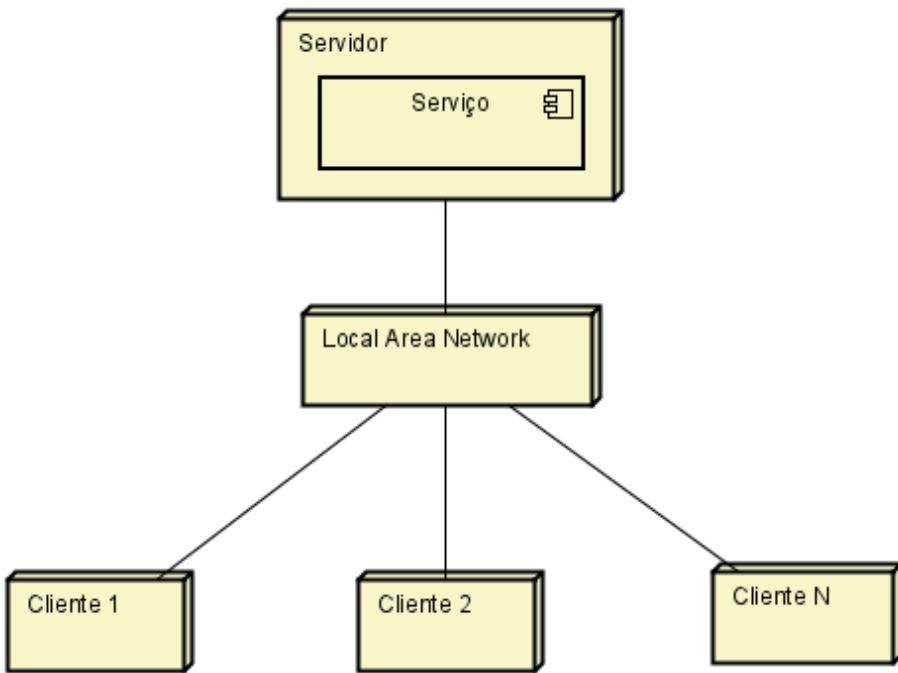


Figura 65 - Arquitetura padrão para um sistema cliente-servidor / Fonte: adaptada de Gomaa (2011, p. 255).

A Figura 65 mostra vários clientes conectados a um serviço executado em um nó de servidor por meio de uma rede local.

MÚLTIPLOS CLIENTES-MÚLTIPLOS SERVIDORES

Você vai se deparar ao longo de sua vida profissional com situações que exigirão sistemas mais complexos, com suporte a vários serviços. Nestes casos o padrão de arquitetura de múltiplos clientes com múltiplos servidores é a melhor solução.

Neste padrão de arquitetura, os clientes solicitam e podem se comunicar com vários serviços e os serviços podem se comunicar entre si. O padrão de múltiplos clientes-múltiplos serviços também pode ser representado por um diagrama de implantação.

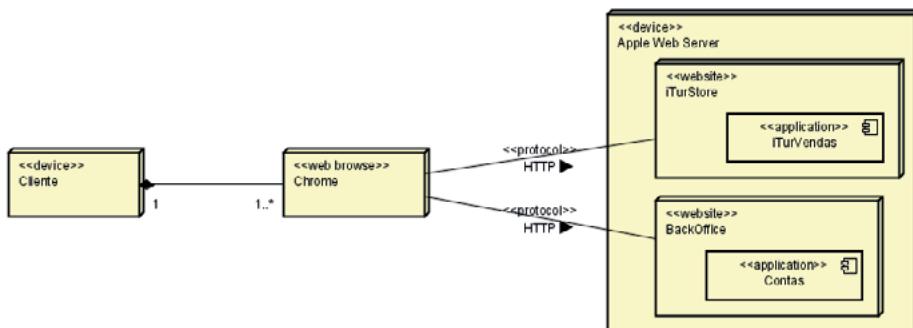


Figura 66 - Arquitetura para um sistema *backoffice* cliente-servidor / Fonte: o autor.

O diagrama de implantação da Figura 66 ilustra que cada serviço reside em um nó de servidor separado e ambos os serviços podem ser chamados pelo mesmo cliente. A comunicação do cliente com os serviços pode ser realizada de forma sequencial ou simultânea.

CLIENTE-SERVIDOR MULTICAMADAS

Como já falamos anteriormente, um dos desafios desse modelo é garantir um gerenciamento de falhas. Para isso vamos supor um exemplo com um平衡ador usado para dividir a carga de rede em vários servidores.

O padrão cliente-servidor de várias camadas tem uma camada intermediária que é um cliente da camada de serviço e também fornece um serviço para seus clientes. É possível ter mais de uma camada intermediária. Quando visto como uma arquitetura em camadas, o cliente é considerado em uma camada mais alta do que o serviço, porque o cliente depende do serviço e o usa.

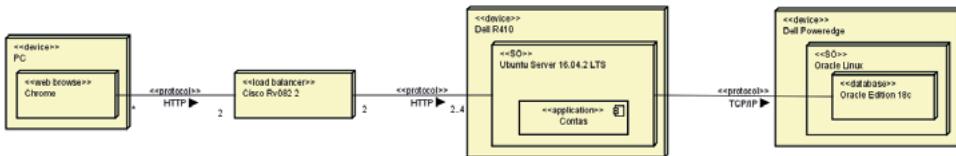


Figura 67 - Arquitetura com múltiplas camadas para um sistema *backoffice* cliente-servidor / Fonte: o autor.

A Figura 67 mostra o balanceador de carga de hardware que combina as funções de平衡amento de carga com a camada de aplicação, compressão do HTTP, desativação do certificado SSL e cache de conteúdo em uma solução.

MIDDLEWARE

Os diagramas nas Figura 66 e Figura 67 mostram um pouco que os componentes de um sistema cliente-servidor podem ser variados, com diferentes de tecnologias de implementação e processamento. O termo middleware se refere a essa camada entre o sistema operacional e as aplicações.

O *middleware* é um software adquirido por desenvolvedores de aplicações que incluem diversos serviços. No entanto, esta disciplina não abordará o contexto tecnológico particular de cada componente de software, que inclui o gerenciamento de comunicação com banco de dados, gerenciadores de transações, conversores de dados e controladores de comunicação.

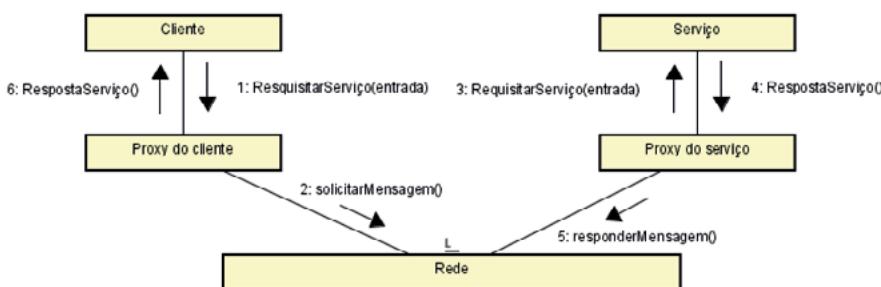


Figura 68 - Método de invocação remota / Fonte: adaptada de Gomaa (2011, p. 262).

A Figura 68 descreve por intermédio de um diagrama de comunicação uma sequência de mensagens a partir do nó cliente para a chamada de um método remoto para um objeto de serviço no nó servidor. A invocação do método local

passa pelo proxy do cliente, que empacota os dados na mensagem e, em seguida, envia a mensagem pela rede. O proxy de serviço no nó remoto recebe a mensagem, desempacota a mensagem e chama o método remoto do objeto de serviço. O proxy de serviço, então, organiza a resposta e a envia pela rede de volta ao proxy do cliente que desempacota a resposta e a devolve ao objeto cliente.



QR CODE

Se o exemplo da Figura 17 aguçou sua curiosidade, que tal dar uma olhada nesse material da documentação do Java que utiliza a tecnologia RMI (Remote Method Invocation) para criar um programa Hello, World! Distribuído em Java?

SOFTWARE COMO UM SERVIÇO

Geralmente, para se ter um sistema no modelo cliente-servidor é necessário instalar no cliente um programa que seja capaz de comunicar com o servidor. Porém, com o uso das tecnologias *web* o *browser* pode ser utilizado como um cliente. Neste caso específico, o software da aplicação funciona como um serviço remoto, que pode ser acessado de qualquer navegador.

Esse tipo de aplicação é conhecido como SaaS (*Software as a Service*, Software como um Serviço), que tem como característica ser hospedado remotamente e acessado por intermédio de uma conexão com a Internet. Outro ponto importante a se destacar é que o software é de propriedade de um fornecedor de software e não por quem o utiliza. O pagamento pelo uso do software varia de acordo com um contrato de prestação de serviço.

Toda a responsabilidade de gestão, manutenção e operação do software é de responsabilidade do provedor do software. Para o cliente, os pontos a serem considerados nesta escolha estão relacionados à falta de controle sobre a evolução do software e a dependência da rede para transmissão de dados.



ARQUITETURA ORIENTADA A SERVIÇO

Hoje em dia é muito comum que uma determinada aplicação que rode na *web* acesse dados de terceiros, que disponibilizam informações e recursos por intermédio de uma interface de *web service*. Podemos tomar como exemplo um sistema comercial que precisa confirmar dados do cartão de crédito ou débito de um cliente para obter autorização da administradora do cartão e efetivar o pagamento.

Sendo assim, nesta seção vamos estudar a arquitetura de software orientada a serviços, SOA (*Service-Oriented Architecture*), que consiste em vários serviços autônomos distribuídos de forma que possam ser executados em nós diferentes com provedores de serviços diferentes. Com isso, você pode desenvolver aplicações com serviços distribuídos, de forma que esses serviços individuais possam ser executados em diferentes plataformas e implementados em diferentes linguagens. Para isso, um protocolo padrão é fornecido para permitir que os serviços se comuniquem entre si e troquem informações.

Você também poderá encontrar empresas de desenvolvimento de software que criam soluções e serviços para serem acessadas por outras empresas de desenvolvimento. Um exemplo disso que estou falando é o acesso e geração a documentos fiscais eletrônicos.

Os protocolos de *web services* cobrem todos os aspectos das SOA. Vamos abordar os padrões para SOAP e REST:

1. **SOAP** (*Simple Object Access Protocol*, Protocolo Simples de Acesso a Objetos): padrão para acessar *web services* por meio de mensagens de requisições e respostas feitas em XML (*Extensible Markup Language*) para criar o arquivo WSDL (*Web Service Description Language*), que é um padrão para a definição de interface de serviço (SOMMERVILLE, 2011) e utiliza a maioria dos protocolos da camada de transporte (HTTP, SMTP, TCP).
2. **REST** (*Representational State Transfer*, Transferência Representacional de Estado): arquitetura criada para ser usada em vários formatos de texto, como CSV, RSS, JSON e YAML. Porém, limita-se aos protocolos HTTP/HTTPS.

Portanto, para se projetar um serviço é necessário:

- Fornecer um conjunto de operações com parâmetros de entrada e saída.
- Separar a interface da implementação.
- Operar independentemente da necessidade de outro serviço.
- Ocultar os detalhes do serviço.
- Fornecer uma interface para as operações.
- Manter o mínimo ou nenhuma informação sobre a atividade específica do cliente.

Na sequência vamos abordar um dos princípios da SOA, a reusabilidade.

SERVIÇOS COMO COMPONENTES REUSÁVEIS

Sommerville (2011) aponta que o modelo de componentização é essencial para o desenvolvimento de *web services*.



Figura 69 - Componentes básicos da arquitetura do Web Service. / Fonte: o autor.

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

A arquitetura de *web service* é constituída por três componentes básicos: o servidor de registro (*broker server*), o provedor de serviços (*service provider*) e o solicitante (*service requestor*) que é o cliente, como você pode observar na Figura 69.

O ciclo de vida de web service é composto por:

1. UDDI (*Universal Description Discovery and Integration*) fornece três funções principais:
 - a. Publicação: consiste na divulgação do serviço pela organização.
 - b. Descoberta: possibilita que um cliente procure e encontre determinado serviço.
 - c. Ligação: permite que o cliente estabeleça ligação (*bind*) e interaja com o serviço.
2. WSDL (*Web Services Description Language*) é uma API (*Application Programming Interface*) em XML para descrever o serviço e os métodos de requisição HTTP (não apenas GET e POST) de *web service*.
3. SOAP (*Simple Object Access Protocol*) é o processo de invocação remota de um método, pela interação entre o cliente e o servidor, especificando o endereço do componente, o nome do método e os argumentos para esse método no formato XML.

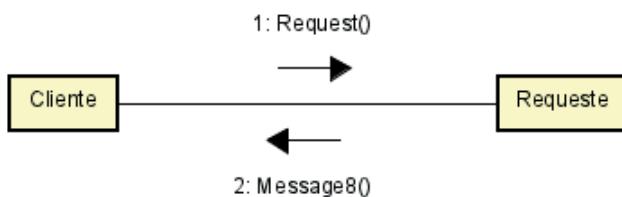


Figura 70 - Arquitetura REST / Fonte: o autor.

O diagrama de comunicação apresentado na Figura 70 ilustra a arquitetura REST, que basicamente utiliza o protocolo HTTP para fazer a comunicação cliente-servidor, possui operações bem definidas (GET, POST, PUT, DELETE, entre outras), todos os seus recursos são disponibilizados via URL e a transferência de informações utiliza diversos formatos padrão, entre eles XML, HTML e JSON.

Um objeto no formato JSON (*JavaScript Object Notation*) é sempre representado na forma de um *array*, como ilustra a Figura 71.

```

1 [ 
2   {
3     "id": 6,
4     "name": "Macarrão Espaguete",
5     "price": 35.9,
6     "description": "Macarrão fresco espaguete com molho especial e tempero da casa.",
7     "imageUri": "https://raw.githubusercontent.com/devsuperior/sds2/master/assets/macarrao_espaguete.jpg"
8   },
9   {
10    "id": 7,
11    "name": "Macarrão Fusili",
12    "price": 38.0,
13    "description": "Macarrão fusili com toque do chef e especiarias.",
14    "imageUri": "https://raw.githubusercontent.com/devsuperior/sds2/master/assets/macarrao_fusili.jpg"
15  },
16  {
17    "id": 8,
18    "name": "Macarrão Penne",
19    "price": 37.9,
20    "description": "Macarrão penne fresco ao dente com tempero especial.",
21    "imageUri": "https://raw.githubusercontent.com/devsuperior/sds2/master/assets/macarrao_penne.jpg"
22  }
]

```

Figura 71 - Arquivo JSON de uma lista de produtos / Fonte: o autor.

Os objetos são especificados entre chaves e cada atributo é formado por um par com nome e valor. Sempre que for representar mais um objeto eles devem estar dentro de um *array*, indicado por colchetes.



QR CODE

Para apimentar um pouco a conversa e entusiasmar você a desenvolver uma API REST vou te indicar uma leitura disponibilizada na documentação do Microsoft Azure. No link a seguir você encontrará o passo a passo para projetar uma API. Boa leitura!

ENGENHARIA DE SERVIÇOS

Agora, dentro do contexto da arquitetura de software, vamos abordar o desenvolvimento de serviços para aplicações SOA. Sommerville (2011) indica que o ponto de partida para criar um novo serviço, muitas vezes, consiste em partir de um componente existente que será convertido em um serviço. Nesse caso a engenharia deve primar pela generalização e remover as especificidades.

Sommerville (2011) sugere um processo de engenharia de serviços dividido em 3 etapas: identificar o serviço candidato, projetar o serviço e implementar o serviço. Ao identificar os possíveis serviços que podem ser implementados e listar os requisitos funcionais deste serviço, você deve definir as mensagens de entradas e saídas, bem como o tratamento de exceções no caso de falhas, por fim o serviço deve ser implementado, testado e disponibilizado para utilização.

Um uso de *web services* pode ser a integração com sistemas legados que, apesar de utilizarem tecnologia obsoleta, são essenciais para o funcionamento da organização. Os serviços nesses casos podem fornecer acesso ao sistema legado para uma variedade de aplicações com tecnologias mais atuais.

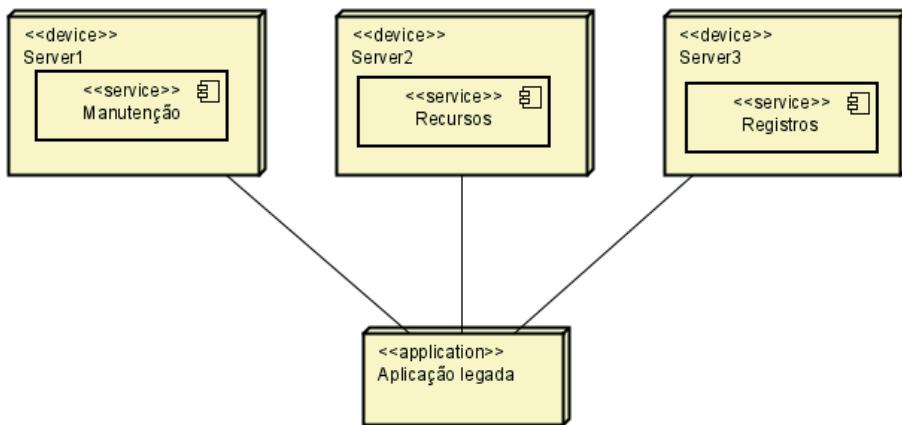


Figura 72 - Fornecimento de serviços a um sistema legado / Fonte: adaptada de Sommerville (2011, p. 367).

A Figura 72 representa um sistema legado que passa a integrar serviços de manutenção, recursos e registros. Cada serviço está relacionado a uma única funcionalidade, o que permite o reuso em outras aplicações.

DESENVOLVIMENTO DE SOFTWARE COM SERVIÇOS

O desenvolvimento de novos serviços consiste nos critérios de estruturação de objetos, na modelagem de interação dinâmica e na interação entre os objetos cliente e os objetos de serviço. A abordagem de determinar as operações de serviço pelas mensagens que chegam a um serviço auxiliam a projetar a interface de classes. As mensagens ajudam tanto a determinar a operação como os parâmetros de entrada e saída.

Você deve fazer uma análise das solicitações de mensagens feitas a cada serviço e, por intermédio delas, determinar as operações de cada serviço.

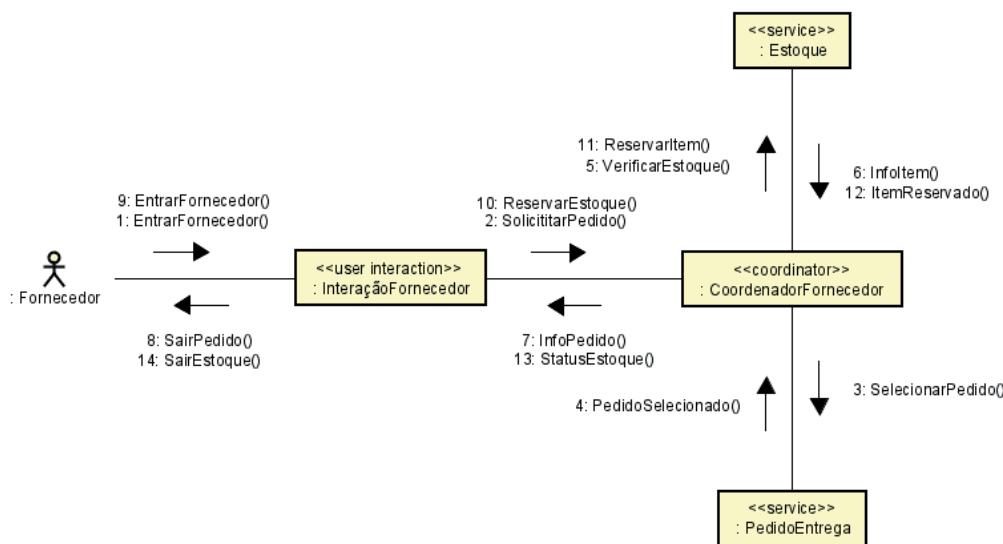


Figura 73 - Diagrama de comunicação do processo de entregar pedido
Fonte: adaptada de Gomaa (2011, p. 293).

Nas interações de objetos representadas pelo diagrama de comunicação na Figura 73, do processo de entregar pedido, especifica a mensagem verificar estoque ao Serviço de Estoque para verificar o estoque e determinar se os itens no pedido de entrega estão disponíveis. Essa solicitação necessita de um parâmetro de entrada de identificação do item e o status do estoque como parâmetro de saída. E assim acontece sucessivamente para cada mensagem trocada entre os objetos participantes.

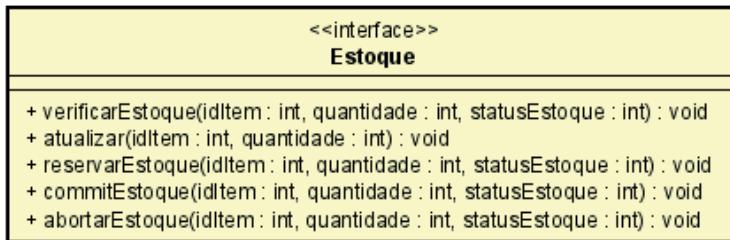
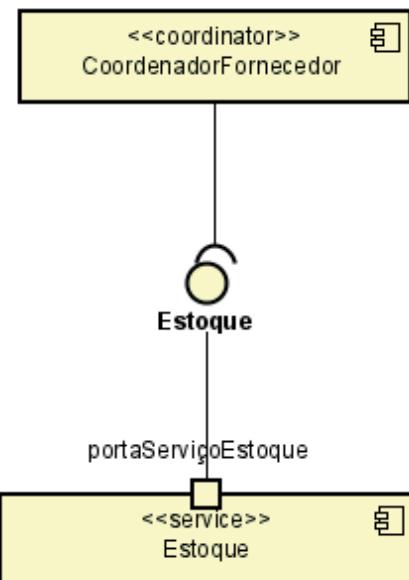


Figura 74 - Diagrama de classe referente à interface de serviço Estoque

Fonte: adaptada de Gomaa (2011, p. 294).

Na Figura 74 você observa a descrição da interface para o serviço de Estoque, que consiste em cinco operações, sendo que duas delas foram encontradas a partir do diagrama de comunicação ilustrado pela Figura 73.



Para concluir a modelagem do serviço, a Figura 75 ilustra um diagrama de componentes do serviço de Estoque, com uma porta fornecida, que oferece suporte à interface fornecida.

Essa sequência analisada para modelar o serviço de Estoque pode ser replicada para determinar as operações dos demais serviços.

Figura 75 - Diagrama de componentes para o serviço Estoque / Fonte: o autor.



ARQUITETURA DE SOFTWARE CONCORRENTE E EM TEMPO REAL

Chegamos à última arquitetura de software a ser estudada, e esta aborda projetos de embarcado, que inclui operações concorrentes e em tempo real. Este tipo de software, geralmente, precisa lidar com vários fluxos de eventos de entrada, sendo assim, o projeto de máquinas de estados finitos é fundamental para modelar a interação e os padrões de controle.

Sommerville (2011) indica que um software embarcado normalmente é do tipo apenas leitura e pode ser usado para controlar desde máquinas domésticas (telefones, fogões, micro-ondas, etc.), aplicações militares, até plantas industriais totalmente automatizadas. Esses sistemas interagem com sensores, coletando informações, e devem reagir a eventos gerados por esses sensores. A reação pode ser disparar um alarme, iniciar uma chamada telefônica, atualizar um painel de controle, abrir ou fechar uma válvula, ou atualizar o status do sistema entre outras funcionalidades.

Aplicações de tempo real no contexto de sistemas embarcados costumam ser complexos porque precisam lidar com vários fluxos independentes de eventos de entrada e ser reativo, ou seja, produzir várias saídas independentes. Esses eventos têm taxas de chegada que geralmente são imprevisíveis, embora devam estar sujeitos às restrições de tempo especificadas nos requisitos do sistema.

REFLITA



Você já utilizou um aplicativo de escritório (editor de texto, apresentação ou planilha) que está sendo executado em nuvem? Observe que de tempo em tempo ele apresenta uma informação de que está salvando as atualizações feitas.

PROJETO DE SISTEMAS EMBARCADOS

Ao desenvolver um sistema embarcado vários detalhes devem ser considerados. Desde o início do projeto é fundamental observar requisitos não funcionais, como o fator desempenho, seja no tempo de resposta ou na gestão de consumo.

No caso do consumo de energia, por exemplo, se o consumo for muito alto, é necessário formular uma solução com uma fonte de energia alternativa capaz de suportar a demanda de consumo.

Por se tratar de um sistema reativo a eventos no ambiente, a abordagem deste projeto de software se baseia em um modelo de estímulo-resposta. Ou seja, sempre que ocorre um evento no ambiente o sistema fornece uma resposta na forma de sinal ou mensagem enviada para o ambiente.

A Figura 60 é um exemplo de um sistema embarcado composto por sensores, controle e atuadores. Para cada sensor existe um processo que coleta e trata os dados dos sensores. Os dados coletados são processados e, então, o atuador é acionado para realizar uma determinada ação.

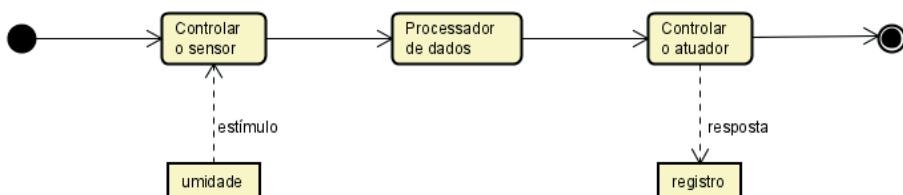


Figura 76 - Diagrama de atividades para os processos de sensores e atuadores
Fonte: adaptada de Sommerville (2011, p. 378).

O diagrama de atividades ilustrado pela Figura 76 representa o sensor de umidade como um objeto que envia um estímulo ao controlador do sensor de umidade. Veja que o estímulo é escrito na forma de dependência para o controlador. No instante em que o controlador recebe o estímulo e faz a leitura encaminha para o processador que, após executar as operações de análise, chama o controlador do atuador, que no caso é um registro. O registro fica na dependência do seu controlador para abrir, fechar, aumentar ou diminuir a vazão.

Agora vamos estudar os padrões de arquitetura para um sistema de tempo real.

PADRÕES DE ARQUITETURA

Como acabamos de conversar, muitos sistemas embarcados têm um controlador. A partir de agora vamos estudar diferentes tipos de padrões de controles: centralizado, distribuído e hierárquico. Outro ponto importante nesse estudo será aplicar esses padrões à arquitetura de software baseada em componentes.

PROJETO DE ARQUITETURA CENTRALIZADO

Nesta arquitetura o componente de controle é único para todos os sensores e, também, para os atuadores. Isso significa que o componente controlador recebe estímulos de todos os sensores e responde para todos os atuadores.

Sistemas embarcados que têm um único microcontrolador tem essa característica. Para isso, considere um sistema simples onde você tem uma placa controladora onde são conectados alguns componentes, um botão de acionamento, um acelerômetro e um LED (*Light-Emitting Diode*, diodo emissor de luz).

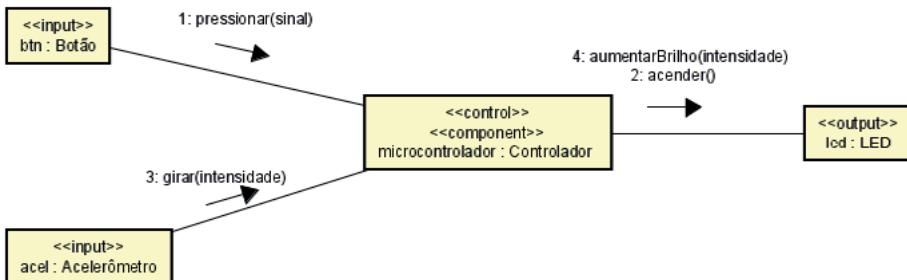


Figura 77 - Arquitetura centralizada para um projeto de software embarcado / Fonte: o autor.

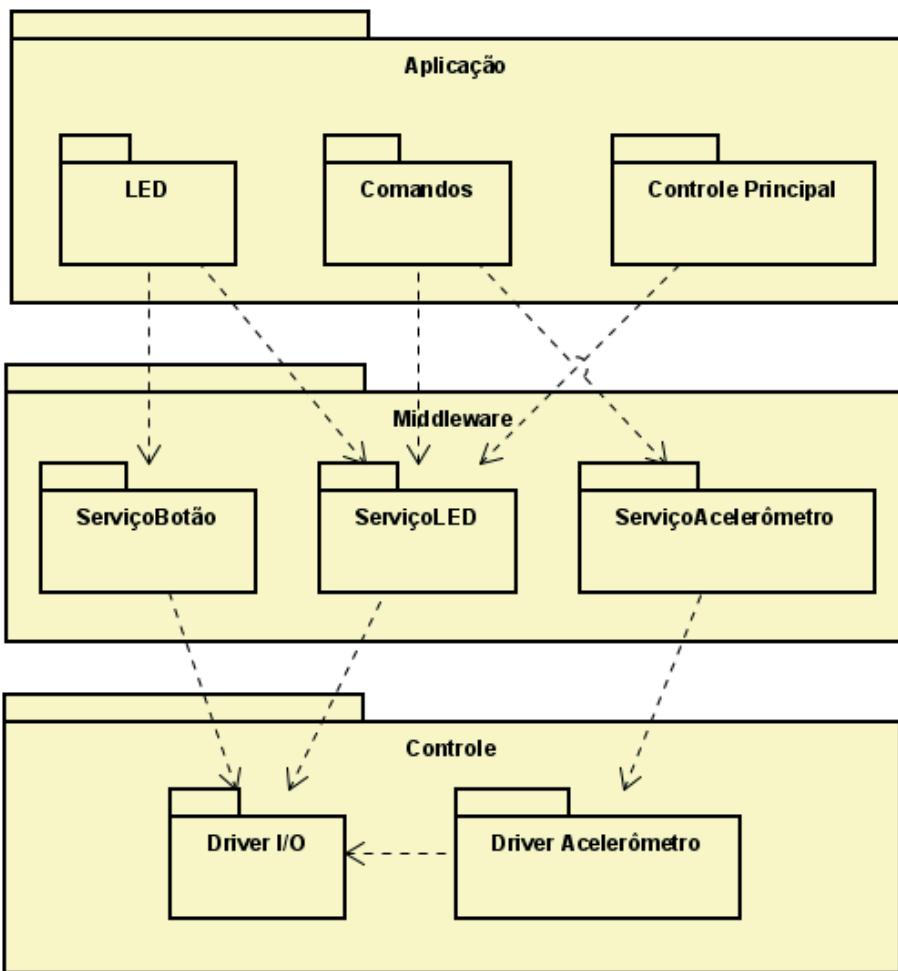
O diagrama de comunicação ilustrado na Figura 77 auxilia o desenvolvedor a compreender o que tem que ser feito com relação ao projeto. O sistema deve tratar comandos seriais, onde o LED apaga e acende de acordo com o clique do botão. O acelerômetro gira no eixo, e a leitura da aceleração possibilita aumentar ou diminuir o brilho (a intensidade) do LED em tempo real. Considere, também, a necessidade de entender sobre drivers e serviços associados a esse projeto, bem como a existência de unidades de software que devem ser desenvolvidas para gerenciar com maior nível de abstração a representação da Figura 77.

PROJETO DE ARQUITETURA DISTRIBUÍDO

Agora é possível detalhar o mesmo problema em nível de camadas, incluindo módulos com responsabilidades e interfaces bem definidas. Para isso vamos pensar em um padrão de projeto com controle distribuído, que contém vários componentes de controle.

Cada componente tem a responsabilidade de controlar uma determinada parte do sistema. Ao invés de pensarmos em um único componente no controle geral, o controle passa a ser distribuído entre os vários componentes de controle existentes.

Para notificarem entre si os eventos importantes, os componentes de controle se comunicam por meio de interfaces de comunicação ponto a ponto. A interação com o ambiente externo acontece por intermédio do controle centralizado.



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

Figura 78 - Arquitetura distribuída para um projeto de software embarcado / Fonte: o autor.

A partir do diagrama de pacotes apresentado na Figura 78, você é capaz de passar a ideia dos módulos funcionais do projeto e como eles se comunicam entre si. Com isso, entre outras coisas, o líder do projeto pode delegar as tarefas entre os desenvolvedores da equipe.

De modo geral, no padrão de controle distribuído, o controle é distribuído entre vários controladores. Cada controlador executa uma máquina de estado, recebendo entradas do ambiente externo por intermédio de sensores e controlando o ambiente externo enviando saídas para os atuadores.

O termo *middleware*, segundo Sommerville (2011) se refere ao software que

gerencia a diversas partes do sistema (diferentes tipos de processadores, representação de informações, protocolos e de comunicação, e assim por diante) e assegura que todas elas possam se comunicar e trocar informações entre si.

PROJETO DE ARQUITETURA HIERÁRQUICO

Agora chegou o momento de você pensar em um padrão de controle multinível, com vários componentes de controle. Nesta arquitetura de projeto, você tem um componente coordenador, que estabelece a comunicação entre os vários componentes de controle e o componente de controle.

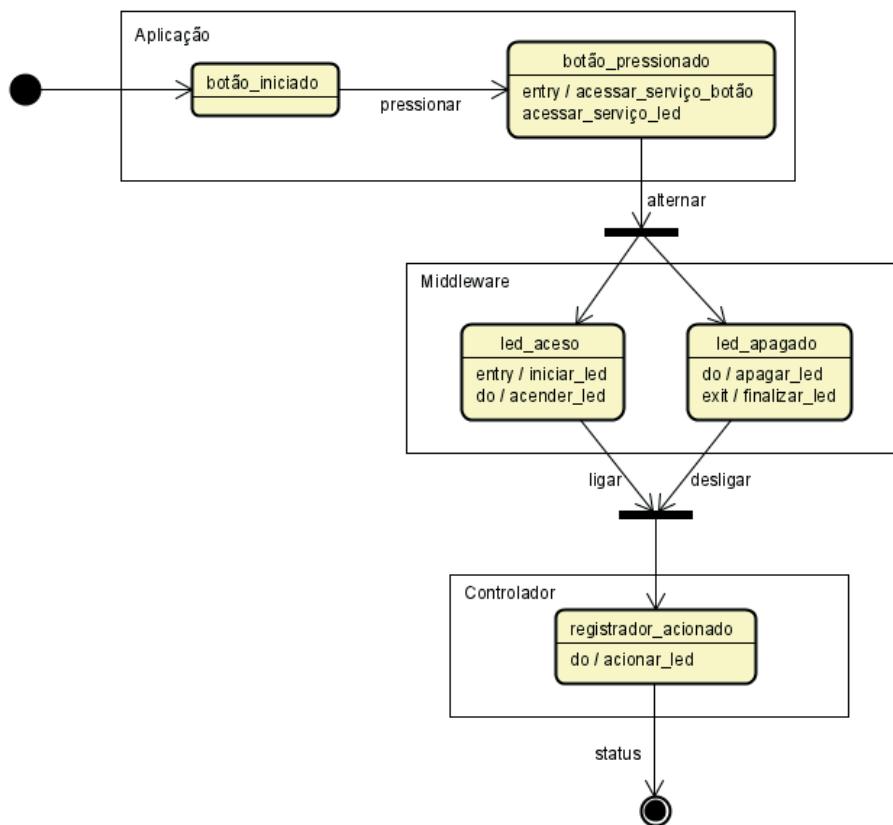


Figura 79 - Diagrama de máquina de estados de um projeto de software embarcado / Fonte: o autor.

Na Figura 79 você pode observar a mudança de status do LED de acordo com o acionamento do botão. A interface do sistema pode acontecer por eventos. Quando falamos em eventos podemos definir um nível de prioridade e executá-los de acordo com essa definição.

No diagrama de máquina de estados ilustrado na Figura 79 foi abordado apenas eventos relacionados ao botão, que é pressionar para acender o LED, e pressionar para desligar o LED. A partir desses eventos desencadeia a sequência de acionamentos até efetivamente ligar ou desligar o LED.

CRITÉRIO DE ESTRUTURA DE TAREFAS DE ENTRADA E SAÍDA

As tarefas de E/S de um sistema embarcado são estruturadas com critérios de decisão sobre as características da tarefa. Você pode considerar critérios como eventos, temporalidade ou demanda. A seguir vamos discutir um pouco mais a respeito desses critérios.

TAREFAS BASEADAS EM EVENTOS

Sempre que você estiver desenvolvendo um sistema embarcado que tenha um dispositivo orientado por interrupção, você terá uma tarefa de E/S orientada por evento com o qual o sistema deve fazer interface. Um dispositivo como um botão requer uma interrupção, por exemplo.

Neste caso de tarefa, a velocidade da operação está restrita à velocidade do dispositivo de E/S com o qual o sistema está interagindo. Segue, portanto, a mesma lógica de interrupções estudada na disciplina de Sistemas Operacionais.

TAREFAS PERIÓDICAS

Diferentemente de uma tarefa orientada por evento, uma tarefa de E/S periódica lida com um dispositivo que é sondado regularmente. Considere o um sensor de umidade do solo, por exemplo.

Nesta situação, embora a ativação da tarefa seja periódica, sua função está relacionada à uma tarefa de E/S. Um temporizador externo executa a ação de leitura do sensor e, em seguida, aguarda o próximo evento de temporizador. O período da tarefa é o tempo entre ativações sucessivas.

As tarefas de E/S periódicas são frequentemente usadas para dispositivos sensores em ambientes industriais para amostragens periódicas. A tarefa periódica é ativada regularmente com o objetivo de fazer varredura nos parâmetros dos sensores.

TAREFAS ORIENTADAS POR DEMANDA

Existem dispositivos que não precisam ser sondados periodicamente. Nessas situações as tarefas de E/S do dispositivo são orientadas por demanda. Um exemplo de sistema embarcado orientado por demanda é um sistema para controle de frota de veículos, que calcula aceleração, velocidade, consumo de combustível, trajeto, entre outras variáveis.

Gomaa (2011) considera dois casos referentes a esses dispositivos:

1. Uma tarefa computacional de entrada orientada por demanda para ler e consumir os dados do dispositivo.
2. Uma tarefa computacional de saída orientada por demanda para produzir dados para o dispositivo.

De modo geral, as tarefas de E/S orientadas por demanda são mais comuns em dispositivos de saída, para exibir o resultado de um cálculo, por exemplo. Em dispositivos de entrada é usual as tarefas de entrada periódicas.

CONSIDERAÇÕES FINAIS

Nesta unidade você aprendeu sobre a importância de se criar um projeto de arquitetura para um software. A arquitetura é importante independente do tipo e do objetivo do software em questão. E é possível utilizar todos os diagramas da UML aprendidos até aqui para construir a arquitetura do software.

O primeiro passo dos nossos estudos foi se familiarizar com os gêneros e os estilos de arquitetura de software. Quando falamos em gênero de arquitetura nos referimos aos softwares voltados para os mais diversos segmentos. Já a arquitetura do software refere-se ao modo como esse software será desenvolvido. Vimos que um projeto de arquiteturas pode ser dividido em camadas, ser centrado em dados ou executar um *pipeline* entre outras possibilidades.

A arquitetura de um software orientada a objetos se baseia em classes. Para isso estudamos os tipos de classes e o que representam dentro da arquitetura de software. Basicamente as classes representam a lógica do negócio, as entidades do banco de dados, as interfaces com o usuário e entre as partes componentes do sistema.

Vimos que a maioria dos sistemas de gestão disponíveis, até por conta das novas tecnologias disponíveis, estão espalhados geograficamente por diversos lugares. Esses sistemas requerem um modelo de software chamado cliente-servidor. Nesta arquitetura o cliente faz uma requisição a um serviço que é executado a partir de um servidor. Para o funcionamento correto desta arquitetura você deve considerar o uso de uma rede de computadores e suas implicações.

Arquiteturas de software que consomem alguma API e rodam em servidores ou de terceiros, comuns em sistemas distribuídos, cada serviço deve ser tratado como um módulo independente. Para consumir essas API's é necessário utilizar um protocolo específico para acessá-la.

Por fim, estudamos a arquitetura dos sistemas embarcados, que operam com sensores, controladores e atuadores em situações que envolvem interrupções, temporalidade ou demanda.

ATIVIDADES



1. A arquitetura de software é uma perspectiva estrutural, estática e dinâmica do projeto de software. De acordo com esse contexto, assinale a alternativa que descreve a arquitetura de software.
 - a) A construção de um software.
 - b) A estrutura de um sistema cliente / servidor.
 - c) A estrutura geral de um sistema de software.
 - d) As classes de software e seus relacionamentos.
 - e) Uma visão em termos de hierarquia de módulos.

2. Um projeto de software orientado a objetos utiliza classes para encapsular informações e operações, além do conceito de herança, dentre outros recursos que são determinados durante a fase de análise.

Com base nesse contexto, analise as afirmações a seguir:

- I. Uma classe de abstração de dados pode ser uma classe de entidade.
- II. Uma janela provavelmente é uma classe de interface gráfica do usuário.
- III. Uma caixa de diálogo será encapsulada em uma classe da lógica de negócios.
- IV. No conceito de herança uma subclasse pode redefinir os atributos herdados da superclasse.

É correto o que se afirma em:

- a) Apenas I está correta.
- b) Apenas I e II estão corretas.
- c) Apenas II e III estão corretas.
- d) Apenas II, III e IV estão corretas.
- e) I, II, III e IV estão corretas.

ATIVIDADES



3. A arquitetura de software cliente-servidor se caracteriza por ser constituída de vários computadores, no qual todos os componentes do sistema executam em um único computador, e a comunicação entre eles acontece por intermédio de uma rede.

Assinale Verdadeiro (V) ou Falso (F):

- () No padrão básico da arquitetura cliente-servidor vários clientes solicitam serviços e um servidor atende às solicitações do cliente.
- () A camada intermediária no padrão de arquitetura cliente-servidor multicamadas é a camada de serviço.
- () A diferença entre o padrão de arquitetura de múltiplos clientes com múltiplos serviços do padrão de arquitetura de múltiplos clientes e serviço único é que um serviço pode responder a solicitações de vários clientes.

Assinale a alternativa correta:

- a) V; V; F.
- b) F; F; V.
- c) V; F; V.
- d) F; F; F.
- e) V; V; V.

- 4 A arquitetura de *web service* é constituída por três componentes básicos: o servidor de registro, o provedor de serviços e o solicitante que é o cliente. Nesse contexto, assinale a alternativa que corresponde ao objeto *broker*.

- a) Um objeto que invade um sistema.
- b) Um objeto que envia solicitações para outros objetos.
- c) Um objeto utilizado para descobrir a localização do serviço.
- d) Um objeto que lida com solicitações enviadas por outros objetos.
- e) Um objeto que intermedia as interações entre clientes e serviços.

ATIVIDADES



5. As tarefas de E/S de um sistema embarcado são estruturadas com critérios de decisão sobre as características da tarefa. Você pode considerar critérios como eventos, temporalidade ou demanda. Tarefa periódica é uma tarefa que _____.

Assinale a alternativa que completa corretamente a afirmação acima.

- a) é ativada por um evento externo.
- b) é ativada por um evento de entrada.
- c) é ativada por um evento temporizador.
- d) responde a cada mensagem que recebe.
- e) é ativada por uma mensagem interna ou evento de outra tarefa



Talvez você ainda esteja se perguntando sobre quais diagramas da UML utilizam para modelar a arquitetura de um software. Pois bem, vamos falar um pouco a respeito dessa questão um tanto espinhosa para alguns.

Como vimos a UML é composta de diagramas estruturais e comportamentais. Na sua versão 2.5 a UML traz 7 (sete) diagramas estruturais e 5 (cinco) diagramas comportamentais. Os diagramas estruturais são os diagramas de classes, objetos, componentes, instalação ou implantação, de pacotes, estrutura composta e perfil. Já os diagramas comportamentais são representados pelos diagramas de atividades, casos de uso, máquina de estados e dois diagramas de interação, representados pelos diagramas de comunicação e sequência.

O objetivo do desenvolvimento de software é o software em si, portanto a documentação deve ser suficiente para que desenvolvedores e clientes possam ter a ideia do software. Portanto, o uso e a escolha dos diagramas vão depender muito do que se precisa representar do sistema.

Vamos fazer uma análise das questões primordiais. Começando pelas interações que o sistema faz com outros sistemas ou usuários. Para isso, a melhor maneira de representação é o diagrama de casos de uso. Contudo, muitas vezes é fundamental conhecer determinado processo para se desenhar o diagrama de casos de uso. Em uma situação como essa você deve se concentrar na elaboração do diagrama de atividades relacionando àquele processo, para então partir para o diagrama de casos de uso.

Se estamos falando em um sistema orientado a objetos, o diagrama de classes é essencial. No entanto, como vimos, o engenheiro deve empacotar as classes de acordo com a sua representatividade - a lógica do negócio, as entidades de banco de dados, os serviços, as interfaces dos sistemas, e assim por diante. Nesse instante, pensando no desenvolvimento, o engenheiro de software vai se deparar com a necessidade de especificar determinado framework e, portanto, fazer uso do diagrama de componentes.

Ainda, com relação às classes, o engenheiro de software deve informar seus atributos e operações. Se atributos e operações não estiverem claros por qualquer razão, em determinado ponto de funcionamento do sistema, você deve elaborar um diagrama de comunicação ou sequência. Estes diagramas representam as mensagens trocadas entre os objetos nas várias camadas do software, incluindo seus parâmetros e retornos.

Por último, no caso de sistemas onde a interação no sistema muda o status dos objetos, você deve elaborar um diagrama de máquina de estados. Esse diagrama apresentará as operações e o status após a conclusão da mesma.

Sendo assim, fechamos quatro visões distintas do sistema, que serão elaboradas de acordo com as necessidades e implicações do software. A visão de contexto, a visão estrutural, a visão de interação e a visão comportamental. As duas primeiras são visões essenciais enquanto que as duas últimas vão depender das especificidades do software em questão.

Fonte: O autor.

MATERIAL COMPLEMENTAR



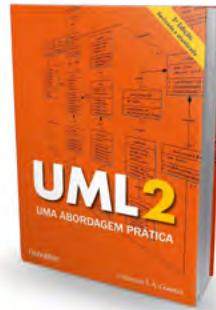
LIVRO

UML 2: Uma abordagem prática.

Gilleanes Guedes

Editora: Novatec

Sinopse: A UML é uma linguagem utilizada para modelar software baseados no paradigma de orientação a objetos. Essa é a linguagem-padrão de modelagem adotada pela Engenharia de Software. Este livro contém exemplos práticos mediante a apresentação de 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, para a modelagem de software. A obra contém diversos estudos de caso modelados como exemplos ao longo dos capítulos.



MODELAGEM E ARQUITETURA ORIENTADA A OBJETOS

UNIDADE

V

Objetivos de Aprendizagem

- Compreender o ciclo de vida do desenvolvimento de software
- Distinguir requisitos funcionais, não funcionais e as regras de negócio
- Desenhar a arquitetura de software que atende às necessidades do estudo de caso
- Projetar o desenvolvimento de software
- Organizar as etapas de construção e operação do software

Plano de Estudo

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

- Processos de software
- Engenharia de requisitos
- Modelagem do sistema
- Modelagem arquitetônica
- Projeto, implementação, teste e evolução do software

INTRODUÇÃO

Olá, cara(o) aluna(o), chegamos à última etapa dos estudos sobre Modelagem de Software. Acompanhando o contexto apresentado até aqui, nesta unidade, você desenvolverá uma aplicação prática de modelagem de software. Vamos abordar uma situação problema e, a partir desse problema, construir o modelo de domínio e a arquitetura do futuro software a ser implementado.

Nesta etapa dos nossos estudos será fundamental recordarmos alguns conceitos fundamentais, como a definição da metodologia de desenvolvimento, a utilização de alguma ferramenta CASE (*Computer-Aided Software Engineering*) para auxiliar na modelagem, além de apontar os próximos passos a partir da modelagem.

O início dos nossos estudos se refere aos modelos de processo, ou seja, definir como será o ciclo de vida de desenvolvimento do software. Podemos classificá-los como linear, iterativo, evolucionário ou paralelo, por exemplo. Após a escolha da metodologia é fundamental levantar, classificar e validar os requisitos.

Como você já estudou, o software deve ser projetado com base em diferentes visões, que compreendem o contexto do software, as interações do software com o ambiente externo, além da visão estática que apresenta a estrutura do software e a visão comportamental diante de determinados eventos que ocorrem durante a execução do mesmo. As diferentes visões do software auxiliam na compreensão da equipe de desenvolvimento e dos diferentes *stakeholders* envolvidos na aquisição do produto.

Com base no paradigma de desenvolvimento orientado a objetos vamos desenhar a arquitetura para a solução do problema apresentado. A arquitetura escolhida será do tipo cliente-servidor, de acordo com a compreensão da equipe e a necessidade do cliente.

O objetivo é focar na análise e modelagem dos requisitos, de tal forma que a equipe de desenvolvedores possa ter subsídios suficientes para as próximas etapas do ciclo de vida do software, que vai da implementação à manutenção do software. Agora começa a nossa jornada. Vamos lá!



PROCESSOS DE SOFTWARE

Esta unidade será dedicada ao estudo de caso da modelagem de um software de compra online. Para isso, ao longo da unidade fornecemos vários diagramas UML de diferentes tipos.

Na condição de engenheiro de software, responsável por atender a solicitação que culminará em um produto de software é importante incluir quatro atividades descritas por Sommerville (2011), que são:

1. Especificação de software.
2. Projeto e implementação de software.
3. Validação de software.
4. Evolução de software.

Embora tratemos de todas elas, de alguma forma, o foco desta unidade consiste na especificação e no projeto de software. Para isso, vamos partir do princípio de que a consultoria contratada para desenvolver a aplicação solicitada já possua um modelo de processo de software.

A padronização dos processos de software tende a melhorar a comunicação entre os envolvidos no processo, além de reduzir o período de treinamento e diminuir os custos de retrabalho.

Na sequência vamos reforçar o conhecimento dos principais modelos de processo e as metodologias ágeis.

MODELOS DE PROCESSO DE SOFTWARE

É muito importante que você tenha em mente que quando falamos de processo estamos nos referindo a um conjunto de atividades de trabalho, ações e tarefas realizadas para se obter algum artefato de um software. O processo de software consiste em uma metodologia que abrange cada ação e o conjunto de tarefas associadas para a sua conclusão.

Vamos fazer um breve estudo dos principais modelos de processo de software propostos: linear, iterativo, evolucionário, paralelo e o processo unificado (RUP).

O método em cascata (linear) representando pela Figura 80 considera cada atividade do processo de desenvolvimento de software como uma fase distinta.

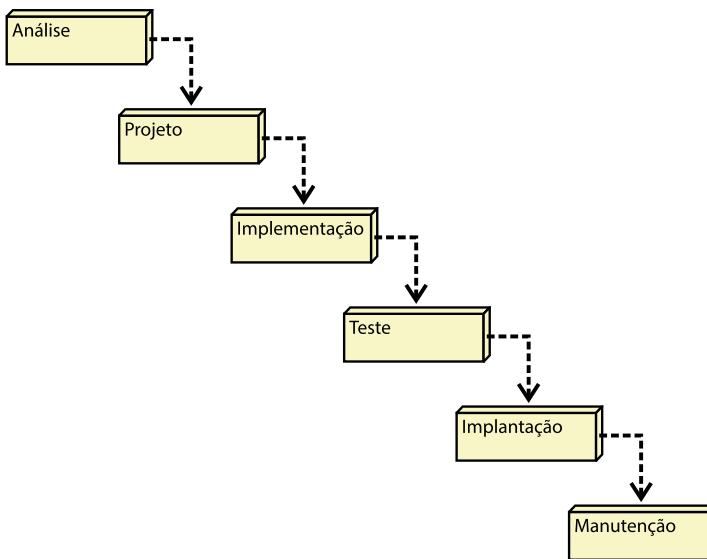


Figura 80 - Fluxo de processo linear / Fonte: o autor.

O modelo de processo iterativo repete uma ou mais das atividades antes de prosseguir para a seguinte. Já o modelo evolucionário tem como característica executar essas atividades de forma circular. Nesse modelo, cada volta completa, da análise até a implantação, indica uma versão do software. A partir desse ponto o software vai evoluindo a cada versão disponibilizada em produção.

O modelo em paralelo permite que uma ou mais atividade seja executada em paralelo. Porém, dependendo da organização para construção do software,

esse modelo pode exigir uma equipe de desenvolvimento com vários integrantes de diversas áreas distintas.

O processo unificado (*Unified Process*), ilustrado na Figura 81, é descrito por Guedes (2018) como um método de desenvolvimento de software dividido em quatro fases: concepção, elaboração, construção e transição.

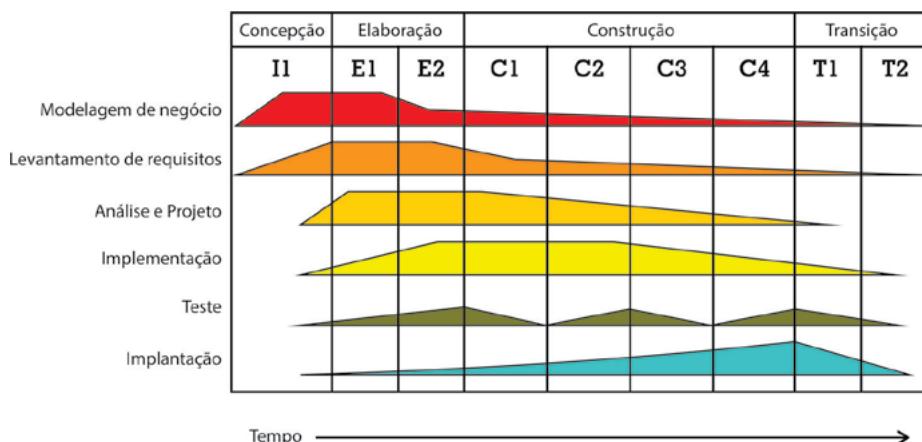


Figura 81 - Fases do Processo Unificado / Fonte: WIKIPEDIA. Fases do Processo Unificado(PU). 2007. Disponível em: <https://commons.wikimedia.org/wiki/File:Development-iterative.png>. Acesso em 24 jun. 2021.

DESENVOLVIMENTO ÁGIL

O desenvolvimento ágil, segundo Pressman (2011), combina filosofia e os princípios do desenvolvimento de software. Tem como objetivo principal a satisfação do cliente e é baseado na noção de desenvolvimento e entrega incremental (SOMMERVILLE, 2011). Os princípios norteadores dos métodos ágeis são:

1. Envolvimento do cliente
2. Entrega incremental
3. Pessoas, não processos
4. Aceitação de mudanças
5. Manter a simplicidade

Os métodos ágeis são mais indicados para o desenvolvimento de produtos de software de pequeno e médio porte em que as regras e regulamentos possam estar sujeitos à mudanças.

As principais abordagens de desenvolvimento ágil são: dirigido a planos, *extreme programming* (XP) e Scrum. A Figura 3 é uma representação visual dos principais artefatos do framework Scrum e o seu relacionamento com a *sprint*.

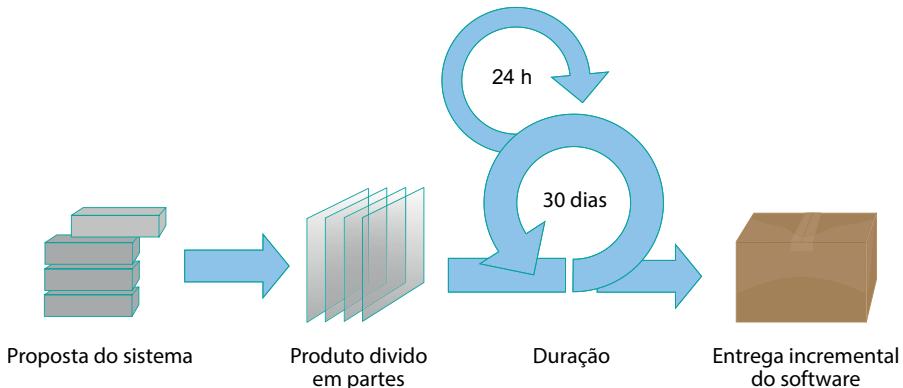


Figura 82 - Framework Scrum. / Fonte: WIKIPEDIA. Representação visual dos principais artefatos do Scrum Framework e seu relacionamento com a Sprint. 2009. Disponível em: [https://pt.wikipedia.org/wiki/Scrum_\(desenvolvimento_de_software\)](https://pt.wikipedia.org/wiki/Scrum_(desenvolvimento_de_software)). Acesso em 24 jun. 2021.

SAIBA MAIS



Pressman (2011) lista alguns dos modelos de processo ágeis. Vamos às principais Metodologias Ágeis:

- **Programação extrema (XP):** Utiliza o paradigma orientado a objetos e divide-se em quatro atividades: planejamento, projeto, codificação e testes.
- **Scrum:** Utiliza um padrão de processo baseado em sprints, como mostra a Figura 3. O destaque é que o problema definido na sprint pode ser modificado em tempo real pela equipe Scrum.
- **Desenvolvimento dirigido a funcionalidade (FDD):** Neste contexto entende-se que uma funcionalidade é uma função valorizada pelo cliente e deve ser implementada em duas semanas ou menos.

DESCRIÇÃO DO CENÁRIO

O cenário descrito a seguir serve de base para o sistema de compra online.

O cliente acessa o site da loja para fazer compras online. O cliente pode solicitar itens, realizar a compra e se registrar como cliente. Ao solicitar itens ao cliente, pode pesquisar e ver os produtos disponíveis. Ele também pode usar a busca para realizar compras. Ao se registrar no site o cliente pode obter cupons ou ser convidado para vendas privadas. Observe que o caso de uso do *checkout* está incluído caso de uso não disponível por si só - o *checkout* faz parte da compra.

Na maioria das vezes o pagamento será efetuado por intermédio do uso de cartão de crédito. Nessas condições, o comerciante envia uma solicitação de transação de cartão de crédito para a operadora de cartão de crédito. A operadora que emitiu o cartão de crédito do cliente pode aprovar ou rejeitar a transação. Se a transação for aprovada, os fundos serão transferidos para a conta bancária do comerciante.

Como o sistema vai funcionar totalmente pela Web, os requisitos de segurança do site exigem a separação das interfaces administrativas das funções comuns fornecidas aos usuários. O sistema deve ter aplicativos separados para administradores e para usuários comuns. Exceto para administradores, alguma parte das interfaces administrativas também deve estar disponível para a equipe de *Help Desk*, pois eles precisam ser capazes de ajudar os clientes com problemas ao usar o site voltado para o cliente.

Com base nessas informações já é possível que você trabalhe no documento de requisitos para esse software.

ENGENHARIA DE REQUISITOS

Considere que o desenvolvimento desse projeto seguirá o modelo incremental. A escolha se deve ao fato que, segundo Sommerville (2011), o desenvolvimento incremental fornece uma implementação inicial aos usuários e contínua

adequação, produzindo novas versões até que o sistema esteja totalmente concluído. Além disso, as metodologias ágeis também são baseadas na noção do desenvolvimento e entrega incremental.

Antes de iniciarmos o estudo é importante salientar a necessidade do uso de uma ferramenta CASE (*Computer-Aided Software Engineering*, Engenharia de Software Auxiliada por Computador).

Existem diversas ferramentas CASE disponíveis no mercado, com vários tipos de licenças disponíveis. É importante salientar que o uso da ferramenta CASE na modelagem de software permite, entre outras coisas, converter os modelos gráficos em scripts para tabelas de banco de dados e código-fonte para classes do sistema, além de possibilitar diversas visualizações do sistema a ser construído.

As ferramentas CASE oferecem uma linguagem única de comunicação entre todos os membros da equipe envolvidos no processo de desenvolvimento do software. As ferramentas CASE facilitam a integração de analistas de sistemas, administradores de dados, desenvolvedores de aplicações e gerentes de projeto.

Para levantar os requisitos você sabe que são necessários realizar reuniões, fazer entrevistas, aplicar questionários, levantar a documentação, acompanhar *in-loco* para observar o processo, até que os requisitos funcionais e não-funcionais estejam validados e homologados. Então vamos lá. Chegou a hora de colocar a mão na massa.

LEVANTAMENTO DE REQUISITOS

Após o levantamento de requisitos iniciais do Sistema de Compra Online, os analistas de negócios participantes chegaram aos requisitos funcionais e não funcionais, que serão expressos em sentenças em linguagem natural.

A lista a seguir indica os requisitos funcionais:

1. O cliente pode pesquisar itens, navegar no catálogo, exibir itens recomendados, adicionar itens ao carrinho de compras ou lista de desejos.
2. Realizar conferência para efetivar a compra.

3. Ao capturar fundos que foram autorizados o comerciante deve concluir uma transação.
4. Cancelar transações que ainda não foram liquidadas.
5. Confirmar dados do cliente.
6. O administrador do site pode criar diferentes grupos de usuários, definir diferentes privilégios ou opções e, posteriormente, alguns grupos de usuários podem ser modificados ou mesmo excluídos.
7. Uma sessão de usuário é criada a cada nova solicitação de entrada depois que o usuário foi autenticado, e o administrador do site pode verificar informações estatísticas, analisar o status, e cancelar uma sessão, se necessário.
8. O administrador do site deve ser capaz de ver o status dos registros de log.

A lista a seguir é dos requisitos não funcionais:

1. A autenticação pode ser feita através da página de *login* do usuário e *cookie* de autenticação do usuário.
2. O *checkout* (conferência) inclui o pagamento que pode ser feito usando cartão de crédito e serviço de pagamento de crédito externo.
3. A conclusão de uma transação pode ser enviada por meio do *gateway* de pagamento ou solicitada sem usar o sistema usando autorização de voz, por exemplo.
4. Detectar intrusão de autenticação quando algum usuário ligar e pedir para bloquear uma conta.
5. Os registros de logs só podem ser incluídos, desde que os registros de log mais antigos não sejam reescritos ou excluídos.

A especificação de requisitos em linguagem natural apesar de intuitiva e universal pode levar a uma compreensão ambígua, dependendo do leitor. Por conta disso, posteriormente, vamos elaborar especificações estruturadas para os requisitos do sistema.

REGRAS DE NEGÓCIO

Outro ponto importante a ser identificado são as regras de negócio. Segundo Bezerra (2015), as regras de negócio descrevem a maneira como a organização funciona. No sistema em que estamos estudando foram identificadas as seguintes regras de negócio:

- [RN00] Exibir itens recomendados e adicionar à lista de desejos exigem que o cliente esteja autenticado.
- [RN01] Um item pode ser adicionado ao carrinho de compras sem autenticação do usuário.
- [RN02] Para efetuar o *checkout* o cliente da Web deve ser autenticado.
- [RN03] O valor solicitado deve ser autorizado pela operadora do cartão de crédito do cliente e, se aprovado, deve ser enviado para liquidação.
- [RN04] A liquidação com cartão de crédito deve ter os valores aprovados na transação depositados na conta bancária do comerciante.
- [RN05] Autorização solicitada e a transação não liquidada.
 - Se nenhuma ação adicional for tomada dentro de alguns dias, a autorização expira.
 - Se os comerciantes quiserem verificar a disponibilidade de fundos no cartão de crédito do cliente.
 - Se o item não estiver em estoque.
 - Se o comerciante quiser revisar os pedidos antes do envio.
- [RN06] Creditar reembolso ao cliente.
 - Se o cliente contesta uma transação que foi processada e liquidada com êxito por meio do sistema
 - Se uma transação que não foi enviada originalmente por meio do gateway de pagamento.

As regras de negócio variam de organização para organização e, normalmente, são identificadas na fase de levantamento de requisitos.

DOCUMENTAÇÃO DA MODELAGEM DE CASOS DE USO

A partir do documento de requisitos e das regras de negócios você consegue identificar os atores e os casos de uso, de tal forma a criar o diagrama de casos de uso. Esse sistema é composto de três subsistemas. O sistema de compra online, propriamente dito, o sistema de processamento do cartão de crédito e o sistema de administração do *Website*.

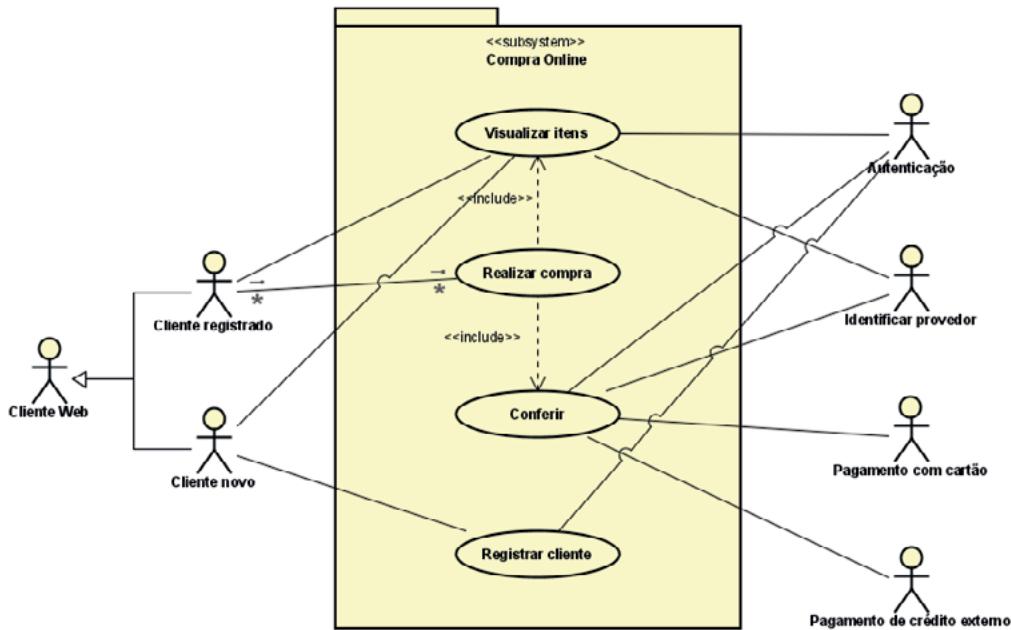


Figura 83 - Diagrama de casos de uso de alto nível da compra *online* / Fonte: o autor.

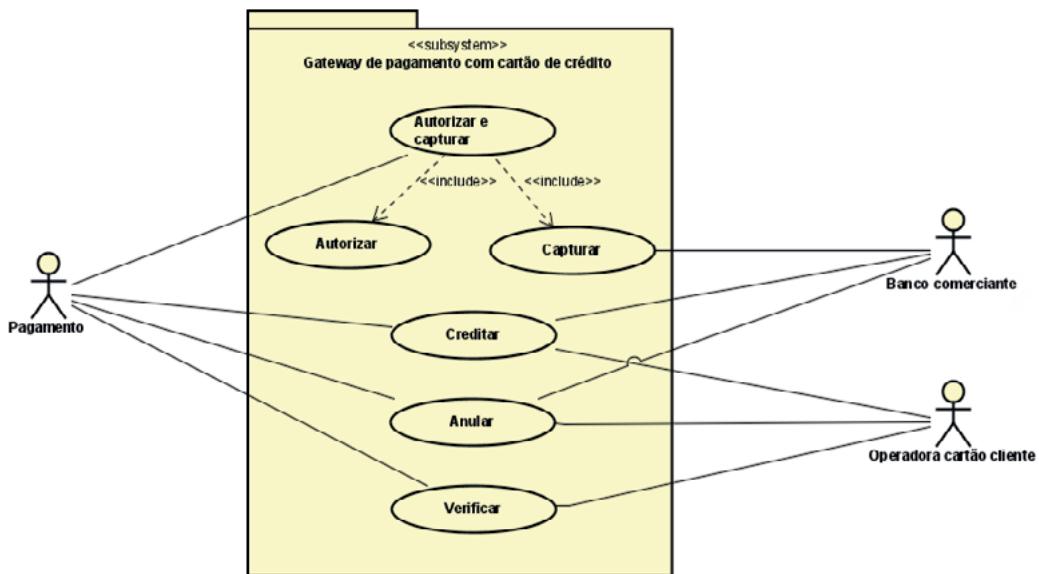


Figura 84 - Diagrama de caso de uso para o sistema de processamento de cartões de crédito / Fonte: o autor.

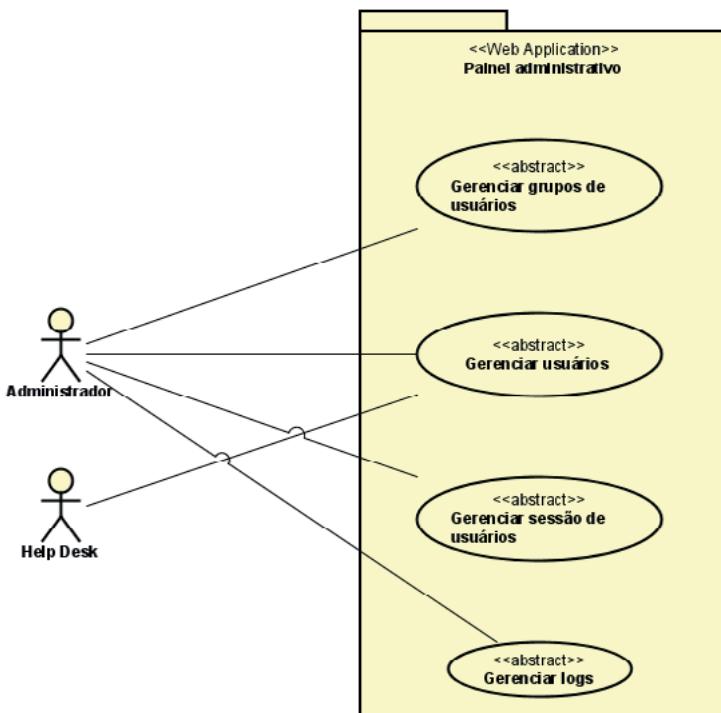


Figura 85 - Diagrama de caso de uso do painel administrativo para o Website / Fonte: o autor.

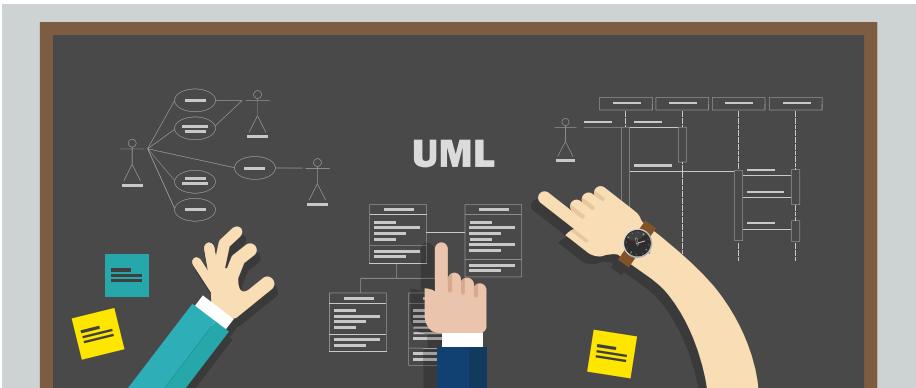
O diagrama de casos de uso ilustrado na Figura 83 identifica os atores e os casos de uso do subsistema de compra *online*. Já o diagrama de casos de uso na Figura 84 representa a entrada de pagamentos com cartão de crédito. Para isso, a entrada é realizada por um módulo do sistema de processamento de cartão de crédito do comerciante. A Figura 85 ilustra um diagrama de alto nível dos casos de uso das funções administrativas fornecidas ao administrador do website.

O diagrama de casos de uso necessita quase sempre de uma descrição para melhor representar as funcionalidades inerentes a ele. A seguir vamos analisar o caso de uso Conferir, apresentado na Figura 83, no diagrama de casos de uso que representa o subsistema de compra online.

Nome do caso de uso	Conferir
Descrição	Permite a conclusão da compra
Autor	Cliente da web
Pré-condições	Adicionar itens ao carrinho
Pós-condições	Compra efetuada com sucesso
Fluxo básico	1. Cliente se autentica 2. Cliente atualiza o carrinho 3. Calcular frete 4. Efetuar pagamento
Fluxos alternativos	2.1 Calcular valor total da compra

Quadro 1 - Especificação estrutural do caso de uso Conferir / Fonte: o autor.

A especificação estruturada varia de acordo com o autor, mas segundo Sommerville (2011) um formulário-padrão para especificar requisitos funcionais deve incluir a descrição, as entradas (pré-condição) e as saídas (pós-condições) e a descrição de efeitos colaterais, caso haja algum.



MODELAGEM DO SISTEMA

Nessa etapa do desenvolvimento do projeto vamos nos dedicar a apresentar uma visão estrutural, interativa e comportamental do sistema. Para isso vamos modelar o domínio do problema por intermédio de um diagrama de classes. Na sequência vamos apresentar o diagrama de comunicação para representar o catálogo *online*, o cliente da *Web* pode pesquisar o estoque, visualizar e comprar itens *online*. Para concluir a modelagem serão desenvolvidos dois diagramas comportamentais, o diagrama de máquina de estados para apresentar o ciclo de vida de uma conta de usuário e um diagrama de atividade para mostrar uma compra *online*.

MODELO DE DOMÍNIO

É de suma importância que você entenda que os requisitos de um sistema são identificados a partir de um domínio, e que o domínio se refere a um conjunto de conceitos, terminologias e atividades específicas de uma área do conhecimento. A equipe de desenvolvimento necessita entender o domínio para que possa automatizá-lo (BEZERRA, 2015).

No contexto do estudo de caso, o modelo de domínio para o sistema de compra online será representado por um diagrama de classes da UML. O diagrama ilustrado na Figura 86 apresenta termos e relações comuns para compras online, como cliente, usuário da *Web*, conta, carrinho de compras, produto, pedido,

pagamento, entre outros. O diagrama de classes permite a comunicação entre os analistas de negócios e os desenvolvedores de software.

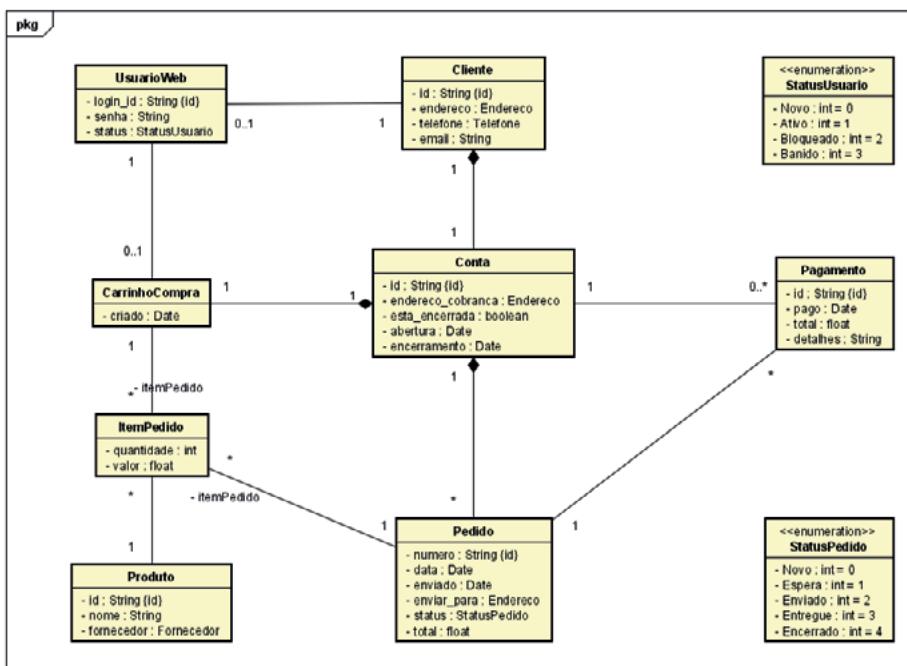


Figura 86 - Diagrama de classes referente ao domínio do sistema de compras online / Fonte: o autor.

Por intermédio deste diagrama de classes é possível compreender que cada cliente possui um *id* único e está vinculado a exatamente uma conta. A conta possui carrinho de compras e pedidos. O cliente pode se registrar como um usuário da Web para poder comprar itens online. O usuário da Web possui um *login* que serve para identificação exclusiva. O usuário da Web pode ser novo, estar ativo, ter sido temporariamente bloqueado ou banido e, ainda, estar vinculado a um carrinho de compras. O carrinho de compras pertence à conta.

A conta possui os pedidos dos clientes. O cliente pode não ter pedidos. Os pedidos dos clientes são classificados e exclusivos. Cada pedido pode se referir a vários pagamentos, possivelmente nenhum. Cada pagamento tem um id único e está relacionado a exatamente uma conta.

Cada pedido tem o status do pedido atual. Tanto o pedido quanto o carrinho de compras têm itens de linha vinculados a um produto específico. Cada

item de linha está relacionado a exatamente um produto. Um produto pode ser associado a muitos itens de linha ou a nenhum item.

Os diagramas a seguir utilizam-se do modelo de domínios para conseguir, por exemplo, identificar as operações de cada classe envolvidas nas associações.

MODELAGEM DAS INTERAÇÕES

Na maioria dos sistemas, os objetos não ficam ociosos; eles interagem uns com os outros passando mensagens. Uma interação é um comportamento que comprehende um conjunto de mensagens trocadas entre um conjunto de objetos dentro de um contexto para cumprir um propósito. Por exemplo, o primeiro requisito do sistema de compra online, indica que o cliente da Web pode pesquisar, visualizar e comprar produtos na loja virtual.

O diagrama de comunicação ilustrado na Figura 87 descreve a troca de mensagens no processo baseado no caso de uso da Figura 83 que representa o subsistema de compra online.

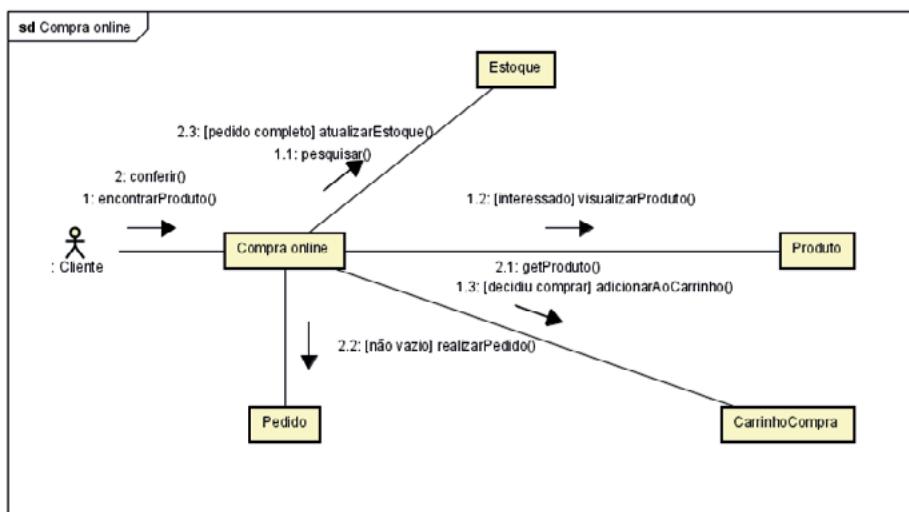


Figura 87 - Diagrama de comunicação para o sistema de compras online / Fonte: o autor.

Ao analisar o diagrama de comunicação na Figura 87, você percebe que a comunicação começa com a mensagem `encontrarProduto()`. Esta mensagem é iterativa,

ou seja, pode ser repetida um número não especificado de vezes. O cliente pesquisa o estoque de produtos e, se estiver interessado em algum dos produtos disponíveis, pode ver a descrição do produto por intermédio da mensagem `visualizarProduto()`. Se o cliente decidir comprar, ele pode adicionar o produto ao carrinho de compras chamando a mensagem `adicionarAoCarrinho()`. Para efetivar a compra a mensagem `conferir()` é invocada e inclui obter a lista de produtos do carrinho de compras, criar pedidos e atualizar o estoque, se o pedido foi concluído.

Para demonstrar as interações entre os diversos objetos de um sistema, o engenheiro de software pode utilizar o diagrama de sequência, também. Segundo Guedes (2018), da mesma forma que no diagrama de comunicação, um diagrama de sequência enfoca um processo, normalmente baseado em um caso de uso.

MODELAGEM DE ESTADOS

Um dos requisitos do sistema é controlar e dar suporte ao ciclo de vida completo da conta do cliente, desde a sua criação até o seu encerramento. Alguns estágios no ciclo de vida da conta, nos quais envolvem condições ou eventos específicos, fazem com que a conta mude seu estado.

A Figura 88 fornece um exemplo de ciclo de vida de conta de usuário no contexto de compras online a partir do diagrama de máquina de estado da UML.

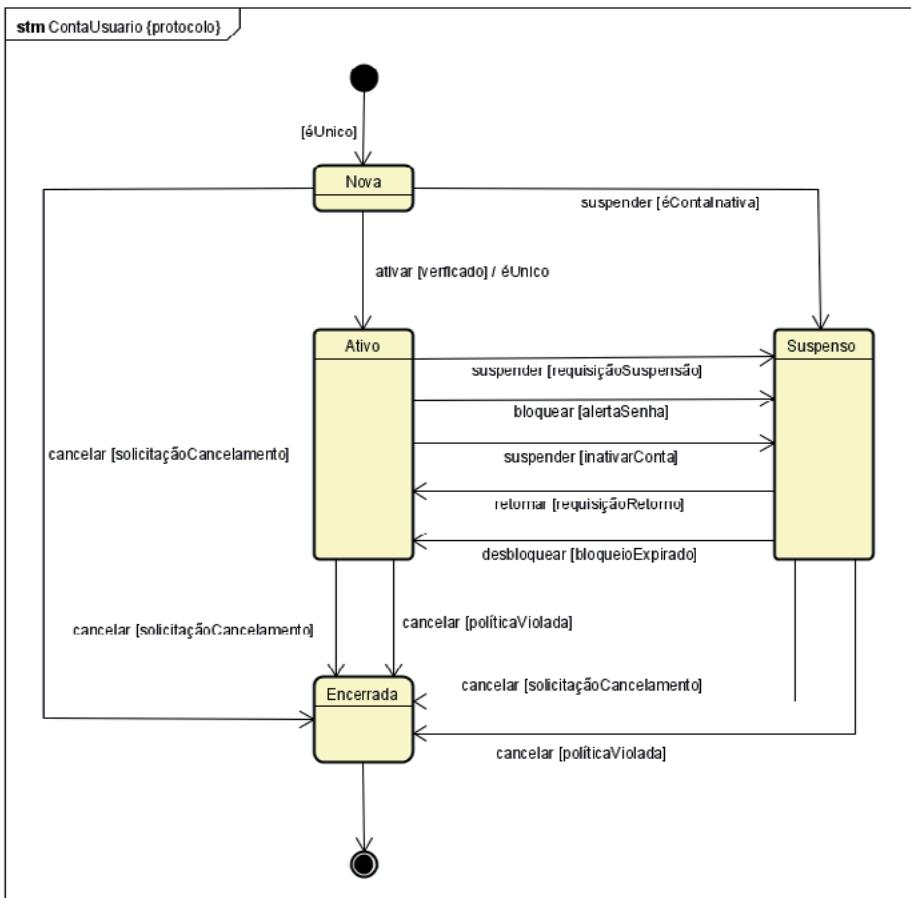


Figura 88 - Diagrama da máquina de estados do protocolo da conta do usuário de compras online
Fonte: o autor.

Para que a conta do usuário seja criada, ela deve atender a alguns requisitos iniciais, como ter o ID do usuário (nome de login) exclusivo. Depois que a conta foi criada é necessário a verificação de e-mail, telefone e/ou endereço. Se a conta não foi verificada durante algum período de tempo predefinido, essa conta pode ser movida para as contas suspensas.

Qualquer conta existente, sejam novas, ativas ou suspensas, pode ser cancelada a qualquer momento por solicitação do cliente. No entanto, para que o cancelamento seja concluído o solicitante não deve possuir saldo pendente.

A conta do usuário pode ser suspensa por motivos de segurança. Se o sistema detectar a intrusão do site bloqueia a conta do usuário por um período de tempo predefinido, se houver várias tentativas de login malsucedidas usando a senha da conta incorreta. Após o tempo limite do bloqueio da conta, a conta é reativada automaticamente.

MODELAGEM DE ATIVIDADES

Como você já estudou anteriormente, o diagrama de atividades é o diagrama com mais ênfase no nível de algoritmo da UML, se assemelha aos fluxogramas utilizados para desenvolver a lógica de programação e determinar o fluxo de controle de um algoritmo (GUEDES, 2018).

A modelagem de atividade enfatiza a sequência e condições para coordenar comportamentos de baixo nível. No contexto do sistema de compra online, a Figura 89 ilustra o processo do pedido de compra.

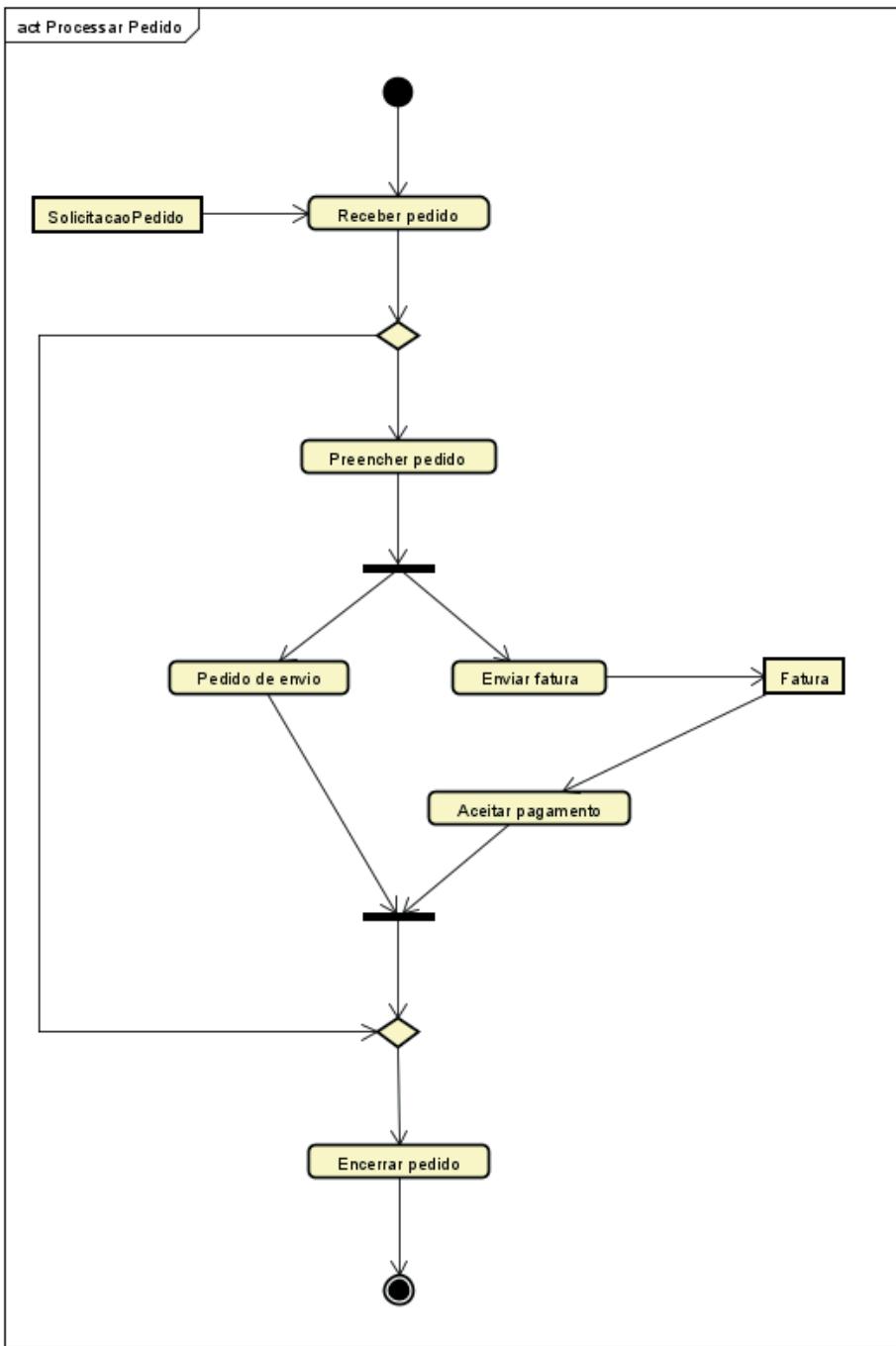


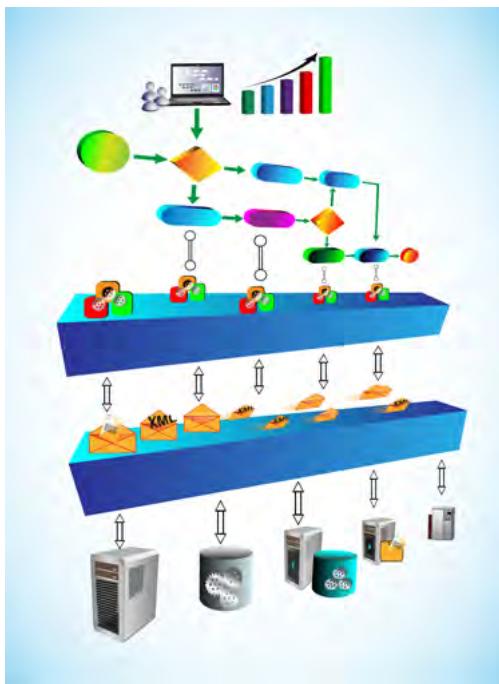
Figura 89 - Diagrama de atividade do processo de pedido de compra / Fonte: o autor.

O processamento de pedidos é um exemplo de atividade do fluxo de negócios de uma loja virtual. O pedido solicitado é o parâmetro de entrada da atividade. Após o pedido ser aceito e todas as informações solicitadas preenchidas, o pagamento é aceito e o pedido é enviado.



SAIBA MAIS

Observe que o fluxo de negócios apresentado na Figura 10 permite o envio do pedido antes do envio da fatura ou da confirmação do pagamento. Outro ponto importante dessa solução é o não uso de partições. As partições deixam mais claro quem é o responsável por realizar cada ação específica. Todas essas questões dependem da regra de negócio. O fluxo de atividades deve explicitar a regra de negócios.



MODELAGEM ARQUITETÔNICA

Você deve se dedicar em representar a arquitetura de software, pois ela facilitará a comunicação entre as partes interessadas no desenvolvimento do sistema. Além disso, a arquitetura possibilita compreender como o sistema será estruturado e como seus componentes trabalham juntos. Além disso, Pressman (2011) afirma que você poderá aproveitar o modelo arquitetônico construído para ser aplicado em projetos de outros sistemas.

Nesta seção vamos considerar a construção de três diagramas da UML para representar a modelagem arquitetônica do sistema de compras online. O diagrama

de objeto para o controlador de *login* de compras online e o diagrama de componentes para a aplicação web.

CONTROLE DE LOGIN DA APLICAÇÃO WEB

O diagrama de classes representado na Figura 90 é um modelo estático do controle de login do usuário, porém, você pode criar diagramas de objetos para representar e simular possíveis situações pelas quais os objetos das classes passarão (GUEDES, 2018).

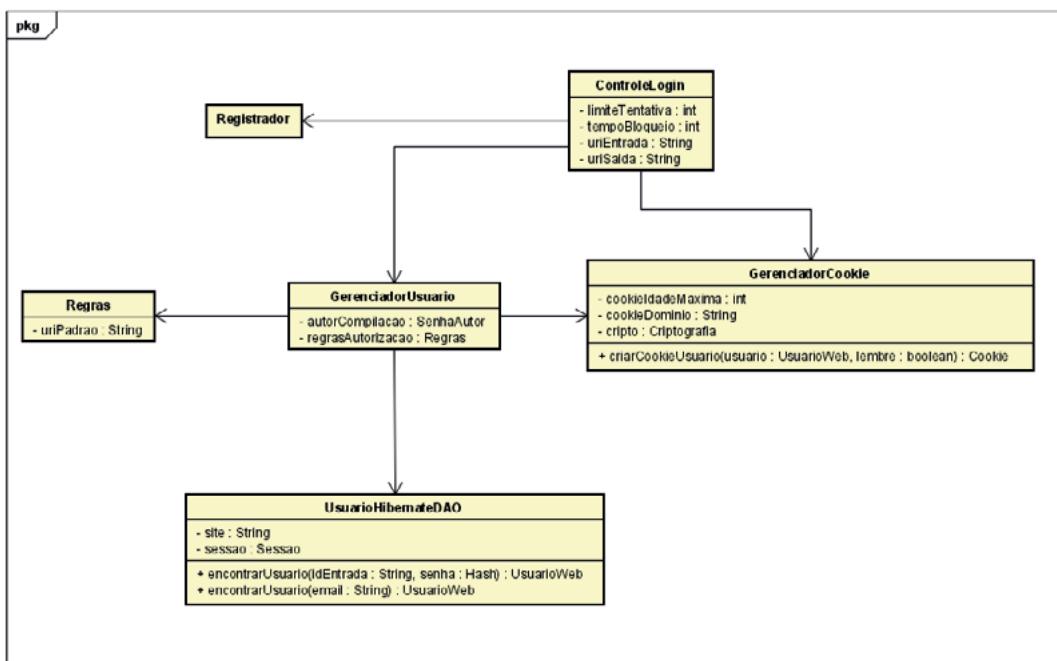


Figura 90 - Diagrama de classes para o controle de login do usuário / Fonte: o autor.

A Figura 91 ilustra alguns objetos em tempo de execução relacionados ao processo de login do usuário da web. O Ctrl de login da instância de classe do Controlador de Login possui várias entradas com recursos estruturais e especificações de valor correspondentes.

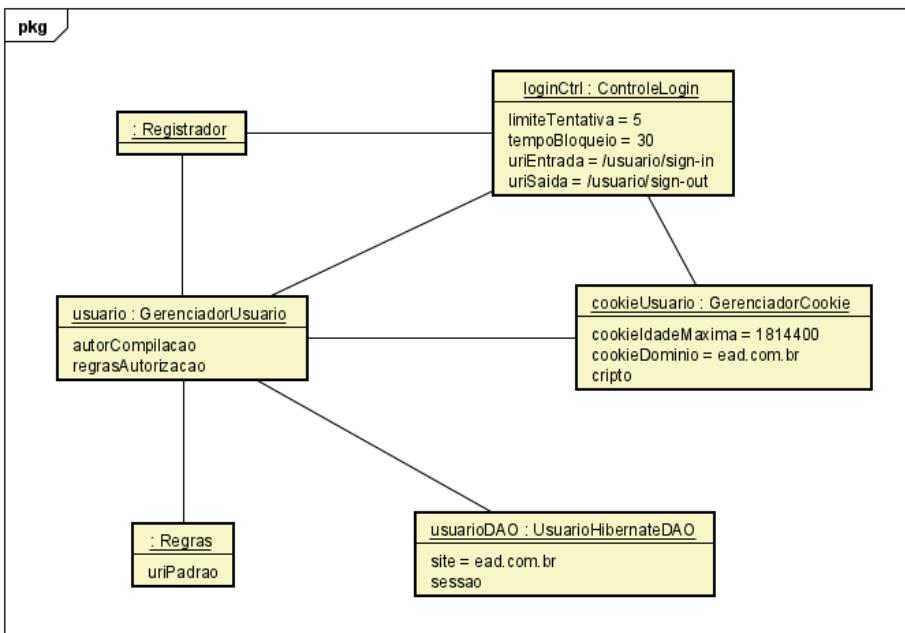


Figura 91 - Diagrama de classes para o controle de login do usuário / Fonte: o autor.

A instância de controle de *login* está associada a instâncias do gerenciador de usuários, gerenciador de cookies e registrador. As instâncias de controle de *login*, gerenciador de usuário e usuário Hibernate DAO (*Data Access Object*, objeto de acesso a dados) compartilham uma única instância do Registrador.

SAIBA MAIS



Uma aplicação com acesso a dados geralmente utiliza o padrão DAO, que abstrai e encapsula a origem dos dados, fazendo a separação das responsabilidades. No contexto da separação da aplicação em camadas, a camada de interface não deve conhecer as particularidades do acesso a dados. Para isso é necessário criar uma camada para realizar as tarefas relacionadas ao acesso a dados. Essa camada tem a responsabilidade de desacoplar o código de acesso e persistência dos dados da lógica da aplicação.

Existem três vantagens em se utilizar este padrão. A primeira está associada à organização do código. A segunda está relacionada a facilidade de manutenção e compreensão do código. A terceira é a possibilidade da reutilização em outras aplicações. Neste padrão, todo o acesso a dados deve ser feito por intermédio das classes DAO, responsáveis por cada objeto de domínio da aplicação e pelas operações CRUD (criar, recuperar, atualizar e excluir) no domínio. Outras transações e sessões devem ser tratadas fora do DAO.

DIAGRAMA DE COMPONENTES DE COMPRAS ONLINE

Um sistema, subsistema, componentes, classes internas de um componente individual representam um componente lógico ou físico, como código-fonte, arquivos de ajuda, bibliotecas, arquivos executáveis e assim por diante.

Segundo Guedes (2018), o objetivo do diagrama de componentes é documentar e identificar os componentes utilizados no desenvolvimento de sistemas baseado em componentes. É útil no gerenciamento de configuração e mudanças, com o objetivo de modelar os módulos de software, versões de bibliotecas, entre outros, para produzir uma versão específica de um sistema.

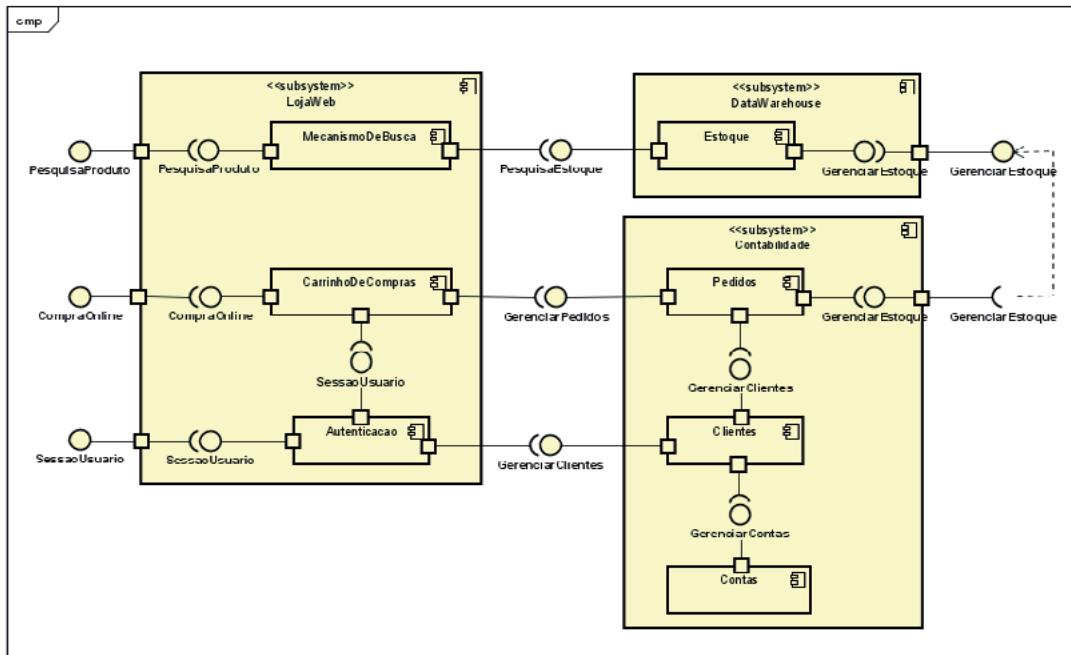


Figura 92- Diagrama de componentes para o sistema de compra *online* / Fonte: o autor.

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

Considerando o sistema de compras online, o diagrama ilustrado na Figura 92 mostra a visão da estrutura interna (lógica) dos subsistemas LojaWeb, DataWarehouse e Contabilidade, que se relacionam.

O subsistema LojaWeb contém três componentes relacionados às compras *online*, mecanismo de pesquisa, carrinho de compras e autenticação. O componente do mecanismo de pesquisa permite pesquisar itens por intermédio da interface fornecida PesquisaProduto e usa a interface necessária PesquisaEstoque fornecida pelo componente Estoque. O componente carrinho de compras usa a interface GerenciarPedidos fornecida pelo componente Pedidos durante a conferência. O componente de autenticação permite aos clientes criar uma conta, entrar ou sair e vincular o cliente a alguma conta.

O subsistema Contabilidade fornece duas interfaces, GerenciarPedidos e GerenciarClientes. Os conectores de delegação ligam estes contratos externos do subsistema à realização dos contratos por componentes de Pedidos e Clientes.

ARQUITETURA DO APLICATIVO WEB DE COMPRAS ONLINE

Ao desenvolver o diagrama de implantação você deve enfocar a organização da arquitetura física sobre a qual o software será implantado e executado em termos de hardware necessários para o sistema, e como os módulos do sistema serão executados nos servidores (GUEDES, 2018).

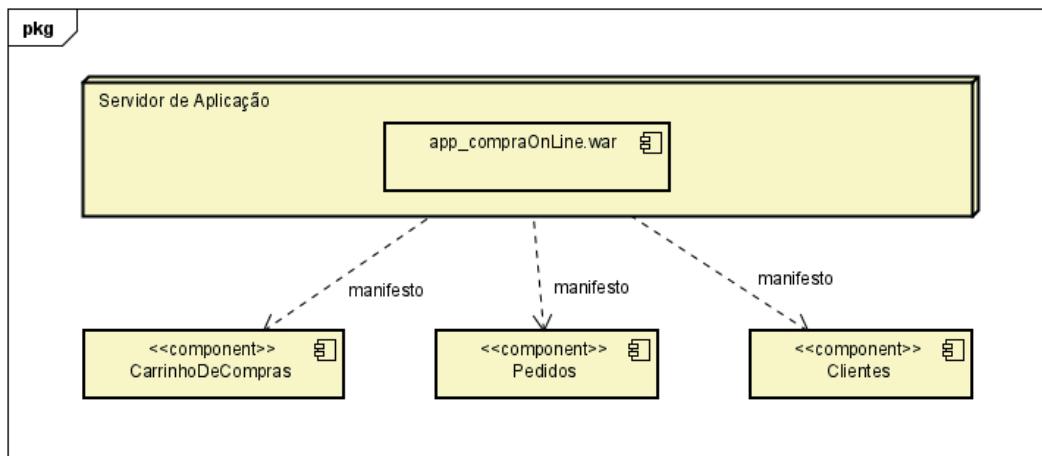


Figura 94 - Diagrama de manifestação para a aplicação web / Fonte: o autor.

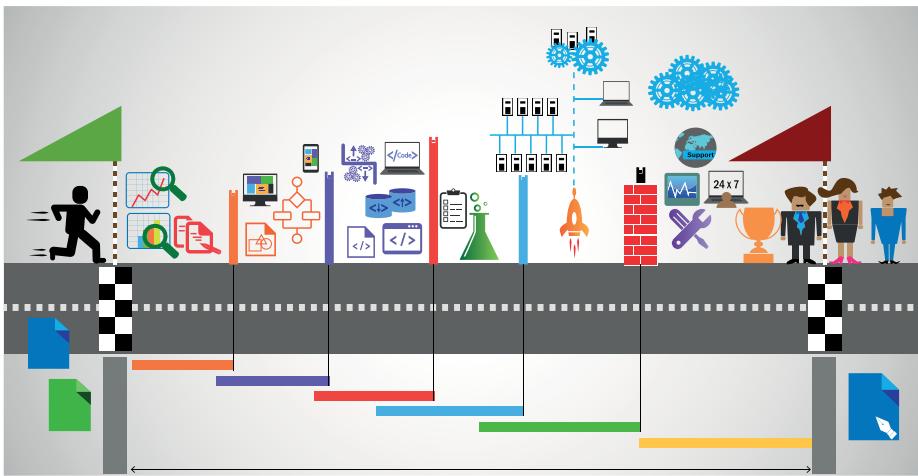
O diagrama de implantação ilustrado na Figura 94 mostra os artefatos e a estrutura interna dos artefatos. Um artefato, segundo Guedes (2018), é um elemento do mundo real, que deve ser implementado em um nó. Um artefato é muitas vezes uma “manifestação” no mundo real de um componente.

REFLITA



Reparou a extensão do artefato que representa a nossa aplicação web? Pois então, um arquivo WAR (Web application ARchive, arquivo de aplicação web) é um conjunto de recursos que constituem uma aplicação web.

Se por um lado os arquivos WAR facilitam desenvolvimento, teste e implantação, por outro lado qualquer alteração não pode ser feita durante o tempo de execução e requer reimplantar o arquivo WAR inteiro.



PROJETO, IMPLEMENTAÇÃO, TESTE E EVOLUÇÃO DO SOFTWARE

O ciclo de desenvolvimento de software passa pelas fases de análise, projeto, implementação, testes, implantação e manutenção. Independente do processo de desenvolvimento escolhido, o ciclo de vida do software é sempre o mesmo.

Nesta seção vamos discutir sobre o projeto de implementação de software. Considerando todas as etapas anteriores de modelagem e a arquitetura do software de uma loja *online*. Na implementação vamos abordar assuntos importantes, como o reuso e o gerenciamento de configuração, que no caso de processos incrementais implica no versionamento do software. Importante abordar os objetivos do teste de software no que diz respeito a qualidade do software. A próxima etapa é colocar o software em produção e, nesse instante, invariavelmente, a necessidade de mudanças para que o sistema continue tendo utilidade.

PROJETO DE IMPLEMENTAÇÃO DE SOFTWARE

O processo de projeto varia de acordo com o nível de detalhamento do modelo. O detalhamento depende do tipo de sistema que você está desenvolvendo. Ao modelar utilizando a UML você vai se deparar com dois tipos, segundo Sommerville (2011):

1. Modelos estruturais: classes e relacionamentos.
2. Modelos dinâmicos: interações entre os objetos do sistema.

No sistema no qual estamos modelando é útil acrescentar a arquitetura de modelos de subsistemas. Esta representação, segundo Sommerville (2011), trata as classes como subsistemas, divididos em pacotes.

Para iniciar a implementação da aplicação web, representada pelo artefato app_compraOnLine.war, vamos desenvolver um diagrama de implantação.

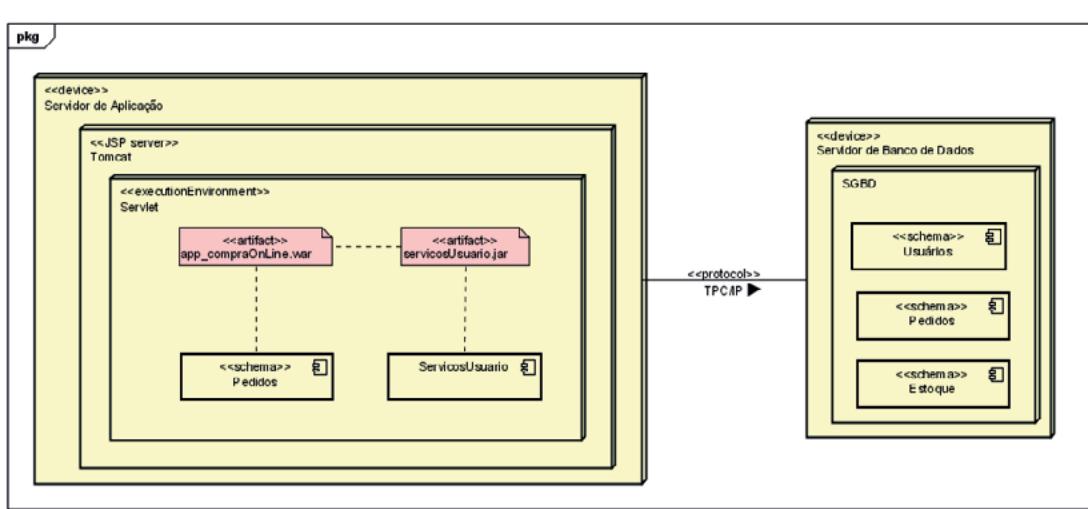


Figura 95- Diagrama de implantação para a aplicação web / Fonte: o autor.

A Figura 95 ilustra a implementação no *container* Servlet/JSP que faz parte do servidor da web Apache Tomcat. O componente de manifestos dos artefatos app_compraOnLine.war para pedidos online, e outro artefato que manifesta o componente de serviços do usuário.

O servidor de aplicação é um computador que se comunica com o servidor da base de dados, que é outro computador, via protocolo TCP/IP.

A partir desse instante já sabemos todas as tecnologias envolvidas na implementação. Então vamos implementar.

IMPLEMENTAÇÃO DE SOFTWARE

A implementação é a fase mais aguardada do ciclo de vida do desenvolvimento de software, pois ela resulta em um programa ou a customização de um programa. Nesta etapa considera-se toda a base tecnológica para a solução ou produto de software.

A modelagem de software acontece independentemente da tecnologia de implementação. Quando se trata de arquitetura de software a tecnologia já passa a ser considerada parcialmente. Mas é na implementação que as estruturas de dados, o banco de dados, o paradigma de programação, entre outros aspectos entram em ação de fato.

Vamos, então, abordar de alguns desses assuntos a partir desse instante:

PARADIGMAS DE PROGRAMAÇÃO

Um paradigma é uma forma particular de abordar um problema e de formular uma solução para o mesmo. No contexto das linguagens de programação podemos ter paradigmas de programação imperativo, funcional, lógico, orientado a objetos e estruturado. Os mais comuns dentro do ambiente acadêmico atualmente são programação estruturada e orientada a objetos.

- **Paradigma estruturado:** a programação estruturada é uma programação modular baseada na criação de subrotinas e funções. As principais linguagens de programação estruturadas são C, Pascal e Cobol, por exemplo.
- Um programa modular consiste em resolver um problema dividindo-o em subproblemas. Essa técnica ajuda na legibilidade e compreensão da solução. No entanto, como esta técnica se baseia em dados e processos, a manutenção se torna mais trabalhosa, principalmente quando há interdependência entre módulos e configura muitas alterações nas funções relacionadas.

- **Paradigma orientado a objetos:** a programação orientada a objetos é consequência da programação estruturada, que consiste na composição e interação de diversos objetos de software. Essa composição acontece por intermédio de relacionamentos e a interação ocorre em razão da troca de mensagens entre os objetos envolvidos na solução. Java, C# e C++ são alguns exemplos de linguagens de programação orientada a objetos.
- Uma das vantagens deste paradigma de programação é a possibilidade de alterar um módulo sem ter que modificar os outros módulos envolvidos. A programação orientada a objetos, no entanto, ainda pode ser um desafio para quem tem larga experiência no desenvolvimento estruturado.

Durante a sua formação como engenheiro de software você terá disciplinas de programação voltadas para o desenvolvimento estruturado e orientado a objetos, além de conhecer os demais paradigmas de programação.

ESTRUTURAS DE DADOS

Quando você iniciou a programação basicamente aprendeu três estruturas fundamentais: sequência, decisão e repetição. Depois passou a explorar estruturas elementares de dados: vetores, matrizes, registros e ponteiros.

Na continuidade dos estudos vai se deparar com as estruturas lineares, que envolvem listas, pilhas e filas. São estruturas similares, porém com regras específicas, e servem para armazenar em memória vários registros obtidos de alguma fonte de dados. Com esses dados em memória é possível manipulá-los e, então, atualizar a fonte de dados.

Dentro do contexto das estruturas de dados é possível, ainda, estudar a prática de recursividade, árvores, grafos e as diversas técnicas de ordenação.

A definição de um registro em uma estrutura de dados configura um tipo abstrato de dados (TAD) que é uma similaridade (limitada) da implementação de objetos em uma linguagem de programação que não incorpora os conceitos de orientação a objetos, como herança, polimorfismo e visibilidade.



REFLITA

Você já ouviu falar em tipo de dados, tipo abstrato de dados e estruturas de dados. São termos parecidos, porém com significados diferentes.

O tipo de dados configura o conjunto de valores que uma variável, constante ou função podem assumir ao longo da execução de um programa. Uma estrutura de dados, por sua vez, pode incluir informações com diferentes tipos de dados. Já o tipo abstrato de dados é definido como um modelo matemático por meio de um par contendo o conjunto de valores e o conjunto de operações sobre esses valores.

SISTEMAS DE BANCO DE DADOS

Não existe um sistema de informação sem a utilização de dados. Um dado é criado, armazenado de forma persistente em uma base de dados para que em determinado momento possa ser recuperado, modificado e excluído. Os sistemas de banco de dados possuem um conjunto de propriedades que garantem as transações em banco de dados, chamada ACID (acrônimo de Atomicidade, Consistência, Isolamento e Durabilidade).

Ao desenvolver um sistema você vai se deparar com a possibilidade de bancos de dados relacionais, não-relacionais, distribuídos, geográficos, orientados a objetos, em nuvem, entre outros. A escolha do tipo de banco de dados vai depender do tipo de aplicação e da solução projetada.

- **Banco de dados relacional:** os bancos de dados relacionais surgiram no final dos anos 1970 e os tipos de bancos de dados dominantes para as aplicações mais comuns envolvendo banco de dados. As técnicas de programação usadas para acessar os sistemas de banco de dados relacionais envolvem a linguagem SQL. Por intermédio da linguagem SQL é possível definir restrições, consultas, visões e funções. As linguagens de programação se conectam a um banco de dados relacional pelos protocolos ODBC (*Open DataBase Connectivity*) e JDBC (*Java DataBase Connectivity*).
- **Banco de dados não-relacional:** o crescimento do volume de dados, principalmente na Web, possibilitou muitas inovações na área de gerenciamento

de dados, como os bancos de dados não-relacionais. Os bancos de dados não-relacionais consideram os requisitos não funcionais de escalabilidade e disponibilidade. Os bancos de dados não-relacionais também são chamados NoSQL, que significa “Não apenas SQL”. Você pode encontrar bancos de dados não-relacionais orientados a documentos, a chave-valor, a grafos, entre outros.

A opção entre um banco de dados ou outro varia muito em função do problema a ser solucionado. Geralmente a escolha de um banco de dados NoSQL está relacionado a um cenário onde a escalabilidade é a principal característica.

REUSO

Você vai se deparar com fábricas de software especializadas em desenvolver sistemas que possuam funcionalidades comuns e variáveis, mas que pertencem à mesma família de sistemas. Estou falando de linha de produtos de software, que envolve requisitos, arquitetura e implementações de componentes para uma família de sistemas, dos quais os produtos são derivados e configurados.

Na modelagem de linha de produtos de software, cada classe pode ser categorizada de acordo com sua característica de reutilização. Segundo Gomaa (2011), a UML utiliza estereótipos diferentes para representar uma característica de reutilização e para representar o papel desempenhado pelo elemento de modelagem. O papel que uma classe desempenha no aplicativo e a característica de reutilização são coisas distintas.

Um software pode ter a capacidade de ser reutilizado. A reutilização tradicional de software inclui uma biblioteca de componentes de código reutilizáveis, como por exemplo uma sub-rotina estatística.

Porém, ao invés de reutilizar um componente individual você pode reutilizar todo um projeto ou subsistema. Essa abordagem consiste em reaproveitar os componentes e suas interconexões, ou seja, reutilizar a estrutura de controle do aplicativo.

Para Gomaa (2011), a abordagem de reutilização da arquitetura é propícia em uma arquitetura de linha de produto de software que captura explicitamente

a semelhança e a variabilidade na família de sistemas que constitui a linha de produto. Neste contexto, se o paradigma de desenvolvimento for orientado a objetos, a semelhança da linha de produto é descrita em termos de classes ou componentes comuns, e a variabilidade da linha de produto é descrita em termos de classes ou componentes opcionais ou variantes.

GERENCIAMENTO DE CONFIGURAÇÃO

A palavra mudança está presente no desenvolvimento de software, ela pode acontecer já na fase de levantamento de requisitos. Para garantir o controle é essencial exercer o gerenciamento de configuração de mudanças. Sommerville (2011) apresenta três atividades fundamentais:

1. Gerenciamento de versões
2. Integração de sistemas
3. Rastreamento de problemas

Existem várias ferramentas que permitem a equipe de desenvolvimento controlar as diferentes versões de componentes de software, identificando inclusive o responsável pela alteração e escrita do código. Da mesma forma, é possível controlar a versão dos componentes que estão em uso no sistema. Por fim, nos casos em que o usuário informa algum tipo de inconsistência ou erro em algum componente funcional do sistema.

TESTES DE DESENVOLVIMENTO

A fase de testes vem ganhando cada vez mais espaço e importância no desenvolvimento de software. Um teste bem feito é a garantia da qualidade do software que está sendo produzido.

As técnicas para teste de software são independentes do paradigma de desenvolvimento utilizado pela equipe de desenvolvimento. A seguir abordaremos as principais técnicas para se testar software:

- **Caixa-branca:** testa a lógica do software, ou seja, o código-fonte. Esse tipo de teste de software está na fase de teste de unidade e de integração.
- **Caixa-preta:** testa a funcionalidade do software, ou seja, o comportamento do software diante das entradas e as saídas produzidas pelo sistema. Esta técnica está presente nas principais etapas antes de se colocar o sistema em operação, como no teste unitário, teste de integração, teste de sistema e teste de aceitação.

Um caso de teste precisa de um *script* ou roteiro de teste, que consiste em um procedimento de teste e nos dados do teste. A partir da realização do caso de teste é possível comparar os resultados esperados com os resultados obtidos.

PROCESSOS DE EVOLUÇÃO

É muito importante que você tenha em mente que o processo de software continua mesmo após o sistema ser entregue. Pois é justamente quando o sistema está em produção que vão surgir as principais solicitações de mudanças.

As mudanças do software em produção caracterizam o processo de evolução do software. Segundo Sommerville (2011), uma solicitação de mudança pode ser feita por necessidade de correção, adaptação tecnológica de hardware e software, melhoria no desempenho ou especificidades funcionais, como a adaptação a uma nova legislação.

Geralmente a solicitação é feita ao departamento de atendimento ao cliente, que abre um chamado e encaminha para a equipe de desenvolvimento ou negócios, dependendo do tipo de mudança identificada.

Toda solicitação de mudança implica no processo de análise e segue o caminho até a implantação de uma nova versão do software, contemplando a alteração do cliente.

CONSIDERAÇÕES FINAIS

Enfim, nesta última unidade, concluímos a nossa disciplina, e você aplicou todos os conceitos aprendidos na resolução de um problema prático. No contexto geral podemos afirmar que o diagrama de casos de uso é fundamental para entendermos o problema em questão, e o diagrama de classes que contém todas as informações necessárias para a programação, são essenciais para modelar um software. Porém podemos estender a modelagem de software em outras visões, como as interações para descobrir as operações por intermédio da troca de mensagens entre os objetos, representados pelos diagramas de sequência e comunicação; também podemos ter uma visão comportamental do software por intermédio do diagrama de atividades ou máquina de estados.

Numa abordagem ágil o envolvimento do cliente e o seu comprometimento com o projeto garantem o sucesso e a qualidade do software. O processo de desenvolvimento consiste em incrementos. Na fase de análise ocorre a aquisição e aprovação de requisitos. A partir das necessidades do cliente se define na fase de projeto os modelos e a equipe de programação deve priorizar os requisitos estratégicos para implementá-los. Com isso, uma primeira versão do sistema é produzida e já deve ser apresentada para o cliente em uma segunda iteração. E, assim por diante, até a entrega do produto final.

Uma discussão recorrente é o quanto de documentação e qual o tipo de documentação deve ser produzido. Ninguém questiona a necessidade de ter documentação. Cada modelo de processo de software tem um perfil para se documentar. O processo de testes, por exemplo, abrange testes unitários e testes de aceitação, e tem por objetivo documentar a análise dos resultados obtidos ao longo do processo.

Concluímos esta unidade fazendo um estudo geral sobre as demais etapas do ciclo de vida do desenvolvimento de software. As próximas disciplinas te darão a amplitude e os conhecimentos da carreira em Engenharia de Software para o desenvolvimento de software.

ATIVIDADES



1. O sistema apresentado no desenvolvimento da Unidade V é um sistema do tipo cliente-servidor. Nesse contexto, assinale a alternativa correta que define um servidor.
 - a) Um subsistema que responde a solicitações de clientes.
 - b) Um sistema de hardware e software que atende clientes.
 - c) Um serviço que pode receber solicitações de vários clientes.
 - d) Um subsistema que faz solicitações e espera pelas respostas.
 - e) Um sistema de hardware e software que fornece um ou mais serviços para vários clientes.

2. “Entidades são coisas que possuem atributos e relacionamentos. Os atributos e relacionamentos são persistidos em uma base de dados relacional. No paradigma de programação orientado a objetos devemos adicionar comportamentos às entidades” (MEDEIROS, 2013).
Analise as afirmações a seguir a respeito de classes de entidade:
 - I) A classe possui vários atributos identificadores.
 - II) Uma classe pode se relacionar com várias outras classes.
 - III) A classe possui um atributo identificador.
 - IV) A chave possui um atributo que a concatena a outra classe.

É correto o que se afirma em:

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

ATIVIDADES



3. No estudo de caso desta unidade modelamos as interações por intermédio do diagrama de comunicação. Analise as alternativas e assinale Verdadeiro (V) ou Falso (F) para a que representa o diagrama de comunicação.

- () Retrata a sequência de objetos externos que se comunicam entre si.
- () Representa as classes e seus relacionamentos.
- () Demonstra os objetos de software e a sequência de suas interações.
- () Auxilia na descoberta das operações entre os objetos de software.

Assinale a alternativa correta:

- a) V; V; V; V.
- b) V; V; F; F.
- c) V; F; V; F.
- d) F; F; V; V.
- e) F; F; F; F.

4. Dentre os diagramas que auxiliam o projeto de software, um deles é muito útil para representar lógica comportamental, que é uma excelente ferramenta para modelagem de fluxos de trabalho e de processos. A notação em questão é o diagrama de:

- a) Classes.
- b) Atividades.
- c) Implementação.
- d) Máquina de estados.
- e) Estruturas compostas.

5. O diagrama que é utilizado pela UML para modelar as interações funcionais entre os usuários e o sistema é denominado de:

- a) Pacotes.
- b) Interação.
- c) Implantação
- d) Caso de uso.
- e) Componentes.



O USO DA UML PELAS EMPRESAS DE DESENVOLVIMENTO DE SOFTWARE

Cada vez mais são necessários produtos de software mais rápidos, mais complexos e melhores para suprir as necessidades da sociedade em geral. O processo de desenvolvimento de software conta com algumas etapas entre planejamento, desenvolvimento e implementação. A modelagem é uma parte central de todas as atividades que levam à implantação de um bom software pela possibilidade de, por meio dela, esboçar como é ou como se deseja que o sistema seja, orientar o desenvolvimento, documentar as decisões tomadas, entre outros. A Unified Modeling Language (UML) surgiu com o intuito de unificar as modelagens e se tornou uma linguagem universal no quesito de modelagem de software, sendo a linguagem mais utilizada em modelagem atualmente.

Contudo, no contexto atual, com o crescimento das metodologias ágeis de desenvolvimento, pouco se sabe sobre o real grau de utilização da UML no cotidiano das empresas da indústria de desenvolvimento de software.

No contexto das indústrias de software a frequência com que os diagramas da UML são utilizados segue a seguinte ordem, respectivamente: diagrama de casos de uso, diagrama de classes, diagrama de atividades. Os diagramas de objeto, sequência, comunicação, componentes e implementação são usados em situações mais específicas.

Entre os principais motivos no uso dos diagramas da UML estão a facilidade de entendimento do projeto e a forma com que isso impacta no desenvolvimento posterior, ainda mais quando se trata de um sistema mais complexo. Por outro lado, a demanda de tempo, o material humano para a elaboração, a diversidade de diagramas existentes, a complexidade e detalhamento necessários para os diagramas são fatores desmotivadores para a utilização da UML.

O mercado, no entanto, mesmo com a tendência de utilização das metodologias ágeis, faz uso considerável da UML.

Fonte: Bento (2020, online).

MATERIAL COMPLEMENTAR



NA WEB

A página oficial do projeto ArgoUML, mantida pela Tigris.org, oferece um conjunto de tutoriais e exemplos que podem ser acessados por qualquer interessado em saber mais sobre a modelagem de software e a linguagem UML®. Ali, é possível também fazer o download das releases da ferramenta e se manter informado sobre as novidades de cada versão.

Aprofunde-se no assunto, acessando a página da Tigris.org pelo endereço disponível em:
<https://argouml-tigris.org.github.io/tigris/argouml/>. Acesso em: 22 abr. 2021.



CONCLUSÃO

Chegamos ao final de mais uma etapa!

Não tenho preocupação em me equivocar ao afirmar que desenvolver software é uma atividade complexa e que grande parte dessa complexidade é inherente a sua própria natureza. Softwares são desenvolvidos por pessoas e exigem, para o funcionamento, um grande número de “estados” possíveis, além disso, estão sujeitos às pressões constantes por mudanças, seja por solicitação do cliente ou por exigência da própria evolução tecnológica. Nesse sentido, percebemos a importância do projeto para o desenvolvimento do software, e a modelagem é parte essencial, senão fundamental, para qualquer projeto de software.

Vimos na unidade I os principais conceitos referentes à atividade de modelagem de software e observamos que a modelagem deve estar relacionada com o processo de engenharia de requisitos, garantindo uma ponte entre as etapas de definição do sistema e projeto, que representam as atividades mais importantes do processo de desenvolvimento de software.

A unidade II propôs a análise dos principais tipos de modelos utilizados nos projetos de desenvolvimento de software. Percebemos que diferentes tipos de modelos são necessários porque existem várias formas de se observar o problema em desenvolvimento; o conjunto de visões tem o objetivo de propiciar formas diferentes de se observar a mesma coisa, sempre considerando a ótica do interessado ou da necessidade naquele momento.

A orientação a objetos está intimamente ligada ao desenvolvimento de software, por isso estudamos também os principais conceitos desse paradigma na unidade III, juntamente com a apresentação da linguagem UML®, em que observamos suas especificações e diagramas utilizados na modelagem de software.

A utilização de ferramentas automatizadas garante inúmeros benefícios, como aumento na qualidade do produto final, aumento da produtividade, redução das atividades de programação, agilidade no retrabalho, diminuição da manutenção e do esforço para isso e redução de custos. O objetivo da unidade IV foi pontuar as vantagens da utilização de ferramentas CASE para a modelagem de software e apresentar algumas delas.

Encerramos nossos estudos aplicando todos os conceitos em um estudo de caso. O estudo de caso utilizado foi retirado do livro Modelagem orientada a objeto com técnicas de análise, documentação e projeto de sistema, da editora Futura, e nos garantiu uma visão prática dos conceitos apresentados.



REFERÊNCIAS

- ATALLA, F. Por que os Projetos de TI Fracassam? **Instituto de Educação Tecnológica** [online]. Disponível em: <http://www.techoje.com.br/site/techoje/categoria/detalhe_artigo/1532>. Acesso em: 04 set. 2015.
- BARCELAR, R. R. Modelagem de Sistemas Orientada a Objetos com UML. Disponível em: <http://ricardobarcelar.com.br/aulas/eng_sw/mod3-uml.pdf>. Acesso em: 22 out. 2015.
- BEZERRA, E. **Princípios de análise e projeto de sistemas com UML**. RJ: Elsevier, 2014.
- BRAZ JUNIOR, G. **Estudo de Caso:** Sistema de Caixa Automático. Departamento de Informática. UFMA. Disponível em: <<http://www.deinf.ufma.br/~geraldob/14Es-tudoCaso.pdf>>. Acesso em: 23 out. 2015.
- BENTO, Luiz Henrique Ten Caten **UML:** Um estudo sobre o uso em empresas de desenvolvimento de software em São Carlos - SP e região / Luiz Henrique Ten Caten Bento. – São Carlos – SP, 2020.
- BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **The Unified Modeling Language User Guide**. 2ª Edição. Addison-Wesley: Pearson Education, 2005.
- CAELUM. Apostila Java e orientação a objetos. Capítulo 10 – Interfaces. Disponível em: <<http://www.caelum.com.br/apostila-java-orientacao-objetos/interfaces/>>. Acesso em: 15 set. 2015.
- CARDOSO, C. **UML na prática:** do problema ao sistema. Rio de Janeiro: Ciência Moderna Ltda., 2003.
- CELEPAR INFORMÁTICA DO PARANÁ. Guia para Modelagem de Interações Metodologia CELEPAR. 2009. Disponível em: <<file:///C:/Users/Usuario/Documents/UNICESUMAR/Modelagem%20de%20Software/Material%20de%20apoio%20MS/guia-ModelagemInteracoes.pdf>>. Acesso em: 06 set. 2015.
- CORREIA, C.; TAFNER, M. **Análise orientada a objetos**. 2. ed. Florianópolis: Visual Books, 2006.
- COSTA, C. A. A aplicação da Linguagem de Modelagem Unificada (UML) para o suporte ao projeto de sistemas computacionais dentro de um modelo de referência. **Gestão e Produção**, Departamento de Engenharia Mecânica, Universidade de Caxias do Sul (UCS), v. 8, n. 1, p. 19-36, abr. 2001.
- DEBONI, J. E. Z. **Modelagem orientada a objeto com UML**. Técnicas de análise, documentação e projeto de sistema. São Paulo: Futura, 2003.
- DIAGRAMA de pacotes. UFPR. Disponível em: <<http://www.inf.ufpr.br/silvia/ES/UML/Diagramadepacotes.pdf>>. Acesso em: 23 out. 2015.
- ENGHOLM JUNIOR, H. **Engenharia de software na prática**. São Paulo: Novatec, 2010.



REFERÊNCIAS

- ESPÍNDOLA, E. C. A importância da modelagem de objetos no desenvolvimento de sistemas. **Linha de Código**. Disponível em: <<http://www.linhadecodigo.com.br/artigo/1293/a-importancia-do-modelagem-de-objetos-no-desenvolvimento-de-sistemas.aspx#ixzz3kn4dKfnA>>. Acesso em: 04 set. 2015.
- FERRAMENTAS para Modelagem. **Portal Educação**. 2013. Disponível em: <<https://www.portaleducacao.com.br/informatica/artigos/27187/ferramentas-para-modelagem>>. Acesso em: 14 set. 2015.
- GASPAROTTO, H. M. Os 4 pilares da Programação Orientada a Objetos. **Devmedia**. 2014. Disponível em: <<http://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264#ixzz3lxbe0htF>>. Acesso em: 16 set. 2015.
- GOMAA, Hassan. **Software modeling and design**: UML, use cases, patterns, and software architectures. Cambridge University Press, 2011.
- GUEDES, Gilleanes T. A. **UML 2**: uma abordagem prática. 3ª Edição. São Paulo: Novatec, 2018.
- IBM Knowledge Center. Rational Software Architect RealTime Edition 9.1.2. Disponível em: <http://www-01.ibm.com/support/knowledgecenter/SS5JSH_9.1.2/com.ibm.xtools.modeler.doc/topics/cinterfc.html?cp=SS5JSH_9.1.2%2F7-3-1-6-3-0&lang=pt-br>. Acesso em: 16 set. 2015.
- JUNGTHON, G.; GOULART, C. M. **Paradigmas de Programação**. Artigo Publicado pela Faculdade de Informática de Taquara (FIT) – Faculdades Integradas de Taquara - FACCAT. Taquara/RN. Disponível em: <https://fit.faccat.br/~guto/artigos/Artigo_Paradigmas_de_Programacao.pdf>. Acesso em: 15 set. 2015.
- MACHADO, G. M.; OLIVEIRA, J. P. M. de. Modelos de contexto para sistemas adaptativos baseados em modelos de usuário e localização. **Cadernos de Informática**, v. 7, n. 1 (2013). Disponível em: <<http://seer.ufrgs.br/cadernosdeinformatica/article/view/v7n1p1-13>>. Acesso em: 05 set. 2015.
- MACORATTI, J. C. Programação Orientada a Objetos em 10 lições práticas – Parte 07. **Imasters**, 2014. Disponível em: <<http://imasters.com.br/desenvolvimento/visual-basic/programacao-orientada-objetos-em-10-licoes-praticas-parte-07/>>. Acesso em: 23 out. 2015.
- MEDEIROS, Higor. Definindo entidades na Java Persistence API. **DEVMEDIA**, 2013. Disponível em <<https://www.devmedia.com.br/definindo-entidades-na-java-persistence-api/28180>>. Acesso em 21 abr. 2021.
- MODELAR. In: Dicionário Michaelis online. Disponível em: <<http://michaelis.uol.com.br/moderno/portugues/index.php?lingua=portugues=portugues&palavra=modelar>>. Acesso em: 07 out. 2015.
- MODELAR. In: Dicionário online de português. Disponível em: <<http://www.dicio.com.br/modelar/>>. Acesso em: 07 out. 2015.

REFERÊNCIAS

- MORAIS, L. Modelagem em uma Visão Ágil. **Revista Engenharia de Software**, n. 32. DevMedia. Rio de Janeiro, 2011. Disponível em: <http://www.devmedia.com.br/websys.5/webreader.asp?cat=48&artigo=3211&revista=esmagazine_32#a-3211>. Acesso em: 04 set. 2015.
- NAGAO, F. Programação orientada a eventos e lambda function em ASP/VBScript. **Revista Imaster**, São Paulo, 2009. Disponível em: <http://imasters.com.br/artigo/11514/asp/programacao_orientada_a_eventos_e_lambda_function_em_asp-vbscript>. Acesso em: 05 set. 2015.
- NELSON, R. R. Project retrospectives: Evaluating project success, failure, and everything in between. **MIS Quarterly Executive**, [S.I.], v. 4, n. 3, p. 361-372, set. 2005. Disponível em: <<http://misqe.org/ojs2/index.php/misqe/article/view/85>>. Acesso em: 04 set. 2015.
- NUNES, M.; O'NEILL, H. **Fundamental de UML**. 7. ed. Lisboa: Editora FCA, 2000.
- OBJECTS BY DESIGN. UML Products by Product. Disponível em: <http://www.objectsbydesign.com/tools/umltools_byProduct.html>. Acesso em: 18 set. 2015.
- OLIVEIRA, F. G. de; SEABRA, J. M. P. Metodologias de desenvolvimento de software: uma análise no desenvolvimento de sistemas na web. **Periódico Científico**, v. 6, n. 1, 2015. Disponível em: <<http://revista.faculdadeprojecao.edu.br/index.php/Projecao4/article/view/497>>. Acesso em: 18 set. 2015.
- OLIVEIRA, J. A. de; SOUZA, P. P. de; FIGUEIREDO, E. **Uma Avaliação de Ferramentas de Modelagem de Software**. Laboratório de Engenharia de Software (LabSoft) - Universidade Federal de Minas Gerais (UFMG), Belo Horizonte-MG . Disponível em: <<http://labsoft.dcc.ufmg.br/lib/exe/fetch.php?media=smes.pdf>>. Acesso em: 18 set. 2015.
- OLIVEIRA, J. de. 12 reflexões que vão te introduzir ao pensamento de Carl Sagan. **Revista Galileu** [online]. Disponível em: <<http://revistagalileu.globo.com/Ciencia/Espaco/noticia/2015/03/12-reflexoes-que-vao-te-introduzir-ao-pensamento-de-carl-sagan.html>>. Acesso em: 23 out. 2015.
- OMG Unified Modeling LanguageTM (OMG UML). Version 2.5. OMG Document Number formal/2015-03-0. Disponível em: <<http://www.omg.org/spec/UML/2.5>>. Acesso em: 15 set. 2015.
- PARADIGMA. In: Dicionário Michaelis online. Disponível em: <<http://michaelis.uol.com.br/moderno/portugues/index.php?lingua=portugues=portugues&palavra=paradigma>>. Acesso em: 23 out. 2015.
- PIMENTEL, A. R. **Projeto de Software Usando a UML** (Apostila). Universidade Federal do Paraná. Disponível em: <<http://www.inf.ufpr.br/andrey/ci221/apostilaUml.pdf>>. Acesso em: 15 set. 2015.
- PRESMAN, R. S. **Engenharia de Software**. Porto Alegre: McGrawHill, 2010.



REFERÊNCIAS

- RODRIGUES, A. F. **Um guia para a criação de modelos de desenho de software no Praxis Synergia.** 2007. 140 f. Dissertação (Mestrado em Ciências da Computação) - Universidade Federal de Minas Gerais. Instituto de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação, Belo Horizonte. Disponível em: <<https://www.dcc.ufmg.br/pos/cursos/defesas/871M.PDF>>. Acesso em: 29 ago. 2015.
- SILVA, C. F. da. **Construção e Realidade nas Imagens dos Livros Didáticos de Física:** implicações e aplicações no ensino da física. 2008. 118 f. Dissertação (Mestrado em Ensino de Física) – Pontifícia Universidade Católica de Minas Gerais, Belo Horizonte. Disponível em: <http://www.biblioteca.pucminas.br/teses/EnCiMat_SilvaCF_1.pdf>. Acesso em: 03 set. 2015.
- SILVA, P. C. B. da. Utilizando UML: Diagrama de Atividade. SQL Magazine, 66. Disponível em:<<http://www.devmedia.com.br/artigo-sql-magazine-66-utilizando-uml-diagrama-de-atividade/13577>>. Acesso em: 23 out. 2015.
- SOMMERVILLE, I. **Engenharia de Software.** São Paulo: Pearson Prentice Hall, 2011.
- SOUZA, D. Introdução ao IBM Rational Software Architect. **Devmedia**, 2012. Disponível em: <<http://www.devmedia.com.br/introducao-ao-ibm-rational-software-architect/26748>>. Acesso em: 18 set. 2015.
- SPARX SYSTEMS. Creating Strategic Models with Enterprise Architect. 2010. Disponível em: <http://www.sparxsystems.com.au/downloads/quick/strategic_modeling_with_enterprise_architect.pdf>. Acesso em: 23 out. 2015.
- SPINOLA, R. O. Projeto de software utilizando UML. **SQL Magazine:** 2007. Versão online disponível em: <<http://www.devmedia.com.br/artigo-sql-magazine-13-projeto-de-software-utilizando-uml/5640#ixzz3mLRWZXYz>>. Acesso em: 19 set. 2015.
- THE UNIFIED Modeling Language. Uml-diagrams. Disponível em: <<http://www.uml-diagrams.org/>>. Acesso em: 18 set. 2015.
- TUCKER, A. B.; NOONAN, R. E. **Linguagens de Programação:** Princípios e Paradigmas. São Paulo: MacGraw Hill, 2008.
- UFCG. Exemplo. Disponível em: <http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/uml/diagramas/classes/images/especificacao_exemplo_banco.GIF>. Acesso em: 23 out. 2015.
- UFCG. Diagrama de Atividades. UML. Disponível em: <http://www.dsc.ufcg.edu.br/~sampaio/cursos/2007.1/Graduacao/SI-II/Uml/diagramas/atividades/diag_atividades.htm>.
- UML – Linguagem de Modelagem Unificada. Disponível em: <<http://pt.slideshare.net/Ridlo/uml-1858376>>. Acesso em: 23 out. 2015.
- UML – Linguagem de Modelagem Unificada. ETELG. Disponível em: <http://www.etelg.com.br/paginaete/downloads/informatica/apostila_uml.pdf>. Acesso em: 23 out. 2015.

REFERÊNCIAS

VALLE, F. A. Ferramentas CASE e qualidade dos dados: O paradigma da boa modelagem. **Devmedia**, 2007. Disponível em: <<http://www.devmedia.com.br/ferramentas-case-e-qualidade-dos-dados-o-paradigma-da-boa-modelagem/6905#ixzz3mDrVBvu4DevMedia>>. Acesso em: 15 set. 2015.

VIEIRA, V.; TEDESCO, P.; SALGADO, A. C. **Modelos e processos para o desenvolvimento de sistemas sensíveis ao contexto**. Biblioteca Digital da Universidade Federal da Bahia. Disponível em: <http://homes.dcc.ufba.br/~vaninha/context/2009_TextoJAI_Final.pdf>. Acesso em: 03 set. 2015.



GABARITO

UNIDADE I

1. Modelagem de software é a atividade de construir modelos que representam a estrutura e o comportamento de um software.
2. - Descrever o que o cliente exige; - Estabelecer a base para a criação de um projeto de software; - Definir um conjunto de requisitos que possam ser validados quando o software for construído.
3. É a atividade de ignorar alguns detalhes, dando foco nos aspectos essenciais para uma determinada visão ou análise.

UNIDADE II

1. O cliente é uma peça importante no desenvolvimento de software, pois é ele quem irá usar o novo sistema e ninguém melhor que ele para saber os requisitos do software, mesmo que não entenda de engenharia de software. Com o cliente participando ativamente do desenvolvimento, fica mais fácil se por acaso os requisitos mudarem, e a fase de treinamento ficará mais curta, já que o cliente já sabe como o software funcionará.
2. Software farmacêutico, Software de caixa eletrônico. Visão Interação - Casos de Uso: conjunto de interações entre o sistema e os agentes externos. Visão estrutural - Projeto: visa construir um sistema que atenta os Casos de Uso.
3. B
4. A

UNIDADE III

1. Generalização é usar uma ideia específica para representar um espectro mais amplo de ideias. Por exemplo, mamífero é um conceito específico que representa uma classe bem maior de definições, já que existem inúmeras espécies de mamíferos. Especialização é especificar um caso geral de maneira mais detalhada, até chegar ao conceito mais restrito. Por exemplo, os cachorros são mamíferos.
2. C
3. D
4. B
5. A
6. C
7. B

GABARITO

UNIDADE IV

1. C
2. B
3. A
4. E
5. C

UNIDADE V

1. E
2. D
3. D
4. B
5. D