



Projeto orientado a objetos

Projeto orientado a objetos

**Paulo Henrique Terra
Anderson Emídio Macedo Gonçalves
Adriano Sepe**

© 2018 por Editora e Distribuidora Educacional S.A.

Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Editora e Distribuidora Educacional S.A.

Presidente

Rodrigo Galindo

Vice-Presidente Acadêmico de Graduação

Mário Ghio Júnior

Conselho Acadêmico

Alberto S. Santana

Ana Lucia Jankovic Barduchi

Camila Cardoso Rotella

Cristiane Lisandra Danna

Danielle Nunes Andrade Noé

Emanuel Santana

Grasiele Aparecida Lourenço

Lidiâne Cristina Vivaldini Olo

Paulo Heraldo Costa do Valle

Thatiane Cristina dos Santos de Carvalho Ribeiro

Revisão Técnica

Ruy Flávio de Oliveira

Editorial

Adilson Braga Fontes

André Augusto de Andrade Ramos

Cristiane Lisandra Danna

Diogo Ribeiro Garcia

Emanuel Santana

Erick Silva Gripe

Lidiâne Cristina Vivaldini Olo

Dados Internacionais de Catalogação na Publicação (CIP)

Terra, Paulo Henrique

T323p Projeto orientado a objetos / Paulo Henrique Terra,
Anderson Emídio Macedo Gonçalves, Adriano Sepe. –
Londrina : Editora e Distribuidora Educacional S.A., 2018.
144 p.

ISBN 978-85-522-0297-4

1. Programação orientada a objetos. 2. Análise de
sistemas. I. Gonçalves, Anderson Emídio Macedo. II. Sepe,
Adriano. III. Título.

CDD 005.12

2018

Editora e Distribuidora Educacional S.A.
Avenida Paris, 675 – Parque Residencial João Piza
CEP: 86041-100 – Londrina – PR
e-mail: editora.educacional@kroton.com.br
Homepage: <http://www.kroton.com.br/>

Sumário

Unidade 1 Iniciando um projeto de sistemas	7
Seção 1 - Projetos	11
1.1 O que é um projeto?	11
1.2 Projeto de software	13
1.3 Entendendo o negócio - o caminho para criação de soluções eficazes	16
1.4 Mapas mentais e requisitos de software	20
1.5 Projetos orientados a objetos	22
Seção 2 - Ciclos de vida	24
2.1 Modelo tradicional ou cascata	24
2.2 Metodologias ágeis	27
Seção 3 - Qualidade de software	31
3.1 Qualidade de software	31
3.2 Qualidade de processo	32
3.3 Qualidade de produto	34
Unidade 2 Análise e projeto	41
Seção 1 - Especificação do software	44
1.1 Especificação de software	44
Seção 2 - Requisitos funcionais	51
2.1 Requisitos funcionais	51
2.2 Requisitos não funcionais	53
Seção 3 - Diagrama de classe	56
3.1 Introdução ao diagrama de classes	56
3.2 Propriedades básicas	57
3.3 Técnicas de refinamento	58
Unidade 3 O paradigma orientado a objetos e a UML	63
Seção 1 - Conceitos de orientação a objetos	66
1.1 Classes	66
1.1.1 Atributos	67
1.1.2 Operações	68
1.1.3 Responsabilidade	69
1.2 Classes avançadas	70
1.2.1 Classificadores	70
1.2.2 Visibilidade	72
1.2.3 Escopo de instância e de estática	73
1.2.4 Multiplicidade	75
1.3 Interface	76
Seção 2 - Introdução à UML	79
2.1 Visão geral	79
2.2 UML na especificação	81
2.3 Documentação	82
2.4 Onde utilizar	82

2.5 Modelo conceitual da UML	83
2.5.1 Itens da UML	84
2.5.2 Relacionamentos	86
2.5.3 Diagramas	87
2.6 Diagramas de classes	88
2.7 Relacionamentos	90
2.7.1 Dependência	92
2.7.2 Generalização	92
2.7.3 Associação	94
2.7.3.1 Nome	94
2.7.3.2 Papel	95
2.7.3.3 Multiplicidade	95
2.7.3.4 Agregação	96
2.8 Diagramas de casos de uso	97
2.9 Diagrama de interação	98
2.10 Diagrama de estado	98
Unidade 4 Tecnologias para projetos orientados a objetos	105
Seção 1 - Linguagens de programação	109
1.1 Linguagens orientadas a objeto	109
1.2 Padrões de projeto	111
1.3 Estrutura de um padrão	112
1.4 Singleton	112
1.5 Outros tipos de padrão de projetos	113
Seção 2 - Modelagem de dados	116
2.1 Modelagem de dados	116
2.2 Mapeamento objeto-relacional	119
2.3 Mapeamento para classes relacionadas por generalização	120
2.4 Frameworks	123
Seção 3 - Arquitetura de software	127
3.1 Projeto de arquitetura	127
3.2 Decisões de projeto de arquitetura	128
3.3 Padrões de arquitetura	130
3.4 Arquitetura em camadas	132
3.5 Arquitetura cliente-servidor	133

Apresentação

Este material didático pretende apresentar aos estudantes de tecnologia os conceitos de projetos de desenvolvimento de software baseados no paradigma orientado a objetos.

A única pretensão deste material é proporcionar ao aluno uma visão global dos princípios que norteiam um projeto de desenvolvimento, afinal, um projeto, seja ele de qualquer natureza, é a oportunidade de concentrar vários conhecimentos e aplicá-los em conjunto, de modo a construir o produto final através do projeto orientado a objetos, no nosso caso, o produto final de software.

Há uma interdisciplinaridade muito grande entre os vários segmentos da engenharia de software, logo, vamos rever alguns conceitos estudados individualmente em outras oportunidades. O objetivo não é realizar uma revisão aprofundada de todas as disciplinas estudadas nas graduações técnicas ou de engenharia na área de tecnologia da informação, porém, vamos abordar os princípios básicos mais fundamentais que devem estar "enraizados", formando uma base sólida de conhecimentos para que todos estes conceitos, uma vez unidos, sejam compreendidos e bem aplicados. É importante ressaltar que projetos podem ser desenvolvidos seguindo metodologias já conhecidas e disseminadas nas várias disciplinas de engenharia da computação, portanto, este material tenta compilar os principais assuntos, inclusive reforçando a importância do conhecimento das etapas definidas nos ciclos de vida de desenvolvimento de software.

Vamos dedicar esta unidade para reforçar conceitos de análise de sistemas e de modelagem das soluções que fazem parte do projeto técnico de desenvolvimento de software. Também abordaremos os princípios da orientação a objetos e da linguagem de modelagem unificada (UML), afinal, estes são os conceitos norteadores e que dão razão ao tema abordado: projetos orientados a objetos.

Na Unidade 1, o aluno conseguirá ter uma visão ampla de conceitos de projetos aplicados ao desenvolvimento de software. Vamos abordar o papel essencial do analista de sistema como condutor da equipe de desenvolvimento e apoiador do cliente ou usuário final, bem como a importância de realizar uma boa engenharia de requisitos para criação de aplicações funcionais e eficientes do ponto de vista de uso e de

sucesso do projeto. Também serão considerados os ciclos de vida tradicionais de desenvolvimento de software em contrapartida com as metodologias ágeis. O fator qualidade será um referencial para que o aluno tenha uma percepção das características que devem ser consideradas tanto para o projeto quanto para o produto final em concepção.

Na Unidade 2, veremos assuntos relacionados às etapas de análise de sistemas e projeto técnico, serão abordados assuntos relacionados à modelagem do problema e da solução técnica. Na análise de sistemas, iremos tratar a condição de refinamento de requisitos, bem como a inclusão de requisitos não funcionais. O aluno deve entender o porquê da importância em modelar um sistema antes da implementação, assim, vamos separar a análise de projeto técnico com a percepção de que análise é a etapa que se aprofunda sobre "o que" deverá ser desenvolvido e que o projeto técnico é a resposta sobre "como" desenvolver.

A Unidade 3 trará à tona os princípios de orientação a objetos que serão foco para todo o andamento do projeto de desenvolvimento de software, serão tratados também os conceitos básicos da UML, com a apresentação dos principais tipos de itens, relacionamento e diagramas utilizáveis na linguagem.

Finalmente, a Unidade 4 abordará temas relacionados às tecnologias que podem ser empregadas nos projetos orientados a objetos. A intenção é esclarecer dúvidas comuns dos iniciantes na área de desenvolvimento de software e propiciar referências mais detalhadas para o aprofundamento do tema.

Por meio dessa estrutura de aprendizagem, esperamos garantir ao aluno uma consolidação dos vários assuntos envolvidos em projetos de desenvolvimento baseados no paradigma orientado a objetos, criando uma base capaz de ajudá-lo em seus futuros desafios, relacionados à análise de sistemas, programação e desenvolvimento de soluções computacionais.

Bons estudos!

Iniciando um projeto de sistemas

Paulo Henrique Terra

Objetivos de aprendizagem

O objetivo desta unidade é proporcionar uma visão geral sobre como um projeto de sistemas é planejado e executado até que se obtenha um produto de software funcional.

Seção 1 - Projetos

Nesta seção, iremos explorar os princípios básicos sobre projetos, construiremos a percepção inicial de que projeto é um conceito amplo e que pode ser aplicado a vários segmentos e que a partir de sua execução pode-se obter um produto ou serviço exclusivo.

Seção 2 - Ciclos de vida

O estudo dos métodos tradicionais e ágeis aplicados aos projetos de desenvolvimento de software é o alvo nesta seção, pois vamos abordar as características básicas dos ciclos de vida tradicionais em analogia aos métodos ágeis.

Seção 3 - Qualidade de software

Qual a qualidade esperada de um produto de software? Nesta seção, discutiremos aspectos de qualidade relacionados ao processo de desenvolvimento e as principais características de qualidade de um software.

Introdução à unidade

Olá aluno, seja bem-vindo à primeira unidade do nosso livro didático!

Atualmente, é cada vez mais comum a presença de aplicativos móveis e sistemas baseados na plataforma web que podem ser acessados por aparelhos celulares, smartphones, smart TVs, tablets e tantas outras variedades de dispositivos. Estes sistemas computacionais estão em constante evolução e presenciamos uma variedade imensa destas aplicações em uma infinidade de setores, como exemplo podemos citar o setor de recreação, representado por aplicativos de jogos, redes sociais, entre outros, o industrial, o de comércio e o de prestação de serviços, pois estes também têm ganhado cada vez mais espaço no mundo virtual. O fato é que, desde os simples os aplicativos de seu dispositivo, até uma grande companhia que pretende aumentar os seus rendimentos explorando o mundo virtual, todos os sistemas computacionais deverão passar por um processo de criação.

Este processo de criação de sistemas computacionais é em sua grande maioria organizado em projetos de desenvolvimento de software, guiados por uma infinidade de boas práticas, metodologias e ciclos de vida. Com o advento de novas formas de realizar as análises necessárias ao entendimento sobre "o que" será desenvolvido e as tecnologias que orientam "como" estes aplicativos serão desenvolvidos, desponta-se o paradigma orientado a objetos.

Unindo-se a ideia de empreendimento organizado por um projeto de software, o paradigma orientado a objetos vem ganhando cada vez mais mercado nas aplicações baseadas em tecnologia, assim, os projetos orientados a objetos não são apenas a junção de dois conceitos, e sim uma realidade largamente utilizada e que exploraremos nesta unidade.

A primeira seção apresentará os conceitos básicos de projeto, estes conceitos são originados antes mesmo da criação do conceito de aplicações que conhecemos nos dias atuais e advêm de outras áreas que passaram pela necessidade de organização de seus empreendimentos, desde a construção civil, passando pela indústria e prestação de serviços.

Nas demais seções, vamos avançar nas práticas de engenharia de

software que orientam o desenvolvimento de software, o objetivo é explorar noções básicas definidas em ciclos de vida, metodologias tradicionais e ágeis que são fundamentais para o entendimento de como um projeto de software orientado a objetos é planejado, executado e acompanhado até obter o seu resultado exclusivo, o produto de software.

Boa leitura!

Seção 1

Projetos

Introdução à seção

Imagine uma sequência de atividades estruturadas cronologicamente que pode se repetir em ciclos de forma que, ao finalizá-las, seja possível obter um produto ou serviço exclusivo (PMI, 2008). Este é o principal objetivo de um projeto e vamos explorar esta ideia nesta seção, nas subseções detalharemos os principais conceitos de projetos e como aplicá-los no desenvolvimento de software, os modelos tradicionais e ágeis, além dos ciclos de vida já descritos nas literaturas de engenharia de software.

1.1 O que é um projeto?

De acordo com o Project Management Institute (PMI, 2008), um projeto é um esforço temporário empreendido para criar um produto, serviço ou resultado exclusivo.

Um projeto deve possuir objetivo e requisitos claramente definidos, este conjunto forma o escopo do projeto. Uma vez que se tenha total clareza do que se pretende atingir, o projeto deverá obedecer a um orçamento previamente aprovado, respeitando o prazo especificado. Vale ainda lembrar que projetos também podem possuir normas de qualidade que devem ser consideradas.

A partir destas definições básicas, podemos listar algumas características que ajudam a definir um projeto:

Figura 1.1 | Características de um projeto

O projeto tem administração e estrutura administrativa própria.	As tarefas de um projeto são interdependentes.	Um projeto utiliza vários recursos para realizar a tarefa
Possui um período de tempo específico, tendo uma data de inicio e uma data para ser concluído.	Deve entregar um resultado único e exclusivo.	Possui um patrocinador que pode ser a empresa, um banco, um consórcio etc.
Possui um grau de incerteza		

Fonte: adaptada de Keeling (2012, p. 5-15) e Clements e Gido (2015, p. 3-5).

Vamos pensar nestas definições básicas aplicadas a um projeto de desenvolvimento de software. Neste caso, o projeto trata da criação de uma aplicação desde o princípio até que se tenha uma versão inicial funcional e entregável ao usuário final. Outros fatores podem motivar o início de um projeto, por exemplo, a necessidade de atualização tecnológica de uma aplicação já existente, a implementação de requisitos funcionais complexos ou um conjunto de requisitos que deverá coexistir com funcionalidades de um software, porém vale ressaltar que a ideia de projeto não deve ser aplicada à rotina de manutenção de uma aplicação existente. Afinal, qualquer modificação de um software que já esteja entregue e em fase de suporte deve ser encarada como rotina de trabalho.

Desta forma, será importante que você compreenda o que não deve ser considerado um projeto, pois este deve ser mantido separado das operações de rotina, já que é pensado e administrado como uma atividade distinta das tarefas do dia a dia para que possa ser controlado com o intuito de atingir o seu resultado específico. Um projeto pode ser encerrado quando os seus objetivos forem alcançados ou por decisão da empresa, quando ela entender que o projeto não é mais viável, e até mesmo necessidades alheias à empresa podem motivar o encerramento de um projeto. As diferenças entre projeto e rotina podem ser observadas no Quadro 1.1

Quadro 1.1 | Características de projetos e atividades rotineiras

Projeto	Atividades contínuas
Estabelecer um bom negócio.	Administrar um negócio consolidado.
Lançar um novo modelo de tablet.	Gerenciar o fornecimento de peças para a linha de montagem do tablet.
Construir um novo aeroporto.	Operar um terminal aeroportuário.
Introduzir um novo sistema de controle de estoque.	Administrar o estoque rotineiramente.
Desenvolver uma concessão para exploração de minério.	Producir minério gerando lucro.
Levantar os requisitos e realizar o desenvolvimento de um website.	Realizar a manutenção das informações de website.
Construir uma nova usina nuclear.	Fornecer o suprimento constante de energia elétrica.

Fonte: Keeling (2012, p. 4).

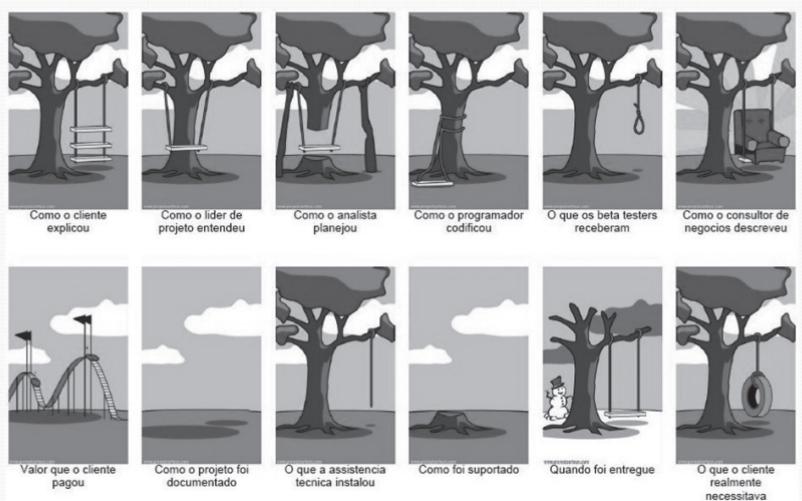
1.2 Projeto de software

A criação de um sistema computacional, seja ele um aplicativo móvel, uma aplicação para a web ou até mesmo uma aplicação *stand-alone*, está vinculada a um processo de desenvolvimento de software e podemos encará-la como uma “receita de bolo” que guiará as etapas necessárias ao atingimento do objetivo do projeto de software.

As metodologias de projeto de software vêm evoluindo durante as últimas décadas, historicamente, as preocupações com o produto final de software tiveram seu foco alterado, ao invés dos engenheiros preocuparem-se somente com a estrutura de dados, algoritmos, dispositivos de entrada e processamento de dados, passa-se a observar com mais rigor as características do negócio e as suas necessidades. Assim, o principal alvo na etapa de planejamento de um projeto de software são os requisitos funcionais estabelecidos pelo negócio, os aspectos técnicos são imensamente importantes, porém eles são a solução de problemas levantados ou até mesmo a resposta a oportunidades encontradas em determinados nichos. O esforço necessário para atingir o resultado em um projeto de software não deve estar concentrado somente nos aspectos técnicos, estes serão parte integrante de algo maior a ser construído. A palavra de ordem é alinhamento, isto mesmo, alinhamento entre o negócio e a tecnologia, não adianta nada dominar uma série de tecnologias sem saber direcioná-las de maneira adequada para o atingimento dos resultados do negócio, muito mais do que atingir um resultado entregando um produto de software, devemos pensar em entregar um produto de software alinhado ao negócio para o qual foi proposto.

A Figura 1.2 ilustra o problema clássico do processo de desenvolvimento de software que não é guiado por um processo bem definido e com foco somente nos aspectos tecnológicos.

Figura 1.2 | O problema clássico



Fonte: adaptado de: <<https://goo.gl/c7NaaI>>. Acesso em: 22 jul. 2017.

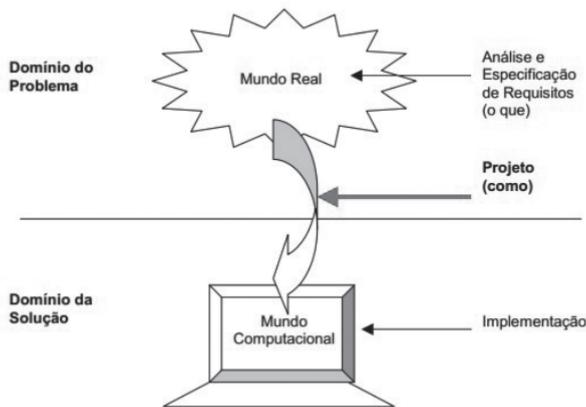
Comparando a primeira imagem superior esquerda com última imagem inferior direita, destacadas pelos retângulos, podemos concluir que a necessidade do cliente não foi bem entendida. É aceitável que o cliente não saiba de maneira precisa o que deseja do software, isto é considerado normal devido à característica abstrata deste tipo de projeto, ou seja, o usuário ou cliente não consegue perceber logo no início do projeto como um software poderá resolver as suas necessidades, portanto, o responsável pelo entendimento das necessidades é o analista que realizou a análise inicial do negócio, ele é o responsável por modelar corretamente “o que” deverá ser feito, conduzindo o projeto técnico como uma resposta assertiva de “como” deverá ser feito. Se assim o fizer, a probabilidade de entregar um produto de software que atenda à necessidade do cliente será substancialmente grande, aumentando, consequentemente, as chances de cumprir com o cronograma e o orçamento estabelecidos.

Para facilitar a compreensão, vamos separar um projeto de software em duas grandes fases, a fase de análise e a fase de projeto. Na primeira, o responsável por entender a necessidade do negócio deve preocupar-se tão somente com “o que” deverá ser construído, ele deve considerar que à sua disposição há tecnologia perfeita, ou seja, capacidade ilimitada de processamento e armazenamento e não passível de falha. Pensando desta forma, o foco não será desviado por possíveis impedimentos que podem inviabilizar o projeto. Somente

na segunda fase será envolvida a modelagem de “como” o sistema será implementado com a adição dos requisitos tecnológicos ou não funcionais.

A Figura 1.3 define claramente esta separação entre as duas fases, enquanto a análise do mundo real que atua no domínio do problema concentra-se em o que deve ser feito, o projeto (técnico) responde como o software deverá ser construído, ele é o elo para o mundo computacional e atua no domínio da solução do problema analisado.

Figura 1.3 | Características de um projeto



Fonte: elaborada pelo autor.

Assim, para atingir os objetivos levantados na fase de análise, deve-se realizar as ações previstas na etapa de projeto de software, portanto, a fase de projeto enfatiza uma proposta de solução que atenda aos requisitos da análise, logo, a análise é uma investigação que se propõe descobrir o que o cliente necessita e o projeto é uma proposta de solução com base no conhecimento obtido na análise.



Questão para reflexão

Apenas o analista do projeto é responsável pela correta interpretação do negócio? Nós somos levados a imaginar que um analista é uma pessoa contratada para este fim, no entanto, várias são as funções que podem assumir esse papel, um programador, um coordenador, enfim, qualquer um que possua as competências necessárias para interpretar os requisitos pode executar esta tarefa. É comum nas empresas pessoas com esta habilidade, mesmo que não tenham como função principal a de analista, realizarem a atividade de análise junto ao cliente.

1.3 Entendendo o negócio – o caminho para criação de soluções eficazes

"Todos nós somos clientes. Portanto, um negócio – sempre – deve entender o que seus clientes valorizam, pois só assim será capaz de elaborar suas estratégias para a entrega de um produto ou serviço de valor." (CAPOTE, 2012, p. 62)

Para compreender a necessidade de um negócio, podemos recorrer a práticas já estabelecidas e consolidadas, por exemplo, o BPM – Business Process Management – ou, simplesmente, Gerenciamento do Processo de Negócio. Empresas são normalmente organizadas por fluxos de trabalho que consistem na execução de atividades sequenciais que fluem entre os departamentos da organização, o BPM possui práticas e notações específicas para que possamos compreender e representar os fluxos através de diagramas. Os diagramas são representações visuais que nos ajudam a ter uma compreensão visual do fluxo de trabalho em uma empresa ou departamento.



Para saber mais

O gerenciamento de processos de negócios é uma prática que possibilita a certificação dos profissionais nesta área, veja como ser um profissional certificado nesta área em: <<http://www.abpmp-br.org/>>. Acesso em: 28 ago. 2017.

Enfim, por que usar uma abordagem de mercado para conhecer fluxos de um determinado negócio? O que isso tem a ver com um projeto de desenvolvimento de software? A resposta para as duas questões é simples: se temos como objetivo desenvolver uma solução computacional que pretenda atender à necessidade do negócio, então, devemos possuir antes de qualquer coisa uma compreensão clara acerca do negócio ao qual a aplicação computacional será empregada.

Surge uma dúvida comum quando chegamos a este ponto: imaginemos um projeto de software com foco no desenvolvimento de um sistema para o departamento de recursos humanos de uma empresa que deva controlar os registros de horário de entrada e saída dos funcionários, para execução deste projeto deve-se conhecer todas as regras da consolidação das leis do trabalho? Na verdade, não todas,

mas as que forem necessárias e que formarem o escopo do projeto, assim, deve-se possuir todo o entendimento das regras relacionadas a jornadas de trabalho, horas extras, e tudo mais que permeia o fluxo estabelecido. Neste caso, não é uma condição apenas da empresa, mas, sim, uma lei imposta a todo o território nacional.

Você, como analista, deve tornar-se um especialista em regras de recursos humanos e jornadas de trabalho? A resposta é não, na verdade, você, como analista ou responsável pelo entendimento do negócio, deverá procurar outras fontes que já possuam este conhecimento, o seu foco é concentrar-se na necessidade, nos requisitos, porém, faz-se necessário a inclusão de pessoas que tenham este conhecimento para auxiliar no entendimento das regras. Lembre-se de que, normalmente, um projeto de software possui um time, e este pode conter pessoas com experiência em determinadas áreas que servirão como apoiadores na análise. Nos modelos mais recentes o próprio cliente é considerado parte do time de projeto e pode auxiliar no entendimento do negócio que é alvo do desenvolvimento, afinal, ninguém melhor para definir o que deve ser feito do que o envolvido diretamente na execução das atividades. Assim, entenda o negócio com quem entende dele. É comum o analista responsável pelo entendimento do negócio e definição de requisitos sair um pouco mais “especialista” no assunto-alvo do projeto, este é um dos motivos de haver profissionais da área de tecnologia da informação com múltiplos conhecimentos. A cada projeto de desenvolvimento em diferentes segmentos temos a oportunidade de aprender um pouco mais de cada negócio e agregar estes conhecimentos às competências profissionais.

Portanto, para que um software seja considerado eficaz em termos de atendimento à necessidade do negócio, é imprescindível que o negócio-alvo do projeto tenha sido devidamente entendido, mapeado e traduzido de maneira clara pelo analista. Outro ponto-chave é a necessidade de sugerir mudanças nos fluxos de negócios atuais, ou até mesmo formalizar as rotinas de trabalho que não estejam bem definidas. Nem sempre o cliente executor realiza as suas atividades da melhor maneira ou tem os seus fluxos de trabalho definidos claramente, assim, neste caso, apenas construir um software baseado em um fluxo ineficaz também tornará a solução pouco válida. O analista torna-se o ponto fundamental na construção de um software eficaz, ele deve ter a capacidade de concentrar-se nos requisitos essenciais que se tornarão a “matéria-prima” para a continuidade do projeto de software.

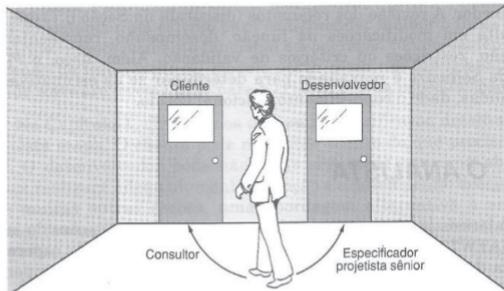
Segundo Pressman (1995, p. 235-236), o analista deve exibir os seguintes traços característicos:

- A capacidade de compreender conceitos abstratos reorganizá-los em divisões lógicas e sintetizar "soluções" baseadas em cada divisão.
- A capacidade de absorver fatos pertinentes de fontes conflitantes e confusas.
- A capacidade de entender os ambientes do usuário/cliente.
- A capacidade de aplicar elementos do sistema de hardware e/ou software aos elementos do usuário/cliente.
- A capacidade de se comunicar bem nas formas escrita e verbal.
- A capacidade de "ver a floresta por entre as árvores".

Provavelmente, a última característica é a que separa verdadeiramente os analistas de destaque dos demais. Pessoas que se atolam em detalhes muito cedo, frequentemente perdem de vista o objetivo global do software. As exigências de software devem ser descobertas de "cima para baixo" – as funções importantes, as interfaces e as informações devem ser plenamente entendidas antes que sucessivas camadas de detalhes sejam especificadas.

A Figura 1.4 demonstra o relacionamento entre o cliente e a equipe de desenvolvimento, se por um lado o analista atua como um consultor para o cliente, apoiando-o na definição das regras de negócio, por outro, ele traduz estas regras em requisitos que deverão ser implementados pela equipe técnica do projeto de software.

Figura 1.4 | O papel do analista

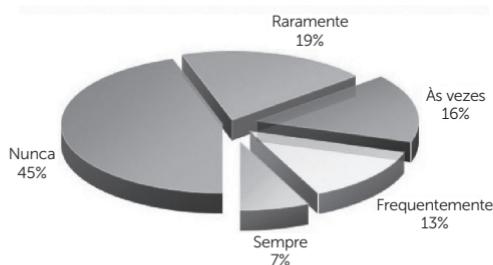


Fonte: Pressman (1995, p. 236).

Podemos, então, concluir que desenvolver um software eficiente dependerá de sua equipe de analistas, estes deverão estar aptos a direcionar o desenvolvimento das funcionalidades em resposta às necessidades do negócio.

Um estudo realizado pelo Standish Group (2000) sobre a frequência da utilização das funcionalidades dos softwares chegou à conclusão de que muito tempo do projeto é despendido criando-se funções que nunca serão utilizadas pelos usuários. A seguir podemos observar um resumo deste estudo.

Figura 1.5 | Frequência da utilização das funcionalidades



- 45% das funcionalidades de um software NUNCA são utilizadas.
- 19% são utilizadas raramente.
- 16% às vezes.
- 13% são utilizadas frequentemente.
- E somente 7% sempre são utilizadas.

Fonte: <http://www.desenvolvimentoagil.com.br/xp/desenvolvimento_tradicional>. Acesso em: 20 jul. 2017.

Ao somar as funcionalidades que sempre são utilizadas às funcionalidades que frequentemente são utilizadas, temos 20% de utilização efetiva de um software, ou seja, 80% das funcionalidades poderiam não ter sido desenvolvidas ou não terem sido priorizadas como parte do escopo principal do produto de software concebido. Assim, a estratégia que também pode ser aplicada ao perceber um ambiente com uma variável muito grande de atividades ou de complexidade é aplicar o princípio de Pareto, que afirma que 80% das consequências advêm de 20% das causas. Aplicando este princípio ao desenvolvimento de software, devemos produzir 20% das

funcionalidades que resolverão 80% das necessidades do negócio.

Além do aumento da produtividade apenas tomando as decisões corretas sobre o que deverá ser desenvolvido, o produto final de software também conterá somente as funcionalidades essenciais que atenderão às principais necessidades estabelecidas pelos analistas.



Para saber mais

Para conhecer o Standish Group, acesse o site, disponível em: <<https://www.standishgroup.com/>>. Acesso em: 28 ago. 2017.

1.4 Mapas mentais e requisitos de software

Várias são as maneiras de se fazer a especificação dos requisitos que serão desenvolvidos em um software. Em um projeto orientado a objetos é comum que a modelagem dos requisitos seja realizada no formato de diagramas da UML (abordaremos este assunto nas próximas seções deste livro), porém, para facilitar a compreensão e leitura geral do projeto, uma abordagem simples pode ser utilizada para a visualização global dos pontos mais relevantes: o uso de mapas mentais. Um mapa mental é uma estrutura em forma de diagrama que pode ser utilizada para representar qualquer ideia ou conceito, ele ilustra no centro de sua estrutura a ideia-alvo do pensamento a ser traduzido em forma visual e ao seu redor são irradiadas as informações relacionadas. Não há regras para criação de sua estrutura, mas podemos pensar que o mapa mental é uma espécie de fotografia do pensamento acerca do que se está representando, assim, caso uma pessoa que esteja visualizando esta fotografia não tenha a ideia clara do que esta imagem representa, isto pode significar que temos um mapa malformado (BUZAN, 2010).

Os mapas mentais têm como objetivo representar o relacionamento conceitual entre as informações que geralmente estão fragmentadas, difusas e pulverizadas no ambiente corporativo, assim, devem possibilitar uma leitura clara da ideia ou conceito que está representando.

Na engenharia de software, os mapas mentais podem ser utilizados para representar a ideia do projeto de desenvolvimento e ajudar a mapear requisitos que posteriormente serão refinados em diagramas da UML. Vamos a um exemplo da aplicação de mapas mentais em

projetos de software.

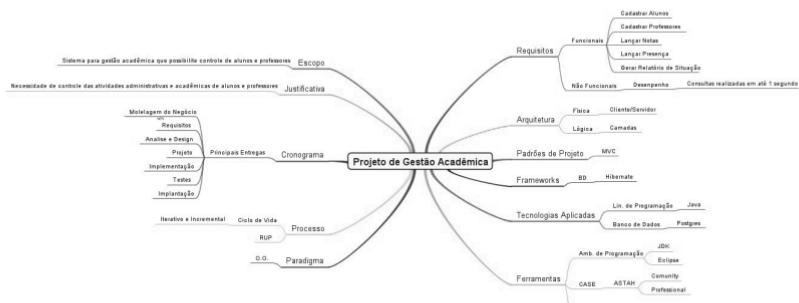
A Figura 1.6 apresenta um mapa mental que mostra de maneira abrangente a ideia de desenvolvimento de um software para gestão acadêmica, os nós que estão representados em torno da ideia central exibem pontos relevantes que deverão ser considerados para a elaboração do projeto. A representação nesta figura ainda não está detalhada, os nós estão a apenas um nível da ideia. Sendo que um mapa mental deve representar de maneira mais detalhada os conceitos relevantes, a Figura 1.7 demonstra como o mapa com mais detalhes ficou estruturado

Figura 1.6 | Mapa mental – projeto de gestão acadêmica



Fonte: elaborada pelo autor.

Figura 1.7 | Mapa mental – projeto de gestão acadêmica



Fonte: elaborada pelo autor.

Figura 1.8 | Mapa mental – detalhe do nó “requisitos”



Fonte: elaborada pelo autor.



Para saber mais

Acesso o link a seguir e veja outros conteúdos relacionados a mapas mentais. Disponível em: <<http://www.mapamental.org>>. Acesso em: 03 set. 2017.

Podemos verificar que o nó superior à direita da ideia central representa os requisitos do software proposto, estes são desmembrados em funcionais e não funcionais, os funcionais são apresentados através de verbos imperativos que indicam as funções que deverão ser desenvolvidas no software. Ainda que os mapas mentais nos ajudem a mapear e especificar os requisitos de um software, ainda são representações sem o detalhamento necessário para que de fato uma equipe de desenvolvimento comprehenda os atributos e funções que deverão ser implementadas, e isto poderá ser facilmente resolvido transformando os requisitos através dos diagramas da UML. No capítulo quatro, vamos transformar os requisitos funcionais de um mapa mental em diagramas de caso de uso, assim teremos um exemplo de como isso pode ser feito utilizando ferramentas CASE que deem suporte a este tipo de função.



Para saber mais

Ferramentas CASE ou Computer Aided Software Engineering são ferramentas para o apoio à construção de software. Veja mais sobre o assunto no link a seguir. Disponível em: <<http://www.devmedia.com.br/ferramentas-case-conhecendo-algumas-boas-opcoes/32034>>. Acesso em: 28 ago. 2017.

1.5 Projetos orientados a objetos

Atualmente, nos projetos de desenvolvimento de software, empregamos com maior frequência o paradigma orientado a objetos,

este é um padrão de raciocínio utilizado nas etapas de análise e projeto em que a equipe de analistas e programadores é levada a pensar de maneira a olhar para os objetos do mundo real, traduzindo-os em objetos no mundo computacional.

Vamos falar mais sobre este paradigma na Unidade 3 deste livro, mas o que precisamos saber neste momento é que projetos orientados a objetos nada mais são do que projetos que empregam este padrão de pensamento (PRESSMAN, 1995). Também é relevante considerar que um projeto baseado neste paradigma deve empregá-lo nas fases em que se aplique este padrão, assim um determinado projeto que passou pela fase de análise, modelando seus artefatos baseados na orientação a objetos, deve seguir este mesmo critério nas práticas de desenvolvimento, utilizando tecnologias que implementem este padrão.

Atividades de aprendizagem

1. Em um projeto de software a etapa de análise tem foco em “o que” deve ser desenvolvido e a etapa de projeto deve responder “como” desenvolver. Assinale a alternativa que contém a expressão que melhor define em qual domínio a etapa de projeto atua.

- a) Domínio público.
- b) Domínio geral.
- c) Domínio do problema.
- d) Domínio da solução.
- e) Domínio da proposta.

2. Atualmente, nos projetos de desenvolvimento de software, empregamos com maior frequência o paradigma orientado a objetos, este é um padrão de raciocínio empregado nas etapas de análise e projeto em que a equipe de analistas e programadores é levada a pensar de maneira a olhar para os objetos do mundo real traduzindo-os em objetos no mundo computacional. Escolha a alternativa que melhor define as etapas em que o paradigma orientado a objetos deve ser utilizado em um projeto de software.

- a) Na etapa de análise apenas.
- b) Na etapa de projeto apenas.
- c) Deve ser empregado na etapa de análise e projeto obrigatoriamente.
- d) Deve ser empregado na etapa de análise obrigatoriamente e opcionalmente na etapa de projeto.
- e) Deve ser empregado na etapa de projeto obrigatoriamente e opcionalmente na etapa de análise.

Seção 2

Ciclos de vida

Introdução à seção

Os projetos de desenvolvimento de software normalmente são conduzidos a partir de um ou mais modelos, um método que define as etapas, suas inter-relações e suas regras. O objetivo desta seção é discutir sobre os modelos tradicionais e ágeis aplicados aos projetos de software.

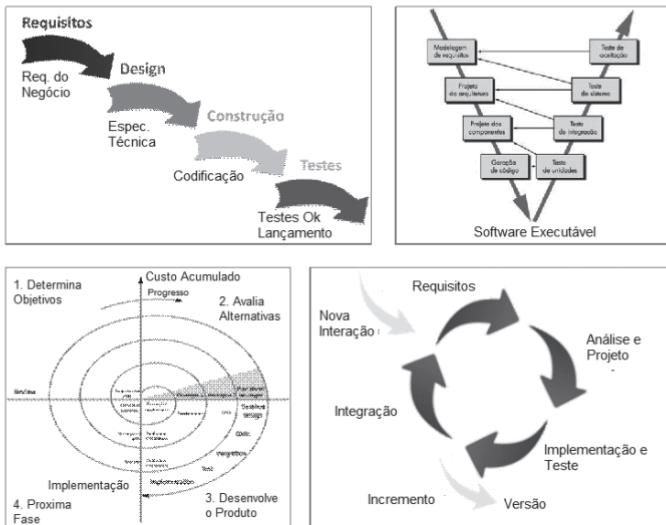
2.1 Modelo tradicional ou cascata

Todo desenvolvimento de software passa tipicamente pelas seguintes etapas:

- Análise.
- Desenho.
- Desenvolvimento.
- Testes.
- Entrega.

Existem diferentes processos formalizados para o desenvolvimento de software, entre eles podemos destacar os modelos cascata, espiral, iterativo e incremental e modelo em V, mas todos contêm tipicamente etapas semelhantes (PRESSMAN, 1995).

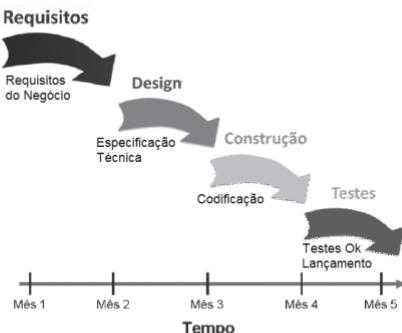
Figura 1.9 | Modelos tradicionais



Fonte: adaptada de: Tiexam (2017, p. 4).

O modelo em cascata (*waterfall*) é o mais tradicional e surgiu como proposta de desenvolvimento de software na década de 1960. Este modelo é a resposta inicial ao período conhecido na engenharia de software como “a crise do software”. Neste período havia grandes dificuldades no processo de desenvolvimento e boas práticas foram compiladas neste ciclo como resposta a estas dificuldades. Neste modelo, o software é desenvolvido por fases, a principal característica deste ciclo de vida é que tudo é definido no início do desenvolvimento e não há uma clara separação entre os processos. Esta característica de definir tudo antecipadamente e tentar cumprir é chamada de pretidivo, então, neste tipo de modelo um software funcional somente é uma saída no final da cascata (PRESSMAN, 1995).

Figura 1.10 | Ciclo de vida cascata



Fonte: adaptada de: Tiexam (2017, p. 5).

Assim, o resultado do planejamento do projeto que usa métodos tradicionais como o modelo cascata é um plano detalhado e o significado de sucesso é o cumprimento deste plano. Neste modelo, espera-se que a sequência de atividades seja executada de forma contínua, sem a necessidade de mudança e há resistência quando elas ocorrem, não havendo planejamento para retorno das atividades. Após a conclusão do projeto, geralmente depois de um longo período, o produto é liberado para o mercado, clientes e usuários, e uma grande versão é entregue. Se o feedback for positivo e indicar aceitação do produto, todos os stakeholders compreendem que houve sucesso no projeto, mas este tipicamente não é o caso.

Figura 1.11 | Ciclo de vida cascata



Fonte: adaptada de: Tiexam (2017, p. 9).

Conforme demonstrado na Figura 1.11, ao final da entrega, após todas as etapas terem sido concluídas, há um grande potencial de retrabalho, isto se deve há alguns fatores, entre eles o feedback tardio do usuário que só ocorre após receber o produto. Neste caso, potenciais correções que poderiam ter sido consideradas durante o desenvolvimento serão percebidas somente ao final do projeto, perde-se oportunidades de melhoria e o produto pode ser entregue obsoleto,

além de sofrer influência de fatores externos, como a receptividade do mercado, novas tecnologias, premissas que foram alteradas, e rapidez das alterações dos cenários comerciais atuais.



Questão para reflexão

Por que utilizar um ciclo de vida em um projeto de software? Na verdade, projetos de software são baseados em um ciclo vida, isto se deve ao fato de que os ciclos de vida implementam etapas e atividades básicas para a construção de um software, portanto, qualquer que seja o ciclo, ele apoiará a evolução da construção do produto final.

2.2 Metodologias ágeis

Desenvolver um software é um problema complexo, envolve muitas pessoas, tecnologia, conhecimento, regra de negócios e exige diversas integrações. A complexidade dos processos de desenvolvimento é ainda multiplicada por fatores, como a evolução dos requisitos, em que um componente que foi desenvolvimento hoje para poucos funcionários pode não funcionar quando possuir vários usuários amanhã, os desenvolvedores que trabalham no projeto podem não estar disponíveis no futuro, a produtividade pode variar, e a tecnologia de hoje pode estar obsoleta amanhã, ou seja, a quantidade de itens desconhecidos no início de um projeto é realmente grande.

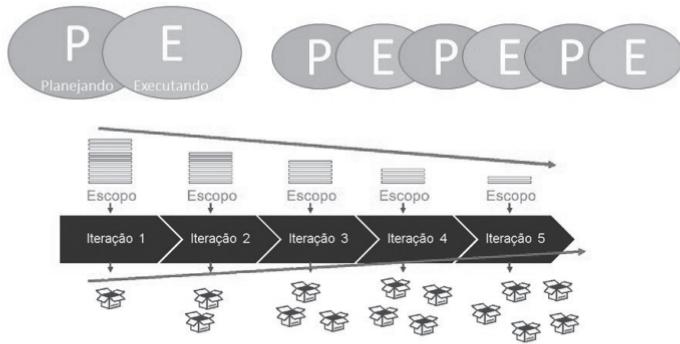
As metodologias ágeis, apesar de existirem há mais de 20 anos, somente no início dos anos 2000 tiveram grande disseminação como resposta à complexidade de desenvolvimento de software. Tais metodologias consideram o processo empírico de desenvolvimento, pois o empirismo afirma que o conhecimento vem da experiência e devemos tomar decisões com base no que se conhece. A ideia por trás disso é dar um passo de cada vez, realizar uma pequena quantia de trabalho para adquirir experiência.

Assim, diferentemente dos modelos preditivos como o cascata, em que todo o planejamento é feito no início do projeto, nos modelos empíricos ou ágeis há replanejamento baseado em inspeção frequente dos resultados (PRIKLADINICKI et al., 2014).

Na Figura 1.12, observamos que enquanto no modelo preditivo o objetivo é planejar e executar, no modelo empírico há vários ciclos

de planejamento e execução. O objetivo é que ao realizar cada uma das etapas de desenvolvimento, chamadas de iteração, uma versão funcional de software seja entregue, formando um produto mínimo. Conforme o projeto evolui pelas iterações, o escopo geral do que deveria ser feito é reduzido e a quantidade de funcionalidades aumenta, incrementando-as ao produto.

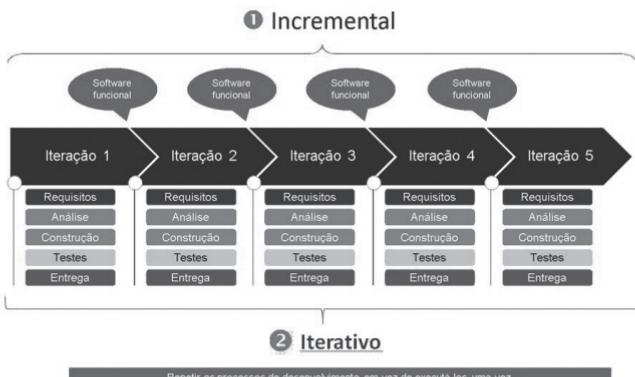
Figura 1.12 | Modelo preditivo e empírico



Fonte: adaptada de Tiexames (2017, p. 13).

Isto significa dizer que dentro de cada uma das iterações serão executadas as etapas de um modelo tradicional, porém, não é necessário levantar todos os requisitos como previsto nos modelos preditivos, esta ação é feita em cada uma das iterações, e pode ser observada na Figura 1.13.

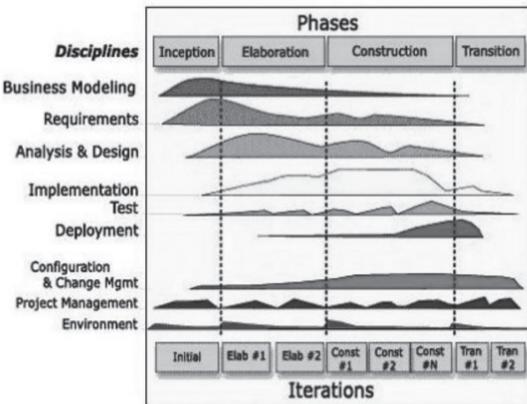
Figura 1.13 | Iterativo e incremental



Fonte: adaptada de Tiexames (2017, p. 15).

Um modelo amplamente utilizado nos projetos atuais de desenvolvimento de software é o Rational Unified Process – RUP –, ou Processo Unificado Rational, este modelo pertence à IBM e mescla as características dos modelos preditivos com as características dos modelos ágeis. Podemos ver na Figura 1.14 o ciclo de vida de um projeto de software baseado no processo unificado.

Figura 1.14 | Fases do RUP



Fonte: adaptada de <<https://www.ibm.com/developerworks/ssa/library/ws-SOAGovernancepart2/index.html>>. Acesso em: 29 jul. 2017.



Para saber mais

Para saber mais sobre as ferramentas do processo unificado acesse o link, disponível em: <<https://www-01.ibm.com/software/br/rational/>>. Acesso em: 29 ago. 2017.

Atividades de aprendizagem

1. O ciclo de vida de um sistema especifica todas as fases de desenvolvimento, de sua concepção até à sua manutenção e declínio. Existem alguns processos conhecidos para o desenvolvimento de software, um destes possui como característica ser iterativo e incremental, ou seja, a cada fase do projeto inicia-se um planejamento prévio. Após a execução da fase verifica-se o progresso e os resultados (riscos, lições aprendidas) incrementando novos objetivos para o ciclo seguinte, evoluindo, então, para a próxima iteração. O processo de software em questão é:

- a) Modelo espiral.
- b) Ciclo de vida em cascata.
- c) Modelo de desenvolvimento evolucionário (prototipação).
- d) O modelo de desenvolvimento ágil.
- e) Método de desenvolvimento em V.

2. Atualmente, os processos empíricos ou ágeis são cada vez mais comuns nos projetos de desenvolvimento de software, algumas metodologias ágeis mesclam em suas características padrões dos modelos preditivos. Sobre estes modelos, assinale a alternativa correta:

- a) O ciclo de vida preditivo é o projetado para métodos ágeis, com forte participação das partes interessadas, interações muito rápidas com tempo e recursos fixos.
- b) O ciclo de vida preditivo é aquele em que escopo, tempo e custos exigidos são determinados o mais cedo possível no ciclo de vida do projeto.
- c) Os ciclos de vida preditivos são preferidos quando pouco se conhece do produto a ser entregue e, por este motivo, são necessárias previsões e estimativas de alto nível.
- d) O ciclo de vida preditivo é utilizado quando uma organização necessita administrar mudanças dos objetivos e escopo, bem como reduzir a complexidade de projetos grandes.
- e) O ciclo de vida preditivo é aquele em que as fases de um projeto intencionalmente repetem uma ou mais atividades à medida que a compreensão do produto pela equipe do projeto aumenta.

Seção 3

Qualidade de software

Introdução à seção

Os aspectos relacionados à qualidade de software serão abordados nesta seção, vamos segregar dois pontos fundamentais dos princípios de qualidade, as questões relacionadas ao processo de desenvolvimento e os critérios de qualidade do produto originado do projeto de software.

3.1 Qualidade de software

O trecho a seguir, extraído do livro *Engenharia de Software* (PRESSMAN, 1995, p. 724) nos leva a pensar sobre a questão de qualidade de software e em como defini-la.

Até mesmo os mais experientes desenvolvedores de software concordarão que a elevada qualidade de software é uma meta importante. Como definir qualidade? Um galhofeiro disse certa vez: "Todo programa faz alguma coisa certa; talvez apenas não seja a coisa que queremos que ele faça."

Podemos facilmente atribuir requisitos de qualidade a uma série de produtos tangíveis, um carro pode ter qualidade por utilizar matéria-prima de primeira linha, um ótimo acabamento com mecânica que proporciona alto desempenho com baixa manutenção.

Quando pensamos em um produto concreto, podemos definir certos critérios de qualidade baseados em pouca experiência como cliente ou usuário, mas como definir critérios de qualidade para um software?

Conforme Pressman (1995, p. 724), qualidade de software é definida como

Conformidade a requisitos funcionais e de desempenho explicitamente declarados, a padrões de desenvolvimento claramente documentados e a características implícitas que são esperadas de todo software profissionalmente desenvolvido.

O autor ainda enfatiza três pontos importantes a serem observados

1. Os requisitos de software são a base a partir da qual a qualidade é medida. A falta de conformidade aos requisitos significa falta de qualidade.

2. Padrões especificados definem um conjunto de critérios de desenvolvimento que orientam a maneira segundo a qual o software passa pelo trabalho de engenharia. Se os critérios não forem seguidos, o resultado quase que seguramente será a falta de qualidade.

3. Há um conjunto de requisitos implícitos que frequentemente não é mencionado, por exemplo, o desejo de uma boa manutenibilidade. Se o software se adequar aos seus requisitos explícitos, mas deixar de cumprir seus requisitos implícitos, a qualidade será suspeita.

A qualidade de software é uma combinação complexa de fatores que variarão de acordo com diferentes aplicações e clientes que as solicitam.

3.2 Qualidade do processo

Pressman (1995) define que ao longo do desenvolvimento de um software, a qualidade do projeto em evolução é avaliada mediante uma série de revisões técnicas. Para que possamos avaliar a qualidade de uma representação de projeto, devemos estabelecer critérios para ele.

1. Um projeto deve exibir uma organização hierárquica que faça uso inteligente do controle entre os componentes de software.

2. Um projeto deve ser modular, ou seja, o software deve ser logicamente dividido em componentes que executem funções e subfunções específicas.

3. Um projeto deve conter uma representação distinta e separável de dados e procedimentos.

4. Um projeto deve levar a módulos que apresentem características funcionais independentes.

5. Um projeto deve levar a interfaces que reduzam a complexidade de conexões entre os módulos e com o ambiente externo.

6. Um projeto deve ser derivado usando-se um método capaz de permitir repetição e que seja orientado pelas informações obtidas durante a análise de requisitos de software.

As características de um bom projeto apresentadas anteriormente não são conseguidas casualmente. O processo de projeto de engenharia de software estimula o bom projeto por meio da aplicação de princípios fundamentais, metodologia sistemática e cuidadosa revisão.

Ainda segundo Pressman (1995), as principais características de qualidade do projeto de software baseados nas diretrizes anteriores são: níveis de abstração, modularidade, ocultação de informações e independência funcional.

Ao considerarmos uma solução modular, muitos níveis de abstração podem ser apresentados. Em um nível de abstração mais elevado, uma solução é declarada em termos amplos, usando-se a linguagem do ambiente do problema. Nos níveis de abstração inferiores, uma orientação procedimento é assumida. A terminologia orientada ao problema é acoplada a uma terminologia orientada à implementação, num esforço para se estabelecer uma solução. Finalmente, no nível de abstração inferior, a solução é estabelecida de forma que possa ser diretamente implementada.

O conceito de modularidade no software de computador tem sido adotado há várias décadas. O software é dividido em componentes separadamente nomeados e endereçáveis, denominados módulos, que são integrados para atender aos requisitos ditados pelo problema. Este conceito leva o projetista de software a uma questão fundamental: como decompor uma solução de software para obter o melhor conjunto de módulos? O princípio da ocultação de informações sugere que os módulos sejam caracterizados pelas decisões de projeto que cada módulo esconde de todos os outros. Assim, os módulos devem ser especificados e projetados de tal forma que as informações (atributos e métodos) contidas num módulo sejam inacessíveis a outros módulos que não tenham necessidade de tais informações.

O conceito de independência funcional é um produto direto da modularidade e dos conceitos de abstração e ocultação de informações. Ela é conseguida desenvolvendo-se módulos com função “com um propósito” e “aversão” a interações excessivas com outros módulos.

Portanto, um projeto de desenvolvimento de software que consiga incluir as diretrizes comentadas anteriormente, certamente incluirá critérios de qualidade que levarão o produto final de software a também obter alguma qualidade, porém, outras características do produto poderão ser observadas em completude ao amplo espectro de qualidade que pode ser dado ao produto.



Questão para reflexão

Por que adotar um ciclo de vida em um projeto de desenvolvimento de software? Já vimos esta questão anteriormente, e segue uma nova resposta dentro do contexto de qualidade: um ciclo de vida geralmente

implementa as características de qualidade de processo de software citadas anteriormente, assim, adotando um ciclo que contenha estas características como base para o processo de desenvolvimento, consequentemente o projeto terá incluso os princípios de qualidade.

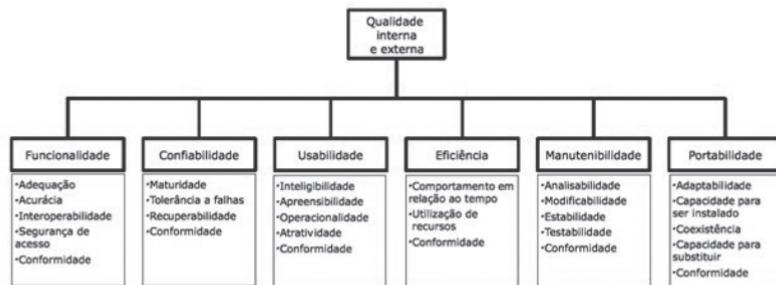
3.3 Qualidade do produto

Quais características do produto final de software podem ser consideradas? Para avaliar a qualidade, suas características, sub características e atributos, a International Organization for Standardization (ISO) criou a norma ISO/IEC 9126, que padroniza a avaliação da qualidade do software.

Esta norma definiu as características internas e externas do software. As internas dizem respeito ao código fonte, arquitetura e outros itens implícitos, já as externas são as observáveis pelo usuário final.

A Figura 1.15 demonstra a organização destas características divididas em seis grandes grupos, definidas nos aspectos de funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade.

Figura 1.15 | Quadro de características



Fonte: adaptada de: <<https://www.tiespecialistas.com.br/2016/10/analise-sobre-iso-9126-nbr-13596/>>. Acesso em: 20 jul. 2017.

A característica de funcionalidade está relacionada ao que o software se propõe a realizar, se as funções definidas satisfazem às necessidades do negócio. Já a característica de confiabilidade diz respeito à capacidade do software manter o seu nível de desempenho sob condições estabelecidas durante um período de tempo estabelecido. A usabilidade é um conjunto de atributos que evidencia o esforço necessário para poder utilizar o software, além do julgamento

individual deste uso por um conjunto de usuários. A eficiência mede os atributos de uso de recursos de hardware, por exemplo, o consumo de processamento e o tempo de execução das funções. A manutenibilidade é um conjunto de atributos do software que evidencia o esforço necessário para modificá-lo, remover seus defeitos ou adaptá-lo a mudanças ambientais e, por sua vez, a portabilidade evidencia a capacidade do software de ser transferido de um ambiente para outro.

Em todas as características encontramos a sub característica "conformidade", isto significa que para cada grupo de características, podemos definir o grau de conformidade para cada tipo de produto de software. Um determinado software pode necessariamente, por condições impostas pelo negócio, apropriar-se da característica portabilidade caso tenha sido previamente estabelecida como critério de aceitação deste software. Assim, para cada um dos tipos de característica deve-se avaliar as condições específicas de uso de cada software a ser construído.



Para saber mais

Para saber mais sobre qualidade de software, acesse o link, disponível em: <<https://www.tiespecialistas.com.br/2016/10/analise-sobre-iso-9126-nbr-13596/>>. Acesso em: 29 ago. 2017.

Atividades de aprendizagem

1. (CESPE, 2010, SAD-PE, Analista de Controle Interno – Tecnologia da Informação) Qualidade de software é o grau para o qual um software possui uma combinação desejável de atributos, que, adicionalmente, deve ser claramente definida, caso contrário, uma avaliação da qualidade será realizada de modo intuitivo. Para que tais atributos de qualidade sejam medidos, faz-se necessário identificar um conjunto apropriado de métricas. Acerca dos conceitos gerais de medição de qualidade de software, assinale a opção correta.

- a) Os atributos de qualidade de software, nos modelos de qualidade ISO, são organizados conforme seis características, sendo três delas internas (eficiência, manutenibilidade e portabilidade) e as outras três, externas (funcionalidade, confiabilidade e usabilidade).

- b) O modelo de referência para medição de qualidade de produto de software da ISO propõe quatro diferentes perspectivas, mediante as quais podem ser desenvolvidas métricas de medição de atributos de qualidade interna, externa, de operação e de uso.
- c) De forma geral, os modelos de qualidade da ISO são focados na qualidade de produtos de software, e os modelos CMMI, na qualidade do processo de software.
- d) No modelo IEEE de qualidade de software, um fator de qualidade é uma entidade mais genérica que um atributo de qualidade.
- e) Os modelos de qualidade de software são organizados segundo a perspectiva de que a qualidade do processo de desenvolvimento influencia, diretamente, a qualidade interna, que, por sua vez, influencia a qualidade externa do produto de software, e esta última exerce influência direta sobre a qualidade do produto em uso.

2. (CESPE,2017, TER-PE, Analista Judiciário – Analise de Sistemas) Uma das características do modelo de qualidade de software está descrita, pela ISO 9126, como a capacidade do produto de software de apresentar desempenho apropriado, relativo à quantidade de recursos usados, sob condições especificadas. Assinale a alternativa que corresponde à característica apresentada:

- a) Confiabilidade.
- b) Usabilidade.
- c) Funcionalidade.
- d) Eficiência.
- e) Manutenibilidade.

Fique ligado

Nesta unidade, estudamos:

- Conceito de projetos, projetos de software e projetos orientados a objetos.
- A importância do entendimento do negócio para elaboração de software eficiente.
- Mapas mentais – aplicação geral e uso na especificação de requisitos de software.
- Ciclo de vida – modelo tradicional e metodologia ágil.
- Qualidade de software – processo e produto.

Para concluir o estudo da unidade

Caro aluno, chegamos ao fim desta unidade!

Você deve ter percebido que a disciplina de projetos de software possui uma ampla abrangência de assuntos. Isto é um fato, é dentro de um projeto que utilizamos muitos dos conhecimentos adquiridos nas disciplinas de engenharia de software, gestão de projetos e as demais disciplinas relacionadas ao assunto. Todo conhecimento adquirido nesta etapa será preciso para que se forme a noção geral sobre como o processo de desenvolvimento de software tem início e fim e o que nele está compreendido. A mensagem final desta unidade é: não pare seus estudos por aqui, continue pesquisando e avance nos tópicos destinados a esse o assunto!

Bons estudos!

Atividades de aprendizagem

1. O BPM ou Gerenciamento do Processo de Negócio vem sendo cada vez mais utilizado para apoiar a aprendizagem dos principais fluxos de trabalho que devem ser observados na etapa de análise de projetos de software. Assim, o BPM se constitui de:

- a) Apenas ferramentas de modelagem de processos que têm como objetivo documentar os processos do negócio de uma empresa.
- b) Workflows que indicam como as atividades são executadas em uma empresa.
- c) Ferramenta para reengenharia de processos de trabalho de uma empresa.
- d) Software de gestão empresarial.
- e) Modo estruturado de gerência e otimização de performance dos processos de negócio de uma empresa.

2. Os mapas mentais são esquemas visuais em forma de diagrama que podem ser utilizados na engenharia de software para facilitar a visão geral do projeto de desenvolvimento de software e especificar requisitos. Sobre mapas mentais é correto afirmar que:

- a) Um mapa mental é um diagrama estruturado e segue as regras da UML 2.0.
- b) Os mapas mentais surgiram na disciplina de engenharia de software

como forma rápida e sem detalhes para especificação de requisitos que não podem ser detalhados em outras linguagens de modelagem.

c) Os mapas mentais surgiram na disciplina de engenharia de software como forma rápida e sem detalhes para especificação de requisitos e podem ser detalhados em outras linguagens de modelagem.

d) Os mapas mentais, apesar de sua estrutura livre de criação, não podem ser empregados nos projetos de desenvolvimento de software.

e) Os mapas mentais não são empregados na engenharia de software devido à sua característica desestruturada incompatível com a disciplina.

3. Quando falamos sobre qualidade de software, devemos considerar aspectos da qualidade do processo de desenvolvimento que irão impactar diretamente no produto final. Acerca do gerenciamento da qualidade de produtos e de processo de software, assinale a opção correta.

a) O gerenciamento de riscos não faz parte do escopo do gerenciamento de qualidade.

b) As revisões da qualidade da documentação e dos processos usados para a produção de um software estão inclusos no controle de qualidade.

c) Somente no plano de riscos estão contidas as definições das metas de qualidade e dos planos de qualidade.

d) O modelo ISO 9000 é inadequado ao gerenciamento de qualidade de produtos virtuais como os softwares e se aplicam somente ao gerenciamento de produtos reais de empresas do ramo fabril.

e) Para evitar a burocracia e documentos desnecessários ao processo de desenvolvimento, deve-se evitar os padrões de documentação nos modelos de qualidade.

4. Um processo de software é um conjunto de atividades relacionadas que levam à produção de um produto de software. Existem muitos processos de software diferentes e todos incluem as atividades fundamentais para seu desenvolvimento. Escolha a alternativa que melhor define as atividades essenciais de um processo de software.

a) Análise e projeto.

b) Análise, projeto e testes.

c) Análise, implementação e testes.

d) Análise/desenho, desenvolvimento, testes e entrega.

e) Análise/desenho, desenvolvimento e testes.

5. Na engenharia de software, a frase "identificar os aspectos importantes, ignorando os detalhes" é empregada nas fases iniciais de um projeto de desenvolvimento de software. A frase mencionada define o princípio da:

- a) Abstração.
- b) Generalização.
- c) Flexibilização.
- d) Decomposição.
- e) Formalidade.

Referências

ALVES, Pereira Willian. **Análise e projeto de sistemas**: estudo prático. São Paulo: Érica, 2017.

ABNT. Associação Brasileira de Normas Técnicas. **NBR ISSO/IEC 9126-1**: engenharia de software – qualidade do produto – parte 1: modelo de qualidade. Rio de Janeiro, 2003.

BUZAN, Tony. **The mind map book**: unlock your creativity, boost your memory, change your life. England: BBC Active, 2010.

CAPOTE, Gart. **BPM para todos**: uma visão geral abrangente, objetiva e esclarecedora sobre gerenciamento de processos de negócio. Rio de Janeiro: Gart Capote, 2012.

CLEMENTS, James P.; GIDO, Jack. **Gestão de projetos**. São Paulo: Cengage Learning, 2015.

COHN, Mike. Are **64% of features really rarely or never used?** Disponível em: <<https://www.mountaingoatsoftware.com/blog/are-64-of-features-really-rarely-or-never-used>>. Acesso em: 15 jul. 2017.

IBM. **Rational unified process**: best practices for software development teams. Disponível em: <https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf>. Acesso em: 20 jul. 2017.

KEELING, Ralph. **Gestão de projetos**: uma abordagem Global. Ed. Especial Anhanguera. São Paulo: Saraiva, 2012.

MARTINEZ, Mariana. **RUP**. Disponível em: <<http://www.infoescola.com/engenharia-de-software/rup/>>. Acesso em: 20 jul. 2017.

PMI. Project Management Institute. PMBOK. **Um guia do conhecimento em gerenciamento de projetos**: GUIA PMBOK. 5. ed. Pennsylvania: Project Management Institute Inc., 2013.

PMI. Project Management Institute. PMBOK. **Um guia do conhecimento em gerenciamento de projetos**: GUIA PMBOK. 4. ed. Pennsylvania: Project Management Institute Inc., 2008.

PRESSMAN, Roger. **Engenharia de software**. 3. ed. São Paulo: Pearson Makron Books, 1995.

PRIKLADNICKI, Rafael et al. **Métodos ágeis para desenvolvimento de software**. Porto Alegre: Bookman, 2014.

TIEXAMES. Curso e-learning preparatório para exame EXIN Agile Scrum Foundation. Disponível em: <http://tiexames.com.br/novosite2015/curso_EXINAgileScrumFoundation.php>. Acesso em: 20 jul. 2017.

Análise e projeto

Anderson Emidio de Macedo Gonçalves

Objetivos de aprendizagem

Compreender a importância de modelar um sistema antes da implementação. Deixar claro que a análise é a etapa que se aprofunda sobre “o que” deverá ser desenvolvido e que o projeto técnico é a resposta sobre “como” desenvolver.

Seção 1 | Especificação do software

Serão abordadas nesta seção atividades básicas da especificação do software, evolução, desenvolvimento e validação do software.

Seção 2 | Requisitos funcionais

Serão abordados nesta seção os requisitos funcionais e não funcionais do processo de análise e projeto de sistemas.

Seção 3 | Diagrama de classe

Esta seção abordará algumas técnicas sobre a elaboração do diagrama de classe e como ele é aplicado no processo de análise e projeto de sistemas.

Introdução à unidade

O processo de análise e projeto de um sistema aborda várias etapas que são cruciais para o desenvolvimento de um produto com qualidade e que atenda às necessidades e expectativas do cliente.

Essa unidade de ensino abordará características de descrição e funcionamento de três assuntos referente à análise e projeto orientado a objetos, são eles: especificação de software, requisitos funcionais e não funcionais e diagramas de classe.

Seção 1

Especificação de software

Introdução à seção

O processo de modelagem envolve a representatividade de objetos do mundo real e identificamos o que esse futuro sistema irá fazer.

Diversas empresas se preocupam em levantar e desenvolver os requisitos funcionais e não funcionais de um determinado sistema, mas se esquecem de construir o modelo lógico do sistema e partem para a fase de implementação, justificando o prazo curto de entrega e um elevado custo em horas de desenvolvimento, assim, pulando o desenvolvimento do modelo lógico, o projeto acabará mais rápido. O que as empresas não percebem é que sem uma especificação dos requisitos, ou ainda se eles forem interpretados incorretamente, podem acarretar um sistema com deficiências graves, gerando problemas futuros.

Uma das metodologias utilizadas para modelagem de software, a Unified Modelling Language (UML) é utilizada neste contexto de análise e projeto. Ela é uma linguagem utilizada na construção, documentação e modelagem de artefatos de um sistema computacional.

Iniciaremos estudando a especificação de software no processo de engenharia de software.

1.1 Especificação de software

Antes de começarmos a falar sobre o processo de modelagem de software, tarefa executada na etapa de análise e projeto de sistemas, vamos falar um pouco sobre a tecnologia que envolve não só a construção da modelagem de um sistema, mas tudo que se refere a um sistema computacional.

Nos dias atuais, os computadores estão presentes em nossa vida de forma fundamental. Em casa, na escola, na faculdade, na empresa, ou em qualquer outro lugar, eles estão sempre entre nós. Dizemos "nos dias atuais", pois as funcionalidades dos computadores podem ser alteradas ao longo do tempo, considerando os avanços da sociedade

e da tecnologia.

Aliás, isso nos faz lembrar de duas frases reflexivas que gostaríamos de deixar com vocês neste momento sobre tecnologia.

É comum fazermos uma confusão histórica entre tecnologia e computadores. Os computadores foram inventados há cerca de 80 anos, porém a tecnologia existe há mais de 2000 anos.

Antes de Cristo, no antigo Egito, mais precisamente na época que viveram os grandes faraós, foram construídas as pirâmides egípcias que existem até hoje. Muito antes desta época a tecnologia está presente na sociedade. Você pode achar estranho, mas analise e reflita: como seria possível sem o auxílio da tecnologia que a população daquela época conseguisse erguer monumentos com tamanha precisão e destreza?

A palavra tecnologia é sinônimo de "método", de acordo com o dicionário da língua portuguesa. As pirâmides egípcias foram construídas no padrão de qualidade de acordo com uma unidade de medida denominada cúbito, uma das mais antigas unidades de medida das quais se tem notícia, utilizada no velho Egito (KOSCIANSKI; SOARES, 2007).

O cúbito consistia na medida do antebraço do faraó da época, ou seja, baseando-se nessa medida eram feitas as métricas para aferir se as pirâmides estavam construídas neste padrão. Caso fosse constatada alguma irregularidade, o responsável era punido severamente (KOSCIANSKI; SOARES, 2007).

Explicado sobre a tecnologia, vamos às frases:



Questão para reflexão

Analise as duas afirmações a seguir, refletindo sobre a veracidade dos fatos:

A tecnologia é tão antiga quanto a humanidade.

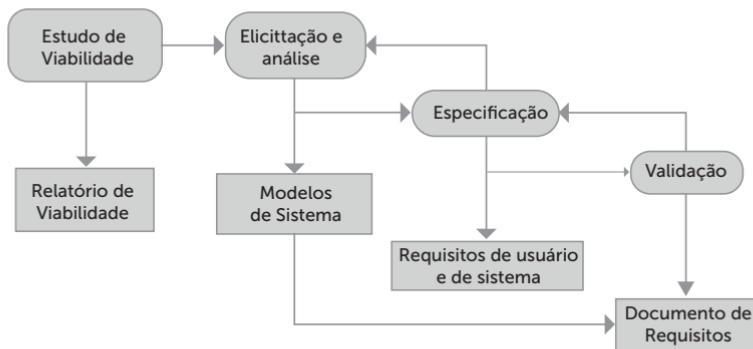
A tecnologia evolui permanentemente como a própria sociedade evolui.

A tecnologia tem seu papel fundamental na sociedade moderna, como também teve na contemporânea.

Voltando à especificação de software, ela estabelece as funções e restrições de um sistema. Essa atividade é denominada engenharia de requisitos e é fundamental no processo. Erros nesta etapa acarretarão em problemas posteriores no projeto e na implementação do sistema.

O processo de engenharia de requisitos, como pode ser observado no Figura 2.1, tem uma documentação que é exatamente a especificação de software (SOMMERVILLE, 2010).

Figura 2.1 | Processo de engenharia de requisitos



Fonte: <<http://progridbb.wikidot.com/engenhariadesoftware>>. Acesso em: 3 ago. 2017.

O processo de engenharia de requisitos produz uma documentação do sistema. Esta documentação é denominada especificação do software.

Os requisitos são apresentados aos **stakeholders** envolvidos no sistema, em dois níveis de detalhamento. Os clientes e operadores do sistema precisam de uma documentação em alto nível, pois não são usuários técnicos, por este motivo não tem a obrigação de entender todos os requisitos necessários para o desenvolvimento do sistema.

Já os usuários técnicos, como desenvolvedores, analistas, engenheiros de software etc., precisam de um maior detalhamento, ou seja, a especificação completa de requisitos.

Figura 2.2 | Modelo em espiral dos processos de engenharia de requisitos



Fonte: <<http://progridbb.wikidot.com/engenhariadesoftware>>. Acesso em: 30 ago. 2017.

O objetivo desses modelos de documentação é a criação e manutenção dos processos de engenharia de requisitos. Observe,v na Figura 2.2, outro modelo de documentação da engenharia de requisitos (SOMMERVILLE, 2010).

O processo de engenharia de requisitos é constituído basicamente por quatro etapas:

- **Viabilidade:** o estudo da viabilidade de um projeto leva em consideração se as expectativas do cliente estão de acordo com suas necessidades, pois nem sempre o que o cliente quer é o que ele precisa, ou o que ele precisa é viável do ponto de vista comercial. Também deve considerar restrições orçamentárias. O resultado deve apontar se o projeto pode prosseguir para uma análise mais detalhada ou não.

- **Levantamento e análise de requisitos:** este processo visa à obtenção dos requisitos por meio dos sistemas já existentes na empresa, sendo eles informatizados ou não, avaliando documentos, entrevistando usuários, sanando dúvidas sobre a regra de negócio da organização etc. Nesta etapa poderão ser desenvolvidos protótipos do sistema para uma melhor compreensão por parte do analista e do cliente.

- **Especificação de requisitos:** nesta etapa, o trabalho é conseguir colocar na forma de documentos do sistema todas as informações

coletadas no levantamento de requisitos. Podem ser incluídos os requisitos do usuário do sistema, e os do sistema, que é uma descrição mais detalhada das funcionalidades nele contidas.

- **Validação de requisitos:** nesta etapa é feita a verificação de todos os requisitos já levantados para testificar sua integralidade e pertinência. É comum nesta etapa a descoberta de inconsistência ou requisitos que ainda não estão completos. Eles devem ser corrigidos para que não haja problemas futuros.

Outra denominação para levantamento de requisitos é elicitação de requisitos, que tem a mesma função do levantamento de requisitos (SOMMERVILLE, 2010).

Figura 2.3 | Exemplo do processo de elicitação de requisitos



Fonte: <<http://progridbb.wikidot.com/engenhariadesoftware>>. Acesso em: 30 jul. 2017.

Como observado na Figura 2.3, a elicitação de requisitos também segue um ciclo que contém as fases de obtenção de requisitos, classificação e organização de requisitos, priorização e negociação de requisitos e documentação de requisitos.

Com todo este processo de engenharia de requisitos, não podemos esquecer que ela continua durante o sistema, pois podem aparecer novos módulos a serem desenvolvidos. Esses módulos precisaram entrar no ciclo de todo o processo, ou seja, elicitação de requisitos,

análise e projeto de sistemas, para só então começar a parte de implementação do novo módulo (SOMMERVILLE, 2010).



Para saber mais

Thread é um conceito muito utilizado na prática em várias aplicações de sistemas comerciais, videogame, aplicações em tempo real etc. Pesquise mais sobre: motivação na criação de *threads*, ciclo de vida de uma *thread*, operações de *thread*, modelos de *thread*, *threads* de usuário, *threads* de núcleo, combinação de *threads* de usuário e de núcleo, considerações sobre implementações de *threads* etc. Você também pode ver como funciona a execução de uma *thread* em diversos sistemas operacionais. Todas estas informações você encontra no livro Sistemas Operacionais ((DEITEL; CHOHNES, 2005, p. 90-109).

Atividades de aprendizagem

1. Durante as etapas do processo de engenharia de requisitos existe a preocupação na fase de elicitação. Ela deve ser feita visando a um bom detalhamento para que não tenham problemas de interpretação e ambiguidades. Estamos falando de qual etapa do processo de engenharia de requisitos? Assinale a alternativa correta.

- a) Gestão dos requisitos.
- b) Elicitação dos requisitos.
- c) Negociação dos requisitos.
- d) Levantamento dos requisitos.
- e) Validação dos requisitos.

2. Dado o desenvolvimento de um produto de engenharia de software, foi necessário a implementação de um novo requisito que não estava previsto. A equipe do projeto decidiu fazer alterações na documentação do sistema que já tinha sido aprovada. Considerando que a análise do sistema já estava adiantada, muitas mudanças já tinham sido realizadas, portanto a equipe de desenvolvimento esqueceu-se de incluir algumas informações de alteração do requisito novo. Isto acarretou uma inconsistência nas informações.

Percebe-se no texto que houve falha, principalmente, no processo de:

- a) Análise de requisitos.
- b) Priorização e negociação de requisitos.
- c) Classificação e organização de requisitos.

- d) Levantamento de requisitos.
- e) Gerenciamento de requisitos.

Seção 2

Requisitos funcionais e não funcionais

Introdução à seção

Os requisitos de sistemas de softwares são classificados como funcionais, não funcionais ou como de domínio.

Na realidade, a diferença entre esses três tipos de requisitos não é tão clara como parece, por exemplo, um requisito relacionado à proteção é um requisito de usuário, porém pode ser confundido com um requisito não funcional, mas quando desenvolvido em detalhe, pode levar a outros requisitos funcionais, como controle de acesso de um usuário ao sistema.

Vamos estudar as características e diferenças entre esses três tipos de requisitos de sistemas (SOMMERVILLE, 2010).

2.1 Requisitos funcionais

Os requisitos funcionais descrevem as funcionalidades e serviços que o sistema tem ou que espera-se que o sistema tenha no mínimo. Esses requisitos dependem do tipo de sistema que está sendo desenvolvido e dos usuários dele. Os requisitos de usuários, geralmente, são descritos de modo bastante geral, já os requisitos de sistema são mais detalhados, específicos e descrevem processos de entrada, saídas e exceções (SOMMERVILLE, 2010).

Figura 2.4 | Requisitos funcionais do sistema



Fonte: <<http://fmecontadores.blogspot.com.br/2014/03/>>. Acesso em: 30 jul. 2017.

Os requisitos de usuário relatam alguns recursos que são fornecidos pelo sistema.

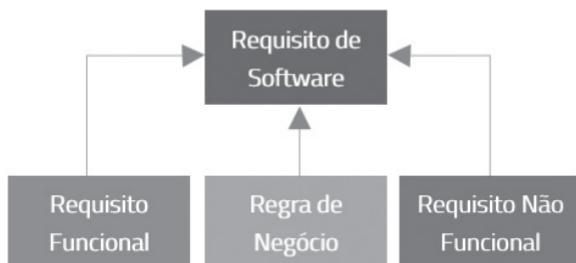
A especificação de requisitos é um fator de inúmeros problemas da engenharia de requisitos. É muito comum um desenvolvedor interpretar um requisito de duas formas para simplificar sua implementação, porém nem sempre é isto que o cliente deseja. Novos requisitos podem surgir a todo momento e é necessário que o sistema seja flexível, permitindo a realização de mudanças. Em contrapartida, isso aumenta os custos do projeto e pode atrasar sua entrega, contudo, é papel dos analistas e projetistas do sistema gerenciarem essas mudanças e implementações em toda a construção do produto.

A especificação de requisitos funcionais de um sistema deve ser completa. Deve-se procurar abstrair o mais profundo possível todas as funcionalidades do sistema. Todas a nível de usuário devem ser projetadas e garantidas pela engenharia de requisitos, todavia, é quase impossível que em sistemas complexos não haja alguns ajustes por meio de requisitos não especificados corretamente. Geralmente, isto só é descoberto em uma análise mais profunda.

Conforme os problemas de requisitos vão surgindo, eles também devem ser modelados e resolvidos durante as revisões ou outras fases do ciclo de vida do projeto.

Os requisitos não funcionais são tão importantes quanto os requisitos funcionais ou os requisitos de domínio. Infelizmente, existem muitas empresas que não levam isso a sério, e por este motivo muitos projetos de software das empresas fracassam (SOMMERVILLE, 2010).

Figura 2.5 | Requisitos funcionais



Fonte: <<http://www.ateomomento.com.br/o-que-e-um-requisito-nao-funcional/>>. Acesso em: 30 jul. 2017.

A Figura 2.5 mostra como os requisitos funcionais realmente fazem parte dos requisitos de softwares aliados aos requisitos não funcionais e regras de negócios de um sistema (SOMMERVILLE, 2010).



Questão para reflexão

Analise as duas questões a seguir, refletindo sobre como resolvê-las.

Como realizar a engenharia de requisitos de um sistema?

Como garantir que os requisitos funcionais e não funcionais sejam projetados corretamente?

Sabemos que quando trabalhamos com sistemas complexos, diversos problemas de **elicitação** de requisitos podem ocorrer.

2.2 Requisitos não funcionais

Quando falamos em requisitos não funcionais, fica um pouco mais difícil materializar isso na cabeça exatamente por conta deles não representarem algo concreto do ponto de vista de função ou funcionalidade que o software precisa realizar.

O que é um requisito não funcional, então? Como o próprio nome diz, é algo não funcional, ou seja, uma não funcionalidade. Isso quer dizer que não representa uma funcionalidade do sistema, mas exige-se que seja modelado, pois ele é realizado.

Existe uma definição que se propaga no meio de TI que diz que o requisito funcional define o que o software fará, e o requisito não funcional define como o software fará. Alguns analistas afirmam que um requisito não funcional mostra como um requisito funcional será implementado.

Na verdade, um requisito não funcional atende aos requisitos do sistema que não são implementados, ou seja, não são funcionalidades do sistema, porém fazem parte do escopo do sistema (SOMMERVILLE, 2010).

Um requisito não funcional muitas vezes está ligado a um requisito funcional, por exemplo, essa associação pode acontecer entre processos de integrações entre um sistema. Um sistema terá que se integrar com outro, fazendo a importação e ou exportação de dados.

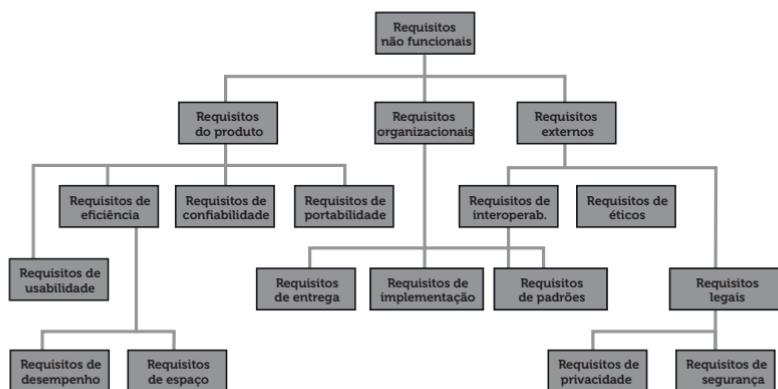
Em suma, toda necessidade que não puder ser atendida por meio de uma funcionalidade é considerada parte de um requisito não funcional.

Os usuários e desenvolvedores são condicionados apenas a pensar no negócio em termos de realização de tarefas. Eles estão pensando em requisitos funcionais e nas funcionalidades que o software tem que gerenciar. Somente empresas de grande porte, com usuários técnicos e operacionais mais conscientes, sabem reconhecer o devido valor dos requisitos não funcionais.

Os requisitos não funcionais são voltados a um conjunto de características de todo o sistema, e não somente a algumas características individuais. Significa que este conjunto de características dos requisitos não funcionais é mais importantes do que uma funcionalidade isolada dos requisitos funcionais (SOMMERVILLE, 2010).

Os requisitos não funcionais aparecem no projeto conforme as necessidades dos usuários em relação aos gastos, políticas organizacionais, necessidade de integração com outros sistemas de software ou hardware ou outros fatores externos ao projeto, como legislação sobre privacidade e regulamentos de segurança. O Figura 2.6 mostra a classificação dos requisitos não funcionais que podem aparecer no decorrer do projeto.

Figura 2.6 | Tipo de requisitos não funcionais



Fonte: <<http://www.devmedia.com.br/introducao-a-requisitos-de-software/29580>>. Acesso em: 30 jul. 2017.

- Requisitos de produtos: são requisitos comportamentais do produto. Os requisitos de desempenho são clássicos neste tipo de exemplo de requisitos de produto, pois ditam quanta memória o sistema precisa para operar etc.
- Requisitos organizacionais: são provenientes de procedimentos e políticas nas organizações do desenvolvedor e do cliente. Entre

alguns exemplos, pode-se citar os padrões de processo que são utilizados, os requisitos de implementação, que seriam a linguagem de programação adotada ou a metodologia de desenvolvimento aplicada e os requisitos de fornecimento que dizem quando o produto será entregue.

- Requisitos externos: são requisitos provenientes de fatores externos ao sistema, como o próprio nome já diz. Eles também têm relação com o processo de desenvolvimento. Os requisitos de interoperabilidade, requisitos legais se destacam (SOMMERVILLE, 2010).



Para saber mais

Análise de sistemas é o estudo dos processos na busca de encontrar a melhor maneira para que o projeto se desenvolva corretamente. O analista de sistemas é responsável por materializar a realidade do cliente por meio da abstração do cenário que será desenvolvido. Todas estas informações você encontra no livro UML Guia do usuário (2012).

Atividades de aprendizagem

1. No processo de engenharia de requisitos, existem os tipos de requisitos de usuário e requisitos de sistema. Assinale a alternativa que condiz com os requisitos de usuário e de sistema, respectivamente.
 - Apenas funcionais; apenas não funcionais.
 - Apenas não funcionais; apenas funcionais.
 - Apenas funcionais; funcionais e não funcionais.
 - Funcionais e não funcionais; apenas não funcionais.
 - Funcionais e não funcionais; funcionais e não funcionais.
2. Existem os tipos de requisitos funcionais, os não funcionais e os de domínio. Ao determinar os requisitos de um projeto é necessário considerar todos estes tipos. Os requisitos não funcionais:
 - Definem com detalhes exatamente o que deve ser implementado.
 - Definem explicitamente as funções que o sistema não deve executar.
 - Indicam os serviços que o sistema deve prestar.
 - Representam restrições aos serviços oferecidos pelo sistema.
 - São descrições de que serviços o sistema deve fornecer aos usuários.

Seção 3

Requisitos funcionais e não funcionais

Introdução à seção

Uma classe é a descrição de um conjunto de objetos do mundo real que compartilha uma lista de características e comportamentos em comum e também implementa uma ou mais interfaces. As classes são abstrações do domínio do problema ou parte dele. Elas podem ser utilizadas para representar itens de software, hardware e até conceituais.

Vamos estudar as características das classes e aprender um pouco sobre os diagramas de classes e seus refinamentos.

3.1 Introdução ao diagrama de classes

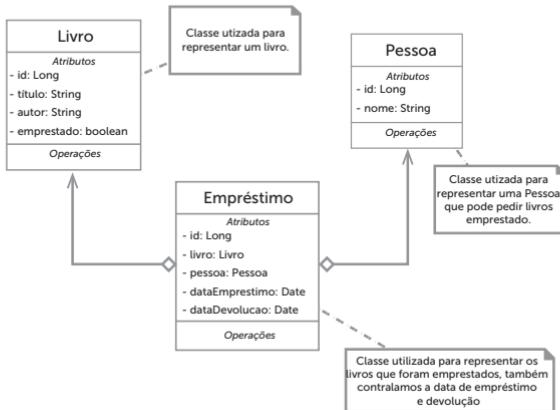
Os diagramas de classes são mais frequentemente encontrados na modelagem de sistemas orientado a objetos e mostram um conjunto de classes, interfaces, colaborações e seus relacionamentos. Eles são base para outros diagramas da UML, como diagrama de implantação e de componentes.

O diagrama de classe é importante para modelagem de sistemas, pois assim como a construção de um veículo possui características em comum, como cor, tipo do motor, categoria etc. e comportamentos, como andar e parar, o mesmo acontece com o diagrama de classe.

Para modelar qualquer objeto do mundo real dentro de um sistema, no paradigma orientado a objetos utilizamos as classes para agrupar esses objetos e ter uma documentação por meio da modelagem de software (SOMMERVILLE, 2010).

A Figura 2.7 mostra um exemplo básico de um diagrama de classe contendo as classes **Livro**, **Pessoa** e **Empréstimo**.

Figura 2.7 | Tipo de requisitos não funcionais



Fonte: <<https://goo.gl/Ngckyq>>. Acesso em: 30 jul. 2017.



Questão para reflexão

Analise as duas questões a seguir refletindo como resolvê-las.

É possível construir um diagrama de classe completo com todas as classes do sistema?

O diagrama de classe é o diagrama inicial para modelagem de um sistema?

Sabemos que quando trabalhamos com diagramas de classes complexos, outros diagramas são necessários para complementar a modelagem do sistema

3.2 Propriedades básicas

Um diagrama de classes tem sempre um nome que é relativo ao problema que está sendo modelado, por exemplo, está sendo modelado um sistema de compras. É possível termos um diagrama de classes com o nome de pedidos ou diagrama de pedidos e assim por diante.

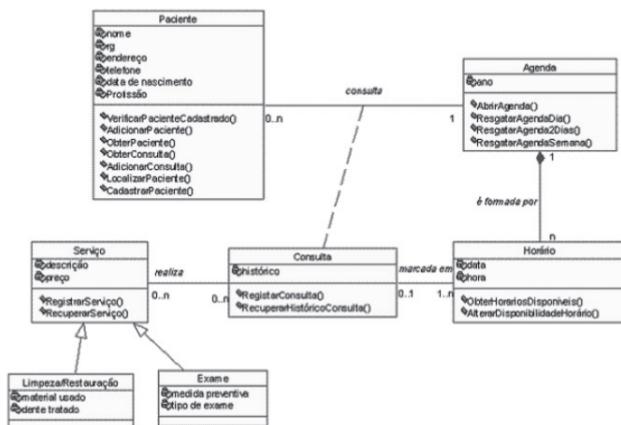
Um diagrama de classes é um tipo especial de diagrama que compartilha as mesmas propriedades de todos os outros diagramas. A diferença com relação aos outros é o seu conteúdo.

Os diagramas de classes geralmente costumam conter alguns itens:

- Classes.
- Interfaces.
- Relacionamentos de dependência, generalização e associação.

Os diagramas de classe, assim como os outros diagramas da UML, podem conter notas e restrições. Alguns também podem ter subsistemas para um melhor agrupamento de elementos do seu modelo em um conjunto maior (SOMMERVILLE, 2010).

Figura 2.8 | Diagrama de classe na ferramenta rational rose



Fonte: <http://www.macoratti.net/net_uml1.htm>. Acesso em: 29 jul. 2017.

A Figura 2.8 traz um diagrama de classe apresentando relacionamentos associativos de herança, entre outros na ferramenta Rational Rose (SOMMERVILLE, 2010).



Para saber mais

Os diagramas de componentes e os diagramas de implantação são semelhantes aos diagramas de classe, exceto pelo fato que, em vez de conterem classes, eles contêm componentes e nós, respectivamente.

3.3 Técnicas de refinamento

Os diagramas de classes são utilizados para fazer a modelagem da visão estática de um sistema, visão esta que serve de auxílio para os requisitos funcionais de um sistema, ou seja, as funcionalidades do

sistema que deverão ser entregues ao usuário com o sistema rodando.

Mediante a isto, algumas técnicas de refinamento de transformação existentes, que utilizam o conceito de refinamento, podem ser realizadas de formas diferentes por meio dos atributos ou das operações (SOMMERVILLE, 2010).

Por meio dos atributos podemos ser:

- Adicionar novos atributos à classe refinada.
- Transformar um atributo em uma nova classe, podendo estar um conjunto de novos atributos.
- Transformar os atributos em um resultado refinado.
- Transformar uma operação em mais operações.

Por meio das operações podemos:

- Acrescentar operações novas a uma classe.
- Fazer a transformação de uma em mais operações.

Atividades de aprendizagem

1. Em UML, os dois tipos de diagramas de interação são denominados de:

- a) Diagrama de classes e diagrama de sequência.
- b) Diagrama de colaboração e diagrama de classes.
- c) Diagrama de componentes e diagrama de classes.
- d) Diagrama de sequência e diagrama de componentes.
- e) Diagrama de sequência e diagrama de colaboração.

2. São diagramas utilizados pela UML, EXCETO:

- a) Diagrama de estado.
- b) Diagrama de classe.
- c) Diagrama de conformidade.
- d) Diagrama de colaboração.
- e) Diagrama de atividade.

Fique ligado

Resumo dos itens estudado nesta unidade:

- Especificação do software.
- Requisitos funcionais e requisitos não funcionais.
- Diagrama de classe.

Para concluir o estudo da unidade

O processo de análise e projeto de software é muito importante em toda a construção de um sistema computacional.

Deve-se levar em consideração a visão dos *stakeholders* para elicitação de requisitos no que diz respeito a tentar abstrair todas as necessidades do sistema, considerando os requisitos funcionais, não funcionais e de domínio, garantindo assim um alto percentual de acerto na criação da análise e projeto do software.

Por fim, a implementação de todos os requisitos levantados e analisados vai garantir o sucesso do produto e a satisfação não só do cliente, mas de toda a equipe técnica e operacional responsável pela construção do sistema.

Atividades de aprendizagem da unidade

1. A UML é uma linguagem ou notação de diagramas para especificar, visualizar e documentar modelos de software orientados a objetos. Qual das alternativas NÃO é um tipo de diagrama utilizado em UML? Assinale a alternativa correta.

- a) Diagrama de classes.
- b) Diagrama de sequência.
- c) Diagrama de trabalhos.
- d) Diagrama de sequência.
- e) Diagrama de estados.

2. Ao iniciar a modelagem de um software que será construído sob o paradigma da orientação a objetos, o analista de sistemas decidiu utilizar a UML (Unified Modeling Language) para representar a estrutura do software. Qual dos seguintes diagramas será escrito pelo analista para representar a estrutura (classes, atributos e métodos) e as relações entre as classes que irão compor o software? Assinale a alternativa correta.

- a) Diagrama de comunicação.
- b) Diagrama de sequência.
- c) Diagrama de classes.
- d) Diagrama de estados.
- e) Diagrama de tempo.

3. Na análise de requisitos, confiabilidade e usabilidade são partes constituintes de qual atributo da engenharia de requisitos? Assinale a alternativa correta.

- a) Requisitos funcionais.
- b) Requisitos de domínio.
- c) Dependências.
- d) Regras de negócio.
- e) Requisitos não funcionais.

4. O modelo de requisitos define um conjunto completo de classes de análise. De acordo com a evolução do projeto, a equipe define este conjunto de classes de análise. Segundo os conceitos de análise e projeto orientado a objetos, qual não é considerado um tipo específico de classes de projeto? Assinale a alternativa correta:

- a) Classes de entidade.
- b) Classes de processo.
- c) Classes do domínio de negócio.
- d) Classes de interface de usuário.
- e) Classes de projeto.

5. De acordo com os requisitos funcionais e não funcionais, podemos dizer que os requisitos funcionais focam na funcionalidade do sistema. Assinale a alternativa que seja um requisito funcional para um sítio de comércio eletrônico.

- a) O sistema deve ser acessível preferencialmente por meio do navegador Mozilla Firefox.
- b) O sistema deve autorizar a compra por cartão de crédito em três segundos, no máximo.
- c) O sistema deve permitir ao usuário a escolha da forma de pagamento dos produtos.
- d) O sistema deve ser capaz de atender até 100.000 usuários simultâneos.
- e) O sistema deve ser capaz de realizar 50.000 conexões, no mínimo.

Referências

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **Uml guia do usuário**. Rio de Janeiro: Campus, 2012.

DEITEL, H.; DEITEL, P.J.; CHOFFNES, D. R. **Sistemas operacionais**. 3. ed. São Paulo: Pearson Prentice Hall, 2005.

DI. **Dicionário Informal**. 2008. Disponível em: <<http://www.dicionarioinformal.com.br/>>. Acesso em: 17 ago. 2017.

FERREIRA, Aurélio B. H. **Dicionário Aurélio da Língua Portuguesa**. 5. ed. São Paulo: Editora Positivo, 2014.

KOSCIANSKI, A.; SOARES, M. **Qualidade de software**: aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software. 2. ed. São Paulo: Novatec, 2007.

SOMMERVILLE, I. **Projeto de sistemas**. São Paulo: Pearson, 2010.

O paradigma orientado a objetos e a UML

Adriano Sepe

Objetivos de aprendizagem

- Compreender os conceitos centrais para a modelagem orientada a objetos, incluindo os conceitos relacionados a classes, classes avançadas, interfaces, pacotes e outros. Através desses conceitos apresentaremos algumas práticas possíveis para elucidação e complementação.
- Entender a função da Linguagem Unificada de Modelagem, UML, e como ela ajudará no processo de visualização, especificação, construção e documentação de artefatos de sistemas de software, compreendendo como proporciona uma forma padrão para a preparação e planejamento de arquiteturas de projetos de sistemas, contemplando os aspectos conceituais, lógicos e concretos.

Seção 1 | Conceitos de orientação a objetos

Compreender os detalhes de modelagem orientada a objetos, discutindo conceitos relacionados a classes, classes avançadas, interfaces, pacotes e outros.

Seção 2 | Introdução à UML

Entender a função da Linguagem Unificada de Modelagem, UML, e como ela ajudará no processo de visualização, especificação, construção e documentação de artefatos de sistemas de software.

Introdução à unidade

Vamos estudar, nesta unidade, conceitos que definirão importantes recursos disponíveis na Linguagem Unificada de Modelagem, UML, além dos conceitos poderemos visualizar as representações gráficas, e assim nos balizarmos para construir nossos diagramas.

Na primeira seção, abordaremos os detalhes de modelagem orientada a objetos, discutindo conceitos relacionados a classes, classes avançadas, interfaces, pacotes e outros. Esses conceitos são fundamentais para análise e projeto orientado a objetos, bem como para a programação, visto que boa parte das linguagens, atualmente, suporta o paradigma orientado a objetos.

Na segunda seção, abordaremos a função da UML e como ela nos ajuda possibilitando a visualização, especificação, construção e documentação de artefatos de sistemas de software. Além disso, abordaremos os conceitos que permitem que a linguagem UML seja um padrão para a preparação e planejamento de arquiteturas de projetos de sistemas, contemplando os aspectos conceituais, lógicos e concretos. Também traremos observações relacionadas à utilização efetiva dessa linguagem, aproveitem!

Seção 1

Conceitos de orientação a objetos

Introdução à seção

Olá aluno, no transcorrer desta seção, vamos compreender os detalhes de modelagem orientada a objetos, discutindo conceitos relacionados a classes, classes avançadas, interfaces, pacotes e outros.

1.1 Classes

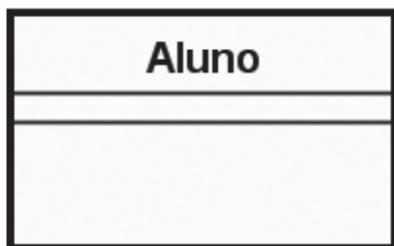
Utilizamos as classes como blocos de construção, com alto grau de importância para qualquer sistema orientado a objetos. Através da classe descrevemos um conjunto de objetos que compartilham recursos e semânticas em comum. Esses recursos correspondem a todo e qualquer elemento pertencente à classe, eles podem variar entre atributos, operações e relacionamentos. Uma classe poderá implementar uma ou mais interfaces, pois pode assumir contratos (as interfaces serão discutidas adiante). Utilizamos as classes para construir o vocabulário do sistema em desenvolvimento, isso quer dizer que boa parte dos assuntos relativos à análise e desenvolvimento do projeto se norteará através dos nomes identificados como classes, visto a relevância existente. As classes podem incluir abstrações que são parte do domínio do problema, assim como classes que efetuam implementações baseadas em contratos, como as interfaces. Utilizamos as classes para representar itens de software, hardware e até os que sejam puramente conceituais (PRESSMAN, 2011).

Uma correta definição das classes proporciona maior equilíbrio sobre as responsabilidades e limites existentes entre os elementos computacionais que serão desenvolvidos para atender às demandas do sistema.

Para construção de uma classe, devemos inicialmente definir um nome que a diferencie de outras. Assim, utilizamos uma sequência de caracteres, podendo variar entre nomes simples, nos quais utilizamos apenas os substantivos para nomeação, ou qualificados, em que

utilizamos além do nome da classe um prefixo com o nome do pacote a que a classe pertence (PRESSMAN, 2011).

Figura 3.1 | Classe aluno



Fonte: elaborada pelo autor.

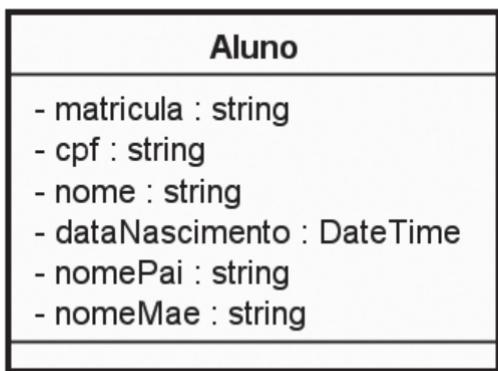
Como exemplo, representaremos a classe Aluno, a qual utilizaremos para evolução dos conceitos nesta primeira seção. Perceba que se estivéssemos modelando um sistema acadêmico, com certeza essa seria uma das principais classes.

1.1.1 Atributos

Como recurso fundamental para as classes, o atributo é uma propriedade nomeada de uma classe, que deve descrever um intervalo de valores que as instâncias da propriedade podem representar. As classes podem ter qualquer número de atributos ou mesmo nenhum. Um atributo representa uma propriedade do item que está sendo modelado, compartilhado por todos os objetos da classe. Como exemplo, podemos avaliar que para a classe Aluno, seja de instituição pública ou privada, ensino médio ou superior, de forma generalizada, todo aluno possui número de matrícula, nome, endereço, data de nascimento e telefone. Devemos compreender que cada atributo define uma característica que todos os objetos que pertencem a uma classe possuem. Em nosso exemplo estamos assumindo que todos os alunos possuem tais características e essas são distintas por indivíduo. Perceba que estamos generalizando, mas compreendemos que existe a individualidade (SOMMERVILLE, 2007).

Um atributo é, portanto, uma abstração do tipo de dados ou estados que os objetos de uma classe podem abranger. Assumindo um espaço de tempo, compreendemos que um objeto de uma classe terá valores específicos para cada atributo, podendo variar perante os demais.

Figura 3.2 | Incluindo atributos



Fonte: elaborada pelo autor.

Os atributos definem as características que os objetos possuiram, e em nosso exemplo destacamos alguns que possivelmente um aluno possuiria, a saber: matrícula, cpf, nome, nome Pai e nome Mae são privados, e isso quer dizer que estão restritos ao próprio objeto. Devemos destacar que todos os atributos foram definidos como string, pois representam uma cadeia de caracteres (vetor de caractere). Já o atributo data Nascimento abstraímos como um atributo privado e do tipo Date Time. Possivelmente, encontraremos um tipo adequado para representar Data/Hora na linguagem selecionada para implementação.

1.1.2 Operações

As operações são tratadas como serviços que podem ser solicitados, ou até mesmo invocados para um objeto em específico, e é esperado que isso gere uma mudança interna no objeto que está executando essa ação. Colocando de outra maneira, uma operação é uma abstração de algo que pode ser feito em um contexto de um objeto, afetando apenas ele, mas que é compartilhada por todos os objetos dessas classes. Uma classe pode ter qualquer número de operações ou até não ter nenhuma.

Da mesma maneira que os atributos, compreendemos que todos os objetos de uma classe podem executar uma determinada operação, mas isso ocorrerá a partir de um objeto em específico. Como exemplo,

podemos destacar o método Finalizar Pedido pertencente à classe Carrinho Compra, construído para um e-commerce. Entenda que essa ação, quando executada, não afetará todos os carrinhos de compra ativos, mas apenas aquele que está sendo manipulado por um usuário através do navegador (SOMMERVILLE, 2007).

Figura 3.3 | Adicionando operações

Aluno	
- matricula	: string
- cpf	: string
- nome	: string
- dataNascimento	: DateTime
- nomePai	: string
- nomeMae	: string
+ solicitarRematricula()	: boolean
+ solicitar2ViaBoleto()	: Boleto
+ recuperarPendenciasFinanceiras()	: ListaDePendencias

Fonte: elaborada pelo autor.

Seguindo o nosso exemplo, imaginamos aqui algumas operações que cada indivíduo de nossa classe deverá ser capaz de executar. Perceba que cada operação pode gerar um resultado, que chamamos de retorno. Outro ponto é o caractere + (mais), que define também a visibilidade, que neste caso é pública, indicando que poderão ser utilizados por clientes externos à classe. Entenda que cada uma dessas ações poderá ser executada no contexto de um aluno, em outras palavras, podemos dizer que para que uma rematrícula seja solicitada, deveremos partir de um aluno em específico, e isso é algo razoável, pois a rematrícula, segunda via de um boleto e pendências financeiras não serão executadas para todos, apenas para os alunos que solicitarem.

1.1.3 Responsabilidade

A responsabilidade deve ser compreendida como um contrato ou obrigações de uma determinada classe. Quando criamos uma classe, estamos também declarando que todos os objetos dela têm os mesmos atributos, tipo de estado e tipo de comportamento. Analisando de forma mais abstrata, esses atributos e operações correspondentes são apenas as características necessárias para atender às responsabilidades de uma classe. Entenda que por questões racionais, uma classe

possuirá apenas as informações pertencente a ela, bem como disporá das ações que lhe pertencem, então, se analisarmos a classe CarrinhoCompra, comentada lá nas operações, entendemos que um objeto gerado a partir dessa classe será responsável por conhecer o seu número, assim como será sua responsabilidade iniciar o processo de comunicação de compra, para que a mercadoria seja separada para entrega. Estamos, neste exemplo, olhando de forma superficial um processo complexo, mas o que precisamos evidenciar é o fato de que as classes definem suas responsabilidades, e os objetos exercemativamente essas responsabilidades (PRESSMAN, 2011).

1.2 Classes avançadas

Conforme já mencionado, as classes realmente são os blocos de construção mais importantes de qualquer sistema orientado a objetos. É fato que as classes são apenas um dos tipos de um bloco de construção ainda mais geral existente na UML: os classificadores. Através do classificador podemos descrever características estruturais e comportamentais. Eles incluem: classes, interfaces, tipos de dados, sinas, componentes, nós, casos de uso e subsistemas.

Além da propriedade mais simples, como os atributos e operações, definidos anteriormente os classificadores, principalmente através das classes, possuem características avançadas, pois podemos modelar a multiplicidade, a visibilidade, as assinaturas, o polimorfismo e outras características. Podemos, através da UML, definir a modelagem da semântica de uma classe, estabelecendo seu significado em qualquer grau da formalidade desejado (BOOCH; RUMBAUGH; JACOBSON, 2012).

Como existem vários tipos de classificadores de classes, devemos escolher aqueles que sejam mais adequados à modelagem da abstração do mundo real.

1.2.1 Classificadores

O classificador deve ser compreendido como um mecanismo para descrever características estruturais e comportamentais. Como já mencionado, eles incluem as classes, interfaces, tipos de dados, sinais,

componentes, nós, casos de uso e subsistemas.

Quando fazemos uma modelagem, abstraímos e definimos itens que pertencem ao mundo real, ou itens que existem apenas na sua solução. Como exemplo, podemos reconstruir a ideia do sistema de e-commerce, no qual abstraímos classificadores do tipo classe Cliente, que por ordem fazem parte do mundo real e executam ações de solicitação de produtos, mas também abstraímos a classe CarrinhoPedido ou Sacola, que não existem fisicamente, mas representam um processo. Cada uma dessas abstrações terá instâncias. Separar a essência e a instância dos itens em seu mundo é uma parte importante da modelagem (BOOCH; RUMBAUGH; JACOBSON, 2012).

A classe, como já mencionado, representa um tipo extremamente importante para a UML, porque a modelagem orientada a objetos só é efetiva se conseguirmos observar e abstrair corretamente através das tão famosas classes. Uma classe é definida como uma descrição de um conjunto de objetos que compartilha os mesmos atributos, operações, relacionamentos e semântica. A UML oferece vários outros tipos de classificadores para ajudá-lo a fazer a modelagem.

- Interface: usamos para definir uma coleção de operações utilizadas para especificação do serviço de uma classe ou de um componente. Em vias gerais, a interface comporta-se como um contrato, seja para quem implementa ou para quem utiliza desse contrato para interação, isso através de unidades clientes.
- Tipos de dados: elementos mais primários que não possuem identidade, incluindo tipos primitivos predefinidos (como números ou sequência de caracteres), além de tipos enumerados (como booleano). Como observado, não farão parte do vocabulário do sistema, pois existem mais no âmbito da programação do que no modelo de negócio.
- Associação: descreve o conjunto de vínculos em que cada um relaciona dois ou mais objetos. Fazendo uso do conceito de objetos se ajudam para executar tarefas, quando há algum tipo de cooperação, então, utilizamos o termo associação.
- Sinal: especifica um estímulo assíncrono, comunicado entre as instâncias.
- Componente: compreendemos como uma parte modular de um sistema que utiliza técnicas para ocultar a sua implementação,

permitindo sua operação através de interfaces externas.

- Nó: representa um recurso computacional que será utilizado em tempo de execução, geralmente dispõe de alguma memória e frequentemente com capacidade de processamento.
- Casos de uso: descrevemos um conjunto de uma sequência de ações, incluindo variantes, que um sistema realiza, proporcionando um resultado observável do valor para um determinado ator.
- Subsistema: assume características de um componente, mas representando uma parte do sistema (BOOCH; RUMBAUGH; JACOBSON, 2012).

1.2.2 Visibilidade

Quando especificamos os atributos para a orientação a objetos, em que a segurança e restrições de acesso são fundamentais, podemos e devemos especificar a visibilidade dos membros de uma classe. Apenas alguns tipos de classificadores podem utilizar-se da visibilidade para uma correta especificação. Podemos, através da UML, definir qualquer um dos quatro níveis de visibilidade (BOOCH; RUMBAUGH; JACOBSON, 2012):

- Público: qualquer classificador poderá acessar uma característica com visibilidade pública; especificado pode ser antecedido pelo símbolo +.
- Protegido: os descendentes do classificador são capazes de usar a característica; especificado pode ser antecedido pelo símbolo #.
- Privado: apenas próprio classificador, é capaz de usar a característica; especificado por ser antecedido pelo símbolo -.
- Pacote: todos os classificadores declarados no mesmo pacote podem usar a característica; especificado por ser antecedido pelo símbolo ~.

Figura 3.4 | Utilizando modificadores de visibilidade

Aluno
<pre>- matricula : string - cpf : string - nome : string - dataNascimento : DateTime - nomePai : string - nomeMae : string - gerarMatricula() : string + solicitarRematricula() : boolean + solicitar2ViaBoleto() : Boleto + recuperarPendenciasFinanceiras() : ListaDePendencias # permiteRematricular() : boolean</pre>

Fonte: elaborada pelo autor.

Utilizamos alguns modificadores em nosso exemplo, privado, público e protegido. Deixemos aqui uma justificativa para tal adaptação, os atributos privados são responsabilidade exclusiva da classe, já os comportamentos públicos poderão ser acionados por clientes externos à classe, e os protegidos serão acessíveis apenas à classe e seus descendentes, assumindo que a geração das matriculas poderá ser necessária para as classes filhas.

1.2.3 Escopo de instância e de estática

Podemos ainda especificar um outro importante detalhe para os atributos e operações de um classificador, esse detalhe se refere ao respectivo escopo. Através do escopo de uma específica, podemos determinar se uma instância do classificador tem seu próprio valor distinto da característica, ou então se haverá um único valor (compartilhando) da característica para todas as instâncias do classificador. Dessa forma, podemos, através da UML, especificar dois tipos de escopo do proprietário, assim chamados por se tratar de membros que pertencem a um classificador. Já o escopo é exercido a partir desse membro, então podemos naturalmente chamá-lo de escopo do proprietário. Não confunda com escopo de visibilidade, termo muito utilizado para o conceito de visibilidade discutido anteriormente (PRESSMAN, 2011).

- Instância: toda instância do classificador possui o seu próprio

valor para a característica. Esse é o mais comum, tratamos como padrão e não requer notação adicional.

- Estática: haverá apenas um valor da característica para todas as instâncias do classificador. Podemos ainda tratá-lo por escopo de classe. Sua notação é o sublinhado.

Figura 3.5 | Métodos escopo da classe

Aluno
<pre>- matricula : string - cpf : string - nome : string - dataNascimento : DateTime - nomePai : string - nomeMae : string</pre>
<pre># gerarMatricula() : string + novoAluno() : Aluno + solicitarRematricula() : boolean + solicitar2ViaBoleto() : Boleto + recuperarPendenciasFinanceiras() : ListaDePendencias # permiteRematricular() : boolean</pre>

Fonte: elaborada pelo autor.

Consideraremos através desse exemplo que a geração da matrícula é algo que pertence ao coletivo, e restrito à classe que a declara e seus descendentes. Já a criação de um novo aluno pertencerá também à classe, pois envolve uma ação relativa ao coletivo e não especificamente ao indivíduo.

Conforme já observado, a maioria das características dos classificadores em sua modelagem terá seu escopo definido pela instância, isso porque a relação que abstraímos através do coletivo, embora os indivíduos possuam seus próprios valores, determinará exatamente a máxima de que o escopo de instância satisfará a maioria dos casos (PRESSMAN, 2011).

É comum que características com escopo estático possuam visibilidade privada, permitindo assim que apenas o conjunto de instâncias da classe possa acessá-los, como exemplo podemos destacar os casos de gerar códigos únicos para novas instâncias de uma classe.

Para operações o escopo estático funciona um pouco diferente,

pois uma operação de instância tem um parâmetro implícito correspondente ao objeto que está sendo manipulado. Já uma operação estática não possui esse parâmetro, pois representa o indivíduo e nesse caso o contexto é o coletivo, então, o comportamento é semelhante a um procedimento global tradicional que não tem objeto-alvo. Como exemplo, podemos destacar que é comum que as operações estáticas sejam utilizadas para operações que criam instâncias ou que manipulam atributos estáticos (PRESSMAN, 2011).

1.2.4 Multiplicidade

Uma classe pode variar quanto à quantidade permitida ou até esperada de instâncias. É razoável assumir que poderá haver qualquer quantidade de instâncias dessa classe. Isso não se aplica naturalmente para uma classe abstrata, pois não haverá qualquer instância direta, mas podendo haver qualquer quantidade de instâncias de suas classes-filhas. Existem casos em que desejamos restringir a quantidade de instâncias que uma classe poderá ter. É fato que existem situações em que não há justificativas para a existência de objetos de uma classe, caracterizando como zero instância, isso se aplica para classes utilitárias. Essas classes funcionam basicamente como um agrupador de comportamentos, podendo no máximo dispor de atributos estáticos. Encontramos muitos casos de restrições para apenas uma instância, isso até origina um padrão de projeto de software chamado Singleton. Concluímos que o caso mais comum será que poderemos definir um número específico de instâncias ou muitas instâncias (BOOCH; RUMBAUGH; JACOBSON, 2012).

Chamamos de multiplicidade o número de instâncias que uma classe pode ter. A multiplicidade especifica o intervalo permitido de cardinalidade que uma entidade poderá assumir. Para especificar a multiplicidade através da UML, devemos escrever a expressão de multiplicidade de uma classe, isso no canto superior direito do ícone da classe. Podemos ainda aplicar a multiplicidade para os atributos, para isso devemos escrever a expressão adequada da multiplicidade entre colchetes logo após o nome do atributo.

1.3 Interface

Através das interfaces conseguimos estabelecer uma separação entre a especificação do que uma abstração realiza e a implementação propriamente dita. Assim, separando esses conceitos, por estarem relacionados através de uma relação de implementação, podemos e devemos exercer tal organização, e isso trará vantagens significativas para o projeto arquitetural. Podemos definir uma interface como uma coleção de operações utilizadas para especificar um serviço de uma classe ou de um componente (PRESSMAN, 2011).

Empregamos as interfaces para visualizar, especificar, construir e documentar a coesão interna do sistema. Podemos utilizar os tipos e as funções (ou papéis) para modelar a conformação estática e dinâmica da interface em um contexto específico.

Em uma interface bem estruturada, observamos que há uma clara separação entre a visão externa e a visão interna de uma abstração. Possibilitando o entendimento e utilização da abstração (visão externa) sem o aprofundamento sobre os detalhes da implementação (visão interna). Isso é um poderoso recurso existente na programação orientada a objetos, em que conseguimos abstrair a interface pública dos detalhes de processamento.

Declarando a interface, podemos estabelecer o comportamento desejado de uma abstração, independentemente de qualquer implementação dessa. Os clientes dela podem desenvolver suas lógicas baseadas no contrato, e a implementação pode ser feita de forma independente, desde que o contrato seja atendido, satisfazendo às responsabilidades e ao contrato denotado pela interface.

Como este conceito torna o processo de desenvolvimento de soluções algo mais robusto e coeso, muitas são as linguagens de programação com suporte para o conceito de interfaces, inclusive Java, C#, Objective C e CORBA IDL. Como vantagem, as interfaces apresentam-se tanto para permitir a divisão entre a especificação e a implementação de uma classe ou componente, como também quando trabalhamos em projetos maiores, pois o uso de interfaces para especificação permitirá uma visão externa de um pacote ou subsistema (BOOCH; RUMBAUGH; JACOBSON, 2012).

Como já definido, uma interface nada mais é do que uma coleção

de operações empregadas para especificar um serviço de uma classe ou de um componente. Diferentemente das classes e dos tipos de dados, as interfaces não especificam qualquer implementação (portanto, poderão não incluir métodos na perspectiva de instruções a serem executadas). Como observamos com relação às classes, uma interface poderá definir qualquer número de operações. Poderemos definir características avançadas, como propriedades de visibilidade, propriedade de concorrência, estereótipos, valores atribuídos e restrições (BOOCH; RUMBAUGH; JACOBSON, 2012).

Podemos utilizar as interfaces para criar relacionamentos de generalização, associação e dependência e ainda utilizá-las para relacionamentos de realização. Adiantando o conceito, a realização é um relacionamento semântico entre dois classificadores, em que um especifica um contrato cuja execução é assegurada pelo outro classificador.

Dessa forma, concluímos com o seguinte entendimento: uma interface especifica o contrato para uma classe ou componente sem determinar sua implementação. Uma classe ou componente poderá realizar várias interfaces.

Figura 3.6 | Utilizando Interface



Fonte: elaborada pelo autor.

Através da interface, conforme já observamos, pode-se definir um contrato, o qual, quando implementado por uma classe, garantirá que seja atendido, permitindo assim que haja herança baseada em comportamento, garantindo que alcancemos um outro grau de abstração.



Para saber mais

Para saber mais a respeito do uso de interfaces, separamos um ótimo artigo que aborda como a herança múltipla pode ser alcançada em linguagens que não suportam tal recurso, isso tudo através das interfaces. Esse artigo chama-se *Utilizando interfaces* e poderá ser acessado através do seguinte link: <<http://www.linhadecodigo.com.br/artigo/80/utilizando-interfaces.aspx>>. Acesso em: 29 set. 2017.



Questão para reflexão

Quando devemos utilizar o escopo de instância ou de estática na modelagem de uma classe?

Atividades de aprendizagem

1. Utilizamos como blocos de construção, com alto grau de importância para qualquer sistema orientado a objetos, e através dela descrevemos um conjunto de objetos que compartilham recursos e semânticas em comum. Esses recursos correspondem a todo e qualquer elemento pertencente a ela, e podem variar entre atributos, operações, relacionamentos. Essas assertivas correspondem a qual conceito listado?

- A) Operação.
- B) Classe.
- C) Classificador.
- D) Visibilidade.
- E) Multiplicidade.

2. A visibilidade de um membro de uma classe, seja um atributo ou método, é, sem sombra de dúvida, um importante recurso do paradigma orientado a objetos. Ao interpretarmos a função da visibilidade privada, compreendemos que ele aplicará a maior restrição de visibilidade. Qual a razão de definirmos um membro privado, sendo que ele terá um escopo de utilização muito limitado?

Seção 2

Introdução à UML

Introdução à seção

Olá aluno! Nesta seção, vamos entender a função da Linguagem Unificada de Modelagem, UML, e como ela nos ajudará no processo de visualização, especificação, construção e documentação de artefatos de sistemas de software.

2.1 Visão geral

Utilizamos a linguagem UML com os objetivos de visualizar, especificar, construir, documentar os artefatos de um sistema complexo de software.

Como a UML apresenta-se com uma linguagem, devemos compreender o seu vocabulário e regras, sendo que essas serão utilizadas para efetuar combinações entre os vocabulários e então comunicar algo. Utilizamos essa linguagem de modelagem para que através do seu vocabulário e regras possamos representar um sistema nas formas conceituais e físicas. Portanto, quando desejamos elaborar a estrutura de um projeto de software, podemos utilizar a linguagem de modelagem UML, e ainda, assumindo que haja uma adoção da indústria de tecnologia por esse padrão, estariam aptos a trabalhar em quaisquer projetos de software (BOOCH; RUMBAUGH; JACOBSON, 2012).

Através da modelagem conseguimos compreender um sistema. Ainda que nenhum modelo seja inteiramente suficiente, haverá sempre a necessidade de outros modelos, conectados entre si, para tornar possível entender qual aspecto, ainda que seja o sistema mais trivial.

Em sistemas que fazem uso intenso de software, torna-se fundamental que uma linguagem seja capaz de abranger as diferentes visões relacionadas à arquitetura do sistema, e como essa arquitetura evolui ao longo do ciclo de vida de desenvolvimento de software.

A linguagem UML define o vocabulário e as regras que indicam como criamos ou lemos os modelos bem formados (consistentes),

mas há na linguagem o indicativo de que modelos deveremos criar, também não há definição do momento dessa criação. Perceba que a UML define os modelos e como devemos criá-los e então combiná-los, mas não define como devemos utilizá-los no ciclo de desenvolvimento. Essa tarefa cabe ao processo de desenvolvimento do software. Quando possuímos um processo bem definido, isso nos servirá como guia para decidir quais artefatos serão produzidos, quais atividades e quais colaboradores serão envolvidos para criá-los e gerenciá-los, e ainda como esses artefatos serão empregados para medir e controlar o projeto como um todo (BOOCH; RUMBAUGH; JACOBSON, 2012).

Para o desenvolvimento de uma solução computacional, estamos tentados a enxergar as etapas de análise, projeto e implementação, como uma coisa só. Pensamos no requisito e transformamos em código. Existem realmente situações em que a melhor forma de especificar algo ou um processo é exatamente através do código, isso porque o texto é uma forma eficiente e direta para escrever expressões e algoritmos.

Nesse tipo de abordagem o desenvolvedor está definindo alguma modelagem, mesmo que seja em memória, mas é óbvio que ele precisará avaliar as estruturas de dados e de armazenamento que serão necessárias para informatização de algum processo. Essa modelagem pode até ser feita de forma manual, utilizando para isso papel ou ferramentas simples de desenho. Por mais que acreditemos que isso eliminará processos que podem ser avaliados como burocráticos, devemos compreender que haverá grandes fragilidades a partir da adoção de um processo tão informal, como dificuldades relacionadas à comunicação com outras pessoas envolvidas, a menos que todos os envolvidos usem a mesma linguagem. Também devemos compreender que o conhecimento relativo à forma como foi analisado e projetado ficará exclusivamente sob responsabilidade de uma pessoa ou equipe, o que pode gerar grandes prejuízos caso estes deixem de estar envolvidos no projeto, deixando assim verdadeiramente um abacaxi para quem quer que abrace o projeto. Uma outra grande limitação em evidência é o fato de que uma visão macro sobre os elementos computacionais envolvidos não existirá, e isso em processos de evolução ou identificação de bugs aumentará drasticamente o tempo para tais ações (BOOCH; RUMBAUGH; JACOBSON, 2012).

O uso da UML para elaboração de modelos abrange uma questão central: modelos explícitos facilitam a comunicação.

2.2 UML na especificação

Analizando o contexto, compreendemos que especificar significa construir modelos precisos, sem ambiguidades e completos. A UML apresenta total suporte para a tomada de decisões relacionadas à análise, projeto e implementação, isso no contexto do desenvolvimento e a implantação de sistemas complexos de software.

Enquanto linguagem de construção, devemos compreender que ela não possui mecanismos de programação visual, mas como baseia-se em conceitos do paradigma orientado a objetos, é natural que caso uma linguagem suporte tal paradigma, então poderá haver meios de conectar os modelos à programação. Dessa forma, podemos mapear os modelos da UML em linguagem de programação tais como Java, C#, C++, Visual Basic .NET ou até tabelas de bancos de dados relacionais ou armazenamento de dados persistentes em um banco de dados orientados a objetos, certo que esse tipo de banco não tem grande relevância para o mercado de tecnologia. Através da UML podemos representar tudo de forma efetiva por meio de visões gráficas, enquanto as linguagens de programação representam grandes limitações, contextos e relações complexas (BOOCH; RUMBAUGH; JACOBSON, 2012).

A partir do fato que dispomos conceitos equivalentes entre linguagem de modelagem e linguagem de programação, ferramentas foram desenvolvidas a partir do conceito de engenharia direta, em que o código é gerado a partir do modelo. Para o caminho inverso, em que diagramas são construídos a partir da interpretação do código, utilizamos a engenharia reversa. Ambas engenharias não fazem milagres, desmistificando a ideia de que a partir de uma ferramenta um sistema completo será produzido. Podemos usufruir do fato que se adotarmos de padronizações, então, podemos desenvolver inteligências que produzam uma estrutura comum (baseada em algum padrão), utilizando para isso a engenharia direta (BOOCH; RUMBAUGH; JACOBSON, 2012).

2.3 Documentação

Empresas de software devem produzir todos os tipos de artefatos disponíveis, pois estes ajudarão na compreensão do funcionamento de seus produtos. Assim, entendemos que a saúde de uma empresa envolve a criação de uma documentação consistente e que cubra todos os processos, além do código executável. Esses artefatos incluem (mas não estão limitados a) o seguinte (BOOCH; RUMBAUGH; JACOBSON, 2012):

- Requisitos.
- Arquitetura.
- Projeto.
- Código-fonte.
- Planos do projeto.
- Testes.
- Protótipos.
- Versões.

A formalidade sobre cada um desses artefatos e até o valor para cada um dependerá diretamente da maturidade da equipe de desenvolvimento, bem como da cultura da empresa. Utilizamos esses artefatos não só para projetar aquilo que desejamos entregar, mas são críticos para controlar, medir e comunicar determinado sistema durante seu desenvolvimento e após sua implantação. Documentos de arquitetura do sistema e seus detalhes podem ser documentados através da UML. Ainda podemos utilizar a UML para proporcionar uma linguagem para a expressão de requisitos e para a realização de testes. Uma outra vertente da UML é podermos utilizá-la para modelar as atividades de planejamento do projeto e gerenciamento de versões.

2.4 Onde utilizar

Podemos utilizar UML para representar desde sistemas de software modestos a sistemas complexos, em que observa-se maior relevância, mas podemos encontrar o uso da UML para atender a domínios

diferentes, tais como (BOOCH; RUMBAUGH; JACOBSON, 2012):

- Sistemas de informação corporativos.
- Serviços bancários e financeiros.
- Telecomunicações.
- Transportes.
- Defesa/espaço aéreo.
- Vendas de varejo.
- Eletrônica médica.
- Científico.
- Serviços distribuídos baseados na web.

Podemos utilizar a UML para modelar algo que não seja software, pois como essa linguagem apresenta-se de forma expressiva, podemos utilizá-la para outros fins, como: projetos de hardware, processos que envolvam pessoas, gestão e fluxo de trabalho.

2.5 Modelo conceitual da UML

O entendimento a respeito da UML poderá ser efetivo a partir da compreensão que existem três elementos principais, sendo eles: os blocos de construção básicos da UML, as regras que determinam como esses blocos poderão ser combinados e alguns mecanismos comuns aplicados na UML.

Caro aluno, após compreender essa dinâmica de forma efetiva, você será capaz de ler modelos e diagramas da UML, e além disso, é entendido que a habilidade de leitura servirá como um trampolim para a construção de modelos básicos. Podemos ainda assumir que a construção de modelos básicos lhe permitirá acumular experiência na aplicação da UML, e então poderá criar novos modelos, usando características e notações mais complexas e avançadas da linguagem.

A UML define três termos específicos, e estes são categorizados como blocos de construção:

1. Itens.
2. Relacionamentos.
3. Diagramas.

Através dos itens, definimos as nossas abstrações e por sua relevância, tanto para a construção de nossos modelos quanto para a construção do próprio software, devemos dar uma maior notoriedade. Utilizamos os relacionamentos para reunir os itens, e então, através dos diagramas agrupá-los (BOOCH; RUMBAUGH; JACOBSON, 2012).

2.5.1 Itens da UML

Vamos começar categorizando os itens por sua relevância. Existem quatro tipos de itens da UML:

1. Itens estruturais.
2. Itens comportamentais.
3. Itens de agrupamentos.
4. Itens anotacionais.

Para a construção de modelos consistentes, utilizaremos os itens como blocos centrais e básicos para a construção de modelos orientados a objetos da UML.

Devemos compreender que os itens estruturais são os substantivos utilizados para a construção de modelos da UML, representando elementos conceituais ou físicos, e estáticos do modelo. Coletivamente, os itens estruturais são chamados classificadores, pois servem para nomear os itens representativos, e quando analisados a partir de um coletivo compreenderemos que classificam os indivíduos (BOOCH; RUMBAUGH; JACOBSON, 2012).

Vamos destacar os seguintes itens estruturais:

- Classes.
- Interface.
- Colaborações.
- Casos de uso.
- Classes ativas.
- Componentes.
- Artefatos.
- Nós.

Entenda que para a construção de um modelo da UML, você poderá contar com estes itens estruturais, que qualificamos como básicos, por sua relevância e utilização.

Já os itens comportamentais são tratados como partes dinâmicas presentes nos modelos de UML, exatamente por refletir uma dinâmica, então, utilizamos os verbos para representá-los no modelo. Através deles, conseguimos representar comportamentos no tempo e no espaço. Existem ao todo dois tipos principais de itens comportamentais (BOOCH; RUMBAUGH; JACOBSON, 2012).

O primeiro comportamento que iremos discutir é a interação, que abrange um conjunto de mensagens trocadas entre um conjunto de objetos em um determinado contexto, isso para a realização de propósitos específicos.

Já a máquina de estado representa o segundo tipo de comportamento. Através dessa máquina conseguimos expressar as sequências de estado pelas quais objetos ou interações passam durante sua existência em resposta a eventos, descrevendo ainda as geradas a partir desses eventos.

Uma terceira proposta de comportamento é representada pela atividade, pois através dela conseguimos especificar a sequência das etapas que um processo computacional realiza. Como objetivo conseguimos identificar o conjunto de objetos que interagem a partir de uma interação.

Para permitir que existam partes organizacionais nos modelos de UML, introduzimos o conceito dos itens de agrupamento. A partir desses blocos conseguimos representar modelos mais complexos, permitindo assim que haja uma decomposição, naturalmente para servir como mecanismo organizacional. Os pacotes representam o único tipo existente como item organizacional, também chamado de item de agrupamento. Através de um pacote conseguimos adicionar meios para efetuar organização dentro do próprio projeto. Diferentemente de outros itens presentes na UML, que proporcionam formas para organizar a implementação da solução, os pacotes permitem que haja uma organização na ótica do projeto da UML (BOOCH; RUMBAUGH; JACOBSON, 2012).

Devemos compreender que os itens anotacionais são partes explicativas dos modelos de UML, e representados através dos comentários. Utilizamos os comentários para descrever, esclarecer

ou observar qualquer detalhe a respeito do modelo, ficando livre para uso conforme conveniência da equipe envolvida. Para o tipo de item anotacional, encontramos apenas um, chamado nota, e deveremos utilizá-lo para desenvolver os comentários pertinentes. Uma nota servirá para restrições e comentários anexados a um elemento ou a uma coleção de elementos (PRESSMAN, 2011).

2.5.2 Relacionamentos

Para a UML, encontramos quatro tipos de relacionamentos, sendo eles:

- Dependência.
- Associação.
- Generalização.
- Realização.

Utilizamos os relacionamentos como blocos relacionais básicos, fundamentais para a construção de modelos da UML. Com destaque apresentaremos o primeiro relacionamento, chamado dependência, que explicitamente apresenta um relacionamento semântico entre dois itens. Esse relacionamento semântico deve ser compreendido da seguinte maneira nele haverá um item que tratamos como independente e que mediante a sua alteração afetará o outro item, que qualificamos como dependente (BOOCH; RUMBAUGH; JACOBSON, 2012).

Um segundo relacionamento que conseguimos expressar através da UML é a associação. Como relacionamento estrutural, a associação descreve um conjunto de ligações entre classes, em que as ligações são conexões entre objetos que são instâncias das classes. Como um tipo especial de associação, podemos destacar a agregação, representando um relacionamento estrutural entre o todo e suas partes.

Um terceiro tipo de relacionamento se chama especialização/generalização, no qual os objetos elementos especializados, também chamados de filhos, são substituíveis por elementos generalizados, chamados de pais. Dessa maneira, os filhos compartilham a estrutura e o comportamento dos pais (BOOCH; RUMBAUGH; JACOBSON, 2012).

Quarto, mas não menos importante, temos a realização que é um relacionamento semântico entre classificadores, em que um classificador especifica um contrato que outro classificador garante executar, isso já foi comentado anteriormente. Encontraremos o relacionamento de realização em dois locais: entre interfaces e as classes ou componentes que as realizam; e entre casos de uso e as colaborações que os realizam.

2.5.3 Diagramas

Utilizamos um diagrama como uma apresentação gráfica de um conjunto de elementos, geralmente eles são representados através de gráficos de vértices (itens) e arcos (relacionamentos). Como são desenhados, desempenham uma grande função, permitir que visualizemos um sistema sob diferentes perspectivas. Nesse sentido, um diagrama constitui uma projeção de um determinado sistema, possibilitando o devido entendimento. O mesmo elemento pode aparecer em todos os diagramas, em apenas alguns (o caso mais comum), ou em nenhum diagrama (um caso muito raro). Em teoria, podemos encontrar qualquer combinação de itens e de relacionamentos em um diagrama, mas na prática aparecerá um pequeno número de combinações comuns, que são consistentes com as visões mais úteis da arquitetura de um sistema complexo de software. Encontramos na UML 13 diagramas que poderão nos auxiliar na difícil arte de construir soluções computacionais (BOOCH; RUMBAUGH; JACOBSON, 2012).

1. Diagrama de classes.
2. Diagrama de objetos.
3. Diagrama de componentes.
4. Diagrama de estruturas compostas.
5. Diagrama de casos de uso.
6. Diagrama de sequências.
7. Diagrama de comunicações.
8. Diagrama de estados.
9. Diagrama de atividades.
10. Diagrama de implantação.
11. Diagrama de pacotes.

12. Diagrama de temporização.
13. Diagrama de visão geral de interação.

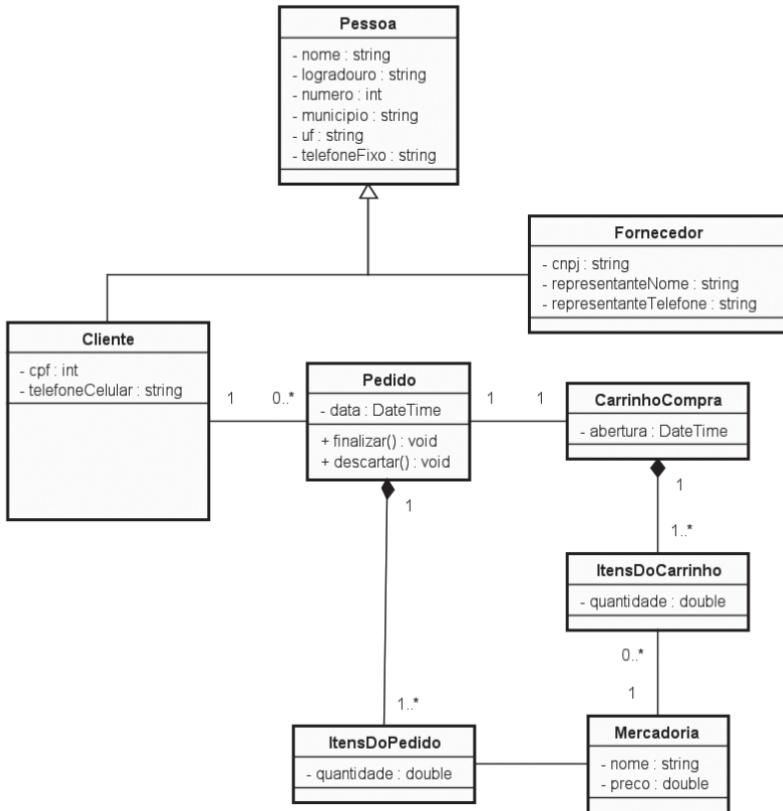
2.6 Diagramas de classes

Encontramos os diagramas de classes com maior frequência na modelagem de sistema orientado a objetos, isso porque como eles permitem uma visão geral entre os substantivos que representam o sistema, bem como seus relacionamentos, comumente serão utilizados para construção da base de dados. Como iniciamos falando sobre isso, o diagrama de classes permitirá que se defina o vocabulário do sistema.

Um diagrama de classes mostra um conjunto de classes, interfaces e colaborações e seus relacionamentos. Utilizamos os diagramas de classes para fazer a modelagem da visão estática do projeto de um sistema. Na maioria dos casos, isso envolve a modelagem do vocabulário do sistema, a modelagem de colaboradores ou a modelagem de esquemas (BOOCH; RUMBAUGH; JACOBSON, 2012).

Utilizamos também os diagramas de classes para construir outros diagramas relacionados, como: diagramas de componentes e diagramas de implantação. Os diagramas de classe são importantes não só para a visualização, a especificação e a documentação de modelos estruturais, mas também os utilizamos para a construção de sistemas executáveis por intermédio de engenharia direta e reversão. Como já observado, a geração completa de uma aplicação estará suscetível a vários aspectos, como padronização, complexidade de processos, padronização de modelos e interações e outros.

Figura 3.7 | Diagrama de classe



Fonte: elaborada pelo autor.

Através desse modelo, utilizamos vários conceitos que já foram abordados ou que serão devidamente observados. Um exemplo de generalização muito comum em um sistema de informação envolve os tipos de pessoas que encontramos em nossa aplicação, que neste modelo foram representados pela classe Pessoa (classe pai), Cliente e Fornecedor (classes filhas). Utilizamos as associações para representar relações estruturais, adotando a multiplicidade para indicar a quantidade de objetos em cada uma das associações. Utilizamos também associação do tipo composição, definida através do diamante preenchido. Essa associação permite definir uma relação de composição, indicando que uma classe é composta por outra, respeitando também a multiplicidade definida.

Analizando o nosso exemplo, compreendemos que a classe

Pedido é composta por ItensDoPedido, já a classe CarrinhoCompra é composta, entre outras características, por ItensDoCarrinho. Todos esses tipos de relacionamentos são definidos com maior propriedade a seguir.

2.7 Relacionamentos

Quando buscamos entender e construir o vocabulário de nossa aplicação, perceberemos que poucas classes trabalharam sozinhas, e ao invés disso a maioria das classes vai de forma colaborativa permitir que as aplicações funcionem. Além de ser comum a colaboração entre as classes, devemos compreender que é fundamental, pois complexidades de negócio e de lógicas computacionais, qualidade de engenharia, coesão, viabilidades, tudo isso será determinado pela capacidade que temos em conseguir visualizar os relacionamentos entre as classes.

Dentro do universo da modelagem orientada a objetos, iremos encontrar três tipos de relacionamentos:

- Dependências: no qual conseguiremos representar relacionamentos que se baseiam no termo de utilização (incluindo relacionamentos de refinamento, rastreamento e vínculos).
- Generalizações: como importantes recursos para a programação que relacionam classes generalizadas a suas especializações.
- Associações: denotam relacionamentos estruturais entre objetos.

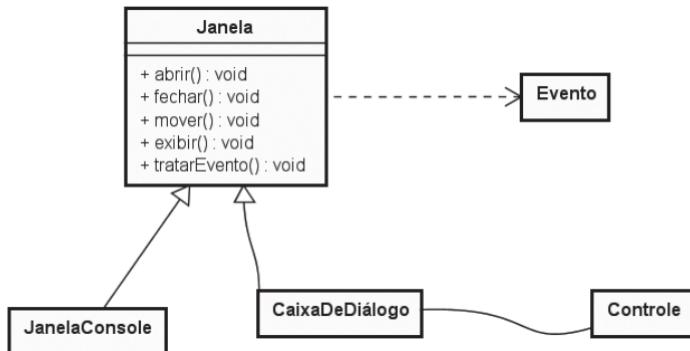
Através desses relacionamentos poderemos exercer abstrações distintas, podendo ainda efetuar combinações (BOOCH; RUMBAUGH; JACOBSON, 2012).

Através da UML, definimos um relacionamento como um meio para modelar os modos pelos quais os itens podem estar conectados a outros, tanto para relacionamentos lógicos como físicos. Nesse modelo iremos encontrar três importantes tipos de relacionamentos: dependências, generalizações e associações (BOOCH; RUMBAUGH; JACOBSON, 2012).

- Dependências: como já destacamos, representam relacionamentos que expressam a ideia de utilização, por exemplo, o motor do carro depende do sistema de arrefecimento para manter a sua temperatura em padrões aceitáveis.
- Associações: como um exemplo de relacionamento estrutural entre instâncias, podemos exemplificar, por exemplo, que uma venda é formada pelos itens de venda.
- Generalizações: conceito mais fácil de ser visualizado a partir da abstração. Através dele conseguimos conectar classes generalizadas a outras mais especializadas, conceito também conhecido como relacionamentos subclasse/superclasse ou filha/mãe. Por exemplo, uma instituição de ensino privada possui alunos, mas estes podem variar de acordo com o contrato e naturalmente dispor de mais ou menos informações quando abstraídos, podemos encontrar alunos mensalistas, alunos bolsistas, alunos com financiamento e outros.

A relevância desses relacionamentos é percebida pelo fato que todos eles são devidamente suportados e serão representados através das linguagens de programação.

Figura 3.8 | Relacionamento de classes



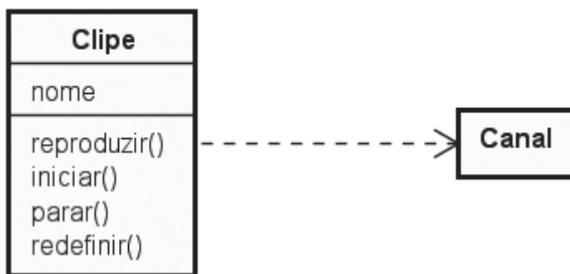
Fonte: adaptado de Booch, Rumbaugh e Jacobson (2012, p. 65).

2.7.1 Dependência

Para representarmos um relacionamento que expõe a ideia de utilização, recorreremos ao relacionamento do tipo dependência. Através dele conseguiremos representar que um item (por exemplo, a classe Computador) utiliza as informações e os recursos disponíveis em outro item (por exemplo, a classe MemoriaPrincipal), mas não obrigatoriamente o inverso. Representamos a dependência graficamente como linhas tracejadas, apontando o item do qual o outro depende. Devemos utilizar a dependência sempre que desejarmos representar a relação de utilização (BOOCH; RUMBAUGH; JACOBSON, 2012).

Encontramos a utilização de relacionamentos de dependência para demonstrar situações em que uma classe utiliza outra como argumento na assinatura de uma operação. Isso é essencialmente um relacionamento de dependência, pois caso a classe utilizada for modificada, então a execução do comportamento da classe dependente poderá ser afetada.

Figura 3.9 | Representando dependência



Fonte: adaptado de Booch, Rumbaugh e Jacobson (2012, p. 66).

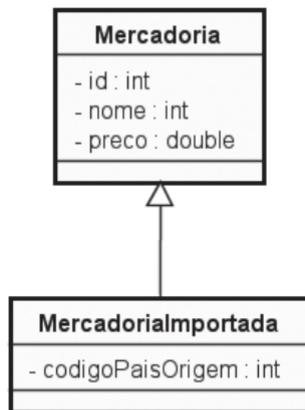
2.7.2 Generalização

Utilizamos o relacionamento de generalização para conectar classes gerais, que comumente chamamos de superclasses ou classes-mães, com classes específicas (chamadas subclasses ou classes-filhas), considerando que as classes específicas representam subgrupos de um grupo maior, representado pela classe mãe. Encontramos autores

que consideram a generalização como um relacionamento do tipo “é um tipo de”: um item (como a classe MercadoriaImportada) é um tipo de item mais geral (por exemplo, a classe Mercadoria). Para que a generalização seja efetiva, um objeto da classe-filha poderá ser utilizado em qualquer processo que envolva a classe-mãe, mas não vice-versa. Seguindo o nosso exemplo, podemos considerar que uma venda poderá ocorrer utilizando uma “Mercadoria” ou então uma “Mercadoria Importada”, já um processo que ocorre junto a um órgão regulamentador de produtos importados somente ocorrerá a partir de uma “Mercadoria Importada” (BOOCH; RUMBAUGH; JACOBSON, 2012).

Também é fato que uma classe filha herda os membros da classe mãe, considerando para isso inicialmente os atributos e operações, mas pode haver mais membros, isso exclusivamente no contexto da programação. Assumindo que uma classe-filha dispõe de características de um subgrupo, podemos compreender que elas terão novos atributos e operações além daqueles encontrados nas respectivas mães. Quando uma operação for declarada tanto para a classe-filha quanto para a classe-mãe, respeitando a mesma assinatura, então prevalecerá a operação da classe-filha em relação à operação da mãe; isso é conhecido como polimorfismo. Para representar a generalização, a UML define graficamente com linhas sólidas uma grande seta triangular apontando a mãe, conforme mostra a Figura 3.10 (BOOCH; RUMBAUGH; JACOBSON, 2012).

Figura 3.10 | Herança



Fonte: elaborada pelo autor.

Uma classe pode ou não ter classes-mãe, e quando uma classe possui classes-filhas, mas nenhuma classe-mãe, ela será chamada de classe-raiz ou classe de base. Já a classe que não possui filhas é chamada de classe-filha. Quando uma classe-filha dispõe de várias classes-mãe, dizemos que houve heranças múltiplas, já em situações em que há apenas uma classe mãe, então, chamamos de herança única.

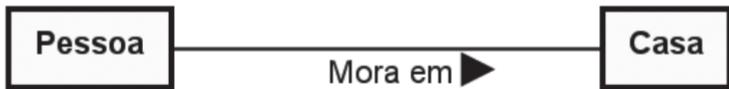
2.7.3 Associação

Utilizamos a associação para indicar um relacionamento estrutural, conectando objetos de uma classe com objetos de outra. Partindo de uma associação, podemos navegar do objeto de uma classe até o objeto de outra classe e vice-versa. É possível que uma classe expresse um autorrelacionamento, isso significa que, a partir de um objeto da classe, você poderá criar vínculos com outros objetos da mesma classe. Para associação entre duas classes, chamamos de associação binária, mas pode haver associações conectando mais de duas classes, chamadas associações enésimas. Uma associação é representada graficamente como uma linha sólida, conectando a mesma classe ou classes diferentes. Sempre que desejamos representar relacionamento estruturais, podemos utilizar a associação (BOOCH; RUMBAUGH; JACOBSON, 2012).

2.7.3.1 Nome

Podemos nomear uma associação, na qual descreveremos a natureza do relacionamento, portanto, quando não há ambiguidade acerca de seu significado, podemos atribuir uma direção para o nome, para isso utilizamos um triângulo de orientação que aponta a direção com o nome que deverá ser lido, conforme a Figura 3.11 (BOOCH; RUMBAUGH; JACOBSON, 2012):

Figura 3.11 | Nome do relacionamento



Fonte: elaborada pelo autor.

2.7.3.2 Papel

Em uma participação em uma associação, uma classe possui um papel específico a executar nesse relacionamento; o papel representa apenas uma das faces que uma classe pode ter, mas impreterivelmente, para com a outra classe do relacionamento. Podemos nomear o papel desempenhado por uma classe na associação. Utilizamos o termo nome da extremidade para a representação desse papel no diagrama. Na Figura 3.12 poderemos observar os papéis que as classes Pessoa e Casa representam nesse relacionamento, em que a pessoa desempenha o papel de morador, já a casa de residência (BOOCH; RUMBAUGH; JACOBSON, 2012).

Figura 3.12 | Papel no relacionamento



Fonte: elaborada pelo autor.

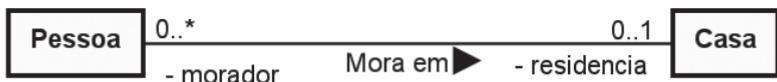
2.7.3.3 Multiplicidade

Na modelagem, uma importante informação que conseguimos representar é que, em relacionamentos do tipo associação, conseguimos determinar a quantidade de objetos que estarão conectados pela instância de uma associação. Chamamos de multiplicidade essa “quantidade” do papel de uma classe na associação, e utilizamos um intervalo de inteiros que especifica o tamanho possível do conjunto de objetos relacionados. Para isso, criamos expressões no diagrama que representarão o valor mínimo e máximo, que podem ser iguais; para separá-los utilizamos dois pontos (BOOCH; RUMBAUGH; JACOBSON, 2012).

Determinamos a partir da seguinte lógica que a multiplicidade é colocada no extremo da associação e configura o conjunto de instâncias necessárias da classe próxima a ela, considerando para isso uma instância da classe na extremidade oposta. O número de objetos deve estar no intervalo dado. Podemos representar uma multiplicidade de exatamente um (1), zero ou um (0..1), muitos (0..*) ou um ou mais (1..*). Também podemos fornecer um intervalo inteiro (como 2..5), ou então o número exato (por exemplo, 3, que equivale a 3..3).

No modelo definido através da Figura 3.13, podemos compreendê-lo da seguinte forma, uma pessoa pode ou não morar em uma residência (multiplicidade 0..1), por outro lado uma casa pode ter nenhum ou muitos moradores (multiplicidade 0..*).

Figura 3.13 | Multiplicidade



Fonte: elaborada pelo autor.

2.7.3.4 Agregação

Quando desejamos representar o relacionamento estrutural, devemos recorrer à associação, significando que as classes estão conceitualmente em um mesmo nível de importância no relacionamento. Encontramos em alguns casos a relação representada pelo conceito do “todo/parte”, em que uma classe representa o item maior (o “todo”), formado por itens menores (as “partes”). Para esse tipo de relacionamento utilizamos a agregação, que representa o relacionamento do tipo “tem-um”, o que significa que um objeto do todo contém os objetos das partes. Compreendemos que a agregação é um tipo especial de associação, dessa forma, representamos através de uma associação simples com um diamante aberto na extremidade do todo, conforme mostra a Figura 3.14 (BOOCH; RUMBAUGH; JACOBSON, 2012):

Figura 3.14 | Agregação



Fonte: elaborada pelo autor.

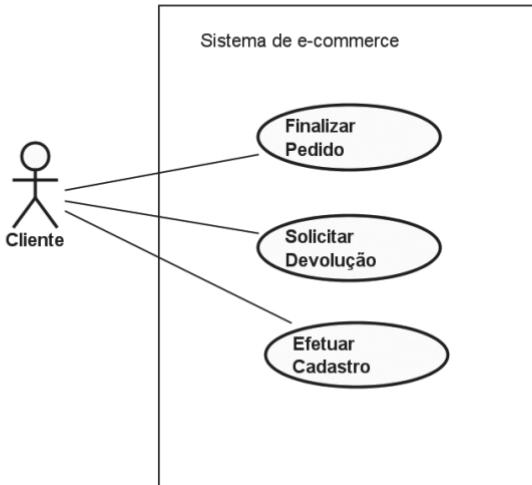
2.8 Diagramas de casos de uso

Já os diagramas de casos de usos são utilizados para a modelagem de aspectos dinâmicos de sistemas. Eles têm um papel central para a modelagem do comportamento de um sistema, de um subsistema ou de uma classe. Cada um mostra um conjunto de casos de uso e atores e seus relacionamentos.

Utilizamos os diagramas de casos de usos para fazer a modelagem da visão de caso de uso do sistema. Para isso, a maioria envolve a modelagem do contexto do sistema, subsistema ou classe ou a modelagem dos requisitos do comportamento desses elementos.

Os diagramas de casos de usos são importantes para visualizar, especializar e documentar o comportamento de um elemento. Através desses diagramas conseguimos obter uma visão externa sobre como esses elementos podem ser utilizados no contexto. Uma outra via de utilização está na aplicação deles para testar sistemas executáveis por meio de engenharia direta e para compreendê-los por meio de engenharia reversão (BOOCH; RUMBAUGH; JACOBSON, 2012).

Figura 3.15 | Diagrama de caso de usos



Fonte: elaborada pelo autor.

A partir de um ator, que corresponde a quem executa a ação, definimos as funcionalidades que existirão, ações dinâmicas que ocorrerão no contexto, neste caso em um sistema de e-commerce.

2.9 Diagrama de interação

Chamamos de diagramas de interação os diagramas de sequência e os diagramas de comunicação. Eles são utilizados na UML para modelagem dos aspectos dinâmicos de sistemas. Através do diagrama de interação podemos observar a interação, formada por um conjunto de objetos e seus relacionamentos, incluindo as mensagens que poderão ser enviadas entre eles. O diagrama de sequência é um diagrama de interação, com ênfase na ordenação temporal das mensagens, já o diagrama de comunicação é um diagrama de interação que dá ênfase à organização estrutural dos objetos que enviam e recebem mensagens.

Esses diagramas podem estar sozinhos para visualizar, especificar, construir e documentar a dinâmica de um determinado conjunto de objetos, que aqui vamos chamar de sociedade, ou podem ser utilizados para fazer a modelagem de um determinado fluxo de controle de caso de uso (BOOCH; RUMBAUGH; JACOBSON, 2012).

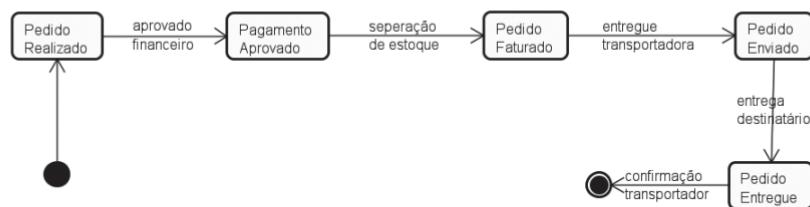
2.10 Diagrama de estado

Através dos diagramas de estado conseguimos modelar os aspectos dinâmicos de um sistema. Em boa parte, isso envolverá a modelagem do comportamento de objetos reativos. Para correta interpretação, um objeto reativo é aquele cujo comportamento é significativo a partir de sua resposta a eventos ativados externamente ao seu contexto. Um objeto reativo possui um tempo de vida cujo comportamento atual é afetado pelo seu passado, dando a ideia de progressão de estado (BOOCH; RUMBAUGH; JACOBSON, 2012).

Através de um diagrama de estado, conseguimos visualizar uma máquina de estados, com ênfase no fluxo de controle de um estado para outro. Já a máquina de estados representa as sequências de estados pelos quais um objeto passa durante seu tempo de vida, sempre em resposta a eventos, juntamente com suas respostas a esses eventos. O estado, compreendemos como uma condição ou situação na vida de um objeto, durante a qual ele satisfaz alguma condição, realiza alguma atividade ou guarda algum evento. Para um evento, este especifica uma ocorrência significativa que tem uma localização no tempo e no espaço.

De forma geral, no contexto de uma máquina de estados, um evento é uma ocorrência de um estímulo, capaz de ativar uma transição de estado. Entendemos uma transição como um relacionamento entre dois estados, indicando que um objeto no primeiro estado realizará certas ações, e então entrará no segundo quando um evento específico ocorrer e as condições esperadas forem satisfeitas. A execução não atômica é chamada de atividade, e ela estará em andamento em uma máquina estados. Já a ação é um procedimento computacional executável e atômico, que resulta em uma alteração do estado do modelo ou no retorno de um valor (BOOCH; RUMBAUGH; JACOBSON, 2012).

Figura 3.16 | Diagrama de estado



Fonte: elaborada pelo autor.

Através desse exemplo podemos observar a mudança de um pedido feito em uma loja em um e-commerce.



Para saber mais

Para uma visão mais abrangente sobre o diagrama de estado, e mais detalhes sobre a sua criação e utilização, você pode consultar o material intitulado *Diagrama de Estado* fornecido pela Prof. Daniel D. Abdala através do link: <<http://www.facom.ufu.br/~abdala/DAS5312/Diagrama%20de%20Estados.pdf>>. Acesso em: 29 set. 2017.



Questão para reflexão

O desenvolvimento de um projeto de software utilizando o paradigma orientado a objetos pode ser bem-sucedido sem a utilização da UML?

Atividades de aprendizagem

1. A UML também é conhecida como uma linguagem, e para iniciarmos devidamente o seu estudo, em que devemos procurar o domínio?

- A) Linguagem orientada a objetos.
- B) Ferramenta de modelagem UML.
- C) Vocabulário e regras da UML.
- D) Boas práticas na modelagem.
- E) Aplicação satisfatória em projetos.

2. Conforme observamos, a UML coloca-se como uma linguagem de construção e devemos compreender que ela não possui mecanismos de programação visual, mas como baseia-se em conceitos do paradigma orientado a objetos, é natural que caso uma linguagem suporte tal paradigma, então poderá haver meios de conectar os modelos à programação. Dessa forma, podemos mapear os modelos da UML em linguagem de programação tais como Java, C#, C++, Visual Basic .NET ou até tabelas de bancos de dados relacionais. A equivalência entre os conceitos da linguagem de modelagem e de programação trazem muitas vantagens. Assinale a alternativa INVÁLIDA quanto às vantagens:

- A) Facilidade na localização de um código a partir de um modelo, ou vice-versa.
- B) Engenharia direta.
- C) Engenharia reversa.
- D) Criação de modelos de banco de dados relacionais.
- E) Criação de sistemas completos a partir dos modelos.

Fique ligado

Nesta unidade, estudamos:

- Conceitos de orientação a objetos.
- Classes, atributos, operações e responsabilidade.
- Classificadores, visibilidade, escopo e multiplicidade.
- Interfaces.
- Introdução à UML.
- Diagramas de classes e relacionamentos.
- Visão geral de casos de uso, interação e estado.

Para concluir o estudo da unidade

Caro aluno, estamos finalizando esta unidade! Através dela você pôde observar importantes conceitos relacionados à programação orientada a objetos, mantidos exclusivamente no âmbito conceitual de forma proposital. Através destes conceitos abordamos importantes recursos existentes na linguagem de modelagem UML, principalmente através do diagrama de classe e seus relacionamentos. Lembrando que a UML é muito mais abrangente e poderosa para a modelagem de um sistema orientado a objetos, mas destacamos aqui aquilo que neste ponto de aprendizagem consideramos que servirá como base para a correta ligação com os demais conceitos abordados neste livro.

Atividades de aprendizagem da unidade

- 1.** Quando desejamos definir uma operação que não é exclusiva de um indivíduo, mas do coletivo, podemos registrá-la como uma operação estática. Qual é a correta notação para uma operação estática no diagrama de classe?
A) Sublinhado.
B) Negrito.
C) Itálico.
D) Tachado.
E) Nome em caixa alta.

2. Para entendermos e construirmos o vocabulário de nossa aplicação, perceberemos que poucas classes trabalharão sozinhas, e ao invés disso a maioria das classes vai de forma colaborativa permitir que as aplicações funcionem. Essa colaboração é fundamental, pois permite compreender complexidades de negócio e de lógicas computacionais, qualidade de engenharia, coesão, viabilidades. Assinale a alternativa correspondente a esse importante recurso definido na UML.

- A) Especificação.
- B) Diagrama.
- C) Multiplicidade.
- D) Relacionamento.
- E) Abstração.

3. Para representarmos o relacionamento estrutural devemos recorrer à associação, com o objetivo de indicar mesmo grau de importância entre elas no relacionamento. Encontramos em alguns casos a relação representada pelo conceito do "todo/parte", em que uma classe representa o item maior (o "todo"), formado por itens menores (as "partes").

Os relacionamentos que expressam o conceito do "todo/parte" também são conhecidos por?

- A) tem-um.
- B) possui-um.
- C) é-um.
- D) tem-muitos.
- E) são-muitos.

4. Objetos são instâncias de classes, que determinam qual informação um objeto contém e como ele pode manipulá-la. Um programa desenvolvido com uma linguagem de programação orientada a objetos manipula estruturas de dados através dos objetos da mesma forma que um programa em linguagem tradicional utiliza variáveis (DAVID, [s.d.]).

Entre os conceitos definidos no relacionamento de generalização, assinale a alternativa que contém a sequência de expressões que completam corretamente a seguinte sentença:

Utilizamos o relacionamento de _____ para conectar classes gerais, as quais comumente chamamos de _____ ou _____, com classes específicas (chamados _____), considerando que as classes específicas representam subgrupos de um grupo maior, representado pela classe-mãe.

- A) Generalização, subclasses, classes-mães, superclasses ou classes-filhas.
- B) Generalização, superclasses, classes-mães, subclasses ou classes-filhas.

- C) Generalização, superclasses, classes-mães, subclasses ou classes contrárias.
- D) Generalização, superclasses, classes-mães, subclasses ou classes generalistas.
- E) Generalização, superclasses, classes-mães, subclasses ou classes estendidas.

5. Nos diagramas de classe de negócios, os tipos de atributo geralmente correspondem a unidades que fazem sentido aos prováveis leitores do diagrama (ou seja, minutos, dólares etc.). No entanto, um diagrama de classes que será usado para gerar código precisa de classes cujos tipos de atributo sejam limitados aos tipos fornecidos pela linguagem de programação ou aos tipos incluídos no modelo que também serão implementados no sistema.

Fonte: Disponível em: <<https://www.ibm.com/developerworks/br/rational/library/content/RationalEdge/sep04/bell/index.html>>. Acesso em: 29 ago. 2017 .

Essa distinção entre o tipo de um atributo voltado para o leitor ou então para a geração do código só faz sentido quando utilizamos uma técnica. Qual alternativa corresponde essa técnica?

- A) Modularização.
- B) Reflexão.
- C) Polimorfismo.
- D) Recursividade.
- E) Engenharia direta.

Referências

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **Uml guia do usuário**. Rio de Janeiro: Campus, 2012.

DEITEL, H.; DEITEL, P.J.; CHOFFNES, D. R. **Sistemas operacionais**. 3. ed. São Paulo: Pearson Prentice Hall, 2005.

DI. **Dicionário Informal**. 2008. Disponível em: <<http://www.dicionarioinformal.com.br/c%C3%BAbito/>>. Acesso em: 17 ago. 2017.

KOSCIANSKI, A.; SOARES, M. **Qualidade de software**: aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software. 2. ed. São Paulo: Novatec, 2007.

SOMMERVILLE, I. **Projeto de sistemas**. São Paulo: Pearson, 2010.

Tecnologias para projetos orientados a objetos

Paulo Henrique Terra

Objetivos de aprendizagem

Nesta unidade, iremos estudar algumas tecnologias aplicadas em projetos de software orientados a objetos, vamos abordar os tipos de linguagens de programação, tipos de sistemas gerenciadores de banco de dados, arquiteturas, *frameworks* e outros artefatos técnicos utilizados para concepção do produto final de software.

Seção 1 | Linguagens de programação

Qual linguagem de programação empregar em um projeto orientado a objetos? Vamos falar sobre os tipos de linguagens de programação que devem ser considerados na fase de construção de uma aplicação de software e sobre a decisão de qual das linguagens disponíveis no mercado utilizar.

Seção 2 | – Modelagem de dados

Vamos abordar nesta seção um assunto que gera dúvidas na aplicação de sistemas de banco de dados relacionais em projetos orientados a objetos, a compatibilidade de paradigmas, que pode e deve ser considerada com intuito de manter princípios básicos de qualidade do projeto e do produto final do software em conformidade.

Seção 3 | Arquitetura de sistemas

Arquiteturas de sistemas não são propriamente tecnologias empregadas diretamente em um software aplicativo, mas a definição sobre qual padrão ou estilo de arquitetura definitivamente irá impactar na escolha das tecnologias que garantirão os requisitos funcionais e não funcionais especificados na engenharia de requisitos.

Introdução à unidade

Você foi persistente e chegou à última unidade do nosso livro didático! É muito bom encontrá-lo nesta etapa dos estudos!

Os conceitos apresentados nas unidades anteriores nos possibilitaram adquirir conhecimentos que serão aplicados na etapa de análise de um projeto orientado a objetos. Nesta etapa, será definido “o que” será desenvolvido. Assim, vamos dar mais um passo e definir “como” um software será desenvolvido. Para isso, entenderemos a aplicação dos requisitos não funcionais, adicionando as características técnicas ao projeto.

Então, pergutamos: que tipos de tecnologias são empregadas na etapa de projeto técnico a fim de alcançar o resultado final? Em outras palavras, como um software é construído do ponto de vista da aplicação das diversas tecnologias disponíveis no mercado? Afinal, atualmente temos uma grande variedade de tecnologias de linguagens de programação, sistemas de banco de dados, frameworks e outros componentes tecnológicos que podem ser empregados na construção de uma aplicação. A cada dia novas tecnologias estão disponíveis não somente como resposta à solução de problemas de desenvolvimento de software, mas também para tornar mais eficiente a construção de sistemas computacionais.

O grande desafio atual dos profissionais de tecnologia da informação não é contornar as tecnologias anteriormente limitadoras, mas lidar com uma infinidade de opções que pode ser utilizada na construção de um produto de software (KENN, 1996). Semelhante a outras áreas que devem conhecer os melhores recursos para construção de seus produtos, a engenharia da computação deve ser aplicada de forma assertiva para cada tipo de projeto de software. Não há uma única tecnologia que será aplicada a diversos projetos, da mesma maneira que em um único projeto podemos utilizar um misto de tecnologias, caberá aos profissionais envolvidos nestes projetos a tarefa de definir quais serão as técnicas mais compatíveis, duradouras, flexíveis e adaptadas àquele projeto.

Nas seções desta unidade, vamos explorar conceitos relacionados aos requisitos técnicos de um projeto de software, tais como: padrões de projetos, estilos de arquiteturas de software, protocolos de

comunicação, linguagens de programação e *frameworks*. Estes vários conceitos aplicados em conjunto darão a chance de obtermos um produto final de software que atenda à necessidade do negócio para o qual o sistema será desenvolvido.

Vamos eliminar dificuldades pessoais e empresariais através da construção de aplicações inteligentes pelo emprego das tecnologias.

Boa leitura!

Seção 1

Linguagens de programação

Introdução à seção

Uma das principais dúvidas quando pensamos em desenvolver um software é: que tipo de linguagem de programação utilizar?

É comum que profissionais de tecnologia da informação envolvidos em projetos de software tenham uma tendência em defendera aplicação da sua linguagem de programação preferida, porém, o importante não é se uma determinada linguagem, preferida por alguns, é melhor que outra, mas sim, se a linguagem a ser utilizada é a mais adequada a um determinado tipo de projeto. Outras questões a serem observadas são se esta linguagem irá reduzir os esforços de desenvolvimento, e se está adaptada aos paradigmas em uso no projeto. Além disso, deverão ser considerados aspectos relacionados até mesmo à mão de obra a ser utilizada, se esta estará capacitada a utilizar tal linguagem ou se há possibilidade de capacitação do time de desenvolvimento para sua utilização. Vamos discutir em um aspecto geral, como determinar qual tecnologia em linguagens de programação deve ser utilizada em um projeto orientado a objetos.

1.1 Linguagens orientadas a objeto

Não poderia ser de outro jeito, em se tratando de um projeto orientado a objetos, tudo que é utilizado neste tipo de projeto deve seguir o mesmo paradigma, inclusive a linguagem de programação.

A programação orientada a objetos é um paradigma que utiliza objetos para construir os programas de computador e deve seguir os mesmos princípios empregados na análise orientada a objetos, principalmente quando esta proporciona uma forma simples e coerente de compreender a realidade, o que torna, por consequência, o design orientado a objetos consistente, facilitando a programação e resultando em programas claros e fáceis de manter.

Conforme definido por Booch (1991, p. 25), a orientação a objetos é igual a dados abstratos somados a tipos de objetos mais tipos de

herança: “Orientação a objetos = dados abstratos + tipos de objetos + tipos de herança”. Mais tarde, Booch (1994) separou tipos de objetos em encapsulamento e modularização e nomeou o tipo de herança como hierarquia. Assim, uma linguagem de programação orientada a objetos deve seguir a definição anterior de Booch, ou seja, ela deve implementar a abstração de dados, possuir tipos de dados derivados de classes e lidar com o conceito de herança.

Adicionalmente ao que foi definido como orientação a objetos por Booch (1994), incluímos as seguintes características às linguagens orientadas a objetos: todos os tipos predefinidos são objetos, todas as operações são realizadas através do envio de mensagens a objetos e todos os tipos definidos pelo usuário são objetos.

Linguagens como C#, Ruby e Smalltalk são orientadas a objetos, pois satisfazem a todas as características anteriores. Outras linguagens, como C++ e PHP são multi-paradigma e possuem mecanismos nativos para implementar orientação a objetos, mas não se preocupam em satisfazer todas as características. Existem linguagens que são ditas orientadas a objeto, mas que não satisfazem a todas as características necessárias. O Java, por exemplo, faz uso de tipos nativos além dos tipos definidos por objetos.

Por mais que as linguagens implementem nativamente ou não todas as características comentadas anteriormente, vale ressaltar que é possível utilizar o paradigma da orientação a objetos na programação, utilizando-se uma linguagem que implemente ao menos a abstração de dados, defina tipos de objetos e carregue o conceito da herança. O importante é ressaltar que a linguagem de programação escolhida seja capaz de atender aos requisitos estabelecidos na fase de análise, seguindo os princípios da orientação a objetos. Se a linguagem tiver esta capacidade, então, outros fatores, como domínio da sintaxe da linguagem pela equipe de desenvolvimento, necessidades mercadológicas, adaptação às plataformas, uso de licenças e evolução da linguagem serão determinantes na escolha da linguagem de programação.

Resumidamente, uma série de linguagens que implementam as características essenciais de OO podem ser utilizadas em projetos orientados a objetos, portanto, a discussão sobre qual delas utilizar permanecerá um desafio a ser resolvido para cada cenário enfrentado pelos profissionais envolvidos nestes projetos.

1.2 Padrões de projeto

Os padrões de projeto podem nos ajudar de duas maneiras ao desenvolver uma aplicação de software, conforme definido por Barnes et al. (2004, p. 235):

Um padrão de projeto descreve um problema comum que ocorre regularmente no desenvolvimento de software e descreve então uma solução geral para esse problema que pode ser utilizada em muitos contextos diferentes. Em geral, para padrões de projeto de software, a solução é uma descrição de um pequeno conjunto de classes e suas interações.

Assim, a primeira maneira é documentando boas soluções para problemas, de modo que passem a ser reutilizadas em problemas semelhantes. Assim, a reutilização não está no nível de código-fonte criado em uma determinada linguagem, mas sim no nível das estruturas das classes.

A segunda maneira é nomeando os padrões e isto ajuda os designers de software a realizarem as conversas sobre seus designs, permitindo economizar detalhes de uma explicação. Assim, a linguagem-padrão introduzida por padrões de projeto apresenta outro nível de abstração, um que permite lidar com a complexidade até em sistemas mais complexos.

O próprio paradigma da orientação a objetos torna-se um padrão de projetos largamente utilizado. Tanto é que estamos falando sobre projetos de software baseados neste padrão, utilizamos várias das soluções deste paradigma nos projetos de software e ao falarmos sobre objetos, relacionamentos, atributos e métodos economizamos maiores explicações por já termos em mente o seu significado. Vamos analisar a seguir um exemplo de padrão de projetos que pode ser aplicado aos projetos orientados a objetos.



Padrões de projeto tornaram-se conhecidos por meio de um livro publicado em 1995, que descreve um conjunto de padrões, suas aplicações e benefícios. Não pretendemos neste livro esgotar os vários padrões de projeto existentes, mas você pode aprofundar seus conhecimentos no assunto por meio de outra literatura.

GAMMA, Eich et al. **Design patterns**: elements of reusable object – oriented software de Addison-Wesley, 1995.

1.3 Estrutura de um padrão

Os padrões são descritos com informações mínimas relativas à estrutura de classes e devem incluir uma descrição do problema resolvido. Além das forças contra ou a favor da utilização deste, o template básico deve possuir no mínimo:

- Nome: deve descrever o padrão de maneira conveniente.
- Descrição: descreve o problema que o padrão pretende solucionar (normalmente dividida em seções como intensão, motivação, aplicabilidade).
- Consequências: do uso do padrão, prevê resultados e compensações.

1.4 Singleton

Este é apenas um dos vários exemplos de padrões de projeto disponíveis e aplicáveis em um código orientado a objetos. Muitas vezes é necessário que em um programa um objeto derivado de uma classe deva possuir uma única instância. O padrão Singleton assegura a criação única desta instância e fornece acesso unificado a ela. Na linguagem Java, um Singleton pode ser definido tornando o construtor privado, assegurando que ele não possa ser acessado de fora da classe.

Figura 4.1 | O padrão Singleton

```
class Parser
{
    public private static Parser instance = New Parser();
    public static Parser getInstance()
    {
        return instance;
    }
    private Parser()
    {
        ...
    }
}
```

Fonte: Barnes et al (2004, p. 327).

A classe Parser, ilustrada na Figura 4.1, define um construtor privado, assim as instâncias desta classe podem ser criadas somente por ela própria. Para que isso seja possível, deve-se utilizar uma parte estática da classe (inicializações de campos estáticos ou métodos estáticos). Um campo estático privado é definido e inicializado com a única instância do analisador de sintaxe. Um método *getInstance* estático é definido para fornecer acesso único, para ganhar acesso aos objetos da classe este método deve ser utilizado: *Parser parser = Parser.getInstance()*.

Perceba que a ideia por traz deste padrão de projeto é um conceito aplicado através de uma linguagem de programação e não existem bibliotecas ou classes prontas para implementar tal ideia. Então, vamos realizar a comparação de conceitos no tópico relacionado a frameworks.

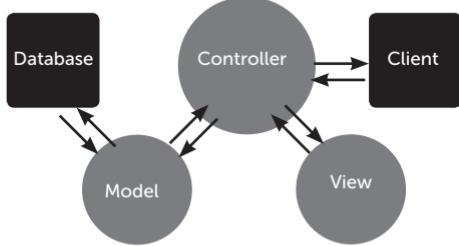
1.5 Outros tipos de padrão de projetos

Outros padrões de projeto com diferentes objetivos podem ser utilizados em um projeto OO, um exemplo de padrão largamente utilizado juntamente com as linguagens Java e .NET é o MVC (Model-View-Cotroller). Este define um padrão arquitetural baseado em camadas e divide um aplicativo de modo que a lógica de negócios resida entre três camadas físicas (MACORATTI, 2017).

No MVC, o modelo (M) representa os dados da aplicação e as regras do negócio que determinam o acesso aos dados persistentes, e também fornece ao controlador a capacidade de acessar as funcionalidades da aplicação (MACORATTI, 2017).

Um componente de visualização (V) renderiza o conteúdo de uma parte particular do modelo e encaminha para o controlador (C) as ações do usuário; além disso, acessa também os dados do modelo via controlador e define como esses dados devem ser apresentados. Por fim, o controlador define o comportamento da aplicação, interpretando as ações dos usuários mapeando-as para chamados do modelo (MACORATTI, 2017).

Figura 4.2 | Estrutura do MVC



Fonte: <<http://www.tutorialized.com/tutorial/Fundamentals-of-an-MVC-Framework/81946>>. Acesso em: 10 ago. 2017..



Questão para reflexão

Apesar do paradigma orientado a objetos representar uma espécie de padrão de projeto, pode-se utilizar outros padrões a projetos orientados a objetos? A resposta é sim, cada padrão de projeto possui um objetivo específico e complementar, são soluções específicas para grupos de problemas diferentes.

Atividades de aprendizagem

1. Na fase de implementação de um projeto orientado a objetos as linguagens de programação aplicadas nesta etapa devem:
 - a) Obrigatoriamente ser orientadas a objetos “puras”.
 - b) Possuir todas as outras características além de abstração de dados, definir tipos de objetos e carregar o conceito da herança.
 - c) Possuir mecanismos de conversão para outros paradigmas.
 - d) Possuir ao menos as características de abstração de dados e definir tipos

de objetos e carregar o conceito da herança.

e) Ser compatíveis somente com sistemas de banco de dados também orientado a objetos.

2. O padrão de projeto Singleton é usado para restringir:

- a) Classes de atributos complexos.
- b) Relações entre classes e objetos.
- c) A instanciação de uma classe para apenas um objeto.
- d) A instanciação de classes concretas.
- e) A quantidade de classes do programa.

Seção 2

Modelagem de dados

Introdução à seção

É comum em projetos orientados a objetos fazermos o uso de sistemas gerenciadores de bancos de dados baseados em outros paradigmas, bem como o uso de sistemas gerenciadores de banco de dados do tipo entidade relacionamento para realizar a persistência dos dados da aplicação.

Este tipo de sistema gerenciador de banco de dados precisa passar por uma adaptação em projetos orientados a objetos, por isso vamos falar sobre a modelagem de dados e como mesclar conceitos de programação orientada a objetos com sistemas de banco de dados baseados em relacionamentos.

2.1 Modelagem de dados

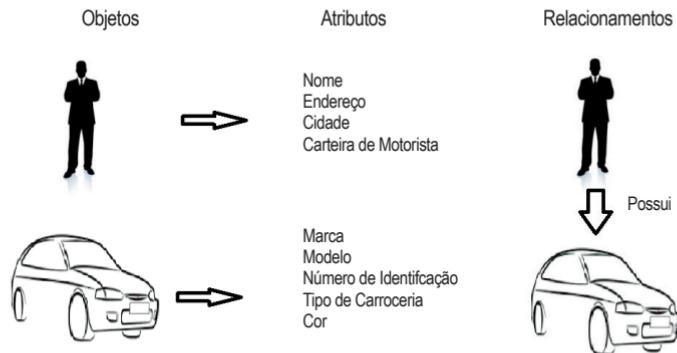
A modelagem de dados é usada em aplicações de banco de dados. Ela proporciona ao analista e ao projetista de banco de dados o esclarecimento sobre os dados e as relações que regem estes dados. A modelagem de dados pode ser usada para representar o conteúdo de depósitos de dados e as relações que existem entre eles.



É importante reconhecer que a OOA e a modelagem de dados são abordagens diferentes, com um ponto de vista bastante diferente. Ambas usam o termo “objeto”, mas a sua definição é muita mais limitada num contexto de modelagem de dados. A modelagem de dados faz exatamente aquilo que seu nome implica – ela modela dados –, sem se preocupar com os processos que devem ser aplicados para transformar os dados. Por conseguinte, ela é uma técnica complementar que atende a uma função de análise específica. Ela deve ser combinada com outra abordagem de modelagem que leve em consideração as questões de processamento, a fim de formar um método de análise de requisitos completo. (PRESSMAN, 1995, p. 341)

Quando usada como técnica de análise de requisitos, o analista começa o seu trabalho criando modelos para os objetos. Um objeto de dados (data object) é definido de uma maneira muito parecida com a usada para definir um objeto para OOA, por exemplo, uma pessoa e um carro podem ser considerados como um objeto de dados no sentido de que qualquer um deles pode ser definido em termos de um conjunto de atributos, conforme Figura 4.3 a seguir.

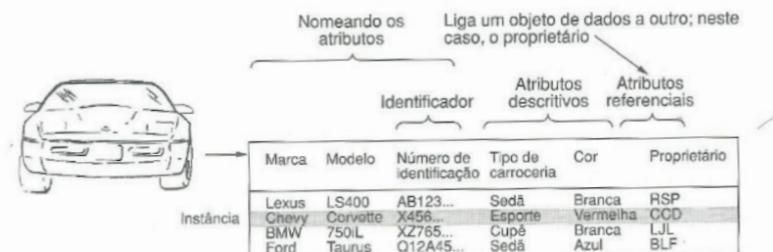
Figura 4.3 | Objetos de dados, atributos e relacionamentos



Fonte: Pressman (2011, p. 343).

Os objetos de dados relacionam-se uns com os outros, neste exemplo uma pessoa pode possuir carro em que o relacionamento possuir conota uma conexão específica entre pessoa e carro. Os relacionamentos são sempre definidos pelo contexto do problema que está sendo analisado. Assim, um objeto de dados não assume a mesma forma que o objeto no conceito de OOA, ele encapsula somente dados não havendo nenhuma referência em si sobre as operações que atuam sobre os dados. Portanto, o objeto de dados pode ser representado como uma tabela. Os títulos dessa tabela refletem atributos do objeto, no exemplo da Figura 4.4 um carro é definido em termos de marca, modelo, ID, tipo de carroceria, cor e proprietário. O corpo da tabela representa instâncias específicas do objeto de dados. Por exemplo, um Chevy Corvette é uma instância do objeto de dados carro (PRESSMAN, 2011).

Figura 4.4 | Objeto de dados como tabela



Fonte: Pressman (1995, p. 344).

As tabelas de objetos de dados seguem regras de normalização para garantir a mínima redundância de dados, assim, a quantidade de informações mantidas para satisfazer uma necessidade ou resolver um problema é reduzida (PRESSMAN, 2011).



Para saber mais

Assista aos vídeos sobre normalização de banco de dados no Youtube.

Vídeo 1: <<https://www.youtube.com/watch?v=e0XtaTvLqmA&list=PLQLGmi9EOFxsGsGNnsQHEROsbGBSd1dY>>.

Vídeo 2: <<https://www.youtube.com/watch?v=3kJKJNKiaD4&list=PLQLGmi9EOFxsGsGNnsQHEROsbGBSd1dY&index=2>>.

Vídeo 3: <<https://www.youtube.com/watch?v=3kJKJNKiaD4&list=PLQLGmi9EOFxsGsGNnsQHEROsbGBSd1dY&index=2.dY&index=2>>.

Vídeo 4: <<https://www.youtube.com/watch?v=mHoZZUYVFzk&index=4&list=PLQQLGmi9EOFxsGsGNnsQHEROsbGBSd1dY>>

Vídeo 5: <<https://www.youtube.com/watch?v=EzvrGEpyNbs&list=PLQQLGmi9EOFxsGsGNnsQHEROsbGBSd1dY&index=5>>. Acessos em: 4 out. 2017.



Para saber mais

O livro *Modelagem de banco de dados* também pode ser utilizado para detalhamento do assunto. Acesse o PDF através do link: <<http://www.unilivros.com.br/pdf/dbmod.pdf>>. Acesso em: 4 out. 2017.



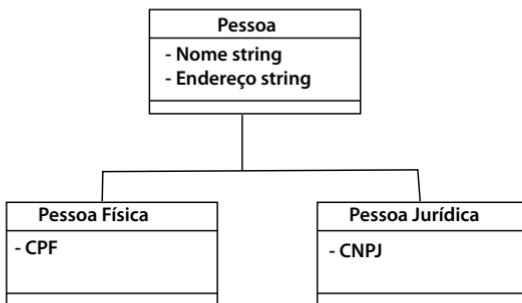
Questão para reflexão

Podemos utilizar tecnologias baseadas em outros paradigmas em projetos orientados a objetos? A resposta certamente é sim, no entanto, temos que aprender a utilizar essas tecnologias de maneira sinérgica com os princípios da orientação a objetos, pois é totalmente possível utilizar um banco de dados relacional em uma aplicação construída com linguagem de programação orientada a objetos desde que se conheçam as regras para realizar o mapeamento objeto-relacional.

2.2 Mapeamento objeto-relacional

Quando consideramos a utilização de um banco de dados relacional em um projeto orientado a objetos, facilmente caímos em uma armadilha, a tentação de transformar as classes mapeadas em um diagrama de classes diretamente em entidades (tabelas) do banco de dados. O clássico exemplo da classe **pessoa** e o relacionamento do tipo generalização para as classes **física** e **jurídica** como derivações da classe **pessoa** são a demonstração de como o raciocínio do desenvolvedor novato e com pouca experiência funcionará, pois facilmente a relação de uma classe para uma entidade surgirá gerando dúvidas de como estruturar as tabelas em uma representação física de um banco de dados através de um diagrama entidade-relacionamento. Aliás, aqui surge outra tentação, a de chamar diagramas de classe de diagrama entidade-relacionamento e vice-versa, o que demonstra claramente a confusão na utilização dos paradigmas OO e relacional.

Figura 4.5 | Exemplo de generalização



Fonte: elaborada pelo autor

Para evitar as armadilhas e confusões, faz-se necessário o pleno entendimento de que mais de um padrão de pensamento aplica-se ao projeto orientado a objetos e o conhecimento das diversas tecnologias em uso é o diferencial para que erros comuns de organização estrutural de código, de banco de dados e outros afetem a arquitetura geral da aplicação. Aqui surge também a necessidade de entender que as classes definidas na análise orientada a objetos deverão ser mapeadas em entidades de um banco de dados com seus devidos relacionamentos, assim as classes anteriores poderiam ser mapeadas através de técnicas de mapeamento objeto-relacional de acordo com o seu tipo de relacionamento definido pela UML (GUEDES, 2011). No exemplo da Figura 4.5 há um exemplo de relacionamento do tipo generalização, sendo que vamos explorar no tópico seguinte o mapeamento para este tipo de relacionamento.

2.3 Mapeamento para classes relacionadas por generalização

Segundo Guedes (2011), para classes relacionadas por generalização, deve-se considerar o tipo da restrição da generalização:

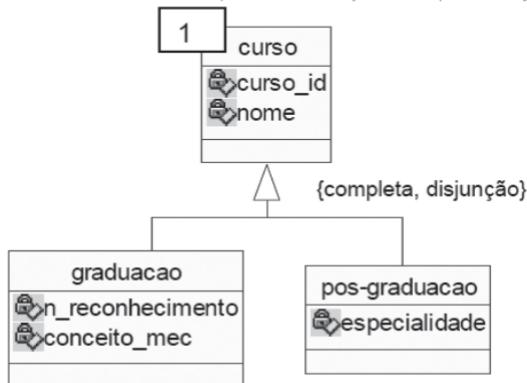
- Quando a relação é completa e disjuntiva: isto significa que as classes filhas estão totalmente definidas e não haverá inclusão de outras classes sob a classe pai e que um objeto instanciado de uma dessas classes pode ser somente um ou outro tipo de objeto, mas nunca os dois tipos de objetos ao mesmo tempo.
- Quando a relação é completa e há sobreposição: da mesma maneira que na restrição anterior, as classes filhas estão totalmente definidas, porém, o objeto instanciado pode ser simultaneamente um ou outro tipo de objeto originado das classes filhas.
- Quando a relação é incompleta: quando as classes filhas originadas de uma classe pai podem admitir novos tipos de objetos sob a classe pai.

Vamos analisar as possibilidades para cada tipo de restrição:

Possibilidade 1: a primeira possibilidade de mapeamento pode ser aplicada a qualquer tipo de restrição, assim, pode-se mapear uma tabela para a superclasse e uma tabela para cada subclasse. Na tabela que representa a superclasse, deve-se incluir um atributo “tipo” com valores definidos que representam as subclasses e nas tabelas que representam as subclasses incluir a chave primária, correspondendo à

tabela da superclasse, veja os exemplos nas figuras a seguir:

Figura 4.6 | Possibilidade 1 – Exemplo 1 – Restrição Completa/Disjunção



Fonte: elaborada pelo autor.

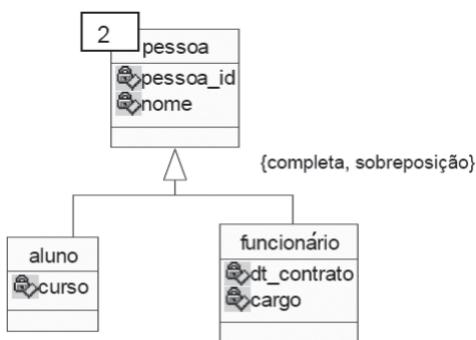
Esquema de mapeamento para tabela:

Curso: (curso_id, nome, tipo[G/P])

Graduacao: (curso_id, n_reconhecimento, conceito_mec)

Pos-Graduacao: (curso_id, especialidade)

Figura 4.7 | Possibilidade 1 – Exemplo 2 – Restrição Completa/Sobreposição



Fonte: elaborada pelo autor.

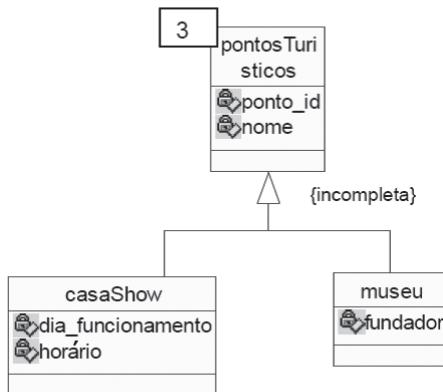
Esquema de mapeamento para tabela:

Pessoa: (pessoa_id, nome, tipo[A/F/AF])

Aluno: (pessoa_id, curso)

Funcionario: (pessoa_id, dt_contrato, cargo)

Figura 4.8 | Possibilidade 1 – Exemplo 3 – Restrição Incompleta



Fonte: elaborada pelo autor.

Esquema de mapeamento para tabela:

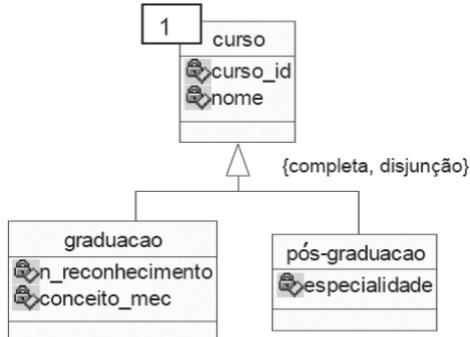
PontosTurísticos: (ponto_id, nome, tipo[CS/M/Outros])

CasaShow: (ponto_id, dia_funcionamento, horário)

Museu: (ponto_id, fundador)

Possibilidade 2: a segunda possibilidade de mapeamento pode ser aplicada apenas em relacionamentos do tipo generalização, quando esta for completa e disjuntiva. Neste caso, deve-se mapear uma tabela para cada subclasse, apenas incluindo os atributos da superclasse, a Figura 4.9 demonstra esta condição.

Figura 4.9 | Possibilidade 2 – Exemplo 1 – Restrição Completa/Disjunção



Fonte: elaborada pelo autor.

Esquema de mapeamento para tabela:

Graduação: (curso_idg, nome, n_reconhecimento, conceito_mec)

Pós-graduação: (curso_idp, nome, especialidade)

Possibilidade 3: a terceira possibilidade de mapeamento pode ser aplicada a qualquer relacionamento do tipo generalização para qualquer tipo de restrição, assim pode-se mapear uma única tabela para todos os atributos da superclasse e subclasses. Neste caso, deve-se incluir um atributo “tipo” com valores definidos que representam todas as classes. Vamos aos esquemas:

Curso: (curso_id, nome, n_reconhecimento, conceito_mec, especialidade, tipo[G/P])

Pessoa: (pessoa_id, nome, curso, dt_contrato, cargo, tipo[A/F/AF])

PontosTurísticos (ponto_id, nome, dia_funcionamento, horário, fundador, tipo[CS/M/Outros])

O domínio das regras de mapeamento baseados nos tipos de relacionamentos definidos na UML e suas restrições faz-se necessário para realizar um mapeamento assertivo de entidades, a fim de modelar corretamente a persistência de dados da aplicação em desenvolvimento.



Para saber mais

O livro *Sistema de banco de dados* no capítulo 4 aborda o tema Modelagem com entidade-relacionamento estendido e UML, vale a pena conferir!

ELMASRI, R. et al. **Sistema de banco de dados**. 4. ed. São Paulo: Makron Books, 2005.

2.4 Frameworks

Quando entramos na etapa de implementação de um software, sentimos inicialmente a necessidade de utilizar códigos previamente construídos, com o objetivo de entender melhor a sintaxe ou a

semântica. Isto é comum quando temos pouca experiência com o desenvolvimento e procuramos exemplos de como aplicar todos os conceitos de análise e design na construção de um código. Com o passar do tempo as necessidades serão modificadas, conforme o programador ganha experiência com uma determinada linguagem de programação, ele começará a questionar o modo como tudo deve ser construído, surge então uma percepção de agilidade. E como ser ágil na construção de um software tendo que construir códigos completos do zero? Como organizar todo o código de maneira a separar lógica de persistência da lógica de programação? Como separar essas lógicas da interface sem ter que construir do zero incontáveis classes, incluir vários componentes, criar vários métodos, utilizar várias bibliotecas e ainda respeitar a organização lógica de tudo isso?

Para começar a responder às questões anteriores, conheceremos um pouco sobre *frameworks*, o que são e quais as vantagens no seu uso na construção e organização dos códigos-fonte que originarão os programas.

A primeira compreensão sobre *frameworks* é que eles não são o sinônimo de padrões de projetos, são componentes utilizáveis normalmente implementados, utilizando linguagens de programação orientada a objetos, enquanto os padrões de projetos são conceitos abstratos que podem ser implementados através dos *frameworks*. Além disso, *frameworks* são mais específicos que padrões de projeto, pois estão relacionados a um domínio de aplicação, um aspecto de infraestrutura ou de integração de middleware, e padrões de projetos são mais gerais e podem ser utilizados em diversas situações, independentemente do domínio da aplicação.

Segundo Fayad e Schmidt (1997), o benefício da utilização de *frameworks* advém da modularidade, reusabilidade, extensibilidade e inversão de controle. O aumento da modularidade de uma aplicação é alcançado através do encapsulamento dos detalhes de implementação proporcionados pelo *framework*. Não há necessidade de alterar classes e criar novos códigos para que ele funcione, inclusive, os locais de mudança de projeto e implementação e implementação da aplicação construída usando o *framework* são localizados, o que diminui o esforço para entender e manter a aplicação.

O reuso de código é incentivado, pois captura o conhecimento de desenvolvedores em determinado domínio ou aspecto de infraestrutura

ou de integração de middleware. A possibilidade de compartilhar funcionalidade definidas em um *framework* proporciona o aumento de produtividade dos desenvolvedores, pois a construção não começa do zero. Já que essas funcionalidades já foram testadas em várias instâncias, também há o aumento da qualidade e confiabilidade do produto final.

Frameworks oferecem pontos de extensão explícitos que permitem aos programadores estender suas funcionalidades para gerar uma aplicação, e por fim, alguns apresentam inversão de controle (Inversion of Control – IOC). Trata-se de transferir o controle da execução da aplicação para o *framework*, que chama a aplicação quando necessário, através da IOC o *framework* controla quais métodos da aplicação e em quais contextos eles serão chamados em decorrência de eventos, como eventos de janela, um pacote enviado para determinada porta, entre outros.

Dois exemplos comuns de *frameworks* largamente utilizados em conjunto com a linguagem de programação Java são:

- Hibernate: a fim de tornar compatível o paradigma da orientação a objetos e o paradigma de entidade e relacionamento, foram desenvolvidos *frameworks* de mapeamento objeto relacional. Sua principal característica é a transformação das classes em Java para tabelas de dados (e dos tipos de dados Java para os da SQL). O Hibernate gera as chamadas SQL e libera o desenvolvedor do trabalho manual de transformação, mantendo o programa portável para quaisquer bancos de dados SQL.
- Spring MVC: O *framework* Spring, além do suporte para a arquitetura N, camadas do padrão MVC, fornece também uma implementação para os padrões inversão de controle e injeção de dependência, entre muitos outros recursos.



Para saber mais

Leia o material indicado:

Componentes, Frameworks e Design Patterns. Disponível em: <<http://www.dca.fee.unicamp.br/~gudwin/ftp/ea976/Patterns.pdf>>. Acesso em: 4 out. 2017. *Utilização do Framework Hibernate em aplicação JAVA WEB*. Disponível em: <<http://web.unipar.br/~seinpar/2013/artigos/Carlos%20Filipe%20Magalhaes.pdf.pdf>>. Acesso em: 4 out. 2017.

Vire o jogo com Spring Framework. Disponível em: <<http://www.rondinha.rs.leg.br/restrito/upload/legislacao/2.pdf>>. Acesso em: 4 out. 2017.

Atividades de aprendizagem

1. O mapeamento objeto-relacional é um mecanismo utilizado para permitir a utilização de banco de dados relacionais em conjunto com linguagens de programação orientadas a objetos. O objetivo principal deste mecanismo é:

- a) Mudar o paradigma relacional dos sistemas de banco de dados para o paradigma OO.
- b) Realizar uma relação direta entre classes e entidades.
- c) Incluir uma camada extra de programação para traduzir todos os objetos da programação OO em objetos relacionais.
- d) Diminuir a impedância da programação orientada a objetos e permitir a utilização do banco de dados relacional
- e) Incluir uma camada extra de programação para traduzir todos os objetos relacionais em objetos da programação OO.

2. Framework pode ser definido como:

- a) Uma especificação de requisitos de um sistema.
- b) Um armazenamento persistente de informações sobre itens de dados encontrados em um diagrama de classes.
- c) Uma combinação de componentes que simplifica a construção de aplicações e pode ser conectada em uma aplicação.
- d) Um conceito abstrato que deve ser implementado através de alguma linguagem de programação.
- e) Uma descrição dos principais recursos de um produto de software.

Seção 3

Arquitetura de software

Introdução à seção

A arquitetura de software, apesar de não ser uma tecnologia, está neste capítulo, pois, a partir da definição dos padrões de arquitetura utilizada no projeto de software é possível definir os tipos de tecnologias que serão empregadas, portanto, torna-se imprescindível a compreensão de como deverá ser organizado o software em termos lógicos e físicos para uma escolha adequada dos recursos tecnológicos.

3.1 Projeto de arquitetura

A arquitetura de software é importante, pois afeta o desempenho e a robustez, bem como a capacidade de distribuição e manutenibilidade de um sistema (BOSCH, 2000).

O projeto de arquitetura está preocupado com a compreensão de como um sistema deve ser organizado com a estrutura geral do sistema. No modelo do processo de desenvolvimento de software, o projeto de arquitetura é o primeiro estágio no processo de projeto de software. É o elo crítico entre o projeto e a engenharia de requisitos, pois identifica os principais componentes estruturais de um sistema e os relacionamentos entre eles. O resultado do processo de projeto de arquitetura é um modelo de arquitetura que descreve como o sistema está organizado em um conjunto de componentes de comunicação. (SOMMERRVILLE, 2011, p. 103)

Os componentes individuais implementam os requisitos funcionais do sistema enquanto os requisitos não funcionais dependem da arquitetura do sistema, a forma como esses componentes estão organizados e se comunicam. Em muitos sistemas, os requisitos funcionais são também influenciados por componentes individuais,

mas não há dúvida de que a arquitetura de sistema é a influência dominante (SOMMERVILLE, 2011).

Sommerville (2011) define, ainda, dois níveis de abstração possíveis para projetar as arquiteturas de software, em pequena e grande escala.

A arquitetura em pequena escala preocupa-se com a arquitetura de programas individuais. Nesse nível, o foco está na maneira como um programa individual é decomposto em componentes. Por outro lado, a arquitetura em grande escala preocupa-se com a arquitetura de programas corporativos complexos que incluem outros sistemas, programas e componentes de programas. Esses sistemas estão distribuídos por diversos computadores, que podem pertencer e ser geridos por diferentes empresas.

3.2 Decisões de projeto de arquitetura

Conforme expresso por Sommerville (2011, p. 105), os arquitetos de software precisam tomar algumas decisões estruturais que impactam o sistema e o seu processo de desenvolvimento.



O projeto de arquitetura é um processo criativo no qual você projeta uma organização de sistema para satisfazer aos requisitos funcionais e não funcionais de um sistema. Por ser um processo criativo, as atividades no âmbito do processo dependem do tipo de sistema a ser desenvolvido, a formação e a experiência do arquiteto de sistema e os requisitos específicos para o sistema. Por isso, é útil pensar em projeto de arquitetura como uma série de decisões, em vez de uma sequência de atividades.

Assim, os arquitetos de software devem considerar as seguintes questões:

1. Existe uma arquitetura genérica que pode atuar como um modelo para o sistema que está sendo modelado?
2. Como a aplicação será distribuída por um número de núcleos/processadores?
3. Padrões de arquitetura podem ser utilizados?

4. Componentes estruturais do sistema serão decompostos em subcomponentes? Como?
5. Há estratégia para controlar o funcionamento dos componentes de sistema?
6. Como organizar a arquitetura para satisfazer os requisitos funcionais e não funcionais do sistema?
7. Como validar o projeto de arquitetura?
8. Como documentar o sistema de arquitetura?

A arquitetura de um sistema de software pode se basear em um determinado padrão ou estilo de arquitetura. Um padrão de arquitetura é uma descrição de uma organização do sistema (SHAW; GARLAN, 1996), como uma organização cliente-servidor ou uma arquitetura em camadas. Os padrões de arquitetura capturam a essência de uma arquitetura que tem sido usada em diferentes sistemas de software. Ao tomar decisões sobre a arquitetura de um sistema, você deve conhecer os padrões comuns, bem como saber em que eles podem ser usados e quais são os seus pontos fortes e fracos (SOMMERVILLE, 2011).

Sommerville (2011) defende ainda que devido à estreita relação entre os requisitos não funcionais e a arquitetura de software, o estilo e a estrutura da arquitetura particular que você escolhe para um sistema devem depender dos requisitos não funcionais do sistema:

1. Desempenho: sendo este um requisito crítico, a arquitetura deve ser projetada para localizar as operações críticas dentro de um pequeno número de componentes, com estes componentes implantados no mesmo computador, ao invés de distribuídos pela rede. Ocasionalmente, haverá o uso de alguns componentes relativamente grandes, que reduzem o número de comunicações entre eles.
2. Proteção: sendo este um requisito crítico, então, deve-se usar uma estrutura em camadas para a arquitetura. Os ativos mais críticos devem ser protegidos nas camadas mais internas, incluindo alto nível de validação de proteção aplicado a estas camadas.
3. Segurança: neste caso, a arquitetura deve ser concebida de modo que as operações relacionadas com segurança estejam localizadas em poucos ou em um único componente proporcionando a redução dos custos e validação da segurança. Fornecer sistemas de proteção que em caso de falhas possam desligar a aplicação.

4. Disponibilidade: sendo este um requisito crítico, deve-se então projetar a arquitetura para incluir componentes redundantes, possibilitando a substituição e atualização de componentes sem a necessidade de parar o sistema.

5. Manutenção: sendo este um requisito crítico, projetar uma arquitetura baseada em componentes autocontidos de baixa granularidade para serem rapidamente alterados. As estruturadas de dados compartilhadas devem ser evitadas separando-se os geradores de dados dos consumidores.

Percebemos a dificuldade em avaliar um projeto de arquitetura, pois, uma vez definida, o principal teste será o quanto bem o sistema satisfaz os requisitos funcionais e não funcionais quando em utilização. Você pode fazer alguma avaliação comparando seu projeto contra arquiteturas de referência ou padrões genéricos de arquitetura.

3.3 Padrões de arquitetura

Os padrões de arquitetura foram propostos na década de 1990 sob o nome de estilos de arquitetura (SHAW; GARLAN, 1996). Sommerville (2011) define estes estilos como uma descrição abstrata, estilizada, de boas práticas experimentadas e testadas em diferentes sistemas e ambientes. Assim, um estilo ou padrão de arquitetura deve descrever uma organização de sistema bem-sucedida em sistemas anteriores, deve também incluir informações quando o uso desse padrão é adequado, seus pontos fortes e fracos.

O próprio padrão de projetos MVC citado na seção anterior é considerado um padrão de arquitetura e está descrito a seguir:

Quadro 4.1 | O padrão Modelo-Visão-Controlador (MVC)

Nome	MVC (Modelo-Visão-Controlador)
Descrição	Separa a apresentação e a interação dos dados do sistema. O sistema é estruturado em três componentes lógicos que interagem entre si. O componente Modelo gerencia o sistema de dados e as operações associadas a esses dados. O componente Visão define e gerencia como os dados são apresentados ao usuário. O componente Controlador gerencia a interação do usuário (por exemplo, teclas, cliques do mouse etc.) e passa essas interações para a Visão e o Modelo. Veja a Figura 4.10.
Exemplo	A Figura 4.11 mostra a arquitetura de um sistema aplicativo baseado na Internet, organizado pelo uso do padrão MVC.
Quando é usado	É usado quando existem várias maneiras de se visualizar e interagir com dados. Também quando são desconhecidos os futuros requisitos de interação e apresentação de dados.

Vantagens	Permite que os dados sejam alterados de forma independente de sua representação, e vice-versa. Apoia a apresentação dos mesmos dados de maneiras diferentes, com as alterações feitas em uma representação aparecendo em todas elas.
Desvantagens	Quando o modelo de dados e as interações são simples, pode envolver código adicional e complexidade de código.

Fonte: adaptado de: Sommerville (2011, p.109).

A Figura 4.10 é uma visão conceitual, enquanto a Figura 4.11 mostra uma possível arquitetura *run-time*, quando esse padrão é utilizado para gerenciamento de interações em um sistema baseado em web.

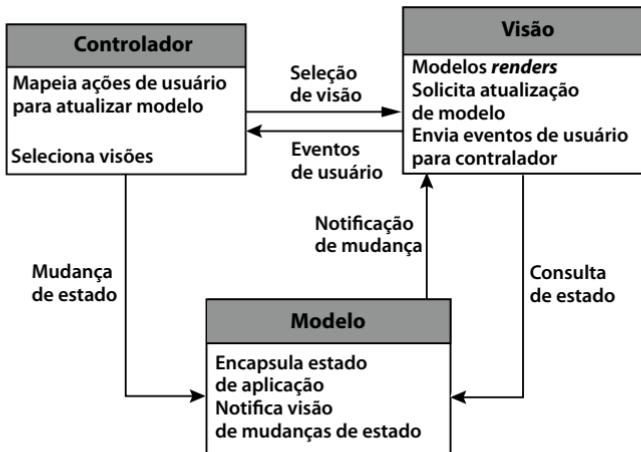
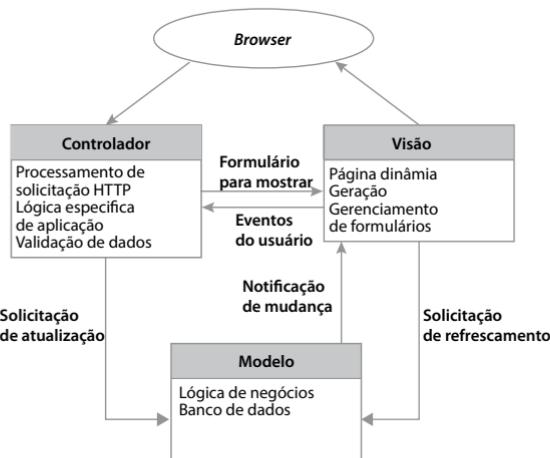


Figura 4.11 | Arquitetura de aplicações web usando o padrão MVC



Fonte: adaptada de: Sommerville (2011, p. 110).

3.4 Arquitetura em camadas

O padrão de arquitetura em camadas é uma maneira de conseguir separação e independência funcional. Neste padrão a funcionalidade do sistema é organizada em camadas separadas em que cada uma depende somente dos recursos e serviços oferecidos pela imediatamente abaixo dela.

Esta abordagem de organização em camadas favorece o desenvolvimento incremental aos sistemas, assim, enquanto uma camada é desenvolvida, alguns dos serviços prestados por ela podem ser disponibilizados aos usuários, além disso, quando a camada de interface muda ou tem novos recursos adicionados, apenas a camada adjacente é afetada (SOMMERVILLE, 2011).

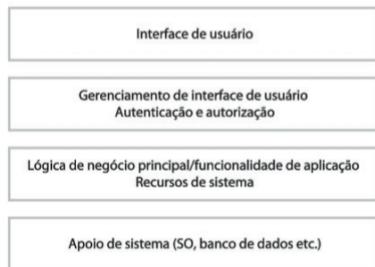
Quadro 4.2 | O padrão de arquitetura em camadas

Nome	Arquitetura em camadas
Descrição	Organiza o sistema em camadas com a funcionalidade relacionada associada a cada camada. Uma camada fornece serviços à camada acima dela; assim, os níveis mais baixos de camadas representam os principais serviços suscetíveis de serem usados em todo o sistema. Veja a Figura 4.12.
Exemplo	Um modelo em camadas de um sistema para compartilhar documentos com direitos autorais, em bibliotecas diferentes, como mostrado na Figura 4.13.
Quando é usado	É usado na construção de novos recursos em cima de sistemas existentes; quando o desenvolvimento está espalhado por várias equipes, com a responsabilidade de cada equipe em uma camada de funcionalidade; quando há um requisito de proteção multinível.
Vantagens	Desde que a interface seja mantida, permite a substituição de camadas inteiras. Recursos redundantes (por exemplo, autenticação) podem ser fornecidos em cada camada para aumentar a confiança do sistema.
Desvantagens	Na prática, costuma ser difícil proporcionar uma clara separação entre as camadas, e uma camada de alto nível pode ter de interagir diretamente com camadas de baixo nível, em vez de através da camada imediatamente abaixo dela. O desempenho pode ser um problema por causa dos múltiplos níveis de interpretação de uma solicitação de serviço, uma vez que são processados em cada camada.

Fonte: adaptado de Sommerville (2011, p. 110).

A Figura 4.12 é um exemplo de arquitetura em camadas, com quatro camadas:

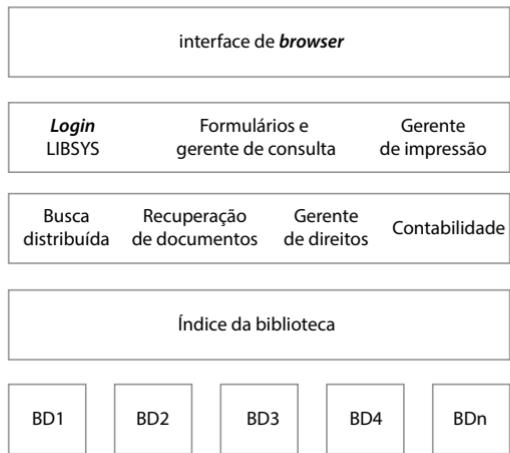
Figura 4.12 | Arquitetura genérica em camadas



Fonte: adaptada de Sommerville (2011, p. 111).

A Figura 4.13 é um exemplo de como esse padrão de arquitetura em camadas pode ser aplicado a um sistema de biblioteca chamado LIBSYS.

Figura 4.13 | Arquitetura do sistema LIBSYS



Fonte: adaptada de Sommerville (2011, p. 111).

3.5 Arquitetura cliente-servidor

Um sistema baseado na arquitetura cliente-servidor é organizado como um conjunto de serviços e servidores associados e clientes que acessam e usam os serviços. Sommerville (2011) define os principais componentes desse modelo:

1. Conjunto de servidores: servidores de impressão, de arquivos, de compilação e outros.
2. Clientes: utilizam os serviços oferecidos pelos servidores, geralmente há várias instâncias de um programa cliente executando simultaneamente em computadores diferentes (multiusuário).
3. Rede de computadores: que permite aos clientes acessar os serviços. A maioria dos sistemas cliente-servidor é implementada como sistemas distribuídos, conectados através de protocolos de internet.

Os clientes acessam os serviços fornecidos por um servidor por meio de chamadas de procedimento remoto, usando um protocolo de solicitação-resposta, tal como o protocolo HTTP usado na internet. Essencialmente, um cliente faz uma solicitação a um servidor e espera receber uma resposta.

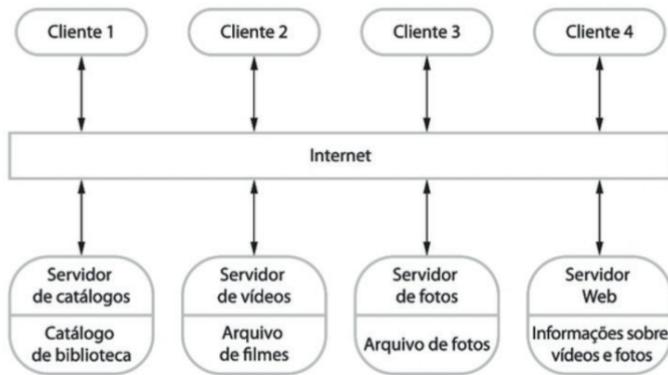
Quadro 4.3 | O padrão cliente-servidor

Nome	Cliente-servidor
Descrição	Em uma arquitetura cliente-servidor, a funcionalidade do sistema está organizada em serviços — cada serviço é prestado por um servidor. Os clientes são os usuários desses serviços e acessam os servidores para fazer uso deles.
Exemplo	A Figura 4.14 é um exemplo de uma biblioteca de filmes e vídeos/DVDs, organizados como um sistema cliente-servidor.
Quando é usado	É usado quando os dados em um banco de dados compartilhado precisam ser acessados a partir de uma série de locais. Como os servidores podem ser replicados, também pode ser usado quando a carga em um sistema é variável.
Vantagens	A principal vantagem desse modelo é que os servidores podem ser distribuídos através de uma rede. A funcionalidade geral (por exemplo, um serviço de impressão) pode estar disponível para todos os clientes e não precisa ser implementada por todos os serviços.
Desvantagens	Cada serviço é um ponto único de falha suscetível a ataques de negação de serviço ou de falha do servidor. O desempenho, bem como o sistema, pode ser imprevisível, pois depende da rede. Pode haver problemas de gerenciamento se os servidores forem propriedade de diferentes organizações.

Fonte: adaptado de Sommerville (2011, p. 113).

A Figura 4.14 é um exemplo de sistema baseado no modelo cliente-servidor. É um sistema multiusuário baseado na internet para fornecimento de uma biblioteca de filmes e fotos. A principal vantagem desse modelo é que se trata de uma arquitetura distribuída, além disso, há relativa facilidade em adicionar um novo servidor e integrá-lo com o resto do sistema de forma transparente, se afetar outras partes do sistema.

Figura 4.14 | Arquitetura cliente-servidor para biblioteca de filmes



Fonte: adaptada de Sommerville (2011, p. 114).

Obviamente, não esgotamos aqui todos os estilos de arquitetura existentes e que podem ser aplicados nos projetos de desenvolvimento de software orientado a objetos, apenas ressaltamos nos tópicos anteriores desta seção alguns exemplos comumente utilizados em aplicações baseadas na web. Assim, vários outros estilos podem ser adotados derivados das decisões tomadas a partir da análise das

condições em que tal sistema deva existir, portanto, lembre-se de que a opção em utilizar um ou outro padrão de arquitetura dependerá das questões relevantes relacionadas aos requisitos funcionais e não funcionais que devem ser atendidos.



Questão para reflexão

Em um projeto de sistemas orientado a objetos a arquitetura definida pode ser definida conforme as principais condições relacionadas aos requisitos não funcionais e funcionais a serem atendidas? Sim, a arquitetura que determinará a organização de uma aplicação orientada a objetos deve levar em consideração os requisitos como segurança, disponibilidade, entre outros, estes, em conjunto com os requisitos funcionais, formarão os pré-requisitos básicos para a definição da arquitetura da aplicação.



Para saber mais

Veja mais sobre arquitetura de software acessando: <<https://imasters.com.br/framework/dotnet/net-definindo-a-arquitetura-de-um-projeto-de-software/?trace=1519021197&source=single>> e <http://www.funpar.ufpr.br:8080/rup/process/workflow/ana_desi/co_swarch.htm>. Acesso em: 4 out. 2017.

Atividades de aprendizagem

1. (Fundação La Salle, 2017, SUSEPE-RS, Agente Penitenciário) O modelo cliente/servidor é bastante utilizado tanto no contexto da intranet, quanto do acesso à internet nas empresas. Sobre este modelo, analise as assertivas:

- I - É um modelo composto por, pelo menos, dois equipamentos interligados em rede.
- II- O cliente envia a solicitação ao servidor que executa o que foi solicitado ou procura a informação solicitada e retorna ao cliente.
- III - O servidor é sempre responsável por iniciar uma comunicação, sendo que o cliente trabalha de forma reativa, respondendo às requisições do servidor.

É (são) característica(s) do modelo cliente/servidor, o que se afirma em:

- a) Apenas I.
- b) Apenas III.
- c) Apenas I e II.

- d) Apenas II e III.
- e) I, II e III.

2. (CESPE, 2016, TRE-PE, Técnico Judiciário – Operação de Computadores)

Tendo em vista que a arquitetura cliente-servidor pode ser modelada em três camadas, apresentação, domínio e fonte de dados, assinale a opção correta:

- a) Na arquitetura em questão, a ligação entre a camada de apresentação e a de armazenamento de dados não é realizada de forma direta.
- b) A camada de apresentação trata da interação entre o usuário e o software, como uma interface gráfica em um navegador.
- c) A solicitação dos usuários pode ser tratada pela camada de apresentação ou pela camada de domínio, como, requisições HTTP e chamadas em linhas de comando, respectivamente.
- d) Traduzir comandos do usuário em ações sobre o domínio é uma função de fontes de dados.
- e) A lógica de negócios está mais bem relacionada à camada de fonte de dados do que à de domínio.

Fique ligado

Nesta unidade, estudamos:

- A aplicação de linguagens orientadas a objeto e padrões de projeto.
- A modelagem de dados, o mapeamento objeto-relacional e frameworks.
- As arquiteturas de sistemas.

Para concluir o estudo da unidade

Muito bem, chegamos ao fim desta unidade! Nela abordamos algumas tecnologias essenciais para a construção de um software implementado através de um projeto orientado a objetos. Você deve ter notado que apesar da variedade de assuntos, há uma vasta quantidade de conceitos e tecnologias que não exploramos e que ainda deverão ser perseguidos para compreender de maneira ampla como projetar

uma aplicação. Não focamos em assuntos específicos relacionados ao hardware ou a redes de computadores, mas, sim, navegamos nas questões lógicas que o levarão automaticamente a questionar-se sobre a utilização de todos os componentes físicos, lógicos, de comunicação ou de organização que devem ser pensados em um projeto. Assim, a dica é trivial, não pare por aqui, aprofunde seus conhecimentos, ganhe experiência e torne-se um profissional com condições de atuar nas várias etapas de um projeto de software.

Sucesso!

Atividades de aprendizagem da unidade

1. (COPESE - UFPI, 2017, UFPI, Analista de Tecnologia da Informação) O conceito de programação orientada a objeto foi amplamente difundido a partir da evolução da linguagem de programação *Smalltalk*, em sua versão 80. Sobre o paradigma de programação orientada a objetos, assinale a opção INCORRETA.

- a) Os tipos de dados abstratos em linguagens orientadas a objeto usualmente são chamados de classes.
- b) Uma classe definida pela herança de outra é comumente chamada de classe derivada ou subclasse.
- c) Uma mensagem consiste de uma chamada a um objeto para invocar um de seus métodos.
- d) Toda e qualquer linguagem orientada a objetos suporta encapsulamento e herança múltipla.
- e) O polimorfismo é definido como o uso de um ponteiro ou referência polimórfica, para acessar um método cujo nome é sobreposto na hierarquia de classes.

2. (FGV, 2011, SEFAZ-RJ, Auditor Fiscal da Receita Estadual) Para que um sistema de informação possa ser útil e confiável, deve ser fundamentado na modelagem de dados, para posterior análise do processo. A modelagem de dados se baseia nos seguintes elementos:

- a) Fluxos de dados, atributos e requisitos.
- b) Fluxos de dados, diagramas e requisitos.
- c) Classes de dados, métodos e componentes.
- d) Objetos de dados, diagramas e componentes.
- e) Objetos de dados, atributos e relacionamentos.

3. Ao modelar um banco de dados devemos considerar as características das entidades que irão compor o banco, pois as características das entidades devem ser relevantes para o contexto da aplicação. As características são representadas por meio dos:

- a) Domínios.
- b) Relacionamentos.
- c) Objetos
- d) Atributos.
- e) Diagramas.

4. (CESPE, 2016, TRE-PI, Técnico Judiciário – Programação de Sistemas) Com relação ao modelo cliente/servidor, assinale a opção correta.

- a) Neste modelo, um cliente requisita serviços e um servidor é definido como o provedor de serviços, de modo que uma única máquina pode atuar como cliente e ao mesmo tempo como servidor.
- b) A comunicação requisição-resposta é assíncrona, uma vez que o processo cliente fica bloqueado até que a resposta seja enviada pelo servidor.
- c) As chamadas de procedimentos a distância (remote procedure call), que são eficientes mecanismos de comunicação usados nos sistemas distribuídos não podem ser utilizadas no modelo cliente/servidor.
- d) A arquitetura cliente/servidor, em relação à concorrência, é considerada um sistema distribuído. Um sistema é concorrente quando é possível aumentar o número de recursos compartilhados, sem que seja necessário mudar softwares de aplicação e sistemas.
- e) No modelo em questão, cada processo servidor deve ser visto como um provedor distribuído dos recursos que gerência.

5. (FCC, 2012, TER-CE, Programador de Computador) Com relação ao framework Hibernate, é correto afirmar que:

- a) Em uma aplicação Java realiza a persistência automatizada dos objetos para as tabelas de um banco de dados relacional, para isso usa metadados (descrição dos dados) que descrevem o mapeamento entre os objetos e o banco de dados.
- b) Em sistemas que fazem muito uso de stored procedures, triggers ou que implementam a maior parte da lógica da aplicação no banco de dados é uma boa opção, pois vai se beneficiar mais com o uso do Hibernate.
- c) Permite o envio unidirecional da representação de dados de um banco

de dados relacional para um modelo de objeto utilizando um esquema baseado exclusivamente em Hibernate Query Language (HQL).

d) A JPA (Java Persistence API) é parte do Enterprise JavaBeans 4.0.

e) Uma aplicação criada com Hibernate possui cada classe de persistência mapeada em um arquivo XML que deve ser salvo obrigatoriamente com o nome da classe seguido pelo sufixo .map.xml.

Referências

ALVES, Pereira Willian. **Análise e projeto de sistemas**: estudo prático. 1. ed. São Paulo: Érica, 2017.

BARNES, David J. et al. **Programação Orientada a Objetos com Java**. São Paulo: Pearson Education do Brasil, 2004.

BOOCH, Grady. **Object oriented design with applications**. EUA: The Benjamin Cummings Publishing Company, 1991.

_____. **Object-oriented analysis and design with applications**. 2. ed. EUA: Addison Wesley Longman, 1994.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **The Unified Modeling Language**. EUA: Addison Wesley Publishing Company, 1999.

BOSCH, Jan. **Design and use of software architecture**. University of Karlskrona/Ronneby, Sweden: Addison Wesley Professional, 2000.

GUEDES, Gilleane T. A. **UML 2**: uma abordagem prática. 2. ed. São Paulo: Novatec, 2011.

FAYAD, Mohamed E.; SCHMIDT, Douglas C. **Object-oriented application frameworks**. Communications of the ACM, v. 40, n. 10, p. 32-38, 1997.

KENN, Peter G. W. **Guia gerencial para a tecnologia da informação**: conceitos essenciais e terminologia para empresas e gerentes. Rio de Janeiro: Campus, 1996.

MACORATTI, José Carlos. **Padrões de projeto**: o modelo MVC - Model View Controller. 2017. Disponível em: <http://www.macoratti.net/vbn_mvc.htm>. Acesso em: 10 ago. 2017.

PRESSMAN, Roger. **Engenharia de software**. 3. ed. São Paulo: Pearson Makron Books, 1995.

RUMBAUGH, James et al. **Object-oriented modeling and design**. 1. ed. New Jersey: Prentice-Hall, 1990.

SHAW, Mary; GARLAN, David. **Software architecture**: perspectives on an emerging discipline. 1. ed. New Jersey: Prentice-Hall, 1996.

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

Anotações

Anotações

Anotações

Anotações

ISBN 978-85-522-0297-4



9 788552 202974 >