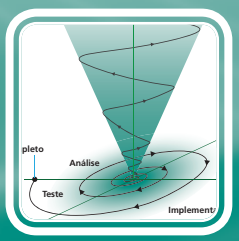
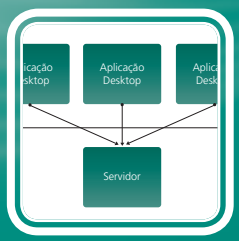
e-Tec Brasil
Escola Técnica Aberta do Brasil

Análise de Sistemas

Luiz Egidio Costa Cunha
José Inácio Serafini

Curso Técnico de Informática

INSTITUTO FEDERAL
ESPÍRITO SANTO

UNIVERSIDADE FEDERAL
DE SANTA CATARINA

Ministério da Educação

e-Tec Brasil
Escola Técnica Aberta do Brasil

Análise de Sistemas

Luiz Egidio Costa Cunha
José Inácio Serafini

Colatina - ES
2011

Presidência da República Federativa do Brasil

Ministério da Educação

Secretaria de Educação a Distância

Este Caderno foi elaborado em parceria entre o Instituto Federal do Espírito Santo e a Universidade Federal de Santa Catarina para o Sistema Escola Técnica Aberta do Brasil – e-Tec Brasil.

Equipe de Elaboração
Instituto Federal do Espírito Santo – IFES

Coordenação Institucional
Guilherme Augusto De Morais Pinto/IFES
João Henrique Caminhas Ferreira/IFES

Coordenação do Curso
Allan Francisco Forzza Amaral/IFES

Professores-autores
Luiz Egidio Costa Cunha/IFES
José Inácio Serafini/IFES

Comissão de Acompanhamento e Validação
Universidade Federal de Santa Catarina – UFSC

Coordenação Institucional
Araci Hack Catapan/UFSC

Coordenação do Projeto
Silvia Modesto Nassar/UFSC

Coordenação de Design Instrucional
Beatriz Helena Dal Molin/UNIOESTE e UFSC

Coordenação de Design Gráfico
André Rodrigues/UFSC

Design Instrucional
Renato Cislaghi/UFSC

Web Master
Rafaela Lunardi Comarella/UFSC

Web Design
Beatriz Wilges/UFSC
Mônica Nassar Machuca/UFSC

Diagramação
Bárbara Zardo/UFSC
Juliana Tonietto/UFSC
Marília C. Hermoso/UFSC
Nathalia Takeuchi/UFSC

Revisão
Júlio César Ramos/UFSC

Projeto Gráfico
e-Tec/MEC

C972a Cunha, Luiz Egidio Costa

Análise de sistemas: Curso Técnico de Informática / Luiz Egidio Costa Cunha. – Colatina: CEAD / Ifes, 2011.
104 p. : il.

ISBN: 978-85-62934-03-2
1. Análise de sistemas. 2. Software - Desenvolvimento. 3. Material didático. I. Instituto Federal do Espírito Santo. II. Título.

CDD: 004.21

Apresentação e-Tec Brasil

Prezado estudante,

Bem-vindo ao e-Tec Brasil!

Você faz parte de uma rede nacional pública de ensino, a Escola Técnica Aberta do Brasil, instituída pelo Decreto nº 6.301, de 12 de dezembro 2007, com o objetivo de democratizar o acesso ao ensino técnico público, na modalidade a distância. O programa é resultado de uma parceria entre o Ministério da Educação, por meio das Secretarias de Educação a Distancia (SEED) e de Educação Profissional e Tecnológica (SETEC), as universidades e escolas técnicas estaduais e federais.

A educação a distância no nosso país, de dimensões continentais e grande diversidade regional e cultural, longe de distanciar, aproxima as pessoas ao garantir acesso à educação de qualidade, e promover o fortalecimento da formação de jovens moradores de regiões distantes, geograficamente ou economicamente, dos grandes centros.

O e-Tec Brasil leva os cursos técnicos a locais distantes das instituições de ensino e para a periferia das grandes cidades, incentivando os jovens a concluir o ensino médio. Os cursos são ofertados pelas instituições públicas de ensino e o atendimento ao estudante é realizado em escolas-polo integrantes das redes públicas municipais e estaduais.

O Ministério da Educação, as instituições públicas de ensino técnico, seus servidores técnicos e professores acreditam que uma educação profissional qualificada – integradora do ensino médio e educação técnica, – é capaz de promover o cidadão com capacidades para produzir, mas também com autonomia diante das diferentes dimensões da realidade: cultural, social, familiar, esportiva, política e ética.

Nós acreditamos em você!

Desejamos sucesso na sua formação profissional!

Ministério da Educação
Janeiro de 2010

Nosso contato
etecbrasil@mec.gov.br

Indicação de ícones

Os ícones são elementos gráficos utilizados para ampliar as formas de linguagem e facilitar a organização e a leitura hipertextual.

Atenção: indica pontos de maior relevância no texto.

Saiba mais: oferece novas informações que enriquecem o assunto ou "curiosidades" e notícias recentes relacionadas ao tema estudado.

Glossário: indica a definição de um termo, palavra ou expressão utilizada no texto.

Mídias integradas: sempre que se desejar que os estudantes desenvolvam atividades empregando diferentes mídias: vídeos, filmes, jornais, ambiente AVEA e outras.

Atividades de aprendizagem: apresenta atividades em diferentes níveis de aprendizagem para que o estudante possa realizá-las e conferir o seu domínio do tema estudado.

Sumário

Palavra do professor-autor

Prezado estudante!

Neste curso você vai aprimorar seus conhecimentos sobre o desenvolvimento de *software*.

Vamos estudar com mais profundidade os aspectos das fases de análise e projeto de sistemas.

O que vamos estudar nesta disciplina é, resumidamente, como transformar os sistemas reais em soluções computacionais que promovam a eficácia dos negócios das organizações. Essa tarefa é que torna um sistema "melhor" do que outro do ponto de vista do desempenho e da sua expansão e atualização.

Nosso conteúdo pode ser considerado uma verdadeira carga de experiência profissional que ajudará o técnico em Informática a entender todas as especificações propostas pelos analistas e/ou a criar seus próprios sistemas.

Vamos examinar algumas questões práticas de projeto de *software* que servirão como regras que você poderá aplicar em seus futuros projetos.

Aproveite e faça todos os exercícios e participe das discussões nos fóruns.

Bom estudo!

Prof. Luiz Egidio Costa Cunha

Apresentação da disciplina

Nossa disciplina estuda técnicas para o projeto e o desenvolvimento de sistemas e está dividida em duas partes: Análise e Projeto, que, atualmente, são tratados de forma conjunta pelos profissionais da área.

Em Análise aprenderemos como planejar o *software* que pretendemos construir ou programar, a fim de que nossos usuários tenham suas necessidades atendidas. Aprenderemos como usar diagramas e símbolos para mostrar o que estamos imaginando de uma forma conceitual, sem muita preocupação com a aderência de nossas ideias à realidade computacional que enfrentaremos na hora do desenvolvimento dos programas e sistemas.

Em Projeto estudaremos exatamente o elo que faltava entre a fase de Análise e a decodificação dos programas. Conheceremos técnicas que nos ajudarão a entender como é possível transformar nossos conceitos (diagramas, métodos, objetos, etc.) em segmentos de *software* que, efetivamente, "rodam" nos computadores.

Para isso estudaremos algumas ferramentas como a UML e seus diagramas, e depois os padrões de construção de classes que são úteis na maioria dos projetos de *software*. Esse conjunto de técnicas e padrões, quando adotado no desenvolvimento dos sistemas, oferece ao projeto final características de responsabilidades e portabilidade que são fundamentais para seu sucesso.

A bibliografia adotada é exatamente a mesma usada por profissionais que atuam no mercado de trabalho. Estudaremos Larman (2007) e o livro do "grupo dos quatro", que é dos autores: Gamma, Helm, Johnson e Vlissides (2000).

É importante a participação ativa de todos nas atividades propostas em nossa disciplina, para que os exercícios discutidos possam servir de base para a prática profissional que todos terão quando concluírem o curso.

Siga em frente!

Prof. Luiz Egidio Costa Cunha

Projeto instrucional

Disciplina: Análise de Sistemas (carga horária: 60 h).

Ementa: Teoria geral dos Sistemas. Modelagem de dados. Metodologias para o desenvolvimento de sistemas. Ferramentas para análise e projeto de sistemas.

AULA	OBJETIVOS DE APRENDIZAGEM	MATERIAIS	CARGA HORÁRIA (horas)
1. Elementos da Teoria Geral dos Sistemas	Conhecer os principais conceitos da TGS. Relacionar os conceitos da TGS com a Computação.	Caderno e Ambiente Virtual de Ensino-Aprendizagem. http://pt.wikipedia.org/wiki/Teoria_geral_de_sistemas	5
2. O processo de desenvolvimento de *software*	Conhecer o histórico do processo de desenvolvimento de *software*. Conhecer os paradigmas de desenvolvimento de *software* mais utilizados. Diferenciar as metodologias mais comuns. Conhecer o estado da arte em metodologias de desenvolvimento de *software*.	Caderno e Ambiente Virtual de Ensino-Aprendizagem. http://www.devmedia.com.br/post-1903-Metodologia-de--desenvolvimento-de-Software.html http://www.andreygomes.com/index.php?option=com_content&view=article&id=1:metodologias-de-desenvolvimento-de-software&catid=1:metodologias&Itemid=2 http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/intro/processo.htm http://pt.wikipedia.org/wiki/Modelos_ciclo_de_vida http://pt.wikipedia.org/wiki/Scrum http://pt.wikipedia.org/wiki/Extreme_programming http://pt.wikipedia.org/wiki/RUP	5

continua

AULA	OBJETIVOS DE APRENDIZAGEM	MATERIAIS	CARGA HORÁRIA (horas)
3. Os requisitos do sistema	Conhecer o que são os requisitos de um sistema. Conhecer técnicas de elicitação de requisitos. Conhecer os elementos do Diagrama de Caso de Uso. Usar o Diagrama de Caso de Uso para especificação das principais funções do sistema.	Caderno e Ambiente Virtual de Ensino-Aprendizagem. http://pt.wikipedia.org/wiki/Engenharia_de_requisitos http://www.youtube.com/watch?v=t0so6Nagjns	10
4. Descrição do comportamento do sistema	Conhecer os elementos dos diagramas de interação. Usar os diagramas de interação da UML para especificação do comportamento do sistema.	Caderno e Ambiente Virtual de Ensino-Aprendizagem. http://www.macoratti.net/vb_uml2.htm http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/uml/diagramas/interacao/sequencia.htm	10
5. Criação de um modelo do sistema	Usar o Modelo de Domínio para delinear o escopo do sistema a ser desenvolvido.	Caderno e Ambiente Virtual de Ensino-Aprendizagem. http://www.dsc.ufcg.edu.br/~jacques/cursos/apoo/html/anal1/anal1.htm http://www.macoratti.net/net_uml2.htm	5
6. Arquitetura do sistema	Conhecer as principais arquiteturas possíveis para um sistema. Escolher entre as arquiteturas a melhor a ser implementada durante a fase de projeto de um sistema.	Caderno e Ambiente Virtual de Ensino-Aprendizagem. http://pt.wikipedia.org/wiki/N_camadas http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/arqu/camadas.html	5
7. Diagrama de classes	Conhecer os elementos do Diagrama de Classes. Usar o Diagrama de Classes para a modelagem dos dados de um sistema.	Caderno e Ambiente Virtual de Ensino-Aprendizagem. http://www.macoratti.net/vb_uml2.htm http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/uml/diagramas/classes/classes1.htm	10

continua

AULA	OBJETIVOS DE APRENDIZAGEM	MATERIAIS	CARGA HORÁRIA (horas)
8. Projeto de objetos	Conhecer os padrões de projeto GRASP. Entender a aplicação dos padrões de acordo com as questões de programação existentes nos sistemas.	Caderno e Ambiente Virtual de Ensino-Aprendizagem. http://pt.wikipedia.org/wiki/Padr%C3%A3o_de_projeto_de_software http://www.macoratti.net/vb_pd1.htm http://codeigniterbrasil.com/passos-iniciais/padroes-de-projeto-ou-design-patterns-o-que-sao-para-que-servem-e-qual-sua-implicacao-de-uso/	10
			conclusão

Aula 1 – Elementos da Teoria Geral dos Sistemas

Objetivos

Conhecer os principais conceitos da TGS.

Relacionar os conceitos da TGS com a Computação.

1.1 Histórico da Teoria Geral dos Sistemas

A Teoria Geral dos Sistemas (TGS) foi concebida em 1937 por Ludwig von Bertalanfy (2008). Ele era um biólogo que procurava entender o comportamento dos animais em suas sociedades e quais as regras os regiam para que mantivessem suas características como grupo. Bertalanfy, considerado o fundador da TGS, após divulgar seus estudos ao longo dos anos de 1945 a 1956, finalmente escreveu seu livro, intitulado: *General System Theory*, em 1968.

Martinelli e Ventura (2006) perceberam que a ciência moderna estava cada vez mais fragmentada em múltiplos compartimentos e especializações oriundas do próprio desenvolvimento científico e de sua crescente complexidade. Perceberam, também, por outro lado, que havia certa semelhança entre os princípios de diversas ciências, sendo assim possível haver um elo entre esses conhecimentos espalhados nas mais diversas áreas de conhecimento.

Outro autor importante nos estudos da TGS foi Kenneth Boulding. Economista e estudioso dos sistemas, Kenneth escreveu um artigo que descrevia a natureza geral da teoria dos sistemas, seu objetivo e importância para o estudo dos fenômenos (MARTINELLI; VENTURA, 2006, p. 9).

Até os dias atuais a TGS é estudada por diversos profissionais como administradores, economistas, biólogos, engenheiros e analistas de sistemas. Apesar dessa teoria ter sido proposta, para alcançar todos os sistemas existentes (sistemas humanos, físicos, cósmicos, computacionais, organizacionais, etc.), a área de conhecimento que mais tem explorado os conceitos da TGS é a Computação.

1.1.1 Conceitos de sistema

O conceito de sistema evoluiu desde que foi proposto na TGS. Vários autores contribuíram para que o entendimento do que é um sistema pudesse ficar claro para todos. Um dos conceitos que possui grande aceitação e aplicação para a nossa área de Computação é definido por Peter Schoderbek (SCHODERBEK et al., 1990 apud MARTINELLI; VENTURA, 2006, p. 6), segundo o qual:

A-Z

sistema
É o conjunto de objetos, com relações entre os objetos e os atributos relacionados com cada um deles e com o ambiente, de maneira a formar um todo". (MARTINELLI; VENTURA, 2006, p. 6).

a) os objetos são os elementos que compõem o **sistema**; os relacionamentos são as fronteiras que ligam os objetos;

b) os atributos são as características tanto dos objetos como dos relacionamentos; o ambiente é o que está fora do sistema, ou seja, não participa do sistema, porém, ele está inserido nesse espaço delimitado ou não.

A Figura 1.1 mostra graficamente o conceito apresentado.

Assista ao vídeo intitulado "Teoria Geral dos Sistemas", em http://www.youtube.com/watch?v=d_c8xvHtdHo para melhorar seus conhecimentos sobre a TGS. Em seguida, cite dois exemplos de sistemas abertos, de acordo com o que foi apresentado no vídeo. Mande para seu tutor esses exemplos e, para cada um deles, escreva uma breve justificativa de sua escolha.

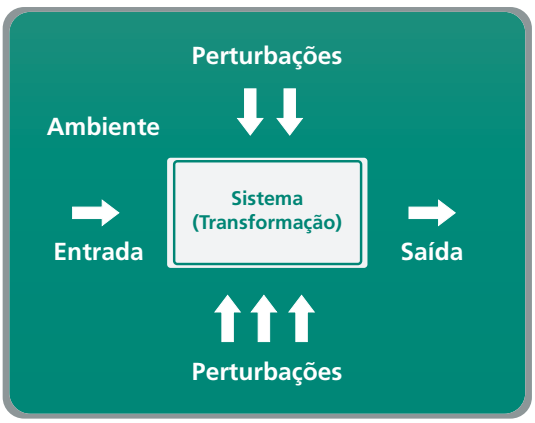


Figura 1.1: Diagrama de sistemas
Fonte: Elaborada pelo autor

1.1.2 Abordagem sistêmica e abordagem analítica

Um sistema pode ser abordado de duas formas clássicas: analítica ou sistemicamente. Ambas as abordagens são muito usuais dependendo do que se quer ao estudar um determinado sistema.

A-Z

"complexidade
Pode ser compreendida como o número de elementos que fazem parte do sistema, seus atributos, suas interações e o seu grau de organização." (MARTINELLI; VENTURA, 2006, p. 4).

Antes de aprendermos o que são essas abordagens, é importante entendermos a definição da **complexidade** de um sistema.

Dessa forma, podemos entender que um sistema complexo é aquele que possui vários elementos com vários atributos e interações entre eles. Podemos concluir ainda que quanto maior o número de elementos de um sistema, maior será sua complexidade.

Não podemos confundir "complexidade" de um sistema com "dificuldade" de se entender ou de se explicar um sistema. A complexidade está associada com a quantidade de partes e suas relações, e não com o desconhecimento ou o não entendimento de como essas partes são compostas, ou como elas interagem entre si.

Agora que já sabemos o que são sistemas complexos, podemos estudar o que são as abordagens sistêmicas e analíticas.

A **abordagem sistêmica** é aquela na qual se "olha" para o sistema de uma forma geral e abrangente. Nela não se observam partes ou as relações entre algumas partes específicas. Tanto o "olhar" quanto a descrição de sistemas de uma forma sistêmica se dão pela de visão de todo. Procura-se identificar as bordas do sistema e o que está dentro e fora dessa borda (limite do sistema).

A **abordagem analítica**, por sua vez, preocupa-se com o detalhamento das partes que compõem um sistema. Nessa abordagem o "olhar" se volta para cada parte e cada relacionamento presentes no interior do sistema. As partes e seus relacionamentos ficam expostos ao observador que consegue defini-los e estudá-los durante o processo de análise do sistema.

Tanto a abordagem sistêmica quanto a analítica são importantes no processo de estudo ou de análise de um sistema. Normalmente, num primeiro contato com um sistema (fase de conhecimento), adota-se a abordagem sistêmica para que se tenha um conhecimento claro do que é o sistema e quais suas relações com o ambiente no qual ele está inserido. Nos contatos seguintes, adota-se uma abordagem mais analítica para que se conheçam com facilidade todas as partes que tem o sistema e como essas partes interagem entre si.

O Quadro 1.1 apresenta uma comparação entre as duas formas de abordagens.

Quadro 1.1: Comparação entre abordagem analítica e sistêmica

ASPECTOS	ABORDAGEM ANALÍTICA	ABORDAGEM SISTÊMICA
Ênfase	Nas partes	No todo
Tipo	Relativamente fechado	Relativamente aberto
Ambiente	Não definido	Um ou mais
Hierarquia	Poucas	Possivelmente muitas
Estado	Estável	Adaptativo, busca novo equilíbrio

Fonte: Schoderbek apud Martinelli e Ventura (2006)

1.1.3 Considerações básicas sobre sistemas

A abordagem sistêmica, estudada na seção anterior, nos leva a considerar que é importante conhecermos algumas características dos sistemas. Essas características nos ajudam a compreender melhor o tipo de sistema que estamos estudando. No caso da Análise e Projeto de Sistemas, precisamos conhecer bem cada sistema no qual trabalhamos para que fique fácil criarmos soluções computacionais para eles. Muitas vezes os problemas informados por nossos usuários ficam mais bem entendidos pelos programadores quando conseguimos identificar essas características:

a) o **objetivo central do sistema** e as **respectivas medidas de rendimento**;

b) o **ambiente do sistema**;

c) os **recursos do sistema**;

d) a **administração do sistema**.

Os **objetivos** significam aquilo que deve ser alcançado pelo sistema, ou seja, o motivo pelo qual um sistema existe e as **medidas de rendimento** indicam o quanto ele está (ou não) alcançando seus objetivos. Esse rendimento pode ser medido de várias maneiras, por exemplo, em um sistema de controle de estoque que tenha como objetivo principal não deixar faltar produtos em um almoxarifado, a quantidade de produtos que não ficaram faltando nas prateleiras pode ser considerada uma medida de rendimento. Em outras palavras, se faltar produtos nas prateleiras por erro nos relatórios do sistema, pode-se concluir que o programa não atendeu a seu objetivo, que era garantir sempre a disponibilidade de produtos.

O **ambiente do sistema** está associado com aquilo que está ao redor dele, ou seja, com todos os outros sistemas ou "coisas" que se encontram externamente próximos dele.

Os objetivos precisam de **recursos** que sejam executados. No caso da Informática, os principais recursos que garantem o alcance dos objetivos são aqueles associados com a Computação, por exemplo: recursos de *hardware* (processador, memória, *clock*, etc.), recursos de *software* (sistema operacional, banco de dados, compiladores, etc.), recursos humanos (analistas, programadores, técnicos, etc.), entre outros que podem ser associados aos sistemas de informação.

A característica **administração do sistema** preocupa-se com as funções de planejamento e de controle associadas ao sistema. Sem um correto planejamento que contemple todas as etapas de desenvolvimento de um sistema não é possível um trabalho de programação de soluções que atendam os nossos usuários. É importante, também, que os administradores acompanhem o trabalho com os sistemas dando retorno aos interessados sobre todos os passos executados e aqueles que ainda não foram completados para a finalização do trabalho.

Podemos conhecer melhor a relação entre os sistemas de informação e a TGS no vídeo intitulado "Teoria Geral dos Sistemas e Tipologia dos Sistemas de Informação", disponível em http://standishgroup.com/newsroom/chaos_manifesto_2011.php. Cite dois exemplos de sistemas de informação que tenham uma relação importante entre eles. Escreva um pequeno texto com os nomes desses sistemas e qual a relação existente entre eles. Envie seu texto para seu tutor a distância.

Resumo

Chegamos ao final de nossa primeira aula e vimos que quanto mais conhecemos as características dos sistemas, mais fácil será prepararmos uma solução computacional que o crie ou o preserve. A TGS também é utilizada por outras áreas de conhecimento, mas cada vez mais a área de Informática tem sido beneficiada com os conceitos propostos por Ludwig Von Bertalanfy. Quando conhecemos bem esses conceitos antes de começarmos a projetar nossos sistemas e, consequentemente, aplicar esses conceitos, garantimos o sucesso de nossos programas, pois eles serão regidos pelos mesmos conceitos. Além das questões ligadas à Informática, a TGS também nos ajuda nas questões sociais ligadas aos nossos sistemas, ou seja, podemos usar os mesmos conceitos para os grupos de usuários que trabalham ligados ao nosso *software*.

Atividades de aprendizagem

1. Considere o sistema Corpo Humano. Explique qual o objetivo geral desse sistema e cite uma medida de rendimento que pode ser aplicada a ele.

2. Descreva o sistema Família de acordo com uma abordagem sistêmica utilizando um texto de, no máximo, três linhas.

3. Descreva o mesmo sistema da questão anterior, porém, agora, na abordagem analítica. Seu texto deve conter no mínimo dez linhas.

4. Usando a Figura 1.1 e o exemplo abaixo como referências, desenhe os sistemas propostos nos seguintes itens:

Exemplo:

a) sistema de Matrícula de Estudantes;

b) sistema de Ar Refrigerado;

c) sistema de Fábrica de Sucos.

Registre suas respostas num arquivo e poste-o no AVEA.

Aula 2 – O processo de desenvolvimento de *software*

Objetivos

Conhecer o histórico do processo de desenvolvimento de *software*.

Conhecer os paradigmas de desenvolvimento de *software* mais utilizados.

Diferenciar as metodologias mais comuns.

Conhecer o estado da arte com metodologias de desenvolvimento de *software*.

2.1 O desenvolvimento de *software*

Os principais motivos que levam as organizações a desenvolverem sistemas são: o aumento da qualidade dos produtos e serviços, através da automatização das rotinas; a redução de custos; o aumento da vantagem competitiva sobre os concorrentes; o aumento do nível de serviço, do desempenho dos recursos humanos; a melhoria do processo de tomada de decisões pela administração da empresa.

Em nossa disciplina estudaremos um método para desenvolver sistemas. Muitas vezes queremos programar diretamente nossos sistemas em vez de projetá-los anteriormente. Porém, essa não é uma boa prática.

Estudos feitos nos EUA para se medir a produtividade no desenvolvimento de sistemas (STANDISH GROUP, 2011) revelaram que:

a) **30% a 40%** dos projetos são **cancelados**;

b) **50%** dos projetos custam mais que o **dobro do custo** inicialmente projetado;

c) **15 a 20%** dos projetos terminam **dentro do prazo e dentro do orçamento.**

Uma importante empresa que faz estudos de produtividade de desenvolvimento de *software* chama-se Standish Group. Veja em http://standishgroup.com/newsroom/chaos_manifesto_2011.php um resumo do *Chaos Manifesto 2011*. Escreva um pequeno texto informando a respeito do manifesto (documento) e envie para seu tutor a distância.

Os principais problemas encontrados foram:

a) falta de qualidade do produto final;

b) não cumprimento dos prazos;

c) não cumprimento dos custos.

A-Z

Processo
É um conjunto de atividades bem definidas, que são executadas em uma determinada ordem.

A partir desses resultados surgiram estudos de **processos** e técnicas para melhoraria da qualidade e confiabilidade dos produtos, e também a produtividade dos desenvolvedores.

Entre os resultados temos:

a) linguagens de programação mais produtivas;

b) metodologias de desenvolvimento de *software*;

c) ferramentas computacionais para o desenvolvimento e gerenciamento de projetos de *software*.

A-Z

Processo de desenvolvimento de *Software*
É um conjunto de atividades (práticas e técnicas) rigorosas, sistemáticas e controláveis que permitem construir sistemas informatizados em um prazo e orçamento finitos.em uma determinada ordem.

Dessa forma a melhor maneira de se aumentar a qualidade do *software* é definindo um **processo** eficaz **de desenvolvimento** que contemple todas as fases necessárias para sua produção.

Outros requisitos importantes devem ser atendidos:

a) **flexibilidade**: os sistemas devem ser facilmente alteráveis para garantir a sua evolução;

b) **confiabilidade**: o número de defeitos deve ser o menor possível;

c) **desempenho**: o sistema deve ter um desempenho razoável;

d) **facilidade de uso**: o sistema deve ser de fácil utilização.

Todo o processo de desenvolvimento de *software* é definido em um conjunto de disciplinas chamado Engenharia de *Software* que o IEEE (**http://www.ieee.org**) define como: aplicação de um processo sistemático, disciplinado e quantificado ao desenvolvimento, à operação e à manutenção de *software*.

2.2 *Unified Modelling Language* (UML)

O primeiro passo para a construção de um *software* é analisar o problema a ser resolvido, entender o sistema que ele representa e, em seguida, criar um modelo que represente sua solução. Para a criação desse modelo utilizaremos uma notação ou linguagem de modelagem chamada **UML**.

A-Z

UML ou *Unified Modelling Language*
É uma Linguagem de Modelagem Unificada: linguagem gráfica que descreve os principais elementos de sistemas de *software*. A esses elementos damos o nome de *artefatos*.

2.3 Modelos de desenvolvimento de *software*

Antes de prosseguirmos em nosso estudo, é importante diferenciarmos **linguagem de modelagem** de **metodologia de desenvolvimento de sistemas**. Enquanto uma metodologia define rigidamente todos os passos para a construção do *software*, desde a análise do problema até a entrega final do sistema, fazendo uso de regras e procedimentos para a execução de cada etapa, a linguagem de modelagem preocupa-se apenas com a definição de elementos que simbolizem o sistema a ser criado. Dessa forma, a linguagem de modelagem apresenta graficamente um modelo que representa o sistema em questão, sem a preocupação de definição de passos formais para se chegar a esse modelo. É importante ressaltar que uma mesma linguagem de modelagem pode ser usada por várias metodologias de desenvolvimento de sistemas; basta que os criadores dessas metodologias a escolham para modelar os elementos de seus sistemas.

Para conhecer melhor a origem da UML, visite o *site*: http://www.omg.org.

Isso é muito importante, pois:

a) uma metodologia que funciona para a empresa X pode não funcionar para a empresa Y;

b) os níveis de qualidade podem variar para cada tipo de projeto, o que se reflete na metodologia;

c) a documentação pode variar para cada tipo de projeto, o que também se reflete na metodologia.

Assista a uma introdução à UML que ajuda a reforçar os conceitos estudados nesta seção no vídeo intitulado "Introdução à UML - 1º Parte", disponível em http://www.youtube.com/watch?v=hfN6n5fJfLc&feature=related.

Em seguida, escreva um pequeno texto sobre os pontos abordados no vídeo que mais chamaram sua atenção sobre a UML e envie para seu tutor a distância.

Além da UML, outras linguagens de modelagem são utilizadas pelos desenvolvedores de *software*, entre elas, podemos citar a ***Modelica***, e a *Java Modelling Language* (JML). Porém, cada vez mais a UML se firma como a linguagem mais utilizada pelos analistas de sistemas. A UML é simplesmente uma linguagem, isto é, uma notação. Ela não diz como desenvolver *software* e sim como criar um modelo desse *software*.

Podemos dizer, então, que a UML é uma linguagem de modelagem e não uma metodologia de desenvolvimento! Conhecer a notação UML não significa utilizar uma metodologia X ou Y. Espera-se que a UML possa ser utilizada independentemente da metodologia de desenvolvimento de sistema!

Independentemente da linguagem de modelagem utilizada para desenvolver um sistema, as metodologias normalmente obedecem a um dos modelos a seguir:

d) modelo em cascata

O modelo em cascata era o mais utilizado até a década de 1980. Nesse modelo cada etapa deve ser completada para que a seguinte possa ser realizada. Esse processo, mostrado da Figura 2.1, é simples e fácil de gerenciar e consiste em dividir a tarefa em tarefas menores e autocontidas.

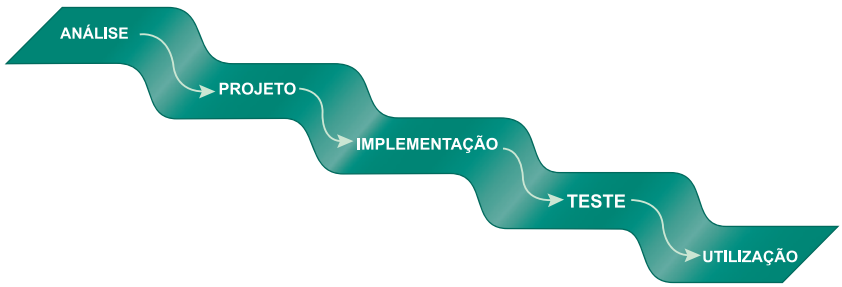


Figura 2.1: Modelo em cascata
Fonte: Serafini e Cunha (2010, p. 15)

Quando esse modelo foi proposto, no início do desenvolvimento dos sistemas, acreditava-se que ao se encerrar uma etapa do modelo, todas as informações necessárias sobre aquela fase já estavam capturadas. Por exemplo: após a entrevista com o usuário sobre as necessidades de seus sistemas, tudo que se precisaria saber para resolver seu problema já estava dito. Assim, os desenvolvedores poderiam se concentrar na fase seguinte – Análise – para darem continuidade ao processo de construção do *software*. O mesmo acontecia com as etapas seguintes. Ao longo do tempo, verificou-se que, muitas vezes, coisas importantes sobre etapas anteriores são descobertas apenas no final do processo de desenvolvimento.

Os principais problemas desse modelo são:

a) os sistemas devem ser totalmente entendidos e analisados, antes que possam ser projetados e implementados. À medida que a complexidade aumenta, mais difícil se torna o seu desenvolvimento e gerenciamento;

b) os riscos são protelados para as etapas finais. Os problemas só aparecem nas etapas finais do projeto e o custo para correção é muito alto;

c) em grandes projetos, cada etapa consome grande quantidade de tempo e recursos.

Como a fase de análise é feita somente no início do projeto, o risco de não se compreenderem os requisitos do cliente é bastante alto. Mesmo seguindo um procedimento rígido de captura de requisitos, as chances de mudanças são altas à medida que o desenvolvedor conhece cada vez mais o sistema e seus requisitos. Para projetos pequenos, o modelo em cascata pode funcionar muito bem.

É importante lembrar que consideramos projetos pequenos aqueles que possuem pouca complexidade e poucos artefatos. Dessa forma, o uso do modelo em cascata não compromete a construção do sistema por não deixar de contemplar tudo o que é necessário em cada uma de suas etapas, já que são simples.

a) modelo espiral

Nesse modelo o problema relatado pelo cliente é abordado em ciclos. Um ciclo completo é chamado de ciclo de vida e compreende as etapas de **análise**, **projeto**, **implementação** e **testes**.

Ao final de cada ciclo, temos a liberação de uma versão aprimorada do *software*, ou seja, uma versão mais completa (e mais correta) do *software* que o cliente necessita para resolver seu problema.

Esse modelo de desenvolvimento (Figura 2.2) é muito vantajoso, pois o grupo de desenvolvimento será capaz de trabalhar em todo um ciclo de vida do projeto (análise, projeto, implementação e testes) e o cliente vai fornecer "*feedback*" bem cedo para a equipe de desenvolvimento, tornando a detecção de problemas mais fácil e rápida.

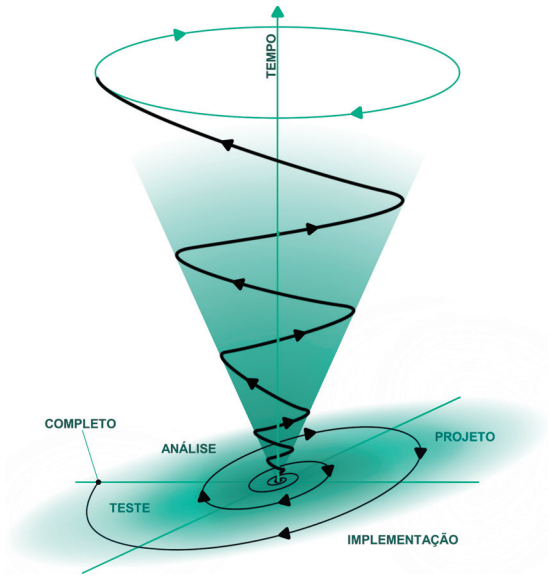


Figura 2.2: Modelo espiral
Fonte: Serafini e Cunha (2010, p. 16)

Outra vantagem importante é que qualquer mudança no sistema pode ser incorporada ao ciclo seguinte.

Mas esse modelo não tem somente vantagens, ele também apresenta algumas desvantagens, que podemos citar:

a) o processo é mais difícil de gerenciar. O modelo espiral não se "encaixa" bem com as ferramentas de gerência de projetos tradicionais, tais como gráfico de Gantt, PERT, etc.;

b) pode ocorrer precipitação e entrega de código, sem que seja feita análise completa do alcance dos seus objetivos.

Normalmente o cliente fica satisfeito ao ver que as versões que são geradas de seu *software* estão ficando conforme sua expectativa. Isso mostra para os desenvolvedores que estão no caminho certo do desenvolvimento do sistema.

c) modelo iterativo e incremental

Esse modelo é um aprimoramento do modelo espiral e incorpora mais formalismos e rigor.

O modelo é dividido em quatro fases:

a) concepção;

b) elaboração;

c) construção;

d) transição.

Essas fases são realizadas em sequência e não devem ser confundidas com as etapas do modelo em cascata.

Vamos estudar cada uma dessas fases.

a) Concepção

Nessa fase estabelecemos o escopo do projeto e definimos uma visão geral dele. Em pequenos projetos, ela pode ser realizada durante um cafezinho com o cliente!

Nessa fase devemos:

- construir um documento, descrevendo a visão geral do sistema;
- fazer uma exploração inicial dos requisitos do cliente;
- construir um glossário inicial dos termos utilizados pelo cliente;
- fazer uma análise do negócio (viabilidade, previsão de investimentos, etc.);
- fazer uma análise de risco preliminar;
- traçar um plano de projeto preliminar.

b) Elaboração

Nessa fase analisamos o problema, acrescentamos mais detalhes ao plano de projeto preliminar e procuramos eliminar ou minimizar os riscos do projeto.

No final da fase de elaboração teremos uma visão geral e um melhor entendimento do problema, devido ao seu detalhamento.

Na UML temos dois modelos que ajudam nesse estágio:

- **Modelo de caso de uso**: que ajuda a entender e capturar os requisitos do cliente;

- **Modelo conceitual** (diagrama de classes): que mostra os principais conceitos do sistema, os quais o cliente entende.

c) Construção

Nessa fase ocorre a construção de fato do *software*. Essa fase do projeto não é executada de modo linear, mas na forma do modelo espiral, fazendo uma série de iterações. Em cada uma dessas iterações utilizamos o modelo em cascata. Tornando cada iteração mais curta possível, evitamos os problemas do modelo em cascata (Figura 2.3).

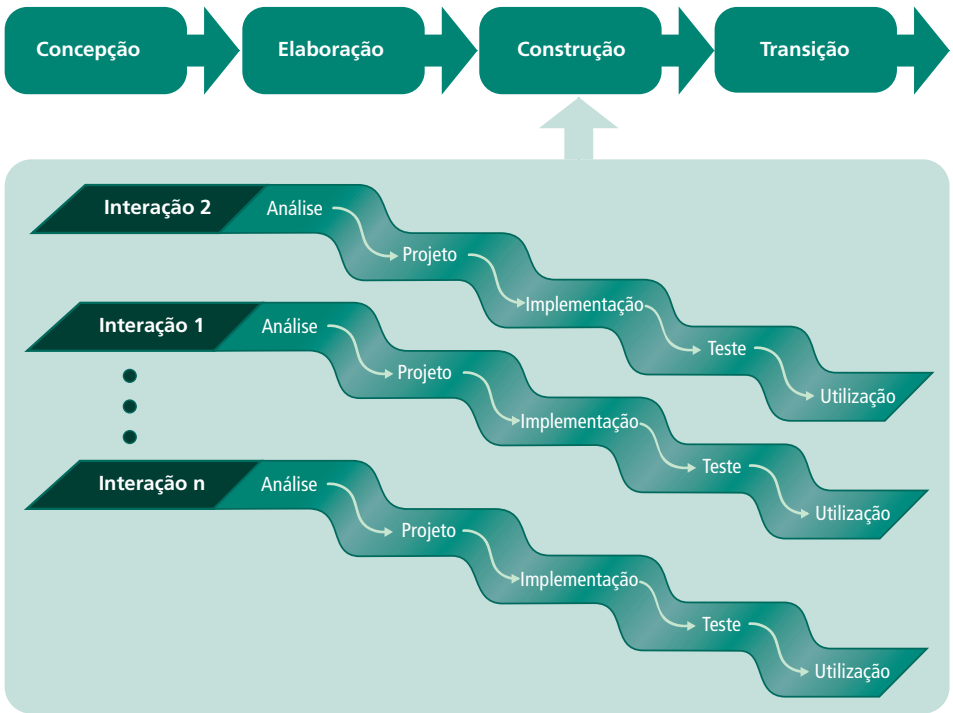


Figura 2.3: Etapas do processo iterativo
Fonte: Serafini e Cunha (2010, p. 18)

Ao término de cada iteração temos o sistema um pouco mais desenvolvido. Após um conjunto de iterações, podemos ter a liberação de uma nova versão preliminar (pré-alfa, alfa, beta, etc.) do *software*.

d) Transição

A fase final está relacionada à colocação do produto final para o cliente.

As atividades típicas dessa fase são:

- liberação de versões Beta para alguns usuários da comunidade;
- teste de validação ou execução paralela;
- conversão de dados de sistemas antigos;
- treinamento de usuários;
- publicidade, distribuição e vendas.

A versão Beta é a versão de um produto que ainda se encontra em fase de desenvolvimento e testes. No entanto, esses produtos muitas vezes são popularizados bem antes de sair a versão final.

A fase de transição não deve ser confundida com a fase de teste tradicional. No início da fase de transição deve existir um produto completo, testado e operacional para os usuários (os clientes).

Resumo

Encerramos assim nossa segunda aula e vimos que muita informação nova foi aprendida. Esse conhecimento é necessário para que continuemos nosso estudo sobre a análise e o projeto de sistemas que atendam, cada vez mais, às necessidades de nossos clientes.

Vale lembrar que, apesar dos problemas naturais para se desenvolver sistemas, é muito importante que sempre trabalhemos na construção de nosso *software* usando ferramentas adequadas. As metodologias e as linguagens de modelagem são fundamentais para esse trabalho. Vimos que a UML é a mais importante linguagem de modelagem usada atualmente e pode ser

trabalhada dentro de várias metodologias. Vimos ainda que as metodologias podem adotar modelos que nos ajudam a alcançar mais qualidade em nossos produtos. Os modelos mais utilizados são aqueles que priorizam a satisfação final do cliente, como o Iterativo e Incremental.

Agora que já tivemos uma visão geral do processo de desenvolvimento de *software* e conhecemos as etapas necessárias para sua construção vamos, na próxima aula, aprender como trabalhar na primeira fase desse processo: o levantamento dos requisitos de um sistema. Essa fase é comum a todas as metodologias e se aplica a qualquer modelo de desenvolvimento de sistemas.

Atividades de aprendizagem

1. Quais as principais diferenças entre linguagens de modelagem e metodologias de desenvolvimento de sistemas?

2. Cite duas desvantagens do modelo em cascata de desenvolvimento de sistemas.

3. Cite dois exemplos de sistemas complexos que são mais bem desenvolvidos se usarmos o Modelo Iterativo e Incremental.

4. Quais problemas encontrados em projetos de *software* que causam insatisfação do usuário no momento da entrega do produto final.

5. Cite dois exemplos de metodologias ágeis de desenvolvimento de sistemas.

Aula 3 – Os requisitos do sistema

Objetivos

Conhecer o que são os requisitos de um sistema.

Conhecer técnicas de elicitação de requisitos.

Usar o Diagrama de Caso de Uso para especificação das principais funções do sistema.

3.1 Fases de concepção

O início da construção de um sistema acontece na sua **fase de concepção**.

A-Z

Fase de concepção
É a etapa do projeto de um *software* em que um sistema é concebido. Nessa fase, após as necessidades dos usuários já terem sido levantadas, projeta-se o produto (*software*) com um escopo definido e se estuda um plano de negócio para seu desenvolvimento e implantação.

Algumas etapas da fase de concepção são:

a) definir uma visão para o produto, ou seja, os objetivos atuais e futuros que os usuários pretendem alcançar com o seu uso;

b) produzir um plano de negócio para o desenvolvimento, produção, implantação e teste do *software*;

c) definir do escopo do projeto; ou seja, seus limites de interface com os demais sistemas existentes na empresa;

d) estimar o custo total aproximado do projeto para que os usuários se capitalizem e se planejem para o projeto.

O tamanho dessa fase depende do tipo de projeto. Por exemplo: um projeto de comércio eletrônico (*E-commerce*) que pretende atingir o mercado rapidamente e as atividades desenvolvidas durante a concepção podem se reduzir a uma definição da visão geral, criando um plano de negócio para obter financiamento. Quando se trata de um projeto de defesa antiaérea, certamente a fase vai requer muito mais trabalho, ou seja, uma análise de requisitos mais elaborada, definições de projeto, estudos preliminares, etc.

É importante na **fase de concepção** termos certeza se o projeto é viável, do ponto de vista financeiro e computacional, e se ele trará algum benefício para a empresa. Caso não consigamos estar certos dessas questões, é aconselhável que o projeto seja repensado.

3.2 Requisitos do sistema

Para desenvolver um sistema, precisamos descobrir o que o nosso cliente deseja.

A-Z

Requisitos
São as capacidades e condições a que o sistema deve atender.

Nosso objetivo é escrever o documento de **requisitos** de sistemas, que deve conter de forma precisa e realista tudo o que deve ser feito, de modo a atender a uma ou mais necessidades do nosso cliente.

Esse documento também deve ser validado pelo cliente, isto é, o cliente deve ler o documento e verificar se todas as suas necessidades estão atendidas.

Para encontrar os requisitos do sistema, precisaremos conversar com nosso cliente. Esse cliente não é necessariamente uma única pessoa e pode representar uma empresa com milhares de funcionários!

Assim, é muito útil a criação de um grupo de modelagem, que deve ser composto de:

a) **investidor**: (em inglês: *Stakeholder*): cliente/financiador do sistema;

b) **usuários**: aqueles que irão utilizar diretamente o sistema ou qualquer pessoa que vá ser afetada pelo sistema;

c) **analistas** ou **desenvolvedores**: nós!

d) **facilitadores**: pessoas que conhecem bem a empresa e possuem habilidade para fazer reuniões; entendem o processo de modelagem dos requisitos e podem fazer perguntas válidas e inteligentes;

e) **escribas**: pessoas que devem escutar bem, possuir habilidade na comunicação oral e escrever bem. São eles que irão anotar os requisitos durante as reuniões e ajudar na escrita do documento de requisitos.

Essa equipe tem a função de levantar as necessidades do sistema solicitado pelo cliente e que será desenvolvido por nós. Cada elemento dessa equipe deve desempenhar seu papel de forma a garantir que sua contribuição seja efetivada para o perfeito conhecimento do que se deseja para o sistema

pretendido. Os clientes são necessários para a composição da equipe, por serem eles os que melhor conhecem os requisitos do sistema a ser projetado. Assim, eles possuem direitos e deveres que devem ser respeitados para que o sucesso seja alcançado.

3.2.1 Direitos e deveres dos clientes

Os principais direitos dos clientes são:

a) receber do desenvolvedor as explicações sobre o processo de desenvolvimento de sistemas; receber do desenvolvedor um sistema que atenda às suas necessidades nos aspectos de funcionalidade e qualidade;

b) ouvir dos desenvolvedores todas as informações técnicas sem o uso de jargões da área de Informática, e sim numa linguagem simples e compreensível.

Seus principais deveres em relação aos desenvolvedores são:

a) explicar o funcionamento da empresa onde o sistema será implantado;

b) disponibilizar tempo para fornecer as informações necessárias;

c) ser preciso e claro na descrição dos requisitos; ser capaz de priorizar os requisitos;

d) ser capaz de tomar decisões relativas ao sistema e suas funcionalidades;

e) realizar, juntamente com o desenvolvedor, revisões dos requisitos.

3.2.2 Técnicas de identificação de requisitos

Para obtermos os requisitos de um sistema, utilizamos duas técnicas muito conhecidas:

a) entrevistas;

b) *brainstorming* (em português: "tempestade cerebral"): técnica em que um grupo de pessoas discute um assunto e diz qualquer coisa relativa ao assunto que lhe venha à mente sem a preocupação da relevância do que foi dito para o assunto em questão.

Quanto mais estudamos essas técnicas, mais requisitos conseguimos levantar sobre o sistema a ser desenvolvido.

Uma boa referência para conhecermos melhor as técnicas de entrevista e *brainstorming* é o livro de Francisco Filho (2008).

Algumas recomendações são importantes para o sucesso dessas técnicas:

- Para realizar entrevistas:

a) forneça uma agenda;

b) esclareça aos participantes quais os motivos do projeto;

c) faça as perguntas críticas em primeiro lugar;

d) não assuma que você tem conhecimento do assunto;

e) identifique o que é "necessário" e o que é "desejável";

f) termine a entrevista com um sumário dos pontos abordados.

- Para realizar *brainstorms*:

a) todas as ideias são boas: as ideias não são julgadas pelo grupo;

b) todas as ideias são do grupo, ninguém é "dono" de uma ideia;

c) todas as ideias são públicas, qualquer pessoa pode expandir ou modificar uma ideia;

d) sempre que surgir uma ideia, ela deve ser imediatamente escrita no quadro-negro;

e) antes da sessão, forneça ao grupo uma cópia das regras de *brainstorming*.

3.2.3 O documento de requisitos

Este documento deve conter a especificação do sistema, ou seja, uma descrição das necessidades ou dos requisitos do sistema a ser desenvolvido.

Nele deve-se descrever:

a) uma visão geral do sistema;

b) quem são os clientes do sistema;

c) quais os objetivos;

d) quais as funções principais do sistema;

e) quais os atributos necessários do sistema.

A seguir, estudaremos os conceitos de funções e atributos do sistema.

a) Funções do sistema

As **funções do sistema** são "**o que**" se supõe que o sistema faça e devem ser identificadas, listadas e agrupadas logicamente.

Uma boa regra para verificar se uma expressão X é, de fato, uma função do sistema, colocamos X na sentença "O sistema deveria fazer X" e essa sentença deve fazer sentido.

Exemplo: Num sistema de registro de notas escolares, poderemos duvidar se "elaborar prova" é uma função do sistema. Colocamos então as palavras "elaborar prova" na frase grifada acima: "O sistema de registro de notas escolares deveria fazer a elaboração das provas". Por experiência, sabemos que as provas de uma escola são elaboradas pelos professores e não por sistemas computacionais. Logo, "elaborar prova" não é uma função desse sistema.

Agrupamos as funções em três categorias:

a) evidente: deve executar e o usuário deve saber que é executada;

b) oculta: deve executar, mas não deve ser visível para os usuários;

c) enfeite/decoração: opcional.

b) Atributos do sistema

Os **atributos do sistema** são qualidades não funcionais do sistema.

Em alguns projetos é conveniente construir um documento exclusivo para os atributos do sistema. Esse documento fica separado do documento que contém as funcionalidades do sistema.

Exemplos de atributos do sistema:

a) facilidade de uso: que tipo de interface o sistema possui com o usuário?

b) tolerância a falhas: qual a expectativa aceitável para falhas no sistema?

c) tempo de resposta: qual o tempo aceitável para a resposta do sistema?

3.2.4 Tipos de requisitos

Para melhor entendermos os requisitos, é comum separá-los em grupos ou tipos.

Os tipos de requisitos são:

a) **funcionalidade**: descreve características, capacidades, segurança, etc., do sistema;

b) **usabilidade**: descreve características relativas a fatores humanos, ajuda *on-line*, documentação;

c) **confiabilidade**: descreve necessidades do sistema quanto à frequência de falhas, recuperabilidade, etc;

d) **performance**: descreve necessidades relativas a tempos de respostas, precisão, disponibilidade, utilização de recursos, etc;

e) **suportabilidade**: descreve necessidades relativas à adaptabilidade, manutenibilidade, internacionalização, configurabilidade, etc;

f) **implementação**: descreve limitações de recursos, linguagens e ferramentas, *hardware*, etc.;

g) **interface**: descreve as restrições impostas pelo interfaceamento com sistemas externos;

h) **operação**: descreve as necessidades relativas à administração do sistema em sua configuração operacional;

i) **aspectos legais**: descrevem licenciamento, etc.

3.2.5 Recomendações para obtenção dos requisitos

Nesta seção são apresentadas recomendações úteis para a obtenção dos requisitos dos sistemas.

a) **Questões fundamentais**

As seguintes questões podem ajudar a esclarecer os requisitos de um sistema:

1. Para quem é esse sistema?

2. O que eles irão fazer com esse sistema?

3. Por que fazemos isso? Por que fazemos assim? Por que fazemos desse modo?

4. Quais as necessidades de negócio esse sistema deve suportar?

5. O que nosso cliente demanda?

6. Como são as mudanças no negócio?

7. O que nossos competidores estão fazendo? Por quê? Como podemos fazer melhor?

8. Precisamos fazer isso?

9. Se estivéssemos iniciando o negócio, faríamos dessa forma?

10. Se fomos bem-sucedidos fazendo dessa forma no passado, seremos bem-sucedidos fazendo assim no futuro?

11. Poderemos combinar várias funções?

12. Como o sistema vai afetar as funções dentro da empresa?

13. De que informações as pessoas necessitam para executar o serviço?

14. O trabalho está sendo executado onde ele faz mais sentido?

15. Existe alguma tarefa simples que pode ser automatizada?

16. As pessoas estão realizando somente atividades complexas que o sistema não pode realizar?

17. O sistema vai se pagar?

18. O sistema suporta trabalho de grupo?

19. Os usuários possuem formação para lidar com o sistema?

20. Que treinamento é necessário?

21. Quais os objetivos e as metas estratégicas da organização? O sistema suporta essas metas?

22. Como podemos fazer mais rápido?

23. Como podemos fazer mais barato?

24. Como podemos fazer melhor?

b) Dez regras para obtenção dos requisitos

Para facilitar a obtenção dos requisitos, a empresa Software Productivity Center (**http://www.spc.ca**) nos apresenta as dez regras, reproduzidas a seguir:

1. a obtenção dos requisitos tem por objetivo descobrir e entender problemas, e não encontrar soluções. Você só é capaz de realmente entender os problemas do cliente quando pode demonstrar que conhece esses requisitos;

2. os clientes costumam confundir aquilo de que precisam com o que desejam. Durante a fase de levantamento dos requisitos, por meio de questionamentos e entendimento, será possível separar o necessário do desejável;

3. documente os problemas e as necessidades do cliente e peça para verificar a sua precisão e completude. Esse é o meio de demonstrar que você entende as necessidades do cliente;

4. analise cuidadosamente os requisitos para que não haja conflito entre eles;

5. quantifique "quão bem" um usuário espera a funcionalidade do sistema como parte dos requisitos. Todos os requisitos devem ser testáveis;

6. formule os requisitos claramente, sem ambiguidades. Requisitos claros eliminam o "retrabalho" e facilitam a entrega do produto no prazo;

7. quando houver pressão para mudanças de requisitos, antes de prosseguir, envolva todos os principais interessados para fazer uma avaliação do impacto e custo da alteração;

8. procure trabalhar com requisitos "bons o suficiente". A busca de requisitos "perfeitos" pode aumentar os riscos e ocasionar atrasos;

9. para coletar e documentar os requisitos, procure pessoas que saibam "escutar, falar e escrever";

10. não dependa somente dos conhecimentos da equipe de desenvolvimento. Os requisitos são, geralmente, definidos por pessoas fora da equipe de desenvolvimento. Garanta que todos os envolvidos que fornecem informações de requisitos entendam o que é um bom requisito e participe de sua criação.

3.3 Casos de uso

Um instrumento muito útil no desenvolvimento de sistemas é o **caso de uso**.

A-Z

Caso de uso
É uma descrição detalhada de um conjunto de interações entre um usuário e o sistema.

Ator
É alguém ou alguma coisa que participa do caso de uso e é externo ao sistema.

Os **casos de uso** são escritos em um documento que serve para orientar a construção dos demais elementos do projeto.

Vamos examinar agora os casos de uso com mais detalhes. Vamos conhecer os elementos e conceitos a eles ligados.

3.3.1 Atores

Na definição de casos de uso encontramos o termo "usuário". Nos casos de uso o "usuário" é chamado de **ator**.

Temos dois tipos de atores:

a) **ator iniciador**: é ele que inicia o caso de uso, geralmente, sendo o seu ator principal. Por exemplo, se estamos fazendo um sistema bancário, o caso de uso "retirar dinheiro" terá como ator iniciador o ator cliente;

b) **ator participante**: é aquele que de alguma forma participa do caso de uso. Por exemplo, no caso de uso "comprar itens", existe um ator atendente que "atende" o ator cliente.

Um ator não precisa, necessariamente, ser uma pessoa. Um ator pode representar:

a) um grupo de pessoas. Por exemplo, clientes, estudantes, etc.;

b) uma função de uma pessoa. Por exemplo, estudante representante da turma;

c) outro sistema externo;

d) um conceito abstrato como hora, data, etc. Por exemplo, o ator data pode iniciar um caso de uso chamado "cancelar pedidos" para pedidos feitos há mais de seis meses.

Um ator geralmente participa de mais de um caso de uso e representa papéis.

Dê um nome ao ator de forma que fique claro seu papel no sistema. Os nomes devem ser condizentes com aqueles já usados na empresa. Dessa forma, nossos clientes compreenderão mais facilmente o significado desse elemento do caso de uso.

Vamos precisar encontrar todos os atores envolvidos no sistema?

A resposta a essa pergunta é SIM. E, se quiser, utilize as perguntas a seguir para auxiliá-lo nessa tarefa:

a) Qual o principal usuário do sistema?

b) Quem fornece as informações para o sistema?

c) Quem obtém as informações do sistema?

d) Quem instala o sistema?

e) Quem opera o sistema?

f) Quem desativa o sistema?

g) Que outros sistemas interagem com o sistema?

h) Algum processo acontece automaticamente em horas predeterminadas?

i) Quem fornecerá, utilizará ou removerá informações do sistema?

j) De onde o sistema obtém informações?

3.3.2 Tipos de casos de uso e sua descrição

Já sabemos que um caso de uso é um documento narrativo que descreve uma sequência de eventos de um ator (um agente externo) que usa um sistema para completar um processo.

Com relação ao nível de detalhes dessa descrição, podemos dividir os casos de uso em dois grupos:

a) **caso de uso de alto nível**: descreve um processo de forma muito breve, usualmente, em duas ou três sentenças. Contém os dados: nome do caso de uso, atores, tipo, descrição.

Exemplo de caso de uso de alto nível:

Caso de uso: Comprar itens.

Atores: Cliente, caixa.

Tipo: Primário.

Descrição: Um cliente chega a um ponto de pagamento, com vários itens que deseja comprar. O caixa registra os itens de compra e recebe o pagamento. Ao término, o cliente deixa a loja com os itens.

b) **caso de uso expandido**: descreve um processo com mais detalhes que um caso de uso de alto nível. Contém uma seção chamada *sequência típica de eventos*, que descreve os eventos passo a passo. Os casos de uso no formato expandido são muito úteis para uma melhor compreensão dos processos e requisitos, já que eles descrevem com detalhes todos os passos de uma operação do sistema.

Observe que a descrição da operação é bem simples e direta.

Um caso de uso expandido contém os seguintes dados:

1. **nome**: Identificador do caso de uso: deve ser escrito em formato de verbo + substantivo e ser suficiente para se perceber a que se refere o caso de uso;

2. **descrição geral**: curto resumo do caso de uso (um ou dois parágrafos, no máximo);

3. **precondições**: listagem das condições que se devem verificar quando se inicia o caso de uso. Não incluem *triggers*;

4. ***triggers*** (em português: "gatilhos"): eventos que ocorrem dando início ao caso de uso;

5. **cenário de sucesso principal (linha de eventos)**: descreve o curso por uma sequência de eventos numerados;

6. **percursos alternativos (extensões)**: descrição de percursos alternativos à linha de eventos principal;

7. **pós-condições**: descrição do estado do sistema após a execução do caso de uso;

8. **regras de negócio**: reservadas para informação adicional relativa à política da empresa ou restrições impostas pelo tipo de negócio;

9. **listas de variações tecnológicas**: especificações dos métodos de entrada e saída de dados e quais formatos podem ser utilizados no caso de uso;

10. **frequência de ocorrência**: frequência de ocorrência do caso de uso por dia, mês, hora, etc. (Influencia no projeto, desenvolvimento ou implantação do sistema?);

11. **notas**: informações adicionais relativamente ao caso de uso, não cobertas pelos dados anteriores;

12. **autor e data**: listagem dos autores e datas das várias versões revistas.

Um bom estudo sobre os diagramas de caso de uso e outros elementos da UML podem ser encontrados em http://www.macoratti.net/vb_uml2.htm.

Ao longo de nossa disciplina veremos vários exemplos desses dois tipos de casos de uso.

a) Identificação dos casos de uso

Os casos de uso são encontrados e descritos por meio de entrevistas e *brainstorms*. O roteiro mais indicado para a aplicação dessas técnicas é:

1. identificação de todos os atores;

2. identificação de todos os casos de uso;

3. descrição simples de cada caso de uso;

4. desenho de um diagrama de casos de uso.

Não espere encontrar todos os casos de uso e atores na primeira sessão. É natural aparecerem casos de uso e atores nas fases seguintes do desenvolvimento, isto é, durante o processo de construção do *software*.

Para ajudar nessa tarefa, as seguintes perguntas podem ser úteis:

a) Quais as principais tarefas desse papel (ou ator)?

b) O que os usuários nesse papel precisam ser capazes de realizar?

c) Por que os usuários nesse papel precisam realizar essa tarefa?

d) Quais informações os usuários nesse papel necessitam examinar, criar ou alterar?

e) Quais informações os usuários nesse papel necessitam saber pelo sistema?

f) Quais informações os usuários nesse papel precisam informar ao sistema?

Um erro comum na identificação de caso de uso é representar como casos de uso passos individuais, operações, ou transações. Um caso de uso é uma descrição completa de um processo relativamente grande, que inclui, tipicamente, muitos passos ou transações.

Exemplo de levantamento

Suponha que você estivesse levantando as informações sobre os casos de uso de um sistema acadêmico usado por funcionários da secretaria de uma escola. Para isso, você escolheu a técnica de entrevista. Quais perguntas você colocaria no roteiro dessa entrevista? Formulando apenas cinco perguntas, poderiam ser utilizadas as seguintes:

1. Quais informações sobre estudantes e professores vocês precisam guardar na secretaria desta escola?

2. Quais as principais tarefas desempenhadas pelo funcionário responsável pela matrícula de estudantes?

3. Quais as informações o diretor da escola precisa ter sobre professores e estudantes?

4. Por que a secretária escolar precisa cadastrar estudantes novatos apenas no início de cada ano letivo?

5. Quais as informações os professores precisam saber sobre suas turmas pelo sistema?

b) Granularidade dos casos de uso

Um aspecto que pode ser complicado para o iniciante é decidir a granularidade dos casos de uso, ou seja, a qual nível de detalhamento devemos descer ao definirmos uma funcionalidade.

Exemplo de levantamento:

Em um sistema de Caixa Eletrônico, temos uma funcionalidade chamada Retirar Dinheiro. Essa funcionalidade irá requerer as seguintes interações:

a) entrar cartão;

b) entrar senha;

c) selecionar quantia desejada;

d) confirmar quantia;

e) remover cartão;

f) retirar recibo.

Cada uma dessas interações deve ser um caso de uso?

A resposta é NÃO. Esse é um erro clássico na construção de casos de uso. Os casos de uso devem ser definidos para as funcionalidades do sistema e não para as interações. Dessa foram, mantemos a complexidade do caso de uso baixa.

Uma boa regra é: um caso de uso deve satisfazer a um objetivo do usuário.

Aplicando essa regra simples ao exemplo acima, podemos fazer a seguinte pergunta: Retirar Recibo (item 6) é um objetivo do usuário? Ou seja, o objetivo do usuário desse sistema é apenas Retirar Recibo? A resposta é NÃO. Então, Retirar Recibo não será um caso de uso.

Aplicando essa regra aos outros itens da lista, vemos que a resposta é NÃO para todos eles. O objetivo real do usuário é retirar dinheiro, logo, Retirar Dinheiro deve ser o caso de uso, isto é, o nome do caso de uso será Retirar Dinheiro.

c) Relação entre casos de uso e funções do sistema

As funções do sistema representam o que o sistema deve realizar. Assim, devemos ter casos de uso que implementem essas funcionalidades.

Algumas vezes uma função resultará em um único caso de uso; outras vezes, várias funcionalidades resultarão em um único caso de uso.

d) Ordem de implementação dos casos de uso

Com o desenvolvimento iterativo vamos implementar um caso de uso em cada ciclo de desenvolvimento. Com casos de uso muito complexos, podemos implementar versões simplificadas dele. Com casos de uso muito simples, podemos implementar mais de um caso de uso em um único ciclo.

Então surgem as perguntas: Que casos de uso devemos implementar primeiro? Qual ordem de implementação dos casos de uso é melhor?

Recomendamos que você implemente primeiro os casos de uso mais importantes.

Para classificar os casos de uso, considere:

a) Causa impacto significativo no projeto da arquitetura?

b) Possui quantidade significativa de informações e compreensão?

c) Possui quantidade significativa de funções de alto risco?

d) Envolve pesquisa significativa ou tecnologia nova e com riscos?

e) Contém representação de processos primários da linha de negócios?

f) Possui influência direta no aumento da receita ou na diminuição de custos?

Quanto mais respostas "SIM" você obtiver nessas perguntas, mais importante será o caso de uso em questão. Assim, podemos criar uma lista de caso de usos numa ordem decrescente de importância. Os primeiros casos de uso listados serão os primeiros a ser implementados.

e) Diagrama de casos de uso

Um diagrama de casos de uso é uma representação gráfica dos casos de uso e dos atores envolvidos no sistema em análise.

Um **diagrama de casos de uso** mostra os atores e casos de uso de forma gráfica. Seu objetivo é dar uma visão geral dos casos de uso necessários ao sistema de forma a facilitar a compreensão por todos os envolvidos na equipe de desenvolvimento do novo *software*.

Os símbolos que utilizamos são mostrados na Figura 3.1:

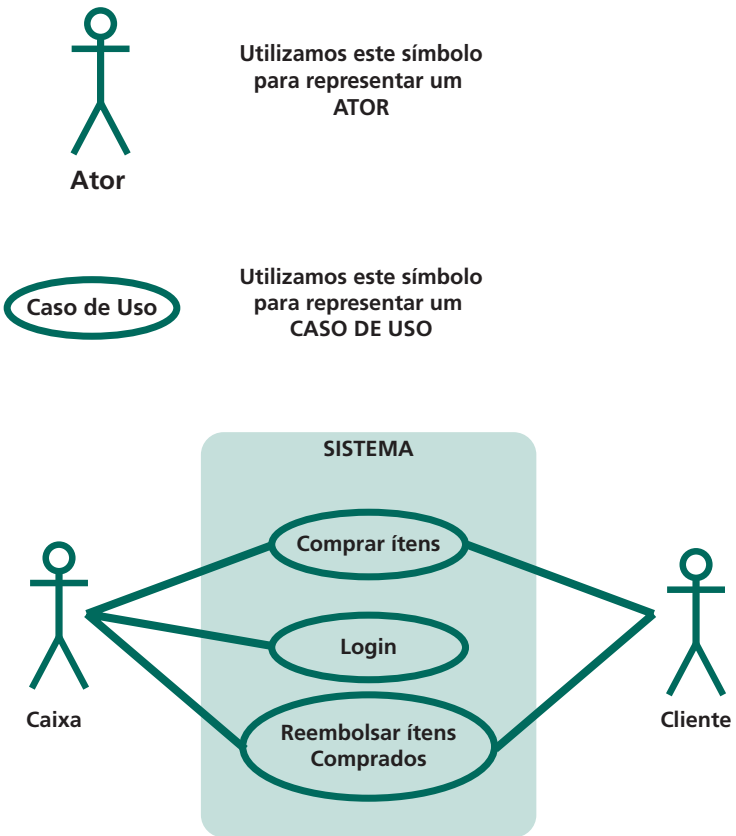


Figura 3.1: Simbologia do diagrama de casos de uso
Fonte: Serafini e Cunha (2010, p. 49)

O diagrama de casos de uso é muito interessante para dar uma visão geral do sistema, com seus atores, casos de uso e a relação entre eles.

Com esse diagrama, podemos ter uma ideia do tamanho do sistema e da sua complexidade de uma forma gráfica.

1. Considerando a Figura 3.1, descreva qual poderia ser objetivo do *software* representado pelo diagrama intitulado "Sistema".

2. Quais as maiores dificuldades encontradas por desenvolvedores durante a fase de concepção de sistemas enquanto se levantam seus requisitos com os usuários?

3. Pesquise na internet um exemplo de diagramas de caso de uso sobre um sistema de vendas. Analise o diagrama e escreva as principais funções do sistema de acordo com os casos de uso existentes nesse diagrama.

Registre suas respostas num arquivo e poste-o no AVEA.

Os passos necessários para criar o planejamento e elaboração dos sistemas são:

a) após as funções do sistema terem sido listadas, defina a fronteira do sistema, e, em seguida, identifique os atores e os casos de uso dando-lhes nomes;

b) descreva todos os casos de uso (conforme orientação da seção 3.3.2) em um formato de alto nível e defina se eles são obrigatórios ou opcionais;

c) desenhe um diagrama de casos de uso;

d) relacione os casos de uso e ilustre o relacionamento no diagrama de casos de uso;

e) escreva os casos de uso mais críticos, influentes e de maior risco para o sistema, no formato expandido;

f) defina a ordem de implementação segundo a importância de cada caso de uso (conforme orientação da seção 3.3.6).

Dessa forma, ao final da Fase de Concepção, teremos a produção do Documento de requisitos, conforme vimos na seção 3.2.3, e o Documento de casos de uso, que conterá os diagramas e as descrições de casos de uso.

No vídeo chamado "Diagramas de caso de uso", disponível em http://www.youtube.com/watch?v=fGA-eF8HCjw&feature=related, podemos ver o uso dos diagramas de caso de uso para o desenvolvimento dos sistemas. Cite três casos de uso que encontramos comumente nos sistemas de informação de secretaria escolar. Envie sua resposta para seu tutor a distância.

Resumo

Nesta aula estudamos os passos iniciais para desenvolver o projeto de um sistema. A partir do levantamento dos requisitos, conseguimos identificar elementos importantes, como casos de uso, atores e seus relacionamentos, de maneira que o diagrama gerado por eles apresente uma visão geral do sistema e sua complexidade.

Estudamos também quais as técnicas mais comumente usadas para o levantamento dos requisitos: entrevistas e *brainstorming*. Pelo seu uso conseguimos entender melhor quais as características de um sistema relatadas por seus usuários.

É importante não esquecermos que todo conhecimento gerado pelo levantamento dos requisitos precisa ser documentado, e para isso produzimos o Documento de requisitos e o Documento de casos de uso, cujos detalhamentos foram discutidos na seção anterior.

Na próxima aula vamos dar um passo à frente para a construção de um sistema. Agora que já aprendemos como conhecer as características do sistema por meio de seus requisitos, vamos aprender como desenhar um diagrama que mostrará qual o comportamento terá nosso sistema de acordo com as operações que ele realizará. O diagrama que aprenderemos a criar será o Diagrama de Sequência do Sistema (DSS).

Atividades de aprendizagem

1. Em que situações devemos usar a descrição de caso de uso de alto nível e expandido?

2. Cite um exemplo de responsabilidade do cliente/usuário em relação ao desenvolvimento de um sistema e explique sua importância.

3. Você acha que o tempo de resposta é um atributo de um sistema de manutenção de computadores? Explique sua resposta.

4. Desenhe três casos de uso de alto nível que mostrem funções de um sistema de aluguel de fitas numa videolocadora (Por exemplo: emprestar um vídeo, devolver um vídeo, pagar uma locação, solicitar reserva de vídeo, etc.). Descreva também os casos de uso desenhados.

5. Escreva uma lista dos casos de uso desenhados na questão 4 e ordene-os por seu grau de importância

Registre suas respostas num arquivo e poste-o no AVEA.

Aula 4 – Descrição do comportamento do sistema

Objetivos

Conhecer os elementos dos diagramas de interação.

Usar os diagramas de interação da UML para especificação do comportamento do sistema.

4.1 Introdução

O **comportamento de um sistema** é definido pelas operações que ele realiza. Esse comportamento demonstra situações em que o sistema pode se encontrar. Por exemplo, se considerarmos um boleto de pagamento em um sistema financeiro empresarial, podemos identificar alguns comportamentos possíveis: boleto pago, boleto não pago, boleto com pagamento atrasado, etc.

Em UML, a representação gráfica utilizada para mostrar as operações do sistema chama-se Diagrama de Sequência de Sistema (DSS).

A-Z

Comportamento do Sistema
É o conjunto de operações que o sistema realiza. Um termo muito usado para referenciar esse comportamento é *Application Programming Interface* (API) do sistema.

4.2 Diagrama de sequência do sistema (DSS)

Podemos resumir a descrição do **DSS** como a representação gráfica dos casos de uso identificados pelo sistema. É mais fácil a identificação dos casos por um diagrama que mostra também a relação desses casos com o ambiente externo ao sistema.

É um diagrama que mostra graficamente a utilização do sistema, de acordo com o caso de uso. Ele representa através de figuras o comportamento possível que o sistema poderá ter.

A-Z

Diagrama de sequência do sistema (DSS)
É um diagrama que mostra graficamente a utilização do sistema, de acordo com o caso de uso. Ele representa através de figuras o comportamento possível que o sistema poderá ter.

4.3 Construção do diagrama de sequência do sistema

Nossos DSS representam as interações entre os atores e o sistema. O sistema é considerado uma "caixa-preta" (ou seja, não conhecemos seu interior, sabemos apenas sua função): não precisamos saber "como" o sistema irá implementar a operação!

Colocamos os atores no lado esquerdo do diagrama e na parte direita colocamos uma caixa que representa o sistema.

Em seguida, acrescentamos uma linha vertical. A linha representa a passagem do tempo. O tempo corre de "cima para baixo" (Figura 4.1):

Figura 4.1: Diagrama de sequência – Passagem do Tempo
Fonte: Elaborada pelo autor

As interações entre o usuário e o sistema são representadas por linhas horizontais com setas que indicam a direção da interação. A descrição da interação é escrita na seta. Uma mensagem é representada por uma linha contínua e uma resposta é representada por uma linha pontilhada (Figura 4.2):

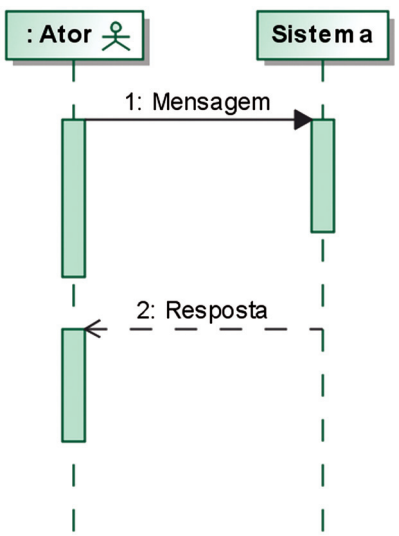


Figura 4.2: Diagrama de sequência - Mensagem/Resposta
Fonte: Elaborada pelo autor

Continuamos a acrescentar as interações (Figura 4.3).

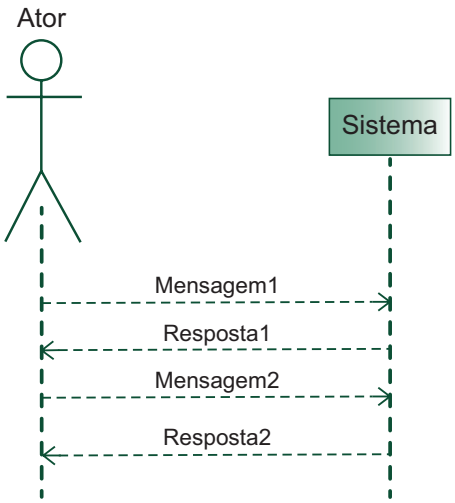


Figura 4.3: Diagrama de sequência - 2 Mensagens/Respostas
Fonte: Elaborada autor

Uma vez que o caso de uso está elaborado, é bem simples desenhar o diagrama de sequência de sistema. Basta ir examinando as interações descritas no Cenário de sucesso principal e nas Extensões.

Podemos considerar um DSS como um resumo gráfico ou uma representação gráfica de um caso de uso. Assim, é importante prestar atenção, ao colocar os nomes das operações, para que seja o mais esclarecedor possível. Devemos começar o nome com um verbo e procurar mostrar a intenção da operação. Por exemplo: entrarItem(idItem) é melhor do que escanear(idItem).

4.4 Objetivos do DSS

Os principais objetivos para construir o DSS são:

a) identificar detalhes dos eventos do sistema

Geralmente a descrição textual dos detalhes que envolvem os eventos de um sistema acaba não sendo claramente percebida pela quantidade de informações que ela traz. Nesses casos, os desenvolvedores deixam de projetar soluções para detalhes importantes que passaram despercebidos durante o projeto. O DSS facilita a percepção de todos os detalhes por apresentar graficamente cada evento e as informações referentes a cada um. Assim, a probabilidade de não se considerar algum detalhe por falta de atenção é pequena;

b) esclarecer quais operações importantes devem ser projetadas para lidar com esses eventos

O DSS permite uma visão ampliada dos eventos de um sistema, ou seja, é possível ver todos os eventos, suas relações e suas características ao mesmo tempo. Dessa forma, é mais fácil analisar quais as operações (soluções) são mais importantes para cada evento descrito. Isso facilita também a decisão de quais desses eventos são imprescindíveis para as versões do sistema, no caso de um ciclo de desenvolvimento de sistema incremental;

c) escrever os contratos de operações

Os contratos de operações especificam os detalhes das soluções propostas para os eventos identificados no projeto. Com a representação gráfica que encontramos no DSS, conseguimos mais facilmente definir os componentes dessas operações. O diagrama ainda facilita a identificação das associações entre os eventos e destes com o ambiente externo.

Resumo

Nesta aula estudamos o comportamento do sistema e sua representação na UML. Sua importância é grande para que seja possível a melhor compreensão do sistema que se pretende desenvolver. Através do diagrama os casos de uso são descritos, além de suas operações, de acordo com a finalidade do sistema. O DSS é a representação gráfica dos casos de uso estudados na fase inicial.

A seguir aprenderemos outra representação importante para o desenvolvimento de um sistema: o modelo de domínio (ou modelo conceitual).

Atividades de aprendizagem

1. Cite duas mensagens enviadas por um ator Estudante para um sistema chamado Acadêmico, referentes à matrícula em um novo curso, conforme o exemplo abaixo:

Mensagem 1 – Solicitar matrícula

Resposta 1 – Enviar lista de documentos necessários

Mensagem 2 – Entregar dados pessoais para matrícula

Resposta 2 – Informar que a matrícula foi concluída.

2. Explique como devem ser escritos os nomes das mensagens num DSS.

3. Quais os principais objetivos de um DSS?

Registre suas respostas num arquivo e poste-o no AVEA.

Aula 5 – Criação de um modelo do sistema

Objetivos

Usar o modelo de domínio para delinear o escopo do sistema a ser desenvolvido.

5.1 Introdução

A criação de modelos é uma atividade comum em várias áreas de trabalho, principalmente nas profissões que fazem projetos, como engenharia, arquitetura, etc.

O uso de modelos ajuda o profissional a entender melhor o funcionamento de um sistema, sua relação com o ambiente onde está inserido e sua troca de informações com os usuários (ou pessoas) que usarão esses sistemas para alguma tarefa.

Em Computação usamos frequentemente modelos para analisarmos e desenvolvermos sistemas de informação. O modelo de domínio é um desses diagramas utilizados pelos analistas.

Em todo projeto é importante o uso de modelos que nos ajudem a compreender melhor o funcionamento dos sistemas. Nesta aula estudaremos o modelo lógico (ou conceitual) de um sistema, sua importância e as técnicas utilizadas para sua criação. Ao final da aula saberemos como representar graficamente esse modelo. **O modelo de domínio não é modelo de projeto de *software*, isto é, não representa componentes de *software* (rotinas, bibliotecas, programas executáveis, etc.**).

5.2 Modelo de domínio

A modelagem de domínio, também chamada de modelagem conceitual, é a **atividade de encontrar quais conceitos são importantes para um determinado sistema**.

Esse processo nos **ajuda a entender melhor o sistema** e o negócio de nosso cliente. Ou seja, ajuda-nos a entender o que nosso cliente precisa para executar o seu trabalho.

A-Z

modelo de domínio
É uma representação de coisas do mundo real, isto é, "coisas" que nosso cliente é capaz de reconhecer como uma de suas tarefas de trabalho, ou ferramentas que ele usa para executar suas funções, etc.

A construção do **modelo de domínio** é basicamente a identificação dos diferentes conceitos no domínio do problema e a documentação dos resultados em um modelo conceitual.

Para Serafini e Cunha (2010) existem definições e recomendações importantes sobre como construir esse modelo. Segundo eles, é importante obter uma decomposição do problema, em termos de conceitos (e objetos) e de seus relacionamentos. No modelo conceitual, procuramos capturar todos os conceitos ou ideias que o cliente reconhece. Para construir um modelo conceitual de nosso sistema, precisamos identificar os **conceitos, os atributos de conceitos e os relacionamentos entre conceitos**.

5.3 Conceitos

Conceito é um objeto, uma coisa ou uma ideia (por exemplo: cliente, estudante, professor, matrícula, manutenção, atendente, venda, etc.).

Um conceito deve possuir:

a) símbolo: palavras ou imagens representando um conceito;

b) intenção: a definição de um conceito;

c) extensão: o conjunto de exemplos aos quais o conceito se aplica.

Exemplos:

a) pedido: em um sistema comercial;

b) jogador: em um sistema de jogos;

c) sala: em um sistema de escola;

d) quarto: em um sistema de hotéis.

Alguns péssimos exemplos de conceitos são:
DisparoDeEventos e FormularioDePedidos. Esses exemplos não são bons, pois eles estão voltados ao projeto (à solução) e não ao problema.

Uma boa regra é: se o cliente não entende um determinado conceito, provavelmente isso não é um conceito. Ou seja, se o cliente não consegue entender a importância do conceito no sistema ou não consegue identificar seu uso no sistema, já temos uma boa indicação de que esse conceito não pertence ao sistema em desenvolvimento.

5.4 Identificação de conceitos

Utilizamos normalmente uma abordagem semelhante àquela utilizada para encontrar os casos de uso, por exemplo, uma reunião (ou várias) com o maior número possível de clientes interessados no sistema.

Nessa reunião, devemos fazer um *brainstorm* para capturar todas as sugestões apresentadas pelos participantes. Em seguida, devemos trabalhar em grupo para discutir e justificar cada sugestão. Aplique a regra que diz que o cliente deve entender o conceito; caso contrário, o conceito deverá ser eliminado.

Lembre-se de que, como os conceitos são objetos, coisas ou ideias, devem ser identificados com um substantivo.

5.5 Atributo

Devemos incluir em nosso modelo conceitual todos os atributos que de alguma forma devem ser memorizados (ou guardados) pelos conceitos.

atributo
é um valor de dado lógico de um conceito.

Uma regra prática para tipo de atributo simples: faça dele um atributo, se ele puder ser visto naturalmente como um número, uma *string*, um *boolean*, uma data, etc.; caso contrário, represente-o como um conceito. Em caso de dúvida, defina-o como um conceito separado, em vez de um atributo.

Chamamos de atributos primitivos aqueles que são naturais para um determinado conceito. Ou seja, são informações que qualificam, particularizam ou definem aquele conceito. Os tipos de dados associados a esses atributos são os clássicos estudados nas disciplinas de programação (*strings*, *booleans*, inteiros, etc.). Existem, porém, atributos chamados de não primitivos. São aqueles compostos de seções separadas, ou que promovem operações que são usualmente associadas a eles próprios, tais como análise sintática ou validação; ou que têm outros atributos; ou que são uma quantidade com uma unidade claramente definida.

5.6 Identificação de atributos

Devido à semelhança entre atributos e conceitos, podemos usar as mesmas técnicas utilizadas em conceitos para a identificação dos atributos. Serafini e Cunha (2010) apresentam as seguintes dicas para a identificação dos atributos:

a) valores simples ou números, geralmente, são atributos;

b) se um conceito, ou uma propriedade, não pode fazer alguma coisa, ele é um atributo.

Exemplo:

Tenho um conceito chamado Estudante em um determinado sistema. Após a reunião com os clientes, foram identificados dois atributos: Nome do Estudante e Curso do Estudante. Porém, considerando que um curso não é um dado particular de estudantes (na verdade ele pode identificar várias outras "coisas" como: colégio, formação profissional, sala de aula, etc.), podemos pensar se Curso do Estudante deve ser um atributo de Estudante ou um novo conceito. A regra é transformar em conceito tudo que deixa dúvida em relação à sua classificação. Assim, Curso de Estudante será criado como um novo conceito em meu Modelo de domínio.

5.7 Associações

A maioria dos sistemas que iremos desenvolver apresentará, no mínimo, alguns conceitos que terão algum tipo de relacionamento com outros conceitos.

Por exemplo, num sistema acadêmico, um Professor leciona uma Disciplina. Nesse caso, temos uma relacionamento entre Professor e Disciplina, e esse relacionamento recebe o nome de "leciona". Ainda, um Estudante escolhe um Curso. O relacionamento é "escolhe". Outro exemplo: um Estudante faz uma Prova. O relacionamento é "faz".

Podemos dizer que uma **associação** é um **relacionamento entre conceitos** que indica uma conexão com significado e interesse definido.

Para que o relacionamento seja identificado como associação, ele deve ser preservado por algum tempo dentro do sistema. Ou seja, não faz sentido indicarmos um relacionamento entre dois conceitos que acontece uma única vez durante a existência do sistema (seria o mesmo que desenvolvermos uma função dentro de um sistema que seria usado somente uma única vez pelo usuário). É importante ainda lembrar que a associação também deve ser identificada por um nome, da mesma forma que o conceito e o atributo.

Usamos a definição de **multiplicidade (ou cardinalidade)** para definir a quantidade de relacionamentos que podemos identificar entre dois conceitos num certo sistema.

multiplicidade (ou cardinalidade)
Define o número de associações que dois conceitos podem ter.

Exemplo:

a) entre os conceitos Professor e Disciplina temos a associação "leciona". Dizemos então que sua cardinalidade é de "1 para muitos", porque um professor pode lecionar muitas disciplinas;

b) entre os conceitos Estudante e Turma temos a associação "pertence a". Dizemos que cardinalidade dessa associação é de "1 para 1", ou seja, um estudante só pode pertencer a uma turma.

Não existe nenhuma limitação quanto ao número de associações que os conceitos podem ter. A definição desse número depende do sistema em questão. Em alguns sistemas podemos encontrar uma cardinalidade "1 para muitos", enquanto em outros "muitos para muitos" para um mesmo par de conceitos. Por exemplo: Estudante pertence a Turma – em algumas instituições de ensino um estudante pode estar matriculado em várias turmas (se ele fizer mais de um curso) e, assim, a cardinalidade seria "1 para muitos", porém, em outras instituições um estudante só pode pertencer a uma turma, pois não se aceita estudante matriculado em mais de um curso.

5.8 A UML e a representação do modelo de domínio

A UML apresenta sua maneira de representar graficamente as definições que estudamos nas seções anteriores: conceito, atributo e associação. Essa representação gera um diagrama que chamamos de **modelo de domínio** ou de **modelo conceitual** da UML.

5.9 Representação de conceitos e atributos

Representamos um conceito com uma caixa dividida em três partes, conforme a Figura 5.1 a seguir.

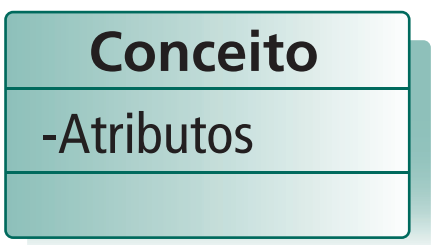


Figura 5.1: Símbolo de classe em UML
Fonte: Elaborada pelo autor

Na parte de cima, colocamos o nome do conceito; na parte do meio, colocamos os atributos do conceito. No modelo conceitual, a parte de baixo está sempre em branco, pois nela colocamos os comportamentos do conceito que não são analisados nesta etapa.

A Figura 5.2 mostra um exemplo em que observamos que o conceito Estudante tem dois atributos: nome e curso.

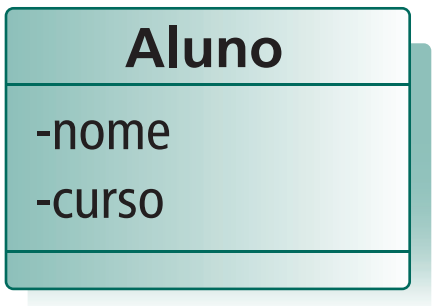


Figura 5.2: Exemplo em UML de um conceito: Estudante
Fonte: Elaborada pelo autor

5.10 Representação de associações

As associações são representadas por uma linha que liga os dois conceitos. Cada associação deverá receber um nome. Os números em cada lado da linha descrevem a cardinalidade (ou multiplicidade) da associação e nos informam quantas instâncias de cada conceito podem existir.

A Figura 5.3 apresenta a notação que utilizamos para mostrar a cardinalidade de uma associação:

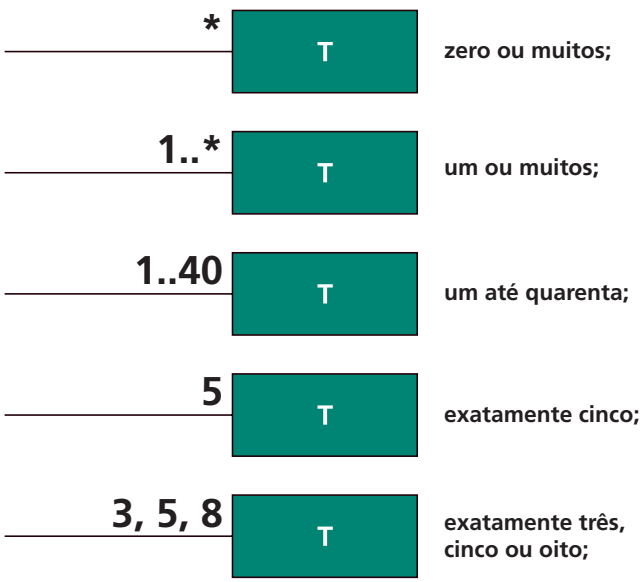


Figura 5.3: Multiplicidades entre conceitos
Fonte: Serafini e Cunha (2010, p. 65)

O símbolo "*" significa "muitos". Note a sutil diferença entre 1..* e *, este último é "muitos", significando zero ou mais conceitos, e o primeiro significa 1 ou mais conceitos.

Serafini e Cunha (2010) lembram que quando formos decidir sobre que nomes daremos às associações, devemos evitar nomes fracos, ou seja, nomes com significado muito genérico. Por exemplo: "tem", "é associado a", "possui", etc. A utilização desses nomes pode esconder problemas ou erros na definição da associação por darem uma ideia muito ampla de seu significado, deixando o desenvolvedor com a possibilidade de criar, no futuro, programas sem utilidade por ter interpretado erroneamente qual tipo de associação existe entre os conceitos.

5.11 O modelo conceitual

Após o estudo das seções anteriores, podemos começar a descrever nosso diagrama do modelo conceitual. As etapas a serem seguidas são:

a) escrever os conceitos e os atributos.

Por exemplo, considerando um sistema hospitalar, a Figura 5.4 apresenta os conceitos identificados após as reuniões com os médicos;

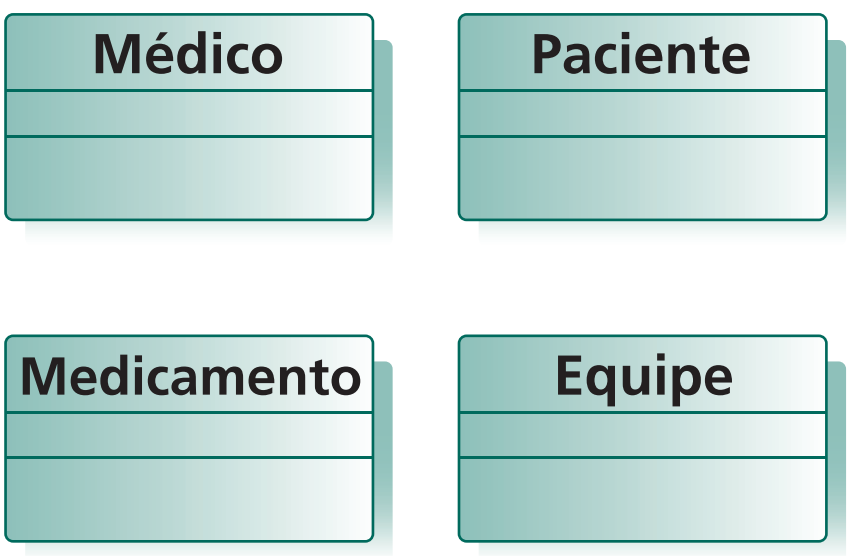


Figura 5.4: Exemplo
Fonte: Elaborada pelo autor

b) identificar as associações (relacionamentos) existentes entre conceitos.

Uma boa técnica é fixar um conceito e considerar as relações com os outros conceitos. Faça a pergunta "Esses dois conceitos estão relacionados?" e, se estiverem, imediatamente decida o nome do relacionamento e sua cardinalidade.

No exemplo anterior, as perguntas seriam feitas da seguinte maneira:

Pergunta 1 – Médico e Paciente estão relacionados?

Resposta 1 – Sim, cada médico atende um ou mais pacientes.

Pergunta 2 – Médico e Equipe estão relacionados?

Resposta 2 – Sim, cada médico dirige uma equipe médica.

Pergunta 3 – Médico e Medicamento estão relacionados?

Resposta 3 – Sim, cada médico prescreve um ou mais medicamentos.

Essas perguntas devem ser feitas até que se esgotem todas as possibilidades de associações entre os conceitos.

Ao final, o diagrama final ficará conforme a Figura 5.5:

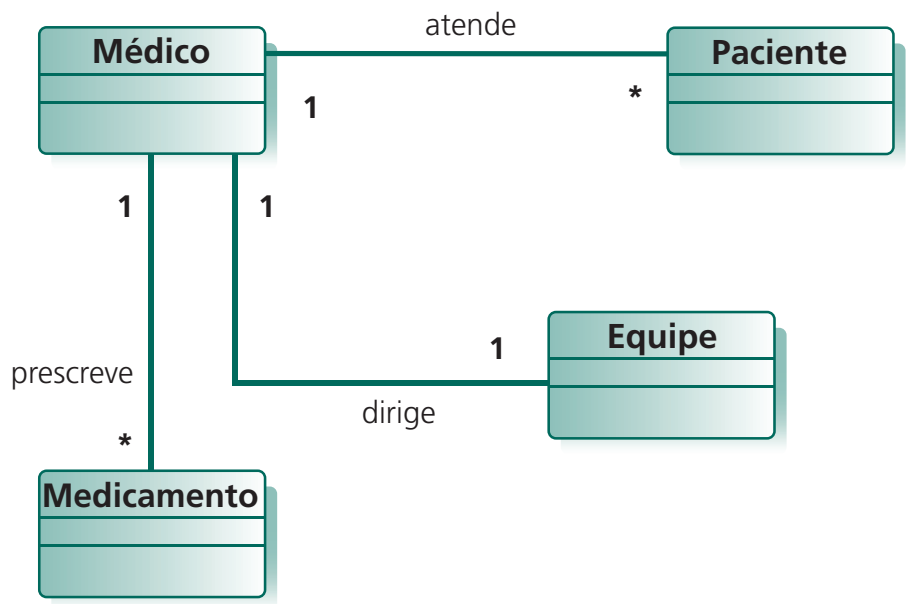


Figura 5.5: Modelo final
Fonte: Elaborada pelo autor

Os autores Larman (2007) e Serafini e Cunha (2010) apontam algumas regras que merecem ser seguidas para não construirmos diagramas incorretos:

a) as associações são menos importantes que os conceitos em um Modelo de domínio. Qualquer associação pode ser acrescentada facilmente durante o projeto, porém a inclusão de novos conceitos em fases mais adiantadas de um projeto é praticamente impossível visto o trabalho que teremos para refazer toda a análise já feita em função do primeiro modelo gerado;

b) "Não caia na tentação de escrever associações em excesso, como forma de se proteger de algum esquecimento. O que você vai acabar obtendo é um diagrama confuso e complexo e que não vai te ajudar nas próximas etapas." (SERAFINI; CUNHA, 2010, p. 66);

c) é importante que seus clientes aprovem o modelo conceitual gerado. Apresente e explique o modelo para todos os usuários envolvidos, a fim de que possam entendê-lo e se sentirem capazes de aprová-lo.

As ações a seguir são um roteiro para o desenho do diagrama do modelo conceitual:

a) listar os conceitos ***candidatos***, usando a identificação de substantivos relacionados com os requisitos que estão sendo considerados;

b) desenhar os conceitos no modelo conceitual (modelo de domínio);

c) escolher um conceito e fixar seu lugar no diagrama (modelo) e colocar graficamente os conceitos relacionados próximos a esse conceito;

d) encontrar, para cada conceito, as associações entre ele e os demais conceitos;

e) acrescentar a associação (relacionamento) entre os conceitos com a escrita de nome significativo;

f) acrescentar na associação sua cardinalidade;

g) acrescentar os atributos necessários para completar os requisitos de informação.

Resumo

Conhecemos o modelo conceitual e seus principais componentes nesta aula: conceito, atributo e associação. Vimos que esse modelo nos apresenta uma visão gráfica e global da relação existente entre os conceitos de um sistema, além de seus atributos. Estudamos também que as associações entre os conceitos podem ser uma ou várias e que elas indicam a cardinalidade da ocorrência do relacionamento entre dois conceitos. Aprendemos como representar graficamente todos os elementos do modelo e ainda como montar o diagrama final. Vimos também que podemos usar regras de levantamento de requisitos e casos de uso para identificar os conceitos e seu comportamento. Na aula seguinte vamos aprender a usar as informações que estudamos até agora (conceituais) para construção da arquitetura do sistema. Através dela definiremos as partes que comporão nosso futuro *software*.

Atividades de avaliação

1. Explique o que é uma associação e dê um exemplo do relacionamento entre os seguintes conceitos:

a) Pedreiro – Casa: constrói **(EXEMPLO)**

b) Professor – Estudante: ensina **(EXEMPLO)**

c) Motorista – Carro:

d) Banco – Correntista:

e) Médico – Paciente:

f) Juiz – Processo:

2. Desenhe um diagrama do modelo conceitual de um sistema hospitalar que atenda pessoas para internação, contendo os seguintes conceitos e associações:

Conceitos:

a) Médico

b) Paciente

c) Enfermeiro

d) Medicamento

e) Enfermaria

f) Prontuário Médico

g) Farmácia Hospitalar

Associações:

1. atende

2. cuida

3. prescreve

4. está internado

5. escreve

6. fornece

Registre suas respostas num arquivo e poste-o no AVEA.

Aula 6 – Arquitetura do sistema

Objetivos

Conhecer as principais arquiteturas possíveis para um sistema.

Escolher entre as arquiteturas na fase de projeto de um sistema.

6.1 Introdução

Da mesma maneira que um arquiteto trabalha organizando os espaços para a construção de uma casa, organizamos os componentes de *software* de maneira a distribuir da forma mais eficaz todos os componentes que farão parte do nosso sistema. Existem várias formas de se projetar um *software*, dependendo de seu objetivo e funcionalidade. Assim, também na **arquitetura de *software*** são definidos modelos que podem ser utilizados para que cada projeto esteja com o formato mais adequado para seu uso.

A-Z

Arquitetura de *Software*
É a forma como dividimos e organizamos o sistema em blocos reutilizáveis que aumentam a qualidade e reusabilidade do *software*

Segundo Booch, Rumbaugh e Jacobson (1997 apud LARMAN, 2007), uma Arquitetura de *software* é um conjunto de decisões significativas sobre a organização de um sistema, a seleção dos elementos estruturais e suas interfaces pelas quais o sistema é composto, juntamente com seu comportamento, suas colaborações e sua composição.

Começamos nesta aula o estudo da forma como nossos sistemas podem agrupar seus componentes. Pelo estudo das camadas nas quais podemos dividir nosso projeto, aprenderemos quais os tipos de arquitetura podemos usar para que tenhamos um bom projeto. Dessa forma, estaremos nos preparando para as futuras fases de desenvolvimento de sistemas (codificação) de forma planejada e que atenda às necessidades dos nossos clientes.

6.2 Definições de arquitetura de *software*

Após a fase em que levantamos os requisitos do sistema, precisamos nos preparar para transformarmos esses requisitos em uma implementação.

Uma boa estratégia para chegarmos à implementação é dividirmos o problema em problemas menores. Assim, esses problemas menores podem ser vistos como blocos, que possuem as seguintes vantagens:

a) podemos reutilizar esses blocos;

b) qualquer dependência dentro de um bloco fica local a ele;

c) podemos trocar blocos equivalentes;

d) alguns blocos podem ser padronizados.

A arquitetura de *software* mostra a forma como um sistema está dividido/organizado; mostra como as classes, os subsistemas e as camadas estão logicamente organizados.

A-Z

Camada
É definida como uma divisão lógica na arquitetura com uma responsabilidade específica sobre um tópico. Cada camada é responsável por um conjunto de classes, pacotes ou subsistemas.

A organização é feita utilizando-se os conceitos de **Camada** e Partições.

A interação entre as camadas se dá pela troca de informações entre elas, as que estão nos níveis superiores e as de níveis mais inferiores.

Larman (2007) aponta dois tipos de camadas (Figura 6.1):

a) **arquitetura em camadas estritas**, quando uma camada solicita apenas os serviços da camada inferior;

b) **arquitetura em camadas relaxadas**, quando uma camada superior solicita serviços de várias camadas inferiores.

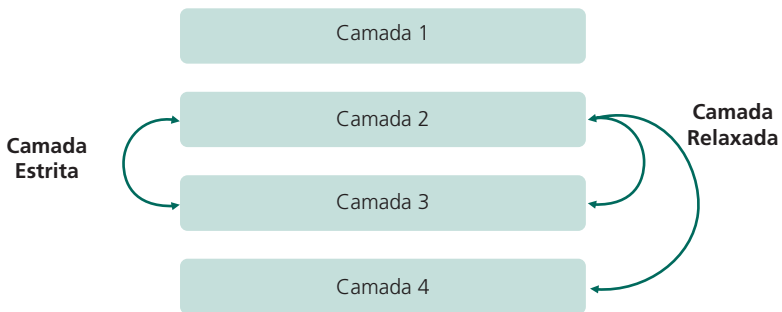


Figura 6.1: Tipos de arquiteturas em camadas
Fonte: Serafini e Cunha, 2010, p. 71

6.3 Divisão de um sistema em camadas

Para facilitar a divisão das camadas de um *software* é importante que:

a) cada camada tenha uma responsabilidade bem definida;

b) as camadas sejam definidas de maneira que as de níveis inferiores executem serviços gerais e as de níveis superiores executem mais serviços específicos do sistema;

c) as interfaces sejam do tipo caixa-preta (quando a camada *n* não sabe nada sobre a camada **n-1** e usa a interface pública para acessá-la) ou do

tipo caixa- branca (quando a camada **n** conhece os detalhes da camada **n-1** e usa esse conhecimento para acessá-la).

Talvez os termos mais adequados para os nomes dessas interfaces na língua portuguesa fossem: caixa opaca e caixa transparente. O nome "preta" é usado por ser uma cor que não permite visibilidade. O nome "branca", pelo contrário, seria a cor que mais permitiria visibilidade. Assim, quando dizemos que uma caixa é preta, estamos querendo dizer que não conseguimos ver o que existe em seu interior, ou seja, não sabemos detalhes da implementação daquele *software*. Por outro lado, a caixa branca seria aquela sobre a qual sabemos tudo sobre a implementação de seu *software*.

6.4 Representação da arquitetura de *software* na UML

A Figura 6.2 apresenta os símbolos que utilizamos em UML para representar os *packages* (pacotes) e subsistemas (partições de camadas complexas de um sistema).

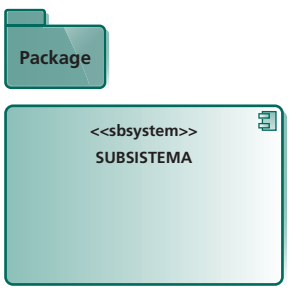


Figura 6.2: Notação UML para *packages* e subsistemas
Fonte: Elaborada pelo autor

As Figuras 6.3 e 6.4 mostram um exemplo de arquitetura.

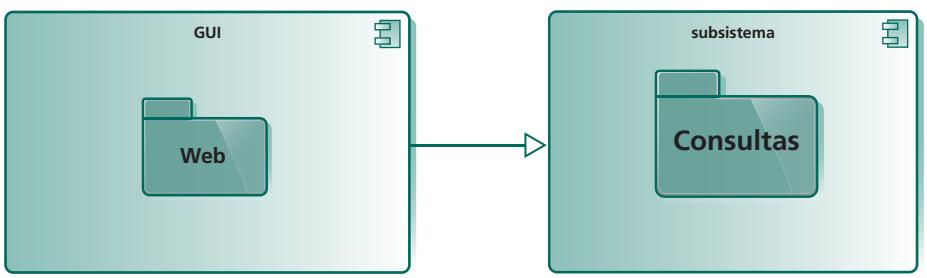


Figura 6.3: Exemplo de arquitetura
Fonte: Elaborada pelo autor

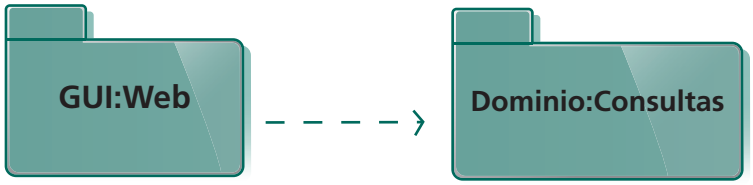


Figura 6.4: Outra notação de arquitetura
Fonte: Elaborada pelo autor

Observe que os exemplos acima são idênticos: mostram a mesma coisa. Ambos mostram os pacotes inseridos em seus respectivos subsistemas, porém com representações gráficas distintas. Use a representação que achar mais interessante em seus projetos.

6.5 Tipos de arquiteturas de sistemas de informação mais comuns

Estudaremos nesta seção os tipos de arquitetura de sistemas que se encontram mais comumente no mercado de desenvolvimento de *software*.

6.5.1 Arquitetura em uma única camada

Essa arquitetura era muito comum até a década de 1980, com a utilização de *mainframes* e terminais de computadores. Todo o processamento era centralizado em um único computador.

6.5.2 Arquitetura em duas camadas

Essa arquitetura surgiu com a difusão da utilização dos microcomputadores nas empresas e permite uma melhor **escalabilidade** na utilização dos recursos computacionais disponíveis. É uma arquitetura adotada nos sistemas cliente-servidor tradicionais.

Escalabilidade
É a capacidade de manipular uma porção crescente de trabalho de maneira uniforme. Estar preparado para crescer (SERAFINI; CUNHA, 2010, p.78).

As duas camadas são (Figura 6.5):

a) camada Cliente: trata da lógica de negócio e da interface com o usuário;

b) camada Servidor: trata dos dados (sistema de banco de dados)

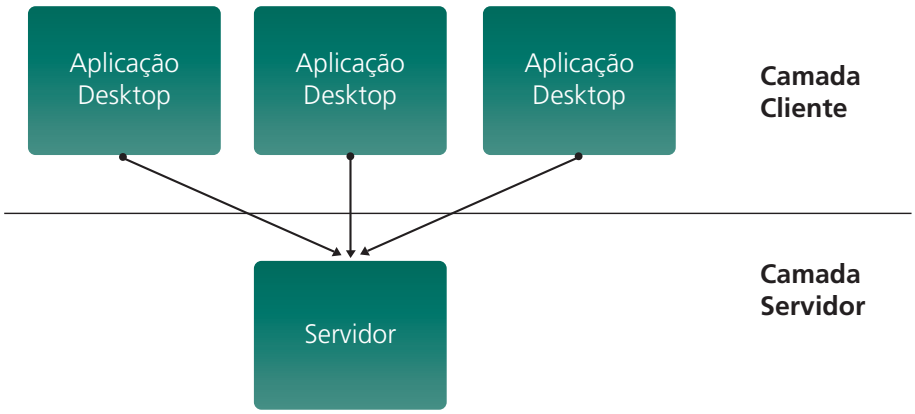


Figura 6.5: Arquitetura em duas camadas
Fonte: Serafini e Cunha (2010, p.77)

6.5.3 Arquitetura em três camadas

Essa arquitetura é uma melhoria da arquitetura em duas camadas e resolve alguns problemas:

a) aumenta a escalabilidade;

b) permite mudança mais fácil na lógica da aplicação ou na interface com o usuário;

c) permite acessar fontes de dados heterogêneas, isto é, sistemas gerenciadores de bancos de dados diferentes.

As camadas são três (Figura 6.6):

a) camada de Apresentação: interface gráfica com o usuário;

b) camada de Aplicação: lógica do negócio;

c) camada de Dados: sistemas de bancos de dados.

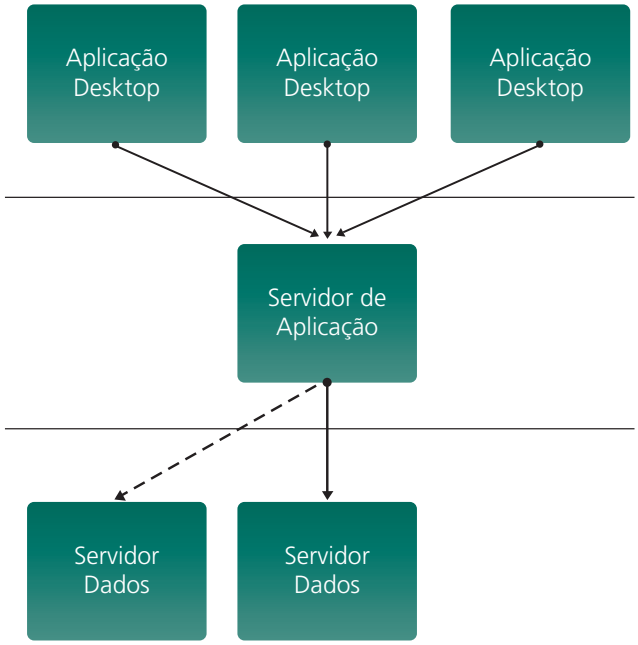


Figura 6.6: Arquitetura em três camadas
Fonte: Serafini e Cunha (2010, p.78)

É importante lembrar que estamos falando de camadas lógicas. Fisicamente, em uma mesma máquina, podemos ter mais de uma camada lógica, mas na maioria das vezes isso não ocorre!

6.5.4 Arquitetura em n-camadas

Com a disseminação da internet, o acesso a sistemas via *web* tem se tornado muito comum e útil.

Assim, surgiram as arquiteturas multicamadas, ou seja, aquelas que têm várias camadas. Suas características são:

a) permitem o acesso por meio de navegadores *web*;

b) qualquer dispositivo pode acessar o sistema via *web*;

c) não necessitam da instalação de *software* nos clientes (só o navegador!);

d) a atualização do *software* é imediata;

e) permitem balancear a carga de serviços em vários servidores.

Uma arquitetura típica n-camadas é mostrada na Figura 6.7 a seguir.

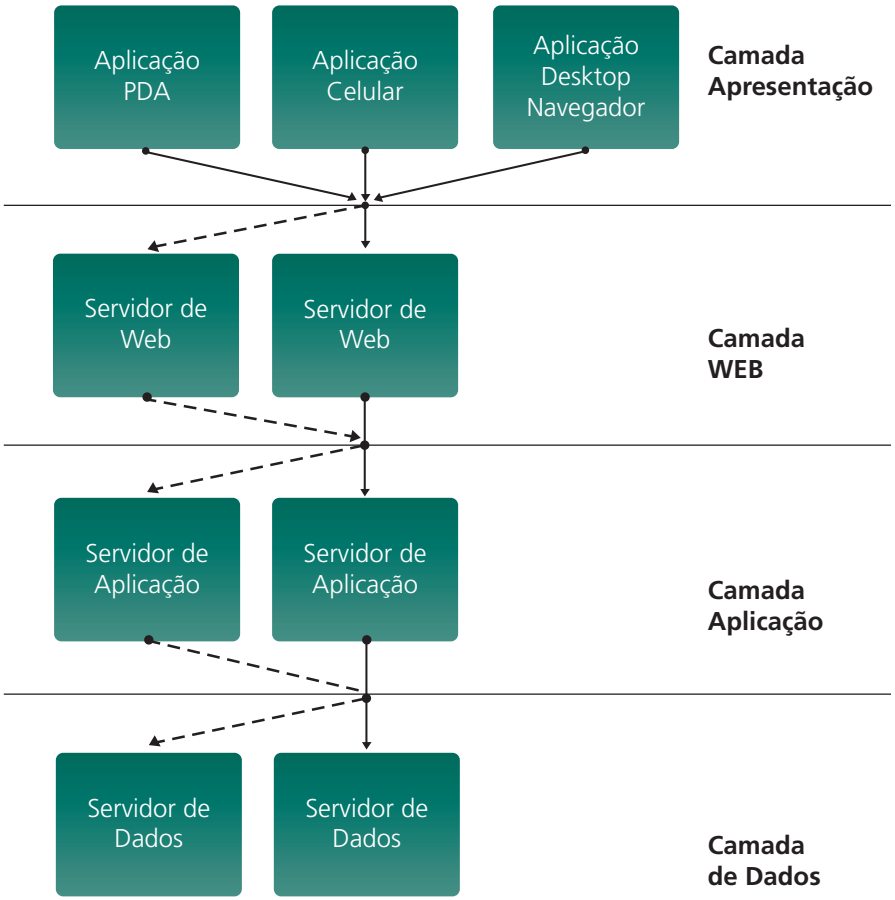


Figura 6.7: Arquitetura em quatro camadas
Fonte: Serafini e Cunha (2010, p.79)

6.6 *Service oriented architecture* (SOA) - Arquitetura orientada a serviços

Uma arquitetura SOA permite que um programa acesse uma aplicação remota. A tecnologia empregada atualmente é a de ***Web Services***.

A SOA expõe as funcionalidades de um sistema para outro programa.

Essa arquitetura apresenta como principais vantagens (SERAFINI; CUNHA, 2010, p. 80):

a) a independência de plataforma. Por exemplo, uma parte do sistema pode ser feita em uma linguagem e a outra parte em uma segunda linguagem de programação;

b) a integração de serviços de forma fácil;

c) composição de serviços para criar serviços maiores e mais complexos.

A-Z

Web Service
É uma solução utilizada na integração de sistemas e na comunicação entre aplicações diferentes. Com essa tecnologia é possível que novas aplicações possam interagir com aquelas que já existem e que sistemas desenvolvidos em plataformas diferentes sejam compatíveis. Os *Web Services* são componentes que permitem às aplicações enviar e receber dados em formato XML. Cada aplicação pode ter a sua própria "linguagem", que é traduzida para uma linguagem universal, o formato *XML*.

Resumo

Nesta aula aprendemos importantes conceitos sobre arquitetura de *software*. A partir do detalhamento das partes que compõem conceitualmente o sistema, chegamos aos pacotes e subsistemas. A arquitetura é a organização desses blocos de forma a aumentar a qualidade e eficácia dos sistemas.

Estudamos também os conceitos de caixa-preta e caixa-branca, que são utilizados quando estamos projetando a arquitetura de um *software*.

Na última seção desta aula vimos quais os tipos de arquiteturas são mais comumente usados no mercado: arquitetura em uma camada, arquitetura em duas camadas, arquitetura em três camadas e, por fim, arquitetura em n-camadas.

A seguir, conheceremos um dos diagramas mais importantes da UML, o diagrama de classes. A partir do levantamento dos requisitos do sistema e da definição do modelo conceitual e da arquitetura, estaremos prontos para definirmos as classes pertencentes ao sistema a ser desenvolvido, bem como suas relações e métodos.

Atividades de aprendizagem

1. Explique as diferenças entre os tipos de arquitetura em **n** camadas e SOA.

2. Cite os tipos de camadas existentes e explique as diferenças entre eles.

3. Quais as representações gráficas para o *package* e subsistemas de acordo com a UML.

4. Após estudar o conceito de escalabilidade, indique quais tipos de arquitetura favorecem a aplicação desse conceito nos sistemas. Explique sua resposta.

Registre suas respostas num arquivo e poste-o no AVEA.

Aula 7 – Diagrama de classes

Objetivos

Conhecer os elementos do diagrama de classes.

Usar o diagrama de classes para a modelagem dos dados de um sistema.

7.1 Introdução

Após os levantamentos feitos na fase inicial, quando conhecemos os requisitos e domínio do sistema, escrevemos o diagrama de classes. Ele apresenta o sistema de forma estática e detalha quais classes e seus relacionamentos foram identificados no sistema a partir da aprovação do cliente.

7.2 Representação dos elementos do diagrama de classes

As **classes** na UML são representadas por um retângulo dividido em três partes (Figura 7.1):

a) na primeira temos o nome da classe;

b) na segunda temos os atributos da classe;

c) na terceira temos os métodos da classe.

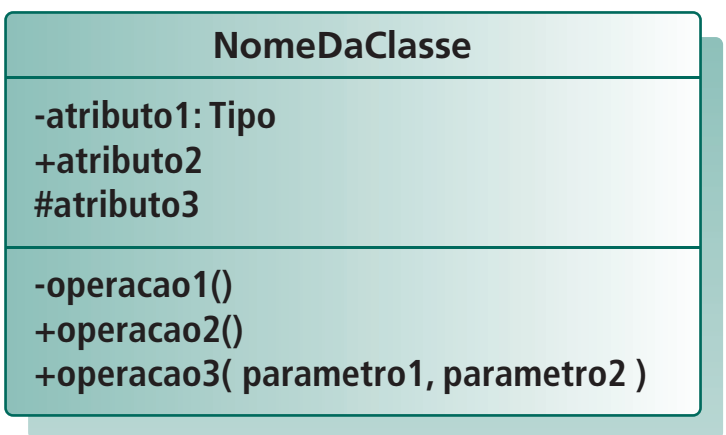


Figura 7.1: Nome da classe
Fonte: Serafini e Cunha (2010, p. 101)

Os **atributos** podem ser mostrados como:

a) um elemento textual (Figura 7.2):

Figura 7.2: Elemento textual
Fonte: Serafini e Cunha (2010, p. 102)

Formato:

[visibilidade] nome do atributo: tipo

ou

b) uma associação (Figura 7.3):

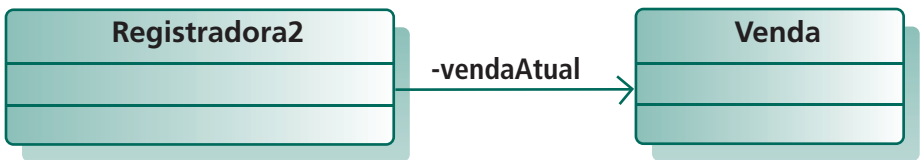


Figura 7.3: Associação
Fonte: Serafini e Cunha (2010, p. 102)

Nesse exemplo Registradora possui uma associação com Venda chamada vendaAtual.

Normalmente usamos o elemento textual para representarmos tipos de dados simples (ou primitivos) disponíveis na linguagem de programação, por exemplo: inteiro, *string*, etc. Para representarmos estruturas como *arrays*, vetores, etc., usamos a associação.

Os **métodos** (ou operações) são mostrados na parte de baixo do símbolo de classe (Figura 7.4):

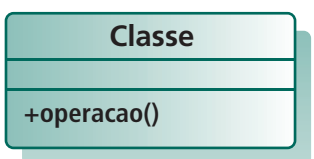


Figura 7.4: Métodos
Fonte: Serafini e Cunha (2010, p.103)

Formato:

[visibilidade] nome da operação (lista de parâmetros): tipo de retorno

Operações especiais que geralmente **não** aparecem nos diagramas:

a) criação de objetos (*create*): geralmente associada ao operador *new*.

b) operadores de acesso: são os métodos *get/set* para acesso aos atributos.

Utilizamos a seguinte notação para a visibilidade de atributos e métodos:

+: *public*

-: *private*

#: *protected*

Podemos utilizar palavras-chave para caracterizar nossas classes.

<<ator>>

<<interface>>

[*abstract*]

[*ordered*]

Utilizamos palavras-chave quando desenhamos as classes a mão, ou seja, sem o apoio de um *software*.

Para mostrar objetos do tipo Interface podemos utilizar o símbolo (Figura 7.5):

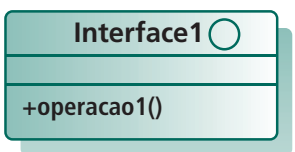


Figura 7.5: Interface (modelo 1)
Fonte: Serafini e Cunha (2010, p.104)

Ou então o símbolo tradicional de classe com a palavra-chave <<interface>> antes do nome da interface (Figura 7.6).

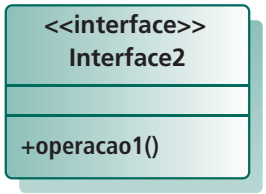


Figura 7.6: Interface (modelo 2)
Fonte: Serafini e Cunha (2010, p.104)

As **coleções de objetos** (*arrays*, vetores, etc.) podem ser mostradas como:

a) um elemento textual (Figura 7.7)

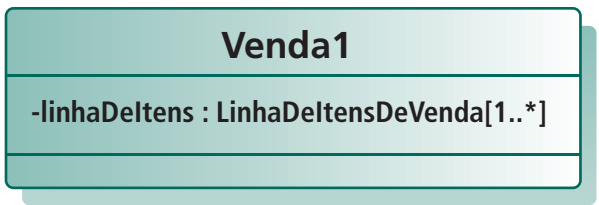


Figura 7.7: Elemento textual
Fonte: Serafini e Cunha (2010, p.105)

Formato:

[visibilidade] nome do atributo: tipo [ocorrências]

b) uma associação (Figura 7.8)

Figura 7.8:Associação
Fonte: Serafini e Cunha (2010, p. 105)

Na Figura 7.8, Venda2 possui uma associação com LinhasDeItensdeVenda chamada linhaDeItens.

Usamos a representação gráfica da Figura 7.9 para inserir comentários em um diagrama de classes.

Figura 7.9: Notação de comentário
Fonte: Fowler (2004, p. 105)

Uma **generalização** é mostrada com o símbolo  , como mostrado na Figura 7.10.

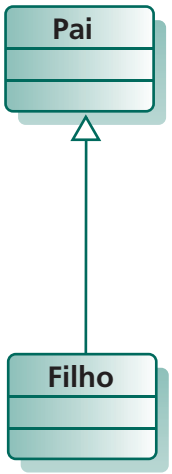


Figura 7.10: Generalização
Fonte: Serafini e Cunha (2010, p. 106)

Uma **dependência** é mostrada com uma linha (tracejada com uma seta partindo do cliente para o fornecedor) entre dois elementos, como mostrado na Figura 7.11.

Figura 7.11: Relação de dependência
Fonte: Serafini e Cunha (2010, p. 106)

Uma dependência indica qualquer tipo de visibilidade entre duas classes.

Veja o exemplo da Figura 7.12:

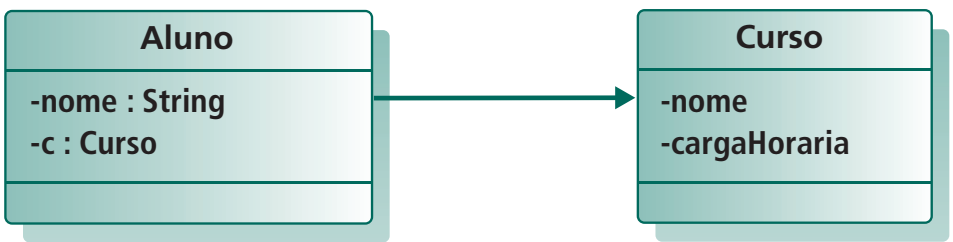


Figura 7.12: Relação de dependência
Fonte: Serafini e Cunha (2010, p. 107)

A **composição** e a **agregação** são usadas para representar as relações de "todo" e "parte" existentes entre alguns objetos.

Denominamos **composição** a relação na qual o objeto "todo" é constituído pelo objeto "parte", de maneira que sem o objeto "todo" o objeto "parte" não tem sentido.

A Figura 7.13 apresenta uma composição:

Figura 7.13: Composição
Fonte: Serafini e Cunha (2010, p.107)

A instância Endereço só faz sentido se houver uma instância correspondente de Cliente. Ou podemos perguntar: para que guardar um endereço se não sei a que cliente ele pertence?

Graficamente falando, a representação da composição contém um pequeno losango preenchido em um dos lados da linha horizontal que liga os dois retângulos.

Quando excluímos o objeto "pai", destruímos também o objeto "filho". No nosso exemplo, quando destruímos o objeto Cliente destruímos também o objeto Endereço.

A **Agregação** acontece quando usamos também um objeto "todo" e outro objeto "parte"; porém, a "vida" dos objetos é independente.

Veja o exemplo na Figura 7.14:

Figura 7.14: Agregação
Fonte: Serafini e Cunha (2010, p.107)

Observe que, nesse caso, a representação gráfica possui o losango NÃO preenchido. No exemplo da Figura 7.14, se eliminarmos o objeto LinhaPedido, não devemos eliminar a DescriçãoProduto. Faz sentido não eliminarmos a DescriçãoProduto pois esse objeto provavelmente deverá ser utilizado por outros objetos. Em termos de programação Java, a diferença entre **composição** e **agregação** é sutil e pode ser ignorada.

Não se preocupe em utilizar **agregação** em seus projetos, utilize **composição** sempre que for apropriado.

7.3 Mapeamento objeto-relacional

Um dos problemas de se projetarem sistemas usando o paradigma de Orientação a Objetos é como implementar os objetos (que possuem informações a serem guardadas) em bancos de dados que não foram criados para este paradigma. Ou seja, os bancos de dados mais comumente encontrados no mercado são relacionais, e, portanto, trabalham com implementação de entidades e relacionamentos (modelo relacional), enquanto o produto gerado pelo desenvolvimento baseado em Orientação a Objetos produz objetos. Assim, precisamos converter os objetos, gerados nos diagramas de classes, em entidades e relacionamentos que poderão ser gerados dentro de um banco de dados relacional.

Sempre tenha seu cartão de referência UML em mãos.
Consulte o *site* www.omg.org para obter a especificação oficial de UML 2.0
Aprenda a usar um programa para desenhar diagramas UML (BlueJ, Astah, Rose, etc.).

Os bancos de dados adequados para o desenvolvimento de aplicações OO são os SGBDOO (Sistemas Gerenciadores de Bancos de Dados Orientados a Objetos); porém, a indisponibilidade desses bancos de dados no mercado, seja devido ao custo, diversidade ou oferta, faz com que seja necessária a busca de técnicas específicas para a solução desse problema, para a criação de objetos em bancos de dados que usa entidades.

Vamos conhecer essa técnica.

Considere o diagrama de classe da Figura 7.12. Nele temos duas classes: Alunos e Curso. A associação entre eles informa que alunos fazem cursos. Esse diagrama pode ter sido gerado a partir de um diagrama de entidades e relacionamentos, onde: a entidade ALUNO equivale à classe Aluno e a entidade CURSO equivale à classe Curso. Para detalharmos melhor o diagrama de classes da Figura 7.12 podemos mostrar os atributos e métodos associados às classes Aluno e Curso.

Assim, cada tabela do banco de dados foi mapeada para uma classe, mantendo-se o mesmo nome da entidade (tabela) na classe.

No diagrama de classes, os atributos dessas classes possuirão visibilidade *private*, ou seja, somente os métodos internos à classe é que poderão acessá-los diretamente. Para objetos externos acessarem esses atributos, é necessário utiliza *get* e *set*. Caso o atributo seja apenas de leitura não será preciso usar a propriedade *set*.

Até aqui transformamos as entidades em classes; porém, os relacionamentos ainda não foram tratados.

No nosso exemplo (Figura 7.12) temos o relacionamento do tipo 1:n (um para n). Assim, podemos ler o relacionamento da seguinte maneira: um aluno pode fazer um curso, porém um curso pode ser feito por vários alunos.

Para construir o relacionamento inserimos a chave estrangeira Curso.Codigo_Curso na tabela Aluno associando-a à chave primária Codigo_Curso da tabela Curso. No diagrama de classes também temos os atributos Curso.Codigo_Curso e Código_Curso. Através de métodos específicos, preenchemos essas chaves e informamos a quais cursos estão ligados cada aluno registrado na tabela Aluno.

Mídias integradas: Um bom resumo do que estudamos nesta aula pode ser visto no vídeo chamado "Diagrama de Classes", disponível em http://www.youtube.com/watch?v=F4fRIvPuZU8. Após assistir ao vídeo, escreva os principais pontos apresentados que, na sua opinião, não podem faltar em um diagrama de classes. Envie para seu tutor sua resposta.

Resumo

Aprendemos nesta aula a representação gráfica do diagrama de classes. Apesar dos novos conhecimentos sobre cada elemento e conceito, é importante lembrarmos sempre da forma correta de representá-los, porque somente assim nossos diagramas poderão ser lidos e entendidos por todos que trabalham no desenvolvimento de um sistema.

Vimos que os conceitos estudados são todos provenientes da Orientação a Objetos e representam a relação existente entre as classes, objetos e suas especificações.

Nosso próximo passo é aprendermos como usar os padrões de projetos de acordo com os estudos e diagramas produzidos até esta aula.

Atividades de aprendizagem

1. Use sua criatividade e desenhe um diagrama de classes que possua os seguintes conceitos:

1. Homem

2. Mulher

3. Certidão de Casamento

4. Filhos

5. Residência

2. Desenhe um diagrama de classe (com apenas duas classes e um relacionamento entre elas) que apresente uma dependência.

3. Desenhe um diagrama com quatro classes que apresente:

a) Método,

b) Atributos,

c) Composição e Generalização

Registre suas respostas num arquivo e poste-o no ambiente virtual.

Aula 8 – Projeto de objetos

Objetivos

Conhecer os padrões de projeto GRASP.

Entender a aplicação dos padrões de acordo com as questões de programação existentes nos sistemas.

8.1 Introdução

Quando identificamos os conceitos em nosso sistema, identificamos o que ele deve ser capaz de realizar, isto é, identificamos as suas responsabilidades. Podemos esperar que os conceitos dependam da ajuda de outros conceitos, isto é, que eles colaborem entre si para realizar as suas responsabilidades." (SERAFINI, 2008, p. 95).

8.2 Responsabilidade e colaboração

Veja o caso de uso "Pedir Emprestado um Livro" em um sistema de Biblioteca. O modelo de domínio (ou conceitual) para o sistema de Biblioteca obtido está representado na Figura 8.1 a seguir.

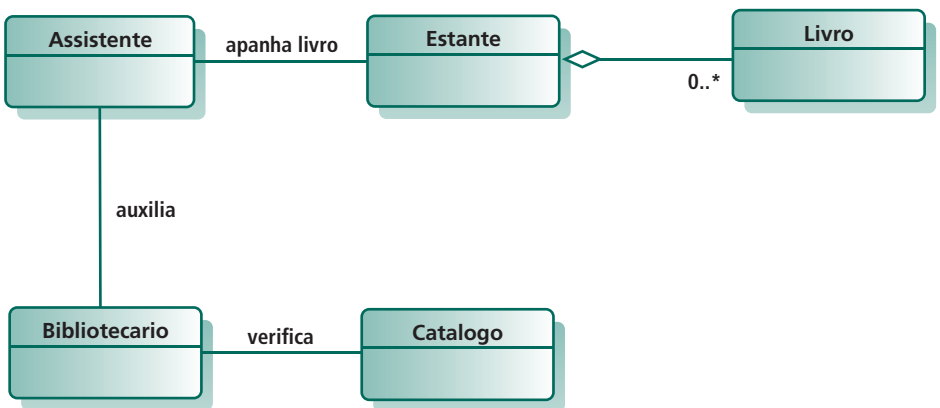


Figura 8.1: Modelo conceitual
Fonte: Serafini (2008, p. 98)

A-Z

responsabilidade
é aquilo que um objeto deve ser capaz de fazer/executar. Esse conceito diz respeito às próprias ações que um objeto deve fazer, ou seja, são suas próprias responsabilidades dentro do sistema.

colaboração
é aquilo que um objeto pode fazer/executar para outro objeto, de modo que o caso de uso possa ser realizado. Dessa forma, um objeto ajuda o outro a executar aquilo que é a responsabilidade do outro.

Podemos identificar a colaboração de vários "objetos":

a) o Usuário pede a colaboração do Bibliotecário;

b) o Bibliotecário pede a colaboração do Catálogo;

c) o Bibliotecário pede a colaboração do Assistente.

Assim temos:

8.3 Definição de padrão

A-Z

padrão
É uma solução muito utilizada e genérica para um problema.

Padrão, de uma forma geral, é tudo aquilo que serve de solução genérica para problemas semelhantes. Os especialistas em qualquer profissão adotam procedimentos padrões para a solução de problemas que ocorrem comumente em suas tarefas.

Vamos examinar algumas definições:

Os padrões são sempre escritos (ou identificados) obedecendo às seguintes informações: nome do padrão, problema que ele se propõe resolver, solução apresentada pelo padrão.

8.4 Os padrões GRASP

Segundo Serafini (2008), Craig Larman adaptou os critérios fundamentais de projeto ao formato dos padrões de projeto e foi muito feliz com essa abordagem.

A utilização dos padrões de projeto propostos por Larman (2007) pode melhorar bastante nossos diagramas de colaboração e, por conseguinte, nossos projetos.

Larman deu nome a seu conjunto de regras de GRASP que significa *"General Responsibility Assigment Software Patterns,"* ou seja, Padrões de "*Software*" Genéricos de Atribuição de Responsabilidades, e esses padrões podem nos ajudar a alocar responsabilidades (ou comportamentos) às classes de uma forma muito prática.

Os padrões GRASP são:

a) "*Expert*" – Especialista

b) "*Creator*" – Criador

c) "*High cohesion*" – Alta coesão

d) "*Low coupling*" – Baixo acoplamento

e) "*Controller*" – Controlador

Os nomes de padrões geralmente não são traduzidos para o português.

8.4.1 Grasp 1: "*Expert*"

É um padrão muito simples e um dos mais desobedecidos.

O padrão "***Expert***" responde à questão:

Dado um comportamento, a que classe deve ser atribuído esse comportamento?

Resposta: À classe que contém a informação desejada. A classe especialista ("*Expert*") na informação!

A resposta a essa pergunta leva à alocação inteligente dos comportamentos, que por sua vez resulta em sistemas mais fáceis de entender, de se expandir, de se reutilizar e ainda mais robustos.

Veja o exemplo a seguir:

Considere três classes: uma representando Pedidos, outra LinhaPedido e outra CatalogoDeProduto. Um trecho do modelo conceitual é (Figura 8.2):

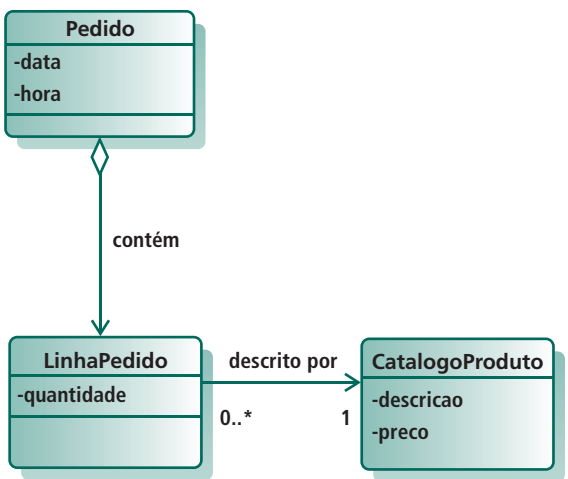


Figura 8.2: Diagrama de classes parcial
Fonte: Serafini (2008, p.101)

Agora suponha que estamos construindo a colaboração para um dos casos de uso e esse caso de uso necessita que o total de um pedido seja apresentado ao usuário.

A que classe deve ser alocado o comportamento calcularTotal()?

O padrão "*Expert*" diz que a única classe que deve ser permitida lidar com o Custo Total de um pedido é a classe Pedido – isso porque ela deve ser especialista sobre todas as "coisas" relativas a pedidos.

Então alocamos o método calcularTotal() à classe Pedidos.

Mas para calcular o total do pedido, este precisa saber o total de cada linha de Pedido.

A opção de deixar que Pedido acessasse as informações de LinhaPedido e realizasse a soma tornaria mais pobre nosso projeto.

Mais uma vez aplicamos o padrão "*Expert*", e temos que a única classe que deveria calcular o total de Linha de Pedido é LinhaPedido; assim, criamos um método calcularSubTotal() e alocamos em LinhaPedido.

Por sua vez, calcularSubTotal() precisa obter o preço unitário do produto; então, o "*Expert*" sugere alocar o método obterPreco() a CatalogoProduto.

Assim teremos o diagrama de classes parcial mostrado na Figura 8.3.

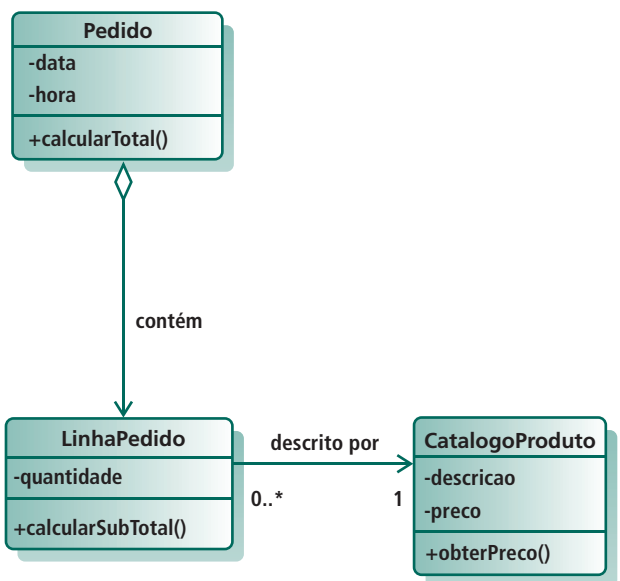


Figura 8.3: Diagrama de classes (Pedido – LinhaPedido – CatalogoProduto)
Fonte: Serafini (2008, p.102)

E nosso diagrama de colaboração está mostrado na Figura 8.4.

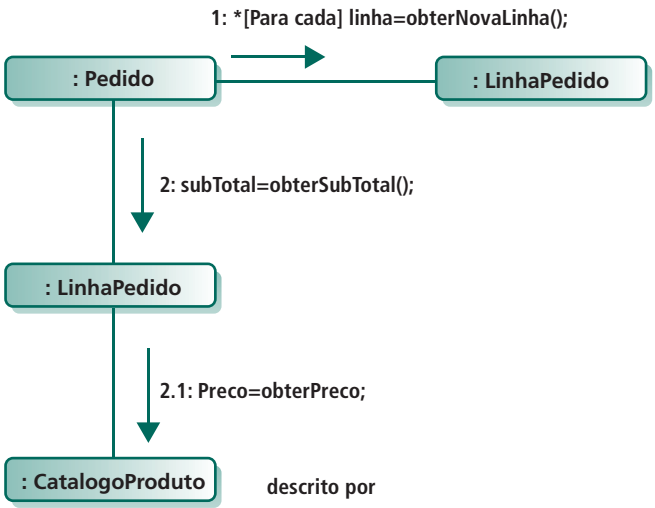


Figura. 8.4: Diagrama de colaboração
Fonte: Serafini (2008, p.102)

Síntese:

Nome: EXPERT

Problema: Dado um comportamento, a que classe deve ser atribuído esse comportamento?

Solução: À classe que contém a informação desejada. A classe especialista ("*expert*") na informação (LARMAN, 2007).

8.4.2 Grasp 2: "*Creator*"

O padrão "*Creator*" é uma especialização do padrão "*Expert*".

Ele responde à questão:

Quem deve ser responsável pela criação de instâncias de uma classe particular?

A resposta é que a classe A deve ser responsável pela criação dos objetos da classe B se:

a) A contém objetos B;

b) A utiliza objetos B;

c) A contém os dados de inicialização que devem ser passados à classe B.

No nosso exemplo anterior, quando um novo pedido é criado, quem deve ser responsável pela criação dos objetos LinhaPedido?

A solução apontada pelo padrão "*Creator*" é que, como pedido contém LinhaPedido (veja a Figura 8.5), então a classe Pedido (e somente ela) deve ser responsável pela criação de Linhas de Pedidos.

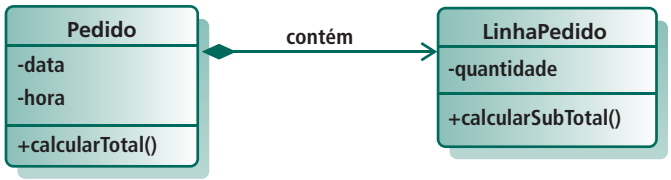


Figura 8.5: Diagrama de classes parcial - (Pedido – LinhaPedido)
Fonte: Serafini (2008, p.103)

O diagrama de colaboração está mostrado na Figura 8.6:

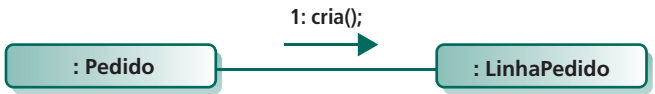


Figura 8.6: Diagrama de colaboração
Fonte: Serafini (2008, p.104)

Síntese:

Nome: CREATOR

Problema: Quem deve ser responsável pela criação de instâncias de uma classe particular?

Solução: a classe A deve ser responsável pela criação dos objetos da classe B se:

a) A contém objetos B;

b) A utiliza objetos B;

c) A contém os dados de inicialização que devem ser passados a classe B.

(LARMAN, 2007)

8.4.3 Grasp 3: "*High cohesion*"

Uma alta coesão significa que as responsabilidades de uma classe são fortemente relacionadas. Uma classe que faz muitas coisas não relacionadas possui uma baixa coesão.

É extremamente importante garantir que as responsabilidades de cada classe sejam focalizadas. Em um bom projeto orientado a objetos, cada classe não deve realizar trabalho excessivo. Um sinal de que o projeto orientado a objetos é bom é cada classe possuir um pequeno número de métodos.

Exemplo

No projeto de um sistema de controle de elevadores, uma classe ControladorElevadores foi projetada (Figura 8.7):

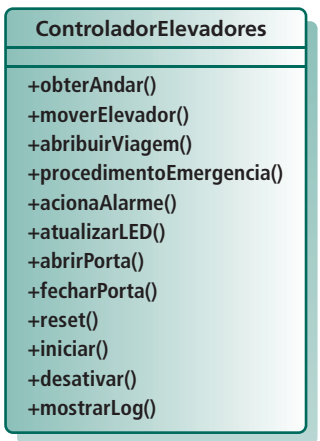


Figura 8.7: Classe ControladorElevadores
Fonte: Serafini (2008, p.104)

Observe como a classe ControladorElevadores realiza várias funcionalidades de trabalho: liga alarmes, inicia, desativa, move o elevador...

Esse não é um bom projeto, porque a classe não é coesiva, ela está tentando realizar muitas coisas diferentes.

Essa classe deve ser difícil de manter, e não está claro o que ela deve fazer, isto é, quais as suas responsabilidades.

Nosso sistema de controle de elevador está tentando modelar no mínimo umas três abstrações: um alarme, as portas do elevador e o registro de falhas...

Separando essas abstrações, teríamos um projeto melhor para o sistema de controle de elevadores. Veja a Figura 8.8:

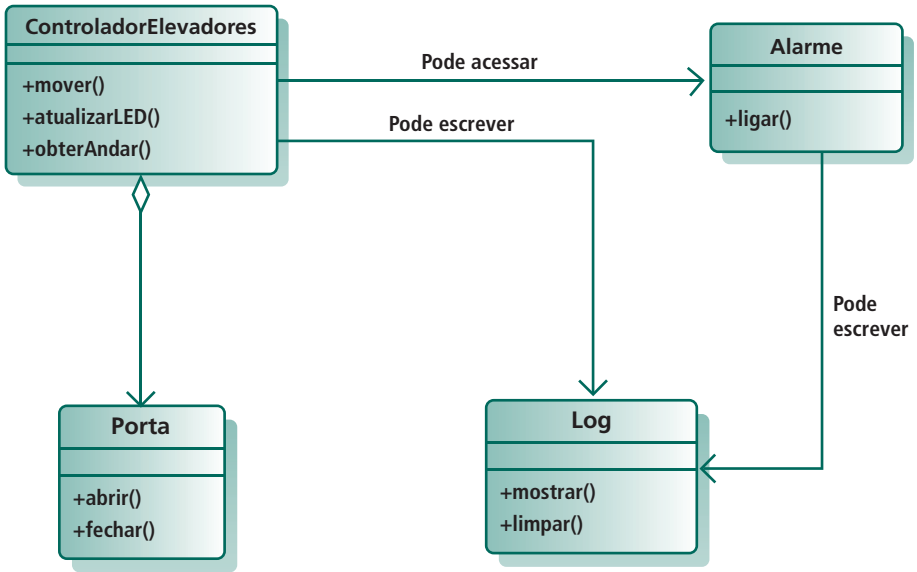


Figura 8.8: Diagrama de classes
Fonte: Serafini (2008, p.105)

Síntese:

Nome: "*High Cohesion*"

Problema: Como manter a coesão alta?

Solução: Cada classe deve representar uma única "coisa" (ou abstração) do mundo real.

(LARMAN, 2007)

8.4.4 Grasp 4: "*Low Coupling*"

Acoplamento é uma medida de quão dependente uma classe é de outra.

Um acoplamento alto entre módulos leva a um projeto que é difícil de manter ou modificar – uma simples mudança em uma classe pode ocasionar mudanças em várias outras classes (SERAFINI, 2008).

O diagrama de colaboração fornece um excelente meio de verificar o nível de acoplamento do projeto. O diagrama de classes também dá uma ideia do nível de acoplamento. Veja o diagrama mostrado na Figura 8.9.

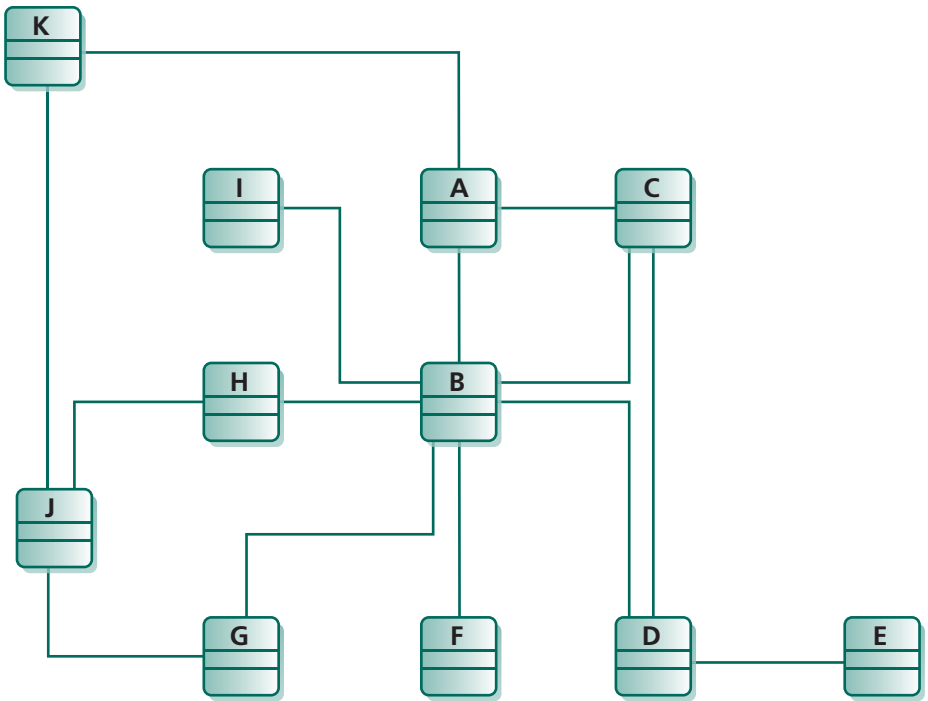


Figura 8.9: Diagrama de classes
Fonte: Serafini (2008, p.106)

Todas essas associações são necessárias?

O projetista desse sistema deveria questionar seriamente alguns detalhes de projeto.

Exemplo:

- Por que a classe C está associada à classe A, se existe uma associação indireta pela classe B? Essa associação pode ter sido criada para melhorar o desempenho, o que seria razoável, ou pode ter sido criada por desleixo...
- Por que a classe B possui tantas associações? Essa classe provavelmente está realizando muito trabalho.

Seguir o modelo conceitual é uma forma excelente de reduzir o acoplamento. Somente envie mensagem de uma classe à outra se a associação foi identificada durante a fase de modelagem conceitual. Agindo assim, você estará se restringindo a introduzir acoplamentos que existem no mundo real.

Exemplos:

No nosso sistema de Pedidos, identificamos no modelo conceitual que Clientes possuem Pedidos.

No caso de uso "Criar Pedidos", que classe deve ser responsável por criar um novo pedido?

O padrão "*Creator*" sugere que a classe Cliente deva ser responsável (Figura 8.10):

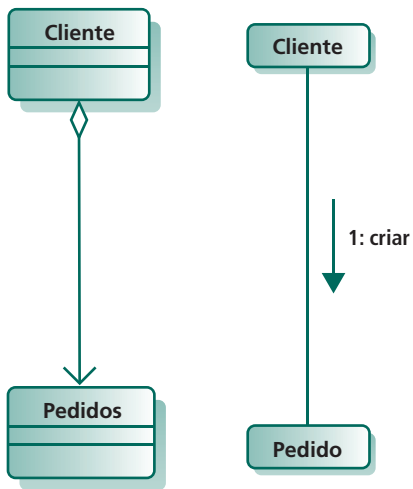


Figura 8.10: Padrão "*Creator*"
Fonte: Serafini (2008, p.108)

Assim, acoplamos Cliente a Pedidos. Neste caso, está "*OK*", pois eles são acoplados no mundo real (veja no modelo conceitual).

A seguir, uma vez que Pedido foi criado, o caso de uso precisa adicionar as linhas de pedidos a Pedidos. Quem deve ser responsável por adicionar as linhas de pedidos?

Uma solução seria Cliente adicionar as linhas (afinal ele possui as informações de inicialização necessárias: quantas linhas, que produtos, que quantidades, etc.). Essa solução é mostrada na Figura 8.11:

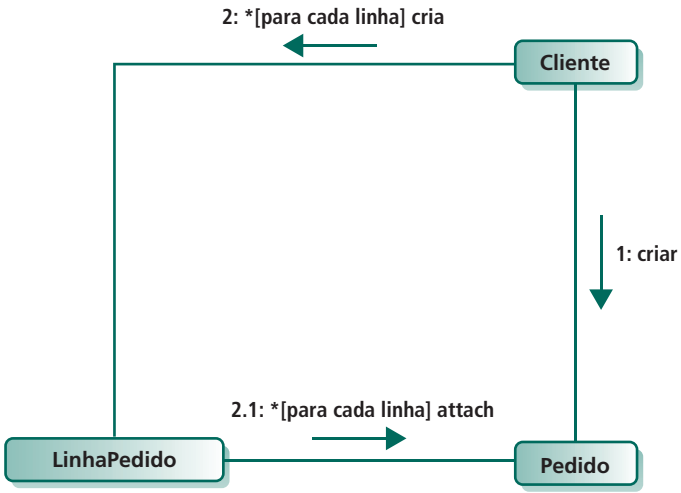


Figura 8.11: Solução 1
Fonte: Serafini (2008, p.108)

Essa solução aumentou o acoplamento artificialmente; temos agora todas as três classes dependentes umas das outras.

Se fizermos Pedido responsável pela criação das linhas de pedidos, faremos com que Cliente não tenha nenhum acoplamento com LinhaPedidos (Figura 8.12):

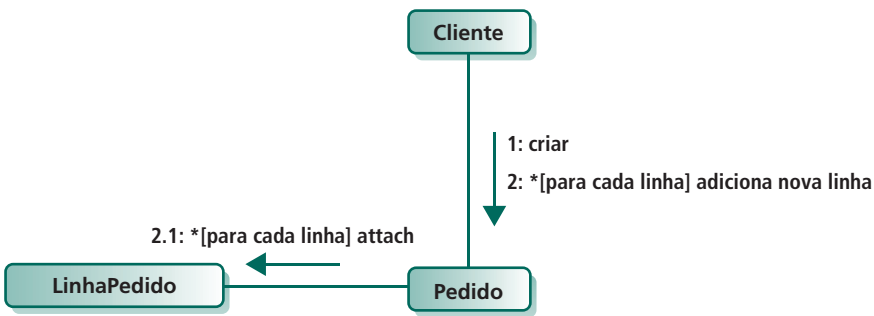


Figura 8.12: Solução 2
Fonte: Serafini (2008, p. 108)

Se a implementação da classe LinhaPedido sofrer alguma mudança, a única classe afetada vai ser a classe Pedido.

Todo acoplamento que existe nesse projeto foi identificado no modelo conceitual:

a) Clientes possuem Pedidos;

b) Pedidos possuem LinhaPedidos.

E faz muito sentido a existência desses acoplamentos.

Síntese:

Nome: "*LOW COUPLING*"

Problema: Como manter o acoplamento baixo?

Solução: Faça uma análise e procure reduzir o acoplamento ao mínimo (LARMAN, 2007).

8.4.2 Grasp 5: "*Controller*"

Vamos examinar um Sistema de Apostas de Jogos e o caso de uso Fazer Apostas.

Para que o usuário possa fazer as apostas, precisamos que o sistema permita entrar com as apostas e mostre os resultados dos jogos.

Como o usuário vai entrar com as apostas e como os resultados serão mostrados?

Vamos examinar a responsabilidade de mostrar os resultados de uma partida para o usuário: Que classe deve ser responsável por satisfazer esse requisito?

A aplicação do padrão "*Expert*", sugere que, como os detalhes pertencem a Partidas, então a classe Partida deve ser a "*Expert*" em mostrar os detalhes relevantes (Figura 8.13):

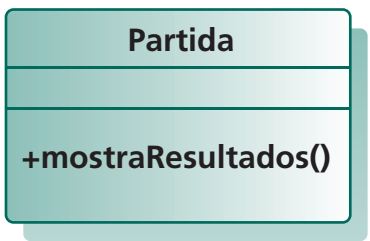


Figura 8.13: Classe Partida
Fonte: Serafini (2008, p.110)

Essa solução pode parecer boa, mas na realidade ela viola o padrão "*Expert*"!

A classe Partida deve ser especialista em todas as informações sobre as Partidas, e não ter a responsabilidade de mostrar os resultados das partidas na Interface Gráfica (GUI).

Geralmente, a inclusão de informações sobre a GUI, Bases de Dados, (ou qualquer outro objeto físico) em nossas classes resultará em um projeto pobre.

Imagine se tivéssemos 100 classes em nosso sistema e muitas dessas classes devessem ler e escrever na tela do computador. O que aconteceria se os requisitos do sistema mudassem e fosse necessário alterar toda a interface com o usuário? Teríamos que alterar cada classe envolvida com a interface com o usuário.

Fica muito melhor manter todas as classes do modelo conceitual "puras" – chamamos essas classes de Classes de Negócio" ou "*Business Classes*".

O padrão "*Controller*" é uma solução possível. Criamos uma nova classe que vai ficar entre o usuário e as classes de negócio. Essa classe funciona como uma ponte (ou controlador) entre a Interface Gráfica e as Classes do Domínio.

O nome dessa classe geralmente é:

<nomeDoCasoDeUSo>Handler.

Então, em nosso projeto, precisamos criar uma classe FazerApostasHandler (Figura 8.14):

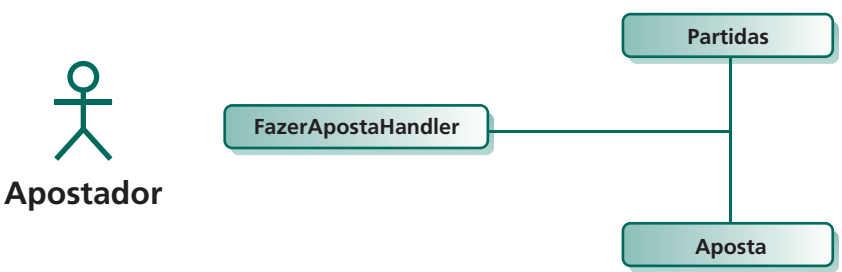


Figura 8.14: Padrão *"Controller"*
Fonte: Serafini (2008, p.111)

No caso de termos de alterar a interface com o usuário, a única classe que precisamos modificar é a classe controladora.

Síntese:

Nome: "*Controller*"

Problema: Como manter a GUI separada das classes de negócio?

Solução: Crie uma classe controladora para o sistema ou caso de uso.

(LARMAN, 2007)

Resumo

Com o estudo dos padrões apresentados nesta aula, aprendemos a definir responsabilidades a objetos dentro de nossos projetos que garantirão a melhoria constante de nossos "*softwares*". Os padrões apresentam soluções bem-sucedidas para vários problemas que comumente os desenvolvedores encontram no momento de definirem seus objetos e sua interação com os demais.

Os padrões propostos por Larman (2007) são conhecidos pelos desenvolvedores de todo o mundo e representam no meio profissional uma excelente solução para situações que enfrentamos no trabalho diário de projetar sistemas.

Além desses padrões, outros também são conhecidos no mercado profissional: são os padrões do GoF (""*Group of Four*"", na língua portuguesa, "Grupo dos Quatro"). São quatro autores de outros padrões que atendem também a outras classes de problemas de projeto, apresentando soluções muito interessantes para a construção de "*software*" de qualidade e efetividade. Seus padrões são apresentados em Gamma et al. (2000).

Atividades de aprendizagem

1. Explique qual o tipo de problema que o padrão "*Creator*" aborda e indique qual a solução que ele propõe.

2. Explique a diferença entre Responsabilidade e Colaboração. Cite um exemplo de cada conceito.

3. Escolha um dos padrões GRASP e desenhe um diagrama de classe (quatro classes no mínimo) representando uma solução proposta por esse padrão escolhido.

4. Pesquise na internet a diferença entre os conceitos de Acoplamento e Coesão, e indique se são relacionamentos indicados ou não para estarem em nossos projetos de "*software*".

Registre suas respostas num arquivo e poste-o no ambiente virtual.

Referências

BERTALANFY, Ludwig Von. **Teoria geral dos sistemas:** fundamentos, desenvolvimento e aplicações. Trad. Francisco M. Guimarães. 3. ed. Petrópolis: Vozes, 2008.

Diagramas de caso de uso. Disponível em: <http://www.youtube.com/watch?v=fGA-eF8HCjw&feature=related>. Acesso em: 19 set. 2011.

Diagrama de classes. Disponível em: <http://www.youtube.com/watch?v=F4fRIvPuZU8>. Acesso em: 19 set. 2011.

FRANCISCO FILHO, Egildo. **Entrevistas:** técnicas e dinâmicas de grupo. São Paulo: Qualitymark, 2008.

FOWLER, Martin. **UML essencial**: um breve guia para a linguagem-padrão. São Paulo: Bookman, 2004.

GAMMA, Erich et al. **Padrões de projeto**: soluções reutilizáveis de software orientado a objetos. Trad. Luiz A. Meirelles Salgado. Porto Alegre: Bookman, 2000.

Introdução a UML – 1a parte. Disponível em: <http://www.youtube.com/watch?v=hfN6n5fJfLc&feature=related>. Acesso em: 19 set. 2011.

LARMAN, Craig. **Utilizando UML e padrões**: uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo. Trad. Rosana Vaccare Braga et al. 3. ed. Porto Alegre: Bookman, 2007.

MARTINELLI, D. P.; VENTURA, C. A. A. (Org.). **Visão sistêmica e administração**: conceitos, metodologias e aplicações. São Paulo: Saraiva, 2006.

SERAFINI, J. I. **Análise de sistemas**. Serra: CEFETES, 2008.

SERAFINI, J. I.; CUNHA, L.E.C. **Análise e projeto de sistemas I**. Serra: CEFETES, 2010.

THE STANDISH GROUP. Disponível em: <http://standishgroup.com>. Acesso em: 19 set. 2011.

TEORIA geral dos sistemas. Disponível em: <http://www.youtube.com/watch?v=d_c8xvHtdHo>. Acesso em: 19 set. 2011.

TEORIA geral dos sistemas e tipologia dos sistemas de informação. Disponível em <http://www.youtube.com/watch?v=_EXOAh44oWg&feature=related>. Acesso em: 19 set. 2011b.

Currículo dos professores-autores

Luiz Egidio Costa Cunha

Graduado em Processamento de Dados pela FAESA e especialista em Análise de Sistemas e Avaliação pela Universidade de Brasília (UnB). Professor de cursos superiores de Computação e Sistemas há mais de 15 anos, com concentração de estudos nas áreas de Sistemas de Informação e Engenharia de "*Software*". Trabalha desde 1982 com análise, projeto e desenvolvimento de sistemas. Na área de educação superior, além de atuar como professor presencial e a distância, atuou como coordenador de cursos e núcleos de pesquisa e extensão universitária. Atua também como avaliador institucional do Instituto Nacional de Ensino e Pesquisa Inep/MEC em processos de avaliação para fins de credenciamento e recredenciamento de instituições de ensino presencial e a distância em todo o Brasil.

José Inácio Serafini

Engenheiro Civil formado pela Universidade Federal do Espírito Santo - UFES. Fez sua pós-graduação em Análise de Sistemas pela UFES. De 1980 a 2000, trabalhou no Núcleo de Processamento de Dados da UFES, como estagiário, programador de computadores e analista de sistemas. Desde 1993 é professor da ETEFES/CEFETES/IFES da Coordenadoria de Informática.

e-Tec Brasil
Escola Técnica Aberta do Brasil

ISBN 978-85-62934-03-2