UNIP
UNIVERSIDADE PAULISTA
Interativa

Projeto de Sistemas Orientado a Objetos

Autor: Prof. Fábio Versolatto
Colaboradores: Prof. Luciano Soares de Souza
Prof. Eduardo de Lima Brito

Professor conteudista: Fábio Versolatto

Mestre em Engenharia de *Software* pelo Instituto de Pesquisas Tecnológicas (IPT) da USP, especializado em Arquitetura, Componentização e SOA pela Universidade Estadual de Campinas (Unicamp), pós-graduado em Tecnologia de Construção de *Software* Orientado a Objetos pelo Centro Universitário Senac e bacharel em Ciência da Computação pelo Centro Universitário da FEI.

Professor do curso de Análise e Desenvolvimento de Sistemas da Universidade Paulista (UNIP), no qual leciona e orienta os alunos no programa de *Projeto Integrado Multidisciplinar*.

Atua na área de pesquisa e possui publicações e participações em eventos na área de Engenharia de *Software* no Brasil e no exterior, além de projetos de pesquisa aplicados à área social.

Atua em desenvolvimento de sistemas de *software* e possui larga experiência em sistemas estratégicos para empresas de grande porte, com ênfase em análise de requisitos, projeto, arquitetura e desenvolvimento de soluções de *software*.

Dados Internacionais de Catalogação na Publicação (CIP)

V564p Versolatto, Fábio Rossi.

Projeto de Sistemas Orientado a Objetos. / Fábio Rossi Versolatto. – São Paulo: Editora Sol, 2015.

152 p., il.

Nota: este volume está publicado nos Cadernos de Estudos e Pesquisas da UNIP, Série Didática, ano XXI, n. 2-151/15, ISSN 1517-9230.

1. Engenharia de software. 2. Sistemas. 3. Tecnologia. I. Título.

CDU 681.3.02

Sumário

Projeto de Sistemas Orientado a Objetos

APRESENTAÇÃO 7
INTRODUÇÃO 7

Unidade I

1 INTRODUÇÃO A PROJETO DE SISTEMAS 9
1.1 Por que "projetar"? 9
2 O PROJETO NO CICLO DE VIDA DA ENGENHARIA DE *SOFTWARE* 11
2.1 A fase de projetos 11
2.2 Por que modelar? 14
2.3 Conceitos do projeto 15
2.3.1 Abstração 15
2.3.2 Modularidade 16
2.4 Fases de projeto 18
2.5 Aspectos humanos da fase de projetos 20
2.6 O que buscamos atingir no projeto? 22
2.7 Introdução ao projeto orientado a objetos 26

Unidade II

3 TECNOLOGIA DE APOIO AO PROJETO ORIENTADO A OBJETOS 34
3.1 A UML 34
3.2 Ferramentas de modelagem UML 39
3.3 As ferramentas CASE 40
3.4 Tecnologia *back-end* 43
3.5 Tecnologia *front-end* 47
3.5.1 Linguagens OO, um breve comparativo 49
4 PASSANDO DA ANÁLISE AO PROJETO 50
4.1 Desenvolver o modelo de classes de projeto refinando o modelo conceitual 51
4.1.1 Modelo conceitual 51
4.1.2 Modelo de projeto 52
4.1.3 Modelo de implementação 54
4.2 Atividades clássicas para passagem da análise para o projeto 55
4.2.1 Detalhamento dos aspectos dinâmicos do sistema 55
4.2.2 Refinamento dos aspectos estáticos e estruturais do sistema 61
4.2.3 Definição de outros aspectos da solução 63

Unidade III

5 PROJETOS DE DADOS E CLASSES E PROJETO ARQUITETURAL 70
5.1 Projeto de dados e classes 70
5.1.1 Introdução ao projeto de dados 70
5.1.2 Introdução a banco de dados relacionais 72
5.1.3 Bancos de dados relacionais *versus* orientação a objetos 77
5.2 Projeto arquitetural 81
5.2.1 Qual o papel da arquitetura? 81
5.2.2 O que é arquitetura de *software*? 82
5.2.3 A importância da arquitetura de *software* 83
5.2.4 Processo de arquitetura de *software* 84
5.2.5 Processo de arquitetura de *software* em aspectos humanos 87
6 VISÕES DA ARQUITETURA DE *SOFTWARE* 88
6.1 Visão estática 88
6.1.1 Utilização de padrões na arquitetura de *software* 89
6.1.2 Estilo arquitetural 90
6.1.3 Estruturação de sistemas em subsistemas e camadas 92
6.1.4 Introdução a padrões de projeto (*design patterns*) 96
6.2 Visão dinâmica 101
6.2.1 Definindo responsabilidades 102
6.2.2 Refinando o diagrama de sequência 103
6.3 Documentação de arquitetura 104

Unidade IV

7 REFINANDO A MODELAGEM DE ASPECTOS DINÂMICOS DO *SOFTWARE* 110
7.1 Diagrama de comunicação 110
7.2 Diagrama de máquina de estado 114
8 PROJETO DE INTERFACES E PROJETO DE COMPONENTES 121
8.1 Projeto de interfaces 121
8.1.1 Especificando as interfaces dos objetos 122
8.1.2 Diagrama de pacotes 123
8.1.3 Introdução ao projeto de interfaces com o usuário 129
8.2 Projeto de componentes 130
8.2.1 Introdução à componentização e ao reúso de *software* 130
8.2.2 Definindo as interfaces externas 132
8.2.3 Diagrama de componentes 133
8.2.4 Diagrama de distribuição 135

APRESENTAÇÃO

A disciplina *Projeto de Sistemas Orientado a Objetos* tem como objetivo apresentar ao aluno as boas práticas para transformar as necessidades das organizações, inseridas em um mercado cada vez mais competitivo, em um projeto de *software* eficiente, com qualidade de desenvolvimento e, principalmente, que possibilite vantagem competitiva à organização, utilizando para isso o paradigma da orientação a objetos.

Para atingir este objetivo a disciplina começa contextualizando para o aluno os problemas que o projeto *software* enquanto atividade do ciclo de vida do *software* se propõe a resolver.

Com o aluno integrado na temática, a disciplina apresenta e discute a importância de as atividades de projeto serem executadas de forma organizada dentro de um projeto de *software*, apresentando as principais metodologias e técnicas empregadas no processo de transformação das necessidades da organização em sistema de *software*.

Conceitos básicos e motivações postos, debate-se a arquitetura de *software* como fase fundamental dentro de um projeto de desenvolvimento, explanando a importância dos modelos e mostrando as principais técnicas para modelagem, documentação e padrões utilizados para definição da arquitetura de um sistema de *software*.

Dentro de todo esse contexto, a disciplina apresenta as tecnologias que dão suporte ao projeto orientado a objetos e os diagramas da UML (Unified Modeling Language) como uma ferramenta para representação e modelagem de um sistema de *software*, utilizando-os como ferramenta de apoio em todo o processo.

INTRODUÇÃO

Desde que o *software* se estabeleceu como uma ferramenta importante na estratégia competitiva das grandes empresas, a indústria de desenvolvimento vem passando por transformações para atender as necessidades cada vez maiores deste mercado.

O desafio lançado não está mais no simples fato de se desenvolver uma série de linhas de código que, agrupadas, compõem um sistema de *software*, mas sim em desenvolver este *software* como um produto, como uma ferramenta de estratégia competitiva que atenda a uma série de exigências e determinados padrões de qualidade.

O velho paradigma de que "*software* bom é aquele que funciona" começa a ser quebrado por esses padrões de qualidade. Atualmente, o mínimo que se espera de um *software* é que ele funcione; é o mínimo que se exige para que tanto a organização que utiliza o sistema quanto aquela que o desenvolveu se mantenham em um mercado cada vez mais competitivo.

Se o *software* deve ser enxergado como uma ferramenta de estratégia competitiva em uma grande corporação, parece-nos razoável que um projeto não deva tomar um tempo excessivo, afinal de contas, em um mercado cada vez mais competitivo, tem a vantagem aquele que larga na frente.

Seguindo a mesma linha de raciocínio, espera-se que um projeto não tenha custos exorbitantes, o que pode inclusive torná-lo inviável. Em casos de projetos malsucedidos, o custo de um processo de negócio realizado de forma não automatizada pode ser mais vantajoso que automatizá-lo. Logo, podemos notar que o *software* deve ter boa relação custo-benefício.

Idealmente, um projeto deve ser executado em um tempo razoável e com um custo viável. Como se isso não bastasse, um sistema de *software* competitivo deve atingir certo nível de qualidade.

Palavras como **manutenibilidade**, **desempenho**, **escalabilidade**, **disponibilidade**, dentre outras, permeiam, cada vez mais, o universo da engenharia de *software* e demonstram o nível de qualidade que se espera de um sistema de *software*.

O nível de qualidade de um sistema de *software* pode ser traduzido não só pelo nível de atingimento de automatização das necessidades de negócio, mapeados na engenharia de requisitos, mas também, pelo nível de eficiência no qual a solução de *software* atinge essas necessidades e pelo custo-benefício do processo de desenvolvimento e construção dessa solução.

Parece-nos razoável, em um mundo onde a tecnologia de *hardware* evoluiu e segue em constante evolução, que devamos nos preocupar em desenvolver sistemas de *software* que se adaptem às diversas tecnologias e plataformas de computadores pessoais, sistemas operacionais, dispositivos móveis, dentre outros. Tudo isso sem perder o foco em fornecer uma experiência rica ao usuário final, fazer que ele tenha segurança e conforto no uso, fazer que ele possa realizar suas atividades de maneira produtiva.

Se o *software* deve ser enxergado como uma ferramenta de estratégia competitiva em uma grande corporação e deve ser desenvolvido em tempo e custo de tal forma que o torne efetivamente competitivo, não é difícil enxergar a necessidade de pensarmos em um projeto que nos possibilite manutenção rápida e eficiente. Afinal de contas, se o mercado competitivo muda com muita rapidez, é possível imaginar que um sistema de *software* também deva se adaptar a essa realidade.

Posta essa reflexão inicial, podemos nos questionar: será que realmente estamos dando a devida importância ao processo de pensar a solução antes de simplesmente nos colocarmos em frente ao computador e sairmos digitando linhas de código a esmo? Por que características de "outras engenharias" como padronização, arquitetura, componentização, projeto e estabelecimento de um processo produtivo nos parecem tão distantes da engenharia de *software*?

O paradigma da orientação a objetos busca oferecer um norte para as melhores práticas de desenvolvimento de tal forma que possamos atingir, como produto final, o *software* como uma ferramenta de estratégia competitiva de qualidade. Dentro do ciclo de vida de um projeto de *software*, a atividade de projeto, ou *design*, busca organizar as soluções de projeto de tal forma que explore todo o potencial da orientação a objetos em prol de um sistema de *software* de qualidade.

Unidade I

1 INTRODUÇÃO A PROJETO DE SISTEMAS

Nesta unidade iremos abordar os aspectos introdutórios que motivam a fase de projetos no paradigma da orientação a objetos.

1.1 Por que "projetar"?

Antes de pensarmos em um projeto de *software*, vamos pensar em outro tipo de projeto tão comum no nosso dia a dia: o projeto de uma casa.

Suponhamos a seguinte situação: você possui um terreno e deseja construir neste terreno uma casa.

Em um primeiro momento você mapeia tudo o que deseja que essa casa tenha, como uma lista de desejos, ou seja, quantidade de quartos, de banheiros, área de lazer, enfim, tudo aquilo que lhe é necessário.

Fazendo um paralelo com a engenharia de *software*, é como se nesse momento você tivesse a sua lista de requisitos.

Com os seus requisitos listados você os valida, verificando se todas as suas necessidades estão listadas e detalhadas tal qual você deseja, se a quantidade de cômodos está adequada, se a metragem desejada para cada cômodo está detalhada e se a metragem da casa é compatível com a metragem do terreno.

Feito isso, basta comprar os materiais e começar a construir, certo? Todos sabem que não é bem assim.

Excluindo desse cenário hipotético todos os problemas legais, é até possível que você consiga uma casa, mas é mais provável que você enfrente muitos problemas, como os que seguem.

- A casa construída pode não atender as suas necessidades, por exemplo, a quantidade de quartos pode até estar adequada aos seus desejos, mas a metragem não.

- Pode ocorrer de a casa ficar com má qualidade: paredes tortas, telhado torto, janelas e portas fora das medidas-padrão.

- Podem ocorrer problemas na infraestrutura: por mau dimensionamento, as instalações eletro-hidráulicas podem ficar incompletas, não funcionar como deveriam, além de oferecer risco à estrutura da casa e a seus ocupantes.

- O dinheiro gasto no fim da obra pode superar o valor que você havia previsto inicialmente.

- O tempo de construção também pode superar o tempo previsto.

Esse é apenas um dos possíveis cenários não muito favoráveis que você poderá encontrar se não adotar uma metodologia, ou um paradigma, para a construção da casa.

Na engenharia civil o paradigma de projeto já é mais que difundido. No caso do seu projeto, após os seus requisitos, ou sua lista de desejos fechados, verificados e validados, a linha certa a ser adotada seria desenvolver o projeto da sua casa.

O projeto da sua casa muito provavelmente não seria desenvolvido por você, mas sim por um profissional ou uma equipe capaz de traduzir os seus desejos em algo mais concreto.

Por exemplo, a planta baixa da sua casa, a planta elétrica, a planta hidráulica, uma maqueta 3-D, enfim, tudo aquilo que possa representar os seus desejos de tal forma que eles possam ser convertidos efetivamente no produto final: a casa.

As plantas produzidas pela equipe de projeto serão utilizadas como guia para a equipe de construção, de tal forma que os riscos associados à má qualidade da casa sejam diminuídos, ou seja, começa a diminuir a probabilidade de problemas relacionados, por exemplo, à metragem dos cômodos, às paredes e a todos os demais que comentamos anteriormente.

Além de servirem como um guia para a equipe de construção, as plantas, ou melhor, chamemos a partir de agora de modelos, servem como importante instrumento de transição entre os seus requisitos e o que será construído.

Os modelos, no projeto da sua casa, são artefatos importantes para a comunicação entre os envolvidos no projeto, neste caso, a comunicação entre você e o projetista, e entre o projetista e a equipe de construção.

Cada modelo é desenvolvido dirigido ao público-alvo a quem se deseja comunicar e com um objetivo específico. Por exemplo, a planta, ou maquete da parte elétrica, possui algumas finalidades:

- Mostrar para você e para a equipe de construção os locais corretos onde haverá tomadas e pontos de luz.

- Fornecer subsídios para que a equipe especializada em eletricidade determine quais componentes serão utilizados, de tal forma que se possa mensurar a capacidade e a segurança do sistema.

Como você pode notar, a fase de projeto, bem como os modelos produzidos nela, são fatores fundamentais para o sucesso do projeto da sua casa.

A engenharia civil é, obviamente, muito mais antiga que a engenharia de *software*; se compararmos, a engenharia de *software* não chega perto do que a civil evoluiu em termos de profissionais, ferramentas e processos.

Façamos o seguinte exercício de analogia: em vez do projeto de construção de uma casa, imagine que seu objetivo agora seja construir um sistema de *software*.

Inicialmente você mapeia, valida e verifica os requisitos do usuário, como você já viu nas disciplinas anteriores.

Ainda no campo da suposição, você é um desenvolvedor, um programador habilidoso, pega os requisitos e parte para a codificação.

A prática e a experiência mostram que os resultados para o seu produto final, o *software*, não fogem muito dos resultados obtidos no projeto de construção da casa.

Um produto de má qualidade, que não atende às necessidades do usuário, que possuí problemas, que não se adapta à infraestrutura, ou que foi desenvolvido fora dos custos e do tempo previsto, é ainda uma realidade na indústria de *software*.

Em um mercado cada vez mais competitivo, problemas dessa natureza passam a ser intoleráveis. A competitividade do *software* começa a trazer à tona discussões sobre outros aspectos importantes do produto, como manutenibilidade e usabilidade.

Como construir o *software* de tal forma que atenda aos requisitos do usuário de forma eficaz e que, ainda assim, seja sustentável e competitivo? Uma das respostas pode estar no paradigma da projetização.

2 O PROJETO NO CICLO DE VIDA DA ENGENHARIA DE *SOFTWARE*

2.1 A fase de projetos

Como você deve lembrar, na engenharia de *software* temos alguns modelos de processo de desenvolvimento.

Observação

O processo de desenvolvimento de *software* resume-se a um conjunto de atividades executadas em uma determinada sequência. Esse conjunto de atividades também pode ser chamado de etapas da engenharia de *software* ou paradigmas da engenharia de *software* (PRESSMAN, 2006).

Muitos são os modelos de processos aplicados e debatidos atualmente. Para exemplificar, vamos trabalhar com um dos modelos mais tradicionais: o Modelo Cascata.

Também chamado de *waterfall* ou também citado na literatura como ciclo de vida clássico, o Modelo Cascata é composto de cinco atividades:

- Análise de requisitos.
- Projeto.
- Codificação.
- Testes.
- Implantação e manutenção.

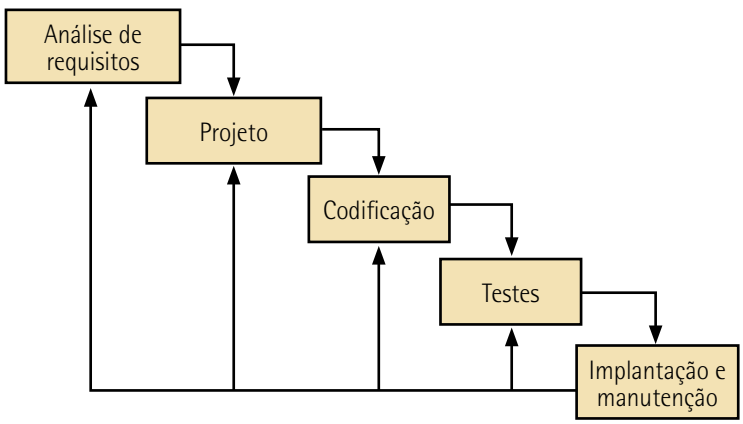

Figura 1 – Sequência de execução das atividades no Modelo Cascata

No Modelo Cascata, as atividades são executadas de forma sequencial. Assim, uma atividade não é iniciada até que sua predecessora seja completamente finalizada. Isso nos permite afirmar, por exemplo, que a construção não é iniciada até que seja finalizada a arquitetura do sistema de *software* na fase de projeto, que, por sua vez, não é iniciada até que todos os requisitos sejam levantados, documentados e aprovados pelo usuário.

A fase de projeto não se inicia até que todos os requisitos sejam elicitados, documentados e aprovados pelo usuário. É comum que requisitos novos surjam e sejam alterados no curso do projeto, pelas mais diversas razões, dentre elas a falta de habilidade do analista em extrair e captar as necessidades do processo de negócio. A mudança nos requisitos pode ocorrer durante a fase de projeto, codificação ou até mesmo na fase de testes, assim como problemas na arquitetura podem ser identificados na construção ou na implantação.

Saiba mais

Existem ainda outros modelos, como o Modelo Incremental, o Modelo Espiral e o Processo Unificado. Para aprofundar seus conhecimentos sobre o assunto, leia:

PRESSMAN, R. S. *Engenharia de software*. 6. ed. São Paulo: McGraw Hill, 2006.

SOMMERVILLE, I. *Engenharia de software*. São Paulo: Pearson, 2010.

Independentemente do modelo de ciclo de vida que adotarmos, peguemos como exemplo o Modelo Cascata. A fase de projetos encontra-se sempre na fronteira entre a análise de requisitos, o problema e a construção, ou implementação.

A fase de projetos sempre se inicia após a fase de requisitos, ou após uma primeira iteração dos requisitos, nos casos em que adotamos um modelo de ciclo de vida iterativo incremental ou qualquer variante dele.

Observação

A diferença fundamental entre os modelos clássico, cascata e iterativo: no modelo incremental não é necessário que todos os requisitos estejam mapeados para se iniciar o ciclo de desenvolvimento.

Logo, enquanto não tivermos os requisitos, ou parte deles, elicitados, verificados e validados, não se inicia a fase de projeto. Isso significa dizer que, uma vez iniciada a fase de projetos, os requisitos não poderão ser alterados?

A resposta é não, pois os requisitos sempre poderão ser alterados, inclusive, é comum que isso aconteça; no entanto, uma vez que haja necessidade de alteração dos requisitos após a fase de projetos ser iniciada, os impactos deverão ser mapeados e mensurados, e as devidas alterações no projeto deverão ser feitas.

Podemos dizer que a fase de projeto dentro do ciclo de vida da engenharia de *software* é a fronteira entre o domínio do problema e o domínio da solução. Como Pressman (2006, p. 206) cita, "o modelo de requisitos dá ênfase no **o que** precisa ser feito, enquanto o modelo de projeto indica o **como** será feito".

Observação

Na literatura a terminologia indicativa de projeto em inglês é *design*. Não se confunda com *project*.

2.2 Por que modelar?

Voltemos ao nosso exemplo do projeto de construção de uma casa. A planta baixa da casa é a representação de uma visão da casa, podemos considerar que ela é um modelo que representa a casa.

Este modelo pode ser utilizado tanto para você validar se todas as suas necessidades serão efetivamente construídas quanto para ser um guia para a equipe de construção.

Imaginemos que, em uma situação hipotética, você não tenha gostado de algo da planta; o projetista, por alguma razão, não conseguiu captar suas necessidades, ou até mesmo você mudou de ideia com relação a alguma necessidade.

Diante dessa situação, basta alterar a planta, ou o modelo, e validá-lo novamente. Os esforços seriam muito maiores se tivéssemos de alterar a construção, em vez do modelo.

Na fase de projeto, os modelos de projeto têm como objetivo representar as diversas visões da solução de um sistema de *software*.

Assim como a planta da casa, esses modelos têm como objetivo representar **como** os requisitos serão atendidos, antecipam possíveis alterações que possam ser feitas quando do momento da construção (codificação), antecipando, assim, alguns dos riscos do projeto.

Modelos são como guia para a construção e são insumos fundamentais para a validação da qualidade do *software*. Trataremos com mais detalhes a seguir.

Saiba mais

A discussão sobre o processo de qualidade é ampla. Leia:

COSTA, I. *et al. Qualidade em tecnologia da informação*. São Paulo: Atlas, 2013.

É na fase de projeto, também, que incorporamos elementos de tecnologia para a solução, como no exemplo da casa, quando falamos sobre os componentes de uma planta elétrica.

Nesse caso, modelos são úteis para que possamos mensurar a infraestrutura necessária e efetuar provas de conceito de possíveis soluções que possamos adotar, sempre, antes do momento da construção.

Observação

O objetivo central deste livro-texto é discutir a fase de projetos de um sistema de *software* que, por sua vez, tem ênfase em como serão resolvidos os problemas e as necessidades do negócio que são levantadas na fase de análise. É importante ter em mente que, em qualquer modelo de ciclo de vida, antes da fase de projeto, temos a fase de análise, que, dentre outras atividades, compreende a engenharia e análise dos requisitos e dos problemas a serrem resolvidos pelo sistema de *software*.

2.3 Conceitos do projeto

Em seu livro, Pressman (2006, p. 212), enumera uma série de conceitos básicos de projeto que "fornecem a organização necessária para estruturá-lo e fazer com que funcione corretamente".

Observação

Roger Pressman e Iam Sommerville são considerados dois dos maiores expoentes da Engenharia de *Software*.

Abstração e modularidade são dois dos principais conceitos apresentados por Pressman (2006).

2.3.1 Abstração

O projeto deve, por finalidade, possuir vários níveis de abstração. Nos níveis mais altos de abstração do *software* nos aproximamos do nível de análise, enquanto nos níveis mais baixos nos aproximamos da solução técnica do *software*.

Observação

O conceito de abstração está ligado a nossa capacidade, enquanto analistas, de resolver problemas, de selecionar determinados aspectos do problema e de isolar o que é importante do que não é, para um determinado propósito. Abstração é dar ênfase àquilo que é essencial.

No começo do projeto é importante darmos ênfase a soluções macro, e à medida que o projeto for avançando, vamos descendo ao nosso nível de solução, ou de abstração.

2.3.2 Modularidade

O *software* deve ser dividido em componentes, ou módulos, que trabalham em conjunto para desempenhar uma determinada atividade e atingir um determinado objetivo.

O grande desafio da modularidade está em equilibrar as responsabilidades de cada componente, de tal forma que não haja uma superdependência ou a sobrecarga de um único componente.

O que se deseja atingir com a modularidade é: baixo acoplamento e alta coesão. Para entender a questão, veja a figura a seguir.

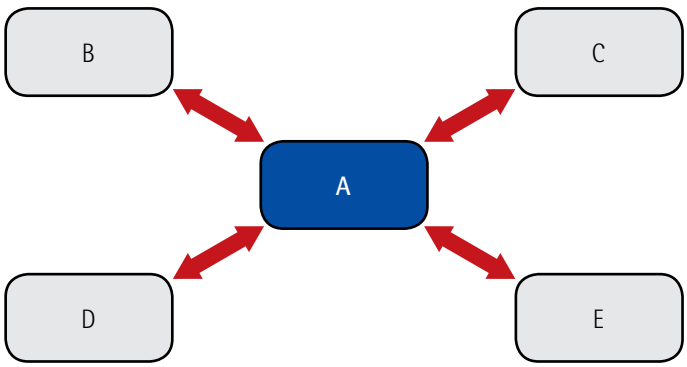

Figura 2 – Exemplo de dependência entre módulos

Imagine que o sistema tenha cinco módulos: A, B, C, D e E. Como indicado pelas setas, na figura anterior, temos as seguintes relações de dependência:

- "B" depende funcionalmente de "A".
- "C" depende funcionalmente de "A".
- "D" depende funcionalmente de "A".
- "E" depende funcionalmente de "A".

Uma característica marcante dessa estrutura, e que pode ser facilmente notada, é uma alta dependência do módulo A. Por exemplo, em uma situação hipotética de falha ou de algum tipo de manutenção no módulo A, todos os demais módulos seriam afetados diretamente, comprometendo, assim, todo o sistema.

No caso hipotético desse sistema, dizemos que ele tem um alto acoplamento e uma baixa coesão.

A definição do dicionário Michaelis (1998) para **acoplamento** que mais se aproxima do cenário de *software* é: "união de dois circuitos, com a finalidade de transferir energia de um para outro".

No nosso caso, acoplamento está relacionado à união de dois componentes, todavia com a finalidade de transferir informações de um para outro.

O mesmo dicionário (MICHAELIS, 1998) define **coeso** como algo "firmemente unido ou ligado".

Para *software* indicamos e buscamos uma alta coesão que sempre se dá pelo baixo acoplamento, ou seja, o ideal é o inverso do que se mostra na figura anterior.

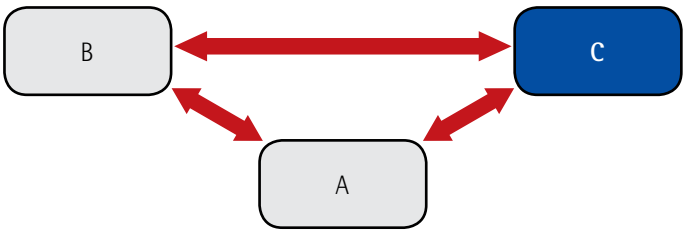

Figura 3 – Exemplo de dependência entre módulos

O mesmo efeito da Figura 2 pode ser notado na Figura 3, na qual temos três módulos: A, B e C, e com o relacionamento indicado pelas setas podemos chegar às seguintes conclusões:

- "A" depende funcionalmente de "B" e "C".
- "B" depende funcionalmente de "A" e "C".

Assim como na Figura 2, uma característica marcante dessa estrutura é a alta dependência em relação ao módulo C. Uma falha ou algum tipo de manutenção no módulo C e todos os demais módulos seriam afetados diretamente, comprometendo, assim, todo o sistema. Todavia existe algo ainda mais marcante nessa estrutura: uma dependência circular, o que igualmente provoca alto acoplamento e baixa coesão.

Nem sempre é fácil enxergar esses problemas no momento em que estamos pensando na solução do projeto, e, para isso, Pressman (2006), cita outros conceitos fundamentais de projeto, todos associados diretamente ou indiretamente a esses problemas, que são:

- Divisão por interesses: indica que um problema complexo pode ser mais facilmente resolvido se for dividido em pequenas partes ou pequenos problemas.
- Encapsulamento de informações: significa dizer que os módulos devem ser projetados de uma forma tal que as informações e sua lógica interna não sejam visíveis por outros módulos, a não ser que seja necessário. Em resumo, é deixar visível apenas o necessário.

Lembrete

Encapsulamento é um dos conceitos fundamentais da orientação a objetos, assim como: classe, objeto, abstração, herança e polimorfismo.

- Independência funcional: é uma das balizas fundamentais quando estamos querendo atingir acoplamento e coesão. Independência funcional está associada a projetar módulos que tenham funções bem-definidas, minimizando a dependência em relação a outros módulos. Em resumo, é dizer que um módulo não deve fazer mais nem menos.

2.4 Fases de projeto

Como debatemos anteriormente, o objetivo da fase de projetos é solucionar tecnicamente, ou dar solução, aos requisitos do usuário mapeados no modelo de requisitos.

As fases, ou subdivisão das atividades, da fase de projeto estão associadas ao que efetivamente deve ser produzido como artefato na fase de projeto. Pressman (2006) divide o modelo de projetos em quatro fases:

- Projeto de componentes.
- Projeto de interfaces.
- Projeto arquitetural.
- Projeto de dados/classes.

Conforme mostra a figura a seguir:

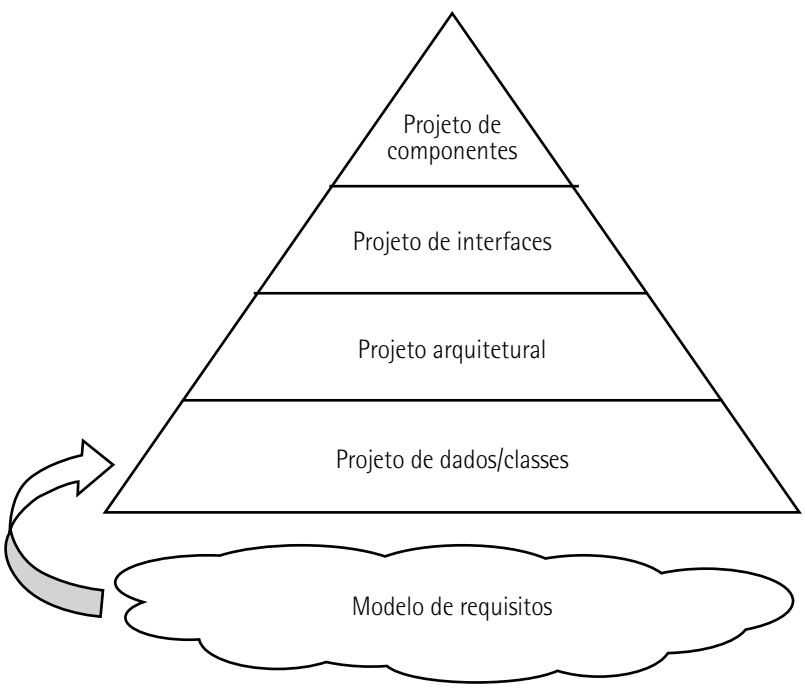

Figura 4 – Fases do modelo de projeto

A figura anterior mostra a sequência de atividades e o fluxo das informações da fase de projeto.

- Projeto de dados/classes: essa fase, que tem como insumo o modelo de requisitos (casos de uso, descrição de casos de uso, modelo de classe conceitual etc.), tem como objetivo a geração do modelo de dados e a transformação de classes e objetos conceituais em classes e objetos equivalentes em projeto.

- Projeto arquitetural: organiza as classes e objetos em componentes do *software* e define seus relacionamentos.

Observação

A discussão sobre arquitetura de *software* é mais ampla do que a simples organização de componentes e dos seus relacionamentos, como mostra o livro de Bass, Clements e Kazman (2010). Trataremos do assunto com mais profundidade nas próximas unidades.

- Projeto de interfaces: descreve todas as possíveis interfaces de um sistema, que podem ser: interfaces internas (como a comunicação entre os componentes será organizada), interfaces externas (comunicação do sistema com outros sistemas) e interfaces com o usuário.

Saiba mais

O projeto de interface com o usuário envolve muitos aspectos e vem sendo considerado cada vez mais como uma disciplina apartada chamada **usabilidade**, como é possível verificar em:

NIELSEN, J. *Usability engineering*. San Francisco: Morgan Kaufmann, 1993.

VERSOLATTO, F. R. *Uma abordagem arquitetural aplicada ao ciclo de vida da Engenharia da Usabilidade em prontuário eletrônico de pacientes*. 2012. Dissertação (Mestrado em Engenharia de *Software*) – Instituto de Pesquisas Tecnológicas de São Paulo (IPT), São Paulo, 2012.

- Projeto de componentes: a partir do modelo desenhado no projeto de arquitetura, refina-os e detalha-os de tal forma que seja possível a descrição procedimental desses componentes.

Neste livro-texto, trabalharemos com os artefatos produzidos em cada fase de projeto da metodologia proposta por Pressman (2006), porém polarizado para o paradigma da orientação a objetos.

2.5 Aspectos humanos da fase de projetos

Conforme vimos nas disciplinas anteriores, em todos os modelos de ciclo de vida, uma fase é composta por uma série de atividades, que produzem um determinado número de artefatos.

Estas atividades, além de produzirem algum resultado, são executadas por pessoas, ou papéis.

Lembrete

Um papel pode ser desempenhado por mais de uma pessoa, assim como uma pessoa pode desempenhar diferentes papéis ao longo do ciclo de vida do projeto.

Nesta seção iremos mapear os papéis envolvidos, diretamente ou indiretamente, na atividade de projeto, baseado no processo unificado especificado no OpenUP (OPENUP, [s.d.]a).

São eles:

Arquiteto

Quadro 1 – Atividades e habilidades do arquiteto no OpenUP

Atividades	• Conduz ou coordena o projeto técnico do sistema e tem a responsabilidade pelas decisões técnicas. • Identifica e documenta os aspectos significativos para a arquitetura do sistema como visões que descrevem requisitos, projeto, implementação e implantação. • Justifica as soluções técnicas equilibrando os interesses das várias partes envolvidas, reduzindo os riscos e garantindo que essas soluções sejam comunicadas, validadas e principalmente seguidas pela equipe de construção. • Trabalha junto com gerentes de projeto, recursos humanos e planejamento do projeto; o processo recomenda que a equipe seja organizada em torno da arquitetura. • Trabalha junto com os analistas e desenvolvedores para garantir que o guia da arquitetura seja seguido.
Habilidades	• Experiência nos domínios do problema e da engenharia de *software*. • Habilidade de liderança. • Habilidade de comunicação. • Habilidade de análise crítica, principalmente, para garantir que o desenvolvimento não fuja da arquitetura definida. • Proatividade com ênfase nos objetivos e nos resultados.

Adaptado de: OpenUP ([s.d.]b).

Analista

Quadro 2 – Atividades e habilidades do analista no OpenUP

Atividades	• Identifica e detalha os requisitos. • Descreve os casos de uso. • Auxilia no desenvolvimento da visão técnica, fornecendo os subsídios necessários.
Habilidades	• Experiência na identificação de problemas e soluções. • Capacidade de articular as necessidades que são associadas com o problema-chave a ser resolvido. • Capacidade de colaborar efetivamente com a equipe a partir de sessões colaborativas de trabalho. • Capacidade de comunicação verbal e escrita. • Conhecimento dos domínios de negócios e de tecnologia ou a capacidade de absorver e compreender essas informações rapidamente.

Adaptado de: OPENUP ([s.d.]c).

Gerente de projetos

Quadro 3 – Atividades e habilidades do gerente de projetos no OpenUP

Atividades	• Liderança da equipe para um bom resultado e da aceitação do produto por parte do cliente. • É responsável pelo resultado do projeto e da aceitação do produto por parte do cliente. • É responsável pela avaliação dos riscos do projeto e por controlar esses riscos por meio de estratégias de mitigação. • Aplica-se o conhecimento de gestão, habilidades, ferramentas e técnicas para uma ampla gama de tarefas para entregar o resultado desejado para um determinado projeto em tempo hábil.
Habilidades	• Capacidades de liderança e formação de equipes. • Experiência completa do ciclo de vida de desenvolvimento de *software* para treinar, orientar e apoiar outros membros da equipe. • Proficiência em resolução de conflitos e técnicas de resolução de problemas. • Bons conhecimentos em apresentação, facilitação, comunicação e negociação.

Adaptado de: OpenUP ([s.d.]d).

Além desses três papéis, envolvidos com a fase de projeto, existem os *stakeholders*, que são, segundo o OpenUP ([s.d.]c), grupos de interesse, cujas necessidades devem ser satisfeitas pelo projeto. É um papel que pode ser desempenhado por qualquer pessoa que é (ou potencialmente será) afetada pelo resultado do projeto.

Um papel secundário na fase de projetos, uma vez que sua ênfase de atuação é na fase de construção, é o do desenvolvedor, que na fase de projetos participa em duas frentes: entendimento da arquitetura e sugestão de soluções técnicas.

Saiba mais

O OpenUP é uma variante de processo unificado que determina um modelo de ciclo de vida iterativo e incremental. Para aprofundar seus conhecimentos, acesse:

<http://epf.eclipse.org/wikis/openup/>.

2.6 O que buscamos atingir no projeto?

Pressman (2006), citando McGlaughlin (1991), coloca três metas fundamentais da fase de projeto:

- O projeto deve contemplar a implementação de todos os requisitos explícitos, e aí uma observação importante e que muitas vezes é deixada de lado, o projeto deve acomodar, ou deixar apto a receber, todos os requisitos implícitos.
- O projeto deve ser um guia para desenvolvedores, testadores e para aqueles de darão suporte ao *software*.
- O projeto deve fornecer uma visão completa do *software* sempre tendo como norte o aspecto implementação (MCGLAUGHLIN, 1991, p. 209 *apud* PRESSMAN, 2006).

Lembrete

Na Engenharia de Requisitos não é tão incomum um requisito não ser mapeado, ou ser mapeado incompletamente. Isso pode ser devido a uma série de fatores, dentre eles, por exemplo, uma falha de comunicação entre o analista de requisitos e o usuário.

Além desses três pontos enumerados por Pressman (2006), a norma ISO 25010 (ISO, 2011a) trata da classificação e da definição de requisitos não funcionais, mas que também definem um modelo de qualidade utilizado como referência para a avaliação de qualidade de *software*.

Observação

O International Organization for Standardization (ISO) é um grupo fundado em 1947, sem relações com órgãos governamentais, localizado na cidade de Genebra, na Suíça, com o objetivo de definir diretrizes normativas.

A norma descreve seis características que definem a qualidade de *software*: funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade. Essas características, também denominadas atributos de qualidade, são comumente usadas quando trabalhamos com requisitos não funcionais.

O quadro a seguir mostra a definição dos atributos de qualidade de acordo com a ISO 25010.

Quadro 4 – Atributos de qualidade segundo a ISO 25010

Atributo	Descrição
Funcionalidade	Está ligado à capacidade do sistema de *software* de prover funcionalidades que atendam às necessidades explícitas e implícitas quando usado sob as condições especificadas.
Confiabilidade	Está ligado à capacidade do sistema de *software* de manter um determinado nível de desempenho quando usado sob as condições especificadas.
Usabilidade	Está ligado à capacidade do sistema de *software* de auxiliar os usuários na realização de suas tarefas, de maneira produtiva (ROCHA; BARANAUSKAS, 2003).
Eficiência	Está ligado à capacidade do sistema de *software* de prover desempenho apropriado, relativo à quantidade de recursos utilizados.
Manutenibilidade	Está ligado à capacidade do sistema de *software* de ser modificado, e essa modificação pode ser uma correção, melhoria ou adaptação.
Portabilidade	Está ligado à capacidade do sistema de *software* de ser portável entre plataformas de ambientes.

Adaptado de: ISO (2011a, p.16-21).

A figura a seguir mostra a estrutura hierárquica dos atributos de qualidade, quanto à sua classificação, proposta pela norma ISO 25010 (ISO, 2011a).

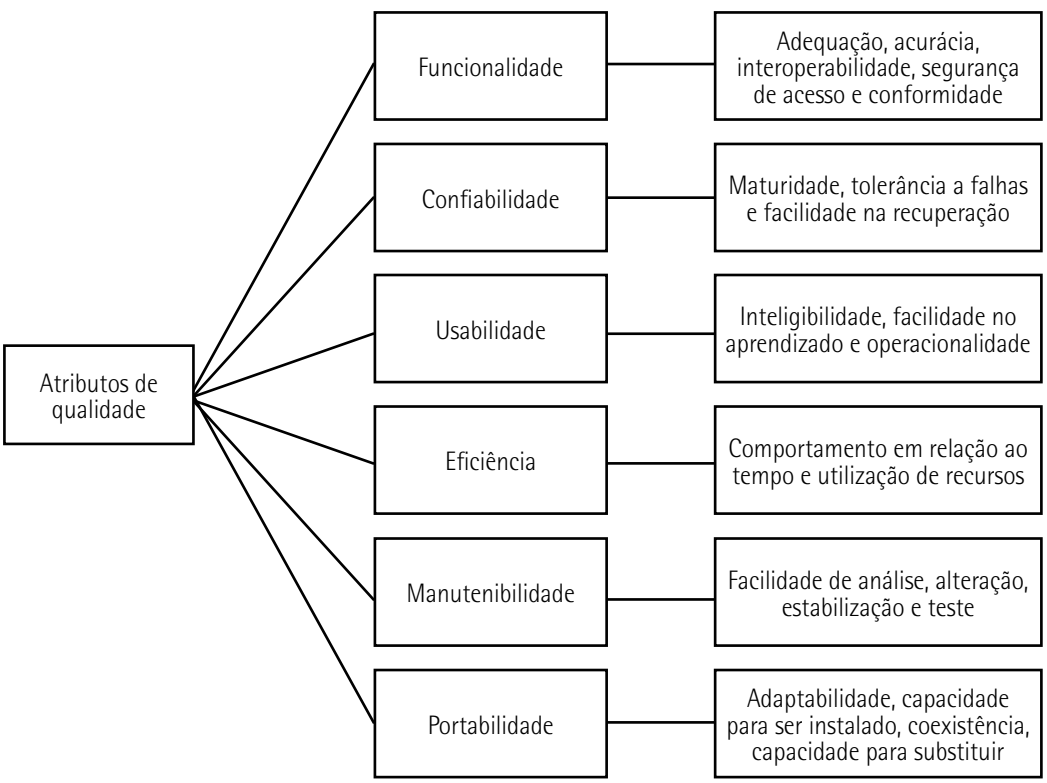

Figura 5 – Requisitos não funcionais, segundo a classificação da ISO 25010

Conforme ilustrado, os seis atributos de qualidade – funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade – também possuem uma estratificação para outros atributos de qualidade, conforme citado também por Cortês e Chiossi (2001).

- Funcionalidade
  - Adequação: é a garantia de que o *software* possui as funções que foram definidas.
  - Acurácia: está relacionada ao grau de precisão dos resultados gerados pelo *software*.
  - Interoperabilidade: é a capacidade do *software* de interagir com outros sistemas.
  - Segurança de acesso: está ligada à capacidade do *software* de prevenir acessos por pessoas ou aplicações não autorizadas.
  - Conformidade: se o *software* foi desenvolvido seguindo os padrões, convenções ou regras preestabelecidas no projeto.
- Confiabilidade
  - Maturidade: está relacionada à baixa frequência de falhas ou defeitos do *software* em operação.
  - Tolerância a falhas: está relacionada a um determinado nível de desempenho que o *software* deve atingir, mesmo em momentos de problemas.
  - Facilidade na recuperação: está ligada à capacidade do *software* de se restabelecer, e de restabelecer seus dados e parâmetros sem que haja necessidade de intervenção, após uma falha.
- Usabilidade
  - Inteligibilidade: medida da facilidade do usuário para reconhecer a lógica de funcionamento do *software*.
  - Facilidade no aprendizado: medida da facilidade encontrada pelo usuário para aprender a utilizar o *software*.
  - Operacionalidade: medida da facilidade para operar o produto com segurança e sem falhas.
- Eficiência
  - Comportamento em relação ao tempo: está relacionado ao tempo de resposta e de processamento do *software* em seus serviços.

– Utilização de recursos: está relacionada à quantidade de recursos utilizados (CPU, memória, processador) e ao tempo de resposta e de processamento do *software* em seus serviços.

- Manutenibilidade

  – Facilidade na análise: medida do esforço necessário para diagnosticar falhas e localizar as partes para corrigir os problemas.

  – Facilidade na alteração: medida do esforço necessário para corrigir as falhas ou adequar o produto a eventuais mudanças.

  – Facilidade na estabilização: medida do esforço necessário em se estabilizar o *software* após as alterações.

  – Facilidade no teste: medida do esforço necessário para testar o produto de *software* alterado.

- Portabilidade

  – Adaptabilidade: está ligada à facilidade em se adaptar o *software* para funcionar em outros ambientes.

  – Capacidade para ser instalado: medida do esforço necessário para instalar o *software* em um ambiente de operação.

  – Coexistência: está relacionado ao nível de conformidade do *software* a padrões referentes à portabilidade.

  – Capacidade para substituir: medida do esforço necessário em substituir um *software* por outro previamente especificado.

Os atributos de qualidade listados na ISO 25010 (2011a) também são tratados na engenharia de requisitos como requisitos não funcionais. É apenas uma questão de mudança de enfoque: enquanto esses atributos de qualidade, na engenharia de requisitos, são tratados como requisitos não funcionais, no projeto, são tratados como diretivas de qualidade de projeto.

Saiba mais

A qualidade de *software* é disciplina extensa e muito importante para ser debatida apenas no contexto deste livro-texto. Veja no trabalho de Guerra e Colombo (2009) como a norma ISO 9126 (que é a norma predecessora da ISO 25010) pode ser aplicada na avaliação da qualidade:

GUERRA, A. C.; COLOMBO, R. M. T. *Tecnologia da informação*: qualidade de produto de *software*. Brasília: PBQP, 2009.

2.7 Introdução ao projeto orientado a objetos

O projeto orientado a objetos partilha exatamente dos mesmos princípios e objetivos que debatemos até agora, ou seja, possui as mesmas fases, é baseado na produção de modelos, é fundamentado nos mesmos conceitos básicos, aspectos humanos e objetivos.

Mas então qual é a diferença?

A diferença fundamental está na utilização do paradigma da orientação a objetos e de seus conceitos fundamentais.

> Um paradigma é um conjunto de regras que ajuda-nos a organizar e a coordenar a maneira como olhamos o mundo entre o que é certo e errado, entre o que é verdadeiro e o que é falso, entre o que se deve fazer e o que não se deve fazer. [...] Ele funciona como um conjunto de categorias que define e orienta o comportamento das pessoas (CHIAVENATO, 2008, p. 8).

O paradigma da orientação a objetos é uma forma de se desenvolver um sistema de *software* que o enxerga como um conjunto de componentes que interagem para resolver um determinado problema. A cada componente, dá-se o nome de objeto.

A motivação da abordagem orientada a objetos se dá pela tentativa de aproximar o desenvolvimento de *software* daquilo que acontece no mundo real.

Vamos relembrar um pouco os conceitos fundamentais do paradigma da orientação a objetos. Podemos dizer que esses conceitos são básicos para conseguirmos desenvolver a fase de projeto.

O paradigma da orientação a objetos é baseado nos seguintes pilares.

Classes

Podemos definir classe, ou classe de objetos, como um grupo de objetos com iguais propriedades (atributos), comportamento (operações), relacionamentos e semântica.

Uma classe deve possuir responsabilidades bem-definidas, e cada responsabilidade representa um contrato ou obrigações dela.

Semanticamente, dizemos que uma classe é uma especificação de um objeto, pois nela está definido tudo o que o objeto possui (atributos) e tudo aquilo que o objeto pode fazer (métodos).

Objeto

Podemos afirmar que todo objeto possui uma identidade, representada por um conjunto de informações conferidas aos seus atributos, e ainda que todo objeto possui um estado, definido também pelo conjunto dessas informações em um determinado espaço de tempo.

Encapsulamento

Encapsulamento significa deixar visível, ou deixar acessível a outros objetos, apenas o que é necessário.

Por exemplo, podemos ocultar dos demais objetos de um sistema detalhes da implementação ou a lógica algorítmica de um método de um determinado objeto, uma vez que esses detalhes não são importantes para os demais, apenas para o objeto que possui essa lógica.

Para os demais objetos que se comunicam com esse objeto, basta que ele faça o que deve fazer, não sendo importante o **como** será feito.

Abstração

Abstração é um dos principais conceitos aplicados à resolução de problemas, que é fundamental para a modelagem da estrutura de um sistema de *software*.

Está ligada a nossa capacidade de selecionar determinados aspectos do problema e isolar o que é importante para algum propósito do que não for para um determinado propósito, ou seja, é dar ênfase àquilo que é necessário. Um conceito que parece simples à primeira vista, entretanto faz toda a diferença na resolução e na modelagem de sistemas complexos.

Na modelagem de um sistema de *software*, abstração está relacionada à nossa capacidade, enquanto analistas, desenvolvedores e arquitetos, de estabelecer um modelo de objetos que resolva o problema da melhor forma possível, isto é, não basta resolver o problema, é preciso pensar em cada objeto como uma unidade autônoma, ou seja, fazendo única e exclusivamente aquilo que precisa fazer e tendo apenas as dependências que deve ter.

A melhor forma de se representar a estrutura estática de um sistema orientado a objetos é utilizando o modelo de classes. São três os modelos utilizados, segundo Bezerra (2006).

- Modelo de classe de domínio: desenvolvido na fase de análise, o modelo de classe de domínio representa os objetos, ou classes, inerentes ao domínio do problema que queremos resolver, deixando de lado, nesta visão, detalhes tecnológicos da solução do problema.

- Modelo de classe de especificação: construído na fase de projeto, o modelo de classe de especificação adiciona ao modelo de classes de domínio, objetos ou classes específicas para a solução do problema sob o aspecto tecnológico, ou seja, é uma extensão do modelo de classe de domínio.

- Modelo de classe de implementação: o modelo de implementação nada mais é que a implementação das classes, especificadas no modelo de especificação, construídas ou codificadas em alguma linguagem de desenvolvimento orientada a objetos, como a linguagem Java ou a linguagem C#.

Observação

Atualmente existe uma distorção grande quando falamos de orientação a objetos e linguagem de desenvolvimento, ou programação orientada a objetos. Como vimos, orientação a objetos é um paradigma, não uma linguagem. Todavia, as linguagens possibilitam ao desenvolvedor implementar um *software* baseado nesse paradigma.

Dentro do ciclo de vida da engenharia de *software*, nas atividades ou fases anteriores à fase de projeto, a ênfase é dada ao modelo de classe de domínio e à elicitação de requisitos nos quais eram produzidos artefatos utilizando como suporte, principalmente, a Unified Modeling Language (UML) como uma ferramenta para representação e modelagem de um sistema de *software*.

Saiba mais

A especificação e a definição completas da UML estão disponíveis no *site* da OMG:

<http://www.uml.org/>.

Pois bem, no paradigma da orientação a objetos, a fase de projeto utiliza como espinha dorsal todos os princípios da fase de projetos que vimos até agora, soma-se aos conceitos fundamentais da orientação a objetos e adentra o modelo de classes de especificação apresentados por Bezerra (2006).

O objetivo da fase de projetos no paradigma da orientação a objetos é dar sequência ao modelo de classes de domínio, transformando as classes que possuem alto nível de abstração, utilizadas fundamentalmente para o entendimento do problema, em classes de projeto.

Classes de projeto é um nível mais baixo de abstração, ou, podemos dizer, um nível mais refinado das classes de análise, que possuem maiores detalhes de implementação, de infraestrutura que suportarão a concretização das regras de negócio (PRESSMAN, 2006).

Assim como os componentes de *software*, que citamos anteriormente, as classes (que também podem ser enxergadas como componentes em outro nível de abstração) devem ser e possuir, respectivamente: completa e suficiente, ou seja, devem possuir nível suficiente de encapsulamento e de número de funcionalidade. Não deve fazer mais, nem menos; alta coesão e baixo acoplamento.

Ambler (2001 *apud* Pressman, 2006, p. 219) enumera os cinco tipos de classes de projetos que podem ser desenvolvidas na fase de projetos:

- Classes de interface do usuário: definem a interação homem-máquina ou IHC (interação homem-computador).
- Classes de domínio de negócio: são refinamentos das classes do modelo de domínio (ou de análise). Possuem atributos e métodos utilizados para implementar elementos do domínio do negócio.
- Classes de processos: têm como objetivo gerir as classes de domínio de negócio.
- Classes persistentes: têm como objetivo a persistência de informações em um repositório de dados, por exemplo, um banco de dados.
- Classes de sistema: têm como objetivo gerir a comunicação do *software* com o seu ambiente computacional interno e externo.

Coad e Yourdon (1993) têm uma visão semelhante, mas polarizada um pouco mais para a visão arquitetural; enxergando os componentes de um *software* como uma composição de classes, definem quatro tipos de componentes:

- Componente do domínio do problema: são componentes responsáveis por implementar os requisitos dos usuários.
- Componente de interação humana: são componentes responsáveis por implementar a interação homem-máquina ou IHC (interação homem-computador).
- Componente de gerência de tarefa: são responsáveis por controlar e coordenar tarefas do *software*.
- Componente de gerência de dados: são responsáveis por controlar a persistência dos objetos em um repositório de dados.

Nas próximas unidades iremos trabalhar como modelar e representar a evolução do modelo de domínio para o modelo de projetos, como iremos representar os tipos de classe de projeto e componentes de projeto, associando aos artefatos que devem ser produzidos, tendo como suporte os diagramas da UML.

Resumo

Esta unidade teve como objetivo apresentar os conceitos introdutórios e fundamentais sobre projeto de sistemas no paradigma da orientação a objetos.

Iniciou-se com o debate e o convite a debatermos sobre a real motivação em seguir um modelo de projetização.

Traçando um paralelo, vimos que, muito em função do seu tempo de existência, incomparavelmente maior, a engenharia civil está muito mais avançada no aspecto projeto que a engenharia de *software*.

Construir um *software* sem um projeto é como construir uma casa sem um projeto, que engloba, dentre outras coisas, um conjunto de plantas e modelos que representam a casa e servem de guia para essa construção.

Na engenharia de *software* temos diversas opções de modelos de processo, ou de ciclo de vida, para desenvolvimento de projetos de *software*, por exemplo, o Modelo Cascata (*waterfall*), o Modelo Iterativo Incremental, Modelo Espiral e o Processo Unificado (UP).

Esses modelos definem e direcionam uma sequência de fases e de atividades realizadas dentro de cada fase com o objetivo de organizar o desenvolvimento de um produto de *software*.

Qualquer que seja o modelo de processo adotado, a fase de projetos está sempre entre a fase de elicitação e análise de requisitos e a construção ou desenvolvimento.

O objetivo da fase de projetos é desenhar a solução para os problemas levantados na engenharia de requisitos, desenho esse que serve como guia para a construção.

A fase de projetos não é iniciada até que todos os requisitos, ou parte deles, sejam elicitados, documentados e aprovados. Depende do modelo de ciclo de vida adotado, em que poderão ser mapeados e validados todos os requisitos do sistema, como no Modelo Tradicional ou apenas uma parte desses requisitos, como no Modelo Iterativo Incremental.

No Modelo Cascata, por exemplo, todos os requisitos são levantados, e após isso se inicia a fase de projetos, enquanto em um Modelo Iterativo Incremental, a cada iteração, uma fatia do total dos requisitos é elicitada e validada para depois se iniciarem as atividades da fase de projetos.

Vimos que, assim como na engenharia civil, que possui plantas de casas como modelos, a técnica de modelagem é fundamental na fase de projeto de um *software*.

Um modelo é uma abstração, uma representação de uma perspectiva do sistema, utilizada para dar ênfase a aspectos importantes, deixando aspectos menos importantes de lado para facilitar o entendimento.

Modelos são fundamentais, pois ajudam no gerenciamento da complexidade, na comunicação entre os envolvidos no projeto, reduzem os custos do desenvolvimento e simulam o comportamento do sistema, servindo, assim, como um laboratório.

Dois são os conceitos fundamentais quando estamos modelando um sistema na fase de projetos: abstração e modularidade.

Abstração, que também é um dos conceitos fundamentais da orientação a objetos, está ligada a nossa capacidade, enquanto analistas e projetistas, de enxergar aspectos relevantes à solução de um problema no seu devido tempo.

Modularidade é a divisão do *software* em componentes, ou módulos, que interagem para resolver um determinado problema. Todavia, a simples modularização não resolve os desafios de se produzir um *software* com qualidade.

O que se busca com a modularidade é: baixo acoplamento e alta coesão. Isso se dá a partir do uso de técnicas de encapsulamento e de uma divisão de responsabilidade de cada módulo, de tal forma que eles tenham funções bem-definidas. Cada componente não deve "fazer demais", nem "fazer pouco".

Dentro da fase de projeto, temos uma subdivisão em quatro atividades (ou fases): projeto de componentes, projeto de interfaces, projeto arquitetural e projeto de dados/classes; cada fase é delimitada pelos artefatos que produz.

Vimos que alguns dos principais envolvidos na fase de projetos são: o arquiteto, o analista, o gerente de projetos, os *stakeholders* e os desenvolvedores, estes últimos em menor grau nessa fase.

São três os objetivos a serem atingidos pelos profissionais envolvidos na fase de projetos: o modelo do *software* deve ser um guia não só para a implementação, que é o principal, mas para todas as fases subsequentes; deve fornecer uma visão completa da solução; e deve contemplar todos os requisitos explícitos mapeados na engenharia de requisitos e, principalmente, estar pronto, ou apto, aos possíveis, e prováveis, requisitos implícitos.

Além desses três objetivos, vimos que a norma ISO 25010 (2011a) utilizada na engenharia de requisitos para determinar um conjunto de requisitos não funcionais, é utilizada na fase de projetos como uma baliza para a avaliação da qualidade de um produto de *software*, em um mercado cada vez mais competitivo.

O projeto orientado a objetos utiliza-se da espinha dorsal de todos os princípios e objetivos de projeto e, somando-se aos pilares do paradigma da orientação a objetos, refina o modelo de classes de domínio, obtido na fase de análise com o objetivo de montar um modelo de classes que possa efetivamente ser construído.

O objetivo das próximas unidades é construir o modelo de classes de projeto, com todos os tipos de classes: de domínio de negócio, de interface do usuário, de processos, persistentes e de sistema e seus respectivos artefatos, tendo como base o paradigma da orientação a objetos e seus conceitos fundamentais e a UML como ferramenta de apoio.

Exercícios

Questão 1. O modelo cascata, também conhecido como ciclo de vida clássico, requer uma abordagem sistemática e sequencial. Esta abordagem apresenta algumas deficiências que são, em parte, endereçadas a outros modelos, como o espiral e os atuais métodos ágeis.

Considerando as características do modelo cascata, avalie as afirmativas a seguir:

I – Em projetos complexos demora-se muito para se obter o *software* em execução.

II – Erros ocorridos nas etapas de levantamento e análise só são percebidos em fases avançadas, o que maximiza o custo de correção destes.

III – Dificilmente consegue-se listar todos os requisitos do *software* logo no início do projeto.

IV – Este modelo pressupõe interação e entrega de pequenos módulos e pacotes de *software* aos usuários, na medida em que o projeto avança.

V – Uma versão executável do *software* só é disponível em uma fase avançada do modelo.

É correto apenas o que se afirma em:

A) I e II.

B) II e IV.

C) I, II, III e V.

D) I, II e III.

E) II, III e V.

Resposta correta: alternativa C.

Análise das afirmativas

I – Afirmativa correta.

Justificativa: o modelo cascata pressupõe que uma fase ou etapa termine para que a outra comece, desta forma o *software* **só é disponibilizado para uso ao final do projeto de desenvolvimento**.

II – Afirmativa correta.

Justificativa: os testes, atividades que permitem que o usuário tenha contato com o sistema, são realizados ao final da fase de codificação e na fase de testes. Estas vêm após as fases iniciais de levantamento e análise. Sendo assim, o acerto de erros nas primeiras fases pressupõe ajustes em quase todo o ciclo de vida transcorrido até o momento.

III – Afirmativa correta.

Justificativa: por mais amplo que seja o levantamento de requistos é comum que novos requsitos apareçam durante o projeto. Quanto mais longo for o projeto maior é a probabilidade do aparecimento de novos requisitos.

IV – Afirmativa incorreta.

Justificativa: este modelo **não** pressupõe interação e entrega de pequenos módulos e pacotes de *software* aos usuários na medida em que o projeto avança.

V – Afirmativa correta.

Justificativa: pela característica de ser sequencial, somente em uma fase avançada é que se disponibiliza uma versão executável do *software*.

Questão 2. O paradigma da orientação a objetos é uma forma de se desenvolver um sistema de *software* que o enxerga como um conjunto de componentes que interagem entre si para resolver um determinado problema.

De acordo com este paradigma, cada componente é chamado de:

A) Classe.

B) Classe persistente.

C) Método.

D) Objeto.

E) Herança.

Resolução desta questão na plataforma.