NILSON JÚNIOR

APOSTILA DE ENGENHARIA DE SOFTWARE

1ª edição

Jaboatão dos Guararapes

Edição do autor

2015

APOSTILA DE ENGENHARIA DE SOFTWARE

[Type the document subtitle]

Esta apostila visa auxiliar os alunos do curso de informática para internet no aprendizado das competências necessárias da Engenharia de Software.

Jaboatão dos Guararapes,

Setembro de 2015

Índice

CAPÍTULO 1

INTRODUÇÃO À ENGENHARIA DE SOFTWARE

Neste primeiro capítulo aprenderemos alguns conceitos básicos relacionados ao surgimento da engenharia de software, suas definições e o campo de atuação.

1.1 - ENGENHARIA DE SOFTWARE

O termo engenharia de software surgiu no final da década de 1960, mais precisamente em 1968, em uma conferência organizada para discutir o que era chamado de crise de software. **Essa crise teve sua origem em função do hardware, que nessa época tinha seu poder de processamento e memória aumentados, sendo que o software não acompanhava esse mesmo ritmo**. Foi percebido na época, que o desenvolvimento de grandes sistemas era desenvolvido de maneira informal, sem seguir regras ou etapas pré-definidas, não era suficiente (SOMMERVILLE, 2007).

Nesta fase os softwares desenvolvidos não eram confiáveis, e não eram desenvolvidos no tempo e custo previsto, sua qualidade e desempenho não eram satisfatórios e era muito difícil para realizar a sua manutenção. Os custos em relação aos hardwares estavam caindo, em função de que a sua produção passou a ser em série, porém, o custo para o desenvolvimento de software ainda não acompanhava essa queda, e muitas vezes, sendo aumentado.

A finalidade da disciplina de engenharia de software seria tornar o desenvolvimento de software um processo sistematizado, em que seriam aplicadas técnicas e métodos necessários para controlar a complexidade inerente aos grandes sistemas (SOMMERVILLE, 2007, p. 4).

Por ser fácil escrever programas de computador sem usar técnicas e métodos de engenharia de software, isso forçou a muitas empresas a desenvolver softwares à medida que seus produtos e serviços evoluíram. Como consequência, e por não utilizar métodos de engenharia de software no dia-a-dia, seus softwares são mais caros e menos confiáveis do que deveriam ser.

Várias pessoas desenvolvem softwares hoje em dia, para os mais diferentes meios (científico, hobby, empresariais), no entanto a maior parte dos softwares é desenvolvido para um propósito específico de negócio. O software profissional, que é usado por alguém além do seu desenvolvedor, é normalmente criado por equipes, em vez de indivíduos. Ele é mantido e alterado durante sua vida. A engenharia de software inclui técnicas que apoiam especificação, projeto e evolução dos programas.

Falar em engenharia de software não é apenas criar programas de computador, mas também engloba toda a documentação associada e dados de configuração necessários para fazer esse programa operar corretamente. Um sistema de software desenvolvido profissionalmente é mais que apenas um programa; ele é normalmente uma série de programas separados e arquivos de configuração usados para configurar esses programas. Isso inclui documentação do sistema (descreve a sua estrutura),

documentação do usuário (explica como usar o sistema), e sites para o usuário baixar informações recentes do produto.

Engenheiros de software preocupam-se em desenvolver produtos de softwares (softwares que podem ser vendidos). Existem dois tipos de produtos de software:

- **Produtos genéricos**: São os sistemas chamados stand-alone, que são os que são vendidos para qualquer cliente que esteja interessado em comprá-los. Exemplos: ferramentas de banco de dados, processadores de texto, pacotes gráficos, etc. E também as aplicações verticais, tais como, sistemas de biblioteca, contabilidade, e manutenção de registros de alguma empresa.

- **Produtos sob encomenda**: São criados para atender um fim específico e um cliente particular. É desenvolvido especificamente para esse cliente. Exemplo: sistema de controle de tráfego aéreo.

O que é um Software?	R: São programas de computador e documentação associada. Quais os atributos de um bom software? R: Prover
Quais os atributos de um bom software?	R: Prover a funcionalidade e o desempenho requerido pelo usuário; deve ser confiável, fácil de manter e usar
Quais as principais atividades da engenharia de software?	R: Especificação de software, desenvolvimento de software, validação de software e evolução de software.
Quais os principais desafios da engenharia de software?	R: Lidar com o aumento de diversidade, demandas pela diminuição do tempo para a entrega e desenvolvimento de software confiável

Sobre a qualidade dos softwares profissionais, devemos lembrar que o software é alterado pelas pessoas, além dos seus desenvolvedores. A qualidade não é apenas o que o software faz, mas também como ele se comporta quando está sendo executado, bem como a estrutura e a organização dos programas do sistema e a documentação associada. Isso reflete nos atributos de software, que são chamados de não funcionais ou de qualidade. Exemplos desses atributos: Tempo de resposta a uma consulta realizada pelo usuário e a compreensão do código do programa. A seguir uma tabela com as características essenciais de um sistema profissional de software:

Manutenibilidade	O software deve ser escrito de forma que possa evoluir para atender às necessidades do cliente. Esse é um atributo crítico, porque a mudança de software é um requisito inevitável de um ambiente de negócios em mudança
Confiança e proteção	Inclui uma série de características como: confiabilidade, proteção e segurança. O software não pode causar prejuízos físicos ou financeiros no caso de falha no sistema e usuários maliciosos não devem ser capazes de acessar ou prejudicar o sistema
Eficiência	O software não deve desperdiçar os recursos de sistema, como memória e ciclos do processador. Deve ser eficiente na capacidade de resposta, tempo de processamento, uso de memória, etc.
Aceitabilidade	O software deve ser aceitável para o tipo de usuário para o qual foi projetado. Isso significa que ele deve ser compreensível, usável e compatível com outros sistemas usados por ele

A engenharia de software não se preocupa apenas com os processos técnicos do desenvolvimento de software. Ela também inclui atividades como gerenciamento de projeto de software e desenvolvimento de ferramentas, métodos e teorias para apoiar a produção de software. Engenharia tem a ver com obter resultados de qualidades requeridos dentro do cronograma e do orçamento.

A abordagem sistemática usada na engenharia de software é, às vezes, chamada processo de software. Um processo de software é uma sequência de atividades que leva à produção de um produto de software. Existem quatro atividades fundamentais comuns a todos os processos de software. São elas:

1. **Especificação de software**: Clientes e engenheiros definem o que será produzido e as suas restrições de operação.

2. **Desenvolvimento de software**: projeto e programação.

3. **Validação de software**: O software é verificado para garantir que está de acordo com o que o cliente quer.

4. **Evolução de software**: O software é modificado para atender a mudança de requisitos do cliente e do mercado.

Além das quatro atividades fundamentais também existem três aspectos gerais que afetam vários tipos diferentes de software:

1. **Heterogeneidade**: O sistema deve rodar em diferentes tipos de computadores e dispositivos móveis. Integração entre sistemas novos e antigos e escritos em linguagens de programação diferentes. Desenvolver um software confiável que seja flexível o suficiente para lidar com essa heterogeneidade.

2. **Mudança de negócio e social**: As técnicas de desenvolvimento devem evoluir junto com a sociedade e o surgimento de novas tecnologias para que o tempo requerido para o software do retorno a seus clientes seja reduzido também.

3. **Segurança e confiança**: Precisamos ter certeza de que os usuários maliciosos não possam atacar nosso software e de que a proteção da informação seja mantida.

Engenharia de software é uma abordagem sistemática para a produção de software; ela analisa questões práticas de custo, prazo e confiança, assim como as necessidades dos clientes e produtores do software. A forma como ela é realmente implementada varia de acordo com a organização que está desenvolvendo, o tipo de software e as pessoas envolvidas no processo de desenvolvimento. Não existe um método global de desenvolvimento, mas um conjunto diverso de métodos e ferramentas.

1.2 - MITOS

É possível apontar, como causas principais dos problemas levantados na seção anterior, três principais pontos:

• a falta de experiência dos profissionais na condução de projetos de software;

• a falta de treinamento no que diz respeito ao uso de técnicas e métodos formais para o desenvolvimento de software;

• a “cultura de programação” que ainda é difundida e facilmente aceita por estudantes e profissionais de Ciências da Computação;

• a incrível "resistência" às mudanças (particularmente, no que diz respeito ao uso de novas técnicas de desenvolvimento de software) que os profissionais normalmente apresentam.

Entretanto, é importante ressaltar e discutir os chamados "mitos e realidades" do software, o que, de certo modo, explicam alguns dos problemas de desenvolvimento de software apresentados.

Mitos de Gerenciamento

Mito 1. "Se a equipe dispõe de um manual repleto de padrões e procedimentos de desenvolvimento de software, então a equipe está apta a encaminhar bem o desenvolvimento."

Realidade 1. Isto verdadeiramente não é o suficiente... é preciso que a equipe aplique efetivamente os conhecimentos apresentados no manual... é necessário que o que conste no dado manual reflita a moderna prática de desenvolvimento de software e que este seja exaustivo com relação a todos os problemas de desenvolvimento que poderão aparecer no percurso...

Mito 2. "A equipe tem ferramentas de desenvolvimento de software de última geração, uma vez que eles dispõem de computadores de última geração."

Realidade 2. Ter à sua disposição o último modelo de computador (seja ele um mainframe, estação de trabalho ou PC) pode ser bastante confortável para o desenvolvedor do software, mas não oferece nenhuma garantia quanto à qualidade do software desenvolvido. Mais importante do que ter um hardware de última geração é ter ferramentas para a automatização do desenvolvimento de software (as ferramentas CASE).

Mito 3. "Se o desenvolvimento do software estiver atrasado, basta aumentar a equipe para honrar o prazo de desenvolvimento."

Realidade 3. Isto também dificilmente vai ocorrer na realidade, alguém disse um dia que "acrescentar pessoas em um projeto atrasado vai torná-lo ainda mais atrasado". De fato, a introdução de novos profissionais numa equipe em fase de condução de um projeto vai requerer uma etapa de treinamento dos novos elementos da equipe; para isto, serão utilizados elementos que estão envolvidos diretamente no desenvolvimento, o que vai, consequentemente, implicar em maiores atrasos no cronograma.

Mitos do Cliente

Mito 4. "Uma descrição breve e geral dos requisitos do software é o suficiente para iniciar o seu projeto... maiores detalhes podem ser definidos posteriormente."

Realidade 4. Este é um dos problemas que podem conduzir um projeto ao fracasso, o cliente deve procurar definir o mais precisamente possível todos os requisitos importantes para o software: funções, desempenho, interfaces, restrições de projeto e critérios de validação são alguns dos pontos determinantes do sucesso de um projeto.

Mito 5. "Os requisitos de projeto mudam continuamente durante o seu desenvolvimento, mas isto não representa um problema, uma vez que o software é flexível e poderá suportar facilmente as alterações."

Realidade 5. É verdade que o software é flexível (pelo menos mais flexível do que a maioria dos produtos manufaturados). Entretanto, não existe software, por mais flexível que suporte alterações de requisitos significativos com adicional zero em relação ao custo de desenvolvimento. O fator de multiplicação nos custos de desenvolvimento do software devido a alterações nos requisitos cresce em função do estágio de evolução do projeto, como mostra a (Figura 1.1).

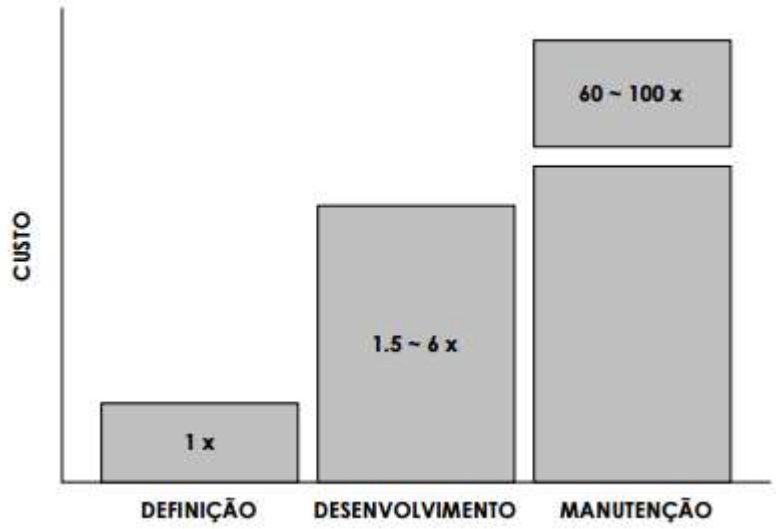

FIGURA 1.1- LINHA DO TEMPO DOS CUSTOS DE UM PROJETO DE SOFTWARE

1.3 – EXERCÍCIOS

1) Qual a finalidade da Engenharia de Software?.

2) Quais as características essenciais de um sistema profissional de software?.

3) Cite as quatro atividades fundamentais comuns aos Processos de Software?.

4) Além das quatro atividades fundamentais também existem três aspectos gerais que afetam vários tipos diferentes de software, quais são esses aspectos?.

5) Quais os principais mitos existente no desenvolvimento de software?.

6) Se um projeto de software tiver modificação durante a fase de desenvolvimento, qual a escala de aumento nos custos que o projeto pode sofrer?.

REFERÊNCIAS

IAN SOMMERVILLE "Engenharia de Software", 8ª Edição, Addison-Wesley, São Paulo, 2007.

CAPÍTULO 2

PROCESSOS DE DESENVOLVIMENTO DE SOFTWARE

Neste capítulo aprenderemos o que é de fato um processo, e quais são os modelos e processos de desenvolvimento de software existentes na literatura, que são fundamentais para o entendimento do ciclo de vida de um software.

2.1 - INTRODUÇÃO

Antes de descobrir o que é modelo de desenvolvimento de Software, o que é processo de Software?

Para Sommerville um processo de software é um conjunto de atividades e de resultados que associados produzem um produto de software. Então podemos concluir que um processo de software é um conjunto de atividades que resulta num produto software. Um processo deve contribuir para reduzir os custos, e aumentar a qualidade de produção. Um processo que não dar suporte a esses objetivos não pode ser considerado um processo adequado.

Atividades comuns a todos os processos:

- Especificação: Definição do que será produzido;
- Desenvolvimento: Projetado e programado;
- Validação: Verificação se o que foi feito é o que o cliente deseja.
- Evolução: Adaptação as mudanças futuras e/ou melhorias;

Diferentes tipos de sistemas necessitam de diferentes tipos de desenvolvimento. Já que temos exemplos: O cliente pode requerer que sejam especificados os requisitos antes do desenvolvimento, já para outros, essas atividades podem ocorrem e forma paralela. O uso de um processo inadequado pode reduzir a qualidade ou a utilidade do produto de software a ser desenvolvido e/ou aumentando os custos de desenvolvimento.

2.2 - MODELO DE DESENVOLVIMENTO DE SOFTWARE – CICLO DE VIDA

Um modelo de ciclo de vida ou modelo de processo de software pode ser visto como uma representação abstrata de um esqueleto de processo, incluindo tipicamente algumas atividades principais, a sua ordem de precedência e, opcionalmente artefatos requeridos e produzidos. De maneira geral, um modelo de processo descreve uma filosofia de organização e sequenciamento de atividades, estruturando as atividades do processo em fases e definindo como essas fases estão relacionadas. De maneira geral, o ciclo de vida de um software envolve, pelo menos, as seguintes fases (FALBO, 2005):

Planejamento - O objetivo do planejamento de projeto é fornecer uma estrutura que possibilite ao gerente fazer Estimativas razoáveis de recursos, custos e prazos. Uma

vez estabelecido o escopo de software, com os requisitos esboçados, uma proposta de desenvolvimento deve ser elaborada, isto é, um plano de projeto deve ser elaborado configurando o processo a ser utilizado no desenvolvimento de software. A medida que o projeto progride, o planejamento deve ser detalhado e atualizado regularmente. Pelo menos ao final de cada uma das fases do desenvolvimento (análise e especificação de requisitos, projeto, implementação e testes), o planejamento como um todo deve ser revisto e o planejamento da etapa seguinte deve ser detalhado. O planejamento e o acompanhamento do progresso fazem parte do processo de gerência de projeto.

Análise e Especificação de Requisitos - Nesta fase, o processo de levantamento de requisitos é intensificado. O escopo deve ser refinado e os requisitos mais bem definidos. Para entender a natureza do software a ser construído, o engenheiro de software tem de compreender o domínio do problema; bem como a funcionalidade e o comportamento esperados. Uma vez capturados os requisitos do sistema a ser desenvolvido, estes devem ser modelados, avaliados e documentados. Uma parte vital desta fase é a construção de um modelo descrevendo o que o software tem de fazer (e como fazê-lo).

Projeto - Esta fase é responsável por incorporar requisitos tecnológicos aos requisitos essenciais do sistema, modelados na fase anterior e, portanto, requer que a plataforma de implementação seja conhecida. O projeto de arquitetura do sistema e projeto detalhado tem como objetivo definir a arquitetura geral do software, tendo por base o modelo construído na fase de análise de requisitos. Essa arquitetura deve descrever a estrutura de nível de software, recursos computacionais e suas interligações, alinhado aos requisitos especificados, dando uma base para a fase de implantação executar o projeto.

Implementação - O projeto deve ser traduzido para uma forma possível de executar pelo computador. A fase de implementação realiza esta tarefa, isto é, cada unidade de software do projeto detalhado é implementada. Testes inclui diversos níveis de testes, a saber, teste de unidade, teste de integração e teste de sistema ou funcionais. Inicialmente, cada unidade de software implementada deve ser testada e os resultados documentados. A seguir, os diversos componentes devem ser integrados sucessivamente até se obtenha o sistema. Finalmente, o sistema como um todo deve ser testado.

Entrega e Implantação - Uma vez testado, o software que for ser colocado em produção, contudo, é necessário treinar os usuários, configurar o ambiente de produção e, muitas vezes, converter bases de dados. O propósito desta fase é estabelecer que o software satisfaça os requisitos dos usuários. Isto é feito instalando o software e conduzindo testes de aceitação. Quando o software tiver demonstrado prover as capacidades requeridas, ele pode ser aceito e a operação iniciada. Operação nesta fase é ver o software ser utilizado pelos usuários no ambiente de produção.

Operação – Esta fase é quando os usuários utilizam o software já implantado no ambiente de produção.

Manutenção – Todos os softwares necessitam de evolução, consequentemente

haverá mudanças após à entrega do software ao usuário, seja por motivos de erros encontrados ou adaptações necessárias, tais como adição de novas funcionalidades. Devido a esses motivos existe a preocupação em tornar o software fácil de se manter, ou fácil de alterá-lo, evoluí-lo, tornando essa preocupação relevante, devido à grande quantidade de mudanças que deverá haver nesta fase.

Nas seções seguintes será apresentado um pouco sobre os modelos de software existentes :

2.3 - MODELO CASCATA

Este é o modelo mais simples de desenvolvimento de software, estabelecendo uma ordenação linear no que diz respeito à realização das diferentes etapas. Como mostra a (Figura 2.1), o ponto de partida do modelo é uma etapa de Engenharia de Sistemas, onde o objetivo é ter uma visão global do sistema como um todo (incluindo hardware, software, equipamentos e as pessoas envolvidas) como forma de definir precisamente o papel do software neste contexto. Em seguida, a etapa de Análise de Requisitos vai permitir uma clara definição dos requisitos de software, sendo que o resultado será utilizado como referência para as etapas posteriores de Projeto, Codificação, Teste e Manutenção.

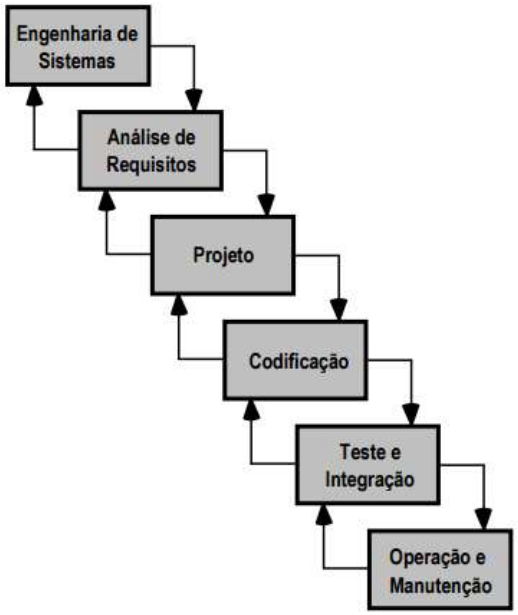

FIGURA 2.1 - MODELO CASCATA

O modelo cascata apresenta características interessantes, particularmente em razão da definição de um ordenamento linear das etapas de desenvolvimento. Primeiramente, como forma de identificar precisamente o fim de uma etapa de o início da seguinte, isto é feito normalmente através da aplicação de algum método de validação ou verificação, cujo objetivo será garantir de que a saída de uma dada etapa é coerente com a sua entrada (a qual já é a saída da etapa precedente). Isto significa que ao final de cada etapa realizada, deve existir um resultado (ou saída) a qual possa ser submetida à atividade de certificação.

Outra característica importante deste modelo é que as saídas de uma etapa são as entradas da seguinte, o que significa que uma vez definidas, elas não devem, em hipótese alguma ser modificadas. Duas diretivas importantes que norteiam o desenvolvimento segundo o modelo cascata, são:

• todas as etapas definidas no modelo devem ser realizadas, isto porque, em projetos de grande complexidade, a realização formal destas vai determinar o sucesso ou não do desenvolvimento; a realização informal e implícita de algumas destas etapas poderia ser feita apenas no caso de projetos de pequeno porte;

• a ordenação das etapas na forma como foi apresentada deve ser rigorosamente respeitada; apesar de que esta diretiva poderia ser questionada, a ordenação proposta pelo modelo, por ser a forma mais simples de desenvolver, tem sido também a mais adotada a nível de projetos de software, porém tem seus problemas.

É importante lembrar, também que os resultados de um processo de desenvolvimento de software não devem ser exclusivamente o programa executável e a documentação associada. Existe uma quantidade importante de resultados (ou produtos intermediários) os quais são importantes para o sucesso do desenvolvimento. Baseados nas etapas apresentadas na (Figura 2.1), é possível listar os resultados mínimos esperados de um processo de desenvolvimento baseado neste modelo: **Documento de Especificação de Requisitos, Projeto do Sistema, Plano de Teste e Relatório de Testes, Código Final, Manuais de Utilização, Relatórios de Revisões**.

Apesar de ser um modelo bastante popular, pode-se apontar algumas limitações apresentadas por este modelo:

• o modelo assume que os requisitos são inalterados ao longo do desenvolvimento; isto em boa parte dos casos não é verdadeira, uma vez que nem todos os requisitos são completamente definidos na etapa de análise;

• muitas vezes, a definição dos requisitos pode conduzir à definição do hardware sobre o qual o sistema vai funcionar; dado que muitos projetos podem levar diversos anos

para serem concluídos, estabelecer os requisitos em termos de hardware é um tanto temeroso, dadas as frequentes evoluções no hardware;

• o modelo impõe que todos os requisitos sejam completamente especificados antes do prosseguimento das etapas seguintes; em alguns projetos, é às vezes mais interessante poder especificar completamente somente parte do sistema, prosseguir com o desenvolvimento do sistema, e só então encaminhar os requisitos de outras partes; isto não é previsto a nível do modelo;

• as primeiras versões operacionais do software são obtidas nas etapas mais tardias do processo, o que na maioria das vezes inquieta o cliente, uma vez que ele quer ter acesso rápido ao seu produto. De todo modo, pode vir a ser mais interessante a utilização deste modelo para o desenvolvimento de um dado sistema do que realizar um desenvolvimento de maneira totalmente anárquica e informal.

2.4 - Desenvolvimento Iterativo e Incremental

Este modelo também foi concebido com base nas limitações do modelo Cascata. A ideia principal deste modelo, ilustrada na (Figura 2.2), é a de que um sistema deve ser desenvolvido de forma incremental, sendo que cada incremento vai adicionando ao sistema novas capacidades funcionais de forma incremental, até a obtenção do sistema final, sendo que, a cada passo realizado, modificações podem ser introduzidas.

Uma vantagem desta abordagem é a facilidade em testar o sistema, uma vez que a realização de testes em cada nível de desenvolvimento é, sem dúvida, mais fácil do que testar o sistema final. No primeiro passo deste modelo uma implementação inicial do sistema é obtida, na forma de um subconjunto da solução do problema global. Este primeiro nível de sistema deve contemplar os principais aspectos que sejam facilmente identificáveis no que diz respeito ao problema a ser resolvido.

Um aspecto importante deste modelo é a criação de uma lista de controle de projeto, a qual deve apresentar todos os passos a serem realizados para a obtenção do sistema final. Ela vai servir também para se medir, num dado nível, o quão distante se está da última iteração. A lista de controle de projeto gerencia todo o desenvolvimento, definindo quais tarefas devem ser realizadas a cada iteração, sendo que as tarefas na lista podem representar, inclusive, redefinições de componentes já implementados, em razão de erros ou problemas detectados numa eventual etapa de análise.

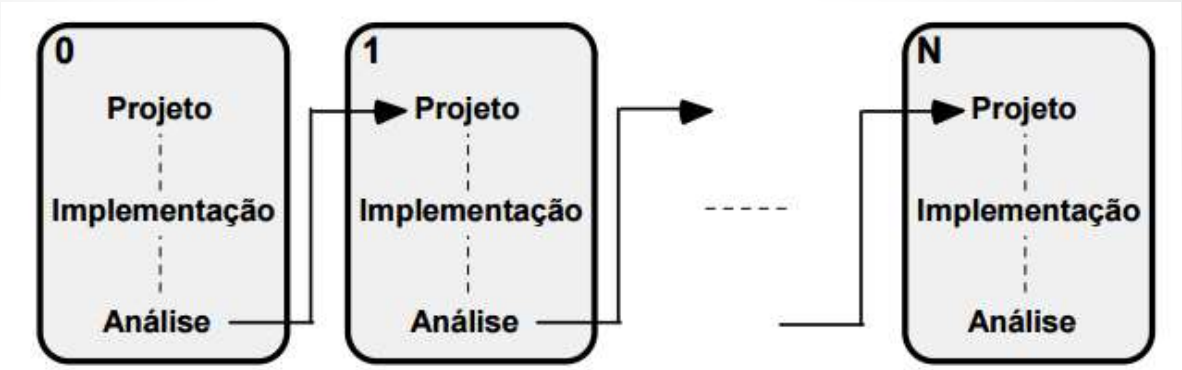

FIGURA 2.2 MODELO ITERATIVO E INCREMENTAL

2.5 - MODELO ESPIRAL

Este modelo é considerado evolucionário, proposto em 1988, sugere uma organização das atividades em espiral, a qual é composta de diversos ciclos. Como mostrado na (Figura 2.3), a dimensão vertical representa o custo acumulado na realização das diversas etapas; a dimensão angular representa o avanço do desenvolvimento ao longo das etapas.

Cada ciclo na espiral inicia com a identificação dos objetivos e as diferentes alternativas para atingi-los assim como as restrições impostas. O próximo passo no ciclo é a avaliação das diferentes alternativas com base nos objetivos fixados, o que vai permitir também definir incertezas e riscos de cada alternativa. No passo seguinte, o desenvolvimento de estratégias permitindo resolver ou eliminar as incertezas levantadas anteriormente, o que pode envolver atividades de prototipação, simulação, avaliação de desempenho, etc. Finalmente, o software é desenvolvido e o planejamento dos próximos passos é realizado.

A continuidade do processo de desenvolvimento é definida como função dos riscos remanescentes, como por exemplo, a decisão se os riscos relacionados ao desempenho ou à interface são mais importantes do que aqueles relacionados ao desenvolvimento do programa. Com base nas decisões tomadas, o próximo passo pode ser o desenvolvimento de um novo protótipo que elimine os riscos considerados.

Por outro lado, caso os riscos de desenvolvimento de programa sejam considerados os mais importantes e se o protótipo obtido no passo corrente já resolve boa parte dos riscos ligados a desempenho e interface, então o próximo passo pode ser simplesmente a evolução segundo o modelo Cascata.

Como se pode ver, o elemento que conduz este processo é essencialmente a consideração sobre os riscos, o que permite, de certo modo, a adequação a qualquer política de desenvolvimento (baseada em especificação, baseada em simulação, baseada em protótipo, etc..).

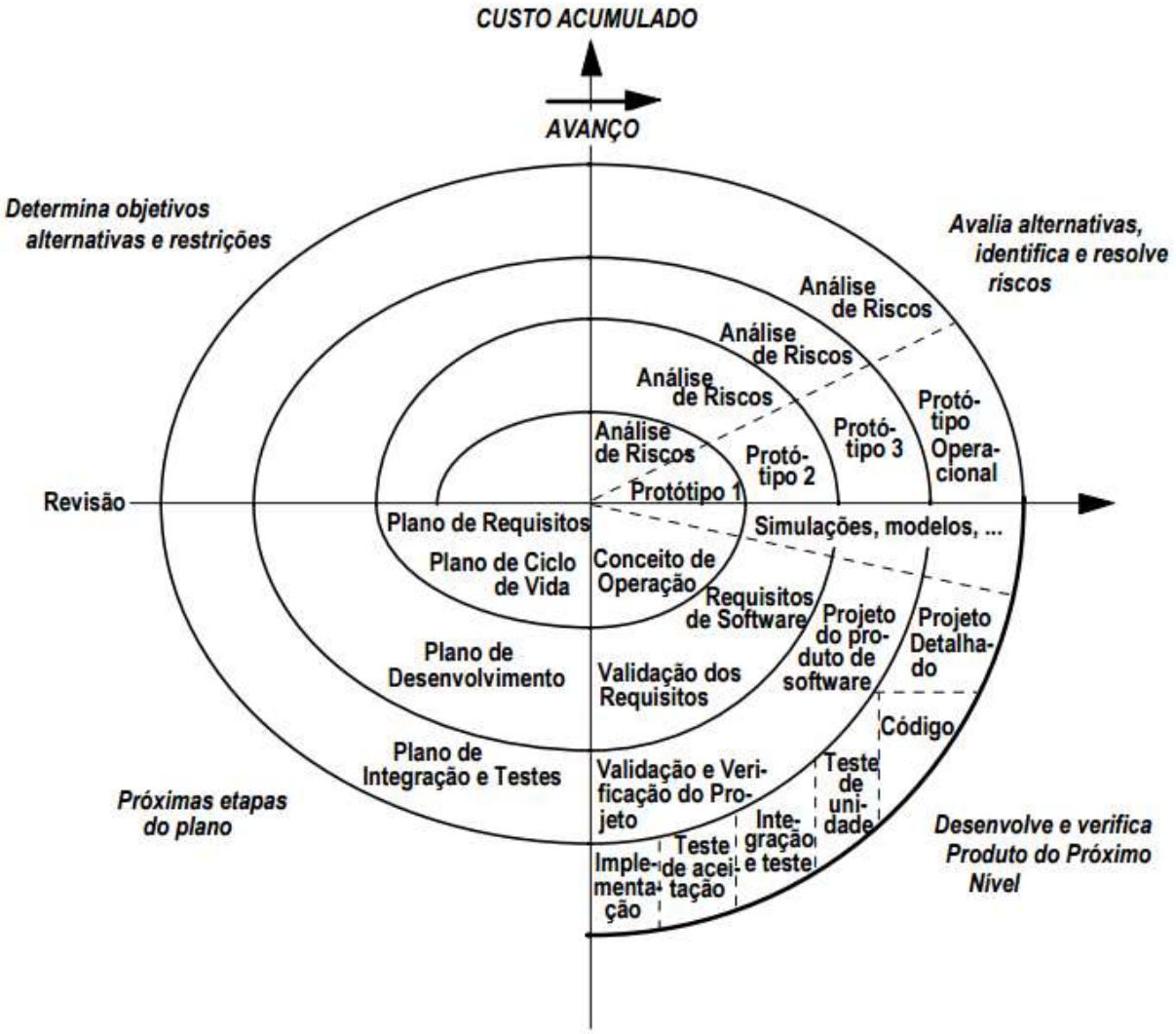

FIGURA 2.3 – MODELO ESPIRAL

2.6 - MODELOS ÁGEIS

O Extreme Programming é um modelo de desenvolvimento de software ágil, criado em 1996, por Kent Bech, no Departamento de Computação da montadora de carros Daimler Crysler, ele possui muitas diferenças em relação a outros modelos, podendo ser aplicado a projetos de alto risco e com requisitos dinâmicos. O XP é um conjunto bem definido de regras, que vem ganhando um grande número de adeptos e por oferecer

condições para que os desenvolvedores respondam com eficiência a mudanças no projeto, mesmo nos estágios finais do ciclo de vida do processo, devido a quatro lemas adotados por seus seguidores, que correspondem a quatro dimensões a partir das quais os projetos podem ser melhorados. São eles:

- Comunicação.
- Simplicidade.
- Feedback.
- Coragem.

No entanto, o XP não deve ser aplicado a qualquer tipo de projeto. O grupo de desenvolvedores deve formar uma equipe de 2 a 10 integrantes, que devem estar por dentro de todas as fases do desenvolvimento. É necessário realizar vários testes, às vezes, alterar o projeto em decorrência destes. A equipe tem de ser bastante interessada e proativa, para assegurar a alta produtividade, e o cliente deve estar sempre disponível para tirar dúvidas e tomar decisões em relação ao projeto.

Seguindo os requisitos expostos, anteriormente, o XP poderá trazer inúmeros benefícios ao mercado, de forma que o processo de desenvolvimento se torne mais ágil e flexível. Um desses benefícios é a agilidade no Planejamento (Planning Games) de não definir uma especificação completa e formal dos requisitos, ao contrário das metodologias tradicionais. Outro é a produção de sistemas simples que atendam aos atuais requisitos, não tentando antecipar o futuro e permitindo atualizações frequentes em ciclos bastante curtos.

A comunicação com o cliente, no XP, mostra-se mais intensa que nos métodos tradicionais, devendo o cliente estar sempre disponível para tirar as dúvidas, rever requisitos e atribuir prioridades, utilizando-se de sistemas de nomes, em vez de termos técnicos para facilitar a comunicação. As equipes devem manter-se integradas com os projetos, melhorando a comunicação e a produtividade. Uma outra característica importante do XP é que o código é sempre escrito em duplas, visando a melhorar a qualidade do código por um custo muito baixo, às vezes menores do que o desenvolvimento individual. O código deve estar padronizado, para que todos na equipe possam entender o que está sendo escrito e possa ser validado durante todo o desenvolvimento, tanto pelos desenvolvedores quanto pelos clientes, a fim de se saber se os requisitos estão ou não sendo atendidos.

A seguir serão descritas as práticas envolvidas no XP (WIKEPEDIA, 2015) para utilizar dos valores e princípios durante o desenvolvimento de software, o XP propõe uma série delas. Há uma confiança muito grande na sinergia entre elas, os pontos fracos de cada uma são superados pelos pontos fortes de outras.

Jogo de Planeamento (Planning Game): O desenvolvimento é feito em iterações semanais. No início da semana, desenvolvedores e cliente reúnem-se para priorizar as funcionalidades. Essa reunião recebe o nome de Jogo do Planeamento. Nela, o cliente identifica prioridades e os desenvolvedores as estimam. O cliente é essencial neste processo e, assim, ele fica sabendo o que está acontecendo e o que vai acontecer no projeto. Como o escopo é reavaliado semanalmente, o projeto é regido por um contrato de escopo negociável, que difere significativamente das formas tradicionais de contratação de projetos de software. Ao final de cada semana, o cliente recebe novas funcionalidades, completamente testadas e prontas para serem colocadas em produção.

Pequenas Versões (Small Releases): A liberação de pequenas versões funcionais do projeto auxilia muito no processo de aceitação por parte do cliente, que já pode testar uma parte do sistema que está comprando. As versões chegam a ser ainda menores que as produzidas por outras metodologias incrementais, como o RUP.

Metáfora (Metaphor): Procurar facilitar a comunicação com o cliente, entendendo a realidade dele. O conceito de rápido para um cliente de um sistema jurídico é diferente para um programador experiente em controlar comunicação em sistemas de tempo real, como controle de tráfego aéreo. É preciso traduzir as palavras do cliente para o significado que ele espera dentro do projeto.

Projeto Simples (Simple Design): Simplicidade é um princípio do XP. Projeto simples significa dizer que caso o cliente tenha pedido que na primeira versão apenas o usuário "teste" possa entrar no sistema com a senha "123" e assim ter acesso a todo o sistema, você vai fazer o código exato para que esta funcionalidade seja implementada, sem se preocupar com sistemas de autenticação e restrições de acesso. Um erro comum ao adotar essa prática é a confusão por parte dos programadores de código simples e código fácil. Nem sempre o código mais fácil de ser desenvolvido levará a solução mais simples por parte de projeto. Esse entendimento é fundamental para o bom andamento do XP. Código fácil deve ser identificado e substituído por código simples.

Time Coeso (Whole Team): A equipe de desenvolvimento é formada pelo cliente e pela equipe de desenvolvimento.

Testes de Aceitação (Customer Tests): São testes construídos pelo cliente em conjunto com analistas e testadores, para aceitar um determinado requisito do sistema.

Ritmo Sustentável (Sustainable Pace): Trabalhar com qualidade, buscando ter ritmo de trabalho saudável (40 horas/semana, 8 horas/dia), sem horas extras. Horas extras são permitidas quando trouxerem produtividade para a execução do projeto.

Reuniões em pé (Stand-up Meeting): Reuniões em pé para não se perder o foco nos assuntos de modo a efetuar reuniões rápidas, apenas abordando tarefas realizadas e tarefas a realizar pela equipe.

Posse Coletiva (Collective Ownership): O código fonte não tem dono e ninguém precisa ter permissão concedida para poder modificar o mesmo. O objetivo com isto é fazer a equipe conhecer todas as partes do sistema.

Programação em Pares (Pair Programming): é a programação em par/dupla num único computador. Geralmente a dupla é criada com alguém sendo iniciado na linguagem e a outra pessoa funcionando como um instrutor. Como é apenas um computador, o novato é que fica à frente fazendo a codificação, e o instrutor acompanha ajudando a desenvolver suas habilidades. Dessa forma o programa sempre é revisto por duas pessoas, evitando e diminuindo assim a possibilidade de erros (bugs). Com isto, procura-se sempre a evolução da equipe, melhorando a qualidade do código fonte gerado.

Padrões de Codificação (Coding Standards): A equipe de desenvolvimento precisa estabelecer regras para programar e todos devem seguir estas regras. Dessa forma parecerá que todo o código fonte foi editado pela mesma pessoa, mesmo quando a equipe possui 10 ou 100 membros.

Desenvolvimento Orientado a Testes (Test Driven Development): Primeiro crie os testes unitários (unit tests) e depois crie o código para que os testes funcionem. Esta abordagem é complexa no início, pois vai contra o processo de desenvolvimento de muitos anos. Só que os testes unitários são essenciais para que a qualidade do projeto seja mantida.

Refatoração (Refactoring): É um processo que permite a melhoria contínua da programação, com o mínimo de introdução de erros e mantendo a compatibilidade com o código já existente. Refatorar melhora a clareza (leitura) do código, divide em módulos mais coesos e de maior reaproveitamento, evitando a duplicação de código-fonte.

Integração Contínua (Continuous Integration): Sempre que realizar uma nova funcionalidade, nunca esperar uma semana para integrar na versão atual do sistema. Isto só aumenta a possibilidade de conflitos e a possibilidade de erros no código fonte. Integrar de forma contínua permite saber o status real da programação.

Depois de apresentar todas as práticas ágeis disponíveis no XP, podemos ilustrar na (Figura 2.4) o seu ciclo de vida.

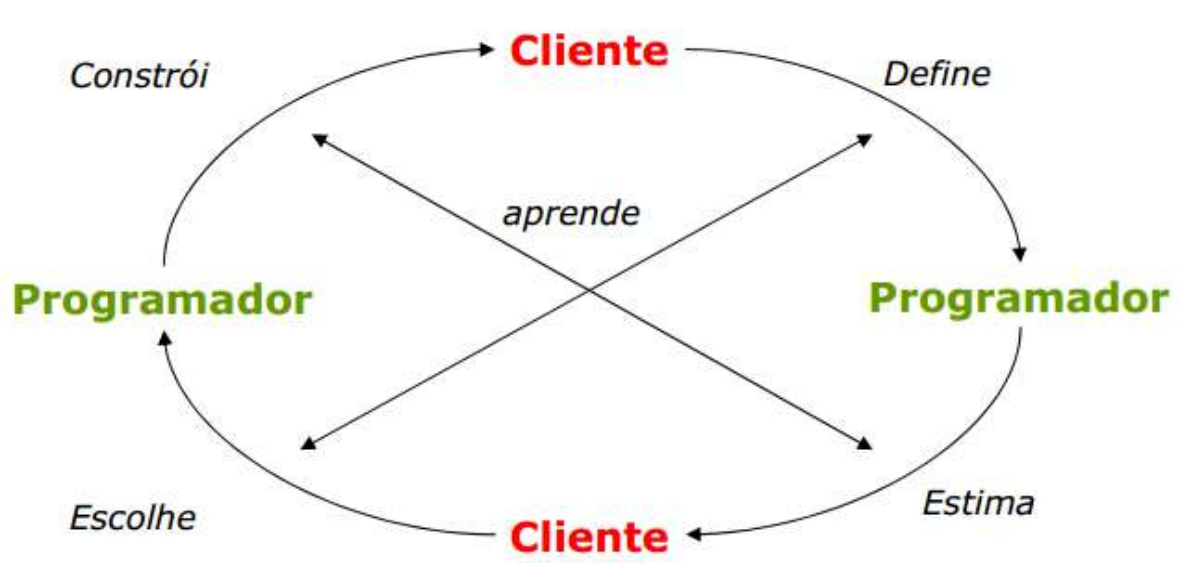

FIGURA 2.4 - CICLO DE VIDA EXTREME PROGRAMMING

2.7 - ENGENHARIA DE SOFTWARE AUXILIADA POR COMPUTADOR (CASE)

A sigla **CASE** significa "Computer Aided Software Engeneering", em português: "Engenharia de Software Auxiliada por Computador".

Ferramentas CASE são ferramentas utilizadas como suporte para desenvolver um software. Essas ferramentas oferecem um conjunto de serviços, fortemente relacionados, para apoiar uma ou mais atividades do processo de desenvolvimento de software epodem minimizar o tempo de desenvolvimento do programa, mantendo o alto nível de qualidade.

As vantagens em se utilizar uma FerramentaCASE é o aumento da produtividade, melhor qualidade, diminuição dos custos, melhor gerenciamento e a grande facilidade de manutenção.

Cada ferramenta tem propósitos diferentes, fornece serviços diferentes, mas possuem algumas características em comum.

As ferramentas CASE podem ser classificadas em:

· Horizontais: oferecem serviços utilizados durante todo o processo de software;

· Verticais: utilizadas em fases específicas do processo de software.

Elas também podem ser classificadas de acordo com os serviços que oferecem, dentre as quais, cita-se.

· Documentação;

· Planejamento e gerenciamento de projetos;

· Comunicação;

· Análise e projeto de software;

· Projeto e desenvolvimento de interfaces;

· Programação;

· Gerenciamento de Configuração;

· Controle de Qualidade.

Ou ainda, categoricamente, elas podem ser:

- ***Upper CASE*** - ferramentas de planejamento, análise e projeto(Front-End);
- ***Lower CASE*** - ferramentas de codificação (Back-end);
- ***Integrated CASE*** - união de Upper e Lower CASE, cobre todo o ciclo de vida do software, desde os requisitos do sistema até o controle final da qualidade;

Um dos componentes indispensáveis de uma ferramenta CASE é a modelagem visual, ou seja, a possibilidade de representar, através de modelos gráficos.

As ferramentas CASE automatizam uma grande variedade de tarefas: Geração de documentação, Testes, Geração de código, Geração de Relatórios para acompanhamento do trabalho entre outras atividades.

Com a necessidade de produtividade e qualidade, os ambientes de desenvolvimento de softwares foram obrigados a desenvolver também software para uso próprio. Inicialmente, era utilizado o termo "*Workbench*", que significa "bancada de trabalho". Ele designava as ferramentas, geralmente automatizadas, que auxiliavam no trabalho dos desenvolvedores de software. Mais tarde surgiu o termo CASE.

Uma ferramenta CASE é qualquer software que auxilia as pessoas que trabalham em um ambiente de desenvolvimento de software. A presença de ferramentas CASE é vital hoje em dia para o bom funcionamento de um projeto de software. Elas existem auxiliando todo o ciclo de desenvolvimento (Análise, Projeto, Implementação e Teste) e são também de suma importância para a manutenção do software, como será visto a seguir. Há também ferramentas CASE para apoiar a gerência dos projetos de desenvolvimento.

As ferramentas CASE trazem diversos benefícios para as pessoas que trabalham com informática, tanto para os desenvolvedores, analistas de sistemas e até mesmo para os usuários finais. Alguns destes benefícios são:

- Fazer a aplicação geral da engenharia de software;
- Tornar prática a prototipação, acelerando o desenvolvimento de sistemas junto aos usuários;
- Agilizar o processo de desenvolvimento;
- Deixar disponível o reuso de componentes do sistema;
- Documentação do sistema gerado pela própria ferramenta;
- Necessidade de uma mudança de atitude profissional de desenvolvimento de sistemas e aprendizado de novas metodologias que as ferramentas utilizam;
- Permite a construção de novos sistemas que atendam há todos os processos;
- Redução de custo da manutenção de sistemas.

2.8 - Responsabilidade Profissional e Ética

Assim como ocorre com os outros engenheiros, os engenheiros de software devem aceitar que seu trabalho envolve responsabilidades mais amplas do que a simples aplicação de habilidades técnicas. Seu trabalho é realizado dentro de uma estrutura legal e social. A engenharia de software é com certeza, delimitada por leis locais, nacionais e internacionais. Os engenheiros de software devem se comportar de maneira responsável, ética e moralmente, para serem respeitados como profissionais.

Nem e preciso dizer que os engenheiros devem manter padrões normais de honestidade e integridade. Eles não devem utilizar suas capacidades e habilidades para se comportarem de maneira desonesta ou que possa trazer descredito a profissão de engenharia de software. No entanto, existem áreas em que os padrões de comportamento aceitável não estão delimitados por leis, mas pela noção mais tênue de responsabilidade profissional. Eis algumas dessas áreas (SOMMERVILLE, 2007):

1. **Confidencialidade** – Os engenheiros devem, normalmente, respeitar a confidencialidade de seus empregadores e empregados ou clientes, quer tenham assinado ou não um acordo formal de confidencialidade.

2. **Competência** - Os engenheiros não devem enganar quanto ao seu nível de competência. Não devem conscientemente aceitar serviços que estejam fora do seu limite de competência.

3. **Direitos de propriedade intelectual** - Os engenheiros devem estar cientes das leis locais que regulam o uso da propriedade intelectual, como patentes e direitos autorais. Eles devem ser cuidadosos, a fim de assegurar que a propriedade intelectual de empregadores e clientes seja protegida.

4. **Má utilização de computadores** - Os engenheiros de software não devem empregar suas habilidades técnicas para o mau uso dos computadores de outras pessoas. 0 mau uso de computadores abrange desde casos relativamente triviais (por exemplo, jogar no computador do empregador) até cases extremamente series (como a disseminação de vírus).

2.9 - EXERCÍCIOS

1) Qual o modelo de desenvolvimento de software mais simples no seu ponto de vista, explique?.

2) Quais as fases que compõem o modelo de desenvolvimento cascata?.

3) Qual é a vantagem do modelo de desenvolvimento Iterativo e Incremental, explique?.

4) Qual o modelo que é considerado evolucionário, proposto em 1988, o qual é composto de diversos ciclos?.

5) Quais as características ágeis que fazem parte do método de desenvolvimento XP (Extreme Programming)?

6) As ferramentas CASE são utilizadas para qual propósito?.

REFERÊNCIAS DO CAPÍTULO

BECK, K. – Extreme Programming Explained: enbrance change. Boston : Addison Wesley /Longman, 1999

FALBO, R.A., 2005. Engenharia de Software, Notas de Aula, UFES – Universidade Federal do Espírito Santo.

PRESSMAN, R.S., Engenharia de Software, McGraw-Hill, 6ª edição, 2006.

SOMMERVILLE, I., Engenharia de Software, 8ª Edição. São Paulo: Pearson – Addison Wesley, 2007.

WIKIPEDIA, 2015 – Programação Extrema, Acessado em 05/08/2015 <https://pt.wikipedia.org/wiki/Programação_extrema>

CAPÍTULO 3

GERENCIAMENTO DO PROJETO DE SOFTWARE

Neste capítulo serão apresentados os mecanismos que auxiliam no gerenciamento do projeto de software, além de disseminar as principais atividades e documentos necessários nesta fase.

3.1 - INTRODUÇÃO

Na maior parte dos trabalhos de Software, o tempo é um fator preponderante. Isto é uma consequência de questão cultural (todos nós aprendemos a trabalhar sob o efeito da pressão de tempo e o mercado impulsiona cada vez mais nesta direção), resultando em uma "pressa" nem sempre justificada, conduzindo, na maior parte das vezes, a prazos totalmente irreais, e que são definidos por pessoas quem não tem um alto grau de envolvimento no projeto.

Na prática, a definição de cronogramas dos projetos é feita de maneira arbitrária; os riscos do projeto só são considerados quando eles se transformam em problemas reais; a organização da equipe nem sempre é clara e feita de forma consciente.

Neste capítulo serão discutidos alguns pontos fundamentais para que o projeto de desenvolvimento de um software seja conduzido de forma a obter resultados satisfatórios em termos de produtividade (do processo) e qualidade (do produto) segundo .

3.2 - ANÁLISE DOS RISCOS

A análise dos riscos se torna uma das mais importantes e essenciais atividades para uma boa realização de um projeto de software. Esta atividade se baseia na realização de quatro tarefas, conduzidas de forma sequencial: a identificação, a projeção, a avaliação e a administração.

3.2.1 - IDENTIFICAÇÃO DOS RISCOS

Para (MAZZOLA e SOMMERVILLE), o objetivo da identificação dos riscos é que sejam levantados por parte do gerente e dos profissionais envolvidos no projeto, todos os eventuais riscos aos quais serão submetidos. Nesta identificação, riscos de diferentes naturezas podem ser detectados:

- **Riscos de projeto**, os quais estão associados a problemas relacionados ao próprio processo de desenvolvimento (orçamento, cronograma, pessoal, etc.);
- **Riscos técnicos**, que consistem dos problemas de projeto efetivamente (implementação, manutenção, interfaces, plataformas de implementação, etc.);
- **Riscos de produto**, os quais estão mais relacionados aos problemas que vão surgir para a inserção do software como produto no mercado (oferecimento de um produto que ninguém está interessado; oferecer um produto ultrapassado; produto inadequado à venda; etc..).

A categorização dos riscos, apesar de interessante, não garante a obtenção de resultados satisfatórios, uma vez que nem todos os riscos podem ser identificados facilmente.

Uma boa técnica para conduzir a identificação dos riscos de forma sistemática é o estabelecimento de um conjunto de questões (checklist) relacionado a algum fator risco. Por exemplo, com relação aos riscos de composição da equipe de desenvolvimento, as seguintes questões poderiam ser formuladas:

- São os melhores profissionais disponíveis?
- Os profissionais apresentam a combinação certa de capacidades?
- Há pessoas suficientes na equipe?
- Os profissionais estão comprometidos durante toda a duração do projeto?
- Algum membro da equipe estará em tempo parcial sobre o projeto?
- Os membros da equipe estão adequadamente treinados?

Em função das respostas a estas perguntas, será possível ao planejador estabelecer qual será o impacto dos riscos sobre o projeto.

3.2.2 - PROJEÇÃO DOS RISCOS

A projeção ou estimativa de riscos permite definir basicamente duas questões:

• Qual a probabilidade de que o risco ocorra durante o projeto?

•. Quais as consequências dos problemas associados ao risco no caso de ocorrência do mesmo?

As respostas a estas questões podem ser obtidas basicamente a partir de quatro atividades:

• estabelecimento de uma escala que reflita a probabilidade estimada de ocorrência de um risco;

• estabelecimento das consequências do risco;

• estimativa do impacto do risco sobre o projeto e sobre o software (produtividade e qualidade);

• anotação da precisão global da projeção de riscos.

A escala pode ser definida segundo várias representações (booleana, qualitativa ou quantitativa). No limite cada pergunta de uma dada checklist pode ser respondida com "sim" ou "não", mas nem sempre esta representação permite representar as incertezas de modo realístico.

Uma outra forma de se representar seria utilizar uma escala de probabilidades qualitativas, onde os valores seriam: altamente improvável, improvável, moderado,

provável, altamente provável. A partir destes "valores" qualitativos, seria possível realizar cálculos que melhor expressassem as probabilidades matemáticas de que certo risco viesse a ocorrer.

O passo seguinte é realizado o levantamento do impacto que os problemas associados ao risco terão sobre o projeto e sobre o produto final, o que permitirá priorizar os riscos. Pode-se destacar três fatores que influenciam no impacto de um determinado risco: a sua natureza, o seu escopo e o período da sua ocorrência.

A natureza do risco permite indicar os problemas prováveis se ele ocorrer (por exemplo, um risco técnico como uma má definição de interface entre o software e o hardware do cliente levará certamente a problemas de teste e integração). O seu escopo permite indicar de um lado qual a gravidade dos problemas que serão originados e qual a parcela do projeto que será atingida. O período de ocorrência de um risco permite uma reflexão sobre quando ele poderá ocorrer e por quanto tempo.

Estes três fatores definirão a importância do risco para efeito de priorização...um fator de risco de elevado impacto pode ser pouco considerado a nível do gerenciamento de projeto se a sua probabilidade de ocorrência for baixa. Fatores de risco com alto peso de impacto e com probabilidade de ocorrência de moderada a alta e fatores de risco com baixo peso de impacto com elevada probabilidade de ocorrência não devem ser desconsiderados, devendo ser processados por todas atividades de análise de risco.

3.2.3 - Avaliação e Respostas aos Riscos

O objetivo desta atividade é processar as informações sobre o fator de risco, o impacto do risco e a probabilidade de ocorrência. Nesta avaliação, serão checadas as informações obtidas na projeção de riscos, buscando priorizá-los e definir formas de controle e respostas destes ou de evitar a ocorrência daqueles com alta probabilidade de ocorrência.

Para tornar a avaliação eficiente, deve ser definido um nível de risco referente. Exemplos de níveis referentes típicos em projetos de Engenharia de Software são: o custo, o prazo e o desempenho. Isto significa que se vai ter um nível para o excesso de custo, para a ultrapassagem de prazo e para a degradação do desempenho ou qualquer combinação dos três. Desta forma, caso os problemas originados por uma combinação de determinados riscos provoquem a ultrapassagem de um ou mais desses níveis, o projeto poderá ser suspenso.

Após identificados os riscos, e de ter adotado as escalas de impacto e priorização, mesmo assim eles podem ocorrer, e se isso acontecer será necessário saber o que fazer,

para isso é preciso ter as respostas aos riscos para trata-los, pois, o gerente ou integrante da equipe do projeto poderá executar o tratamento ao risco de forma imediata.

3.2.4 - MONITORAMENTO DOS RISCOS

Uma vez avaliados os riscos de desenvolvimento, é importante que medidas sejam tomadas para evitar a ocorrência dos riscos ou que ações sejam definidas para a eventualidade da ocorrência dos riscos. Este é o objetivo da monitoração dos riscos. Para isto, as informações mais importantes são aquelas obtidas na tarefa anterior, relativa à descrição, probabilidade de ocorrência e impacto sobre o processo, associadas a cada fator de risco.

Por exemplo, considerando a alta rotatividade de pessoal numa equipe um fator de risco, com base em dados de projetos passados, obtém-se que a probabilidade de ocorrência deste risco é de 0,70 (muito elevada) e que a sua ocorrência pode aumentar o prazo do projeto em 15% e o seu custo global em 12%. Sendo assim, pode-se propor as seguintes ações de administração deste fator de risco:

- Reuniões com os membros da equipe para determinar as causas da rotatividade de pessoal (más condições de trabalho, baixos salários, mercado de trabalho competitivo, etc..);
- Tomada de providências para eliminar ou reduzir as causas "controláveis" antes do início do projeto;
- No início do projeto, pressupor que a rotatividade vai ocorrer e prever a possibilidade de substituição de pessoas quando estas deixarem a equipe;
- Organizar equipes de projeto de forma que as informações sobre cada atividade sejam amplamente difundidas;
- Definir padrões de documentação para garantir a produção de documentos de forma adequada;
- Realizar revisões do trabalho entre colegas de modo que mais de uma pessoa esteja informado sobre as atividades desenvolvidas;
- Definir um membro da equipe que possa servir de backup para o profissional mais crítico.

É importante observar que a implementação destas ações pode afetar também os prazos e o custo global do projeto. Isto significa que é necessário poder avaliar quando os custos destas ações podem ser ultrapassados pela ocorrência dos fatores de risco.

Num projeto de grande dimensão, de 30 a 40 fatores de risco podem ser identificados; se para cada fator de risco, sete ações forem definidas, a administração dos riscos pode

tornar-se um projeto ela mesma. Por esta razão, é necessário que uma priorização dos riscos seja efetuada, atingindo normalmente, a 20% de todos os fatores levantados.

Todo o trabalho efetuado nesta tarefa é registrado num documento denominado Plano de Administração e Monitoração de Riscos, o qual será utilizado posteriormente pelo gerente de projetos (particularmente, para a definição do Plano de Projeto, que é gerado ao final da etapa de planejamento). Uma ilustração sintetizando os passos essenciais desta tarefa é apresentada à (Figura 3.1).

Figura 3.1 – Plano de Administração e Monitoração dos Riscos

3.3 - DEFINIÇÃO DE CRONOGRAMA

Para (FALBO, 2005), a definição de um cronograma pode ser obtida segundo duas diferentes
Abordagens

- A primeira, é baseada na definição prévia de um prazo de entrega do software, neste caso, o planejamento deve ser feito de modo a distribuir os esforços ao longo do prazo estabelecido.

- A segunda, está relacionada a uma discussão de limites cronológicos aproximados para cada etapa do desenvolvimento, sendo que o prazo de entrega do software seja estabelecido a partir de técnicas de planejamento da Engenharia de Software.

Evidentemente, a primeira abordagem é a mais encontrada nos projetos de software. Um cronograma bem definido pode trazer enormes benefícios a um projeto de software, sendo às vezes mais importante que a própria definição de custos. Numa visão de desenvolvimento de software como produto, um adicional nos custos de produção pode ser absorvido por uma redefinição nos preços ou pela amortização em função de um elevado número de vendas. Já, um acréscimo imprevisto no prazo de entrega de um software pode provocar grandes prejuízos ao produto, como por exemplo: a queda no impacto de mercado, insatisfação dos clientes, elevação de custos internos entre outros.

Então algumas perguntas são feitas com o propósito de tentar desenvolver um cronograma realista, tais como:

- Como relacionar o tempo cronológico com o esforço humano?
- Quais tarefas e grau de paralelismo podem ser obtidos?
- Como medir o progresso do processo de desenvolvimento (indicadores de progresso)?
- Como o esforço pode ser distribuído ao longo do processo de Engenharia de Software?
- Que métodos estão disponíveis para a determinação de prazos?
- Como representar fisicamente o cronograma e como acompanhar o progresso a partir do início do projeto?

Em primeiro lugar, é preciso dizer que, à medida que um projeto ganha dimensão, um maior número de pessoas deve estar envolvido. Além disso, deve-se tomar o cuidado de não levar a sério o mito, apresentado no capítulo I, de que "Se o desenvolvimento do software estiver atrasado, basta aumentar a equipe para honrar o prazo de desenvolvimento."

Deve-se considerar, ainda que, quanto maior o número de pessoas, maior o número de canais de comunicação, o que normalmente requer esforço adicional e, consequentemente, tempo adicional.

Uma das vantagens de se ter uma equipe com mais de uma pessoa são as possibilidades de se realizar determinadas tarefas em paralelo. O paralelismo de tarefas é um mecanismo interessante como forma de economia de tempo na realização de qualquer trabalho de engenharia. Para isto, basta que um gerenciamento dos recursos necessários a cada tarefa (recursos humanos, recursos de software, etc..) seja feito de forma eficiente.

Sendo assim, as tarefas a serem realizadas durante um projeto de desenvolvimento de software podem ser expressas na forma de uma rede (rede de tarefas) a qual apresenta todas as tarefas do projeto, assim como as suas relações em termos de realização (paralelismo, sequência, pontos de encontro, etc.. Um exemplo desta representação é apresentado na (Figura 3.2)

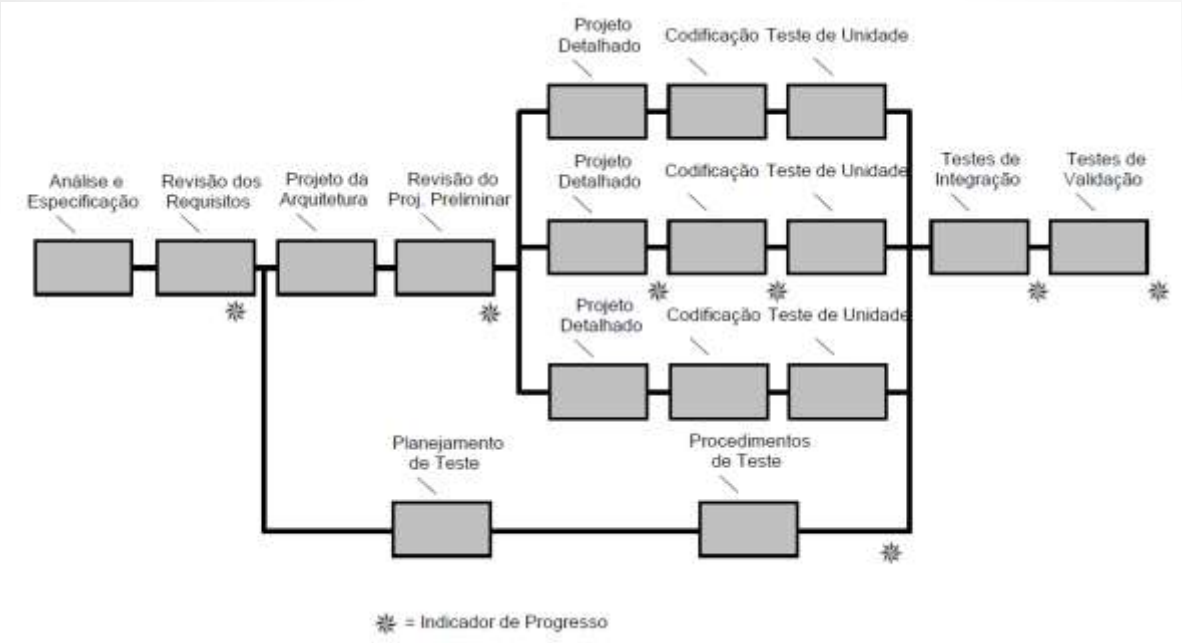

FIGURA 3.2 - REDE DE TAREFAS

As técnicas de estimativas para projetos de software normalmente conduzem a uma definição do esforço em forma de número de homens-hora, homens-mês ou homens-ano, que são necessárias para realização do desenvolvimento de um projeto. Um exemplo de distribuição dos esforços que é bastante utilizada nos projetos é ilustrada na (Figura 3.3), a qual é baseada numa regra denominada Regra 40-20-40, a qual sugere, como etapas onde o esforço deve ser maior, as dos extremos do processo de desenvolvimento (análise/projeto e testes), a codificação sendo a tarefa que deve envolver a menor concentração de esforço. Isto pode ir contra o grau de importância em termos de esforço que muitos desenvolvedores dão a cada uma destas atividades.

É evidente que estes valores não podem ser levados à risca para todos os projetos de software, sendo que as características de cada projeto vão influenciar na parcela de esforço a ser dedicada a cada etapa.

No entanto, é importante entender a razão pela qual o esforço dedicado à etapa de codificação aparece com menor intensidade que os demais.

Na realidade, se a etapa de projeto foi realizada utilizando as boas regras da Engenharia de Software, a etapa de codificação será, consequentemente, minimizada em termos de esforço.

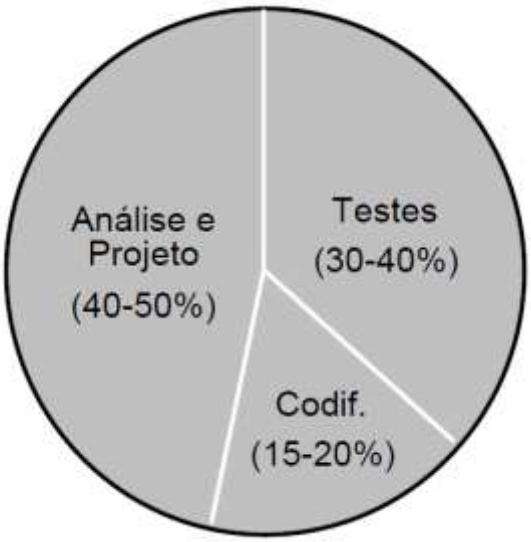

FIGURA 3.3 - DISTRIBUIÇÃO DE ESFORÇOS NO PROJETO DE SOFTWARE

A definição do cronograma é uma das tarefas mais difíceis de se definir na etapa de planejamento do software. O gerente deve levar em conta diversos aspectos relacionados ao projeto taus como: a disponibilidade de recursos no momento da execução de uma dada tarefa, as interdependências das diferentes tarefas, a ocorrência de possíveis estrangulamentos do processo de desenvolvimento e as operações necessárias para agilizar o processo, identificação das principais atividades, revisões e indicadores de progresso do processo de desenvolvimento, etc.

A forma de representação dos cronogramas depende da política de desenvolvimento adotada, mas pode ser apresentada, numa forma geral por um documento do tipo apresentado na (Figura 3.4).

As unidades de tempo consideradas no projeto (dias, semanas, meses, etc.) são anotadas na linha superior da folha de cronograma. As tarefas do projeto, as atividades e os indicadores de progresso são definidos na coluna da esquerda. O traço horizontal permite indicar o período durante o qual determinada atividade será realizada, o tempo necessário sendo medido em unidades de tempo consideradas. Quando atividades puderem ser realizadas em paralelo, os traços vão ocupar unidades de tempo comuns. O cronograma deve explicitar as atividades relevantes do desenvolvimento e os indicadores de progresso associados. É importante que os indicadores de progresso sejam representados por resultados concretos (por exemplo, um documento). A disponibilidade de recursos deve também ser representada ao longo do cronograma. O impacto da

indisponibilidade dos recursos no momento em que eles são necessários deve também ser representado, se possível, no cronograma.

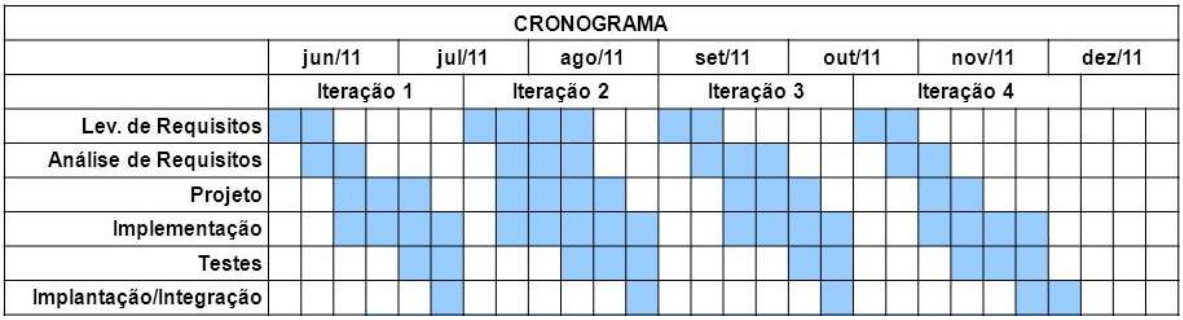

FIGURA 3.4 - CRONOGRAMA DE UM PROJETO DE SOFTWARE

3.4 MÉTRICA DE SOFTWARE

Em primeiro lugar, é importante estar ciente que as medidas são uma forma clara de avaliação da produtividade no desenvolvimento de software. Sem informações quantitativas a respeito do processo de desenvolvimento de softwares por parte de uma empresa ou equipe de desenvolvedores de produto, é impossível tirar qualquer conclusão sobre de que forma está evoluindo a produtividade (se é que está evoluindo?).

Através da obtenção de medidas relativas à produtividade e à qualidade, é possível que metas de melhorias no processo de desenvolvimento sejam estabelecidas como forma de incrementar estes dois importantes fatores da Engenharia de Software.

Particularmente no que diz respeito à qualidade, é possível, com uma avaliação quantitativa deste parâmetro, promover-se pequenos ajustes no processo de desenvolvimento como forma de eliminar ou reduzir as causas de problemas que afetam de forma significativa o projeto de software.

No que diz respeito aos desenvolvedores, a obtenção de medidas pode auxiliar a responder com precisão uma série de perguntas que estes elementos se fazem a cada projeto:

- Que tipos de requisitos são os mais passíveis de mudanças?
- Quais módulos do sistema são os mais propensos a erros?
- Quanto de teste deve ser planejado para cada módulo?
- Quantos erros (de tipos específicos) pode-se esperar quando o teste se iniciar?

3.4.1 - MEDIDAS DE SOFTWARE

Citado por (FALBO, 20015) e de forma análoga a outras grandezas do mundo físico, as medições de software podem ser classificadas em duas categorias principais:

- **Medições diretas**, por exemplo, o número de linhas de código (LOC) produzidas, o tamanho de memória ocupado, a velocidade de execução, o número de erros registrados num dado período de tempo, etc.
- **Medições indiretas**, as quais permitem quantizar aspectos como a funcionalidade, complexidade, eficiência, manutenibilidade, etc.

As medições diretas, tais quais aquelas exemplificadas acima, são de obtenção relativamente simples, desde que estabelecidas as convenções específicas para isto. Por outro lado, aspectos como funcionalidade, complexidade, eficiência, etc., são bastante difíceis a quantizar.

As medições de software podem ser organizadas em outras classes, as quais serão definidas a seguir:

- **Métricas da produtividade**, baseadas na saída do processo de desenvolvimento do software com o objetivo de avaliar o próprio processo;
- **Métricas da qualidade**, que permitem indicar o nível de resposta do software às exigências explícitas e implícitas do cliente;
- **Métricas técnicas**, nas quais encaixam-se aspectos como funcionalidade, modularidade, manutenibilidade, etc.
- **Métricas orientadas ao tamanho**, baseadas nas medições diretas da Engenharia de Software;

3.5 PLANO DE PROJETO

Ao final desta etapa, um documento de **Plano de Projeto** deverá ser gerado, o qual deverá ser revisto para servir de referência às etapas posteriores. Ele vai apresentar as informações iniciais de custo e cronograma que vão nortear o desenvolvimento do software.

Ele consiste de um documento, o qual é encaminhado às diversas pessoas envolvidas no desenvolvimento do software. Dentre as informações a serem contidas neste documento, é possível destacar:

• o contexto e os recursos necessários ao gerenciamento do projeto, à equipe técnica e ao cliente;
• a definição de custos e cronograma que serão acompanhados para efeito de gerenciamento;
• dá uma visão global do processo de desenvolvimento do software a todos os

envolvidos.

A (Figura 3.5) apresenta uma possível estrutura para este documento. A apresentação dos custos e cronograma pode diferir dependendo de quem será o leitor do documento.

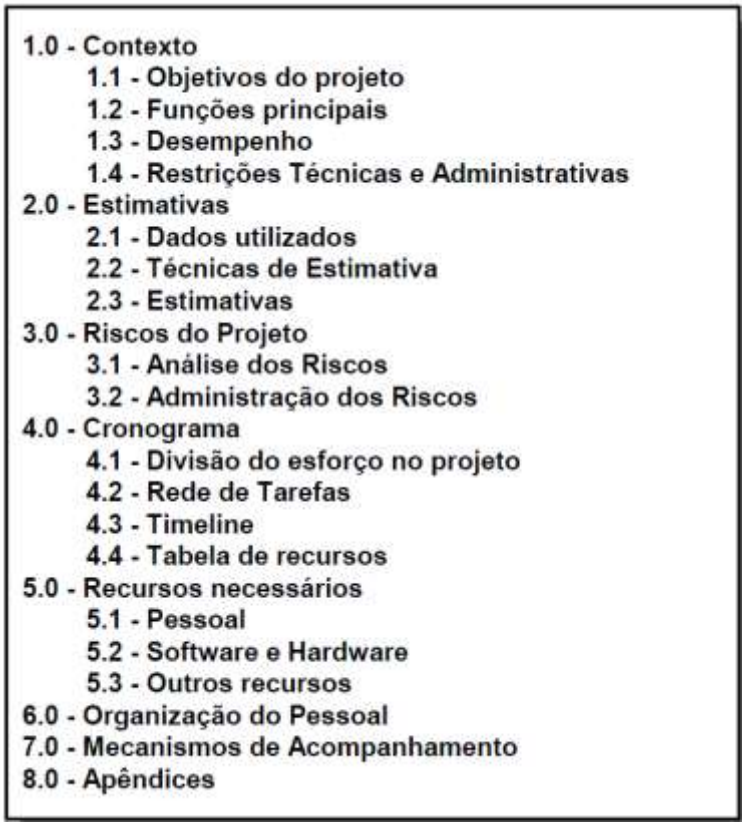

FIGURA 3.5 - ESTRUTURA DO DOCUMENTO DO PLANO DE PROJETO

3.6 – EXERCÍCIOS

1) Qual as principais naturezas de riscos existentes em um projeto de software?.

2) No que se refere análise de riscos, quais as atividades sequenciais que a compõe?.

3) O que são métricas de software?.

4) Qual a diferença entre medições diretas e indiretas?

5) Cite os principais tipos de medições de software?.

REFERÊNCIAS DO CAPÍTULO

FALBO, R.A., Engenharia de Software, Notas de Aula, UFES – Universidade Federal do Espírito Santo.

MAZZOLA, B. V., Apostila de Engenharia de Software

PRESSMAN, R.S., Engenharia de Software, McGraw-Hill, 6ª edição, 2006.

SOMMERVILLE, I., Engenharia de Software, 8ª Edição. São Paulo: Pearson – Addison Wesley, 2007.

CAPÍTULO 4

REQUISITOS DE SOFTWARE

Neste capítulo serão apresentados os conceitos que visam o entendimento das atividades existentes dedicadas aos requisitos de softwares, desde o levantamento dos requisitos até a sua validação.

4.1 - INTRODUÇÃO

O completo entendimento dos requisitos de software é um ponto fundamental para o sucesso de um projeto de software. Independente da precisão com a qual um software venha a ser projetado e implementado, ele certamente trará problemas ao cliente/usuário se a sua análise de requisitos foi mal realizada.

A Análise de Requisitos é uma tarefa que envolve, antes de tudo um trabalho de descoberta, refinamento, modelagem e especificação das necessidades e desejos relativos ao software que deverá ser desenvolvido. Nesta tarefa, tanto o cliente como o desenvolvedor vão desempenhar um papel de grande importância, uma vez que caberá ao primeiro a formulação (de modo concreto) das necessidades em termos de funções e desempenho, enquanto o segundo atua como indagador, consultor e solucionador de problemas.

Esta etapa é de suma importância no processo de desenvolvimento de um software, principalmente porque ela estabelece o elo de ligação entre a alocação do software a nível de sistema (realizada na etapa de Engenharia de Sistema) e o projeto do software. Desta forma, ela permite que o engenheiro de sistemas especifique as necessidades do software em termos de funções e de desempenho, estabeleça as interfaces do software com os demais elementos do sistema e especifique as restrições de projeto. Ao engenheiro de software (ou analista), a análise de requisitos permite uma alocação mais precisa do software no sistema e a construção de modelos do processo, dos dados e dos aspectos comportamentais que serão tratados pelo software. Ao projetista, esta etapa proporciona a obtenção de uma representação da informação e das funções que poderá ser traduzida em projeto procedimental, arquitetônico e de dados. Além disso, é possível definir os critérios de avaliação da qualidade do software a serem verificados uma vez que o software esteja concluído.

Existem diversas definições para requisito de software na literatura, dentre elas: • Requisitos de um sistema são descrições dos serviços que devem ser fornecidos por esse sistema e as suas restrições operacionais (SOMMERVILLE, 2007).

• Um requisito de um sistema é uma característica do sistema ou a descrição de algo que o sistema é capaz de realizar para atingir seus objetivos (PFLEEGER, 2004).

• Um requisito é alguma coisa que o produto tem de fazer ou uma qualidade que ele precisa apresentar (ROBERTSON; ROBERTSON, 2006).

Com base nessas e em outras definições, pode-se dizer que os requisitos de um sistema incluem especificações dos serviços que o sistema deve prover, restrições sob as quais ele deve operar, propriedades gerais do sistema e restrições que devem ser satisfeitas no seu processo de desenvolvimento.

As várias definições acima apresentadas apontam para a existência de diferentes tipos de requisitos. Uma classificação amplamente aceita quanto ao tipo de informação documentada por um requisito faz a distinção entre requisitos funcionais e requisitos não funcionais.

• **Requisitos Funcionais**: são declarações de serviços que o sistema deve prover, descrevendo o que o sistema deve fazer (SOMMERVILLE, 2007). Um requisito funcional descreve uma interação entre o sistema e o seu ambiente (PFLEEGER, 2004), podendo descrever, ainda, como o sistema deve reagir a entradas específicas, como o sistema deve

se comportar em situações específicas e o que o sistema não deve fazer (SOMMERVILLE, 2007).

• **Requisitos Não Funcionais**: descrevem restrições sobre os serviços ou funções oferecidas pelo sistema (SOMMERVILLE, 2007), as quais limitam as opções para criar uma solução para o problema (PFLEEGER, 2004). Neste sentido, os requisitos não funcionais são muito importantes para a fase de projeto (design), servindo como base para a tomada de decisões nessa fase.

Os requisitos não funcionais têm origem nas necessidades dos usuários, em restrições de orçamento, em políticas organizacionais, em necessidades de interoperabilidade com outros sistemas de software ou hardware ou em fatores externos como regulamentos e legislações (SOMMERVILLE, 2007). Assim, os requisitos não funcionais podem ser classificados quanto à sua origem. Existem diversas classificações de requisitos não funcionais. Sommerville (2007), por exemplo, classifica-os em:

- **Requisitos de produto**: especificam o comportamento do produto (sistema). Referem-se a atributos de qualidade que o sistema deve apresentar, tais como confiabilidade, usabilidade, eficiência, portabilidade, manutenibilidade e segurança.
- **Requisitos organizacionais**: são derivados de metas, políticas e procedimentos das organizações do cliente e do desenvolvedor. Incluem requisitos de processo (padrões de processo e modelos de documentos que devem ser usados), requisitos de implementação (tal como a linguagem de programação a ser adotada), restrições de entrega (tempo para chegar ao mercado - time to market, restrições de cronograma etc.), restrições orçamentárias (custo, custo-benefício) etc.
- **Requisitos externos**: referem-se a todos os requisitos derivados de fatores externos ao sistema e seu processo de desenvolvimento. Podem incluir requisitos de interoperabilidade com sistemas de outras organizações, requisitos legais (tais como requisitos de privacidade) e requisitos éticos

Os requisitos devem ser redigidos de modo a serem passíveis de entendimento pelos diversos interessados (stakeholders). Clientes[1], usuários finais e desenvolvedores são todos interessados em requisitos, mas têm expectativas diferentes. Enquanto desenvolvedores e usuários finais têm interesse em detalhes técnicos, clientes requerem descrições mais abstratas. Assim, é útil apresentar requisitos em diferentes níveis de descrição. Sommerville (2007) sugere dois níveis de descrição de requisitos:

[1] É importante notar a distinção que se faz aqui entre clientes e usuários finais. Consideram-se clientes aqueles que contratam o desenvolvimento do sistema e que, muitas vezes, não usarão diretamente o sistema. Eles estão mais interessados nos resultados da utilização do sistema pelos usuários do que no sistema em si. Usuários, por outro lado, são as pessoas que utilizarão o sistema em seu dia a dia. Ou seja, os usuários são as pessoas que vão operar ou interagir diretamente com o sistema.

• **Requisitos de Usuário**: são declarações em linguagem natural acompanhadas de diagramas intuitivos de quais serviços são esperados do sistema e das restrições sob as quais ele deve operar. Devem estar em um nível de abstração mais alto, de modo que sejam compreensíveis pelos usuários do sistema que não possuem conhecimento técnico.

• **Requisitos de Sistema**: definem detalhadamente as funções, serviços e restrições do sistema. São versões expandidas dos requisitos de usuário usados pelos desenvolvedores para projetar, implementar e testar o sistema. Como requisitos de sistema são mais detalhados, as especificações em linguagem natural são insuficientes e para especificá-los, notações mais especializadas devem ser utilizadas.

Uma vez que requisitos de usuário e de sistema têm propósitos e público alvo diferentes, é útil descrevê-los em documentos diferentes. Pfleeger (2004) sugere que dois tipos de documentos de requisitos sejam elaborados:

• **Documento de Definição de Requisitos, ou somente Documento de Requisitos**: deve ser escrito de maneira que o cliente possa entender, na forma de uma listagem do que o cliente espera que o sistema proposto faça. Ele representa um consenso entre o cliente e o desenvolvedor sobre o que o cliente quer.

• **Documento de Especificação de Requisitos:** redefine os requisitos de usuário em termos mais técnicos, apropriados para o desenvolvimento de software, sendo produzido por analistas de requisitos. Vale ressaltar que deve haver uma correspondência direta entre cada requisito de usuário listado no documento de requisitos e os requisitos de sistema tratados no documento de especificação de requisitos.

4.2 - Atividades de Análise de Requisitos

A Engenharia de Requisitos de Software é o ramo da Engenharia de Software que envolve as atividades relacionadas com a definição dos requisitos de software de um sistema, desenvolvidas ao longo do ciclo de vida de software (KOTONYA; SOMMERVILLE, 1998).

O processo de engenharia de requisitos envolve criatividade, interação de diferentes pessoas, conhecimento e experiência para transformar informações diversas (sobre a organização, sobre leis, sobre o sistema a ser construído etc.) em documentos e modelos que direcionem o desenvolvimento de software (KOTONYA; SOMMERVILLE, 1998).

A Engenharia de Requisitos é fundamental, pois possibilita, dentre outros, estimar custo e tempo de maneira mais precisas e melhor gerenciar mudanças em requisitos. Dentre os problemas de um processo de engenharia de requisitos ineficiente, podem-se

citar (KOTONYA; SOMMERVILLE, 1998): (i) requisitos inconsistentes, (ii) produto final com custo maior do que o esperado, (iii) software instável e com altos custos de manutenção e (iv) clientes insatisfeitos. A Engenharia de Requisitos pode ser descrita como um processo, ou seja, um conjunto organizado de atividades que deve ser seguido para derivar, avaliar e manter os requisitos e artefatos relacionados. Uma descrição de um processo, de forma geral, deve incluir, além das atividades a serem seguidas, a estrutura ou sequência dessas atividades, quem é responsável por cada atividade, suas entradas e saídas, as ferramentas usadas para apoiar as atividades e os métodos, técnicas e diretrizes a serem seguidos na sua realização.

Na (Figura 4.1) observamos como é o processo de engenharia de requisitos, no qual vamos trabalhar neste capítulo.

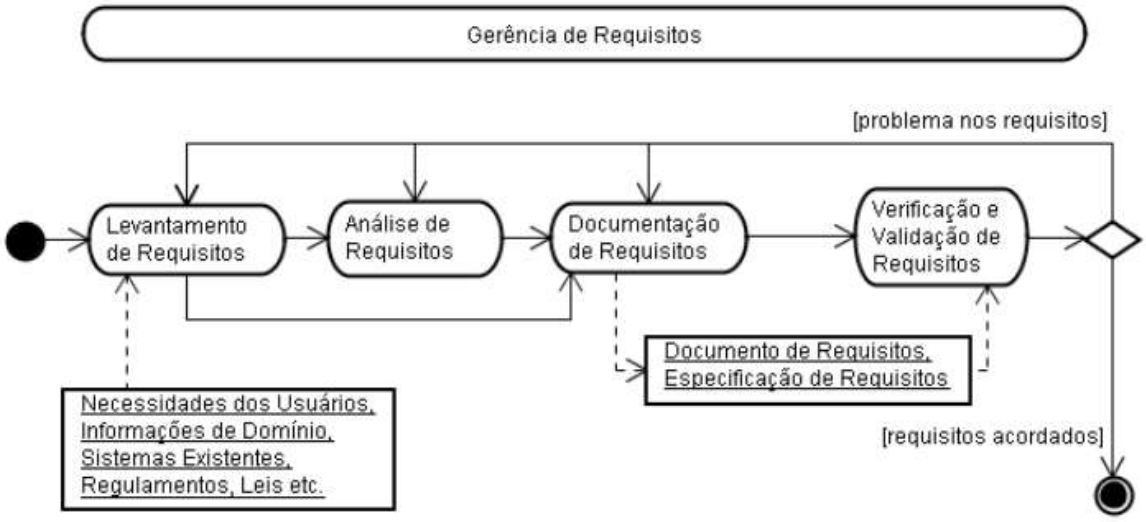

FIGURA 4.1 – PROCESSO DE ENGENHARIA DE REQUISITOS

4.2.1 – LEVANTAMENTO DE REQUISITOS

O levantamento de requisitos corresponde à fase inicial do processo de engenharia de requisitos e envolve as atividades de descoberta dos requisitos. Nessa fase, um esforço conjunto de clientes, usuários e especialistas de domínio é necessário, com o objetivo de entender a organização, seus processos, necessidades, deficiências dos sistemas de software atuais, possibilidades de melhorias, bem como restrições existentes. Trata-se de uma atividade complexa que não se resume somente a perguntar às pessoas o que elas

desejam, mas sim analisar cuidadosamente a organização, o domínio da aplicação e os processos de negócio no qual o sistema será utilizado (KOTONYA; SOMMERVILLE, 1998).

Para levantar quais são os requisitos de um sistema, devem-se obter informações dos interessados (stakeholders), consultar documentos, obter conhecimentos do domínio e estudar o negócio da organização. Neste contexto, quatro dimensões devem ser consideradas, como ilustra a (Figura 4.2) (KOTONYA; SOMMERVILLE, 1998):

FIGURA 4.0.2 - DIMENSÕES DO LEVANTAMENTO DOS REQUISITOS

- **Entendimento do domínio da aplicação**: entendimento geral da área na qual o sistema será aplicado;
- **Entendimento do problema**: entendimento dos detalhes do problema específico a ser resolvido com o auxílio do sistema a ser desenvolvido;
- **Entendimento do negócio**: entender como o sistema irá afetar a organização e como contribuirá para que os objetivos do negócio e os objetivos gerais da organização sejam atingidos;
- **Entendimento das necessidades e das restrições dos interessados**: entender as demandas de apoio para a realização do trabalho de cada um dos interessados no sistema, entender os processos de trabalho a serem apoiados pelo sistema e o papel de eventuais sistemas existentes na execução e condução dos processos de trabalho. Consideram-se interessados no sistema, todas as pessoas que são afetadas pelo sistema de alguma maneira, dentre elas clientes, usuários finais e gerentes de departamentos onde o sistema será instalado.

Para se iniciar um levantamento de requisitos que consiga atender estes pontos acima, não é uma tarefa fácil, o analista deve entender qual a melhor forma de abordagem para evitar que os problemas abaixo causem problemas no futuro. Pode ser difícil compreender e coletar informações quando existem muitos termos desconhecidos, manuais técnicos etc.

• Pessoas que entendem o problema a ser resolvido podem ser muito ocupadas e não ter muito tempo para, juntamente como analista, levantar os requisitos e entender o sistema.

• Políticas organizacionais podem influenciar nos requisitos de um sistema.

• Os interessados não sabem muito bem o que querem do sistema e não conhecem muitos termos.

Na próxima seção veremos as técnicas usadas no levantamento de requisitos.

4.2.1.1 - TÉCNICAS DE LEVANTAMENTO DE REQUISITOS

Hoje existem diversas técnicas que podem ser utilizadas no levantamento de requisitos, as quais podem possuir diferentes objetos de investigação ou podem ter foco em tipos diferentes de requisitos. Assim, é útil empregar várias dessas técnicas concomitantemente, de modo a se ter um levantamento de requisitos mais eficaz. Dentre as várias técnicas, podem ser citadas (FALBO, 2012; KENDALL, 2010; SOMMERVILLE, 1998;):

• **Entrevistas**: Esta técnica é amplamente utilizada, e consiste em conversas direcionadas com um objetivo específico e no formato "pergunta-resposta". Sua finalidade é descobrir problemas a serem tratados, levantar procedimentos importantes e entender a opinião e as expectativas dos entrevistados sobre o sistema.

• **Questionários**: o uso de questionários possibilita ao analista obter informações como postura, crenças, comportamentos e características de várias pessoas que serão afetadas diretamente pelo sistema.

• **Observação**: esta técnica consiste em realizar uma observação do comportamento e o ambiente dos indivíduos de vários níveis organizacionais. Utilizando-se essa técnica, é possível capturar o que realmente é feito e qual tipo de suporte computacional é realmente necessário. Ajuda a confirmar ou refutar informações obtidas com outras técnicas e ajuda a identificar tarefas que podem ser automatizadas e que não foram identificadas pelos interessados.

• **Análise de documentos**: pela análise de documentos existentes na organização, analistas capturam informações e detalhes difíceis de conseguir por entrevista e observação. Documentos revelam um histórico da organização e sua direção.

• **Cenários**: com o uso desta técnica, um cenário de interação entre o usuário final e o sistema é montado e o usuário simula sua interação com o sistema nesse cenário, explicando ao analista o que ele está fazendo e de que informações ele precisa para realizar a tarefa descrita no cenário. O uso de cenários ajuda a entender requisitos, a expor o leque de possíveis interações e a revelar facilidades requeridas.

• **Prototipagem**: um protótipo é uma versão preliminar do sistema, muitas vezes não operacional e descartável, que é apresentada ao usuário para capturar informações específicas sobre seus requisitos de informação, observar reações iniciais e obter sugestões, inovações e informações para estabelecer prioridades e redirecionar planos.

• **Dinâmicas de Grupo**: há várias técnicas de levantamento de requisitos que procuram explorar dinâmicas de grupo para a descoberta e o desenvolvimento de requisitos, tais como Brainstorming e JAD (Joint Application Development). Na primeira, representantes de diferentes grupos de interessados engajam-se em uma discussão informal para rapidamente gerarem o maior número possível de ideias. Na segunda, interessados e analistas se reúnem para discutir problemas a serem solucionados e soluções possíveis. Com as diversas partes envolvidas representadas, decisões podem ser tomadas e questões podem ser resolvidas mais rapidamente. A principal diferença entre JAD e Brainstorming é que, em JAD, tipicamente os objetivos do sistema já foram estabelecidos antes dos interessados participarem. Além disso, sessões JAD são normalmente bem estruturadas, com passos, ações e papéis de participantes definidos.

4.2.2 - ANÁLISE DE REQUISITOS

Uma vez identificados requisitos, é possível iniciar a atividade de análise, quando os requisitos levantados são usados como base para a modelagem do sistema. Conforme discutido anteriormente, requisitos de usuário são escritos tipicamente em linguagem natural, pois eles devem ser compreendidos por pessoas que não sejam especialistas técnicos. Contudo, é útil expressar requisitos mais detalhados do sistema de maneira mais técnica. Para tal, diversos tipos de modelos podem ser utilizados. Esses modelos são representações gráficas que descrevem processos de negócio, o problema a ser resolvido e o sistema a ser desenvolvido. Por utilizarem representações gráficas, modelos são geralmente mais compreensíveis do que descrições detalhadas em linguagem natural (SOMMERVILLE, 2007).

A análise de requisitos é uma atividade extremamente vinculada ao levantamento de requisitos. Durante o levantamento de requisitos, alguns problemas são identificados e tratados. Entretanto, determinados problemas somente são identificados por meio de

uma análise mais detalhada. A análise de requisitos ajuda a entender e detalhar os requisitos levantados, a descobrir problemas nesses requisitos e a obter a concordância sobre as alterações, de modo a satisfazer a todos os envolvidos. Seu objetivo é estabelecer um conjunto acordado de requisitos completos, consistentes e sem ambiguidades, que possa ser usado como base para as demais atividades do processo de desenvolvimento de software (KOTONYA; SOMMERVILLE, 1998).

De forma resumida, pode-se dizer que a análise atende a dois propósitos principais:

(i) Prover uma base para o entendimento e concordância entre clientes e desenvolvedores sobre o que o sistema deve fazer e

(ii) Prover uma especificação que guie os desenvolvedores na demais etapas do desenvolvimento, sobretudo no projeto, implementação e testes do sistema (PFLEEGER, 2004).

O processo de engenharia de requisitos é dominado por fatores humanos, sociais e organizacionais. Ele envolve pessoas de diferentes áreas de conhecimento e com objetivos individuais e organizacionais diferentes. Dessa forma, é comum que cada indivíduo tente influenciar os requisitos para que seu objetivo seja alcançado, sem necessariamente alcançar os objetivos dos demais (KOTONYA; SOMMERVILLE, 1998).

Problemas e conflitos encontrados nos requisitos devem ser listados. Usuários, clientes, especialistas de domínio e engenheiros de requisitos devem discutir os requisitos que apresentam problemas, negociar e chegar a um acordo sobre as modificações a serem feitas. Idealmente, as discussões devem ser governadas pelas necessidades da organização, incluindo o orçamento e o cronograma disponíveis. No entanto, muitas vezes, as negociações são influenciadas por considerações políticas e os requisitos são definidos em função da posição e da personalidade dos indivíduos e não em função de argumentos e razões (KOTONYA; SOMMERVILLE, 1998).

A maior parte do tempo da negociação é utilizada para resolver conflitos de requisitos. Quando discussões informais entre analistas, especialistas de domínio e usuários não forem suficientes para resolver os problemas, é necessária a realização de reuniões de negociação, que envolvem (KOTONYA; SOMMERVILLE, 1998):

- Discussão: os requisitos que apresentam problemas são discutidos e os interessados presentes opinam sobre eles.
- Priorização: requisitos são priorizados para identificar requisitos críticos e ajudar nas decisões e planejamento.
- Concordância: soluções para os problemas são identificadas, mudanças são feitas e um acordo sobre o conjunto de requisitos é acertado.

4.2.3 - Documentação dos Requisitos

Os requisitos e modelos capturados nas etapas anteriores devem ser descritos e apresentados em documentos. A documentação é, portanto, uma atividade de registro e oficialização dos resultados da engenharia de requisitos. Como resultado, um ou mais documentos devem ser produzidos.

Uma documentação que fornece uma boa fonte de informações para o alcance de benefícios precisa conter as seguintes características:

- Facilitar a comunicação dos requisitos;
- Reduzir o esforço de desenvolvimento, pois sua preparação força usuários e clientes a considerar os requisitos atentamente, evitando retrabalho nas fases posteriores;
- Fornece uma base realística para estimativas;
- Fornece uma base para verificação e validação;
- Facilitar na transferência do software para novos usuários e/ou máquinas;
- Servir como base para futuras manutenções ou incremento de novas funcionalidades.

A documentação dos requisitos tem um conjunto diversificado de interessados, dentre eles (SOMMERVILLE, 2007; KOTONYA; SOMMERVILLE, 1998):

- Clientes, Usuários e Especialistas de Domínio: são interessados na documentação de requisitos, uma vez que atuam na especificação, avaliação e alteração de requisitos.
- Gerentes de Cliente: utilizam o documento de requisitos para planejar um pedido de proposta para o desenvolvimento de um sistema, contratar um fornecedor e para acompanhar o desenvolvimento do sistema.
- Gerentes de Fornecedor: utilizam a documentação dos requisitos para planejar uma proposta para o sistema e para planejar e acompanhar o processo de desenvolvimento.
- Desenvolvedores (analistas, projetistas, programadores e mantenedores): utilizam a documentação dos requisitos para compreender o sistema e as relações entre suas partes.
- Testadores: utilizam a documentação dos requisitos para projetar casos de teste, sobretudo testes de validação do sistema.

O **Documento de Definição de Requisitos**, ou simplesmente Documento de Requisitos, deve conter uma descrição do propósito do sistema, uma breve descrição do domínio do problema tratado pelo sistema e listas de requisitos funcionais, não funcionais e regras de negócio, descritos em linguagem natural (requisitos de usuário).

Para facilitar a identificação e rastreamento dos requisitos, devem-se utilizar identificadores únicos para cada um dos requisitos listados. O público-alvo deste documento são os clientes, usuários, gerentes (de cliente e de fornecedor) e desenvolvedores.

O **Documento de Especificação de Requisitos** deve conter os requisitos escritos a partir da perspectiva do desenvolvedor, devendo haver uma correspondência direta com os requisitos no Documento de Requisitos, de modo a se ter requisitos rastreáveis. Os vários modelos produzidos na fase de análise devem ser apresentados no Documento de Especificação de Requisitos, bem como glossários de termos usados e outras informações julgadas relevantes.

4.2.4 - VERIFICAÇÃO E VALIDAÇÃO DOS REQUISITOS

As atividades de Verificação & Validação (V&V) devem ser iniciadas o quanto antes no processo de desenvolvimento de software, pois quanto mais tarde no projeto, os defeitos forem encontrados, o custo será maior para realizar à sua correção, uma vez sabendo que os requisitos são a base para o desenvolvimento. Também é fundamental que eles sejam cuidadosamente avaliados. Assim, os documentos produzidos durante a atividade de documentação de requisitos devem ser submetidos à verificação e à validação.

É importante realçar a diferença entre verificação e validação.

O objetivo da verificação é assegurar que o software esteja sendo construído de forma correta. Deve-se verificar se os artefatos produzidos atendem aos requisitos estabelecidos e se os padrões organizacionais (de produto e processo) foram consistentemente aplicados.

Já o objetivo da validação é assegurar que o software que está sendo desenvolvido é o software correto, ou seja, assegurar que os requisitos, e o software deles derivado, atendem ao uso proposto (ROCHA; MALDONADO; WEBER, 2001). No caso de requisitos, a verificação é feita, sobretudo, em relação à consistência entre requisitos e modelos e à conformidade com padrões organizacionais de documentação de requisitos. Já a validação tem de envolver a participação de usuários e clientes, pois somente eles são capazes de dizer se os requisitos atendem aos propósitos do sistema.

Mas no caso de requisitos, a verificação é feita, sobretudo, em relação à consistência entre requisitos e modelos e à conformidade com padrões organizacionais de documentação de requisitos. Já a validação tem de envolver a participação de usuários e clientes, pois somente eles são capazes de dizer se os requisitos atendem aos propósitos do sistema.

Em via de regra os requisitos tanto funcionais quanto os não funcionais devem conter os seguintes aspectos:

- **Completo**: o requisito deve descrever completamente a funcionalidade a ser entregue (no caso de requisito funcional), a regra de negócio a ser tratada (no caso de regras de negócio) ou a restrição a ser considerada (no caso de requisito não funcional). Ele deve conter as informações necessárias para que o desenvolvedor possa projetar, implementar e testar essa funcionalidade, regra ou restrição.
- **Correto**: cada requisito deve descrever exatamente a funcionalidade, regra ou restrição a ser construída.
- **Consistente**: o requisito não deve ser ambíguo ou conflitar com outro requisito.
- **Realista**: deve ser possível implementar o requisito com a capacidade e com as limitações do sistema e do ambiente de desenvolvimento.
- **Necessário**: o requisito deve descrever algo que o cliente realmente precisa ou que é requerido por algum fator externo ou padrão da organização.
- **Passível** de ser priorizado: os requisitos devem ter ordem de prioridade para facilitar o gerenciamento durante o desenvolvimento do sistema.
- **Verificável e passível de confirmação**: deve ser possível desenvolver testes para verificar se o requisito foi realmente implementado.
- **Rastreável**: deve ser possível identificar quais requisitos foram tratados em um determinado artefato, bem como identificar que produtos foram originados a partir de um requisito.

4.3 EXERCÍCIOS

1) Quais os principais tipos de requisitos de software?.

2) Existem diversas classificações de requisitos não funcionais, como Sommerville classifica-os?.

3) Quais são os dois principais tipos de documentos de requisitos?.

4) Quais as fases que compõem o gerenciamento de requisitos?.

5) Quais as principais técnicas de levantamento de requisitos?.

6) Quais os aspectos que tanto os requisitos funcionais e não funcionais devem possuir?.

REFERÊNCIAS DO CAPÍTULO

FALBO, R.A., Engenharia de Requisitos, Notas de Aula, UFES – Universidade Federal do Espírito Santo.

KENDALL, K.E., KENDALL, J.E.; Systems Analysis and Design, Prentice Hall, 8th Edition, 2010.

KOTONYA, G., SOMMERVILLE, I., Requirements engineering: processes and techniques. Chichester, England: John Wiley, 1998.

PFLEEGER, S.L., Engenharia de Software: Teoria e Prática, São Paulo: Prentice Hall, 2ª edição, 2004.

PRESSMAN, R.S., Engenharia de Software, McGraw-Hill, 6ª edição, 2006.

ROCHA, A.R.C., MALDONADO, J.C., WEBER, K.C., Qualidade de Software: Teoria e Prática. São Paulo: Prentice Hall, 2001.

SOMMERVILLE, I., Engenharia de Software, 8ª Edição. São Paulo: Pearson – Addison Wesley, 2007.

CAPÍTULO 5

ANÁLISE E PROJETO DE SOFTWARE

Neste capítulo serão apresentados os principais aspectos relativos ao projeto de software, como etapa fundamental para a obtenção de um software de qualidade. Serão apresentados os passos fundamentais de um projeto de software e seus principais aspectos.

5.1 - INTRODUÇÃO

O que é "Projeto de Software”? É a representação significativa de alguma coisa que será construída. Em engenharia de software, o Projeto de Software é a fase de desenvolvimento, na qual são feitos modelos com todas as entidades que serão construídas posteriormente a partir dos requisitos do sistema. O projeto de software foca em 4 áreas, como: dados, arquitetura, interface e componentes. Para garantir que um projeto está sendo feito com qualidade é necessário avaliar continuamente pontos referentes à sua corretude, completude, clareza e consistência com os requisitos do sistema.

A etapa de Projeto antecede a fase de Desenvolvimento. O Projeto consiste na aplicação de um conjunto de técnicas e princípios, de modo a definir um sistema num nível de detalhe suficiente à sua realização física. A tarefa do projetista nada mais é do que produzir um modelo de representação do software que será implementado. Nesta etapa, os requisitos definidos na etapa anterior devem servir de referência para a obtenção da representação do software.

5.2 - PROJETO PRELIMINAR E DETALHADO

A etapa de projeto é caracterizada pelo conjunto de atividades que vai permitir traduzir os requisitos definidos na etapa anterior em uma representação do software a ser construído. Na sua forma mais clássica, o primeiro resultado obtido no projeto é uma visão da estrutura do software em termos de componentes, sendo que, a partir de procedimentos de refinamento, chega-se a um nível de especificação bastante próxima da codificação do programa.

Do ponto de vista do gerenciamento do processo de desenvolvimento, a etapa de projeto é conduzida basicamente em dois principais estágios:

- O **projeto preliminar**, o qual permite estabelecer, a partir dos requisitos, a arquitetura do software e da informação relacionada;
- O **projeto detalhado**, o qual permite aperfeiçoar a estrutura do software e definir representações algorítmicas de seus componentes.

No contexto dos projetos preliminar e detalhado, um conjunto de atividades técnicas de projeto são desenvolvidas. Num ponto de vista mais genérico, pode-se destacar os projetos *arquiteturais*, *procedimentais* e *de dados*. Em projetos mais recentes, e, particularmente, naqueles envolvendo interatividade com o usuário, o *projeto de interfaces* tem assumido um papel de fundamental importância, sendo considerada uma atividade do mesmo nível dos demais projetos.

5.3 - Aspectos Fundamentais do Projeto

Durante esta etapa, é importante que alguns conceitos sejam levados em conta para que se possa derivar um projeto contendo as características citadas acima. As seções que seguem discutirão alguns destes conceitos.

5.3.1 - Modularidade

O conceito de modularidade tem sido utilizado já há bastante tempo, como forma de obtenção de um software que apresente algumas características interessantes como a facilidade de manutenção. Este conceito apareceu como uma solução aos antigos softwares "monolíticos", os quais representavam grandes dificuldades de entendimento e, consequentemente para qualquer atividade de manutenção. A utilização do conceito de modularidade oferece resultados a curto prazo, uma vez que, ao dividir-se um grande problema em problemas menores, as soluções são encontradas com esforço relativamente menor. Isto significa que, quanto maior o número de módulos definidos num software, menor será o esforço necessário para desenvolvê-lo, uma vez que o esforço de desenvolvimento de cada módulo será menor. Por outro lado, quanto maior o número de módulos, maior será o esforço no desenvolvimento das interfaces, o que permite concluir que esta regra deve ser utilizada com moderação.

Finalmente, é importante distinguir o conceito de modularidade de projeto com o de modularidade de implementação. Nada impede que um software seja projetado sob a ótica da modularidade e que sua implementação seja monolítica. Em alguns casos, como forma de evitar desperdício de tempo de processamento e de memória em chamadas de procedimentos (salvamento e recuperação de contexto), impõe-se o desenvolvimento de um programa sem a utilização de funções e procedimentos. Ainda assim, uma vez que o projeto seguiu uma filosofia de modularidade, o software deverá apresentar alguns benefícios resultantes da adoção deste princípio.

5.3.2 - Arquitetura de Software

O conceito de arquitetura de software está ligado aos dois principais aspectos do funcionamento de um software: a estrutura hierárquica de seus componentes (ou módulos) e as estruturas de dados. A arquitetura de software resulta do desenvolvimento de atividades de particionamento de um problema, encaminhadas desde a etapa de Análise de Requisitos. Naquela etapa, é dado o pontapé inicial para a definição das estruturas de dados e dos componentes de software. A solução é encaminhada ao longo do projeto, através da definição de um ou mais elementos de software que solucionarão uma parte do problema global.

A arquitetura do software define as relações entre os principais elementos estruturais do software, os “padrões de projeto”, que podem ser usados para satisfazer os requisitos que tenham sido definidos para o sistema e as restrições que afetam o modo pelo qual os padrões de projeto arquitetural podem ser aplicados. A representação arquitetural pode ser derivada da especificação do sistema, do modelo de análise e da interação dos subsistemas definida no modelo de análise.

É importante lembrar que não existe técnica de projeto que garanta a unicidade de solução a um dado problema. Diferentes soluções em termos de arquitetura podem ser derivadas a partir de um mesmo conjunto de requisitos de software. A grande dificuldade concentra-se em definir qual a melhor opção em termos de solução. Segue um exemplo de arquitetura de software conforme (Figura 5.1).

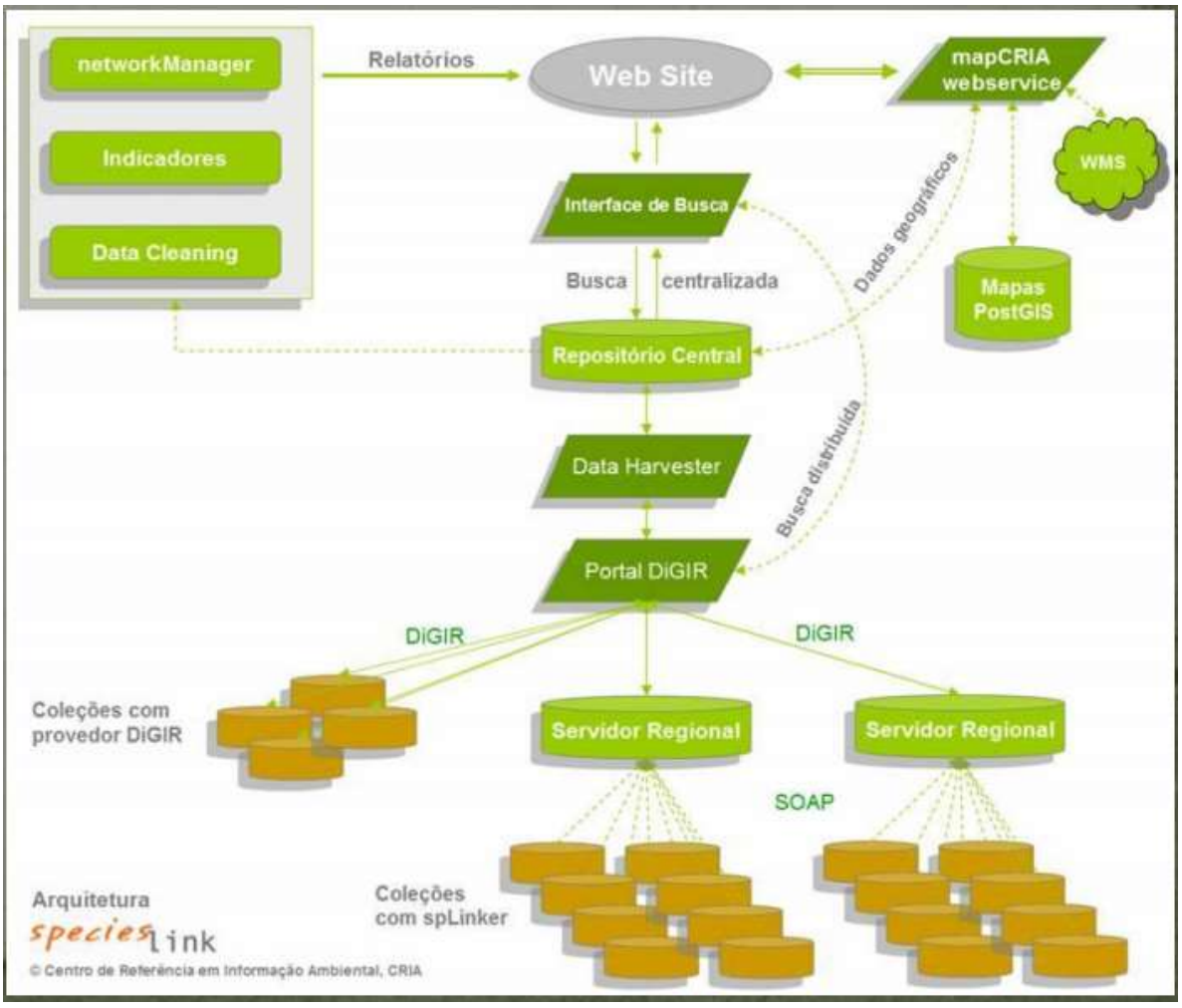

FIGURA 5.1 - EXEMPLO DE ARQUITETURA DE SOFTWARE

5.3.3 - ARQUITETURA DE DADOS

Transforma o modelo do domínio de informação, criado durante a análise, nas estruturas de dados que vão ser necessárias para implementar o software. Os objetos de dados e as relações definidas no Diagrama Entidade-Relacionamento, bem como o conteúdo detalhado dos dados mostrado no dicionário de dados, fornecem a base para a atividade de projeto de dados. Ë importante ressaltar que no nível de dados e arquitetura, o projeto concentra-se em padrões que interessam a aplicação a ser construída conforme exemplo ilustrado na (Figura 5.2).

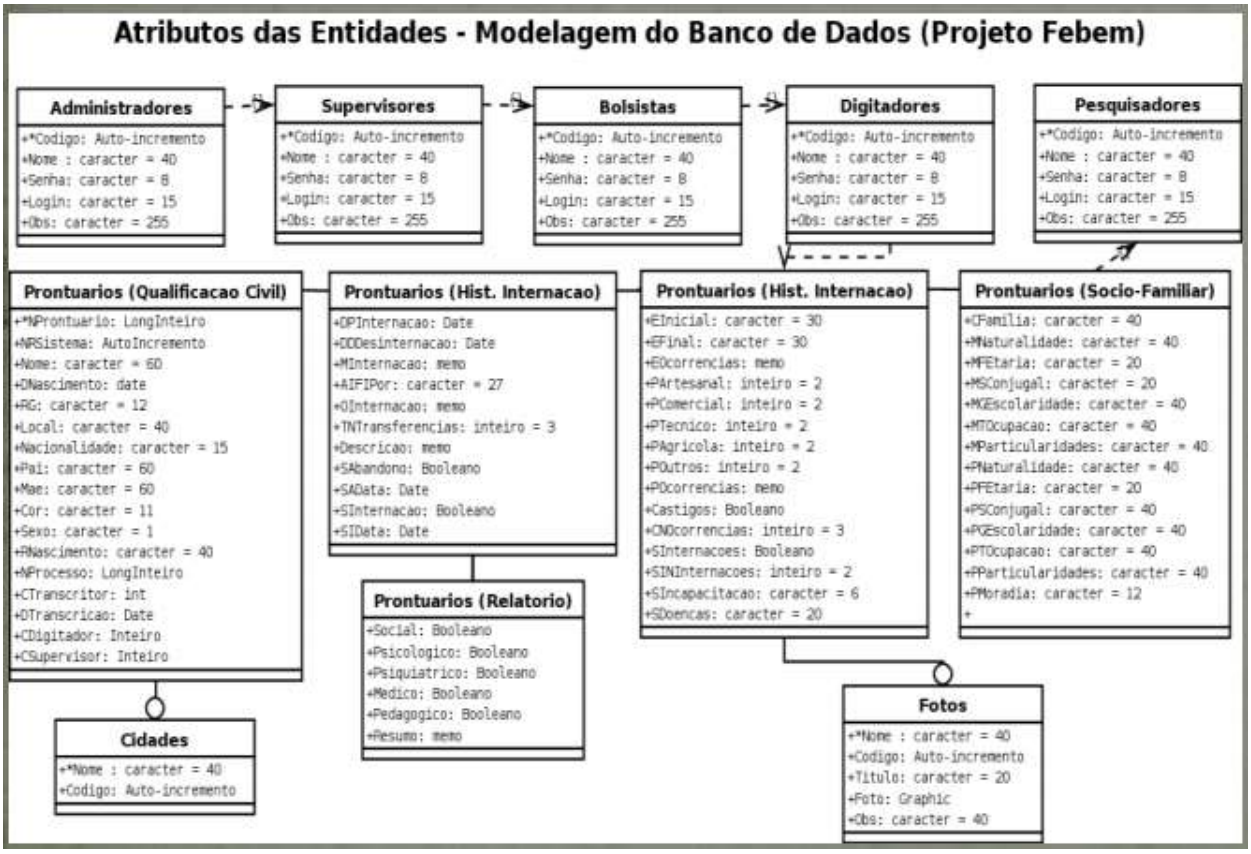

FIGURA 5.2 - EXEMPLO DE UMA ARQUITETURA DE DADOS

5.3.4 - PROJETO DE INTERFACE

Descreve como o software se comunica com ele mesmo, com os sistemas que interoperam com ele e com as pessoas que o utilizam. Uma interface implica um fluxo de informação (p.ex., dados e ou controle) e um tipo de comportamento especifico). Portanto o projeto de Interface tem o objetivo de descrever como deverá se dar a comunicação entre os elementos da arquitetura (interfaces internas), a comunicação do sistema em desenvolvimento com outros sistemas (interfaces externas) e com as pessoas

que vão utilizá-lo (interface com o usuário).. Na (Figura 5.3 e 5.4) podemos observar um projeto de Interface em duas perspectivas.

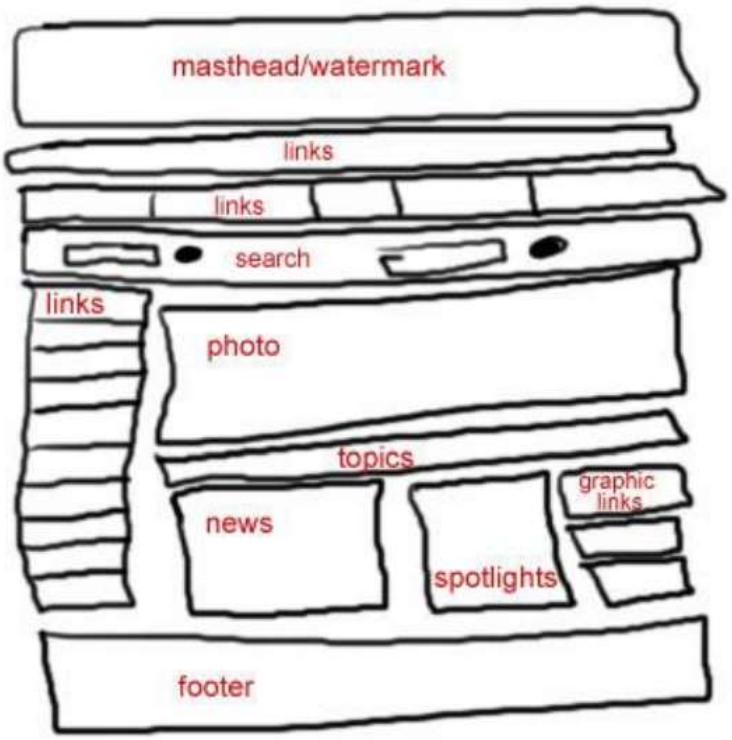

FIGURA 5.3 - EXEMPLO DE PROJETO DE INTERFACE DESENHADA

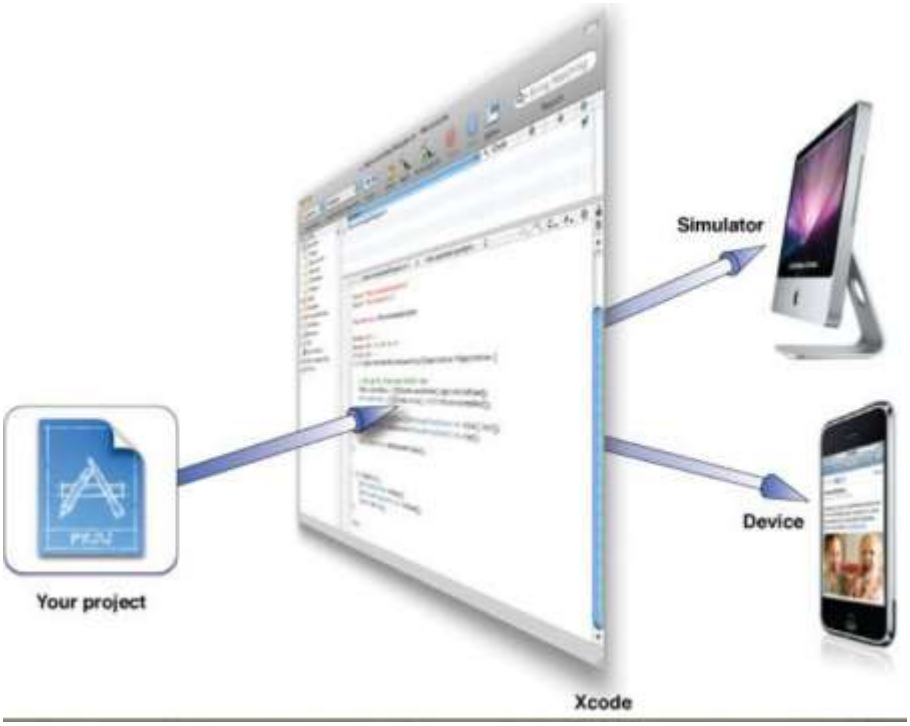

FIGURA 5.4 - EXEMPLO DE PROJETO DE INTERFACE COMPLEMENTAR

5.3.5 - ARQUITETURA DE COMPONENTES

Transforma elementos estruturais da arquitetura de software numa descrição procedimental dos componentes de software. A importância do projeto de software pode ser definida com uma única palavra "Qualidade". Projeto é a etapa na qual a qualidade é incorporada na engenharia de software. O projeto é o único modo pelo qual pode-se traduzir precisamente os requisitos do cliente num produto ou sistema de software acabado.

A fase de projeto tem início com o modelo de requisitos. Onde deve ser transformado em quatro níveis de detalhes:

- Arquitetura de dados (Diagrama Entidade/Relacionamento)
- Arquitetura do sistema (Diagrama de Classes, Diagrama de Caso de Uso Projeto de Arquitetura, outros diagramas UML),
- Representação da interface (Arquitetura de Interface)
- Detalhe a nível de componente (Diagrama de Componentes).

5.4 - ARQUITETURAS DE SOFTWARES

Nesta seção iremos demonstrar os diversos tipos de arquitetura de software existente, com o objetivo de conceituar o aluno nas variadas possibilidades.

Com o passar do tempo e com o surgimento dos computadores pessoais, cada vez mais microcomputadores estavam disponíveis nas mesas dos usuários, fato que foi tornando necessária a utilização do poder de processamento destas máquinas dentro do sistema. Também devido à grande expansão das redes de computadores, os métodos de desenvolvimento de *software* foram aos poucos evoluindo para uma arquitetura descentralizada, na qual não somente o servidor é o responsável pelo processamento, mas as estações clientes também assumem parte desta tarefa.

Dentro deste contexto que surgiu os seguintes modelos:

Modelo de duas camadas ou cliente servidor – Tem o objetivo de dividir a carga de processamento entre o servidor e as máquinas clientes. Igualmente conhecido como modelo cliente e servidor de duas camadas, esta técnica é composta por duas partes distintas: uma executada na estação cliente e outra no servidor.

A camada cliente tem a função de prover a interface para que os usuários possam manipular as informações, ou seja, através dela realiza-se a interação entre o usuário e o sistema. É desenvolvida para se conectar diretamente ao banco de dados, tendo como

responsabilidade fazer as solicitações dos dados necessários ao servidor, sendo que este os processa e devolve o resultado.

Neste modelo, as regras de negócios (tais como funções, validações entre outros) podem ficar armazenadas no cliente, no servidor ou em ambos. Quando contidas no cliente, apresentam-se na forma de códigos da linguagem de programação que está sendo utilizada. Já quando localizadas no servidor, estão na forma de recursos do banco de dados, como *triggers* e *stored procedures*, por exemplo. O cliente recebe a denominação de "cliente gordo" quando a maior parte das regras são nele implementadas, enquanto que o servidor recebe a qualificação de "servidor gordo" quando as regras são nele desenvolvidas em maior número.

Em suma, a base do funcionamento desta técnica consiste em armazenar determinado volume de dados em um computador central e deixa-lo encarregado de manipulá-los e devolve-los à estação cliente que os requisitou. A (Figura 5.5) ilustra como é uma arquitetura de duas camadas.

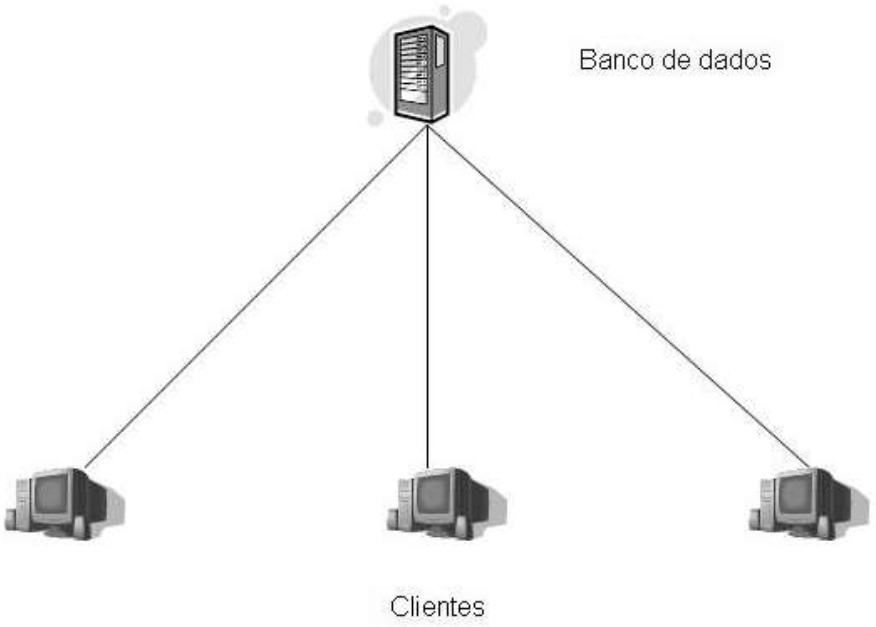

FIGURA 5.5 – ARQUITETURA DE DUAS CAMADAS (CLIENTE/SERVIDOR)

Arquitetura de Multicamadas também conhecido como modelo cliente e servidor de várias camadas, este método é uma evolução da tecnologia de duas camadas e tem como

princípio básico o fato de que a estação cliente jamais realiza comunicação direta com o servidor de banco de dados, mas sim com uma camada intermediária, e esta, com o banco de dados. Isto proporciona uma série de vantagens sobre a técnica de duas camadas, as quais serão explanadas adiante.

Um sistema multicamadas faz uso de objetos distribuídos aliados à utilização de interfaces para executar seus procedimentos, o que torna o sistema independente de localização, podendo estar tanto na mesma máquina como em máquinas separadas. Desta forma, a aplicação pode ser dividida em várias partes, cada uma bem definida, com suas características e responsável por determinadas funções. Em um aplicativo nestes moldes, pelo menos três camadas são necessárias: apresentação, regras de negócios e banco de dados.

A seguir, cada uma das partes do modelo é explicada.

Apresentação

A camada de apresentação fica fisicamente localizada na estação cliente e é responsável por fazer a interação do usuário com o sistema. É uma camada bastante leve, que basicamente executa os tratamentos de telas e campos e geralmente acessa somente a segunda camada, a qual faz as requisições ao banco de dados e devolve o resultado. É também conhecida como cliente, regras de interface de usuário ou camada de interface.

Regras de negócios

Em um sistema seguindo este modelo, a aplicação cliente nunca acessa diretamente a última camada que é a do banco de dados, pois quem tem essa função é a camada de regras de negócios, na qual podem se conectar diversas aplicações clientes.

Esta parte do sistema é responsável por fazer as requisições ao banco de dados e todo o seu tratamento, ou seja, somente ela que tem acesso direto ao banco de dados. É também conhecida como lógica de negócios, camada de acesso a dados, camada intermediária ou servidor de aplicação por geralmente se tratar de um outro computador destinado somente ao processamento das regras. O servidor de aplicação é, geralmente, uma máquina dedicada e com elevados recursos de *hardware*, uma vez que é nele que ficam armazenados os métodos remotos (regras de negócios) e é realizado todo o seu tratamento e processamento.

Banco de dados

É a última divisão do modelo, na qual fica localizado o sistema gerenciador de banco de dados. É também conhecida como camada de dados.

Adicionalmente a essas três divisões, também pode ser implementada uma camada somente para validação, na qual são executados todos os procedimentos necessários para garantir a integridade dos dados digitados na camada de apresentação.

A (Figura 5.6) ilustra o esquema de comunicação de um sistema multicamadas.

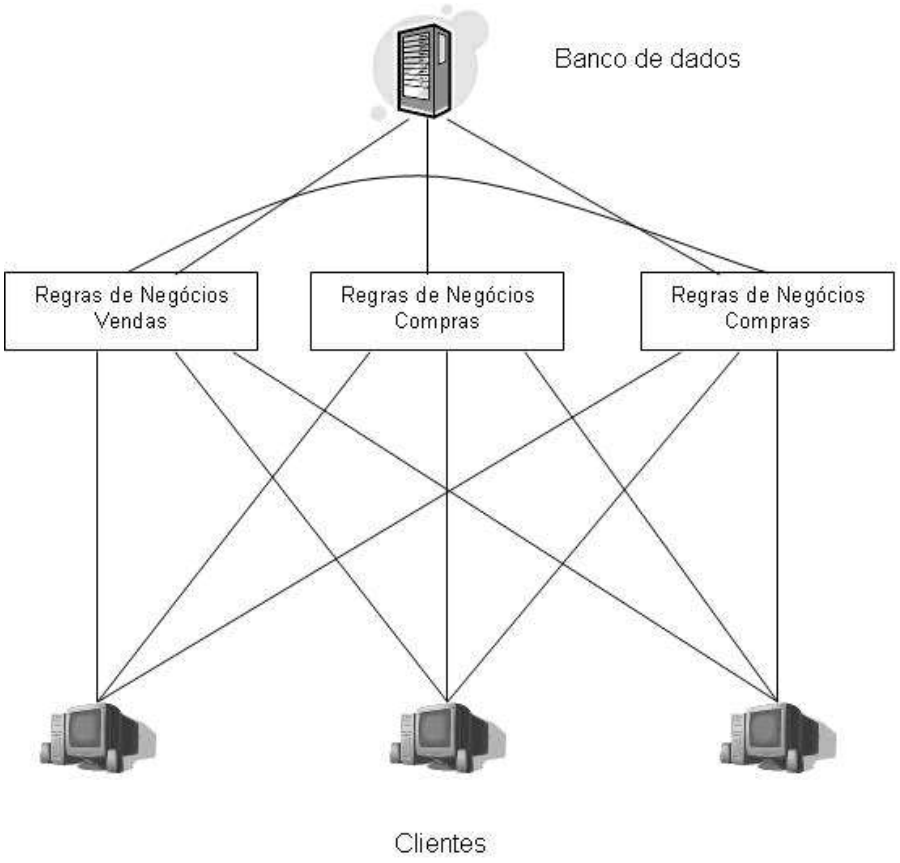

FIGURA 5.6 - ARQUITETURA MULTICAMADAS

O padrão arquitetural **Model-View-Controller (MVC)** é uma forma de quebrar uma aplicação, ou até mesmo um pedaço da interface de uma aplicação, em três partes: **o modelo, a visão e o controlador**.

O MVC inicialmente foi desenvolvido no intuito de mapear o método tradicional de entrada, processamento, e saída que os diversos programas baseados em GUI utilizavam. E para demonstrar melhor como é seu funcionamento, segue sua ilustração na (Figura 5.7).

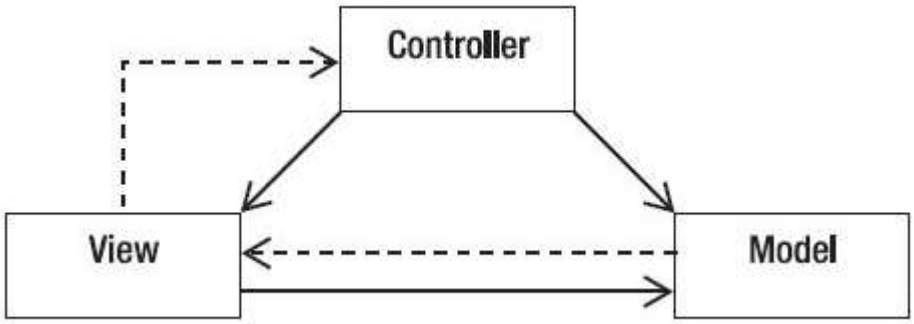

FIGURA 5.7 - ARQUITETURA MVC

Explicando cada um dos objetos do padrão MVC tem-se primeiramente o controlador (Controller) que interpreta as entradas do mouse ou do teclado enviado pelo usuário e mapeia essas ações do usuário em comandos que são enviados para o modelo (Model) e/ou para a janela de visualização (View) para efetuar a alteração apropriada. Por sua vez o modelo (Model) gerencia um ou mais elementos de dados, responde a perguntas sobre o seu estado e responde a instruções para mudar de estado. O modelo sabe o que o aplicativo quer fazer e é a principal estrutura computacional da arquitetura, pois é ele quem modela o problema que está se tentando resolver. Por fim, a visão (View) gerencia a área retangular do display e é responsável por apresentar as informações para o usuário através de uma combinação de gráficos e textos. A visão não sabe nada sobre o que a aplicação está atualmente fazendo, tudo que ela realmente faz é receber instruções do controle e informações do modelo e então exibir elas. A visão também se comunica de volta com o modelo e com o controlador para reportar o seu estado.

5.5 - PADRÕES DE PROJETOS

Citado por (LEITE, 2015), Padrões de Projeto (Desing Patterns) são soluções testadas e aprovadas para um problema geral. Um padrão vem com diretrizes sobre quando usá-lo, bem como vantagens e desvantagens de seu uso, sendo levado em consideração que já foram cuidadosamente considerados por outras pessoas e aplicado diversas vezes na solução de problemas anteriores. Um padrão de projeto normalmente tem o formato de um par nomeado problema/solução, que pode ser utilizado em novos contextos, com orientações sobre como utilizá-lo.

Embora um padrão seja a descrição de um problema, de uma solução genérica e sua justificativa, isso não significa que qualquer solução conhecida para um problema possa constituir um padrão, pois existem características obrigatórias que devem ser atendidas pelos padrões:

1. Devem possuir um nome, que descreva o problema, as soluções e consequências. Um nome permiti definir o vocabulário a ser utilizado pelos projetistas e desenvolvedores em um nível mais alto de abstração.
2. Todo padrão deve relatar de maneira clara a qual(is) problema(s) ele deve ser aplicado, ou seja, quais são os problemas que quando inserido em um determinado contexto o padrão conseguirá resolvê-lo. Alguns podendo exigir pré-condições.
3. Solução descreve os elementos que compõem o projeto, seus relacionamentos, responsabilidades e colaborações. Um padrão deve ser uma solução concreta, ele deve ser exprimido em forma de gabarito (algoritmo) que, no entanto pode ser aplicado de maneiras diferentes.
4. Todo padrão deve relatar quais são as suas consequências para que possa ser analisada a solução alternativa de projetos e para a compreensão dos benefícios da aplicação do projeto.

A seleção de um padrão de projeto é um dos passos mais difíceis para iniciar o uso de padrões para um projeto particular. A seguir veremos um breve resumo sobre os principais padrões de projeto (LEITE, 2015):

- **Singleton** - Assegura que uma classe tem apenas uma instância.
- **Iterator** - Permite o acesso aos elementos de uma agregação de uma forma sequencial sem expor a representação.
- **Façade** - Fornece uma interface uniforme a um conjunto de interfaces num subsistema. Este padrão define uma interface de alto nível que permite que os subsistemas sejam mais simples de utilizar.
- **Factory Method** - Define uma interface para a criação de um objeto, mas deixa às subclasses a tarefa de definir que classes instanciarem.
- **Abstract Factory** - Fornecer uma interface para a criação de famílias de objetos relacionados, ou dependentes, em especificar as suas classes concretas.
- **Observer** - Define uma dependência 1-para-n entre objetos, de modo que quando o estado de um objeto é alterado todos seus dependentes são notificados e atualizados automaticamente.
- **Adapter** - Converte a interface de uma classe em outra interface que os clientes esperam. Este padrão permite que classes que não poderiam trabalhar juntas devido a interfaces incompatíveis trabalhem juntas.
- - Visa desacoplar uma abstração de sua implementação, de modo que as duas possam variar independentemente.
- **Decorator** - Anexa responsabilidades adicionais a um objeto dinamicamente, além de oferecer uma alternativa flexível ao uso de herança para estender uma funcionalidade.

- **Memento** - Não viola o encapsulamento, a captura e a externalização do estado interno de um objeto, para que o objeto possa ter esse estado restaurado posteriormente.

5.6 - EXERCÍCIOS

1) Do ponto de vista do gerenciamento do processo de desenvolvimento, a etapa de projeto é conduzida basicamente em dois principais estágios, quais são eles?.

2) Um projeto de software deve possuir algumas características importantes, quais são as principais características que não podem faltar em um bom projeto de software?.

3) O que é Projeto de Software ?.

4) Qual é o tipo de projeto que se dedica a iteração com o usuário através das telas no qual se comunicam entre si?.

5) Quais são as principais arquiteturas de software?.

6) Qual o padrão de projeto que assegura que uma classe tem apenas uma instância?.

7) Qual o padrão de projeto que fornece uma interface uniforme a um conjunto de interfaces num subsistema. Este padrão define uma interface de alto nível que permite que os subsistemas sejam mais simples de utilizar?.

REFERÊNCIAS DO CAPÍTULO

LEITE F. A., Padrões de Projeto, DEVMEDIA, Acessado em 15/08/2015, <http://www.devmedia.com.br/conheca-os-padroes-de-projeto/957>.

PRESSMAN, R.S., Engenharia de Software, McGraw-Hill, 6ª edição, 2006.

SOMMERVILLE, I., Engenharia de Software, 8ª Edição. São Paulo: Pearson – Addison Wesley, 2007.

MAZZOLA, B. V., Apostila de Engenharia de Software

CAPÍTULO 6

DESENVOLVIMENTO E TESTE DE SOFTWARE

Neste capítulo veremos os aspectos relativos ao desenvolvimento e testes de software. Serão apresentadas as características das linguagens de programação, e como selecionar a melhor linguagem para seu projeto, além das abordagens de testes de software e seus diversos tipos.

6.1 - Introdução ao Desenvolvimento de Software

Ainda que um projeto bem elaborado facilite a implementação, essa tarefa não é necessariamente fácil. Muitas vezes, os projetistas não conhecem em detalhes a plataforma de implementação e, portanto, não são capazes de (ou não desejam) chegar a um projeto algorítmico passível de implementação direta. Além disso, questões relacionadas à legibilidade e Manutenção do código e reuso têm de ser levadas em conta.

Deve-se considerar, ainda, que programadores, geralmente, trabalham em equipe, necessitando integrar, testar e alterar código produzido por outros. Assim, é muito importante que haja padrões organizacionais para a fase de implementação. Esses padrões devem ser seguidos por todos os programadores e devem estabelecer, dentre outros, padrões de nomes de variáveis, formato de cabeçalhos de programas e formato de comentários, recuos e espaçamento, de modo que o código e a documentação a ele associada sejam claros para quaisquer membros da organização.

6.1.1 - Características das linguagens de programação

Nesta fase o importante é implementar o que está desenvolvido na arquitetura do software, nas suas diversas visões utilizando as melhores ferramentas, linguagem de programação mais indicada, levando em consideração diversos aspectos que veremos a seguir:

- Uniformidade - Esta característica está relacionada à consistência (coerência) da notação definida para a linguagem, a restrições arbitrárias e ao suporte a exceções sintáticas e semânticas. Um exemplo (ou contraexemplo) significativo desta característica está na linguagem Fortran, que utiliza os parênteses para expressar dois aspectos completamente diferentes na linguagem — a precedência aritmética e a delimitação de argumentos de subprogramas).

- Ambiguidade - Esta é uma característica observada pelo programador na manipulação da linguagem. Embora o compilador interprete a instrução de um único modo, o leitor (programador) poderá ter diferentes interpretações para a mesma expressão. Um caso típico de problema relacionado a esta característica é a precedência aritmética implícita. Por exemplo, a expressão X = X1/X2*X3 pode levar a duas interpretações por parte do leitor: X = (X1/X2)*X3 ou X = X1/(X2*X3).

- Concisão - Esta é uma outra característica psicológica desejável nas linguagens de programação que está relacionada à quantidade de informações orientadas ao código que deverão ser recuperadas da memória humana. Os aspectos que

influem nesta característica são a capacidade de suporte a construções estruturadas, as palavras-chave e abreviações permitidas, a diversidade com relação aos tipos de dados, a quantidade de operadores aritméticos e lógicos, entre outros.

- Linearidade - A linearidade está relacionada à manutenção do domínio funcional, ou seja, a percepção é melhorada quando uma sequência de instruções lógicas é encontrada. Programas caracterizados por grandes ramificações violam esta característica.

- Eficiência do Compilador - Esta característica contempla os requisitos de tempo de resposta e concisão (exigência de pouco espaço de memória) que podem caracterizar determinados projetos. A maioria dos compiladores de linguagens de alto nível apresenta o inconveniente de geração de código ineficiente.

- Portabilidade - Este aspecto está relacionado ao fato do código-fonte poder migrar de ambiente para ambiente, com pouca ou nenhuma alteração. Por ambiente, pode-se entender o processador, o compilador, o sistema operacional ou diferentes pacotes de software.

- Ambiente de Desenvolvimento - Um aspecto fundamental é, sem dúvida, a disponibilidade de ferramentas de desenvolvimento para a linguagem considerada. Neste caso, deve ser avaliada a disponibilidade de outras ferramentas que as de tradução, principalmente, ferramentas de auxílio à depuração, edição de código-fonte, bibliotecas de sub-rotinas para uma ampla gama de aplicações (interfaces gráficas, por exemplo).

- Manutenibilidade - Outro item de importância é a facilidade em alterar o código-fonte para efeito de manutenção. Esta etapa, em muitos casos negligenciada nos projetos de desenvolvimento de software, só poderá ser conduzida eficientemente a partir de uma completa compreensão do software. Embora a existência de documentação relativa ao desenvolvimento do software seja de fundamental importância, a clareza do código-fonte vai ser fator determinante para o sucesso das tarefas de manutenção.

6.1.2 - COMO SELECIONAR A MELHOR LINGUAGEM DE PROGRAMAÇÃO

As características apresentadas acima devem ser levadas em conta no momento da escolha de uma linguagem de programação no contexto de um projeto de software. Por outro lado, outros critérios podem auxiliar na decisão, entre eles:

- a área de aplicação para a qual o software está sendo construído;
- a complexidade computacional e algorítmica do software;
- o ambiente no qual o software vai executar;
- os requisitos de desempenho;
- a complexidade da estrutura de dados;
- o conhecimento da equipe de desenvolvimento;
- a disponibilidade de boas ferramentas de desenvolvimento.

6.2 - INTRODUÇÃO AOS TESTES DE SOFTWARE

Citado por (MAZOLA), o desenvolvimento de software utilizando as metodologias, técnicas e ferramentas da Engenharia de Software não oferece a total garantia de qualidade do produto obtido, apesar de melhorá-la significativamente. Por esta razão, uma etapa fundamental na obtenção de um alto nível de qualidade do software a ser produzido é aquela onde são realizados os procedimentos de teste, uma vez que esta é a última etapa de revisão da especificação, do projeto e da codificação.

A realização, de forma cuidadosa e criteriosa, dos procedimentos associados ao teste de um software assume uma importância cada vez maior dado o impacto sobre o funcionamento (e o custo) que este componente tem assumido nos últimos anos. Por esta razão, o esforço despendido para realizar a etapa de teste pode chegar a 40% do esforço total empregado no desenvolvimento do software (MAZZOLA).

No caso de programas que serão utilizados em sistemas críticos (aqueles sistemas dos quais dependem vidas humanas, como controle de voo e a supervisão de reatores nucleares), a atividade de teste pode custar de 3 a 5 vezes o valor gasto nas demais atividades de desenvolvimento do software. O objetivo deste capítulo é apresentar, de forma breve, os principais conceitos e técnicas relacionados ao teste de software.

6.2.1 - OBJETIVOS DOS TESTES DE SOFTWARE

Os principais objetivos do teste de software podem ser expressos, de forma mais clara, pela observação das três regras:

- A atividade de teste é o processo de executar um programa com a intenção de descobrir um erro;
- Um bom caso de teste é aquele que apresenta uma elevada probabilidade de revelar um erro ainda não descoberto;
- Um teste bem-sucedido é aquele que revela um erro ainda não descoberto.

Segundo (MYERS, 1979), as três regras expressam o objetivo primordial do teste que é o de encontrar erro, contrariando a falsa ideia de que uma atividade de teste bem-sucedida é aquela em que nenhum erro foi encontrado. A etapa de teste deve ser conduzida de modo que o maior número de erros possível seja encontrado com um menor dispêndio de tempo e esforço.

A realização, com sucesso, da etapa de teste de um software deve ter, como ponto de partida, uma atividade de projeto dos casos de teste deste software. Projetar casos de teste para um software pode ser uma atividade tão complexa quanto a de projeto do próprio software, mas ela é necessária como única forma de conduzir, de forma eficiente e eficaz, o processo de teste. Os princípios básicos do teste de qualquer produto resultante de uma tarefa de engenharia são:

• conhecida a função a ser desempenhada pelo produto, testes são executados para demonstrar que cada função é completamente operacional, este primeiro princípio deu origem a uma importante abordagem de teste, conhecida como o teste de caixa preta (black box);

• com base no conhecimento do funcionamento interno do produto, realiza-se testes para assegurar de que todas as peças destes estão completamente ajustadas e realizando a contento sua função; à abordagem originada por este segundo princípio, foi dado o nome de teste de caixa branca (white box), devido ao fato de que maior ênfase é dada ao desempenho interno do sistema (ou do produto).

6.2.2 - TIPOS DE TESTE DE SOFTWARE

A seguir veremos os dois principais tipos de testes de software e uma descrição sobre a finalidade de cada um deles, para depois aprendermos quais são as modalidades

de testes que fazem parte desta disciplina que visa agregar qualidade ao produto final de software.

6.2.3 - Teste de Caixa Preta

Quando o procedimento de teste está relacionado ao produto de software, o teste de caixa preta refere-se a todo teste que implica na verificação do funcionamento do software através de suas interfaces, o que, geralmente, permite verificar a operacionalidade de todas as suas funções. É importante observar que, no teste de caixa preta, a forma como o software está organizado internamente não tem real importância, mesmo que isto possa ter algum impacto na operação de alguma função observada em sua interface.

6.2.4 - Teste de Caixa Branca

Um teste de caixa branca está relacionado a uma verificação da estrutura interna do software de seus detalhes procedimentais. Os caminhos lógicos definidos no software devem ser exaustivamente testados, pondo à prova conjuntos bem definidos de condições ou laços. Durante o teste, o "status" do programa pode ser examinado diversas vezes para eventual comparação com condições de estado esperadas para aquela situação.

Apesar da importância do teste de caixa branca, não se deve guardar a falsa ideia de que a realização de testes de caixa branca num produto de software vai oferecer a garantia de 100% de correção deste ao seu final. Isto porque, mesmo no caso de programas de pequeno e médio porte, a diversidade de caminhos lógicos pode atingir um número bastante elevado, representando um grande obstáculo para o sucesso completo desta atividade.

6.3 - Modalidades de Testes de Software

6.3.1 - Testes Estáticos

Os testes estáticos são aqueles realizados sobre o código-fonte do software, utilizando como técnica básica a inspeção visual. Este tipo de teste é de simples implementação, uma vez que não há necessidade de execução do programa para obter-se resultados. Eles podem ser utilizados utilizando a técnica de leitura cruzada, onde um leitor é atribuído para avaliar o trabalho de cada programador do software. Uma outra forma de realizar o teste é por inspeção, onde uma equipe designada analisa o código à luz de um questionário concebido a um checklist.

Os responsáveis pela realização do teste não realizam nenhum tipo de correção no código, limitando-se a assinalar os erros encontrados. Eventualmente, o teste estático

pode ser automatizado com o auxílio de ferramentas de análise estática, que podem ser simples geradores de referências cruzadas ou, no caso de ferramentas mais sofisticadas, serem dotadas de funções de análise do fluxo de dados do programa.

6.3.2 - TESTES DINÂMICOS

Enquanto a análise estática está focada no código da aplicação, a análise dinâmica se comporta como um hacker tentando encontrar uma vulnerabilidade em tempo de execução da aplicação. Para encontrar o mesmo problema do tipo SQL Injection sem ter que varrer todo o código, uma análise dinâmica tem uma abordagem menos teórica: o analista de segurança executa a aplicação e procura por campos em formulários ou outro tipo de entrada de dados, manipula os dados que são passados para a aplicação e de acordo com o resultado verifica na prática a existência ou não de uma vulnerabilidade.

A análise dinâmica tem a vantagem de explorar outras vulnerabilidades na aplicação que não podem ser detectadas apenas no código, tais como uma configuração no servidor web ou uma falha no middleware utilizado pela aplicação (como o banco de dados ou o application server).

6.3.3 - TESTES FUNCIONAIS

Uma vez que testes exaustivos não são viáveis, características do domínio de entrada são examinadas para que se tente descobrir formas de derivar um conjunto de dados de teste representativo que consiga exercitar completamente a estrutura do sistema. Os dados de teste precisam ser derivados de uma análise dos requisitos funcionais e incluir elementos representativos de todas as variáveis do domínio. Este conjunto deve incluir tantos dados de entrada válidos quanto inválidos. Geralmente, os dados no conjunto de dados de teste podem ser classificados em três classes: de fronteira, não de fronteira e especiais. Exemplificando: Se X for um número e se tivermos X tal que, $a < X < b$, então X deve ser testados para os valores $a + 1$ e $b - 1$ da classe válida, para os valores a e b da classe inválida e para valores especiais tais como zero, vazio, ou caracteres alfabéticos. São considerados métodos de testes funcionais ou de caixa preta: o particionamento de equivalência, a análise de valor limite, a técnica de grafo de causa-efeito e os testes de comparação.

6.3.4 - TESTES DE UNIDADE

Concentra-se no esforço de verificação da menor unidade de projeto de software: o módulo. Através do uso da descrição do projeto detalhado como guia, caminhos de controle importantes são testados para descobrir erros dentro das fronteiras do módulo. Este teste baseia-se sempre na caixa branca, e esse passo pode ser realizado em paralelo para múltiplos módulos. Nos testes de unidade são verificados: a interface com o módulo, a estrutura de dados local, as condições de limite, todos os caminhos independentes através da estrutura de controle e todos os caminhos de tratamento de erros.

6.3.5 - TESTES DE INTEGRAÇÃO

É o teste de software que os módulos são combinados e testados em grupo. Ela sucede o teste de unidade, em que os módulos são testados individualmente, e antecede o teste de sistema, em que o sistema completo (integrado) é testado num ambiente que simula o ambiente de produção. O teste de integração é alimentado pelos módulos previamente testados individualmente pelo teste de unidade, agrupando-os assim em componentes, como estipulado no plano de teste, e resulta num sistema integrado e preparado para o teste de sistema.

6.3.6 - TESTES DE VALIDAÇÃO

O teste de validação pode ser considerado bem-sucedido quando o software funciona da maneira esperada pelo cliente. Ou seja, verifica-se se o produto certo foi construído, seguindo a especificação de requisitos do software. A validação do software, na fase de testes, é realizada por meio de uma série de testes de caixa preta que demonstram a conformidade com os requisitos.

6.3.7 - TESTES ALFA E BETA

São os testes de aceitação, feitos pelo usuário, que dificilmente opera o sistema da forma prevista, e visam descobrir erros cumulativos que poderiam deteriorar o sistema no decorrer do tempo. O teste alfa é executado por um cliente nas instalações do desenvolvedor, sendo acompanhado pelo desenvolvedor, que registra os problemas encontrados no uso. O ambiente é controlado. Já o teste beta é realizado em uma ou mais instalações do cliente pelo usuário final do software. Geralmente o desenvolvedor não está presente. Assim, o teste beta é uma aplicação real do software, sem que haja controle por parte do desenvolvedor. Os problemas são registrados pelo usuário e repassados regularmente ao desenvolvedor, que corrige o software antes de lançar o produto para venda.

6.3.8 - TESTE DE SEGURANÇA

O teste de segurança tenta verificar se todos os mecanismos de proteção embutidos em um sistema o protegerão de acessos indevidos. Todas as formas de ataque devem ser simuladas. A finalidade é dificultar o acesso indevido de forma que seja mais interessante e barato obter a informação de forma correta e legal.

6.3.9 - TESTES DE ESTRESSE

O teste de estresse é feito para confrontar o sistema com situações anormais. O teste de estresse executa o sistema de forma que exige recursos em quantidade, frequência ou volume anormais.

6.3.10 - TESTE DE DESEMPENHO

O teste de desempenho é idealizado para testar o desempenho do *software* quando executado dentro do contexto de um sistema integrado. Algumas vezes os testes de desempenho são combinados com os de estresse e podem ser executados durante todo o processo de desenvolvimento.

6.4 - EXERCÍCIOS

1) Dentre as diversas características que as linguagens de programação possuem, qual é o aspecto que faz com que o programador possa ter diferentes interpretações para a mesma expressão?.

2) Qual é o aspecto que está relacionado ao fato do código-fonte poder migrar de um ambiente para outro, com pouca ou nenhuma alteração?.

3) Cite quatro critérios que possam auxiliar na escolha da linguagem de programação mais adequada para seu projeto?.

4) Qual a diferença de testes caixa preta e teste caixa branca?.

5) Qual é o tipo de teste que se concentra no esforço de verificação da menor unidade de projeto de software?.

6) Qual é o tipo de teste onde os módulos do software são integrados e testados?.

REFERÊNCIAS DO CAPÍTULO

PRESSMAN, R.S., Engenharia de Software, McGraw-Hill, 6ª edição, 2006.

SOMMERVILLE, I., Engenharia de Software, 8ª Edição. São Paulo: Pearson – Addison Wesley, 2007.

MAZZOLA, B. V., Apostila de Engenharia de Software

MYERS, GLENFORD J. The Art of Software Testing,. Wiley, New York, 1979.

CONTEÚDOS COMPLEMENTARES

RUP (RATIONAL UNIFIED PROCESS)

Acesse: http://www.wthreex.com/rup/

UML (UNIFIED MODELING LANGUAGE)

Acesse: http://www4.serpro.gov.br/inclusao/conteudos-educacionais-livres/uml

Acesse: http://www.uml.org/

SCRUM (METODOLOGIA ÁGIL DE PROJETOS)

Acesse: http://www.mindmaster.com.br/scrum/