Análise e Projeto de Software

Romualdo Rubens de Freitas

Cuiabá - MT

2015

Presidência da República Federativa do Brasil
Ministério da Educação
Secretaria de Educação Profissional e Tecnológica
Diretoria de Integração das Redes de Educação Profissional e Tecnológica

© Este caderno foi elaborado pela Universidade Tecnológica Federal do Paraná – PR, para a Rede e-Tec Brasil, do Ministério da Educação em parceria com a Universidade Federal de Mato Grosso.

Equipe de Revisão

Universidade Federal de Mato Grosso – UFMT

Coordenação Institucional
Carlos Rinaldi

Coordenação de Produção de Material Didático Impresso
Pedro Roberto Piloni

Designer Educacional
Alceu Vidotti

Diagramação
Tatiane Hirata

Revisão de Língua Portuguesa
Celiomar Porfírio Ramos

Revisão Final
Claudinet Antonio Coltri Junior

Universidade Tecnológica Federal do Paraná – PR

Curso Técnico de Informática

Coordenação Geral e-TEC
Edilson Pontarolo

Coordenação de Tecnologia na Educação
Henrique Oliveira da Silva

Coordenação de Curso
Maria Teresa Garcia Badoch

Projeto Gráfico
Rede e-Tec Brasil/UFMT

Apresentação Rede e-Tec Brasil

Prezado(a) estudante,

Bem-vindo(a) à Rede e-Tec Brasil!

Você faz parte de uma rede nacional de ensino, que por sua vez constitui uma das ações do Pronatec - Programa Nacional de Acesso ao Ensino Técnico e Emprego. O Pronatec, instituído pela Lei nº 12.513/2011, tem como objetivo principal expandir, interiorizar e democratizar a oferta de cursos de Educação Profissional e Tecnológica (EPT) para a população brasileira, propiciando caminho de acesso mais rápido ao emprego.

É neste âmbito que as ações da Rede e-Tec Brasil promovem a parceria entre a Secretaria de Educação Profissional e Tecnológica (Setec) e as instâncias promotoras de ensino técnico como os institutos federais, as secretarias de educação dos estados, as universidades, as escolas e colégios tecnológicos e o Sistema S.

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 ao promover o fortalecimento da formação de jovens moradores de regiões distantes, geograficamente ou economicamente, dos grandes centros.

A Rede e-Tec Brasil leva diversos cursos técnicos a todas as regiões do país, incentivando os estudantes a concluir o ensino médio e a realizar uma formação e atualização contínuas. Os cursos são ofertados pelas instituições de educação profissional e o atendimento ao estudante é realizado tanto nas sedes das instituições quanto em suas unidades remotas, os polos.

Os parceiros da Rede e-Tec Brasil acreditam em uma educação profissional qualificada – integradora do ensino médio e da educação técnica - 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
Agosto de 2015

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: remete o tema para outras fontes: livros, filmes, músicas, *sites*, programas de TV.

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.

Reflita: momento de uma pausa na leitura para refletir/escrever sobre pontos importantes e/ou questionamentos.

Palavra do Professor-autor

É uma grande satisfação apresentar a disciplina "Análise e Projeto de Software". É muito importante ressaltar o empenho que cada um deve ter para se tornar bom profissional. Sabemos que o mercado clama por profissionais de alto desempenho e que isso tem se tornado cada vez mais raro. Sabemos, também, que a competência não acontece por acaso. Ela é fruto de dedicação, de estudo, de trabalho, de vontade de aprender e de vencer. Espero que você esteja nesse pequeno time de pessoas que desejam fazer a diferença.

Tenho certeza de que você aproveitará ao máximo esta oportunidade de aprimoramento.

Um grande abraço,

Prof. Romualdo Rubens de Freitas

Apresentação da Disciplina

A disciplina de Análise e Projeto de Software se desenvolverá em 05 horas/aula e cada aula está organizada da seguinte forma:

I – Os objetivos específicos;

II – Apresentação do tema da unidade e sua importância na formação profissional;

III – O(s) tema(s) e sua(s) respectiva(s) seção(ões);

IV – O conteúdo e o(s) objetivo(s) atrelado(s) a ele;

V – O desenvolvimento conceitual do(s) tema(s);

VI – Ao final de cada seção, uma breve síntese do tema trabalhado;

VII – Uma ou duas atividades para fortalecimento da aprendizagem;

Desejamos que você tire o maior proveito possível desta disciplina e do curso como um todo e que saiba que pode contar conosco sempre que julgar necessário. Sobretudo, caro aluno, estimule em si próprio o hábito de leitura.

Bom curso e sucesso!

Sumário

Aula 1. Análise e projetos de software: bases tecnológicas 13
1.1 Histórico e evolução de engenharia de software 13
1.2 Papel do software 16
1.3 Características do software 16
1.4 Processo de software, intraestrutura do processo 17
1.5 Ciclo de vida 19
1.6 Codificar e consertar (Code-and-fix) 19
1.7 Modelo tradicional 20

Aula 2. Prototipação 23
2.1 Protótipos 23
2.2 Modelo incremental 25
2.3 Modelo espiral 26
2.4 Engenharia de requisitos 28
2.5 Ferramentas CASE (Computer-Aided Software Engineering) 31

Aula 3. Análise 33
3.1 Tarefas 33
3.2 Função de qualidade 34
3.3 Notações de modelagem 35
3.4 Análise estruturada 36

Aula 4. Análise orientada a objetos 43
4.1 Abordagem 43
4.2 Classes e objetos 44
4.3 Atributos 45
4.4 Operações, métodos e mensagem 45
4.5 Encapsulamento, herança e polimorfismo 46
4.6 Análise orientada a objetos 47

Aula 5. Projeto 53
5.1 Modelo de análise orientada a objetos 53
5.2 Conceito de pirâmide de projeto para software 54

5.3 Padrões de projeto 58

Palavras Finais **59**

Referências **60**

Aula 1. Análise e projetos de software: bases tecnológicas

Objetivos:

- entender o papel da engenharia de software no desenvolvimento de software; e
- compreender a evolução da engenharia de software e do processo de software.

Nessa aula vamos trabalhar a análise e projetos de softwares. Seja bem-vindo(a) ao nosso primeiro encontro.

Pois bem, quando falamos em engenharia já nos vem a noção de algo que deve ser criado, construído, analisado, desenvolvido e que se deve promover a manutenção. No nosso dia a dia é, perfeitamente, possível encontrar exemplos da engenharia em edifícios, casas, ruas, estradas, hidrelétricas, veículos e tantos outros. Cada um desses produtos da engenharia deve ser analisado para saber quais os componentes mais adequados à sua constituição, seus projetos devem ser criados e desenvolvidos e, em seguida, construídos e, uma vez prontos, precisam, de acordo com cada caso, sofrer manutenções adequadas para que suas durabilidades sejam asseguradas.

1.1 Histórico e evolução de engenharia de software

O termo Engenharia de Software foi criado pelo professor Friedrich Ludwig Bauer, hoje aposentado, na década de 1960, época em que ele lecionava na Universidade de Tecnologia de Munique, Alemanha, e, utilizado oficialmente em 1968 na Conferência sobre Engenharia de Software da OTAN (*NATO Conference on Software Engineering*). Segundo o professor Bauer, "Engenharia de Software é a criação e a utilização de sólidos princípios da engenharia a fim de se obter software de maneira econômica, que seja confiável e que trabalhe eficientemente em computadores reais".

Desde então, este termo é amplamente utilizado para identificar a área do conhecimento da informática cujo objetivo é o de especificar (descrever), desenvolver e promover a manutenção em sistemas de software (programas de computador), aplicando tecnologias e boas práticas (práticas consideradas as mais adequadas) provenientes das disciplinas de ciência da computação e gerência de projetos, entre outras, levando, assim, a produtividade e qualidade.

Sendo assim, pode-se dizer que a engenharia de software preocupa-se com os aspectos da construção de um sistema de software, desde o início de sua especificação até a manutenção, após ter sido posto em operação. De acordo com Sommerville (2003), há duas frases que descrevem esse conceito:

1) Disciplina de engenharia: os engenheiros fazem uso de teorias, métodos e ferramentas para tornar os produtos úteis, procurando soluções para problemas, mesmo que não existam teorias aplicáveis e métodos de auxilio;

2) Todos os aspectos da produção de software: não somente os aspectos técnicos são importantes, as atividades de gerenciamento de projetos e desenvolvimento de ferramentas, os métodos e as teorias de apoio à produção de software também são questões importantes dentro da engenharia de software.

Nos anos 40 os primeiros usuários de computadores escreviam código de máquina (instruções executadas pelos computadores) à mão. As primeiras ferramentas de software, tais como, macro *assemblers* (montadores de código) e interpretadores, foram criados nos anos 50 com o objetivo de melhorar a qualidade do código escrito e a produtividade dos programadores, surgindo, assim, a primeira geração de compiladores que aperfeiçoavam (otimizavam) o código de máquina. Nos anos 60 essas ferramentas evoluíram ainda mais permitindo uma maior qualidade nos programas, bem como uma produtividade mais alta por parte dos programadores.

Nesta época o conceito de engenharia de software começou a ser discutido tendo como propósito aplicar os conceitos de engenharia ao desenvolvimento de sistemas de software complexos, caracterizados, como pode ser observado a seguir:

> [...] por um conjunto de componentes abstratos de software (estruturas de dados e algoritmos) descritos na forma de procedimentos,

funções, módulos, objetos ou agentes interconectados entre si, compondo, assim, a arquitetura de software, e que devem ser executados em sistemas computacionais .

Disponível em: <http://pt.wikipedia.org/wiki/Engenharia_de_software> Acesso em: 7 out. 2013.

A partir de então, ferramentas colaborativas e repositórios de código apareceram na década de 1970. Nos anos 80 os computadores pessoais e as estações de trabalho (computadores com maior poder de processamento) tornaram-se comuns. Os profissionais de desenvolvimento de software também viram surgir nesta época a primeira linguagem de programação orientada a objetos comercial, denominada Smalltalk.

A programação orientada a objetos e os processos ágeis, como XP (*Extreme Programming* - Programação Extrema), na qual equipes pequenas ou médias desenvolvem software cujos requisitos não são claros ou que mudam constantemente, obtiveram uma maior aceitação nos anos 90. Ainda nos anos 90 os computadores de maior processamento tiveram uma queda acentuada de preços. A complexidade das aplicações de software cresceu mais com o avanço da tecnologia, a Internet e os dispositivos móveis permitiram que o software se tornasse cada vez mais disponível. A partir de 2000 surgiu o código gerenciado e plataformas interpretadas, tais como, Java, NET, Ruby, Python e PHP, proporcionaram facilidades na criação de aplicações de software.

A engenharia de software teve importância mais acentuada ainda com o advento do B2B (*Business to Business*) e do B2C (*Business to Commerce*).

Para saber mais, visite a Wikipédia em: http://pt.wikipedia.org/wiki/Engenharia_de_software#Processo_de_Software.

Atualmente, o conceito de Web 2.0, uma evolução no desenvolvimento de aplicação para Web, permitindo uma experiência de uso mais próxima as aplicações de software instaladas e executadas diretamente no computador do usuário e o conceito de Serviços Web (*Web Services*), no qual uma aplicação utiliza um serviço, um tipo de funcionalidade computacional, que foi criado por outra pessoa e que reside em algum computador conectado a Internet, são os mais recentes desafios da engenharia de software.

De acordo com Pressman, 2006, a engenharia de software é uma tecnologia em camadas. Todas as camadas, processos, métodos e ferramentas, têm como base o foco na qualidade do software desenvolvido.

1.2 Papel do software

Quanto mais as aplicações de software são utilizadas, mais elas facilitam a vida das pessoas. Terminais de autoatendimento (caixas eletrônicos), e-mail e software de mensagens instantâneas (p. ex. Microsoft Messenger – MSN e Yahoo Messenger), aparelhos eletroeletrônicos (televisores e fornos de microondas) podem receber as mais diversas programações realizadas pelos usuários através de um controle remoto ou em um painel digital, carros que realizam várias verificações no sistema elétrico e de injeção de combustível e emitem sinais alertando sobre o não uso do cinto de segurança, aviões capazes de realizar muitas de suas operações, praticamente, sem a intervenção humana, aplicações de software que interpretam e leem textos auxiliando portadores de deficiência visual são apenas alguns exemplos de como o software está inserido na vida cotidiana.

Quanto mais a tecnologia evolui, quanto mais a humanidade se globaliza, quanto mais atividades as pessoas procuram realizar, maior é o impacto do software na sociedade e na cultura. O software exerce papel importante na vida do ser humano ajudando, auxiliando, conduzindo e executando as mais diversas tarefas com maior eficiência (cada vez mais rápido) e eficácia (cada vez melhor). Assim, os profissionais de desenvolvimento de software procuram criar tecnologias que permitem a construção de softwares com maior produtividade e qualidade.

Conforme Pressman (2006), o software atualmente exerce dois papéis: 1) ele é um produto e 2) ele é um veículo para a entrega do produto. Como produto, é possível ter acesso ao potencial do computador ou rede de computadores. Independente da capacidade computacional do equipamento em que o software está instalado, seja em um telefone celular, em um equipamento doméstico, nos carros, nos aviões, em um computador pessoal ou em um computador de grande porte, o software age sobre a informação, transformando, gerando, adquirindo, modificando, exibindo ou transmitindo essa informação. Como veículo de entrega do produto, o software é utilizado no controle do computador (sistemas operacionais), na comunicação (redes de computadores) e na a criação de outros softwares (ferramentas e ambientes de desenvolvimento de software).

1.3 Características do software

Antes de entender quais as principais características de um software é preciso compreender o aspecto do processo de criação do ser humano. Quando

um produto qualquer é construído várias etapas se seguem: análise, projeto, desenvolvimento e teste. Uma vez que estas etapas tenham sido seguidas um produto passa a existir na forma física. Como exemplo, tem-se a construção de uma casa, uma vez que o engenheiro civil tenha conhecimento do local no qual a casa será erguida e os desejos do futuro morador, ele realiza uma análise para saber quais os materiais mais adequados para a construção, então constrói um esboço da casa na forma de projeto. Uma vez que o projeto esteja pronto, o mestre de obras é encarregado de orientar os pedreiros na construção da casa, ao mesmo tempo em que realiza verificações e testes para confirmar que a construção atende aos desejos do morador e as especificações do projeto. Assim que a construção é terminada, tem-se um produto físico, algo palpável.

Quando o assunto é software, a idéia de construção é semelhante a qualquer outro produto, porém com diferentes abordagens. O software faz parte de um sistema lógico e não de um sistema físico, conforme atesta Pressman (2006).

Com isto, as principais características de um software são:

1. O software é desenvolvido e não manufaturado;

2. Software não se desgasta;

3. Apesar da possibilidade de se desenvolver um software a partir de componentes pré-existentes, a maior parte das aplicações de software ainda é desenvolvida sob demanda, de acordo com as necessidades do usuário.

1.4 Processo de software, intraestrutura do processo

O processo de software é um esboço, uma estrutura inicial para as tarefas necessárias à construção de um software de alta qualidade. Sendo assim, cabe a seguinte indagação: processo de software é sinônimo de engenharia de software? Para esta questão há duas respostas, dependendo de como a ela é percebida. Tanto o processo de software quanto a engenharia de software descrevem como a construção de um software deve ser realizada. Porém, a engenharia de software vai além, ela acrescenta tecnologias (métodos técnicos e ferramentas automatizadas) que formam o processo.

Os métodos abrangem um conjunto de tarefas necessárias à construção de um software. Estas tarefas são análise de requisitos, projeto, construção de programas, testes e validações e manutenção. As ferramentas, que podem ser automatizadas ou semi-automatizadas, apoiam o processo e os métodos. Se a informação criada por uma ferramenta puder ser usada por outra se diz que a engenharia de software é auxiliada pelo computador ou, do inglês, CASE (*Computer Aided Software Engineering*). Uma ferramenta CASE pode ser apenas um software que inclua editores de projeto e dicionários de dados, por exemplo, ou uma combinação de software, hardware e um repositório de informações a respeito de análise, projeto, construção e testes. Ferramentas CASE são apresentadas mais adiante.

De acordo com a Figura 1, o processo de software é caracterizado como tendo uma estrutura comum de processo, aplicado a projetos de qualquer porte e complexidade; um conjunto de tarefas, que permite a aplicação das atividades de estrutura às características do projeto e às necessidades da equipe de projeto; e, por fim, atividades guarda-chuva, que são independentes de qualquer atividade de estrutura e acontecem durante todo o projeto.

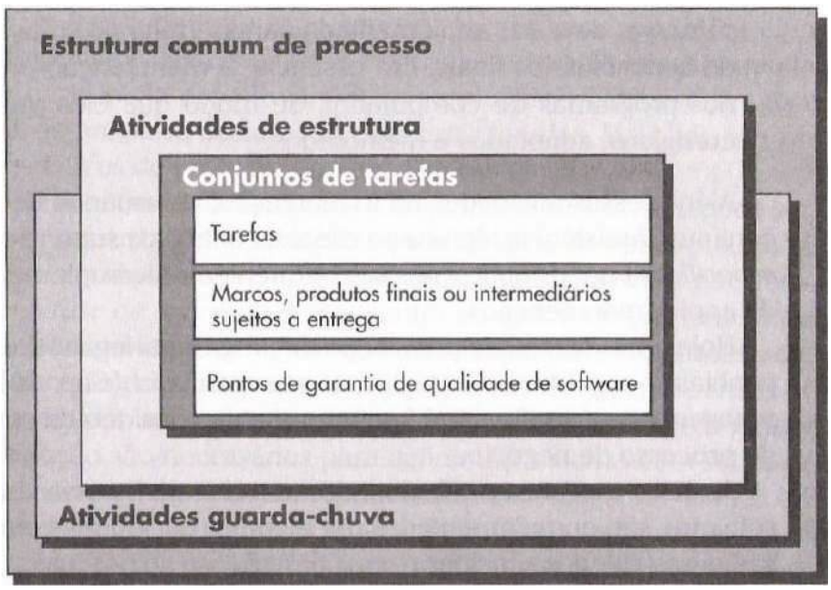


Figura 1: O processo de software
Fonte: Pressman (2006)

Para saber mais sobre o processo de desenvolvimento de software, consulte a Wikipédia (uma enciclopédia disponível na Internet – online), no endereço http://pt.wikipedia.org/wiki/Processo_de_desenvolvimento_de_software.

1.5 Ciclo de vida

O ser humano passa por um processo durante existência, um **ciclo de vida** que se inicia com o nascimento, passa pela adolescência, pela juventude e entra na fase adulta e, finalmente, a velhice nos alcança, deixando claro que o fim está se aproximando. Isto acontece com tudo e todos. Os seres vivos nascem, crescem e morrem; os seres inanimados (sem vida) são criados (pela natureza ou pelo homem), utilizados, se desgastam e se tornam inúteis. Um ditado bem conhecido e que expressa essa verdade diz: "Na vida a única certeza é a morte".

O desenvolvimento de software passa por processo semelhante. Ele é concebido, projetado, codificado e passa a ser utilizado para o fim a que se propõe. Ao contrário do ser humano, um software não se desgasta, mas pode sofrer modificações para que possa continuar atendendo aos seus usuários.

Um software passa por etapas distintas que podem ser resumidas em: *análise, projeto, codificação, teste e uso.* O ciclo de vida é utilizado para descrever o que acontece em cada uma destas etapas. O modelo de ciclo de vida é importante, pois com ele é possível determinar as etapas existentes, a ordem das atividades a serem desenvolvidas e quais os critérios a serem adotados para transição entre as etapas.

Os modelos de ciclo de vida vão desde o simples codificar e consertar (do inglês *code-and-fix*) até modelos mais complexos, como o espiral. Cada modelo possui características que o torna útil para determinada situação ou projeto de software, bem como, desvantagens inerentes ao próprio modelo.

A seguir, tem-se uma descrição de cada um dos modelos de ciclo de vida mais aplicados no desenvolvimento de aplicações de software, pois se constituem em modelos para sistemas práticos. Além dos modelos apresentados a seguir, outros modelos são descritos em Pressman (2006) e Sommerville (2003).

1.6 Codificar e consertar (*Code-and-fix*)

Nos primórdios do desenvolvimento de software o processo se concentrava nos programas e, normalmente, era realizada por uma pequena equipe ou, muitas vezes, por uma única pessoa. O processo é iniciado a partir de uma idéia geral do que se pretende construir e utiliza uma combinação de projeto e programação informal. Este modelo também ficou conhecido como

desenvolvimento artesanal ou *ad-hoc* e consistia em realizar a programação e, à medida que as inconsistências e erros eram encontrados, promoviam-se os acertos, sendo estes passos repetidos até que o projeto fosse concluído.

1.7 Modelo tradicional

Um dos modelos mais antigos e ainda utilizados, também conhecido como Modelo Cascata ou *Waterfall*. Consiste em um conjunto de fases que são executadas sequencialmente uma após a outra. A fase seguinte somente pode ser iniciada se a fase atual for completamente concluída. Por isso, ao final de cada fase uma revisão é realizada para se saber se a fase foi completada a contendo. Se houver alguma falha na fase atual, o projeto permanecerá nela até que os problemas sejam resolvidos.

A Figura 2, a seguir, apresenta das fases do modelo tradicional.

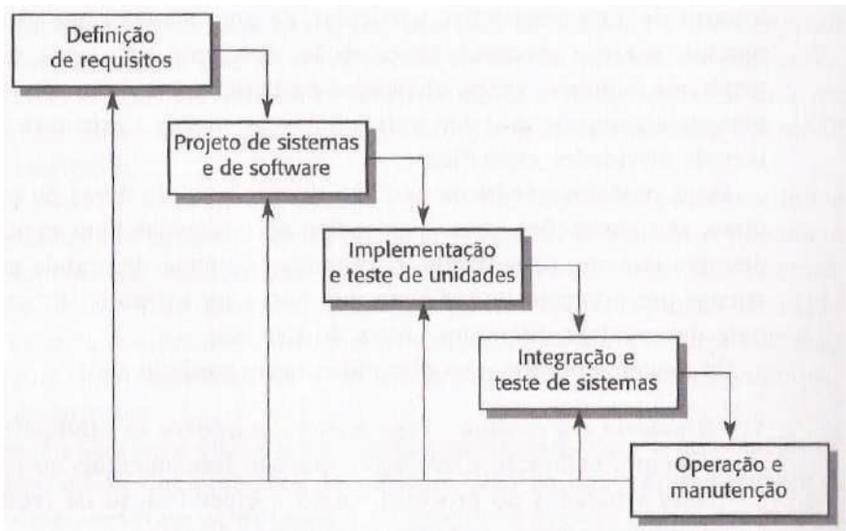


Figura 2 - Fases do modelo de ciclo de vida tradicional
Fonte:Sommerville (2003)

Como é possível verificar na Figura 2, as fases do modelo apresentam as principais atividades de desenvolvimento, assim descritas, de acordo com Sommerville (2003):

1. Análise e definição de requisitos: nesta etapa, diversas entrevistas são realizadas com os futuros usuários no intuito de se obter informações suficientes de tal forma que se chegue ao objetivo do software, ao conjunto de funcionalidades e suas restrições;

2. Projeto de sistemas e de software: o processo de projeto de sistemas estabelece uma arquitetura do sistema geral seja de hardware ou de software, enquanto o projeto de software se preocupa com a identificação e a descrição dos requisitos fundamentais do sistema de software;

3. Implementação e testes de unidades: uma vez que o projeto de software seja codificado, ele passe a ser compreendido por um conjunto de programas ou unidades de código. Os testes têm o objetivo de certificar cada unidade de código de acordo com sua especificação;

4. Integração e teste de sistemas: esta é a etapa em que as unidades de código são integradas, o sistema é testado como completo e finalizado para garantir que ele atenda os requisitos levantados junto aos usuários;

5. Operação e manutenção: ao término da fase de testes o sistema é implantado e posto em operação. Esta etapa pode ser longa, pois ela envolve em descobrir e corrigir falhas que não foram verificadas nas etapas anteriores.

Resumo

Aprendemos nesta aula o histórico e evolução da engenharia de softwares. Vimos, também, o seu papel, suas características e seu ciclo de vida. Trabalhamos, ainda, o seu processo, infraestrutura e codificação. Por fim, conhecemos o modelo tradicional.

Chegamos ao fim da nossa primeira aula. Espero que tenha aproveitado ao máximo esse nosso primeiro encontro sobre análise de projeto. Vamos para a segunda aula?

Aula 2. Prototipação

Objetivos:

- conhecer a classificação dos protótipos;
- entender o modelo incremental e espiral;
- compreender a engenharia de requisitos; e
- conhecer as ferramentas case.

Seja bem-vindo(a) à nossa segunda aula. Vamos conversar sobre prototipação? Então vamos...

Ao contrário do Modelo Tradicional em que cada etapa deve ser concluída e validada antes que a seguinte possa se iniciar, este modelo, também conhecido como Prototipagem, propõe a expansão, gradativa, do sistema através da análise, projeto e construção das várias partes do sistema. O resultado deste processo é conhecido como protótipo.

2.1 Protótipos

Protótipos são divididos em 4 (quatro) categorias: ilustrativo, no qual somente as telas são projetadas dando ao usuário uma idéia de como será a interface (“cara” do programa) final; simulado, que simplesmente simula o acesso ao sistema de persistência, possivelmente uma base de dados; funcional, neste protótipo são implementados apenas um subconjunto de todas as funcionalidades requeridas pelo software; evolucionário, este protótipo começa com uma versão menor do software que crescerá a medida que as próximas etapas forem concluídas.

O conjunto de etapas da prototipagem é apresentado na Figura 3.

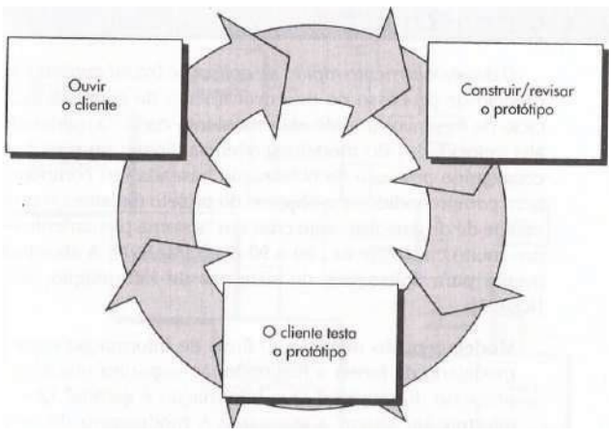


Figura 3 - Conjunto de etapas realizadas durante a prototipagem
Fonte: Pressman (2006)

O processo inicia-se com a definição dos requisitos, sendo que os objetivos do software são definidos e identificados, bem como identificadas aquelas áreas que merecem um maior refinamento. Uma vez levantados os requisitos, um projeto inicial é construído e apresentado ao usuário, que realiza a avaliação do protótipo. Desta forma, o usuário interage com o processo de desenvolvimento ajudando a refinar os requisitos e adquirindo uma maior confiança no software.

À medida que os protótipos são construídos, uma base de código é produzida. Assim, os próximos protótipos podem reusar o código já escrito. A reutilização de software é um conceito muito importante e se baseia não somente na reutilização de código escrito anteriormente, mas, também, na experiência adquirida na construção da base de código.

Em Sommerville (2003), verifica-se a existência de 3 (três) problemas do ponto de vista da engenharia e de gerenciamento para este modelo:

1. O processo não é visível: se o software é construído muito rapidamente não há como medir o progresso do desenvolvimento;

2. Os sistemas são frequentemente mal estruturados: acrescentar modificações pode se tornar difícil e de alto custo à medida que o software cresce;

3. Exigência de ferramentas e técnicas especiais: ferramentas e técnicas que auxiliam na produtividade permitindo um desenvolvimento mais rápido podem não ser compatíveis com outras ferramentas ou técnicas, bem como o número de pessoas especializadas com conhecimento suficiente nestas

ferramentas ou técnicas podem não ser suficientes.

Apesar de Pressman (2006), levantar problemas semelhantes, percebe-se, ao mesmo tempo, que este modelo pode se tornar um bom modelo para a engenharia de software, desde que as regras entre desenvolvedor e o cliente sejam claras e bem definidas, no sentido de que o protótipo será usado com o objetivo de validar os requisitos levantados e que ele poderá ser descartado em todo ou em parte para que o software real seja construído à luz da engenharia buscando, com isso, qualidade, confiabilidade e manutenibilidade.

2.2 Modelo incremental

Este modelo caracteriza-se por ser evolucionário e os modelos evolucionários são interativos. Assim, os engenheiros de software podem produzir versões mais completas do software, pois à medida que ele evolui, seus requisitos sofrem modificações impossibilitando o uso direto do modelo tradicional, que considera o software um produto final após suas etapas serem cumpridas, nem tampouco o uso da prototipagem, pois este modelo tem o objetivo de ajudar o cliente na compreensão das suas reais necessidades.

O modelo incremental é também considerado híbrido, pois as etapas de especificação, projeto e codificação são divididas em um conjunto de estágios, cada um sendo desenvolvido após o outro, lembrando o modelo tradicional, e, de forma semelhante a prototipagem, permite produzir um produto parcial, mas ao contrário desta, o produto produzido é completamente funcional e, dificilmente, será descartado. A aplicação do modelo incremental pode ser observada na Figura 4.

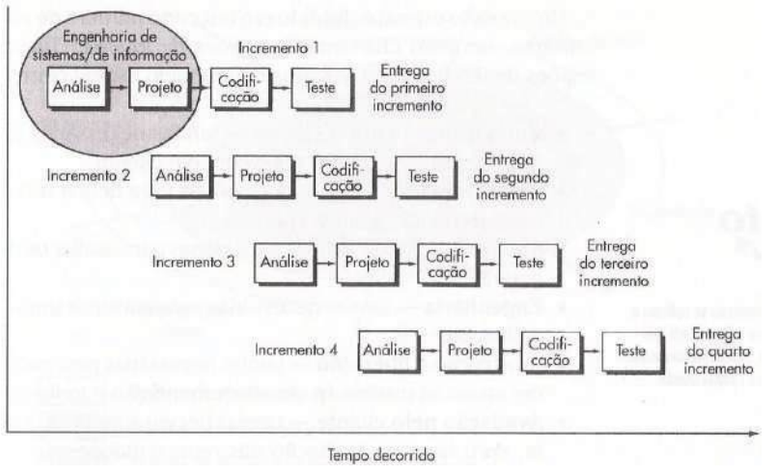


Figura 4 - O Modelo incremental
Fonte: Pressman (2006)

Conforme Pressman (2006), quando este modelo é utilizado o primeiro incremento é denominado núcleo do produto e este contempla os requisitos básicos, então o produto é entregue ao cliente para validação ou, simplesmente, sofre uma revisão mais detalhada.

Quando o conjunto de requisitos para um incremento está bem definido, o modelo tradicional é utilizado. Porém, quando os requisitos não são completos ou possuem definição parcial, o modelo de prototipagem pode ser utilizado.

Sommerville (2003) aponta vantagens e problemas dessa abordagem. Como vantagens, destacam-se:

1. O primeiro estágio, denominado núcleo do produto por Pressman, atende os requisitos mais importantes do usuário;

2. Os primeiros incrementos podem levar a novos requisitos ou requisitos mais refinados para os próximos incrementos;

3. O software como um todo atenderá as necessidades do usuário. Mesmo que alguns incrementos apresentem falhas, outros, provavelmente a maioria, estará de acordo com os requisitos;

4. Normalmente, os requisitos mais importantes são implementados primeiro permitindo, assim, que as partes mais importantes sejam testadas exaustivamente, possibilitando ao usuário maior confiabilidade no software.

Os problemas são provenientes da dificuldade de se mapear todos os requisitos dentro de um incremento, bem como perceber requisitos que se aplicam a funcionalidades básicas utilizadas por várias partes do software já que os detalhes serão definidos somente quando o incremento estiver desenvolvido, tornando difícil a identificação de tais funcionalidades comuns a todos os incrementos.

2.3 Modelo espiral

Da mesma forma que o modelo incremental, o modelo espiral também é interativo. Entretanto, este modelo é representado por uma série de espirais, nas quais cada espiral descreve uma fase do processo de software.

Sommerville (2003), classifica cada espiral em 4 (quatro) setores, conforme mostra a Figura 5.

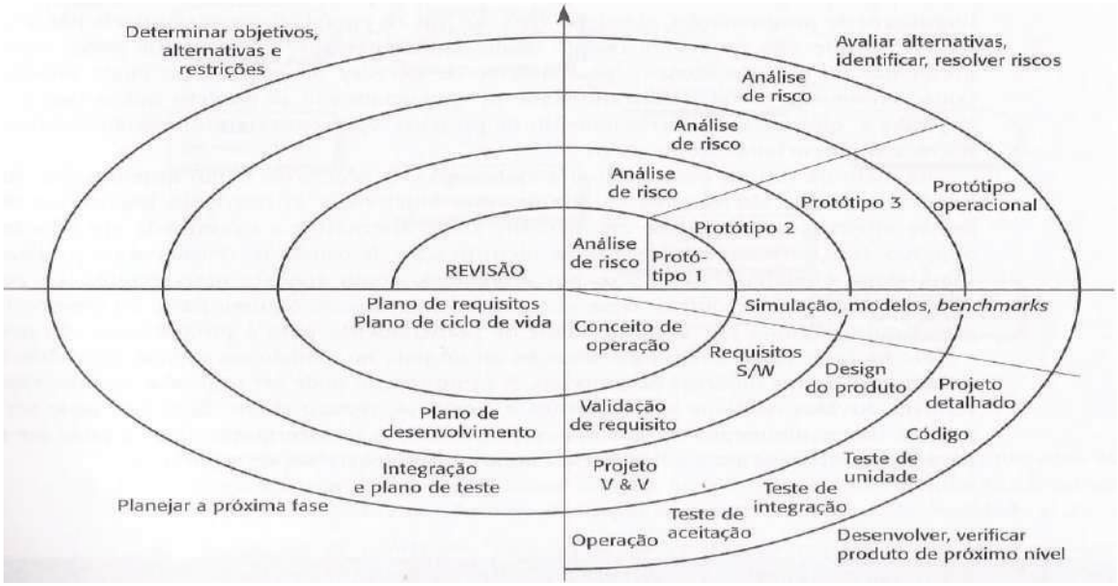


Figura 5 - O Modelo espiral na visão de Ian Sommerville
Fonte: Sommerville (2003)

A Figura 6 apresenta a visão de Pressman (2006), mostrando que este modelo é dividido em atividades, formando estruturas, denominadas regiões de tarefas, que são divididas entre 3 (três) a 6 (seis) regiões.

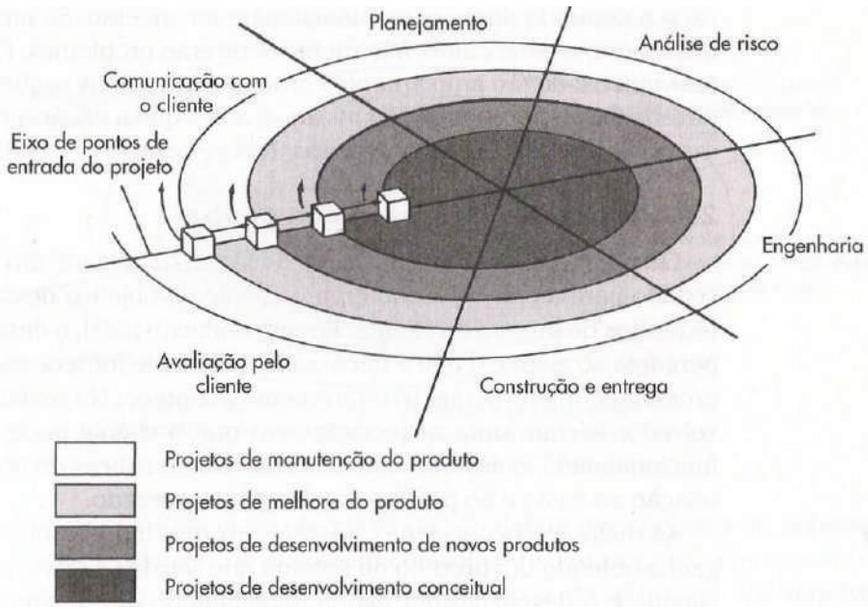


Figura 6 - O Modelo espiral sob a ótica de Roger S. Pressman
Fonte: Pressman (2006)

De qualquer forma, ambos concordam que este modelo é mais bem aplicado a projetos de médio para grande porte, nos quais os riscos, que significa simplesmente algo que pode acontecer de errado, podem acontecer com maior probabilidade.

2.4 Engenharia de requisitos

Ao final do processo de engenharia de software é obtida a especificação de um produto de software. Esta especificação deve garantir que as necessidades e expectativas do cliente e dos usuários do sistema sejam atendidas. Esta garantia é dada pela engenharia de requisitos que, se bem conduzida, fornecerá subsídios, na forma de documentos de requisitos, para garantir que a especificação está em acordo com o que se espera do produto final de software.

O processo de engenharia de requisitos é composto por um conjunto de atividades que visa criar e manter o documento de requisitos do software.

Sommerville (2003), destaca que as atividades do processo de engenharia de requisitos são divididas em 5 (cinco) passos:

1. Obtenção de requisitos;

2. Análise e negociação de requisitos;

3. Especificação de requisitos;

4. Modelagem do sistema;

5. Validação de requisitos.

Pressman (2006), por outro lado, descreve 4 (quatro) atividades genéricas no processo de engenharia de requisitos:

1. Estudo de viabilidade do sistema;

2. Obtenção e análise de requisitos;

3. Especificação de requisitos e sua documentação;

4. Validação de requisitos.

A interação entre estas atividades é ilustrada na Figura 7.

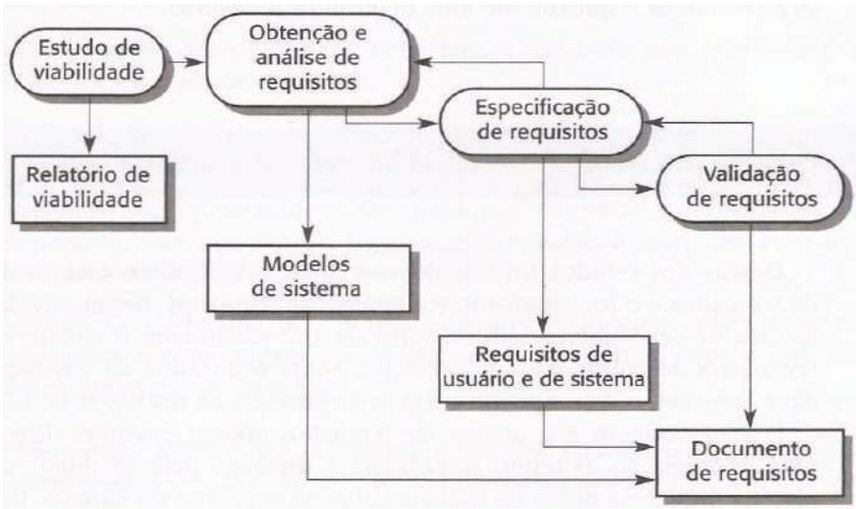


Figura 7 - Interação das atividades no processo de engenharia de requisitos
Fonte: Pressman (2006)

Ambos apresentam, praticamente, as mesmas descrições para as atividades que devem ser realizadas com o intuito de se chegar a uma documentação ou conjunto de documentos que garantam que a especificação obtida no processo de engenharia de software resultará em um produto de software adequado aos seus usuários. Notadamente Sommerville (2003), acrescenta a atividade de modelagem do sistema e diferencia a obtenção da análise de requisitos, enquanto Pressman (2006) reforça que um estudo de viabilidade é importante.

Conforme dito anteriormente, o resultado da engenharia de requisitos é um ou mais documentos de requisitos que garantam que a especificação do sistema satisfaz as necessidades do cliente e dos usuários. A garantia de que a especificação atenda tanto cliente quanto usuários é realizada por meio de várias atividades já relacionadas. A seguir apresentam-se as descrições das atividades de ambos os autores.

1. Estudo de viabilidade do sistema: com base em uma descrição geral do sistema e de sua utilização, entrevistas são realizadas com os responsáveis pela organização e os futuros usuários com a intenção de se obter um relatório que indicará se o sistema a ser desenvolvido é ou não viável do ponto de vista a contribuir com os objetivos da empresa;

2. Levantamento de requisitos: esta atividade, aparentemente simples, se resume em realizar entrevistas com o cliente e os usuários na tentativa de se obter informações necessárias sobre o domínio do problema (para atender que tipo de problema o sistema será desenvolvido). Entretanto, alguns problemas podem se tornar obstáculos na busca de tais informações: 1) problemas de escopo: o objetivo do sistema é mal definido ou os usuários contribuem com informações irrelevantes; 2) problemas de entendimento: os usuários não têm certeza sobre o que é realmente necessário ou têm dificuldade em expressar as verdadeiras necessidades; e 3) problemas de volatilidade: os requisitos mudam com constantemente;

3. Análise de requisitos: o objetivo desta atividade resume-se em categorizar os requisitos, explorando a relação entre eles, e, agrupá-los em subconjuntos para obter respostas a algumas perguntas, tais como: 1) Cada requisito atende aos objetivos do sistema? 2) Há conflito entre requisitos? e 3) Ao ser implementado, um determinado requisito pode ser testado?

4. Especificação de requisitos: a palavra especificação pode ter diferentes significados em diferentes situações e para diferentes pessoas. Em se tratando de sistemas de software a especificação de requisitos descreve a função e o desempenho do sistema e as restrições aplicadas ao seu desenvolvimento;

5. Modelagem do sistema: no contexto da engenharia de requisitos desenvolver um modelo do sistema é importante, pois permite que a "estética" e relação entre os vários componentes e como eles se encaixam e possa ser avaliada;

6. Validação de requisitos: esta é a atividade na qual a especificação é avaliada para garantir que os requisitos tenham sido declarados de acordo com o objetivo final do produto de software e que possíveis falhas tenham sido detectadas e sanadas.

Uma vez que essas atividades tenham sido cumpridas, o resultado será uma especificação das características operacionais do software (sua função, seus dados e comportamento), em como o software se relacionará com outras partes do sistema e quais restrições que o software deve satisfazer.

2.5 Ferramentas CASE (*Computer-Aided Software Engineering*)

Quando se trata de software, o termo ferramenta descreve um programa de computador que, de alguma maneira, auxilia o usuário na execução de suas tarefas. Assim, ferramentas CASE são um conjunto de programas de computador construídos para auxiliar engenheiros de software no processo de desenvolvimento de sistemas. Estas ferramentas podem ser simples, como um ambiente integrado de desenvolvimento (IDE – *Integrated Development Environment*), que permite a escrita de programas, até as mais complexas, que fornecem opções para realizar a análise de requisitos de modelagem, além da programação e testes. Ademais, ferramentas CASE podem ser utilizadas para o gerenciamento de configuração, modelagem de dados, transformações de modelos, bem como fornecer editores para diagramas de fluxo de dados e de entidade-relacionamento, entre outros. A Figura 8 apresenta um modelo genérico de ferramenta CASE para análise e projeto.

Figura 8 - Modelo de ferramenta CASE para análise e projeto
Fonte: Sommerville (2003)

Uma organização americana (ISO – I*nternational Standardization Organization*) responsável por diversos padrões internacionais estabeleceu a norma ISO/IEC 14102 que fornece diretrizes para seleção e avaliação de ferramentas CASE, cobrindo parcial ou totalmente o projeto de sistemas de software. Mais informações sobre esta norma pode ser obtida na página oficial da organização:**http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=23593**

As ferramentas CASE aumentam a produtividade no processo de software, melhoram a qualidade do produto final, agilizam o tempo na tomada de decisão, diminuem a quantidade de código escrito manualmente e reduzem os custos de manutenção de software. Por outro lado, há uma grande incompatibilidade entre ferramentas criadas por diferentes empresas, muitas vezes impossibilitando utilizar as informações geradas por uma ferramenta como entrada para outra ferramenta, além do alto custo de aquisição e de treinamento que essas ferramentas exigem.

Resumo

Nessa segunda aula, vimos a classificação dos protótipos e os modelos incremental e espiral. Trabalhamos, também, a engenharia de requisitos e Conhecemos as ferramentas case.

Aula 3. Análise

Objetivos:

- compreender a função da qualidade;
- conhecer as notações de modelagem; e
- entender o que é uma análise estruturada.

Olá! Chegamos ao nosso terceiro encontro. Chegou o momento de trabalharmos os princípios da análise, indo do entendimento da função da qualidade à análise estruturada. Boa aula!

Anteriormente, viu-se que o papel da engenharia de requisitos dá-se por um conjunto de atividades que resultam na especificação do software. Na etapa da análise essa especificação é refinada pelo engenheiro de software ou analista, como é comumente conhecido, mediante a elaboração de modelos de dados, funcional e comportamental.

3.1 Tarefas

Esta é uma etapa que se encontra entre a engenharia de requisitos e o projeto do software, conforme mostra a Figura 9.

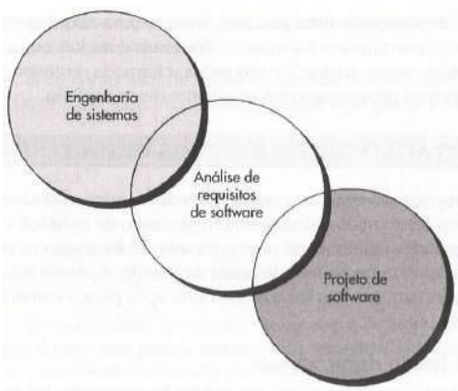


Figura 9 - Etapa de análise entre a engenharia de requisitos e o projeto de software
Fonte: Pressman (2006)

Durante a análise, os objetos de dados são definidos, o fluxo e o conteúdo da informação são avaliados, o comportamento do software perante o sistema deve ser compreendido, as interfaces do sistema são estabelecidas e quais restrições extras precisam ser descobertas.

Uma vez que estas tarefas tenham sido cumpridas, o analista inicia o detalhamento de uma ou mais soluções, iniciando pelos objetos de dados, funções de processamento e o comportamento do sistema. Enquanto este detalhamento é conduzido o analista se preocupe com o "que" e não com o "como". "Que" dados o sistema processa, "que" funções o sistema possui e "que" comportamentos são desempenhados. Para que isto seja possível, o analista lança mão de modelos que o ajudam a ter uma melhor compreensão dos dados e do fluxo de controle, o processamento funcional, o comportamento operacional e o conteúdo da informação.

Antes que os requisitos sejam analisados, modelados ou especificados eles precisam ser levantados e isto é feito mediante várias entrevistas ou reuniões, como foi dito anteriormente. Algumas vezes, no entanto, estas entrevistas ou reuniões não têm o resultado esperado, seja por deficiência de conhecimento sobre o domínio do problema; o futuro ou os futuros usuários querem uma solução computadorizada para um problema existente, mas não sabem como se expressar ao certo.

Para amenizar a distância e os empecilhos existentes entre usuários e analistas, uma abordagem baseada em equipe foi criada para uso nos primeiros encontros. A FAST (*Facilitated Application Specification Techniques* – Técnicas Facilitadas de Especificação de Aplicação) sugere a criação de uma equipe entre usuários e desenvolvedores para que ambos consigam elencar problemas e propor soluções.

3.2 Função de qualidade

Outra técnica utilizada para se obter os requisitos de software baseia-se em levantar as necessidades do usuário. O Desdobramento da Função de Qualidade (QFD – *Quality Function Deployment*) objetiva levantar o que realmente é valorizado pelo usuário e, em seguida, esses valores são desmembrados aplicando-se o processo de engenharia.

O QFD identifica 3 (três) tipos de requisitos: *requisitos normais*: objetivos e metas estabelecidos durante reuniões com o usuário, como

funções específicas do software, por exemplo; *requisitos esperados*: apesar de não serem claros para o usuário, estes requisitos estão implícitos no software, como, por exemplo, facilidade no uso do software; e *requisitos excitantes*: descrevem características que o software possui e que vão além das expectativas do usuário, como a capacidade que os usuários tem de enviar mensagens a outros usuários do software, por exemplo.

No transcorrer das reuniões, com FAST ou QFD, os requisitos são levantados e permitem ao analista criar cenários, conhecidos como casos de uso, dando ao analista uma visão de como o sistema será executado. Ao criar os casos de uso o analista identifica quem ou o quê se comunicará com o sistema. As pessoas, dispositivos e outros sistemas que interagem com o sistema em construção são denominados atores. E a interação entre atores e sistema é exercida por papéis. Usuários e atores possuem papéis diferentes, enquanto um usuário pode desempenhar diversos papéis, um ator representa um conjunto de entidades externas ao sistema e desempenha apenas um papel.

Da mesma forma que os requisitos não são todos elencados nas primeiras reuniões, o mesmo acontece com os atores. Inicialmente, os atores principais são identificados como sendo aqueles que utilizam o software diretamente com o intuito de obter a função desejada; ao passo que os atores secundários, identificados posteriormente, são aqueles que fornecem suporte ao sistema, permitindo que os atores principais realizem seu trabalho.

Após a identificação dos atores, os casos de uso são desenvolvidos e mostrarão como um ou mais atores interagem com o sistema. Normalmente, eles são criados na forma de uma descrição narrativa da interação entre atores e o sistema.

3.3 Notações de modelagem

Ao longo do tempo várias notações de modelagem foram criadas com o objetivo de auxiliar o analista. No entanto, cada uma apresenta seu próprio ponto de vista sobre esta etapa, porém, todas se relacionam através de um conjunto de princípios:

1. O domínio da informação de um problema deve ser compreendido e representado;

2. As funcionalidades do software precisam ser definidas;

3. O comportamento software, de acordo com eventos externos, necessita de uma representação;

4. Os modelos representativos de informação, função e comportamento devem ser particionados em detalhes na forma de camadas;

5. A análise parte da informação essencial até a implementação.

Para maiores detalhes, consultar o Capítulo 11 de Pressman (2006).

Após o levantamento e a análise de requisitos do software a ser construído, o analista desenvolve um modelo de análise, considerado a primeira representação técnica do software. Este modelo ou, na verdade, modelos são abordados por diversos métodos propostos no decorrer do tempo. Entretanto, dois métodos tornaram-se dominantes: a Análise Estruturada e a Análise Orientada a Objetos.

3.4 Análise estruturada

A Análise Estruturada é um método de modelagem clássico. Nesta abordagem o modelo de análise objetiva: 1) descrever as necessidades do usuário; 2) estabelecer critérios para a criação de um projeto de software; e 3) definir um conjunto de requisitos a ser validado quando o software for construído. Com estes objetivos, o modelo de análise é obtido por meio de ferramentas utilizadas durante a etapa de análise, sendo que a Figura 10 apresenta as principais.

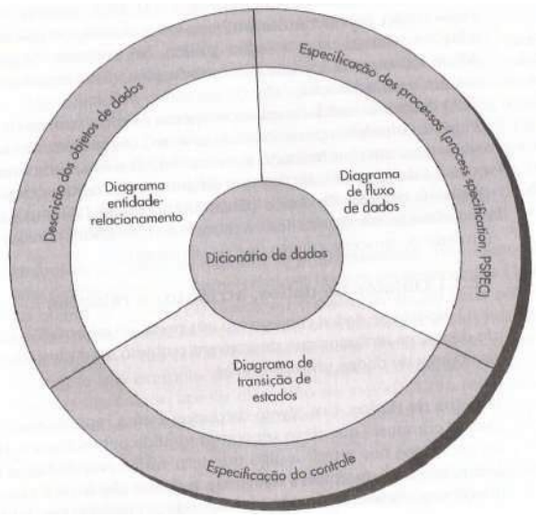


Figura 10 - Ferramentas da análise estruturada
Fonte: Pressman (2006)

O Dicionário de Dados (DD) é o pilar desta técnica de modelagem, por meio desta ferramenta é descrito o controle, todas as funções e todos os objetos (ou itens) de dados criados ou usados pelo software. Mesmo não tendo unanimidade entre as ferramentas CASE de análise e projeto estruturado, o DD é composto pela seguinte informação:

- Nome: nome principal do item de dado, controle, depósito de dados ou entidade externa;
- Sinônimo: outros nomes pelos quais o item de dado seja conhecido;
- Onde usado/como usado: processos nos quais o item de dado ou controle é usado e como ele é usado, como, por exemplo, entrada, saída, depósito de dados ou entidade externa;
- Descrição do conteúdo: descrição do conteúdo do item de dado;
- Informação suplementar: informações extras sobre o tipo do dado, valores iniciais, restrições e limitações, entre outros.

O Diagrama Entidade-Relacionamento (DER) é utilizado para modelar a relação entre 3 (três) partes: objetos (itens) de dados, atributos que descrevem os objetos de dados e as relações existentes entre os objetos de dados. A Figura 11 apresenta um exemplo de DER.

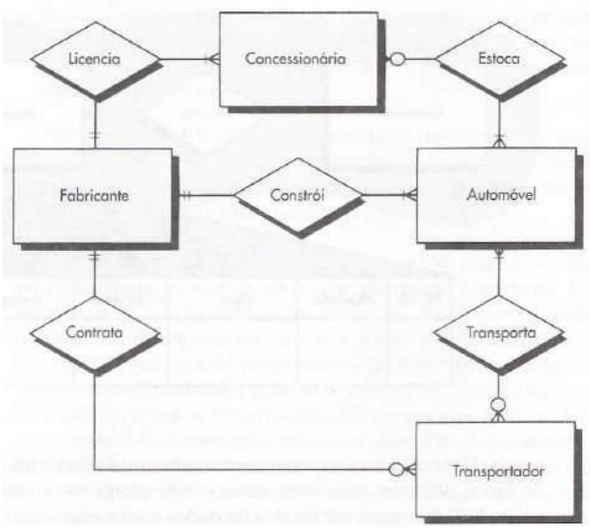
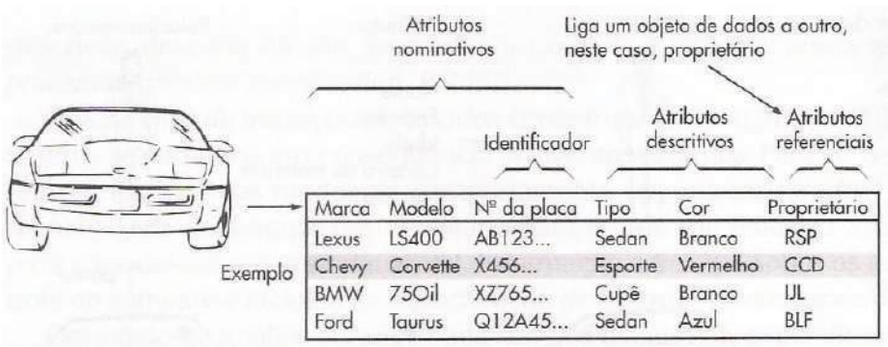


Figura 11 - Exemplo de diagrama entidade-relacionamento
Fonte: Pressman (2006)

Os objetos de dados representam, praticamente, toda a informação que precisa ser entendida pelo software, podendo ser uma entidade externa (algo ou alguém que produza ou consuma informação), algo (um relatório), uma ocorrência (um software cliente pede informação a um software servidor) ou um evento (um alarme), entre outros. Por exemplo, um **aluno** está matriculado em uma **disciplina**. Um objeto de dados refere-se apenas a dados, não havendo qualquer indicação sobre operações que agem sobre os dados, o que é parte integrante do Paradigma Orientado a Objetos, abordado em Análise Orientada a Objetos.

As propriedades que definem os objetos de dados são descritas pelos atributos. Como exemplo, tem-se o objeto de dados **automóvel**, ilustrado na Figura 12, que pode ser descrito, entre outros, pelos atributos **número da placa** e **tipo**.



Marca	Modelo	Nº da placa	Tipo	Cor	Proprietário
Lexus	LS400	AB123...	Sedan	Branco	RSP
Chevy	Corvette	X456...	Esporte	Vermelho	CCD
BMW	750il	XZ765...	Cupê	Branco	IJL
Ford	Taurus	Q12A45...	Sedan	Azul	BLF

Figura 12 - Representação do objeto de dados automóvel
Fonte: Pressman (2006)

Quando é necessário obter um automóvel em específico, usa-se um atributo especial, denominado identificador. Algumas vezes, identificadores têm valores únicos, o que não é obrigatório.

As relações ou relacionamentos conectam os objetos de dados uns aos outros. No modelo de dados é preciso que haja uma representação do número de ocorrências dos objetos em uma relação. Esta representação define a quantidade de ocorrências que dois objetos de dados quaisquer podem ter em uma relação. Por exemplo, uma nota fiscal de venda relaciona-se apenas com um cliente; um automóvel é composto de vários pneus; e um aluno está matriculado em diversas disciplinas, enquanto estas possuem vários alunos matriculados. Assim, o número máximo de objetos de dados que podem participar em um relacionamento é conhecido como cardinalidade.

Além da cardinalidade, é possível expressar quando a participação de um objeto de dados em uma relação é opcional ou obrigatória. Por exemplo, um funcionário pode ter ou não dependentes (opcional), entretanto, um funcionário deve possuir um endereço (obrigatório). Em casos como estes, a modalidade é utilizada para expressar a obrigatoriedade ou não de um objeto de dados em uma determinada relação. A Figura 13 apresenta um exemplo entre **cliente** e **ação de reparo**. Neste exemplo um cliente se relaciona com zero ou várias ocorrências de ação de reparo e uma ação de reparo, quando existir, deve se relacionar com somente um cliente.

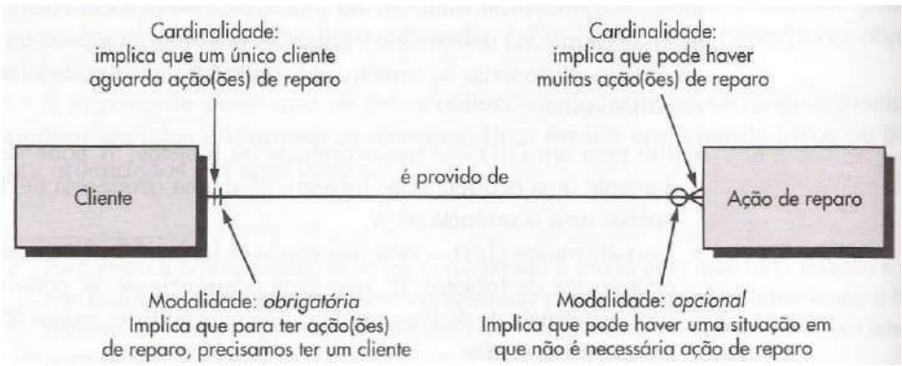


Figura 13 - Cardinalidade e modalidade
Fonte: Pressman (2006)

À medida que os dados trafegam pelo sistema eles sofrem algum tipo de transformação e essa transformação é realizada pelas funcionalidades existentes no software. Essas são características modeladas com o auxílio do Diagrama de Fluxo de Dados (DFD), que é uma representação gráfica demonstrando o fluxo da informação e as transformações sofridas pelos dados à medida que os mesmos se movem da entrada para a saída. A Figura 14 apresenta um exemplo genérico de DFD.

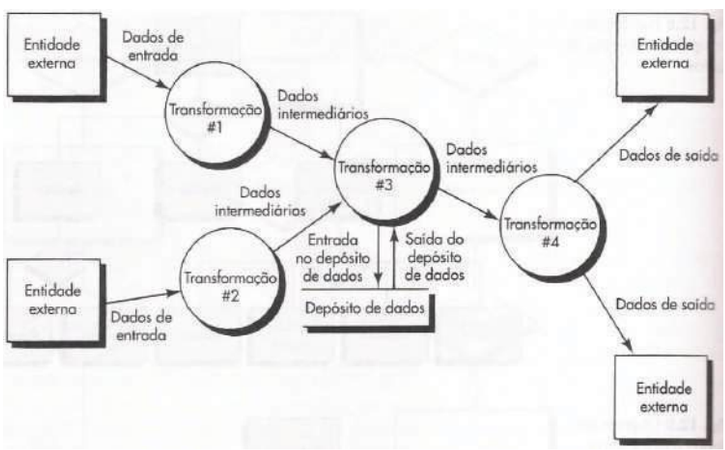


Figura 14 - Modelo de diagrama de fluxo de dados
Fonte: Pressman (2006)

Na Figura 14 percebe-se que o diagrama é construído de quadrados para demonstrar entidades externas, bolhas ou círculos para demonstrar processos ou transformações, e setas, que devem ser rotuladas, para identificar itens de dados e retângulos ou linhas duplas para demonstrar o armazenamento de dados.

Um DFD pode ser dividido, criando, assim, um DFD para cada nível de detalhe que seja necessário ou que se queira expor. Com o particionamento do DFD tem-se um mecanismo para a realização da modelagem funcional, bem como para a modelagem do fluxo de informação. No nível mais alto ou abstrato, o DFD, chamado de DFD 0 (zero) ou DFD Nível 0, é conhecido como Modelo de Contexto ou Diagrama de Contexto, que representa todo o software como uma única bolha, com setas indicando a entrada e saída dos dados, conforme ilustra o exemplo da Figura 15.

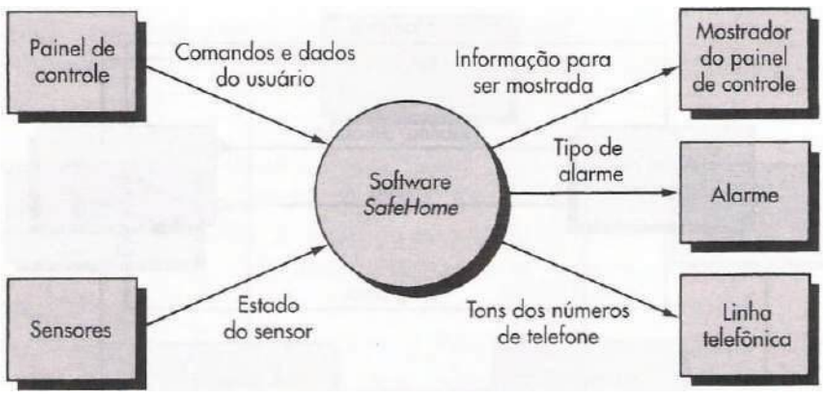


Figura 15 - DFD Nível 0 para o sistema *SafeHome*
Fonte: Pressman (2006)

O DFD que representa o conjunto de entidades externas, funcionalidades e depósitos é denominado DFD 1 ou DFD Nível 1 e um exemplo é apresentado na Figura 16.

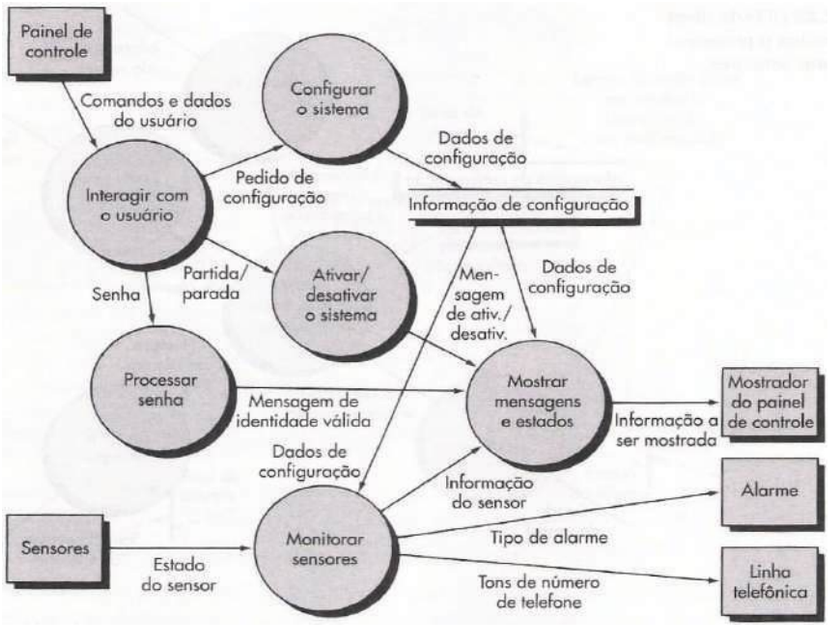


Figura 16 - DFD Nível 1 para o sistema *SafeHome*
Fonte: Pressman (2006)

Após este primeiro particionamento, cada novo detalhe que se deseja expressar o será em um novo DFD, lembrando, sempre, que a continuidade do fluxo de informação deve ser mantida. Em outras palavras, cada particionamento é apenas uma expressão detalhada de um processo no nível anterior.

A descrição de cada uma das funcionalidades a serem executadas pelo software é modelada através de uma Especificação de Processos, que é um texto descritivo detalhando cada processamento modelado no DFD, através das bolhas.

Finalmente, o Diagrama de Transição de Estados (DTE) possibilita ao analista modelar como o software reage a eventos externos. Esta modelagem representa os diversos comportamentos, conhecidos como estados, do software e a forma como é realizada a transição entre eles. Como exemplo tem-se o software de controle de uma máquina fotocopiadora. A Figura 17 ilustra a representação gráfica do DTE para este exemplo.

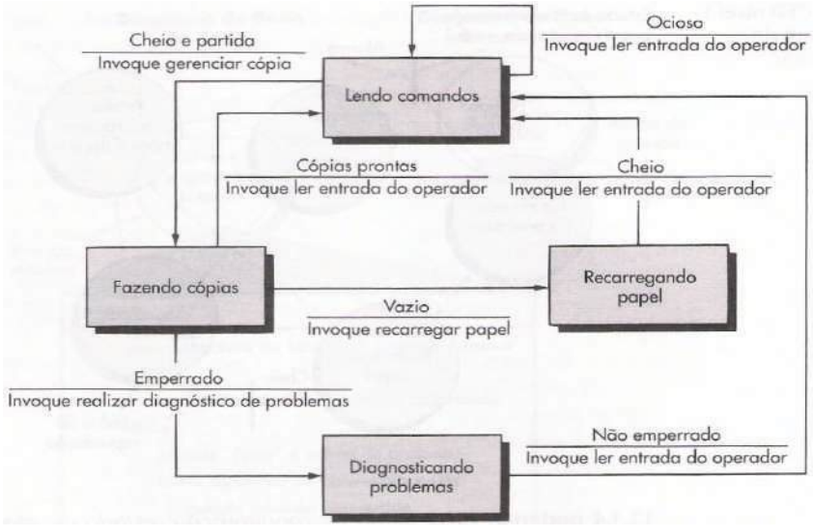


Figura 17 - DTE para o software de máquina fotocopiadora
Fonte: Pressman (2006)

A Figura 17 ilustra que os estados do sistema são representados por retângulos e as setas representam as transições entre estados e são duplamente rotuladas. O rótulo superior indica o evento que gera a transição e o rótulo inferior indica a ação que ocorre como conseqüência do evento. Assim, como exemplo, quando a bandeja está **cheia** e o botão **partida** é pressionado (gerou-se um evento), há a transição do estado lendo comandos para o estado fazendo cópias.

Para saber mais sobre a Análise Estruturada consulte DeMarco (1989) e Yourdon (1990); e sobre a Análise Essencial Pompilho (1995) e McMenamim & Palmer (1991).

Resumo

Nessa nossa terceira aula, compreendemos a função da qualidade, as notações de modelagem e entendemos o que é e como é uma análise estruturada.

Encerramos, aqui, a nossa terceira aula. Na próxima aula vamos trabalhar a análise orientada a objetos. Até lá!

Aula 4. Análise orientada a objetos

Objetivos:

- conhecer as classes de objetos e seus atributos; e
- compreender o que é uma análise orientada a objetos.

Chegamos a nossa penúltima aula desta disciplina. Vamos conhecer a análise orientada a objetos?

Pois bem, outra metodologia amplamente utilizada é a Análise Orientada a Objetos. De acordo com Pressman (2006), esta metodologia aborda o domínio do problema como um conjunto de objetos que têm atributos e comportamentos específicos. Os objetos são manipulados com uma coleção de funções, denominadas métodos, operações ou serviços e comunicam-se uns com os outros através de mensagens. Assim, quando um objeto é definido, seus atributos, operações e mensagens são levadas em consideração.

4.1 Abordagem

A importância desta abordagem está no fato de que os objetos encapsulam (guardam em si) não só os dados, mas o processamento aplicado a eles. Com isto, bibliotecas de classes, contendo objetos reusáveis, são desenvolvidas e, conforme já mencionado, reuso é um conceito de extrema importância, principalmente, em orientação a objetos, pois leva à criação mais rápida de software e a programas com maior qualidade. Além disto, o software orientado a objetos possui estrutura desacoplada, evitando, assim, efeitos colaterais quando uma parte do software é alterada e essas alterações não afetarão outras partes, permitindo, com isso, uma maior adaptabilidade e ampliação facilitada.

Sistemas orientados a objetos evoluem com o tempo. Desta forma, a engenharia de software orientada a objetos é mais bem descrita com uma abordagem que incentiva a criação de componentes (reuso) vinculada ao modelo

de processo evolutivo. Ao longo da espiral evolutiva, que começa com as entrevistas com os usuários, as classes básicas são levantadas.

Da mesma forma que na abordagem clássica não é possível o levantamento de todos os requisitos de software nas primeiras entrevistas, no modelo de análise e projeto orientado a objetos nem todas as classes necessárias são descobertas inicialmente, tornando-se necessárias classes adicionais ao longo do processo, demonstrando, assim, a natureza evolutiva desta abordagem.

Para que a abordagem orientada a objetos seja compreendida se faz necessário que alguns conceitos sejam assimilados.

4.2 Classes e objetos

Uma classe é um conceito da orientação a objetos que encapsula as abstrações de dados e procedimentais necessárias para descrever o conteúdo e o comportamento de alguma entidade do mundo real.

As abstrações de dados (atributos) que descrevem a classe são envolvidas por uma "parede" de abstrações procedimentais (operações ou métodos), que são capazes de manipular os dados de alguma forma. Esta característica leva à ocultação de informação e reduz possíveis efeitos que ocorreriam em caso de modificações. Os métodos, por manipular um conjunto pequeno de atributos, diz-se que eles são coesivos e, sendo o acesso aos atributos feito somente através dos métodos, a classe tende a ser desacoplada de outros elementos do sistema.

Em alguns casos, classes mais genéricas são construídas e, partir destas, classes mais especializadas são derivadas ou herdadas. Isto leva a uma hierarquia de classes, na qual classes superiores são denominadas de superclasses e classes derivadas subclasses. As subclasses herdam todos os atributos e métodos das superclasses e declaram seus próprios atributos e métodos. A Figura 18 apresenta uma hierarquia de classes para mobiliário.

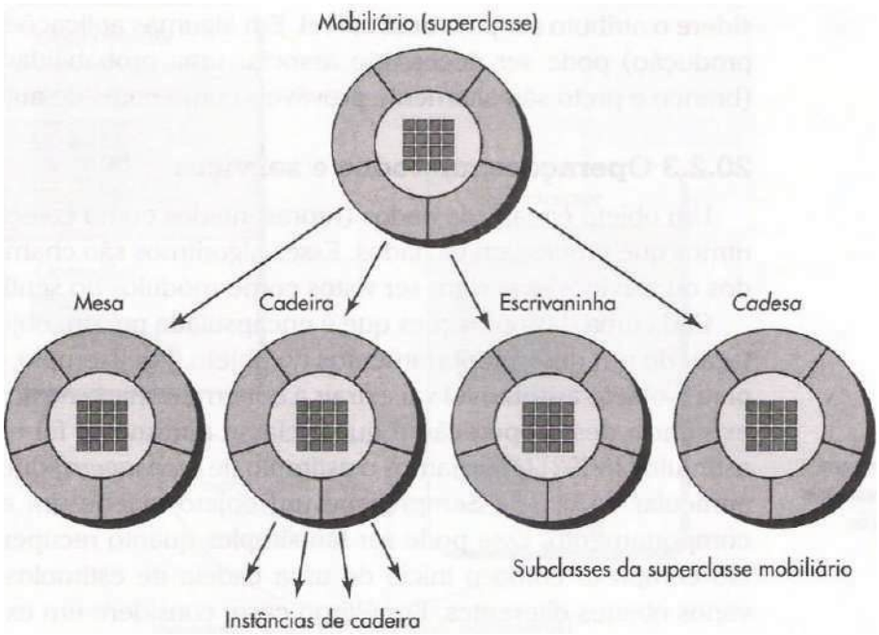


Figura 18 - Hierarquia de classes
Fonte: Pressman (2006)

Nesta hierarquia, as classes Mesa, cadeira, escrivaninha e cadeira herdam as características (atributos e métodos) da classe mobiliário.

4.3 Atributos

Atributos são descrições para classes de objetos. Uma pessoa, por exemplo, descrita pela classe "pessoa", tem características, tais como o nome, data de nascimento, altura, cor dos olhos e idade, entre outros. A classe "venda", que descreve uma nota fiscal de venda, por exemplo, é descrita por data da venda, vendedor, valor total da venda e um conjunto de itens que descreve os produtos vendidos.

4.4 Operações, métodos e mensagem

As operações (ou métodos) descrevem os algoritmos que processam os dados de um objeto. Cada operação descreve um comportamento encapsulado por um objeto. De acordo com os exemplos de classes e atributos citados anteriormente, um exemplo de método é obter total da venda, que buscará dentro do objeto o valor que representa o total da venda, possível a partir do atributo valor total da venda. Isso faz com que o objeto da classe "venda" receba um estímulo e este estímulo é denominado mensagem.

Uma mensagem é a forma pela qual os objetos interagem entre si. Em outras palavras, uma mensagem estimula que algum comportamento aconteça no objeto que a recebe.

4.5 Encapsulamento, herança e polimorfismo

Sistemas orientados a objetos se beneficiam de características importantes por encapsularem dados e operações em uma única entidade (classe):

- A ocultação de informação permite que detalhes dos dados e da implementação das operações permaneçam escondidas do mundo externo;
- Tanto dados quanto operações que os processam são encerrados (declarados) dentro de uma entidade;
- As mensagens enviadas entre objetos não precisam conhecer detalhes das estruturas interna dos dados.

A herança, conforme já visto, possibilita que novas classes possam fazer uso de classes já existentes, por meio do conceito de herança. Toda classe que herda de uma classe pré-existente possui, além de seus atributos e métodos, os atributos e métodos da classe a partir da qual se herda.

A palavra polimorfismo tem suas origens no Grego e significa muitas formas (poli = muitas, morphos = formas). De forma simplista, polimorfismo é a habilidade que um tipo A pode se comportar como um tipo B. Como exemplo genérico tem-se uma classe denominada Animal, que contém o método falar. Este método apresentará um texto que representa o som que o animal faz. Porém, em Animal não é possível implementar tal método, visto que um animal por si só não emite sons. Entretanto, a classe Cachorro e Gato herdam da classe Animal, conseqüentemente, implementarão (sobrescreverão) o método falar, pois estes animais sabem como "falar". Assim, quando uma mensagem for enviada a um objeto cachorro ordenando que ele "fale", o método falar apresentará o texto "au-au" e quando a mesma mensagem for enviada a um objeto gato ordenando que ele "fale", o método falar apresentará o texto "miau".

Outro exemplo a considerar é o de uma aplicação que desenha elementos gráficos (círculos, quadrados e triângulos). Em uma aplicação convencional

cada elemento gráfico necessitaria de um módulo de desenho, que teria uma lógica para saber qual o elemento a ser desenhado. O problema dessa abordagem reside no fato de que o acréscimo de novos elementos implicaria em novos módulos de desenho. Em orientação a objetos os elementos gráficos são descritos como classes que herdam, por exemplo, de uma superclasse "Gráfico", contendo o método desenhar. Empregando o conceito de sobrecarga, cada subclasse define seu próprio algoritmo no método desenhar que herdou da classe "Gráfico". Assim, o método desenhar pode ser invocado em qualquer um dos objetos que herde de "Gráfico". O objeto que receber a mensagem invocará seu próprio método desenhar fazendo com que o elemento gráfico representado pela classe do objeto seja desenhado.

4.6 Análise orientada a objetos

A análise orientada a objetos tem a finalidade de definir as classes que são importantes ao problema a ser resolvido, descobrindo seus atributos e operações, as relações existentes entre elas e o comportamento exibido por elas. As tarefas necessárias para alcançar tais objetivos são:

1. levantar os requisitos básicos junto aos usuários;

2. identificar as classes (atributos e operações);

3. especificar uma hierarquia de classes;

4. representar as relações entre objetos;

5. modelar o comportamento dos objetos;

6. aplicar as tarefas anteriores repetidamente (interativamente) até que o modelo seja obtido.

No final dos anos 80 e durante os anos 90 vários métodos de análise orientada a objetos foram sugeridos. Estes métodos eram compostos por um processo de análise, um conjunto de diagramas e uma notação utilizada para criar o modelo de análise. Entre todos, o que mais se destacou, na verdade, foi uma combinação de 3 (três), que propunha a combinação dos métodos idealizados por Grady Booch (Método Booch), James Rumbaugh (Técnica de Modelagem de Objetos (OMT – *Object Modeling Technique*) e Ivar Jacobson (Engenharia de Software Orientada a Objetos (OOSE – *Object-Oriented*

Software Engineering). Assim, surgiu a UML (*Unified Modeling Language* – Linguagem de Modelagem Unificada), cujo propósito foi o de agrupar os pontos fortes dos 3 (três) métodos.

Na UML a modelagem do software é expressa mediante uma notação de modelagem, que é regida pelas seguintes regras:

- **Sintáticas: define quais símbolos devem existir e como são combinados para formas sentenças;**
- **Semânticas: diz respeito ao significado de cada símbolo e como ele é interpretado por si só e no contexto de outros símbolos;**
- **Pragmáticas: corresponde às regras para a criação de sentenças através da definição das intenções dos símbolos.**

Um sistema é representado em UML por 5 (cinco) diferentes visões, cada uma definida por um conjunto de diagramas. A seguir serão apresentadas 2 (duas) das 5 (cinco) visões com as quais a análise se preocupa. As visões restantes são descritas na Unidade III – Projeto.

Na **visão do modelo do usuário** o sistema é representado sob o ponto de vista do usuário, denominado ator, e utiliza a abordagem de casos de uso, que descrevem os cenários de uso do sistema a partir da perspectiva do usuário. Em outras palavras, os casos de uso descrevem a interação dos atores com o sistema e como este é usado.

A **visão do modelo estrutural** se preocupa com a modelagem da estrutura estática do sistema (classes, objetos e relacionamentos) visando os dados e funcionalidades do sistema.

Como dito, o processo de análise orientada a objetos se preocupa em compreender como o sistema será usado. Uma vez que o cenário de uso tenha sido definido, inicia-se a modelagem do software.

Os casos de uso modelam o sistema do ponto de vista do usuário e servem como base para o elemento inicial do modelo de análise. Com o auxílio da UML uma representação gráfica do caso de uso é criada. Conhecida como Diagrama de Caso de Uso, ele apresenta atores e casos de uso.

Como exemplo, tem-se um sistema hipotético de segurança domiciliar. Entre outros, identificou-se o ator proprietário e 2 (dois) casos de uso, representados pela figura oval, apresentados na Figura 19.A. Cada caso de uso pode ser detalhado a níveis mais baixos, semelhante ao DFD, visto anteriormente. A Figura 19.B mostra o caso de uso para a função interage, apresentada na Figura 19.A.

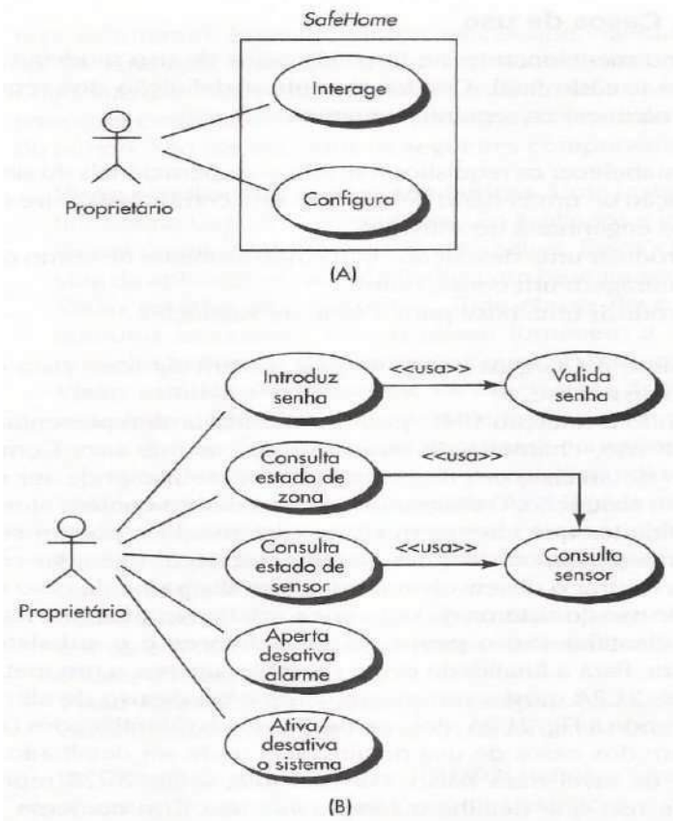


Figura 19 - Diagrama de caso de uso de alto nível (A) e detalhado (B)
Fonte: Pressman (2006)

Uma vez levantados os cenários de uso, passa-se a identificar as possíveis classes e quais são suas responsabilidade e colaborações. Este processo recebe o nome de modelagem classe-responsabilidade-colaboração (*class-responsbility-collaborator* – CRC) e pode ser realizado com fichas de indexação reais, com exemplo apresentado na Figura 20, ou virtuais. O objetivo da modelagem CRC é criar uma representação organizada de classes, que são levantadas a partir de uma análise gramatical na descrição do sistema. Todos os substantivos são classes em potencial. Os atributos e operações identificam as responsabilidades, que é qualquer coisa que a classe sabe ou faz. A classe se vale das colaborações com outras classes, que fornecem a ela informações necessárias para finalizar uma responsabilidade.

Nome da classe:	
Tipo da classe: (p. ex., dispositivo, propriedade, papel, evento)	
Característica da classe: (p. ex., tangível, atômica, concorrente)	
responsabilidades:	colaborações:

Figura 20 - Ficha de indexação do modelo CRC
Fonte: Pressman (2006)

Tendo sido estabelecidas as responsabilidades, a tarefa de modelagem objeto-relacionamento define as classes que colaboram entre si para satisfazer cada responsabilidade. Isto estabelece a associação (ou conexão) entre classes. Pressman (2006) sugere examinar os verbos na declaração dos casos de uso do sistema para que se possam encontrar os possíveis relacionamentos. A UML vale-se da notação para modelagem objeto-relacionamento a partir de uma simbologia adaptada da modelagem entidade-relacionamento. As associações entre objetos são realizada usando-se relacionamentos rotulados juntamente com a cardinalidade da associação. A Figura 21 apresenta um exemplo de relacionamentos entre objetos.

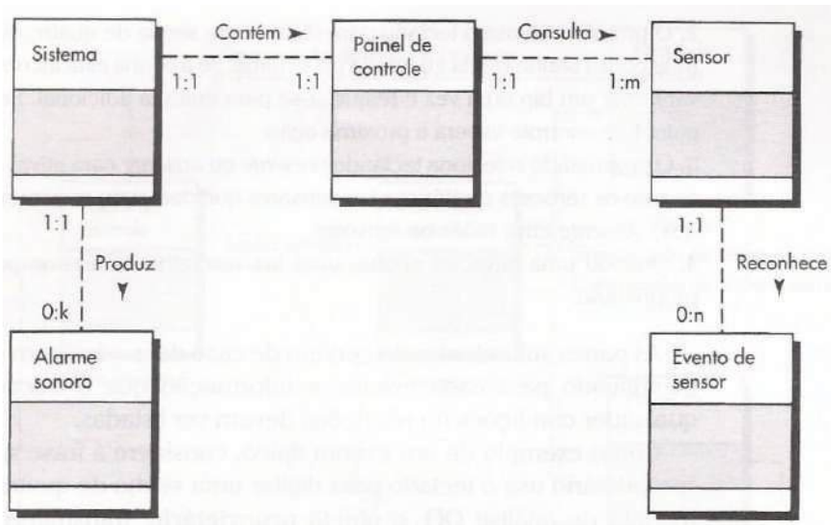


Figura 21 - Exemplo de relacionamento entre objetos
Fonte: Pressman (2006)

Tanto a modelagem CRC quanto a modelagem objeto-relacionamento correspondem a elementos estáticos no modelo de análise orientada a objetos. Para demonstrar o comportamento dinâmico do sistema é necessário representar o comportamento do sistema como uma função de eventos e tempo, ambos específicos. Neste sentido, a modelagem objeto-comportamento descreve como o sistema orientado a objetos vai responder a eventos e estímulos externos e é criado de acordo com os seguintes passos:

1. A interação interna do sistema é avaliada utilizando-se todos os casos de uso;

2. Identificar e entender os eventos, como eles direcionam as interações e como se relacionam a objetos específicos;

3. Para cada caso de uso deve-se criar uma marcação de eventos;

4. Construir um Diagrama de Transição de Estados para todos os casos de uso;

5. Verificar a precisão e consistência do modelo objeto-comportamento.

Um exemplo de transições de estados é ilustrado na Figura 22.

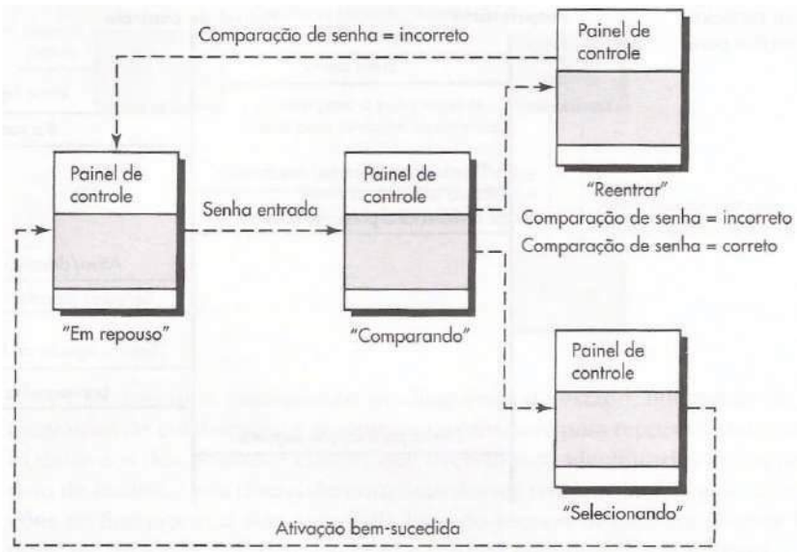


Figura 22 - Representação de transição de estados ativos
Fonte: Pressman (2006)

Marin (1994) e Martin, Odell (1995) detalham os princípios e as característi-

cas da análise orientada a objetos.

Resumo

Nesta aula estudamos as classes, objetos e seus atributos. Vimos as operações, métodos e mensagem. Trabalhamos, também, o encapsulamento, a herança e a poliformia. Por fim, estudamos a análise orientada a objeto.

E, assim, terminamos o nosso penúltimo encontro. Na próxima aula, vamos trabalhar a questão do projeto de software. Vamos para a nossa última aula?

Aula 5. Projeto

Objetivos:

- conhecer o modelo de análise orientada a objeto;
- conceituar pirâmide de projeto para software; e
- compreender os padrões de projeto.

Chegamos à nossa última aula. Vamos aprender sobre projetos de softwares?

Pressman (2006) citando Gamma e seus colegas (1995) resume o projeto orientado a objetos como sendo uma etapa difícil, senão impossível, de se obter de forma correta na primeira vez, pois um software específico que atenda os requisitos do usuário e, ao mesmo tempo, genérico o suficiente de tal forma que possa resolver problemas futuros, precisa ter os objetos necessários, fatorados e refinados, com interfaces adequadas e hierarquia de classes com relações-chave estabelecidas entre elas.

Conforme Sommerville (2003), um projeto orientado a objetos relaciona objetos com a solução que está sendo desenvolvida. Esta relação pode ser entre objetos do problema e objetos da solução, mas que, inevitavelmente, o projetista de software necessitará adicionar novos objetos e transformar objetos do problema tal que uma solução seja implementada.

5.1 Modelo de análise orientada a objetos

O modelo de análise orientada a objetos seleciona um conjunto de classes que serão utilizadas no desenvolvimento de um sistema orientado a objetos. Algumas dessas classes são reconhecidas logo de início. Entretanto, outras precisam ser levantadas a partir do zero. Por outro lado, se os padrões de projeto adequados forem seguidos classes poderão ser reusadas.

O projeto orientado a objetos visa a criação de uma documentação de proje-

to cujo objetivo é permitir ao engenheiro de software aplicar o reuso, conseguindo, com isto, agilidade no desenvolvimento e uma maior qualidade do produto final. As duas principais atividades deste processo são:

1. Projeto do sistema: esta atividade se preocupa em criar uma arquitetura em camadas que realizam funções específicas do sistema, bem como identificar as classes que residem em cada camada e que são pertinentes a cada subsistema;

2. Projeto de objeto: preocupa-se com os detalhes internos das classes, definindo atributos, operações e mensagens.

Para que um projeto orientado a objetos seja elaborado corretamente é preciso que sejam feitas revisões quanto à clareza, correção, completeza e consistência entre o produto final e os requisitos do usuário. Os principais componentes do sistema devem ser organizados em subsistemas (módulos). Os dados e operações que manipulam os dados são encerrados (encapsulados) em objetos, que é considerado uma forma modular.

O projeto orientado a objetos é construído sobre 4 (quatro) importantes conceitos: abstração, que é a capacidade de extrair do mundo real os requisitos essenciais para o sistema; ocultamento de informação, que impede que o mundo fora da classe tenha acesso aos dados da mesma; independência funcional, cada objeto executa as tarefas (os algoritmos) pelas quais ele é responsável; e modularidade, a organização dos objetos dá-se de forma modular, caracterizando o bloco de construção de uma sistema orientado a objetos.

5.2 Conceito de pirâmide de projeto para software

Um projeto orientado a objetos pode ser apresentado mediante o conceito de pirâmide de projeto para software. Esta pirâmide é dividida em 4 (quatro) camadas:

1. Camada de subsistema: apresenta cada um dos subsistemas que satisfazem as necessidades dos usuários;

2. Camada de classes e objetos: contém hierarquias de classes para a criação do sistema usando generalizações e especializações, além das repre-

sentações dos objetos;

3. Camada de mensagens: estabelece a interface externa e interna do sistema, detalhando como cada objeto se comunica com seus colaboradores;

4. Camada de responsabilidades: descreve as estruturas de dados de todos os atributos e os algoritmos de todas as operações de cada objeto.

Além destas camadas, existe outra camada com papel fundamental na construção do sistema orientado a objetos e que é responsável pela criação dos objetos do domínio, também conhecidos como padrões de projeto. Estes objetos fornecem suporte a atividades de interface homem/máquina, gestão de tarefas e gestão de dados.

Da mesma forma que o projeto convencional de software, o projeto orientado a objetos também aplica o projeto de dados, quando os atributos são representados; projeto de interface, para o desenvolvimento de um modelo de mensagens e projeto em nível de componente (procedimental) para o projeto de operações. Apesar da semelhança, a arquitetura de um projeto orientado a objetos baseia-se nas colaborações entre objetos mais do que no fluxo de controle entre os componentes do sistema.

Conforme visto nas aulas anteriores, vários métodos foram propostos para a Análise Orientada a Objetos e o que mais foi bem sucedido, na verdade, foi uma combinação de 3 (três) métodos e que ficou conhecida como UML. A proposta da UML é a representação do sistema através de visões. A **visão do modelo do usuário** e a **visão do modelo estrutural** são representadas no modelo de análise e fornecem o entendimento dos cenários de uso do sistema (propondo diretrizes para a modelagem comportamental) e estabelecendo fundamentação para as visões do modelo de implementação e do ambiente, através da identificação e descrição dos elementos estruturais estáticos do sistema e foram abordadas na aula 2 Análise.

No projeto orientado a objetos, 3 (três) das 5 (cinco) visões são tratadas na modelagem do projeto UML. A **visão do modelo comportamental** representa os aspectos dinâmicos ou comportamentais do sistema, bem como mostra as interações ou colaborações entre os vários elementos estruturais apresentados nas visões do modelo do usuário e do modelo estrutural. A **visão do modelo de implementação** representa como os aspectos estruturais e comportamentais devem ser construídos. A **visão do modelo do**

ambiente representa os aspectos estruturais e comportamentais do ambiente no qual o sistema será implementado.

As duas principais atividades de projeto da UML são: projeto de sistema e projeto de objetos. A arquitetura do software é representada no projeto de sistema. O projeto de objetos tem foco na descrição dos objetos e as interações entre si. No decorrer do projeto de objetos é criada uma especificação detalhada das estruturas de dados dos atributos e um projeto procedimental de todas as operações.

Ambos os projetos são estendidos para considerar o projeto das interfaces com o usuário, gestão de dados e gestão de tarefas nos subsistemas.

O projeto da interface com o usuário cria um meio efetivo de comunicação entre o ser humano e o computador através de princípios de projeto de interface, fornecendo um cenário que é detalhado interativamente até tornar-se um conjunto de classes de interface. Atualmente, este conjunto de classes compõe uma biblioteca de componentes reusáveis de software, tornando mais rápido o projeto e a criação da interface gráfica com o usuário (GUI – *Graphical User Interface*).

O projeto de gestão de dados estabelece um conjunto de classes direcionadas à persistência dos dados, normalmente, realizada mediante o uso de um SGBD (Sistema Gerenciador de Bancos de Dados).

Finalmente, a organização dos subsistemas em tarefas e o gerenciamento de suas ocorrências ficam a cargo do projeto de gestão de tarefas.

A Figura 23 ilustra o fluxo do processo de projeto orientado a objetos.

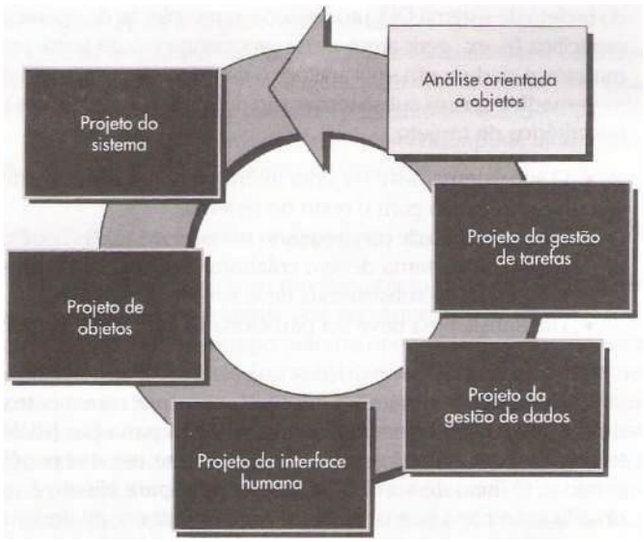


Figura 23 - Fluxo do processo de projeto orientado a objetos
Fonte: Pressman (2006)

Conforme mencionado, as duas principais atividades da UML no processo de projeto orientado a objetos é a atividade do processo de projeto do sistema e do projeto de objetos.

No processo de projeto do sistema fornece detalhes para a construção da arquitetura de um sistema e é dividido nas seguintes atividades:

- Particionamento do modelo de análise em subsistemas;
- Identificação da concorrência que é ditada pelo problema;
- Alocação de subsistemas a processadores e tarefas;
- Desenvolvimento do projeto para a interface com o usuário;
- Escolha da estratégia para a gestão de dados;
- Identificação dos recursos globais e os mecanismos de controle necessários para acesso a estes recursos;
- Projeto de um mecanismo de controle para o sistema, incluindo gestão de tarefas;
- Consideração para a manipulação das condições-limite;

- Revisão e consideração sobre concessões.

O processo de projeto de objetos foca nos detalhes dos objetos e de suas interações, preocupando-se, em particular, com a especificação dos tipos de atributo e o funcionamento das operações.

5.3 Padrões de projeto

Um padrão de projetos descreve uma solução para um problema recorrente (que acontece com frequência) no desenvolvimento de software orientado a objetos. Os padrões de projeto têm como objeto facilitar a reutilização de soluções de projeto, isto é, soluções levantadas na fase do projeto orientado a objetos, sem se preocupar com a reutilização de código.

No decorrer do projeto orientado a objetos o projetista de software deve procurar reusar padrões de projeto existentes ao invés de criar novos.

O texto mais conhecido sobre padrões de projeto é o livro **Design Patterns: elements of reusable object-oriented software** (Padrões de Projeto: elementos de software orientado a objetos reusável, em tradução livre), publicado por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides em 1995, no qual os autores descrevem 23 (vinte e três) padrões de projeto para problemas freqüentes em projetos de software orientados a objetos. Os autores ficaram conhecidos como o "Grupo dos Quatro" (*Gang of Four*) ou mais especificamente pelo acrônimo GoF.

Resumo

Nesta aula, vimos a função do modelo de análise orientada a objetos, o conceito de pirâmide de projeto para software e suas camadas e os padrões de projetos.

E, assim, terminamos a nossa quinta e última aula da disciplina de Análise e Projeto de Software. Siga em frente que ainda tenho uma mensagem para você!

Uma boa fonte de pesquisa na Internet a respeito de padrões de projeto é a Wikipédia: A Enciclopédia livre, tanto a versão em português http://pt.wikipedia.org/wiki/Padr%C3%B5es_de_projeto, quanto a versão em inglês http://en.wikipedia.org/wiki/Design_pattern_(computer_science), apresentam boas informações a respeito dos padrões de projeto de software, além de ambas possuírem referências (*links*) para outros documentos eletrônicos sobre o assunto.

Palavras Finais

Espero que tenha tirado o máximo proveito da nossa disciplina. Como disse no início dos nossos encontros, o mercado carece de profissionais competentes. Assim, espero que tenha se dedicado e que, com nossa disciplina, tenhamos contribuído para a sua formação. É importante lembrar que o bom desempenho é característica de quem quer vencer.

Desejo que continue estudando e se dedicando para que consiga realizar os seus sonhos.

Um grande abraço e sucesso!

Prof. Romualdo Rubens de Freitas

Referências

DEMARCO, Tom. **Análise Estruturada e Especificada de Sistemas**. Rio de Janeiro: Campus, 1989.

FURLAN, José Davi. **Modelagem de Objetos através da UML** – the Unified Modeling Language. São Paulo: Makron Books, 1998.

Engenharia de software. Disponível em: <http://pt.wikipedia.org/wiki/Engenharia_de_software> Acesso em: 7 out. 2013.

LARMAN, Craig. **Utilizando UML e Padrões** – Uma Introdução a Análise e ao Projeto Orientado a Objeto. 3 ed. Porto Alegre: Bookman, 2007.

MARTIN, James. **Princípios de Análise e Projeto baseados em Objetos**. Rio de Janeiro: Campus, 1994.

MARTIN, James, ODELL, James J. **Análise e Projeto Orientados a Objeto.** São Paulo: Makron Books, 1995.

MCMENAMIM, Stephen M., PALMER, John F. **Análise Essencial de Sistemas**. São Paulo: McGraw-Hill, 1991.

MEDEIROS, Ernani. **Desenvolvendo Software com UML 2.0** – Definitivo. São Paulo: Makron Books, 2004.

POMPILHO, S. **Análise Essencial**. Rio de Janeiro: Infobook, 1995.

PRESSMAN, Roger S. **Engenharia de Software**. 6 ed. Rio de Janeiro: McGraw-Hill, 2006.

SILVA, Nelson Peres da. **Análise e Estruturas de Sistemas de Informação**. São Paulo: Érica, 2007.

SOMMERVILLE, Ian. **Engenharia de Software**. 6 ed. São Paulo: Addison Wesley, 2003.

WAZLAWICK, Raul Sidnei. **Análise e Projeto de Sistemas Orientados a Objetos**. Rio de Janeiro: Campus, 2004.

YOURDON, Edward. **Análise Estruturada Moderna**. Rio de Janeiro: Campus, 1990.