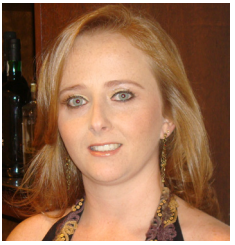
ENGENHARIA DE SOFTWARE

CURSOS DE GRADUAÇÃO – EAD

Engenharia de Software – *Prof. Drª. Ana Paula do Carmo Marcheti Ferraz e Prof. Ms. Jaciara Silva Carosia*

Meu nome é **Ana Paula do Carmo Marcheti Ferraz**. Sou doutora em Engenharia de Produção pela Escola de Engenharia de São Carlos (USP) e mestre pelo mesmo programa. Fiz especialização em Administração de Empresas no Centro Universitário Moura Lacerda (Ribeirão Preto – SP), em Psicoeducação na Faculdade de Filosofia, Ciências e Letras FFCL-USP (Ribeirão Preto – SP) e em Análise de Sistemas na Unaerp (Ribeirão Preto – SP). Minha formação inicial foi em Ciências da Computação pela Unesp (Bauru – SP). Atualmente, atuo como avaliadora associada ao Ministério da Educação para cursos de Graduação e Pós-graduação das modalidades a distância e presencial. Sou professora de Informática Aplicada, Linguagem de Programação e Projeto de Conclusão de Curso nos cursos presenciais, e tutora na modalidade de EAD das disciplinas Linguagem de Programação Estruturada I e II.



Meu nome é **Jaciara Silva Carosia**. Sou mestre em Computação Aplicada pelo Instituto Nacional de Pesquisas Espaciais (INPE), com pesquisa na área de Engenharia de *Software*. Atualmente, atuo como professora na FATEC Mococa nos cursos de: Tecnologia em Informática para Banco de Dados e Redes de Computadores e Tecnologia em Informática para Gestão de Negócios. Além disso, sou professora e coordenadora do curso de Ciência da Computação na Unip, *campus* de São José do Rio Pardo.

Profª. Drª. Ana Paula do Carmo Marcheti Ferraz
Profª. Ms. Jaciara Silva Carosia

ENGENHARIA DE SOFTWARE

Plano de Ensino
Caderno de Referência de Conteúdo
Caderno de Atividades e Interatividades

Claretiano
CENTRO UNIVERSITÁRIO - CEUCLAR

Trabalho realizado pelo Centro Universitário Claretiano de Batatais (SP)

Cursos: Graduação
Disciplina: Engenharia de Software
Versão: jul./2011

Centro Universitário Claretiano
Rua Dom Bosco, 466 - Bairro: Castelo – Batatais SP – CEP 14.300-000
cead@claretiano.edu.br
Fone: (16) 3660-1777 – Fax: (16) 3660-1780 – 0800 941 0006
www.claretiano.edu.br

SUMÁRIO

UNIDADE 3 – PLANEJAMENTO E GERENCIAMENTO DE PROJETOS

UNIDADE 4 – ANÁLISE DE REQUISITOS DE SOFTWARE

UNIDADE 5 – GERENCIAMENTO DE CONFIGURAÇÕES

UNIDADE 6 – GARANTIA DE QUALIDADE DE SOFTWARE

UNIDADE 7 – TESTE DE SOFTWARE

UNIDADE 8 – MANUTENÇÃO DE SOFTWARE

UNIDADE 9 – FERRAMENTAS CASE

UNIDADE 10 – QUALIDADE DE SOFTWARE

Plano de Ensino

PE

1. INTRODUÇÃO

O aprendizado da informática nas instituições educacionais é embasado em técnicas, modelos e metodologias fundamentados em teorias e normas já experimentadas. Desta forma, quando você aplicar o conhecimento adquirido, você não estará testando uma teoria, mas, sim, adaptando-a a sua realidade prática, isto diminui o risco do insucesso.

Os formalismos intrínsecos ao estudo da Engenharia de *Software* podem, em um primeiro momento, dar-lhe a impressão de serem burocráticos. Porém, você perceberá que tais formalismos lhe oferecerão os instrumentos necessários para que tenha um posicionamento crítico em relação aos processos utilizados no desenvolvimento de *software*.

A disciplina de *Engenharia de Software* é teórica, mas lhe fornecerá os instrumentos necessários para a efetiva aplicação em projetos reais. Esta disciplina vai ajudá-lo a compreender as técni-

cas de gerenciamento e desenvolvimento de projetos de *software*. Uma vez compreendidos, você poderá optar por utilizar as técnicas que mais se adequarem a sua realidade profissional.

O desenvolvimento de *software* baseado em uma metodologia efetiva o levará à construção de *softwares* mais adequados à necessidade dos usuários, produzidos através de um processo que possa ser gerenciado e melhorado a cada novo projeto de *software*.

A busca pela produtividade na indústria do *software* só se torna meta atingível quando os desenvolvedores se conscientizam de que é necessário o enfoque na qualidade dos métodos e ferramentas e na capacitação dos recursos humanos dedicados à construção dos *softwares*. Assim, é importante que a qualidade seja palavra de ordem na produção do *software*.

Há muito tempo a qualificação dos processos produtivos é utilizada como instrumento de inovação na indústria de manufatura em geral, e, vale ressaltar, que tem dado ótimos resultados. É importante que a indústria do *software* também utilize a qualidade como instrumento de inovação e competitividade. O desenvolvedor de *software* precisa superar o amadorismo dos processos que levam à construção de *software* com características que não agradam ao cliente e que extrapolam em muito os prazos e custos predefinidos no início do projeto.

A Engenharia de *Software* está em constante transformação, trazendo para a comunidade de desenvolvedores métodos cada vez mais adequados a construção dos softwares.

Durante esta disciplina você conhecerá os conceitos fundamentais relacionados à produção de *software*. Você conhecerá os principais paradigmas da Engenharia de *Software*. Serão abordadas atividades ligadas ao desenvolvimento de *software*, tais como: planejamento e gerenciamento de projetos, levantamento de requisitos, teste, manutenção, gerenciamento de configuração e garantia de qualidade de *software*. Além disso, serão apresentadas as ferramentas de apoio ao desenvolvedor de *software*, visando ao

aumento da produtividade da indústria de *software*. Finalmente, você conhecerá as características relacionadas a um *software* de qualidade e verificará a importância do processo para a qualidade do produto de *software*.

No decorrer desta disciplina, você entrará em contato com as variáveis envolvidas no ambiente de desenvolvimento de *software*, tais como: a diversidade de aplicações, as diferentes necessidades dos clientes, as múltiplas características da equipe de desenvolvedores. Ao final desta disciplina, você terá adquirido uma visão mais realística do desenvolvimento de *software*.

A Engenharia de *Software* não lhe fornecerá uma receita de bolo de como desenvolver *software* com qualidade, mas sim a bagagem necessária para que você saiba adaptar as propostas da disciplina à sua realidade profissional.

Espero que você aproveite bastante os fundamentos de Engenharia de *Software* e que eles possam contribuir para seu sucesso profissional.

Bons estudos!

2. DADOS GERAIS DA DISCIPLINA

Ementa

Introdução à Engenharia de *Software*. Características do *software*. Crise do *Software*. Ciclo de vida de desenvolvimento de *software*. Gerenciamento de projetos: métricas, estimativas, análise de riscos, planejamento e acompanhamento do projeto. Análise e projeto de *software*. Teste de *software*. Manutenção de *software*. Ferramentas CASE. Qualidade de *software*.

Objetivo geral

Os alunos de *Engenharia de Software* dos cursos de graduação, na modalidade EAD do Claretiano, dado o Sistema Geren-

ciador de Aprendizagem e suas ferramentas, obterão um conhecimento colaborativo e cooperativo e a habilidade de trabalhar com equipes virtuais, competências exigidas pelo mercado de trabalho. Considerando os temas abordados, poderão aprofundar o conhecimento sobre a questão de avaliação de aprendizagem, objetivos educacionais, definição de conteúdos etc.

Com esse intuito, os alunos contarão com recursos técnico-pedagógicos facilitadores de aprendizagem, como material didático mediacional, bibliotecas físicas e virtuais, ambiente virtual e acompanhamento do tutor complementado por debates no Fórum e na Lista.

Ao final desta disciplina, de acordo com a proposta orientada pelo tutor, os alunos, além de terem uma visão abrangente da Engenharia de *Software* e da importância das técnicas existentes para análise, modelagem, definição de ciclo de vida de sistemas, documentação, manutenção, testes e padronização de *softwares*, garantia de qualidade e gestão da configuração, realizarão uma atividade sobre o tema estudado nesta disciplina. Para esse fim, levarão em consideração as ideias debatidas na Sala de Aula Virtual, por meio de suas ferramentas, bem como o que produziram durante o estudo.

Competências, habilidades e atitudes

Ao final deste estudo, o aluno dos cursos de graduação contarão com uma sólida base teórica para fundamentar, criticamente, sua prática profissional. Além disso, o aluno irá adquirir as habilidades necessárias não somente para cumprir seu papel de profissional nesta área do saber, mas também para agir com ética e com responsabilidade social.

Modalidade

( ) Presencial (X) A distância

Duração e carga horária

A carga horária da disciplina *Engenharia de Software* é de **30 horas**. O conteúdo programático para o estudo das **dez unidades** que a compõe está desenvolvido no *Caderno de referência de conteúdo*, anexo a este *Guia de disciplina*, e os exercícios propostos constam do *Caderno de atividades e interatividades* (CAI).

É importante que você releia, no *Guia Acadêmico* do seu curso, as informações referentes à **Metodologia** e à **Forma de Avaliação** da disciplina *Engenharia de Software*. A síntese dessas informações consta no "cronograma" na Sala de Aula Virtual – SAV.

3. CONSIDERAÇÕES GERAIS

Nesta disciplina, abordaremos vários assuntos relacionados ao desenvolvimento, ao gerenciamento e à manutenção de desenvolvimento de projetos de *software*.

É importante que você tenha uma visão clara dos objetivos que desejamos atingir com o estudo da presente disciplina, porque ela será fundamental para a sua atuação profissional. Esperamos que esta seja uma disciplina que o encante. Lembre-se de sempre pesquisar, pois os assuntos abordados podem ser considerados dinâmicos em virtude do panorama tecnológico vigente.

Participar é importante. Use o seu *login* e a sua senha para entrar na Sala de Aula Virtual e utilize as várias ferramentas colocadas à sua disposição.

O segredo do sucesso em um curso na modalidade Educação a Distância é PARTICIPAR, ou seja, INTERAGIR, procurando sempre cooperar e colaborar com seus colegas e tutores.

Durante o estudo de cada unidade, procure sempre interagir com seus colegas. Apresente suas dúvidas e discuta sobre os temas apresentados. A Engenharia de Software oferece vários te-

mas para discussão. Não adote uma posição passiva diante destes temas. Em caso de dúvidas, consulte o seu tutor. O importante é não acumulá-las.

Não limite seu aprendizado somente a este material, pesquise sempre, procure se atualizar constantemente. Na área de software as mudanças ocorrem de forma muito rápida e o bom profissional é aquele que consegue manter-se atualizado.

A cada unidade, faça as questões autoavaliativas e teste seus conhecimentos.

4. BIBLIOGRAFIA BÁSICA

PFLEEGER, S.L. *Engenharia de software*: teoria e prática. São Paulo: Prentice Hall, 2004.

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

_____. *Engenharia de software*. São Paulo: Makron Books, 2006.

SOMMERVILLE, I. *Engenharia de software*. 6. ed. São Paulo: Pearson Addison Wesley, 2005.

5. BIBLIOGRAFIA COMPLEMENTAR

CROSBY, P. B. *Qualidade é Investimento*. Rio de Janeiro: José Olympio, 1991.

FERNANDES, A. A. *Gerência Efetiva de Software através de Métricas*: garantindo a qualidade do projeto, processo e produto. 2. ed. São Paulo: Atlas, 1995.

JURAN, J. M. *Juran planejando para a qualidade*. São Paulo: Pioneira, 1990.

MATOS, A. V. *UML* - Prático e descomplicado. São Paulo: Érica, 2002.

MOLINARI, L. *Testes de Software*: Produzindo Sistemas Melhores e Mais Confiáveis. 3.ed. São Paulo: Érica, 2006.

O'BRIEN, J. A. *Sistemas de Informação*: as decisões gerenciais na era da Internet. São Paulo: Saraiva, 2004.

PAULA, F. W. P. *Engenharia de software*: fundamentos, métodos e padrões. 2. ed. Rio de janeiro: LTC, 2001.

POMPILHO, J. *Analise essencial*: Guia prático de analise de sistemas. Rio de Janeiro: Ciência Moderna, 2002.

TONSIG, S. L. *Engenharia de software*. São Paulo: Futura, 2003.

YOURDON, E. *Análise estruturada moderna*. 3. ed. Rio de Janeiro: Campus, 1990.

Caderno de Referência de Conteúdo

CRC

1. INTRODUÇÃO

Nos primórdios da evolução da informática, o *hardware* era o elemento mais caro e, portanto, considerado o mais importante componente do sistema computacional. Naquela época, o *software* era visto como um coadjuvante, enquanto ao *hardware* era atribuído o mérito de ator principal.

No entanto, com o passar dos anos, motivado pelos avanços da eletrônica e consequentemente com o barateamento do *hardware*, o *software* foi ganhando lugar de destaque, tornando-se o elemento foco do sistema computacional. Com a expansão do uso dos sistemas computacionais ficou evidente que o *software* é quem capacita o *hardware* às mais diversas aplicações, isto é, o *software* é o responsável pela versatilidade e flexibilidade do sistema baseado em computador.

Os sistemas computacionais tornaram-se fundamentais para a execução de muitas tarefas, sendo praticamente impensável a

execução de muitas atividades sem o apoio destes sistemas. O surgimento e crescente utilização do computador e, mais tarde, da internet, gerou grandes mudanças no panorama econômico, político, social e cultural. Com o aumento extraordinário dos produtos e serviços oferecidos por sistemas baseados em computador, estes passaram a desempenhar um papel essencial e crítico na sociedade, aumentando a preocupação dos desenvolvedores em entender e melhorar a qualidade dos sistemas produzidos.

Assim, considerando o destaque que o *software* alcançou no contexto dos sistemas computacionais e a crescente demanda por novos e complexos sistemas, o processo de desenvolvimento de *software* tornou-se uma preocupação.

Quando o *software* passou a ser produzido em larga escala, começaram a surgir problemas envolvidos ao desenvolvimento desses *softwares*, tais como: dificuldade em cumprir prazos e custos, entrega de produtos com a qualidade abaixo da desejada pelo cliente, dificuldade de manutenção nos *softwares* existentes, entre outros.

Tendo em vista a importância do *software* na vida humana nos mais diversos aspectos, surgiu a Engenharia de *Software* para apoiar sistemática, racional e economicamente a construção do *software*. A Engenharia de *Software* é uma disciplina que enfoca todos os aspectos da produção de *software*, isto é, desde as fases iniciais de levantamento dos requisitos do sistema até a sua implantação e manutenção. A Engenharia de *Software* visa organizar o processo de desenvolvimento de *software*.

Ao longo dos anos, para minimizar os problemas da indústria de *software*, a Engenharia de *Software* tem proposto vários métodos para aumentar a produtividade, através da diminuição do custo de produção, do prazo de entrega e do aumento da qualidade do produto.

No entanto, apesar do esforço da Engenharia de *Software* em sugerir métodos para produção de *software* com qualidade,

ainda hoje, muita coisa precisa ser colocada em prática. Os clientes da indústria de *software* ainda têm tido problemas na hora de contratar uma empresa de desenvolvimento de *software* que realmente lhe forneça um melhor serviço, a um custo não muito alto e, o mais importante, com qualidade e agilidade na entrega. O que ainda tem ocorrido é que muitas pessoas adquirem um determinado sistema, iniciando assim um difícil e exaustivo processo de implantação e correção de erros.

Para mudar este quadro, é importante o compromisso do desenvolvedor de *software* com o uso de métodos que melhorem seu processo produtivo e minimizem a insatisfação do cliente no que diz respeito à qualidade, prazo e custo do *software*. Parte destes transtornos poderia ser evitada se a empresa desenvolvedora de *software* se preocupasse com a inserção de características de qualidade no produto durante o seu desenvolvimento. Com este intuito, surgiram vários modelos e normas que auxiliam a melhoria do processo de desenvolvimento de *software*.

No decorrer do estudo desta disciplina você conhecerá os principais modelos e técnicas da Engenharia de *Software*. Os conceitos apresentados neste material serão de grande valia para a difusão dos fundamentos da Engenharia de *Software* e para melhorar a qualidade do seu trabalho enquanto parte integrante da comunidade de desenvolvedores de *software*. Os conceitos aqui apresentados lhe darão a base necessária para que você possa executar seu trabalho de desenvolvedor de *software* de forma clara e rápida.

O ambiente que envolve o desenvolvimento de *software* é composto de inúmeras variáveis, tais como: diversidade de aplicações de *software*, diferenças nas necessidades dos clientes, características da equipe de desenvolvedores, enfim, são muitos e diversos os componentes envolvidos neste contexto. Assim, você conhecerá os principais conceitos da Engenharia de *Software*. Porém, por não ser uma ciência tão exata quanto as outras engenha-

rias, você deverá analisar as suas reais necessidades para escolher e adaptar as técnicas aqui apresentadas.

Por ser uma disciplina nova, você perceberá que ainda existem algumas controvérsias sobre algumas técnicas e métodos. No entanto, você, enquanto profissional de *software*, poderá contribuir com a divulgação e com o aperfeiçoamento dos fundamentos aqui propostos.

A Engenharia de *Software* aponta o caminho e as ferramentas que poderão facilitar sua caminhada, mas o esforço da caminhada e o sucesso na chegada dependerão de você.

Bons estudos!

2. ORIENTAÇÕES SOBRE O ESTUDO DA DISCIPLINA

Abordagem geral da disciplina

Neste tópico, apresenta-se uma visão geral do que será estudado nesta disciplina. Aqui, você entrará em contato com os assuntos principais deste conteúdo de forma breve e geral e terá a oportunidade de aprofundar essas questões no estudo de cada unidade. No entanto, essa Abordagem Geral visa fornecer-lhe o conhecimento básico necessário a partir do qual você possa construir um referencial teórico com base sólida – científica e cultural – para que, no futuro exercício de sua profissão, você a exerça com competência cognitiva, ética e responsabilidade social. Vamos começar nossa aventura pela apresentação das ideias e dos princípios básicos que fundamentam esta disciplina.

Na Unidade 1, são mostradas as dificuldades e características envolvidas no desenvolvimento de *software*. Você compreenderá os reais objetivos da Engenharia de *Software* e os fatores que contribuíram para o seu surgimento. Saberá quais os principais problemas enfrentados pela indústria do *software* no passado e também nos dias atuais.

Na Unidade 2 você compreenderá o que é um processo de *software* e qual a sua relação com a qualidade do produto. São apresentados os principais paradigmas de desenvolvimento de *software*, mostrando a evolução destes modelos com o passar do tempo.

Na Unidade 3, você aprenderá a importância do planejamento e do gerenciamento para que um projeto de *software* seja bem-sucedido. Conhecerá tarefas e o esforço necessário para a criação de um plano realístico e administrável.

A Unidade 4, mostra a importância das atividades de levantamento e gerenciamento de requisitos. Você compreenderá quais atividades devem ser executadas para a melhor análise das necessidades do cliente. Além disso, conhecerá a problemática da comunicação entre as pessoas envolvidas nestas atividades.

Na Unidade 5, verá que a mudança de requisitos é uma realidade no desenvolvimento de *software*. Para conviver com tamanha instabilidade, você compreenderá a importância do Gerenciamento de Configurações para administrar a incorporação das mudanças no projeto de *software*.

Na Unidade 6, irá compreender a importância de focar na qualidade do *software* durante todo o seu processo de desenvolvimento. Entenderá, também, como a Garantia de Qualidade atua para que o *software* desenvolvido atenda às especificações definidas para ele.

Na Unidade 7, é apresentada a importância dos testes de *software*, tarefa esta muitas vezes ignorada pelos desenvolvedores, e as principais técnicas e estratégias de teste.

A Unidade 8 descreve sobre a atividade de manutenção. Você compreenderá a importância de se produzir um *software* que seja fácil de ser alterado depois de entregue ao cliente. Além disso, verá que o trabalho do desenvolvedor não termina após a implantação do sistema.

A Unidade 9 aborda as ferramentas que automatizam o trabalho do desenvolvedor de *software* durante as diversas fases do processo produtivo e até que ponto estas ferramentas interferem na produtividade da indústria do *software*.

Finalmente, a Unidade 10 tem como foco as características de qualidade do produto e do processo de *software*, mostrando características e fatores que afetam a qualidade do *software*. Além disso, você conhecerá as principais normas que visam à melhoria da qualidade de *software*.

Glossário de Conceitos

O Glossário de Conceitos permite a você uma consulta rápida e precisa das definições conceituais, possibilitando-lhe um bom domínio dos termos técnico-científicos utilizados na área de conhecimento dos temas tratados na disciplina *Engenharia de Software*. Veja, a seguir, a definição dos principais conceitos desta disciplina:

1) **Cliente**: "é a empresa, organização ou pessoa que está pagando para o sistema de software ser desenvolvido" (PFLEEGER, 2004, p. 11).
2) **Configuração**: "é um conjunto de componentes do sistema fornecidos a um cliente específico" (PFLEEGER, 2004, p. 318). Segundo Pressman (1996, p. 171), pode também ser definida como "um sistema funcionando é somente uma parte de uma configuração de software", ou ainda como:

   > [...] a saída do processo de software é a informação, que pode ser dividida em três amplas categorias: programas de computador, produtos de trabalho que descrevem programas de computador e dados. Os itens que compreendem toda a informação produzida como parte do processo de software são chamados coletivamente de configuração do software (2006, p. 600).

3) **Desenvolvedor**: "é a empresa, organização ou pessoa que está construindo o sistema de software para o cliente" (PFLEEGER, 2004, p. 11).
4) **Engenharia de Software**: "aplicação de uma abordagem sistemática, disciplinada e quantificável, para o desenvolvimen-

to, operação e manutenção do software; isto é, a aplicação da engenharia ao software" (PRESSMAN, 2006, p. 17).

5) **Garantia de qualidade de software**: é "o estabelecimento de uma estrutura de procedimentos e de padrões organizacionais, que conduzam ao software de alta qualidade" (SOMMERVILLE, 2005, p. 459). Segundo Pressman (2006, p. 579) a garantia de qualidade de *software* "consiste em um conjunto de funções para auditar e relatar que avalia a efetividade e completeza das atividades de controle de qualidade".

6) **Gerenciamento de configuração**: "identifica, controle, audita e relata modificações que invariavelmente ocorrem enquanto o software está sendo desenvolvido e depois de ele ter sido entregue ao cliente" (PRESSMAN, 2006, p. 621).

7) **Gerenciamento de Requisitos**: Segundo Pressman (2006, p. 121):

> [...] é um conjunto de atividades que ajudam a equipe de projeto a identificar, controlar e rastrear requisitos e modificações de requisitos em qualquer época, à medida que o projeto prossegue.

8) **Ferramentas de Engenharia de Software**: proporcionam apoio automático ou semiautomático aos métodos (PRESSMAN, 1996, p.32).

9) **Manutenção de software**: "qualquer trabalho efetuado para modificar o sistema, depois que ele estiver em operação" (PFLEEGER, 2004, p. 379).

10) **Métodos de engenharia de software**: Segundo Pressman (1996, p. 31):

> [...] proporcionam detalhes de 'como fazer' para construir o software. Os métodos envolvem um grande número de tarefas que incluem: planejamento e estimativa de projeto, análise de requisitos de software e de sistema, projeto da estrutura de dados, arquitetura de programa e algoritmo de processamento, codificação, teste e manutenção.

11) **Padrões de processo**: Segundo Sommerville (2005, p. 461):

> [...] são os padrões que definem os processos a serem seguidos durante o desenvolvimento de software. Eles podem incluir definição de especificação, processos de projeto e validação, e uma descrição dos documentos que devem ser gerados no curso desses processos.

12) **Paradigma da Engenharia de Software**: "representa a abordagem ou filosofia em particular para a construção de software" (PFLEEGER, 2004, p. 3); Segundo Pressman (1996, p. 33):

> [...] a engenharia de software compreende um conjunto de métodos, ferramentas e procedimentos [...] estas etapa são muitas vezes citadas como paradigmas de engenharia de software.

13) **Procedimentos de Engenharia de Software**: constituem o elo "que mantém juntos os métodos e as ferramentas e possibilita o desenvolvimento racional e oportuno do software de computador" (PRESSMAN, 1996, p. 32).

14) **Processo de Software**: "um arcabouço para as tarefas que são necessárias para construir softwares de alta qualidade" (PRESSMAN, 2006, p. 16);

15) **Requisito**: "é uma característica do sistema ou a descrição de algo que o sistema é capaz de realizar, para atingir os seus objetivos" (PFLEEGER, 2004, p. 111); Segundo Sommerville (2005, p. 82):

> [...] em alguns casos um requisito é visto como uma declaração abstrata, de alto nível, de uma função que o sistema deve fornecer ou de uma restrição do sistema. Em outro extremo, ele é uma definição detalhada, matematicamente formal, de uma função do sistema.

16) **Sistema baseado em computador**: é "um conjunto ou disposição de elementos que é organizado para executar certo método, procedimento ou controle ao processar informações" (PRESSMAN, 1996, p. 179).

17) ***Software***: "o software é a parte lógica que dota o equipamento físico de capacidade para realizar todo tipo de trabalho" (ALCADE, 2001, p. 5); conforme Pressman (1996, p. 179) são "programas de computador, estruturas de dados e documentação correlata que servem para efetivar o método, processo ou controle lógico necessário".

18) **Usuário**: "é a pessoa, ou pessoas, que realmente utilizarão o sistema; aqueles que se sentarão em frente ao terminal para inserir dados ou ler resultados" (PFLEEGER, 2004, p.11).

Esquema dos Conceitos-chave

Para que você tenha uma visão geral dos conceitos mais importantes deste estudo, apresentamos, a seguir (Figura 1), um Esquema dos Conceitos-chave da disciplina. O mais aconselhável é que você mesmo faça o seu esquema de conceitos-chave ou até mesmo o seu mapa mental. Esse exercício é uma forma de você construir o seu conhecimento, ressignificando as informações a partir de suas próprias percepções.

É importante ressaltar que o propósito desse Esquema dos Conceitos-chave é representar, de maneira gráfica, as relações entre os conceitos por meio de palavras-chave, partindo dos mais complexos para os mais simples. Esse recurso pode auxiliar você na ordenação e na sequenciação hierarquizada dos conteúdos de ensino.

Com base na teoria de aprendizagem significativa, entende-se que, por meio da organização das ideias e dos princípios em esquemas e mapas mentais, o indivíduo pode construir o seu conhecimento de maneira mais produtiva e obter, assim, ganhos pedagógicos significativos no seu processo de ensino e aprendizagem.

Aplicado a diversas áreas do ensino e da aprendizagem escolar (tais como planejamentos de currículo, sistemas e pesquisas em Educação), o Esquema dos Conceitos-chave baseia-se, ainda, na ideia fundamental da Psicologia Cognitiva de Ausubel, que estabelece que a aprendizagem ocorre pela assimilação de novos conceitos e de proposições na estrutura cognitiva do aluno. Assim, novas ideias e informações são aprendidas, uma vez que existem pontos de ancoragem.

Tem-se de destacar que "aprendizagem" não significa, apenas, realizar acréscimos na estrutura cognitiva do aluno; é preciso, sobretudo, estabelecer modificações para que ela se configure como uma aprendizagem significativa. Para isso, é importante considerar as entradas de conhecimento e organizar bem os materiais de aprendizagem. Além disso, as novas ideias e os novos conceitos devem ser potencialmente significativos para o aluno, uma vez que, ao fixar esses conceitos nas suas já existentes estruturas cognitivas, outros serão também relembrados.

Nessa perspectiva, partindo-se do pressuposto de que é você o principal agente da construção do próprio conhecimento, por meio de sua predisposição afetiva e de suas motivações internas e externas, o Esquema dos Conceitos-chave tem por objetivo tornar significativa a sua aprendizagem, transformando o seu conhecimento sistematizado em conteúdo curricular, ou seja, estabelecendo uma relação entre aquilo que você acabou de conhecer com o que já fazia parte do seu conhecimento de mundo (adaptado do *site* disponível em: <http://penta2.ufrgs.br/edutools/mapasconceituais/utilizamapasconceituais.html>. Acesso em: 11 mar. 2010).

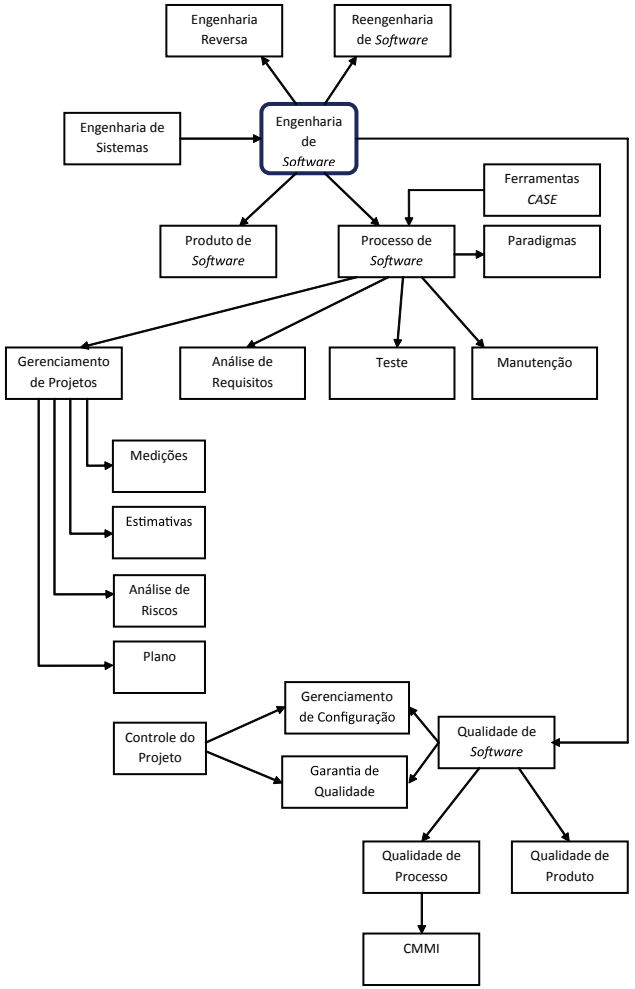


Figura 1 *Esquema dos Conceitos-chave da disciplina Engenharia de Software*.

Como você pode observar, esse Esquema dá a você, como dissemos anteriormente, uma visão geral dos conceitos mais importantes deste estudo. Ao segui-lo, você poderá transitar entre um e outro conceito desta disciplina e descobrir o caminho para construir o seu processo de ensino-aprendizagem.

O Esquema dos Conceitos-chave é mais um dos recursos de aprendizagem que vem se somar àqueles disponíveis no ambiente virtual, por meio de suas ferramentas interativas, bem como àqueles relacionados às atividades didático-pedagógicas realizadas presencialmente no polo. Lembre-se de que você, aluno EAD, deve valer-se da sua autonomia na construção de seu próprio conhecimento.

Questões Autoavaliativas

No final de cada unidade, você encontrará algumas questões autoavaliativas sobre os conteúdos ali tratados, as quais podem ser de **múltipla escolha**, **abertas objetivas** ou **abertas dissertativas**.

Responder, discutir e comentar essas questões, bem como relacioná-las com sua prática profissional pode ser uma forma de você avaliar o seu conhecimento. Assim, mediante a resolução de questões pertinentes ao assunto tratado, você estará se preparando para a avaliação final, que será dissertativa. Além disso, essa é uma maneira privilegiada de você testar seus conhecimentos e adquirir uma formação sólida para a sua prática profissional.

Você encontrará, ainda, no final de cada unidade, um gabarito, que lhe permitirá conferir as suas respostas sobre as questões autoavaliativas de múltipla escolha.

As **questões de múltipla escolha** são as que têm como resposta apenas uma alternativa correta. Por sua vez, entendem-se por **questões abertas objetivas** as que se referem aos conteúdos matemáticos ou àqueles que exigem uma resposta determinada, inalterada. Já as **questões abertas dissertativas** obtêm por resposta uma interpretação pessoal sobre o tema tratado; por isso, normalmente, não há nada relacionado a elas no item Gabarito.

Você pode comentar suas respostas com o seu tutor ou com seus colegas de turma.

Bibliografia Básica

É fundamental que você use a Bibliografia Básica em seus estudos, mas não se prenda só a ela. Consulte, também, as bibliografias apresentadas no *Plano de Ensino* e no item *Orientações para o estudo da unidade*.

Figuras (ilustrações, quadros...)

Neste material instrucional, as ilustrações fazem parte integrante dos conteúdos, ou seja, elas não são meramente ilustrativas, pois esquematizam e resumem conteúdos explicitados no texto. Não deixe de observar a relação dessas figuras com os conteúdos da disciplina, pois relacionar aquilo que está no campo visual com o conceitual faz parte de uma boa formação intelectual.

Dicas (motivacionais)

O estudo desta disciplina convida você a olhar, de forma mais apurada, a Educação como processo de emancipação do ser humano. É importante que você se atente às explicações teóricas, práticas e científicas que estão presentes nos meios de comunicação, bem como partilhe suas descobertas com seus colegas, pois, ao compartilhar com outras pessoas aquilo que você observa, permite-se descobrir algo que ainda não se conhece, aprendendo a ver e a notar o que não havia sido percebido antes. Observar é, portanto, uma capacidade que nos impele à maturidade.

Você, como aluno dos cursos de Graduação na modalidade EAD, necessita de uma formação conceitual sólida e consistente. Para isso, você contará com a ajuda do tutor a distância, do tutor presencial e, sobretudo, da interação com seus colegas. Sugerimos, pois, que organize bem o seu tempo e realize as atividades nas datas estipuladas.

É importante, ainda, que você anote as suas reflexões em seu caderno ou no Bloco de Anotações, pois, no futuro, elas poderão ser utilizadas na elaboração de sua monografia ou de produções científicas.

Leia os livros da bibliografia indicada, para que você amplie seus horizontes teóricos. Coteje-os com o material didático, discuta a unidade com seus colegas e com o tutor e assista às videoaulas.

No final de cada unidade, você encontrará algumas questões autoavaliativas, que são importantes para a sua análise sobre os conteúdos desenvolvidos e para saber se estes foram significativos para sua formação. Indague, reflita, conteste e construa resenhas, pois esses procedimentos serão importantes para o seu amadurecimento intelectual.

Lembre-se de que o segredo do sucesso em um curso na modalidade a distância é participar, ou seja, interagir, procurando sempre cooperar e colaborar com seus colegas e tutores.

Caso precise de auxílio sobre algum assunto relacionado a esta disciplina, entre em contato com seu tutor. Ele estará pronto para ajudar você.

EAD

Introdução à Engenharia de Software

1

1. OBJETIVOS

- Compreender os elementos que compõem um sistema computacional.
- Entender os fatores que influenciam e dificultam a construção do *software*.
- Compreender o contexto histórico da indústria do software.
- Conhecer e entender os mitos envolvidos no desenvolvimento de *software*.
- Conhecer os reais objetivos da Engenharia de *Software*.

2. CONTEÚDOS

- Componentes do sistema computacional.
- Produtos de *software*.
- Contexto histórico da indústria do *software*.

- Mitos do *software*.
- Engenharia de Software.
- Reengenharia e engenharia reversa.

3. ORIENTAÇÕES PARA O ESTUDO DA UNIDADE

Para iniciar o estudo desta unidade, é importante que você tenha em mente que desenvolver *software* é muito mais do que a atividade de codificação, isto é, a implementação utilizando uma linguagem de programação.

À medida que os sistemas baseados em computador ficaram mais complexos, outras atividades foram inseridas ao processo de desenvolvimento, antes e depois da codificação.

Para o bom aproveitamento deste conteúdo, durante seus estudos, interaja com seus colegas. Em caso de dúvida, entre em contato com seu tutor. Lembre-se de que, ao final de cada unidade, você encontrará questões autoavaliativas, por meio das quais você poderá testar os conhecimentos adquiridos.

Quando for realizar seus estudos, escolha um ambiente que possa lhe proporcionar concentração, ou seja, um lugar calmo, arejado e estimulante. Esteja certo de que o ambiente e o meio contribuirão de maneira significativa para sua aprendizagem.

A palavra **sistema** é muito utilizada na área de informática, assim, será bastante citada durante esta disciplina. Não iremos definir a palavra sistema, pois não é este o nosso objetivo. Nesta disciplina, vamos nos limitar à sua aplicação em sistemas computacionais. É interessante, porém, que você entenda o conceito desta palavra que é muito empregada não só na informática, mas em muitas outras áreas. Certamente, você já a deve ter utilizado para se referir ao sistema solar, ao sistema educacional, ao sistema respiratório etc.

4. INTRODUÇÃO À UNIDADE

Nesta primeira unidade, você estudará os conceitos básicos de Engenharia de *Software*, tais como sistema computacional e produto de *software*.

Você vai conhecer as características dos *softwares* atuais e o seu contexto histórico. Além disso, terá a oportunidade de compreender as dificuldades envolvidas na construção dos sistemas computacionais e, com base em todos esses conceitos, compreenderá os reais objetivos, desafios e limitações da Engenharia de *Software*.

Bons estudos!

5. CONCEITOS INICIAIS

Elaborar um sistema é tanto arte quanto ciência

Quando o assunto é desenvolvimento de *software*, essa é uma frase comum de se ouvir, mas gostaria que você parasse um instante para analisá-la.

Para você, essa afirmação é verdadeira? Por quê?

Você acredita que se tivéssemos como única competência conhecer (tecnicamente) uma linguagem (sintaxe, semântica, estrutura etc.) desenvolveríamos bons sistemas, ou que se tivéssemos outras competências complementares desenvolveríamos melhores sistemas?

Apenas ter o “dom” para desenvolver sistemas de forma rápida não lhes garante a eficiência, a eficácia e a otimização.

Foi nesse contexto que a Engenharia de *Software* começou a “tomar forma”, com o propósito de colocar ordem na atividade de desenvolvimento de *softwares* e sistemas, que, até há pouco tempo, era considerada caótica.

Para compreender melhor a afirmação anterior, é fundamental que reflita um pouco.

- Ao ouvir ou ler a palavra ENGENHARIA, o que você pensa?
- O que é um *Software*?
- E o que é um Sistema?

Quando falamos em *software*, na verdade, estamos englobando conceitos de:

- Programas – quando executados, produzem a função e o desempenho desejados.
- Documentos – descrevem a operação e a utilização dos programas.
- Estrutura de dados – possibilitam que os programas manipulem adequadamente as informações.

Em uma das definições mais simples, Alcade (1991) descreve o *software* como a parte lógica que dota o equipamento físico de capacidade para realizar todo tipo de trabalho; considerando que um sistema computacional é composto por três pilares básicos, sendo eles: o elemento físico (*hardware*), o elemento lógico (*software*) e o elemento humano.

Segundo Pressman (2006), ao definir o *software* de uma forma clássica, podemos dizer que ele faz parte de um conjunto de instruções que, ao serem executadas, produzem a função e o desempenho desejados. Um *software* também possui estruturas de dados que permitem que as informações relativas ao problema a ser resolvido sejam manipuladas adequadamente e uma documentação específica sobre seu funcionamento, requisitos, funcionalidades, testes e manutenção. Essa documentação é necessária para um melhor entendimento de sua operação e uso.

O *software* é, portanto, um dos elementos que compõem um sistema computacional. Para Paula Filho (2001), o *software* é o elemento central: realiza estruturas complexas e flexíveis que trazem funções, utilidades e valor ao sistema. Contudo, há outros

componentes também indispensáveis, que são: as plataformas de *hardware*, os recursos de comunicação de informações, os documentos de diversas naturezas, as bases de dados e até os procedimentos manuais que integram aos automatizados.

Pressman (1995, p. 179) inclui como elementos de um sistema computacional: o *software*, o *hardware*, as pessoas, o banco de dados, a documentação e os procedimentos, sendo eles:

1) *Software*: programas de computador, estruturas de dados e documentação correlata que servem para efetivar o método, processo ou controle lógico necessário.
2) *Hardware*: dispositivos eletrônicos que fornecem a capacidade ao computador e dispositivos eletromecânicos que oferecem funções ao mundo externo.
3) Pessoas: usuários e operadores de *hardware* e *software*.
4) Bando de dados: uma grande e organizada coleção de informações a que se tem acesso pelo *software* e é parte integrante da função do sistema.
5) Documentação: manuais, formulários e outras informações descritivas que retratam o uso do sistema.
6) Procedimentos: os passos que definem o uso específico de cada elemento do sistema ou o contexto processual em que o sistema reside.

Estes elementos interagem uns com os outros para transformar informações. Assim, para que um sistema computacional cumpra realmente seus objetivos, é importante não apenas se preocupar com a construção de seus elementos, mas também é fundamental que todos os elementos estejam integrados entre si de forma efetiva.

Ao pensarmos no contexto da Engenharia de *Software*, o *software* será visto como um produto a ser desenvolvido. Independentemente das características do produto de *software*, é sempre importante considerar os conceitos desta disciplina em seu desenvolvimento. Logo, tanto no desenvolvimento de sistemas de uso

não restrito (sistemas utilizados por vários usuários, com características de integração com outros sistemas e com bases de dados internas e externas) quanto no desenvolvimento de sistemas de uso restrito (sistemas menores, restritos a um único usuário), a documentação, os testes, a manutenção, as técnicas de modelagem, a portabilidade, a flexibilidade, entre outros, não devem ser desprezados, e, sim, adaptados à realidade de cada aplicação.

É importante que você considere que, nesta disciplina, os conceitos são relacionados a programas de acesso não restrito, ou seja, para clientes maiores, que possuem características de integração com outros programas/sistemas e bases de dados internos e externos.

Para *softwares* menores, restritos, com um único usuário/cliente, os conceitos de Engenharia de *Software* não se aplicam em toda a sua extensão. Nesses casos, a documentação, os testes, a manutenção, as técnicas de modelagem, a portabilidade, a flexibilidade, etc., associados são poucos ou, em sua maioria, inexistentes, não por ser inviável realizá-los, mas por não serem necessários.

6. PRODUTO DE *SOFTWARE*

Um **produto de *software*** é sistematicamente destinado a pessoas com formações e experiências diferentes, isso significa que é necessário ter uma preocupação não apenas com as características de desenvolvimento, mas também com a interface e a documentação (seja ela de sistemas ou de usuário). Sem contar que, nesse caso, é necessário testar o *software* exaustivamente antes de entregá-lo ao cliente para detectar e corrigir as eventuais deficiências.

Resumindo, um programa desenvolvido para resolver um dado problema (usuário restrito e único) e um produto de *softwa-*

re destinado à resolução do mesmo problema (para vários clientes e com a meta de comercialização) são duas coisas diferentes. É óbvio que o esforço e o consequente custo associado ao desenvolvimento de um produto serão superiores aos de acesso restrito, até mesmo devido à sua concepção de comercialização futura.

Para melhor caracterizar o significado de *software*, é importante levantar algumas características a partir da comparação com outros "produtos", uma vez que ele é um elemento lógico, e não físico.

Pressman (1996) afirma que:

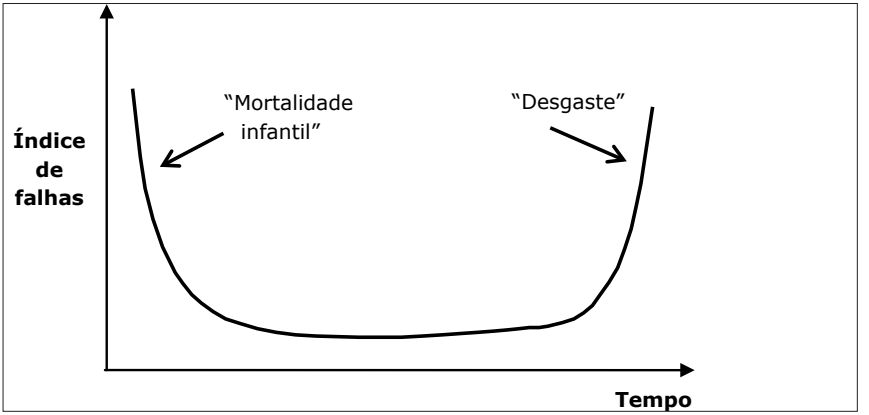
a) O *software* é concebido e desenvolvido como resultado de um trabalho de engenharia e não manufaturado no sentido clássico.

Neste sentido, é interessante que você reflita novamente sobre o conceito de engenharia. Pense em uma atividade de engenharia que lhe seja familiar (engenharia civil, mecânica, produção etc.). Imagine-a como ciência de construção, na qual, por meio de técnicas, ela pode desenvolver partes que pertencerão a um produto específico e chegue às suas próprias conclusões, fundamentado no fato de que o *software* não é um produto manufaturado, embora possua algumas características "conceituais" relacionadas à manufatura.

b) O *software* não se desgasta, ou seja, ao contrário da maioria dos produtos, o *software* não se caracteriza por um aumento na possibilidade de falhas, à medida que o tempo passa, devido ao desgaste físico e à ação ambiental (como, por exemplo, a poeira e o calor). O que pode acontecer é ele se tornar obsoleto e suas funcionalidades não mais satisfazerem à necessidade do usuário. Portanto, pode-se dizer que o *software* não se desgasta, mas, sim, deteriora-se**.**

Você poderá compreender melhor as diferenças entre o desgaste do *hardware* e a deterioração do *software* observando os gráficos das curvas de falhas propostos por Pressman (2006) a seguir:

Gráfico 1 Curva de falhas do *hardware*.



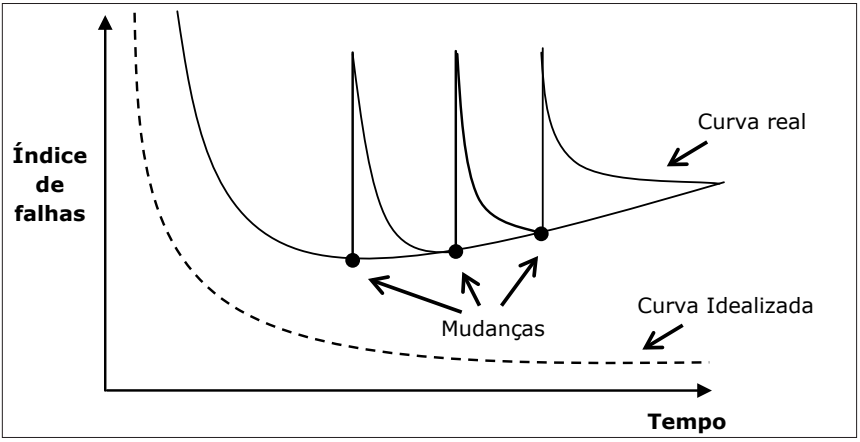
Fonte: Adaptado de Pressman (2006, p. 14).

Como você pode observar, no Gráfico 1, inicialmente, o *hardware* apresenta um elevado índice de falhas, as quais, normalmente, são atribuídas a problemas de projeto ou de fabricação. Com o passar do tempo, aferições são realizadas no *hardware* e essas falhas vão sendo corrigidas. Em seguida, são realizados ajustes e a curva se estabiliza, indicando que todos os defeitos foram eliminados ou que o nível falhas se encontra em um limite aceitável para o produto. No entanto, com o passar do tempo, o índice de falhas aumenta devido ao desgaste do produto, levando à sua substituição.

Observe, agora, o Gráfico 2, em que são apresentadas as duas curvas de falhas de *software*: **a curva real e a curva ideal**.

Na curva idealizada, representada pela linha tracejada no Gráfico 2, o *software*, sendo um elemento lógico, não sofre os desgastes físicos e ambientais comuns ao *hardware*. Assim, teoricamente, uma vez eliminadas as falhas iniciais inseridas durante o projeto e a construção do *software* (semelhante à construção do *hardware*), a curva se manterá estabilizada sem falhas ou com o menor índice possível de falhas. Portanto, o *software* não se deterioraria.

Gráfico 2 Curvas de falhas do *software*.



Fonte: Adaptado de Pressman (2006, p. 15).

No entanto, a curva ideal apresenta uma utopia do desenvolvimento de *software*: uma vez construído o *software*, todos os problemas acabaram. A realidade, porém, é bem diferente, e é mostrada na curva real de falhas do *software*, também no Gráfico 2. Durante a vida do *software*, este passará por várias e inevitáveis mudanças. À medida que as mudanças são realizadas, novas falhas podem ser inseridas, representando na curva real os picos de aumento de índice de falhas. Antes mesmo de se eliminarem as falhas inseridas na última modificação, nova necessidade de mudança pode ocorrer. Com o passar do tempo e as frequentes modificações, o índice mínimo de falhas vai aumentando, isto é, o *software* deixa de atender aos requisitos para os quais foi construído. Portanto, o *software* não se desgasta, mas, sim, deteriora-se.

c) A maioria dos produtos de *software* é concebida inteiramente sob medida, sem a utilização de componentes preexistentes. Atualmente, os conceitos de componentes e reuso são bastante utilizados no desenvolvimento de *software*, eles fazem parte de um conceito mais amplo, que é o de melhorar as técnicas de desenvolvimento de *software*.

Em razão de possuir características diferenciadas, o processo de desenvolvimento de *software* provoca várias dificuldades, as quais influenciam diretamente na **qualidade final do produto construído**.

Dentre os fatores que dificultam a construção do *software*, podemos citar o contexto histórico do desenvolvimento de *software* e os mitos envolvidos em sua construção. Além da própria característica "soft" – flexível – do *software*, que dificulta a proposta de uma metodologia de desenvolvimento padronizada, como ocorre em linhas de montagens de produtos físicos (manufatura). Essa flexibilidade do *software* refere-se às mais diversas aplicações para as quais são feitos (desde o controle de máquinas de lavar roupas até sistemas de informações que auxiliam na gestão de empresas inteiras), às características distintas de seus usuários, às diversas necessidades dos clientes da indústria do *software* e às diferentes realidades das próprias empresas que desenvolvem *software*. Todos estes fatores dificultam a construção do produto de *software*, podendo comprometer sua qualidade.

Considerando o desenvolvimento de *softwares* a nível industrial, já que estamos nos referindo a sistemas desenvolvidos por meio de técnicas de **manufatura**, e para atender diversos clientes com características diferenciadas, a Engenharia de *Software* procura responder a algumas questões que caracterizaram as preocupações com o processo de desenvolvimento de *software*, tais como:

- Por que o *software* demora tanto para ser concluído?
- Por que os custos de produção são tão elevados?
- Por que não é possível detectar todos os erros antes que o *software* seja entregue ao cliente?
- Por que é tão difícil medir o progresso durante o processo de desenvolvimento de *software*?

Essas são algumas das questões que a **Engenharia de *Software*** ajuda a resolver.

Com relação ao termo manufatura, este se trata de um processo de produção de bens em série, padronizada, ou seja, são produzidos muitos produtos iguais e em grande volume. As manufaturas surgiram durante a Revolução Industrial. Eram pequenas oficinas já com produção em série, porém com trabalho pratica-

mente manual. As fábricas ou indústrias tinham porte e mecanização muito maior. Atualmente, não existe mais essa distinção, e o termo manufaturado é sinônimo de industrializado.

Segundo Pressman (1996), embora seja difícil responder definitivamente às questões anteriores, é possível enumerar alguns problemas que originam tais questionamentos:

a) Durante o desenvolvimento de um *software*, raramente é dedicado um tempo para coletar os dados sobre o processo de desenvolvimento propriamente dito e, por não ter a quantidade de informações suficientes, as tentativas em estimar a duração/custo de produção de um *software* têm conduzido a resultados insatisfatórios; além disso, a falta dessas informações impede uma avaliação eficiente das técnicas e metodologias empregadas em seu desenvolvimento.

b) A insatisfação do cliente com o sistema "concluído" ocorre frequentemente, em decorrência, especialmente, do fato de que os projetos de desenvolvimento são baseados em informações vagas sobre as necessidades e os desejos do cliente (problema de comunicação entre cliente e fornecedor).

c) A qualidade do *software* é quase sempre suspeita. Esse problema é resultante da pouca atenção durante seu desenvolvimento (até porque o conceito de qualidade de software é algo relativamente recente).

d) Um *software* tem um período de vida sem atualização e, normalmente, é muito difícil de manter em operação, o que significa que o custo do *software* acaba sendo incrementado significativamente devido às atividades relacionadas à manutenção; isso é um reflexo da pouca importância dada à manutenibilidade no momento da concepção dos sistemas.

7. HISTÓRICO DE DESENVOLVIMENTO DE *SOFTWARE*

Desde os primórdios da computação, o desenvolvimento dos programas, ou a programação, era visto como uma forma de

arte, sem utilização de metodologias formais e sem qualquer preocupação com a documentação, entre outros fatores importantes. A experiência do programador era adquirida por meio de tentativa e erro. A verdade é que essa tendência ainda se verifica nos dias atuais. Com o crescimento dos custos de *software* (em relação aos de *hardware*) no custo total de um sistema computacional, o processo de desenvolvimento de *software* tornou-se um item de fundamental importância na produção de tais sistemas (PRESSMAN, 2006).

Se voltarmos no tempo e tentarmos montar uma linha cronológica sobre o desenvolvimento de softwares, nos defrontaremos com algumas décadas e acontecimentos que foram fundamentais para definir e estruturar o processo/tecnologia existente atualmente.

Na década de 1940 teve início a evolução dos sistemas computadorizados, e a grande maioria dos esforços, e consequentemente custos, estavam concentrados no desenvolvimento do *hardware*, em razão, especialmente, das limitações e dificuldades encontradas na época. O foco era desenvolver o *hardware*; e o desenvolvimento do *software*, na pesquisa, estava em segundo plano. Em outras palavras, nesta época, o *software* era o coadjuvante, enquanto o *hardware* era o ator principal.

Conforme a tecnologia de *hardware* foi sendo dominada, as preocupações voltaram-se para o *software* e, apenas no início da década de 1950, o desenvolvimento dos sistemas operacionais começava a entrar em foco. Nessa época, as linguagens de programação de alto nível, como FORTRAN e COBOL, e respectivos compiladores começaram a ser o centro da atenção.

Vale ressaltar que, nesta época, os sistemas computacionais eram privilégios de poucos; normalmente, de instituições governamentais ou de pesquisa. Além disso, na maioria das vezes, o desenvolvedor era o próprio usuário do sistema, ele desenvolvia para uso próprio, se fosse necessário fazer modificações, ele mes-

mo fazia. O fato de o cliente e o desenvolvedor serem a mesma pessoa, obviamente que facilitava o processo de levantamento das necessidades do cliente. Afinal, a pessoa conhecia suas próprias necessidades.

Entenda por "desenvolvedor" o profissional que participa da construção de *software*, podendo ser um programador ou um analista. O "usuário" é a pessoa que efetivamente fará uso do *software* produzido pelos desenvolvedores. O "cliente", que não necessariamente é também usuário, é a pessoa que contratou os serviços de um desenvolvedor de *software* ou de uma empresa desenvolvedora.

Nessa época, a tendência era ocultar do usuário questões relacionadas ao funcionamento interno da máquina, permitindo que este concentrasse seus esforços apenas na resolução dos problemas computacionais ao invés de preocupar-se com os problemas relacionados ao funcionamento da máquina (PRESSMAN, 1996).

Já no início da década de 1960, surgem os sistemas operacionais com características de multiprogramação, o que possibilitou a eficiência e o crescimento da utilização dos sistemas computacionais. Esse fato contribuiu de maneira significativa em relação às questões de queda de preço do *hardware* e ao aumento das vendas de máquinas e *software*.

A principal característica desse período foi a necessidade de desenvolver grandes sistemas de *software* em substituição aos pequenos programas aplicativos que eram utilizados até o momento, pois a grande necessidade das empresas era comprar máquinas com *softwares* "que funcionassem nelas".

Em decorrência da necessidade do mercado, surgiu um problema relacionado à falta de experiência e a não adequação dos métodos de desenvolvimento existentes para pequenos programas. A falta de condições dos desenvolvedores em suprir a necessidade do mercado, ainda na década de 1960, caracterizou a "crise do software".

As equipes de desenvolvimento tinham dificuldade em atender à demanda, pois era gasto mais tempo corrigindo erros e modificando *softwares* que já existiam no mercado do que desenvolvendo novos *softwares*.

Até então, os sistemas computacionais eram simples e desenvolvidos sem qualquer metodologia. Como dizem alguns autores, desenvolver *software* era "uma forma de arte", cada um o fazia da forma que considerava melhor. De repente, o mercado de sistemas expandiu-se e esses desenvolvedores viram-se obrigados a modificar os *softwares* para adequá-los às necessidades dos diversos usuários. Contudo, como estes *softwares* tinham sido feitos sem qualquer padronização ou documentação, o trabalho de modificá-los era bem difícil, especialmente se quem desenvolveu não era a mesma pessoa que iria modificar. Você consegue perceber a importância de se disciplinar o desenvolvimento do *software*? Pois este é o objetivo da Engenharia de *Software*.

A dificuldade de manutenção, a falta de documentação adequada, a rápida velocidade de mudanças no ambiente dos sistemas computacionais, a alta demanda por *software* e vários outros fatores ainda estão trazendo complicações para a indústria de *software*, dando continuidade a tal crise. A busca pela produtividade, tão inerente às outras indústrias, também é um grande desafio para indústria do *software*.

A **crise do *software*** foi considerada um marco importante, pois possibilitou o nascimento do termo "Engenharia de *Software*".

Com o passar do tempo, o preço de *hardware* foi diminuindo enquanto o de *software* não obedeceu à mesma tendência.

Atualmente, o *software* corresponde a uma porcentagem cada vez maior no custo global de um sistema informatizado; sobretudo porque a tecnologia de desenvolvimento de *software* implica, ainda, grande carga de trabalho, realizado por várias pessoas trabalhando "juntas" e em um prazo relativamente longo de desenvolvimento. Na maioria das vezes, o desenvolvimento desses sistemas é realiza-

do de forma **ad-hoc**, ou seja, elaborado especificamente para uma determinada ocasião ou situação, o que pode conduzir a uma extrapolação do tempo definido inicialmente para o desenvolvimento, o que acarreta um aumento no custo final do sistema.

Apesar de ter surgido na década de 1960, ainda hoje sentimos alguns sintomas da crise do *software*. Mesmo com os avanços na Engenharia de *Software* para minimizar os efeitos dessa crise, ainda nos deparamos com problemas como o do não cumprimento de prazos e custos e, ainda, entregamos produtos com a qualidade aquém da desejada pelo cliente.

O termo "Aflição Crônica" é utilizado como sinônimo de Crise do *Software*. Você deve estar achando estranho, não é? No livro de Pressman (1995) você encontra a explicação: a palavra crise denota algo decisivo no curso de algo. Portanto, se a Crise do *Software* teve início há mais de 30 anos e sentimos alguns de seus efeitos até hoje, é mais bem definida quando chamada de "Aflição Crônica". Pois aflição é algo que incomoda e, crônico, que demora longo tempo.

Se pensarmos no objetivo geral desta disciplina, considerando o que estudamos até o momento, podemos dizer que é apresentar propostas de soluções às questões relacionadas ao desenvolvimento de *software*, por meio da transferência dos principais conceitos relativos à Engenharia de *Software*, particularmente no que diz respeito à utilização de técnicas, metodologias e ferramentas de desenvolvimento de *software*.

Portanto, para entendermos melhor algumas técnicas de desenvolvimento de *software*, primeiro conheceremos alguns mitos que surgiram ao longo do tempo e estão relacionados aos principais aspectos do *software*. A crença em falsos mitos, por parte de desenvolvedores, clientes e gerentes de projetos de *software*, também contribuem para dificultar a construção do produto de *software* e comprometer a sua qualidade.

8. SOFTWARE: MITOS E REALIDADE

Segundo Sommervile (1992) e Pressman (2006), é possível apontar como causas principais dos problemas (estouro de cronograma, elevação de custo etc.) alguns aspectos, tais como:

a) falta de experiência dos profissionais na condução de projetos de *software*;
b) falta de treinamento no que diz respeito à utilização de técnicas e métodos formais para o desenvolvimento de *software*;
c) "cultura de programação" que é difundida e facilmente aceita por estudantes e profissionais da área de computação;
d) resistência às mudanças (particularmente no que diz respeito à utilização de novas técnicas de desenvolvimento de *software*) que os profissionais, normalmente, apresentam.

Esses aspectos levaram à definição e à discussão dos chamados "mitos e realidades" dos *softwares* apontados por Pressman (1996) e que, de certo modo, explicam alguns dos problemas de desenvolvimento apresentados anteriormente.

Mitos de gerenciamento

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

Realidade 1 – O fato de possuir um manual não é o suficiente. É preciso que a equipe aplique efetivamente os conhecimentos apresentados no manual. É necessário que as observações que constem no dado manual reflitam a moderna prática de desenvolvimento de *software* e que seja exaustivo com relação a todos os problemas de desenvolvimento que poderão aparecer no percurso. E, além disso...

Será que o manual é utilizado?

Os profissionais sabem de sua existência?

Ele reflete a prática moderna de desenvolvimento de *software*?

Ele é completo?

Mito 2 – A equipe dispõe de computadores de última geração, e consequentemente, suas ferramentas de desenvolvimento de *software* também são de última geração.

Realidade 2 – Ter à sua disposição o último modelo de computador pode ser bastante confortável para o desenvolvedor do *software*, mas não oferece nenhuma garantia quanto à qualidade do *software* desenvolvido. É preciso muito mais do que os mais recentes computadores para se fazer um desenvolvimento de *software* de alta qualidade, pois 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 – Aumentar a equipe também não será a solução. Alguém disse um dia que "... acrescentar pessoas em um projeto atrasado vai torná-lo ainda mais atrasado...". De fato, a introdução de novos profissionais em uma equipe em fase de condução de um projeto vai requerer uma etapa de treinamento dos novos elementos da equipe; para isso serão utilizados elementos que estão envolvidos diretamente no desenvolvimento, o que vai, consequentemente, implicar maiores atrasos no cronograma.

Mitos do cliente

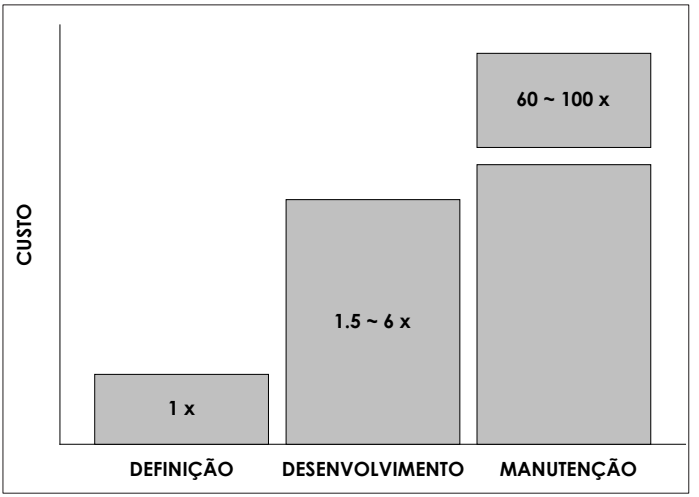
Mito 4 – Uma descrição breve e geral dos requisitos do *software* é o suficiente para iniciar seu projeto e maiores detalhes podem ser definidos posteriormente.

Realidade 4 – Esse é um dos problemas que podem conduzir um projeto ao fracasso. Portanto, é fundamental que o cliente procure 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 isso 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 seja que suporte alterações de requisitos significativas com adicional zero em relação ao custo de desenvolvimento. O aumento nos custos de desenvolvimento do *software* cresce à medida que as alterações de requisitos acontecem em estágios avançados do desenvolvimento do projeto, como demonstra a Figura 1.



Fonte: Adaptado de Pressman, 2006.
Figura 1 *Influência das alterações de requisitos no custo de um sistema.*

Mito 6 – Após a edição do programa e sua colocação em funcionamento, o trabalho está terminado.

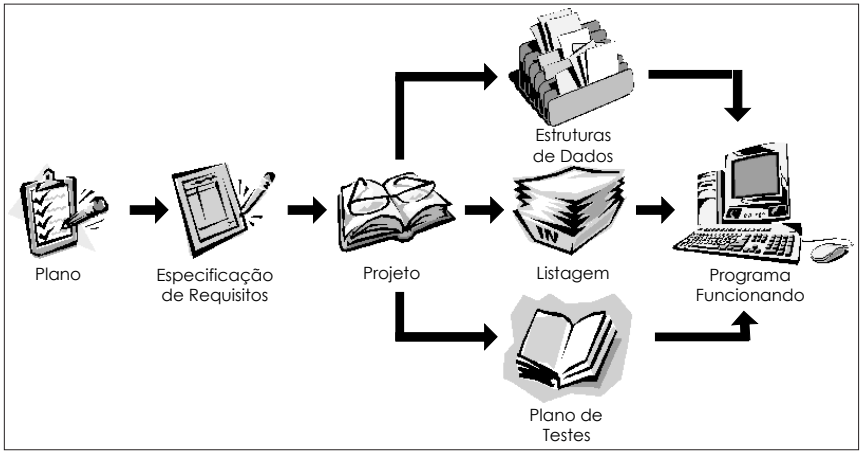
Realidade 6 – Na realidade, ocorre completamente diferente disso, pois de 60% a 80% do esforço de desenvolvimento de um *software* é despendido após sua entrega ao cliente (manutenção).

Mito 7 – Enquanto o programa não entrar em funcionamento, é impossível avaliar sua qualidade.

Realidade 7 – A preocupação com a garantia da qualidade do *software* deve fazer parte de todas as etapas do desenvolvimento, e, ao fim de cada etapa, os documentos do projeto devem ser revisados observando critérios de qualidade.

Mito 8 – O produto a ser entregue ao final do projeto é o programa funcionando.

Realidade 8 – O programa em funcionamento é um dos componentes do *software*. Um bom projeto caracteriza-se pela produção de um conjunto importante de documentos, os quais podem ser identificados com auxílio da Figura 2.



Fonte: Adaptada de Braz Junior, 2007.
Figura 2 *Conjunto de documentos e componentes.*

9. ENGENHARIA DE *SOFTWARE*

Vamos, agora, pensar em um sentido mais amplo: engenhar algo seria o mesmo que construir algo?

Você já pensou no significado da palavra **engenharia**? A palavra engenharia pode estar relacionada a cálculos e equações

matemática, não é? É exatamente por esse motivo que se torna necessário compreender melhor o termo engenharia e como ela se aplica à construção de *software*.

No dicionário Michaelis, dentre as inúmeras definições para "engenharia", encontramos: arte de aplicar os conhecimentos científicos à invenção, aperfeiçoamento ou utilização da técnica industrial em todas as suas determinações.

Sem querer ser exaustivo, vamos procurar a definição de engenharia em outro dicionário. No dicionário Aurélio, engenharia é a ciência, técnica e arte da construção de obras de grande porte, mediante a aplicação de princípios matemáticos e das ciências físicas.

Se você relacionou engenharia à matemática, você estava certo, pois nas suas mais diversas áreas, engenharia é a aplicação de princípios matemáticos na construção de algo. E na construção de *software*, isso também se aplica?

Essa é a questão que leva à má compreensão dos reais objetivos da Engenharia de *Software*. Você já viu que a Engenharia de *Software* surgiu na tentativa de resolver os problemas do desenvolvimento de *software* caracterizados pela crise do *software*.

Considerando o que você estudou até agora, será que todos os problemas da construção de *software* podem ser resolvidos por meio da matemática?

Desde o seu surgimento, a Engenharia de *Software* tem proposto vários métodos para minimizar os efeitos da crise do *software*, mas como já foi mencionado, até os dias de hoje encontramos muitos problemas envolvidos na construção do *software*. O ambiente que envolve a sua construção (características do cliente, do usuário, do desenvolvedor, as mudanças tecnológicas) e as crescentes aplicações para os novos *softwares* impossibilitam a criação de uma "receita de bolo" que garanta a construção do *software* com qualidade absoluta. Assim, os métodos propostos pela Enge-

nharia de *Software* devem ser adaptados à realidade de cada ambiente de desenvolvimento e às características do *software* que será construído.

Para Pfleeger (2004), a Engenharia de *Software* é uma disciplina da engenharia que se preocupa com todos os aspectos da produção de *software*, desde os estágios iniciais de especificação do sistema até a manutenção desse sistema, depois que ele entrou em operação.

Isso significa que a Engenharia de *Software* não se preocupa somente com aspectos técnicos de desenvolvimento de *software*, mas também com atividades como o gerenciamento de projetos e o desenvolvimento de ferramentas, métodos e teorias que deem apoio à produção de *software*.

Ao pensarmos sobre a questão: "o que é Engenharia de *Software*?", podemos dizer que é uma disciplina que reúne metodologias, métodos e ferramentas a serem utilizados, desde a percepção do problema até o momento em que o sistema desenvolvido deixa de existir, visando resolver problemas inerentes ao processo de desenvolvimento e ao produto de *software* (PFLEEGER, 2004).

É importante ressaltar que a Engenharia de *Software* foi criada, originalmente, para ser aplicada em grandes projetos de *software* desenvolvidos por grandes empresas.

Analisando o contexto brasileiro, em que predominam micro e pequenas empresas, será que os conceitos estudados não são válidos? Claro que são válidos, porém há a necessidade de adaptação dos conceitos a cada realidade das empresas e dos projetos. Além disso, já há na literatura métodos e técnicas de Engenharia de *Software* direcionadas a contextos e ambientes menores.

Bastante relacionado ao conceito de Engenharia de *Software* há, também, a Engenharia de Sistemas. A seguir, você conhecerá do que trata essa outra engenharia.

A engenharia de sistemas baseada em computadores preocupa-se com todos os aspectos de desenvolvimento e evolução de sistemas complexos em que o *software* desempenha papel principal atualmente. A engenharia de sistemas é, portanto, uma atividade interdisciplinar destinada a solucionar problemas do sistema como um todo. A engenharia de sistemas:

- delimita a função, o desempenho, as restrições e as interfaces entre dos elementos do sistema;
- aloca cada função a um ou mais elementos de sistema (*software*, *hardware*, pessoas etc.);
- Propõe e analisa alocações alternativas.

Para Sommervile (2006), os engenheiros de sistema estão envolvidos com a especificação do sistema computacional, na definição de sua arquitetura geral e na integração das diferentes partes necessárias para criar o sistema completo. Enfim, detalhes de construção de cada um dos componentes (como *hardware* e *software*) ficam a encargo das respectivas engenharias (Engenharia de *Hardware* e Engenharia de *Software*).

A Engenharia de *Software* busca prover a tecnologia necessária para desenvolver *software* de alta qualidade com baixo custo e é um grande desafio que os engenheiros de *software* têm a enfrentar. O engenheiro de *software* utiliza o computador como ferramenta para solucionar problemas de diversas áreas.

Segundo Pressman (2006), a Engenharia de *Software* possui três elementos fundamentais: métodos, ferramentas e procedimentos.

- **Métodos:** são os detalhes de "como fazer" para construir o *software*, trata-se de um procedimento formal para produzir algum resultado. Os métodos, normalmente, envolvem atividades como planejamento e gerenciamento, análise de requisitos, projeto, codificação, teste e manutenção.

- **Ferramentas:** proporcionam apoio automatizado ou semiautomatizado aos métodos. A ferramenta pode tornar os métodos mais precisos, eficientes e produtivos e melhorar a qualidade do produto resultante.
- **Procedimentos:** são as ligações entre os métodos e as ferramentas, possibilitando o desenvolvimento racional e oportuno do *software*. Em outras palavras, os procedimentos definem a sequência em que os métodos serão aplicados, os produtos que se exigem que sejam entregues, os controles que ajudam a assegurar a qualidade e a coordenar as mudanças e os marcos de referência que possibilitam aos gerentes de *software* avaliar o progresso.

Para que você compreenda melhor os conceitos de métodos, ferramentas e procedimentos, vejamos a analogia que Pfleeger (2004) faz aplicando esses conceitos na culinária. Na culinária? Pode parecer estranho, mas é util. Vejamos.

Um chefe de cozinha pode preparar um molho empregando ingredientes sequencialmente combinados em uma ordem e momentos específicos, de tal maneira que o molho engrosse, mas não coagule ou desande, esse é o método. As ferramentas são os utensílios de cozinha que podem ser empregados no preparo do molho. Um procedimento é como uma receita, combinando ferramentas e métodos que em harmonia produzem o resultado específico.

Além disso, há diferentes filosofias ou abordagens para cozinhar (cozinha francesa, japonesa, italiana etc.), que são chamadas de paradigmas.

Na nossa analogia, podemos dizer que os paradigmas da Engenharia de *Software* são conjuntos de métodos, ferramentas e procedimentos propostos pela Engenharia de *Software*. Esses paradigmas serão apresentados na próxima unidade.

10. REENGENHARIA E ENGENHARIA REVERSA

Além do conceito de Engenharia de *Software*, há também dois outros conceitos relacionados: reengenharia e engenharia reversa.

O conceito de reengenharia está bastante relacionado à manutenção do *software*.

Os sistemas computacionais desenvolvidos para apoiar as atividades empresariais, chamados de sistemas de informação, devem se adaptar às mudanças ocorridas na empresa, para que continue a atender as necessidades dos usuários. Estes sistemas são os que normalmente mais sofrem mudanças depois de sua implantação.

> Um sistema de informação pode ser definido tecnicamente como um conjunto de componentes inter-relacionados que coleta (ou recupera), processa, armazena e distribui informações destinadas a apoiar a tomada de decisões, a coordenação e o controle de uma organização. Além de dar suporte à tomada de decisões, à coordenação e ao controle, esses sistemas também auxiliam os gerentes e trabalhadores a analisar problemas, visualizar assuntos complexos e criar novos produtos (LAUDON e LAUDON, 2006, p. 7).

Segundo Laudon e Laudon (1999), o ambiente empresarial está se alterando e é fluido, isto é, novas tecnologias, tendências econômicas, desenvolvimentos políticos e regulamentações que afetam os negócios estão constantemente emergindo. Geralmente, quando as empresas falham, é porque negligenciaram a resposta ao ambiente mutável.

Assim, no contexto atual, o ambiente empresarial mostra-se bastante instável. A empresa que não se adaptar às mudanças ambientais deixa de ser competitiva. Da mesma forma, os sistemas de informação destas empresas devem também se adaptar às suas novas necessidades, para que continuem sendo úteis e cumpram com seu papel de apoiar as decisões.

Porém, muitos dos sistemas de informação fundamentais aos negócios estão se tornando difíceis de serem alterados. Para Pressman (1996) remendos são feitos sobre remendos, resultando

em *softwares* que funcionam de forma ineficiente e falham com frequência, não correspondendo às necessidades dos usuários. Portanto, a manutenção de sistemas em fase de envelhecimento tornou-se proibitivamente dispendiosa para muitos sistemas de informação.

Quando, ao longo das manutenções, as alterações são feitas sem administração, sem atualização da documentação, na verdade, quando são feitos "remendos", à medida que o tempo passa novas alterações ficam cada vez mais difíceis de serem feitas. Além disso, sistemas feitos com tecnologia muito arcaica, muitas vezes, não acomodam as necessidades atuais. É exatamente nestes casos que é feita a reengenharia do *software*. Reengenharia, em poucas palavras, significa reconstruir o *software* utilizando uma nova tecnologia, para melhorar a facilidade de manutenção.

De acordo com Pressman (1996), a reengenharia não somente recupera as informações de projeto de um *software* existente, mas usa estas informações para alterar ou reconstruir o sistema existente, num esforço para melhorar sua qualidade. Logo, um sistema que tenha sofrido reengenharia reimplementa a função do sistema existente, acrescido de novas funções.

> A reengenharia pode envolver redocumentar, organizar e reestruturar o sistema, traduzir o sistema para uma linguagem de programação mais moderna e atualizara estrutura e os valores dos dados do sistema (SOMMERVILLE, 2005, p. 533).

Agora que você já compreendeu o significado do termo reengenharia, vamos apresentar a engenharia reversa.

Engenharia reversa

A engenharia reversa tem sua origem no mundo do *hardware*. Uma empresa desmonta o produto de um concorrente para entender os segredos do projeto e da fabricação do mesmo. Estes segredos poderiam ser obtidos através das especificações do projeto e fabricação do produto, mas estes documentos não são

disponíveis à empresa que está fazendo a engenharia reversa. A engenharia reversa de *software* é bastante semelhante. No entanto, na maioria das vezes, o *software* que passa pela engenharia reversa não é o de um concorrente, mas é trabalho desenvolvido pela própria empresa há muitos anos. Os segredos a serem descobertos são obscuros porque nenhuma documentação foi desenvolvida (PRESSMAN, 2006).

Logo,

> [...] a engenharia reversa é o processo de análise de um programa, em um esforço para representá-lo em uma abstração mais alta do que o código-fonte. A engenharia reversa é um processo de recuperação de projeto (PRESSMAN, 2006, p. 687).

Para Sommerville (2005, p. 534) na "engenharia reversa o programa é analisado e as informações são extraídas dele, a fim de ajudar a documentar sua organização e funcionalidade".

A engenharia reversa não é o mesmo que reengenharia. O objetivo da engenharia reversa é derivar o projeto ou a documentação de um sistema a partir de seu código-fonte. Já o objetivo da reengenharia é produzir um novo sistema com manutenção mais fácil. Assim, a engenharia reversa para desenvolver uma melhor compreensão de um sistema é com frequência parte do processo de reengenharia (SOMMERVILLE, 2005). Em outras palavras, a engenharia reversa pode anteceder um processo de reengenharia, num esforço de compreender os requisitos que levaram a construção do *software* a ser reconstruído.

Para concluir, vamos relembrar: a engenharia de *software* é o processo de desenvolver o *software* a partir de um estudo das necessidades do cliente e futuros usuários. O processo de reengenharia de *software* tem como partida o conhecimento das características do sistema antigo, a ser reconstruído. As características do sistema antigo podem ser levantadas com o auxílio da engenharia reversa.

11. QUESTÃO AUTOAVALIATIVA

Sugerimos que você procure responder, discutir e comentar as questões a seguir que tratam da temática desenvolvida nesta unidade. Se você encontrar dificuldade em respondê-las, procure revisar os conteúdos estudados para sanar as suas dúvidas. A autoavaliação pode ser uma ferramenta importante para você testar seu desempenho. Este é um momento impar para você fazer uma revisão desta unidade. Lembre-se de que no ensino a distância a construção do conhecimento se dá de forma cooperativa e colaborativa, portanto compartilhe com seus colegas suas descobertas.

1) A Engenharia de Software:

I – Não visa o desenvolvimento de teorias e fundamentações, preocupando-se unicamente com as práticas de desenvolvimento de software.

II – Tem como foco o tratamento dos aspectos de desenvolvimento de software, abstraindo-se dos sistemas baseados em computadores, incluindo hardware e software.

III – Tem como métodos as abordagens estruturadas para o desenvolvimento de software que incluem os modelos de software, notações, regras e maneiras de desenvolvimento.

IV – Segue princípios, tais como o da Abstração, que identifica os aspectos importantes sem ignorar os detalhes e o da Composição, que agrupa as atividades em um único processo para distribuição aos especialistas.

É correto o que se afirma em:

a) III e IV, apenas.
b) I, II, III e IV.
c) I e II, apenas.
d) I, II e III, apenas.
e) II, III e IV, apenas.

Fonte: Prova aplicada em 01/2010 para o concurso do(a) DPE - SP - 2010 - Agente, realizado pelo órgão/instituição Defensoria pública de São Paulo, área de atuação Jurídica, organizada pela banca FCC, para o cargo de Agente de Defensoria - Programador, nível superior , área de formação Tecnologia da Informação .

Gabarito

Depois de responder às questões autoavaliativas, é importante que você confira o seu desempenho, a fim de que possa saber se é preciso retomar o estudo desta unidade. Assim, confira, a

seguir, as respostas corretas para as questões autoavaliativas propostas anteriormente:

1. (d)

12. CONSIDERAÇÕES

Chegamos ao final do estudo da nossa primeira unidade. Espero que tenha sido proveitoso para você!

Nesta unidade, você teve a oportunidade de discutir o quanto o desenvolvimento de um *software* está relacionado aos conceitos técnicos e às habilidades de programar.

Relembre os conceitos de métodos, ferramentas e procedimentos de Engenharia de *Software*, estes conceitos serão uteis para a continuidade dos estudos.

Você compreendeu as dificuldades envolvidas na construção de um *software*, que iniciaram nos primórdios da indústria do *software* e que muitos dos problemas ainda existem. Há mitos que ainda distorcem as reais características da construção do *software*. Além disso, você conheceu a problemática envolvida na construção do *software* desde os primórdios e pôde verificar que, até hoje, muitas dificuldades ainda existem.

Com base em todos estes conceitos estudados, você pôde conhecer o que é a Engenharia de *Software*, seus reais objetivos, limitações e desafios. E perceberá que uma das referências mais citadas durante esta disciplina e em várias fontes sobre Engenharia de *Software*, é de Roger S. Pressman, ou simplesmente Pressman, uma das personalidades conceituadas e que mais publica na área de Engenharia de *Software*.

Na próxima unidade, você conhecerá o que é um processo de *software* e os principais paradigmas de desenvolvimento de *software*.

13. REFERÊNCIAS BIBLIOGRÁFICAS

ALCADE, E.; GARCIA, M; PENUELAS, S. *Informática Básica*. São Paulo: Makron Books, 1991.

BRAZ JUNIOR, O. P. *Engenharia de software*. Disponível em: <http://inf.unisul.br/~osmarjr/download/apostila/apostila_engenharia_software.zip>. Acesso em: 13 maio 2007.

FERREIRA, A. B. DE H. *Dicionário Aurélio On-line*. Disponível em: <http://www.dicionariodoaurelio.com/>. Acesso em: 04 jan. 2010.

LAUDON, K. C; LAUDON, J. P. *Sistemas de Informação*. 4. ed. Rio de Janeiro: LTC, 1999.

_____. *Sistemas de Informação Gerenciais*. Administrando a empresa digital. 5. ed. São Paulo: Pearson Education do Brasil, 2006.

MICHAELIS. *Moderno dicionário da língua portuguesa*. São Paulo: Cia. Melhoramentos, 1998.

PAULA FILHO, W. P. *Engenharia de Software*: fundamentos, métodos e padrões. 2. ed. Rio de janeiro: LTC, 2001.

PFLEEGER, S. L. *Engenharia de software*: teoria e prática. São Paulo: Prentice Hall, 2004.

PRESSMAN, R. S. *Engenharia de software*. São Paulo: Makron Books, 1996.

_____. *Engenharia de software*. São Paulo: Makron Books, 2006.

SOMMERVILLE, I. *Engenharia de Software*. 6. ed. São Paulo: Pearson Addison Wesley, 2005.

EAD

Paradigmas de Desenvolvimento de *Software*

2

1. OBJETIVOS

- Compreender o que é um processo de *software*.
- Entender a importância do processo para o sucesso do produto desenvolvido.
- Conhecer e aplicar os principais paradigmas de desenvolvimento de *software*.

2. CONTEÚDOS

- Processo de *software*.
- Paradigmas de *software*.
- *Extreme Programming*.

3. ORIENTAÇÕES PARA O ESTUDO DA UNIDADE

No início e durante o desenvolvimento das unidades, é importante que você fique sempre atento às informações contidas

neste Material Didático. Programe, organize seus estudos e participe, ativamente, na Sala de Aula Virtual (SAV). Ter disciplina para estudar pode ajudar você a tirar o máximo de proveito em seu curso na modalidade EAD.

Durante esta unidade você conhecerá os principais modelos de processo de *software*. Porém, são muitos os modelos propostos pela Engenharia de *Software*. Portanto, não limite seu aprendizado apenas aos modelos aqui apresentados, estude, pesquise, busque novos conhecimentos!

Adquira o hábito da pesquisa, visite *sites*, pesquise em livros e periódicos. Sugiro que você dê pequenas pausas na leitura e pesquise sobre os conceitos abordados nas unidades. E lembre-se de compartilhar suas descobertas com seus colegas de turma.

4. INTRODUÇÃO À UNIDADE

Na Unidade 1, você estudou os componentes do sistema computacional. Verificou a importância do *software* para o bom funcionamento do sistema e as características do produto de *software*.

Conheceu, também, as dificuldades envolvidas na produção de *software*, o contexto histórico e os mitos que envolvem a indústria do *software*. Por meio desses conceitos, você pôde compreender o que é a Engenharia de *Software*.

Nesta unidade, você compreenderá o que é um processo de *software* e a importância desse conceito na Engenharia de *Software*.

Além disso, você conhecerá alguns dos paradigmas para o desenvolvimento de *software*.

5. PROCESSO DE *SOFTWARE*

De acordo com Pressman (2006), resolver os problemas levantados até o momento, assim como compreender a existência

dos mitos, é possível; mas, antes de tudo, é preciso ter ciência de que não existe uma abordagem mágica que seja a melhor solução. É necessário que haja uma combinação de métodos que sejam abrangentes a todas as etapas do desenvolvimento de um *software*.

Além disso, é importante e desejável que os métodos sejam suportados por um conjunto de ferramentas que permitam automatizar o desenrolar das etapas, juntamente com uma definição clara de critérios de qualidade e produtividade de *software*. São esses os aspectos que mais influenciam a disciplina de Engenharia de Softwares.

Como já foi dito anteriormente, ao pensarmos sobre a questão: "o que é engenharia de *software*?", podemos dizer que é uma disciplina que reúne metodologias, métodos e ferramentas a serem utilizados, desde a percepção do problema até o momento em que o sistema desenvolvido deixa de existir, visando resolver problemas inerentes ao processo de desenvolvimento e ao produto de *software* (PFLEEGER, 2004).

A Engenharia de *Software* auxilia no processo de desenvolvimento de sistemas de forma a produzi-los com:

- maior qualidade;
- maior rapidez;
- menor custo.

Na Engenharia de *Software*, há dois termos dependentes, bastante usados, mas que podem ser definidos distintamente: **produto de *software*** e **processo de *software***. Você já aprendeu o que é um produto de *software* na Unidade 1. Vamos aprender, agora, o que é um processo de *software*. Para que você compreenda os processos de desenvolvimento de *software*, é importante, antes, contextualizarmos o que é um **processo**.

Antes de continuarmos, pare um pouco e tente definir o que é processo de *software*, reflita sobre essa palavra. O que você entende por processo?

Paula Filho (2001) define processo como:

- uma receita que é seguida por um projeto, o qual concretiza uma abstração, que é o processo;
- um conjunto de passos parcialmente ordenados, constituídos por atividades, métodos, práticas e transformações, usado para atingir uma meta.

Além disso, não se deve confundir o processo com o produto, ou com a execução do processo por meio de um projeto.

Ainda segundo o autor citado, um **processo definido** é aquele que possui documentação que descreve:

a) o que é feito (produto);
b) quando (passos);
c) por quem (agentes);
d) as coisas que usa (insumos);
e) as coisas que produz (resultados).

Pressman (2006) define processo de *software* como um arcabouço (estrutura) para as tarefas que são necessárias para construir *softwares* de alta qualidade. Assim, um processo de *software* define a abordagem que é adotada quando o *software* é desenvolvido. O processo é o alicerce da Engenharia de *Software* e permite o desenvolvimento racional e oportuno de *softwares* de computador. Dessa forma, os processos de *software* formam a base para o controle gerencial de projetos de *software* e estabelecem o contexto no qual os métodos são aplicados.

Devido às inúmeras variáveis que envolvem o desenvolvimento de *software*, não há um método em particular que seja o melhor para a solução dos problemas de construção de *software*. No entanto, combinando os métodos e ferramentas, os engenheiros de *software* conseguem adaptar um processo de *software* apropriado aos produtos que constroem e às diferentes características dos ambientes de construção. A definição de um processo envolve a escolha dos métodos, ferramentas e procedimentos para projetar,

construir e manter grandes sistemas de *software* de forma profissional.

Para Sommerville (2005), não há um processo ideal, há muitas oportunidades de trabalho para melhorá-los, em muitas organizações.

O **Processo de desenvolvimento** corresponde ao conjunto e ao ordenamento de atividades de forma que o produto desejado seja obtido (SOMMERVILLE, 1994).

O resultado do esforço de um processo de desenvolvimento de *software*, ou simplesmente **processo de *software***, é um **produto de *software***. Se o processo de *software* for ruim, certamente o produto de *software* sofrerá, inevitavelmente, as consequências.

Há, hoje, ***vários modelos*** de processo de *software*, também, chamados de Paradigmas da Engenharia de *Software*, e cada um representa uma tentativa de colocar ordem em uma atividade inerentemente caótica (PRESSMAN, 1996). Você verá este assunto no próximo tópico. Acompanhe!

6. PARADIGMAS DA ENGENHARIA DE *SOFTWARE*

Modelo de desenvolvimento, modelo de processo, paradigma ou ciclo de vida são termos utilizados como sinônimos. Portanto, paradigma é uma representação abstrata do processo de desenvolvimento que, em geral, definirá como as etapas relativas à criação do *software* serão conduzidas e inter-relacionadas para atingir o objetivo, que é a obtenção de um produto de *software* de alta qualidade a um custo relativamente baixo.

A Engenharia de *Software* surgiu para minimizar os feitos da crise do *software* e propôs modelos prescritivos de processo, criados, originalmente, para colocar ordem no caos do desenvolvimento de *software*. A organização que desenvolve *software* sem um processo definido desenvolve de forma caótica. Em um ambiente caótico, indisciplinado, a qualidade do produto é imprevisível.

Os modelos prescritivos não são perfeitos, mas fornecem um roteiro útil para o trabalho da Engenharia de *Software*. São prescritivos porque prescrevem um conjunto de elementos de processo – atividades, ações, tarefas, produtos de trabalho, mecanismos de garantia de qualidade e de controle de modificações para cada projeto. Esses modelos também descrevem um fluxo de trabalho, ou seja, a forma como cada elemento do processo deve ser interligado aos outros (PAULA FILHO, 2001).

Genericamente, segundo Pressman (1996), os processos de desenvolvimento de *software* possuem três fases: definição, desenvolvimento e manutenção.

- Definição: foco no "o quê", isto é, nesta fase, o desenvolvedor tenta identificar quais informações devem ser processadas, quais as funções e os desempenhos desejados, quais interfaces devem ser estabelecidas, quais as restrições e os critérios de validação para definir um sistema com qualidade.
- Desenvolvimento: foco no "como", ou seja, o desenvolvedor deve definir como a estrutura de dados e a arquitetura do *software* devem ser projetadas e construídas. Nesta fase, o *software* deve ser construído e testado.
- Manutenção: o desenvolvedor concentra-se nas mudanças associadas à correção de erros, adaptações e ampliações a serem feitas no *software*.

Você encontrará detalhes sobre essas três fases na Leitura Complementar 1. Estas são as atividades básicas em qualquer paradigma de desenvolvimento de *software*. Em qualquer modelo de processo, certamente você terá que descobrir as necessidades do cliente. Uma vez descobertas estas necessidades, você terá que materializá-las através da construção do *software*. Depois de construído e entregue inevitavelmente este *software* será modificado.

Independentemente do paradigma (modelo) escolhido, de forma explícita ou implícita, com mais ou menos detalhes, certa-

mente estas fases ocorrerão durante o processo de desenvolvimento de *software*.

Vários são os modelos de desenvolvimento existentes na literatura:

1) Sequencial linear:
   - Modelo cascata ou Ciclo de vida clássico.
   - Prototipação.
   - Modelo RAD (Rapid Application Development).
2) Modelos evolutivos de processo de *software*:
   - Modelo incremental.
   - Modelo espiral.
   - Modelo de montagem de componentes.
   - Modelo de desenvolvimento concorrente.
3) Modelos de métodos formais.
4) Técnicas de quarta geração.

Como já mencionado, são vários os modelos propostos pela Engenharia de *Software*. Por questão de tempo, veremos nesta disciplina somente os modelos mais conhecidos. No entanto, não limite seu aprendizado a estes, pesquise sobre outros modelos existentes.

A seguir, serão descritos alguns dos modelos conhecidos e utilizados em desenvolvimento de *software*.

Modelo cascata ou ciclo de vida clássico

Características do modelo:

O modelo cascata ou ciclo de vida clássico é o mais simples de desenvolvimento de *software*, estabelecendo uma ordenação linear referente à realização das diferentes etapas.

Esse modelo, que é o mais antigo e amplamente utilizado pela engenharia de *software* por ter sido modelado em virtude do ciclo da engenharia convencional.

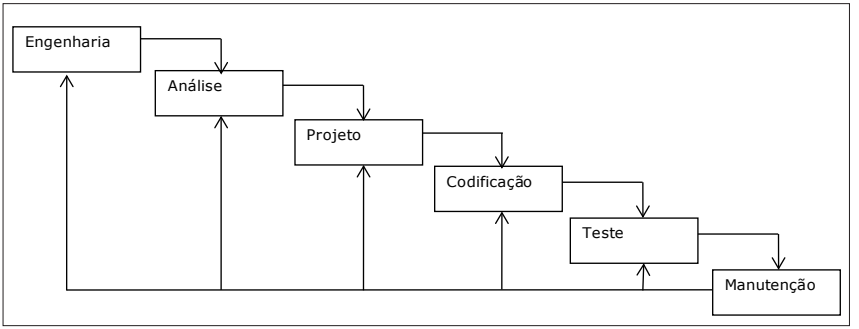
Requer uma abordagem sistemática, sequencial ao desenvolvimento de *software*, e o resultado de uma fase constitui-se na entrada da outra.

Há inúmeras variações desse modelo, dependendo da natureza das atividades e do fluxo e do controle entre elas. Os estágios principais do modelo estão relacionados às atividades fundamentais de desenvolvimento.

Acompanhe, na Figura 1, a descrição gráfica desse modelo, cuja estrutura lembra a queda d'água, derivando daí o seu nome.

Pressman (1996) descreve cada uma destas etapas:

1) Engenharia de Sistemas: estabelecem-se os requisitos gerais de todos os elementos do sistema computacional (*software*, *hardware*, pessoal, banco de dados, documentações e procedimentos).
2) Análise de requisitos: são levantados os requisitos específicos do *software*, de forma mais detalhada.
3) Projeto: são projetados a estrutura dos dados, a arquitetura do *software*, os detalhes procedimentais e de interface.
4) Codificação: concretiza-se o projeto, sendo criado o *software* utilizando uma linguagem de programação.
5) Testes: uma vez gerado o código, este deve ser testado no que diz respeito a aspectos internos e externos.
6) Manutenção: refere-se às mudanças (correções, adaptações e expansões) que o *software* sofrerá depois de ser entregue ao cliente.



Fonte: Adaptado de Pressman (1996, p.33)
Figura 1 *Modelo em Cascata.*

Observe que o modelo **Cascata** apresenta características interessantes, particularmente em razão da definição de um ordenamento linear das etapas de desenvolvimento.

Inicialmente, como forma de identificar precisamente o fim de uma etapa e o início da seguinte, um mecanismo de certificação (ou revisão) é implementado ao final de cada etapa; isso é realizado, normalmente, por meio da aplicação de algum método de validação ou verificação, cujo objetivo será garantir que a saída de uma dada etapa é coerente com sua entrada (a qual já é a saída da etapa precedente). E, ao final de cada etapa realizada, deve existir um resultado (ou saída) que possa ser submetido à atividade de certificação.

De acordo com Pressman (2006), duas diretivas que norteiam o desenvolvimento, segundo o modelo **Cascata**, são:

1) Todas as etapas definidas no modelo devem ser realizadas, isso porque, em projetos de grande complexidade, a realização formal das etapas determinará o sucesso ou não do desenvolvimento.
2) Respeitar rigorosamente a ordenação das etapas na forma como foi apresentada.

A realização informal e implícita de algumas dessas etapas poderia ser realizada apenas no caso de projetos de pequeno porte.

Vale ressaltar, também, que os resultados de um processo de desenvolvimento de *software* não devem ser exclusivamente o programa executável e a documentação associada.

Cada fase demonstrada na Figura 1 pode nos gerar resultados que darão origem à documentação do *software*. Em cada fase, documentos específicos são gerados e são denominados como: documento de especificação de requisitos, projeto do sistema, plano de teste e relatório de testes, código final, manuais de utilização, relatórios de revisões etc.

Apesar de ser mais popular, podem-se apontar algumas limitações apresentadas pelo modelo **Cascata**:

- projetos reais raramente seguem o fluxo sequencial que o modelo propõe;
- há dificuldade em estabelecer explicitamente todos os requisitos logo no início, pois, no começo dos projetos, sempre há uma incerteza natural;
- paciência do cliente, porque uma versão executável do *software* só será disponível em uma etapa avançada do desenvolvimento.

Esse modelo apresenta algumas contribuições, tais como:

- O processo de desenvolvimento de *software* deve ser sujeito à disciplina, planejamento e gerenciamento.
- A implementação do produto deve ser prorrogada até que os objetivos tenham sido completamente entendidos.

Foi apresentado aqui apenas um resumo do Modelo em Cascata. Este modelo serviu de base para a construção de muitos outros modelos que surgiram na tentativa de eliminar as desvantagens do Ciclo de Vida Clássico. Para ampliar seus conhecimentos sobre o modelo, que é a base da Engenharia de *Software*, pesquise em livros de Engenharia de *Software* ou na internet. Depois de pesquisar, reflita se você considera o Modelo em Cascata muito burocrático para o desenvolvimento de *software*?

Modelo de prototipação

Prototipação é um modelo de processo de desenvolvimento que busca contornar algumas das limitações existentes no modelo cascata. Isso é realizado por meio da obtenção de um protótipo, com base no conhecimento dos requisitos iniciais para o sistema.

O desenvolvimento desse protótipo é realizado obedecendo a diferentes etapas mencionadas anteriormente, tais como: a análise de requisitos, o projeto, a codificação e os testes. Essas etapas não são realizadas necessariamente de modo explícito ou formal.

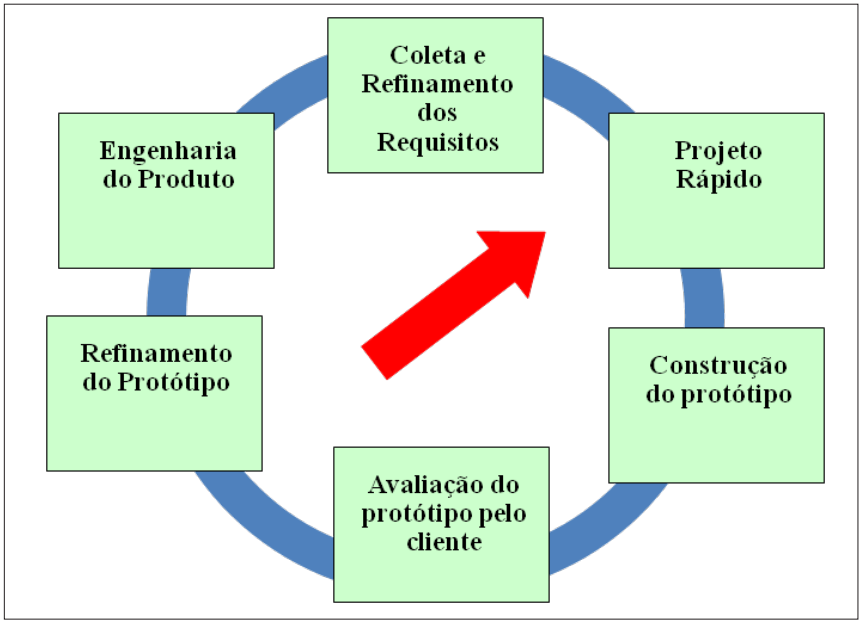
A prototipação tem como objetivo entender os requisitos do usuário e, assim, obter uma melhor definição dos requisitos do sistema, possibilitando que o desenvolvedor crie um modelo (protótipo) do *software* que será construído. É apropriado para quando

o cliente definiu um conjunto de objetivos gerais para o *software*, mas não identificou detalhadamente esses requisitos.

Após criar o protótipo, será colocado à disposição do cliente, pois este auxiliará na melhor compreensão do que será o sistema desenvolvido. Além disso, por meio da manipulação desse protótipo, é possível validar ou reformular os requisitos para as etapas seguintes do sistema.

Segundo Pressman (2006), esse modelo, conforme ilustrado na Figura 4, apresenta algumas características interessantes, tais como:

- facilita o desenvolvimento para sistemas de grande porte, os quais representem certo grau de dificuldade para exprimir rigorosamente os requisitos;
- ao construir um protótipo do sistema, é possível demonstrar sua possibilidade de desenvolvimento;
- é possível obter uma versão, mesmo simplificada, do que será o sistema, com um pequeno investimento inicial.



Fonte: Pressman (1996, p.36).
Figura 2 *Modelo de Prototipação*.

Esse modelo apresenta alguns problemas, tais como:

1) O cliente não sabe que o *software* que ele vê não considerou, durante o desenvolvimento, a qualidade global e a manutenibilidade a longo prazo.
2) O desenvolvedor, frequentemente, faz uma implementação comprometida (utilizando o que está disponível) com o objetivo de produzir rapidamente um protótipo.
3) Os protótipos não são sistemas completos e deixam a desejar em alguns aspectos, como, por exemplo, na interface com o usuário.
4) Os esforços de desenvolvimento são concentrados, especialmente, nos algoritmos que implementam as principais funções associadas aos requisitos apresentados, a interface pertence a esse nível, o qual é considerado parte supérflua do desenvolvimento, o que permite caracterizar essa etapa por um custo relativamente baixo.

A prototipação apresenta algumas contribuições. Observe, a seguir, a descrição de cada uma.

1) a prototipação é um ciclo de vida eficiente, mesmo que ocorra problemas;
2) as regras do jogo devem ser definidas logo no início, o que é fundamental para que o sistema seja satisfatório;
3) o cliente e o desenvolvedor devem concordar que a função principal do protótipo é permitir que ele seja um mecanismo favorável à definição dos requisitos;
4) a experiência adquirida no desenvolvimento do protótipo será de extrema utilidade nas etapas posteriores do desenvolvimento do sistema real, permitindo reduzir o seu custo, resultando, também, em um sistema melhor concebido.

Modelo incremental

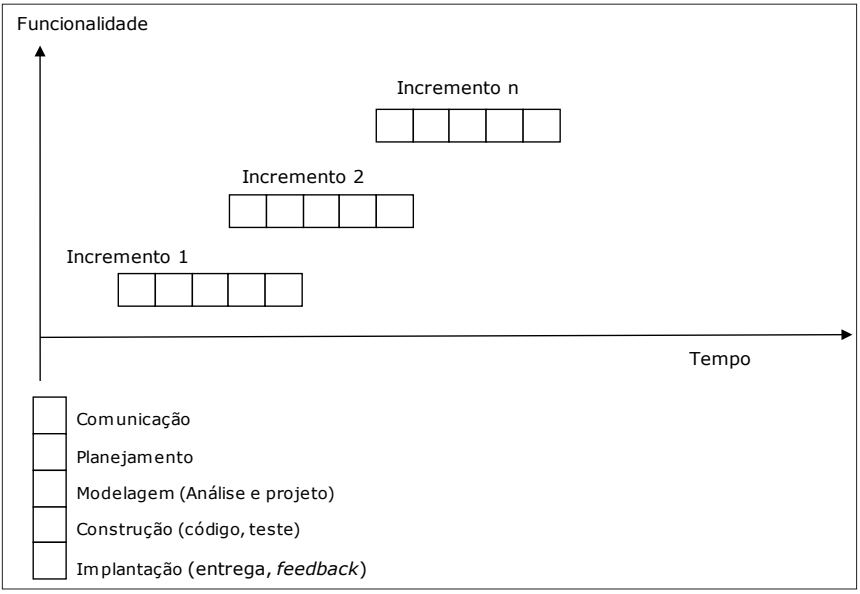
O modelo incremental também foi concebido baseado nas limitações do modelo cascata e combina as vantagens deste modelo com as do modelo prototipação.

A ideia principal deste modelo, conforme ilustrado na Figura 5, é a de que um sistema deve ser desenvolvido de forma incremental; dessa forma, cada incremento adiciona ao sistema novas capacidades funcionais, até a obtenção do sistema final. A cada passo realizado, modificações podem ser introduzidas.

Uma das vantagens dessa abordagem é a facilidade em testar o sistema durante cada fase de desenvolvimento.

A criação de uma **lista de controle de projeto** é um aspecto importante desse modelo. Essa lista deverá apresentar todos os passos a serem realizados para a obtenção do sistema final e servirá, também, para se medir, em um dado nível, a distância da última iteração.

A lista de controle de projeto gerencia todo o desenvolvimento, definindo quais tarefas serão realizadas a cada iteração. As tarefas na lista podem representar, ainda, redefinições de componentes implementados anteriormente, em razão de erros ou problemas detectados em uma eventual etapa de análise.



Fonte: Adaptada de Pressman (2006, p. 40).
Figura 3 *Modelo de desenvolvimento incremental.*

É fundamental considerar alguns aspectos desse modelo.

1) A versão inicial é, frequentemente, o núcleo do produto (a parte mais importante).

- o desenvolvimento começa com as partes do produto mais bem entendidas.
- a evolução acontece quando novas características são adicionadas conforme a sugestão do usuário.

2) O desenvolvimento incremental é importante quando é difícil, ou mesmo impossível, estabelecer a priori uma especificação detalhada dos requisitos.
3) As primeiras versões podem ser implementadas com poucas pessoas; se o núcleo do produto for bem recebido, novas pessoas participam do desenvolvimento da nova versão.
4) As novas versões podem ser planejadas de modo que os riscos técnicos possam ser administrados.

Exemplo:

- O sistema pode exigir a disponibilidade de um hardware que está em desenvolvimento e cuja data de liberação é incerta.
- É possível planejar pequenas versões de modo a evitar a utilização desse hardware.
- O *software* com funcionalidade parcial pode ser liberado para o cliente.

Modelo espiral

O modelo espiral foi proposto em 1988 e sugere uma organização de desenvolvimento em espiral (por isso tem esse nome).

Esse modelo **acopla** a natureza **iterativa** da **prototipação** com os aspectos controlados e **sistemáticos** do modelo **cascata**.

O modelo espiral é dividido em uma série de **atividades de trabalho** ou **regiões de tarefa**.

Há, tipicamente, de três a seis regiões de tarefa, observe que, na Figura 4, encontra-se um exemplo de quatro regiões definidas pelos quadrantes.

Cada ciclo na espiral inicia-se com a identificação dos objetivos e as diferentes alternativas para se atingir aqueles objetivos, assim como as restrições impostas.

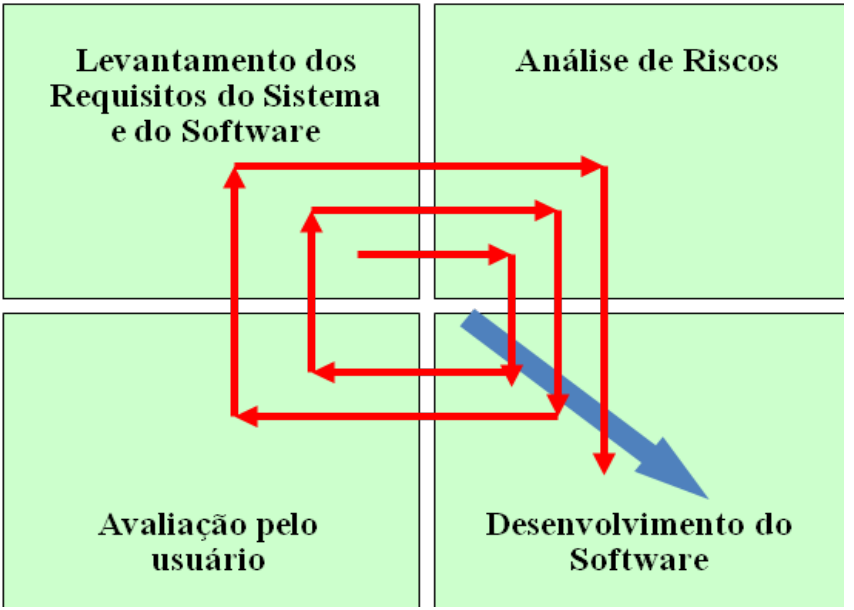
O próximo passo no ciclo é a avaliação das diferentes alternativas baseadas nos objetivos fixados, o que permitirá, também, definir incertezas e riscos de cada alternativa. No passo seguinte, o desenvolvimento de estratégias irá resolver ou eliminar as incertezas levantadas anteriormente, o que pode 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 por causa dos riscos remanescentes, como, por exemplo, decidir se os riscos relacionados ao desempenho ou à interface são mais importantes do que aqueles relacionados ao desenvolvimento do programa. Fundamentado 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 resolve boa parte dos riscos relacionados ao desempenho e à interface, então o próximo passo pode ser, simplesmente, a evolução segundo o modelo cascata.

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



Fonte: Adaptado de Pressman (1996, p. 39)
Figura 4 *Modelo espiral*.

Observações relacionadas ao modelo espiral:

1) cada "loop" no espiral representa uma fase do processo de *software*;
2) o "loop" mais interno está concentrado nas possibilidades do sistema;
3) o próximo "loop" está concentrado na definição dos requisitos do sistema;
4) o "loop" um pouco mais externo está concentrado no projeto do sistema;
5) a análise de risco engloba as melhores características do ciclo de vida clássico e da prototipação, adicionando um novo elemento;
6) a abordagem de passos sistemáticos do ciclo de vida clássico incorpora-os em uma estrutura iterativa que reflete mais realisticamente o mundo real;
7) a prototipação é utilizada em qualquer etapa da evolução do produto, como mecanismo de redução de riscos;

8) a abordagem mais realística para o desenvolvimento de *software* em grande escala na atualidade;
9) utiliza uma abordagem que capacita o desenvolvedor e o cliente a entender e a reagir aos riscos em cada etapa evolutiva;
10) dificuldade em convencer os clientes que uma abordagem "evolutiva" é controlável;
11) exige considerável experiência na determinação de riscos e depende dessa experiência para obter sucesso;
12) o modelo é relativamente novo e não tem sido amplamente utilizado;
13) a eficácia desse modelo poderá demorar muitos anos, até que possa ser determinada com certeza absoluta.

Uma característica importante desse modelo é o fato de que cada ciclo é encerrado por uma atividade de revisão, na qual todos os produtos do ciclo são avaliados, incluindo o plano para o próximo passo (ou ciclo).

Esse modelo é apropriado, especialmente, a sistemas que representem um alto risco de investimento para o cliente.

Modelo Codifica-remenda

O que, normalmente, causa o fracasso dos projetos de *software*, evidenciando, ainda, a existência da chamada Crise do *Software* é a ausência de disciplina no desenvolvimento de *software*.

Paula Filho (2001) descreve a ausência de disciplina como o paradigma "Codifica-Remenda", isto é, partindo apenas de uma especificação (ou nem isso), os desenvolvedores começam, imediatamente, a codificação, "remendando" à medida que os erros vão sendo descobertos. Enfim, nenhum processo definido é seguido.

O codifica-remenda é também conhecido por Modelo Balbúrdia, por razões óbvias...

Infelizmente, talvez este seja o paradigma mais usado, por ser considerado o mais fácil e não exigir nenhuma sofisticação téc-

nica ou gerencial. É, porém, um modelo de alto risco, difícil de ser gerenciado e que não permite assumir compromissos confiáveis.

Como já foi dito, não há um consenso sobre um modelo ideal, os modelos devem ser escolhidos e adaptados a cada necessidade. O único consenso que existe é que é necessário que a empresa que desenvolve *software* siga um método, seja qual for, pois desenvolver *software* sem uma metodologia não é mais aceitável na atual conjuntura da indústria de *software*. Nossos clientes, como os de qualquer indústria, querem produtos com qualidade. E a qualidade do produto só é alcançada por meio de um bom processo de desenvolvimento.

Recentemente, surgiram as metodologias consideradas ágeis para o desenvolvimento de *software*. Essa metodologia proporciona a satisfação do cliente e a entrega incremental do *software* logo de início. Essa metodologia propõe equipes pequenas, altamente motivadas; métodos informais; produtos de trabalhos mínimos e simplicidade no desenvolvimento. Pesquise na internet sobre as metodologias ágeis de desenvolvimento e conheça essa novíssima filosofia de desenvolvimento de *software*.

7. EXTREME PROGRAMMING

À medida que as características dos *softwares*, dos desenvolvedores e dos clientes mudam, a Engenharia de *Software* procura adaptar seus modelos à nova realidade. A metodologia *Extreme Programming*, também conhecida como XP, ou Programação Extrema, proposta por Kent Beck em 1998 (TELES, 2004), é um dos mais novos métodos da Engenharia de *Software* e é um exemplo de metodologia ágil para desenvolvimento de *software*. Por ser um modelo novo e bastante aceito, vejamos a seguir alguns de seus detalhes.

> O XP é um processo de desenvolvimento que busca assegurar que o cliente receba o máximo de valor de cada dia de trabalho da equipe de desenvolvimento. Ele é organizado em torno de conjunto de valores e práticas que atuam de forma harmônica e coesa para asseguras que o cliente sempre receba um alto retorno do investimento em software (TELES, 2004, p. 21).

O ciclo de vida do XP tem quatro atividades básicas: codificar, testar, escutar e modelar, demonstradas através de quatro valores: a comunicação, a simplicidade, o *feedback* e a coragem.

A prática do XP tem comunicação contínua entre o cliente e a equipe, com o objetivo de criar o melhor relacionamento entre o cliente e desenvolvedor, dando preferência às conversas pessoais do que através de outros meios. É incentivada também a comunicação intensa entre desenvolvedores e gerente do projeto. A comunicação entre o cliente e a equipe permite que todos os detalhes do projeto sejam tratados com a atenção e a agilidade que merecem, ou seja, codificar uma funcionalidade preocupando-se apenas com os problemas de hoje e deixar os problemas do futuro para o futuro.

A simplicidade foca sempre no mais simples que possa funcionar, visando minimizar o código, desprezando funções consideradas necessárias. O importante é ter em mente que os requisitos são mutáveis. Sendo assim, o interessante no XP é implementar apenas que é estritamente necessário.

O contato incessante com o cliente a respeito do projeto é o que se pode chamar de *feedback* constante. As informações sobre o código são dadas por teste periódico, os quais indicam erros tanto individuais quanto do *software* integrado. O cliente terá sempre uma parte do *software* funcional para avaliar.

A grande maioria dos princípios do XP são práticas de senso comum e que fazem parte de qualquer processo disciplinado. Por exemplo, simplicidade significa foco nas partes do sistema que são de alta prioridade e somente depois pensado em soluções para problemas que, até então, não são relevantes (e talvez nunca sejam devido às grandes modificações no sistema), pois o cliente aprende com o sistema que utiliza e re-avalia as suas necessidades, gerando o *feedback* para a equipe de desenvolvimento.

As equipes XP acreditam que errar é natural e quebrar o que vinha funcionando acontece cedo ou tarde. É necessário ter cora-

gem para lidar com este risco, o que em XP se traduz em confiança nos seus mecanismos de proteção. A coragem encaixa-se na implantação dos três valores anteriores. São necessários profissionais comunicativos e com facilidade de se relacionar. A coragem auxilia a simplicidade, quando a possibilidade de simplificar o *software* é detectada. Desse modo, a coragem é necessária para garantir que o *feedback* do cliente ocorra com frequência, pois esta abordagem implica na possibilidade de mudanças constantes do produto.

Enfim, o sistema é desenvolvido de forma incremental, a equipe está continuamente fazendo a manutenção do *software* e criando novas funcionalidades, por isso a equipe precisa ser corajosa e acreditar que, utilizando as práticas e valores do XP, será capaz de fazer o *software* evoluir com segurança e agilidade.

Segundo Teles (2004), a metodologia XP apresenta 13 práticas básicas, que são relacionadas entre si:

1) Cliente presente.
2) Jogo de planejamento.
3) *Stand up meeting*.
4) Programação em par.
5) *Refactoring*.
6) Desenvolvimento guiado pelos testes.
7) Código Coletivo.
8) Padrões de codificação.
9) Design Simples.
10) Metáfora.
11) Ritmo Sustentável.
12) Integração Contínua.
13) Releases Curtos.

Cliente presente

O XP trabalha com uma visão diferente dos outros modelos, preferindo que o cliente esteja presente no dia a dia do projeto. A participação do cliente contribui para o sucesso do projeto, enquanto a ausência é um sério fator de risco. Sua participação via-

biliza a simplicidade do processo. Além disso, permite-se que o projeto seja conduzido através de uma série de pequenos ajustes, e não através de mudanças bruscas ao longo do caminho.

Obviamente que o cliente não pode dedicar-se exclusivamente à equipe de desenvolvimento, esta não é a proposta da prática. A ideia é que o cliente, e outras pessoas importantes no projeto, tenham disponibilidade para ajudar os desenvolvedores sempre que tiverem dúvidas. O objetivo é fazer que o cliente seja acessível e compreenda a importância disso para o bom andamento do projeto, pois irão surgir dúvidas durante o projeto e estas precisam ser respondidas imediatamente. Isto contribui com a agilidade do processo e evita o trabalho especulativo.

A ideia da proximidade física entre equipe e cliente de desenvolvimento está relacionada ao valor da comunicação no XP. De acordo com Teles (2004), apesar da existência de vários meios de comunicação, o diálogo presencial é a forma mais eficaz, porque os interlocutores têm acesso a vários elementos da comunicação, tais como palavras verbalizadas, gestos, expressões faciais, postura, tom de voz, entre outros. No conjunto, esses elementos colaboram para a melhor compreensão das mensagens transmitidas. As expressões faciais que demonstram falta de entendimento não são visíveis por telefone. As percepções destas expressões representam um *feedback* importante, porque provavelmente farão o interlocutor explicar melhor o assunto. Por telefone não é possível observar isso e talvez a pessoa decida não verbalizar sua dúvida. Assim começam as interpretações equivocadas.

As equipes XP baseiam-se no princípio de que o resultado final de um projeto depende não apenas dos desenvolvedores, mas também do cliente e de quaisquer outras pessoas que possam ter algum tipo de contribuição para o projeto. Apesar de ser um ponto de vista aceito por muitas pessoas, vem sendo ignorado fortemente em diversos casos, especialmente nas grandes corporações, onde é comum se criar uma distância significativa entre o cliente e a equipe de desenvolvimento.

Jogo de planejamento

O XP utiliza o planejamento com o objetivo de que a equipe de desenvolvimento esteja sempre trabalhando naquilo que irá agregar maior valor para o cliente a cada dia de trabalho. Este planejamento é feito diversas vezes ao longo do projeto, para que todos tenham a oportunidade de revisar as prioridades. Assim, garantir que a equipe esteja sempre trabalhando naquilo que é o mais importante para o cliente.

Todo projeto em XP é dividido em *releases* e iterações, de modo que a equipe e o cliente tenham diversas oportunidades de se reunir para revisar o planejamento. Segundo Teles (2004), *releases* são módulos do sistema que geram um valor bem definido para o cliente. Iterações são períodos de tempo de poucas semanas, em que a equipe implementa um conjunto de funcionalidade de acordo com a necessidade do cliente.

No início de cada *release* e cada iteração ocorre o jogo do planejamento, que se trata de uma reunião em que o cliente avalia as funcionalidades que devem ser implementadas e prioriza aquelas que farão parte do próximo *release* ou da próxima iteração. No XP, as funcionalidades são descritas em pequenos cartões e são chamadas de estórias. Essas estórias devem ser escritas sempre pelo próprio cliente. Não existem regras de formato para escrevê-las, mas o cliente deve respeitar o espaço do cartão, dessa forma as estórias são sempre criadas e mantidas da forma mais simples possível, sendo o suficiente para transmitir a ideia desejada. Quando o cliente escreve uma estória ele é forçado a pensar melhor na funcionalidade, pois formaliza pensamento e analisa melhor o assunto sobre o qual ira tratar. O cliente assume uma responsabilidade sobre aquilo que está sendo solicitado.

A equipe de desenvolvimento utiliza os cartões para identificar as funcionalidades desejadas pelo cliente e escolhem as estórias que irão implementar a cada dia de trabalho. Durante o jogo do planejamento, as estórias são estimadas para que o cliente possa conhecer o custo da implementação de cada uma delas.

No XP o produto *software* é entregue de forma incremental, de modo que após cada entrega o cliente possa começar a utilizar o sistema e obter os benefícios que ele oferece, essas entregas tratam-se dos *releases*. O XP trabalha com conceito com *releases* pequenos, isto é, os *releases* têm curta duração, em torno de 2 meses (TELES, 2004). *Releases* pequenos permitem a entrega rápida de algo que tem valor para o cliente, que pode começar a usufruir dos benefícios do *software*.

No início do projeto, o cliente e a equipe devem dividir o projeto em *releases* de tamanhos fixos e pequenos, estabelecendo uma ordem. Em seguida, o cliente atribui as estórias aos *releases*. Porém, não é necessário que todas as estórias sejam distribuídas, visto que à medida que os *releases* são desenvolvidos, o cliente aprende sobre o sistema e pode gerar alterações nas estórias dos *releases* subsequentes.

O cliente poderá alterar as estórias se for necessário. Durante um *release*, ele pode criar estórias que substitua estórias existentes, e assim remover as estórias previsíveis, incorporando, desse modo, o seu aprendizado no sistema. Dessa forma, o XP simplifica a incorporação de mudanças ao projeto, que gera benefícios diretos para o cliente, essas alterações refletem o crescente aprendizado do cliente quanto ao sistema.

O XP procura dar flexibilidade ao cliente, fazendo com que a equipe trabalhe de forma eficiente. Durante todo o projeto, o cliente pode fazer mudanças nas estórias. Porém, dentro de uma iteração, e somente neste caso, o cliente terá de concordar com o que foi decidido na reunião de planejamento e, com isso, terá que esperar o fim da iteração para solicitar mudanças.

Stand up meeting

Um dia de trabalho de uma equipe XP sempre começa com *stand up meeting*, que pode ser traduzido por reunião em pé. Referindo uma reunião rápida que acontece todos os dias e que en-

volve todos os membros da equipe de desenvolvimento, fazendo o grande diferencial.

O *stand up meeting* serve para que todos da equipe comentem rapidamente o trabalho que executaram no dia anterior. Isto faz com que a equipe fique atualizada sobre o andamento do projeto naquele dia. Além disso, é também uma oportunidade de apresentar aos demais os desafios enfrentados e as soluções criadas por eles. Assim, quando os problemas semelhantes acontecerem, todos saberão o que fazer ou, no mínimo, saberão a quem procurar.

Esta reunião é realizada diariamente, normalmente de manhã, pela equipe de desenvolvimento com o objetivo de compartilhar informações sobre o projeto e priorizar suas atividades. Trata-se de um diálogo entre todos os membros da equipe, se possível envolvendo também a presença do cliente. Referindo-se ao dia anterior, o *stand up meeting* dá a oportunidade de todos os dias avaliar o trabalho. Assim, há chance de identificar pontos negativos no processo. A identificação dos pontos fracos é essencial para que possam aperfeiçoar diariamente o seus trabalhos.

No *stand up meeting* a equipe decide em conjunto o que serão abordados naquele dia e quem cuidará de cada cartão (chamado de estória). Lembrando que, no início de cada iteração, o cliente e a equipe se reúnem para o jogo do planejamento. O resultado é um conjunto de cartões contendo estórias que deverão ser implementadas na iteração.

É importante saber que a decisão sobre o que fazer ao longo do dia não é decidida por uma pessoa, mas a decisão é tomada em equipe. Quando todos se reúnem, é possível ter uma visão de todo o desenvolvimento e não apenas de uma parte, com isso é possível decidir com mais eficácia quais são as prioridades do dia. Os participantes não precisam estar em pé, embora seja aconselhável. As pessoas têm que dar importância ao conteúdo da reunião e participação ativamente. Todos têm que falar, ao menos, por alguns mi-

nutos. E, por fim, é importante que a equipe procure ser objetiva e gaste pouco tempo na reunião.

Programação em par

Os desenvolvedores não trabalham sozinhos em um projeto que utilize XP, sempre em pares. A programação em par é uma técnica na qual dois programadores trabalham em um mesmo problema, ao mesmo tempo e em um mesmo computador. Em outras palavras, uma pessoa (condutor) assume o teclado e digita os comandos que fará parte do programa, a outra (o navegador) a acompanha fazendo um trabalho de estrategista.

A programação em par é utilizada para fazer a revisão do código em par, enquanto o condutor digita, o navegador está permanentemente revisando o código e tentando evitar que eventuais erros do condutor passem despercebidos. Assim, pequenos erros que são simples de serem corrigidos no ato gastam muito tempo para serem corrigidos depois. A programação em par elimina estes erros na medida em que são detectados pelo parceiro que não estão com as mãos no teclado.

O uso da programação em par gera ideias mais eficazes e, ao mesmo tempo, mais simples. A questão da simplicidade é bastante nítida. Se um parceiro sugere uma ideia muito complicada, o outro consegue notar a complexidade, na medida em que ele terá dúvidas o outro poderá sempre sugerir simplificações, tendo como resultado final uma solução mais clara e simples. O código feito é quase livre de defeitos. Devido aos processos contínuos de revisão, pouquíssimos erros permanecem no código, sendo importante durante o projeto porque significa que a equipe gastará menos tempo futuro descobrindo defeitos.

No dia a dia de um programador, ele se depara com diversas fontes de distração: email, mensagens instantâneas, *sites* de internets, cansaço são elementos para que o programador diminua seu desempenho de trabalho no código. Estes desvios são corrigidos através da pressão do par. O programador está acompanhado de

outra pessoa, ele deixa de ter um compromisso apenas consigo mesmo. O seu compromisso aumenta e passa a envolver mais alguém.

Existe revezamento entre eles (condutor/navegador), às vezes um ou outro estão cansados e pede substituição. Portanto, o revezamento é fundamental e deve ser praticado sempre que se fizer necessário. E não se aplica o revezamento apenas à questão de quem conduz o teclado, mas também entre pares distintos. Em um projeto, é preciso que seja trocado o par diariamente, com frequência, visto que essa troca é importante para a troca de conhecimento.

Refactoring

De acordo com Teles (2004), o *refactoring* é o ato de alterar um código sem afetar a funcionalidade que ele executa. É utilizado para tornar o *software* mais simples de ser manipulado e se utiliza fortemente dos testes para garantir que as modificações não interrompam seu funcionamento.

Se um desenvolvedor tem uma dúvida em uma parte do código, este deve procurar compreendê-lo e, em seguida, reescrever de modo que fique mais legível, não alterando a funcionalidade, já que o código deve continuar fazendo a mesma coisa. O XP faz com que o código se torne mais legível dando a ele o nome de *refactoring*.

O código-fonte mal formulado cria dificuldades sérias para quem tiver a necessidade de modificar ou compreender por alguma razão. Quando o código está legível é possível modificar rapidamente. Em contrapartida, se estiver uma desordem, pode ser necessário gastar muito tempo para entendê-lo. Assim, o *refactoring* não deve afetar o comportamento original do código, deve apenas torná-lo mais legível.

O *refactoring* está diretamente relacionado ao código coletivo. Ele permite que os desenvolvedores atuem em qualquer parte do sistema, mantendo a legibilidade do código e tornando simples a prática do código coletivo, pois os desenvolvedores conseguem ler qualquer parte do código facilmente.

Quando não se dedica atenção ao *refactoring*, o código vai crescendo e se tornando mais desorganizado e ilegível a cada dia e a equipe tendo mais dificuldades em incorporar mudanças. Com o *refactoring* a equipe minimiza este problema e consegue ter agilidade e flexibilidade.

É importante saber que sem o *refactoring* um projeto em XP não tem a menor chance de sobrevivência. O XP baseia-se em agilidade de gerar *feedback* rapidamente para o cliente e só é possível fazer alterações e adição de novas funcionalidades alterando o código com facilidade e agilidade. Sem uma boa organização do projeto não é possível ser ágil e realizar as alterações necessárias no código, pois só é possível fazer isso se o projeto for organizado continuamente através do *refactoring*.

> O XP enfatiza uma continua remodelagem utilizando a refatoração a qualquer momento que seja necessário, no entanto há uma pequena valorização a documentação de modelagens detalhadas. Tipicamente os desenvolvedores jogam fora documentação de modelos após terem escrito o código, entretanto eles mantêm se pode ser útil. As equipes também passam a manter a documentação quando o cliente para de vir com novas estórias (TELES, 2004, p. 101).

Porém, o *refactoring* também tem um preço. Ele é acompanhado pelo risco de código parar de funcionar. Trata-se de um risco sério que nunca deve ser ignorado. Porém, o XP possui mais uma prática que também combate o risco de se quebrar um código que esta funcionando. Trata-se de testes automatizados que servem para informar se o código continua produzindo as mesmas respostas mesmo depois de ter sido alterado. Os testes proveem uma enorme segurança ao processo de *refactoring*.

Desenvolvimento guiado pelos testes

A ideia principal da programação guiada pelos testes é fazer com que o tempo total dedicado à depuração em um projeto seja o mínimo possível. Quando se testa, faz-se um investimento e se espera que ele gere um retorno no futuro.

Os desenvolvedores aprendem sempre, ou seja, aprendem com os testes. Quanto mais os desenvolvedores têm a oportunidade de aprender, melhor eles codificam e menos defeitos surgem do sistema. Sempre que um teste detecta uma falha, a equipe ganha tempo de desenvolvimento, pois tem a oportunidade de corrigir o problema rapidamente, o que evita longas sessões de depuração que costumam tomar boa parte do tempo dos projetos.

O XP trabalha com dois tipos de testes, sendo eles: testes de unidade e teste de aceitação. Você aprenderá detalhes sobre teste de *software* na Unidade 7 deste material.

O teste de unidade é aquele realizado sobre cada classe do sistema para verificar se os resultados gerados são corretos. Os testes de aceitação são efetuados sobre cada funcionalidade, ou estória do sistema, verificando não apenas uma classe em particular, mas a interação entre um conjunto de classes que implementam uma dada funcionalidade.

Código coletivo

Quando um projeto é implementado por uma equipe, é comum que seja dividido em partes, de forma que cada membro da equipe fique responsável por uma ou mais partes.

O XP busca eliminar as ilhas de conhecimento, pois não existe uma pessoa responsável por um subconjunto do código. Cada desenvolvedor tem acesso a todas as partes do código e total liberdade para alterar o que necessitar, ou seja, o código é coletivo. Ilhas de conhecimento nos projetos de *software* ocorrem quando uma pessoa que possui um domínio sobre uma área do projeto, sabendo que só ela conhece aquela parte ou aquele código.

O XP incentiva a propriedade coletiva do *software*, isto é,

> [...] qualquer um pode modificar qualquer parte do código a qualquer momento, fortificando a necessidade de uma integração contínua, assim como testes de regressão contínua e a programação em pares que protege contra a perda do controle das configurações (TELES, 2004, p. 146).

Assim, cada parte do *software* pode eventualmente ser observada por diferentes desenvolvedores ao longo da sua vida. A consequência disso é que o código está sempre sendo revisado. Vale ressaltar que o código coletivo leva a equipe a ser mais ágil no desenvolvimento. Como não existe uma pessoa responsável por certa parte do código, qualquer pessoa pode mexer no código, tão logo sinta a necessidade de fazê-lo. Não é necessário esperar pelo especialista.

Em compensação, o código coletivo exige coragem por parte dos desenvolvedores. Afinal, diversas vezes ao dia o desenvolvedor terá de mexer em partes do sistema que não foram construídas por ele. Será preciso coragem para fazer isso e acreditar que o sistema continuará funcionando mesmo depois das suas mudanças. A prática do código coletivo depende fortemente de duas outras: o *refactoring* e o desenvolvimento guiado pelos testes.

Padrões de codificação

Como já dito, um dos valores fundamentais do XP é a comunicação, isso é válido também ao código do sistema, sendo que a equipe deve se comunicar de forma clara através do código. Para ajudar, o XP sugere a utilização de padrões de codificação.

Como existem códigos de diferentes programadores, provavelmente percebe-se a existência de diferentes estilos. Essas diferenças dificultam a compreensão do código por todos os membros da equipe. Isso pode ser evitado através da adoção de um mesmo estilo de codificação por todos. Desta forma, qualquer programador poderá ler o código com velocidade, sem se preocupar em compreender estilos de formatação diferentes.

É importante que se procure adotar um padrão que seja fácil, simples e esteja documentado em algum lugar. Além disso, é necessário sempre negociar de modo a construir um padrão que atenda as necessidades do projeto e que seja aceito pela maioria.

Design simples

As práticas do XP são voltadas para a criação de valor para o cliente. O XP determina que a equipe deva sempre assumir o *design* mais simples possível e que seja capaz de fazer uma funcionalidade passar nos testes.

Em um projeto de *software* a mudança é constante, pois requisitos, design, tecnologia, equipe e os membros mudam. O problema não está na mudança em si, mas na incapacidade de lidar com a mudança. No decorrer dos anos, a Engenharia de *Software* dedicou-se para reduzir os custos de uma alteração de *software*, através da melhoria das linguagens de programação, da tecnologia de banco de dados, das práticas de programação, novos ambientes e ferramentas de desenvolvimento e novas anotações.

De acordo com Teles (2004), para se trabalhar com um *design* simples, o XP recomenda que o desenvolvedor utilize a seguinte estratégia:

1) Comunicação: um *design* simples comunica a sua intenção de forma mais eficaz que um *design* complicado. Isto ocorre porque é mais fácil compreendê-lo. Geralmente, o *design* precisa ser simples, mas também deve ser capaz de comunicar todos os aspectos importantes do *software*.
2) Simplicidade: um *design* simples torna o *software* mais leve e mais fácil de ser alterado a qualquer momento, visto que é mais fácil de ser compreendido.
3) *Feedback*: criando um *design* simples, a equipe é capaz de avançar de forma mais ágil e obter *feedback* do cliente mais rapidamente.
4) Coragem: a equipe trabalha o *design* apenas até o ponto em que ele resolve o problema de hoje. Problemas do futuro, mesmo que já sejam previstos, são deixados para o futuro. É necessário coragem para assumir que a equipe será capaz de resolver o problema no futuro caso ele realmente se manifeste.

Ao longo das iterações, o *design* precisa evoluir, mas deve manter-se simples e claro para que a equipe possa fazer alterações no *software* a qualquer momento, com facilidade. Por esta razão, o *refactoring* tem um papel fundamental no *design* do XP.

Metáfora

Muitas vezes, alguém está explicando algo e não há nada que faça o interlocutor compreendê-lo. Depois de algumas tentativas, a pessoa faz uma comparação daquele assunto com algo que o interlocutor conhece bem. Neste momento, em um instante, consegue entender toda a explicação simplesmente porque a outra pessoa utilizou um artifício que é chamado de metáfora.

O XP utiliza a metáfora para comunicar diversos aspectos do projeto. É necessário usar a metáfora para criar uma visão comum sobre o projeto em si. A equipe deve buscar uma metáfora que permite explicar as ideias centrais do projeto de forma simples e objetiva, passando a ser capaz de se comunicar com o cliente e com outros envolvidos no projeto de forma muito mais eficaz.

Falar da metáfora é muito fácil, o problema é criá-la. Desenvolver uma metáfora é um exercício de criatividade e abstração. Segundo Teles (2004), metáfora, muitas vezes, é olhar para os lados e pensar nas coisas mais absurdas, tentando relacioná-las com o seu problema ou o seu projeto. Às vezes deixa a mente livre de pensamentos e, de repente, aparece uma ideia, um estalo. Este estalo, o resultado de um complexo processamento no subconsciente, é um elemento de grande importância no desenvolvimento do *software*. Normalmente, um simples estalo pode economizar meses de trabalho.

A utilização da metáfora é uma prática que anda de mãos dadas com ritmo sustentável. Por isso que os desenvolvedores devem chegar descansados todos os dias. Pessoas descansadas produzem melhor, têm ideias eficazes, são mais atentas e poupam tempo do desenvolvimento, ou seja, não é vantagem trabalhar mais, mas sim trabalhar de forma mais eficaz e inteligente.

Ritmo sustentável

O XP recomenda que ritmos sustentáveis sejam mais produtivos, e por isso não seja utilizada hora extra por duas razões: permitir que os desenvolvedores trabalhem apenas oito horas por dia é mais humano e demonstrar respeito com a individualidade e a integridade de cada membro da equipe aumenta a agilidade do projeto.

Programadores exaustos cometem mais erros, assim, a equipe que não trabalha por um tempo excessivo se mantém mais efetiva. Astels (2002, p. 35) ilustra a necessidade de equilíbrio entre a carga de trabalho e a de descanso quando recomenda "trabalhe a 100% durante as 40 horas e descanse a 100% no resto. Se algum deles não tiver foco feito com 100%, um afetará o outro".

O *Extreme Programming* compreende a natureza humana do desenvolvimento de *software* e percebe que trabalhar mais horas significa diminuir o ritmo de produção, significa atrasar o projeto, por mais estranho que isto possa parecer no início.

Assim, a adoção do ritmo sustentável é um desafio a ser vencido. As equipes que conseguiram adotar esta prática colhem frutos dela. Seus projetos são mais ágeis, as pessoas convivem de forma mais harmoniosa e não precisam sacrificar a vida pessoal em nome do trabalho.

Integração contínua

Quando uma nova funcionalidade é incorporada ao sistema ela pode afetar outras que já estavam implementadas e a integração contínua entra em prática levando os pares a integrarem seus códigos com o restante do sistema diversas vezes ao dia. Quando o par faz isso ele executa todos os testes para assegurar que a integração tenha ocorrido de forma satisfatória por outra pessoa para que o problema seja resolvido.

O XP propõe uma estratégia de integração diferente. Ao invés de esperar muito tempo para integrar as partes do sistema, a equipe deve integrar com a maior frequência possível. Normalmente, uma equipe XP deve integrar e testar novamente o sistema inteiro diversas vezes por dia. Essa prática é chamada de integração contínua (JEFFRIES, 2001).

Uma das dificuldades de se integrar diversas vezes ao dia diz respeito ao tempo necessário para fazer o *build* de todo o sistema. Em tempo precisa ser baixo e precisa ser mantido baixo ao longo de todo o projeto. Existem diversas formas de fazer isso, como por exemplo:

- Fazer com que a interdependência entre os arquivos seja baixa.
- Fazer com que o compilador só compile os arquivos que tiverem sido editados e utilize diretamente o código objeto para os demais.
- Trabalhar com a carga dinâmica de bibliotecas quando isso for possível.

Se o *build* começa a ficar lento, a equipe precisa tratar logo da questão e descobrir uma forma de manter o *build* rápido. Isso é uma prioridade para o desenvolvimento. O *build* tem que ser executado rapidamente, assim como os testes de unidade. Isso faz com que a equipe possa avançar com agilidade e a motiva a integrar várias vezes por dia, o que diminui a possibilidade de erros no processo de integração.

O código coletivo permite que a equipe avance mais rapidamente porque não é necessário esperar por outra pessoa para que o problema seja resolvido. O código permanece mais claro, pois é revisado sempre por diversos programadores e eles não precisam contornar áreas problemáticas do sistema.

Para que faça a integração contínua, serão necessárias duas categorias de ferramentas: ferramenta de *build* e sistema de controle de versão, o qual é chamado simplesmente de repositório.

A ferramenta de *build* é utilizada para compilar todo o sistema e executar outras atividades que se façam necessárias para que ele possa ser executado, ajudando os desenvolvedores a ganharem agilidade para que possam acontecer suas mudanças, gerarem o sistema imediatamente e testá-lo.

O repositório armazena todos os códigos fonte do sistema mantendo a versão deles. Trata-se de uma ferramenta essencial no processo de integração, pois quando utilizado pode fazer uma diferença enorme na produtividade da equipe.

Segundo Jeffries (2001), o processo de integração é feito correto quando ele não obstrui o avanço da equipe. Deve ser fácil e rápido obter o código fonte do qual o desenvolvedor necessita e armazenar as suas mudanças. O repositório deve detectar conflitos e é importante que seja simples resolvê-los. Não deve esperar, isto é, se um par precisa editar alguma coisa, ele deve seguir adiante sem que nada o impeça.

Releases curtos

O *Extreme Programming* tem como objetivo gerar um fluxo contínuo de valor para o cliente, sendo que a equipe produz um conjunto reduzido de *software* no dia a dia e se beneficiam dele cada vez mais incorporando mais funcionalidades e gerando mais valor. O XP recomenda que o projeto faça pequenos investimentos gradualmente e busque grandes recompensas o mais rapidamente possível. Para isso acontecer, o XP trabalha com *releases* curtos.

O XP sugere que a equipe de desenvolvimento implemente as funcionalidades mais importantes o quanto antes. Ou seja, o cliente deve selecionar o conjunto de funcionalidades que poderá gerar o valor mais alto mais rapidamente. Assim, os primeiros *releases* deverão fornecer o máximo de valor, de modo que o cliente consiga obter o máximo de retorno rapidamente.

Trabalhando com *releases* curtos, o cliente tem oportunidade de avaliar o projeto rapidamente e com frequência. Tendo a

oportunidade de tomar uma decisão informada: continuar o projeto ou suspendê-lo para evitar perdas ainda maiores (em certos casos). Além disso, na medida em que os clientes começam a obter retorno do investimento mais cedo, ele tem mais chances de obter o retorno total ou, no mínimo, de reduzir eventuais perdas.

A abordagem tradicional é mais arriscada que a do XP, pois os usuários não têm a chance de avaliar o *software* sempre, e o projeto só começa a gerar receita após a conclusão.

Segundo Brooks (1995), é comum o cliente fazer um grande investimento na construção de um *software* e este simplesmente não atender as suas necessidades levando a lançar um novo projeto para a construção de um *software*, que assim irá atender aos anseios dos usuários.

Por todas essas razões, os projetos em XP trabalham com *releases* que não consumam mais que dois meses. Isso permite incorporar uma boa quantidade de funcionalidade em cada *release* e permite que os usuários aprendam com o *software* com bastante frequência.

O XP está sendo aplicado em larga escala em vários projetos no mundo todo, porém ainda temos muito a evoluir em sua compreensão e aplicação. Nota-se isso principalmente em pontos polêmicos como testes unitários, programação em dupla, rodízio de pessoas, propriedade coletiva do código e otimização de jornadas, que são práticas que se mal utilizadas podem realmente trazer aumentos no custo e no prazo de projetos. Ou seja, é de extrema importância que se entenda bem a essência de XP e, principalmente, que se tenha disciplina e criatividade, duas qualidades básicas em quem pretende usá-la em projetos. Somente a partir de uma visão criativa sobre a metodologia e uma disciplina equilibrada para cumpri-la é que todos poderão usar e ter benefícios através de *Extreme Programming*.

8. QUESTÕES AUTOAVALIATIVAS

Sugerimos que você procure responder, discutir e comentar as questões a seguir que tratam da temática desenvolvida nesta unidade. Se você encontrar dificuldade em respondê-las, procure revisar os conteúdos estudados para sanar as suas dúvidas. A autoavaliação pode ser uma ferramenta importante para você testar seu desempenho. Este é um momento ímpar para você fazer uma revisão desta unidade. Lembre-se de que no ensino a distância a construção do conhecimento se dá de forma cooperativa e colaborativa, portanto compartilhe com seus colegas suas descobertas.

1) Um analista foi contratado para desenvolver um sistema de pesquisa de DVDs em lojas virtuais. O sistema deverá solicitar ao usuário um título de DVD, que será usado para realizar a pesquisa nas bases de dados das lojas conveniadas. Ao detectar a disponibilidade do DVD solicitado, o sistema armazenará temporariamente os dados das lojas (nome, preço, data prevista para entrega do produto) e exibirá as informações ordenadas por preço. Após analisar as informações, o cliente poderá efetuar a compra. O contratante deverá testar algumas operações do sistema antes de ele ser finalizado. Há tempo suficiente para que o analista atenda a essa solicitação e efetue eventuais modificações exigidas pelo contratante. Com relação a essa situação, julgue os itens a seguir quanto ao modelo de ciclo de vida.

   I – O entendimento do sistema como um todo e a execução sequencial das fases sem retorno produzem um sistema que pode ser validado pelo contratante.

   II – A elaboração do protótipo pode ser utilizada para resolver dúvidas de comunicação, o que aumenta os riscos de inclusão de novas funcionalidades não prioritárias.

   III – A definição das restrições deve ser a segunda fase a ser realizada no desenvolvimento do projeto, correspondendo à etapa de engenharia.

   IV – Um processo iterativo permite que versões progressivas mais completas do sistema sejam construídas e avaliadas.

   Estão corretos apenas os itens.

   a) I e II.
   b) I e III.
   c) II e III.
   d) II e IV.
   e) III e IV

Fonte: ENADE 2008 – Tecnologia em Análise e Desenvolvimento de Sistemas.

2) À medida que se avança pelo modelo ocorre uma iteração e o *software* evolui para estágios superiores, normalmente com aumento da complexidade.

Cada iteração está provida das atividades determinadas pelos quadrantes planejamento, avaliação de alternativas e riscos, desenvolvimento do *software* e avaliação do cliente. No ciclo de vida de desenvolvimento de *software*, trata-se da propriedade do modelo:

a) Cascata.
b) Incremental.
c) Espiral.
d) Prototipação.
e) Balbúrdia.

Fonte: Prova aplicada em 03/2010 para o concurso do(a) TRT - 20ª REGIÃO (SE) - 2009, realizado pelo órgão/instituição Tribunal Regional do Trabalho da 20ª Região - Sergipe, área de atuação Jurídica, organizada pela banca FCC, para o cargo de Analista Administrativo - Informática, nível superior, área de formação Tecnologia da Informação .

Gabarito

Depois de responder às questões autoavaliativas, é importante que você confira o seu desempenho, a fim de que possa saber se é preciso retomar o estudo desta unidade. Assim, confira, a seguir, as respostas corretas para as questões autoavaliativas propostas anteriormente:

1) (d)

2) (c)

9. CONSIDERAÇÕES

Chegamos ao final do estudo da Unidade 2, na qual você teve a oportunidade de aprender o que é um processo de *software* e perceber a importância de ter um processo definido e que o processo utilizado para desenvolver um *software* influencia diretamente na qualidade do produto desenvolvido.

Você verificou, ainda, que a Engenharia de *Software* propõe vários modelos para o desenvolvimento de sistemas com o objetivo de minimizar custo, tempo e maximizar qualidade.

Percebeu, também, que há inúmeros modelos e a escolha de qual utilizar está diretamente relacionada às características específicas do sistema a ser desenvolvido.

Na próxima unidade, você poderá verificar que, além de ter um processo de *software* bem definido, é necessário que se tenha mecanismos para gerenciar o processo para que se atinja o objetivo de um *software* com qualidade.

10. REFERÊNCIAS BIBLIOGRÁFICAS

ASTELS, David; MILLER, Granville; NOVAK, Miroslav. *Extreme Programming*: Guia prático. Rio de Janeiro, Campus, 2002.

PAULA FILHO, W. P. *Engenharia de Software*: fundamentos, métodos e padrões. 2. ed. Rio de janeiro: LTC, 2001.

PRESSMAN, R. S. *Engenharia de software*. São Paulo: Makron Books, 1996.

_____. *Engenharia de software*. São Paulo: Makron Books, 2006.

SOMMERVILE, I. *Software engineering*. 4th ed. Addison: Wesley, 1994.

_____. *Engenharia de Software*. 6. ed. São Paulo: PearsonAddison Wesley, 2005.

TELES, V. M. *Extreme Programming*. São Paulo: Novatec, 2004.

EAD

Planejamento e Gerenciamento de Projetos

3

1. OBJETIVOS

- Analisar e discutir os conceitos relacionados à especificação de requisitos.
- Compreender a importância do gerenciamento de projetos para o sucesso da Engenharia de *Software*.
- Compreender a importância e a relação existente entre o planejamento e a gerência de projetos de *software*.

2. CONTEÚDOS

- Medições em *software*.
- Gerenciamento de Projetos.
- Estimativa de *software*.
- Ferramentas para gerência e planejamento de *software*.
- Análise de risco.
- Documentação – plano de *software* e plano de projeto.

3. ORIENTAÇÕES PARA O ESTUDO DA UNIDADE

Antes de iniciar o estudo desta unidade, é importante que você leia as orientações a seguir:

1) Durante as duas unidades anteriores, você pôde perceber que o desenvolvimento de *software* envolve inúmeras tarefas. Estas tarefas precisam ser executadas para que se produza um *software* que atenda às necessidades do cliente.
2) À medida que se expandiram as aplicações para *software*, este passou a ser utilizado nas mais variadas áreas. Um dos desafios da Engenharia de *Software* é aumentar a produtividade dos processos de desenvolvimento para que a indústria de *software* consiga suprir a crescente demanda por novos *softwares*. Porém, além de focar na velocidade dos processos de *software*, é importante que os produtos entregues tenham a qualidade esperada.
3) Para que um projeto seja bem-sucedido, é importante que este seja planejado e gerenciado. Isto vale para todos os projetos, para *software* não seria diferente.
4) Durante seus estudos, você deve considerar os conceitos de planejamento e gerenciamento de projetos. Tenha disciplina em seus estudos, programe-se, tenha metas a serem alcançadas semanalmente e procure atingi-las.
5) Seu aprendizado só depende de você. Dedique-se, interaja com seus colegas e não acumule dúvidas. Consulte o glossário sempre que necessário e peça auxílio a seu tutor.
6) É importante que você conheça, também, alguns métodos de estimativa de tempo para execução de determinadas atividades. Dentre elas, podemos citar o Método do Caminho Crítico (COM – *Critical Path Method*), que é bastante utilizado. No entanto, devido à carga horária de nossa disciplina, não teremos como o abordar. Sugiro que você pesquise na bibliografia citada, em *sites*, em livros e em periódicos para conhecer esse assunto.
7) Sugiro que você faça uma pesquisa sobre os *softwares excel da microsoft ®*, *netoffice*, *ms-project da microsoft ®*, *planner,*

dotproject, granttproject e, se considerar necessário, faça o *download* (alguns *softwares* são *free*). É importante que você realize a pesquisa para conhecer a funcionalidade de ferramentas como essa e como se relacionam na gestão de projetos de *software*.

8) Faça uma pesquisa sobre as ferramentas que utiliza o Diagrama de Gantt como recurso acoplado ou independente e verifique as principais características.

4. INTRODUÇÃO À UNIDADE

Na Unidade 2, você teve a oportunidade de estudar os modelos de desenvolvimento de *software* e de compreender que optar por um deles é considerar características específicas do *software* e dos requisitos e, especialmente, planejar o desenvolvimento – que inclui custo, tempo, investimento, pessoas etc.

Nesta unidade, você terá a possibilidade de entender o processo estudado na unidade anterior e de conhecer algumas técnicas que o irão auxiliar nesse sentido.

Não importa o tamanho ou a funcionalidade de seu *software* – atividades de análise, planejamento e gerência nunca deverão ser ignoradas ou menosprezadas, pois delas pode depender o sucesso de seu projeto.

Quando pensamos em executar alguma atividade em nossa vida, seja uma viagem, arrumar o guarda-roupa ou levar o carro para lavar, temos, inicialmente, de **planejar** e, posteriormente, de **gerenciar** se o planejado foi executado dentro do esperado.

Construir um *software* é uma tarefa semelhante.

> O tempo é o bem mais valioso que está disponível a um Engenheiro de Software.
>
> Se houver tempo disponível, um software pode ser adequadamente analisado, uma solução pode ser compreensivamente planejada, o código fonte cuidadosamente implementado e testado. O problema é que Nunca há tempo suficiente... (PRESSMAN, 1996, p. 129).

É importante que o desenvolvedor de *software* aprenda a conviver com a pressão de tempo. Nossos clientes querem sempre uma solução "para ontem", muitas vezes estes estão tendo prejuízos financeiros devido à ausência de um sistema computadorizado. Assim, para conseguir entregar um produto dentro do menor prazo possível e com a qualidade adequada às necessidades do cliente, é fundamental a organização das atividades e gerenciamento das mesmas.

São inúmeras as causas desta pressão por tempo. Pressman (1996) descreve algumas delas: grande parte da pressão nos prazos é causada por prazos finais arbitrários e irreais, definidos por pessoas que não estão plenamente envolvidas com o desenvolvimento, como, por exemplo, o próprio cliente. Porém, muitas vezes a pressão é criada pelos próprios desenvolvedores, quando o planejamento é feito casualmente, não são considerados os riscos, a equipe não é treinada para agilizar o processo. Além disso, muitas vezes a pressão é causada pela ausência de planejamento.

5. ACOMPANHAMENTO DO PROCESSO

A atividade de gerenciamento e planejamento de um projeto de *software* pode ser percebida como uma **atividade guarda-chuva**, pois abrange todo o **processo** de desenvolvimento, possibilitando **compreender** o escopo do trabalho, os riscos, os recursos exigidos, as tarefas a serem executadas, a programação (cronograma) a seguir, o esforço despendido, dentre outras.

Segundo Pfleeger (2004, p. 63), "um software somente é útil se realizar uma função desejada ou fornecer um serviço necessário". Por isso um projeto, geralmente, começa quando um cliente faz um pedido de um sistema ou quando, por meio da percepção visionária de alguém, um *software* começa a ser idealizado, pensado, projetado.

Imagine uma situação na qual uma pessoa lhe procure para você construir um *software*. Esse *software* deverá efetuar a matrí-

cula de alunos de uma escola que atende desde o maternal até o ensino médio; é preciso, também, que os pais, por meio da *web*, tenham acesso a algumas informações específicas.

É necessário que você tenha uma reunião com seu cliente para esclarecer alguns aspectos. Provavelmente, ao final da reunião, você teria de responder algumas questões para seu cliente, dentre elas:

- Você entendeu o que eu quero?
- Você pode projetar esse sistema?
- Quanto tempo você levaria para projetá-lo?
- Quanto vai custar o desenvolvimento "disso"?

Responder às duas últimas questões requer um cronograma de projeto bem planejado.

> Um cronograma de projeto descreve o ciclo de desenvolvimento de software para um projeto específico, enumerando as etapas ou estágios e dividindo cada um deles em tarefas ou atividades a serem realizadas. O cronograma também retrata as interações entre essas atividades e estima o tempo necessário para a realização de cada tarefa ou atividade. Portanto, o cronograma é uma linha de tempo que mostra quando as atividades começarão e terminarão, e quando estarão prontos os produtos relacionados ao desenvolvimento (PLFEEDGER, 2004, p. 66).

Como podemos observar, tudo se inicia com um projeto e, a partir dele, definem-se o tempo e os valores.

Para se desenvolver um projeto, é necessário definir o processo de análise e, a partir dele, a definição de atividades e marcos.

Mas o que é atividade e marco?

Às vezes, ouvimos a expressão "isso foi um marco na vida dele", mas o que isso significa?

Podemos definir **atividade** como uma parte do projeto que acontece ao longo de determinado período, enquanto **marco** é a finalização de uma atividade, ou seja, um momento específico no tempo que pode ou não marcar o início de outra atividade.

Ao pensar em um projeto como um todo, podemos (e devemos) considerar que seu desenvolvimento é o resultado de uma sucessão de fases. Cada fase, para ser executada, possui etapas e cada uma delas possui uma ou várias atividades inseridas, como demonstra a figura a seguir.

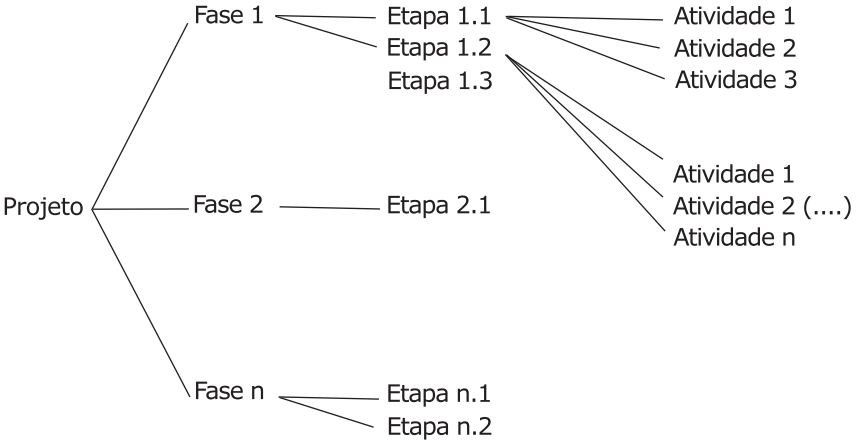


Figura 1 *Ilustração de um projeto dividido em fases e etapas.*

Para ilustrarmos a Figura 1, pense no desenvolvimento do projeto solicitado no início desta unidade – um sistema escolar. Esse sistema deverá ser dividido em grandes fases como, por exemplo, análise, codificação e teste.

Na **análise**, definiremos algumas etapas – falar com o cliente, analisar os dados e requisitos coletados, modelar o sistema dentre outras.

A etapa "falar com o cliente" pode definir algumas atividades, como, por exemplo, marcar uma reunião, definir as perguntas a serem realizadas, anotar as solicitações dos clientes etc.

A definição das atividades permite que os marcos sejam gerados. As atividades e os marcos funcionarão como referências para acompanhar o desenvolvimento ou a manutenção do projeto.

Em cada atividade, quatro parâmetros devem ser definidos:

- o precursor;
- a duração;
- a data prevista;
- o encerramento.

Podemos dizer que um **precursor** é um evento, ou um conjunto de eventos, que deve ocorrer antes da atividade começar. Ele descreve uma série de condições necessárias para que a atividade tenha início. A **duração** é o tempo necessário para completar a atividade. A **data prevista** é aquela em que a atividade deverá ser concluída, em geral, os prazos são determinados por meio de contratos. O **encerramento**, um marco ou um requisito que foi concluído.

Baseando-se nesses parâmetros é possível começar a definição de uma **estrutura de divisão do trabalho**.

O conceito de divisão de trabalho, nesse contexto, está relacionado ao momento em que são montadas as equipes de análise, de desenvolvimento, de análise de estimativas, de testes e de manutenção **para determinado projeto** e as responsabilidades começam a ser oficialmente definidas.

Cada projeto possuirá equipes diferentes, de acordo com sua especificidade e com a competência de cada pessoa que trabalha na empresa.

Essa equipe pode ser alocada para participar durante todo o projeto ou apenas em algumas fases; por exemplo, a equipe de analista pode participar apenas na fase inicial do projeto na qual é definido os requisitos e modelado o sistema, e, ao final dessas funções, a equipe de analista pode desligar-se do projeto “A” e ser realocada para o projeto “B”.

Os itens (precursor, data, divisão do trabalho etc.), ao serem planejados, darão origem à documentação do projeto que servirá de “mapa” para o gerenciamento do desenvolvimento: o plano do projeto.

Pressman (2006) descreve os princípios que se aplicam a qualquer planejamento, independentemente de seu rigor:

1) Princípio número 1 - Entenda o escopo do projeto: é impossível usar um roteiro se você não souber onde está indo. O escopo oferece à equipe de *software* um destino.
2) Princípio número 2 – Envolva o cliente na atividade de planejamento: o cliente define prioridades e oferece restrições de projeto. Para acomodar tais realidades, os desenvolvedores de *software* precisam negociar a ordem de entrega, prazos e outros tópicos relacionados ao projeto.
3) Princípio número 3 – Reconheça que o planejamento é iterativo: um plano de projeto nunca é gravado na pedra, isto é, quando se inicia o trabalho é provável que as coisas se modifiquem. Assim, o plano deve ser adaptado a estas mudanças.
4) Princípio número 4 – Estime com base no que é sabido: a intenção de uma estimativa é indicar o esforço, o custo e a duração das tarefas com base em um entendimento atual da equipe quanto ao trabalho que deve ser feito. Se a informação não for suficiente ou não for confiável, as estimativas serão também não confiáveis.
5) Princípio número 5 – Considere riscos à medida que você define o plano: se a equipe definir os riscos que tem grande impacto e alta probabilidade de ocorrerem, é necessário um plano de contingência.
6) Princípio número 6 – Seja realista: em um dia de trabalho, sempre ocorrem os ruídos, ou seja, omissões e ambiguidades. Logo, modificações ocorrerão, pois mesmo os melhores profissionais cometem erros. Essas e outras realidades devem ser consideradas quando um plano for estabelecido.
7) Princípio Número 7 – Ajuste a granularidade à medida que você define o plano: a granularidade trata-se do nível de detalhes que é introduzido à medida que um plano de projeto é desenvolvido. O planejamento de atividades a serem executadas a curto prazo é apresentado em granularidade fina, isto é, com riqueza de detalhes.

Já tarefas a serem executadas a longo prazo são planejadas com granularidade grossa, ou seja, oferece tarefas de trabalho mais amplas. Logo, a granularidade move-se de fina para grossa à medida que a linha do tempo se afasta da data atual. Atividades que não vão ocorrer nos próximos meses não exigem granularidade fina, pois muita coisa pode mudar.

8) Princípio número 8 - Defina como você pretende garantir a qualidade: o plano deve identificar como a equipe de *software* pretende garantir a qualidade, se serão utilizadas revisões técnicas formais, ou outras formas de acompanhamento devem ser explicitados no plano. Na Unidade 6 você conhecerá detalhes relacionados a esta atividade.

9) Princípio número 9 - Descreva como você pretende acomodar mudanças: diante de modificações descontroladas, mesmo os melhores planejamentos podem ser comprometidos. Assim, a equipe de *software* deve identificar como as modificações devem ser acomodadas à medida que o desenvolvimento progride. Na Unidade 5 você estudará detalhes do como pode ser realizado a Gestão de Configuração.

10) Princípio número 10 – Acompanhe o plano com frequência e faça ajustes quando necessário: é importante acompanhar um projeto diariamente, identificando áreas problemáticas e situações em que o que foi planejado se destorce da realidade. Quando um desvio for encontrado, o plano deve ser ajustado.

O que significa gerenciar?

Gerenciar significa “cuidar” de:

- pessoas: que participa do processo;
- produtos: que será desenvolvido;
- processos: do “como será feito;
- projetos: das fases, etapas e atividades.

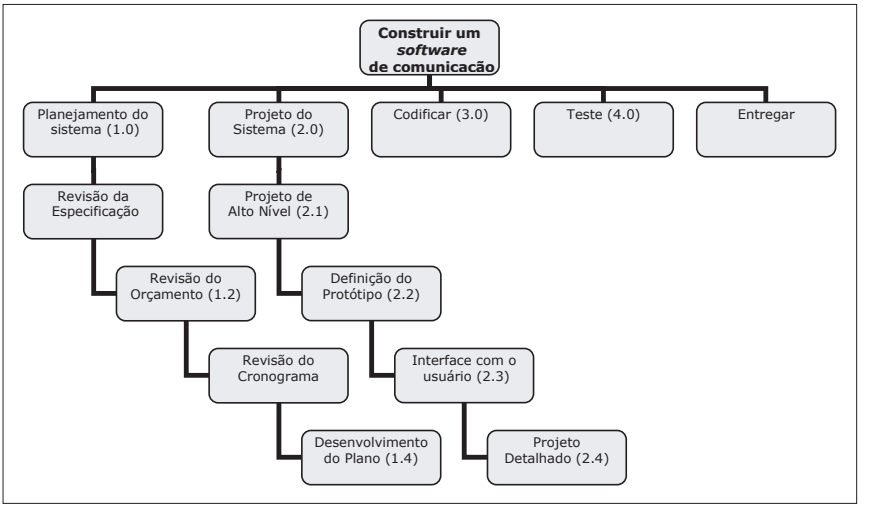
O gerente que esquece que o trabalho da Engenharia de Software está intensamente relacionado com a atividade de gerenciar pessoas nunca terá sucesso na atividade de gerência de projetos. O gerente que não incentivar uma compreensiva comunicação com o cliente, logo de início do projeto, corre o risco de construir uma solução não muito boa para um problema errado.

6. FERRAMENTA PARA ACOMPANHAR O PROCESSO

Há inúmeras ferramentas que podemos utilizar para acompanhar o projeto, algumas são mais simples, como, por exemplo: *Excel da Microsoft ®*, *Netoffice (LINUX)*, *Ms-Project da Microsoft ®*, *Planner*, *DotProject*, *GranttProject* etc.

Para verificar qual das ferramentas pode ser utilizada, uma boa dica é analisar a estrutura do trabalho, o projeto em si e, depois, optar por uma ou pela combinação de diferentes ferramentas.

Considere a estrutura de divisão de trabalho conforme a Figura 2, a seguir:



Fonte: Adaptada de Pfleeger (2004, p. 72).
Figura 2 *Exemplo de uma divisão do trabalho.*

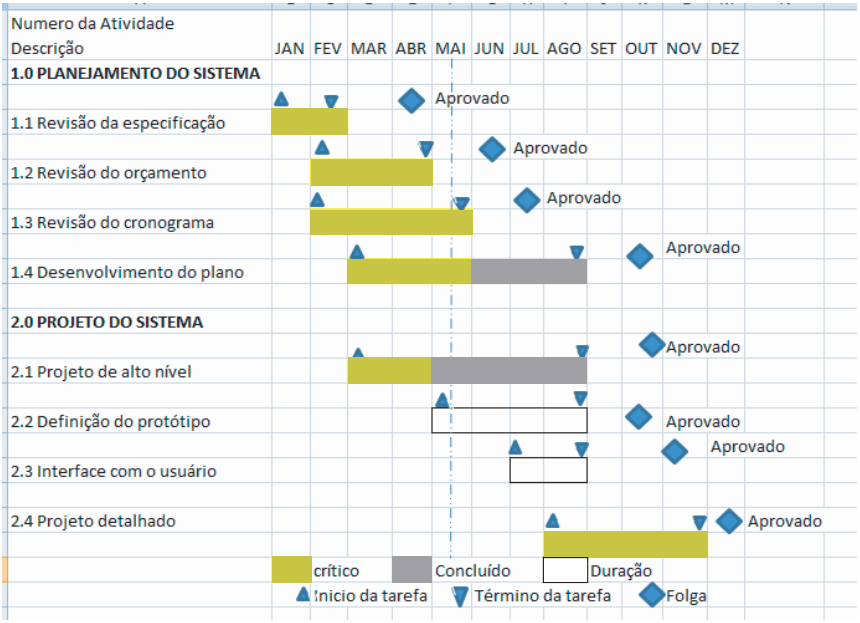
Com base na divisão do trabalho, podemos discutir, com mais precisão o tempo e o gerenciamento do desenvolvimento do projeto.

Algumas ferramentas podem ser utilizadas para visualizar e gerenciar o projeto. Uma das mais comuns é o chamado **Diagrama de Gantt**. Neste diagrama, as atividades são mostradas em paralelo, com o grau de conclusão indicado por uma cor ou um ícone. Ele

auxilia o gerente a perceber quais atividades podem ser executadas simultaneamente e quais são críticas para o cumprimento das etapas. A utilização correta de uma ferramenta que produza um **Diagrama de Gantt** pode ser fundamental para o cumprimento de etapas, fases e atividades dentro do prazo e custo estipulado.

Uma ferramenta que é bastante utilizada e que produz o Diagrama de Gantt é o *MS-Project* ® da *Microsoft*.

A seguir, observe o diagrama de Gantt, a partir da divisão do trabalho mostrada na Figura 2.



Fonte: Adaptada de Pfleeger (2004, p. 72).
Figura 3 *Diagrama de Gantt para o exemplo da Figura 2.*

7. MEDIÇÕES

Você consegue imaginar uma engenharia sem números? Como já vimos, nem tudo na Engenharia de *Software* pode ser traduzido em números. Mas muita coisa pode, estas são as medidas na Engenharia de *Software*.

Pressman (1995) indica algumas razões para que o *software* seja medido:

a) indicar a qualidade do produto;
b) avaliar os benefícios de novos métodos;
c) formar uma linha básica para estimativas;
d) justificar pedidos de novas ferramentas.

Há algumas classificações para as medidas na Engenharia de *Software*, dentre estas, as medidas podem ser: diretas ou indiretas.

- **Medidas diretas:** são obtidas contando algo ou utilizando algum instrumento para realizar uma medição.
- **Medidas indiretas:** são obtidas submetendo medidas diretas a algum cálculo.

Você entendeu? Vamos explicar melhor!

Vamos imaginar estas classificações na engenharia civil, veja que:

Medida direta: altura de uma parede (para conseguir esta medida, você pode usar uma fita métrica ou uma trena - instrumento de medida).

Medida indireta: produtividade de um pedreiro (conseguida, por exemplo, pela quantidade de metros quadrados de parede que ele constrói em um dia de trabalho).

Exemplos de medidas na Engenharia de *Software*?

Medidas diretas: quantidade de linhas de um código-fonte, tempo de execução de um programa, tamanho de espaço na memória, quantidade de páginas de documentação, custo do *software*, tempo gasto no desenvolvimento, quantidade de defeitos encontrados etc.

Medidas indiretas: complexidade, qualidade, produtividade, manutenibilidade, corretitude etc.

Para Pressman (2006), a medição é essencial quando se quer alcançar qualidade, oferecendo a visão necessária para criar modelos efetivos de análise, projeto, codificação, testes e manutenção.

Um processo de medição pode ser caracterizado por cinco atividades, segundo Pressman (2006):

a) **Formulação:** escolha da medida adequada para o *software* em questão.
b) **Coleta:** mecanismo usado para acumular dados para derivação das métricas.
c) **Análise:** cálculo das métricas e aplicações de ferramentas da matemática.
d) **Interpretação:** avaliação das métricas.
e) **Realimentação:** recomendações adquiridas durante a interpretação devem ser transmitidas à equipe de desenvolvimento para a melhoria das medidas futuras.

Medidas são, ainda, uma polêmica na Engenharia de *Software*, pois, muitas vezes, não sabemos o que medir, como medir, e, uma vez obtidas as medidas, não sabemos como utilizá-las para melhorar a qualidade do produto e a produtividade do processo. Este panorama, porém, vem mudando com o avanço da Engenharia de *Software*.

As medidas são importantes para se ter referências numéricas para fazer as estimativas. Ao registrar uma medida, é possível manter **dados históricos** de projetos do passado, para servir de base às estimativas de projetos futuros.

Segundo Pressman (1995), as informações históricas permitem:

a) Repetir o que deu certo.
b) Melhorar áreas problemáticas.
c) Obter estimativas mais seguras.
d) Estabelecer prazos para evitar dificuldades passadas.
e) Reduzir o risco global.

8. ESTIMATIVAS

A **estimativa de tamanho** é uma métrica relacionada com o esforço despendido em um projeto e serve de ponto de partida para o planejamento.

Segundo Paula Filho (2003, p. 239-240), "um bom estimador deve atender aos seguintes critérios":

- ser contável por meio de um procedimento bem definido;
- ser calculável a partir da informação contida na especificação dos requisitos de *software*;
- apresentar boa correlação com o esforço de desenvolvimento.

Lembre-se de que, embora vários estimadores possam preencher esses critérios, é importante verificar com dados reais se realmente existe a correlação suposta nos itens anteriores.

Nesse tipo de estimativa, a complexidade do produto é medida por meio de estimadores de tamanho derivado dos requisitos.

Segundo Paula Filho (2003), um estimador de tamanho bastante utilizado na indústria de *software* é o **número de pontos de função**. Esse método estima o tamanho baseando-se na complexidade das interfaces de um produto, por meio de regras padronizadas de contagem, as quais devem ser rigorosas e detalhadas, por avaliadores treinados.

A **estimativa de esforço** é um item importante para o desenvolvimento de um projeto, ou seja, é o esforço que será despendido.

Essa estimativa está relacionada à quantidade de pessoas/dia que será necessária para desenvolver o projeto dentro do cronograma estipulado.

O esforço é, certamente, o componente de custo mais incerto, pois capacitação, experiência, interesse e treinamento podem influir no tempo de execução de uma tarefa (PLEEDGER, 2004).

Estimativas de custo, de cronograma e de esforços devem ser realizadas o quanto antes, durante o ciclo de vida do projeto, pois elas afetam a distribuição de recursos e a viabilidade do projeto.

Estimar um cronograma é uma das tarefas mais difíceis de realizar na etapa de planejamento do *software*. Essa etapa é considerada difícil porque são muitos os aspectos envolvidos, como:

a) a disponibilidade de recursos no momento da execução de uma dada tarefa;
b) as interdependências das diferentes tarefas;
c) a ocorrência de possíveis estrangulamentos do processo de desenvolvimento;
d) as operações necessárias para agilizar o processo;
e) a identificação das principais atividades, revisões e indicadores de progresso do processo de desenvolvimento.

Uma das formas de se garantir que os cronogramas sejam cumpridos é pensar no projeto como um todo, ou seja, pensar as interdependências existentes e como o documento pode ser trabalhado com o Diagrama de Gantt, pois, baseado nele, é possível verificar quais atividades são dependentes e qual é o tempo total máximo para sua execução.

O cronograma explicita as atividades relevantes do desenvolvimento e os indicadores de progresso associados. Portanto, é importante que os indicadores de progresso sejam representados por resultados concretos (por exemplo, um documento).

A disponibilidade de recursos também será representada ao longo do cronograma. O impacto da indisponibilidade dos recursos no momento em que eles são necessários deve constar, se possível, no cronograma.

Quando a tarefa é fixar prazos para os projetos de *software*, diversas questões podem ser formuladas:

a) Como relacionar o tempo cronológico com o esforço humano?
b) Que tarefas e que grau de paralelismo podem ser obtidos?

c) Como medir o progresso do processo de desenvolvimento (indicadores de progresso)?
d) Como o esforço pode ser distribuído ao longo do processo de Engenharia de *Software*?
e) Que métodos estão disponíveis para a determinação de prazos?
f) Como representar fisicamente o cronograma e como acompanhar o progresso desde o início do projeto?

Estimativa de custo

Um custo mal dimensionado acarreta problemas: se superestimado, pode provocar uma rejeição do cliente na hora de fechar o negócio; uma estimativa de preço abaixo do real pode provocar a equipe a trabalhar muito além do tempo sem o devido retorno financeiro.

Portanto, uma boa estimativa de custo faz com que o projeto se desenvolva de forma adequada; auxilia na definição das etapas, da quantidade de pessoas necessária no desenvolvimento do projeto e do perfil e competência necessária de cada integrante da equipe, antecipadamente.

Há inúmeras razões para estimativas imprecisas.

Pleedger (2004, p.81) define algumas dessas razões:

a) frequente solicitação de mudança pelos usuários;
b) tarefas negligenciadas;
c) falta de entendimento do usuário sobre suas próprias exigências;
d) análise insuficiente no desenvolvimento de uma estimativa;
e) falta de coordenação do desenvolvimento do sistema, dos serviços técnicos, das operações, do gerenciamento de dados e de outras funções, durante o desenvolvimento;
f) complexidade do sistema proposto;
g) necessidade de integração com os sistemas existentes;

h) complexidade dos programas no sistema;
i) capacidade dos membros da equipe de projeto;
j) experiência da equipe do projeto com a aplicação;
k) quantidade de padrões de programas e documentação;
l) disponibilidade de ferramentas, tais como geradores de aplicação;
m) experiência da equipe com o *hardware*.

As razões pontuadas anteriormente não são as únicas que podem provocar uma estimativa mal feita, mas, com certeza, ao observarmos esses pontos, estaremos dando um bom passo em direção a uma estimativa de custo adequada.

Quando falamos em custo ou orçamento de projeto, estamos falando de vários tipos: instalações, pessoal, métodos e ferramentas.

É importante pensarmos em todos os custos diretos e indiretos na hora de dimensionarmos o valor. Custo de instalações, por exemplo, englobam *hardware*, espaço, mobília, telefones, *modems*, sistemas de aquecimento ou ar-condicionado, cabos, discos, papel, caneta, copiadores e todos os outros itens que fazem parte do ambiente no qual será desenvolvido o projeto.

É preciso pensar, também, em custos ocultos, que não são visíveis aos gerentes e desenvolvedores, como espaço físico para trabalhar, silêncio (há pessoas que preferem e produzem mais quando em ambientes extremamente silenciosos e que, ao serem colocados em um ambiente barulhento, podem ter sua produtividade prejudicada) etc.

Estimativa de recursos humanos

As habilidades dos recursos humanos necessárias para construir o *software* dependerão das características do *software* a ser desenvolvido.

Há uma grande diversidade no que diz respeito aos modos de organização das equipes de desenvolvimento de *software*. A escolha de como a equipe de desenvolvimento será organizada é um dos pontos que devem ser definidos nesta etapa. Considerando que um processo de desenvolvimento de *software* vai ocupar uma equipe de *n* pessoas com uma duração de *k* anos, é possível comentar algumas opções organizativas (Pressman, 1995):

- *n* indivíduos são alocados a *m* diferentes tarefas com pequeno grau de interação, sendo que a coordenação da equipe fica a cargo do gerente de projeto;
- *n* indivíduos são alocados a *m* diferentes tarefas, com $m \leq n$, formando equipes informais de desenvolvimento, com um responsável *ad-hoc* de cada equipe, sendo que a coordenação entre as equipes é da responsabilidade do gerente do projeto;
- *n* indivíduos são organizados em *k* equipes, cada equipe é alocada para uma ou mais tarefas; a organização de cada equipe é específica a ela própria, a coordenação fica a cargo da equipe e do gerente do projeto.

Sem discutir em detalhes os argumentos contrários ou a favor de cada uma das opções, é importante dizer que a constituição em equipes formais de desenvolvimento (como sugere a opção 3) é mais produtiva.

O principal objetivo da formação de equipes é o desenvolvimento de um conceito de projeto como sendo o resultado de uma reunião de esforços. A criação de equipes evita o sentimento de "ego-programação" que pode atingir as pessoas envolvidas em um desenvolvimento, transformando o "meu programa" em "nosso programa".

De um ponto de vista geral, pode-se estabelecer uma referência no que diz respeito à organização de uma equipe de desenvolvimento de *software*, sendo que o número de equipes e o número de membros de cada equipe vão variar em função da grandeza do projeto.

O núcleo de uma equipe vai ser composto dos seguintes elementos:

- um **engenheiro sênior** (ou programador chefe), responsável do planejamento, pela coordenação e pela supervisão de todas as atividades relativas ao desenvolvimento do *software*;
- o **pessoal técnico**, de dois a cinco membros, que realizam as atividades de análise e desenvolvimento;
- um **engenheiro substituto**, que atua no apoio ao engenheiro sênior e que pode, eventualmente, substituí-lo sem grandes prejuízos ao desenvolvimento do *software*.

Além desse núcleo, a equipe de desenvolvimento pode, ainda, receber a colaboração dos seguintes elementos:

- um **conjunto de especialistas** (telecomunicações, bancos de dados, interface homem-máquina etc.);
- **pessoal de apoio** (secretárias, editores técnicos, desenhistas etc.);
- um **bibliotecário**, o qual será responsável pela organização e catalogação de todos os componentes do produto de *software* (documentos, listagens, mídia magnética, coleta de dados relativos ao projeto, módulos reutilizáveis etc.).

A atividade de estimar experiência, o acesso a boas informações históricas e a coragem para comprometer-se com medidas quantitativas, mesmo que essas medidas não sejam explícitas.

Pressman (1995) descreve alguns fatores que influenciam nos riscos das estimativas:

1) **Complexidade do projeto:** tem forte efeito sobre a incerteza, mas é relativa à familiaridade que a equipe tem com o esforço passado. Quanto maior a complexidade, maiores os riscos das estimativas.
2) **Tamanho do projeto:** à medida que o tamanho aumenta, a interdependência entre os elementos do *software*

também aumentam. Enfim, projetos maiores têm maior risco de estimativas incorretas.

3) **Grau de estrutura do projeto:** refere-se à facilidade com que as funções podem ser dispostas em compartimentos e à natureza hierárquica com que as informações podem ser processadas. À medida que o grau de estrutura aumenta, a capacidade de se avaliar com precisão é melhorada e os riscos diminuem.
4) **Qualidade da definição do sistema e a variabilidade:** requisitos mal compreendidos e sujeitos a mudanças aumentam os riscos das estimativas (instabilidade de custo e prazo).

9. ANÁLISE DE RISCOS

A análise dos riscos é uma das atividades essenciais para o bom encaminhamento de um projeto de *software* e, consequentemente, de sua gerência.

Essa atividade está baseada na realização de quatro tarefas, conduzidas de forma sequencial:

a) identificação dos riscos;
b) projeção dos riscos;
c) avaliação dos riscos;
d) administração e monitoração dos riscos.

Identificação dos riscos

O objetivo da primeira tarefa é destacar, pelos responsáveis gerentes e analistas, “todos” os eventuais riscos aos quais o projeto poderá ser submetido. É importante salientar que é difícil identificar todos os riscos, mas, considerando a experiência, é possível identificar uma quantidade significativa de riscos e com isso trabalhar em um projeto mais coeso.

Segundo Pressman (1996), de acordo com sua natureza, os riscos podem ser divididos em três categorias.

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

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*) relacionadas a algum fator de risco.

Veja, a seguir, um exemplo de *checklist* sugerido por Pressman (1996, p. 132):

> *Checklist* para riscos de composição de pessoal:
>
> - São as melhores pessoas disponíveis?
> - As pessoas têm combinação correta de habilidades?
> - Há pessoas suficientes?
> - Há pessoas se dedicando parcialmente ao projeto?
> - O pessoal está comprometido com toda a duração do projeto?
> - Receberam o treinamento necessário?
>
> As respostas a estas perguntas permitem a identificação do impacto dos riscos na composição de pessoal.

Projeção dos riscos

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

- Qual a probabilidade do risco ocorrer durante o projeto?
- Quais as consequências dos problemas associados ao risco no caso de ocorrência deste?

De acordo com Pressman (1996), as respostas a tais questões podem ser obtidas, fundamentalmente, por quatro atividades:

1) estabelecimento de uma escala que reflita a probabilidade estimada de ocorrência de um risco;
2) estabelecimento das consequências do risco;
3) estimativa do impacto do risco sobre o projeto e sobre o *software* (produtividade e qualidade);
4) 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 essa representação permite representar as incertezas de modo realístico.

A natureza do risco permite indicar os problemas prováveis, caso ele ocorra, como, 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.

Avaliação dos riscos

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

Para tornar a avaliação eficiente, é necessário definir um **nível de risco referente**. Exemplos de níveis referentes típicos em projetos de Engenharia de *Software* são: o custo, o prazo e o desempenho. Isso significa que se terá um nível para o excesso de

custo, para a ultrapassagem de prazo e para a degradação do desempenho ou qualquer combinação dos três.

Dessa 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. Normalmente, é possível estabelecer um limite, denominado de ponto referente (*breakpoint*) em que tanto a decisão de continuar o projeto ou de abandoná-lo podem ser tomadas.

Administração e monitoração dos riscos

Uma vez avaliados os riscos de desenvolvimento, é necessário que medidas sejam tomadas para evitar a ocorrência dos riscos ou que ações sejam definidas para a eventualidade da ocorrência dos riscos. Esse é o objetivo da tarefa de administração e monitoração dos riscos. Para isso, 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 (PRESSMAN, 1996).

Por exemplo, considerando a alta rotatividade de pessoal em uma equipe, um fator de risco, baseados em projetos anteriores que a probabilidade de ocorrência desse risco é de 0,70 (muito elevada) e que sua ocorrência pode aumentar o prazo do projeto em 15% e seu custo global em 12%.

Assim, Pressman (1996) propõe as seguintes ações de administração desse fator de risco:

1) 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.);
2) providências são necessárias para eliminar ou reduzir as causas “controláveis” antes do início do projeto;
3) no início do projeto, pressupor que a rotatividade vai ocorrer e prever a possibilidade de substituição de pessoas quando estas deixarem a equipe;

4) organizar equipes de projeto de forma que as informações sobre cada atividade sejam amplamente difundidas;
5) definir padrões de documentação para garantir a produção de documentos de forma adequada;
6) realizar revisões do trabalho entre colegas de modo que mais de uma pessoa esteja informada sobre as atividades desenvolvidas;
7) definir um membro da equipe que possa servir de *backup* para o profissional mais crítico.

É importante observar que a implementação dessas ações pode afetar, também, os prazos e o custo global do projeto.

Todo o trabalho efetuado nessa tarefa é registrado em um 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).

Mas antes de falarmos de Plano de Projeto devemos conhecer o documento que engloba todos os assuntos vistos nesta unidade, que é o **plano de *software***.

10. PLANO DE PROJETO DE *SOFTWARE*

Ao final da etapa de estimativa, de discussão sobre riscos e equipe está prevista a criação de um documento chamado de **plano de projeto de *software* ou plano de gerenciamento de projeto de *software*** o qual deverá ser revisto para servir de referência às etapas posteriores.

Esse plano servirá como “plano de ação” para as próximas etapas e poderá ser atualizado sempre que necessário.

Ele apresentará as informações iniciais de custo e cronograma que vão nortear o desenvolvimento do *software*. Ele consiste em um documento relativamente breve, que será encaminhado às diversas pessoas envolvidas no desenvolvimento do *software*.

Dentre as informações que constam nesse 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;
- a possibilidade de permitir uma visão global do processo de desenvolvimento do *software* a todos os envolvidos.

Vários autores definem, para o mesmo conteúdo, estruturas similares que compõem os itens do **plano de *software***.

A Figura 4 apresenta uma possível estrutura para esse documento, sugerida por Pressman (1996). A apresentação dos custos e o cronograma podem diferir dependendo de quem será o autor do documento, assim como da política de descrição desses itens adotada pela empresa.

1.0 - Contexto
- **1.1 - Objetivos do projeto**
- **1.2 - Funções principais**
- **1.3 - Desempenho**
- **1.4 - Restrições Técnicas e Administrativas**

2.0 - Estimativas
- **2.1 - Dados utilizados**
- **2.2 - Técnicas de Estimativa**
- **2.3 - Estimativas**

3.0 - Riscos do Projeto
- **3.1 - Análise dos Riscos**
- **3.2 - Administração dos Riscos**

4.0 - Cronograma
- **4.1 - Divisão do esforço no projeto**
- **4.2 - Rede de Tarefas**
- **4.3 - Timeline**
- **4.4 - Tabela de recursos**

5.0 - Recursos necessários
- **5.1 - Pessoal**
- **5.2 - Software e Hardware**
- **5.3 - Outros recursos**

6.0 - Organização do Pessoal
7.0 - Mecanismos de Acompanhamento
8.0 - Apêndices

Fonte: PRESSMAN, 1996, p. 167.
Figura 4 *Proposta de estrutura para o documento de Plano do Software.*

Não é necessário que o **plano de *software*** seja um documento extenso e complexo, pois seu objetivo é auxiliar a análise de viabilidade dos esforços de desenvolvimento. Esse documento está associado aos conceitos de "**o que**", "**quanto**" e "**quão longo**" do desenvolvimento do *software*. E as etapas posteriores estarão associadas ao conceito de "**como**" (PRESSMAN, 1996).

Para comunicarmos a análise e o gerenciamento de riscos, a estimativa de custo, o tempo e o esforço, assim como a equipe e as diretrizes de teste, manutenção, garantia de qualidade do sistema etc., é necessário criarmos um **plano de projeto (PP)**.

O plano de projeto descreverá as necessidades do cliente, assim como a forma que será idealizada a execução dessas atividades.

Como o sistema ainda está em análise, a documentação chama-se **plano**; ao final do projeto, esse documento deverá ser atualizado e se tornará a **documentação oficial do projeto**, que será uma referência para novas atualizações, manual de usuário etc.

Um bom plano de projeto inclui os seguintes itens:

1) **Escopo do projeto:** descrição geral do projeto.
2) **Plano de *software*:** informações que servirão de documentação para as etapas posteriores.
3) **Descrição técnica, procedimentos e ferramentas propostas para o projeto:** definição de requisitos e modelagem.
4) **Plano de garantia da qualidade:** definição das estratégias para garantir a qualidade do *software.*
5) **Plano de gerência de configurações:** definição das estratégias para gerenciar as alterações de *software.*
6) **Plano de documentação:** descrição de como está prevista a documentação e quais são os itens e informações necessários para serem documentados.
7) **Plano de gerência de dados:** plano de gerenciamento de dados e integração de BD.
8) **Plano de gerência de recursos:** plano de gerenciamento de recursos de desenvolvimento (pessoal, técnico etc.).

9) **Plano de testes:** descrição dos casos de testes que serão realizados.
10) **Plano de treinamento:** descrição dos procedimentos de treinamento.
11) **Plano de segurança:** descrição do plano de segurança.
12) **Plano de gerência de riscos:** descrição do plano de gerenciamento de riscos.
13) **Plano de manutenção:** descrição do plano de manutenção.

11. AQUISIÇÃO DE *SOFTWARE*

Como você pode perceber, planejar e controlar o processo de desenvolvimento de *software* não são tarefas fáceis. Se o planejamento não for bem feito e o desenvolvimento bem acompanhado, há grandes chances de o projeto não ser bem-sucedido. Enfim, são vários os riscos envolvidos no desenvolvimento de *software*.

No que diz respeito a esses riscos, Pressman (1995) dá duas dicas ao planejador:

- Se no mercado já houver um *software* que satisfaça plenamente aos requisitos do cliente, adquira-o. O custo de aquisição de um *software* existente, na maioria das vezes, será menor do que o custo para desenvolver um *software* equivalente. Além, disso você não corre os riscos inerentes ao desenvolvimento.
- No entanto, se o *software* existente exigir modificações para atingir os objetivos do cliente, tenha cautela. O custo de modificar um *software* existente pode ser muito maior do que o custo para desenvolvê-lo completamente.

Embora a Engenharia de *Software* sugira o desenvolvimento de *software*, em alguns casos pode ser mais interessante adquirir o *software* do que o desenvolver. Essa é uma decisão que o gerente de um projeto pode ter de tomar no contexto de um projeto.

Com relação à aquisição de *software*, diversas ações podem ser tomadas, tais como:

- adquirir (ou licenciar) um pacote comercial que atenda às especificações estabelecidas;
- adquirir um pacote comercial e modificá-lo de forma que o novo *software* atenda às especificações de projeto;
- encomendar o *software* a terceiros para que este atenda às especificações.

Os procedimentos que devem ser efetuados para a aquisição de um *software* vão depender do quanto as especificações são críticas e do seu custo. No caso de um *software* de custo relativamente baixo (um *software* para PC, por exemplo), pode ser mais interessante adquiri-lo e fazer uma análise de adequação às especificações de projeto.

No caso de *softwares* mais caros, uma análise mais cuidadosa faz-se necessária e os procedimentos para essa análise podem ser os seguintes:

a) desenvolver uma especificação funcional e de desempenho para o *software* a ser projetado, estabelecendo, quando possível, valores mensuráveis;
b) realizar a estimação do custo interno de desenvolvimento;
c) escolher um conjunto de pacotes comerciais que poderiam atender às especificações;
d) desenvolver um esquema de análise comparativa que permita confrontar as funções-chave das soluções alternativas;
e) avaliar cada pacote com base na qualidade do produto, no suporte do vendedor (garantia), reputação do vendedor e direcionamento do produto;
f) entrevistar usuários do *software*, colhendo suas opiniões sobre os pacotes.

Com base nos resultados obtidos, a decisão entre comprar ou desenvolver o *software* estará fundamentada nos seguintes critérios:

- a data de entrega do produto precede a data de finalização do produto se desenvolvido internamente?

- o custo de aquisição mais o custo de "customização" será inferior ao custo de desenvolvimento interno?
- o custo de suporte externo (contrato de manutenção) é inferior ao custo do suporte interno?

Assim como há modelos propostos pela Engenharia de *Software* para o processo de desenvolvimento de *software*, há, também, modelos que auxiliam e normatizam a aquisição de *softwares*.

12. QUESTÕES AUTOAVALIATIVAS

Sugerimos que você procure responder, discutir e comentar as questões a seguir que tratam da temática desenvolvida nesta unidade. Se você encontrar dificuldade em respondê-las, procure revisar os conteúdos estudados para sanar suas dúvidas. A auto-avaliação pode ser uma ferramenta importante para você testar seu desempenho. Este é um momento impar para você fazer uma revisão desta unidade. Lembre-se de que no ensino a distância a construção do conhecimento se dá de forma cooperativa e colaborativa, portanto compartilhe com seus colegas suas descobertas.

1) Na gerência de projetos, um planejamento tido como bem estruturado é aquele que primeiramente define quais atividades devem ser realizadas. Neste contexto, considere:

   A definição da atividade deve conter um descritivo sobre ela, devidamente justificado, uma vez que as pessoas precisam saber o motivo da existência de tais atividades, quais os recursos humanos que nela serão alocados, em que ordem elas devem ser realizadas, quais técnicas ou recursos devem ser empregados e o local de sua realização.

   Na afirmação anterior, os elementos que permitirão estimar quanto será despendido em cada atividade, tanto nas fases quanto no projeto como um todo, em termos de tempo e custo, estão subentendidos, sucessivamente, na matriz:

   a) O que X (quem, quando, por que, como, onde).
   b) O que X (por que, quem, quando, como, onde).
   c) Quando X (o que, por que, quem, como, onde).
   d) Quando X (por que, o que, quem, como, onde).
   e) O que X (quem, quando, como, por que, onde).

Fonte: Prova aplicada em 03/2010 para o concurso do(a) TRT - 20ª REGIÃO (SE) - 2009, realizado pelo órgão/instituição Tribunal Regional do Trabalho da 20ª Região - Sergipe, área de atuação Jurídica, organizada pela banca FCC, para o cargo de Analista Administrativo - Informática, nível superior, área de formação Tecnologia da Informação.

2) Quais das seguintes afirmações são verdadeiras? As Métricas de *software* servem para:

I – Indicar a qualidade do produto e avaliar a produtividade.

II – Auxiliar na melhoria do processo.

III – Formar uma base para as estimativas e justificar a aquisição de ferramentas.

IV – Determinar se a utilização de um método traz benefícios ou não.

Escolha a alternativa que representa, APENAS, as afirmações CORRETAS.

a) Todas as alternativas.
b) Apenas as alternativas I, II e IV.
c) Apenas as alternativas I, IV.
d) Apenas as alternativas II e III.
e) Nenhuma delas.

Fonte: POSCOMP 2002

Gabarito

Depois de responder às questões autoavaliativas, é importante que você confira o seu desempenho, a fim de que possa saber se é preciso retomar o estudo desta unidade. Assim, confira, a seguir, as respostas corretas para as questões autoavaliativas propostas anteriormente:

1) (b)

2) (a)

13. CONSIDERAÇÕES

Nesta unidade, você teve a oportunidade de acompanhar o processo de desenvolvimento de um *software*, analisou os riscos mais comuns relacionados ao desenvolvimento dele e compreendeu a importância e a relação existente entre o planejamento e a gerência de desenvolvimento de um *software*.

Na próxima unidade, discutiremos a definição de requisitos de *softwares*, que direcionará todo o desenvolvimento.

Definir requisitos é uma fase crucial na Engenharia de *Software*; significa determinar o que é necessário para que o projeto atinja seus objetivos, dando origem à documentação de requisitos que servirá como orientador aos desenvolvedores.

14. REFERÊNCIAS BIBLIOGRÁFICAS

BRAZ JUNIOR, O. P. *Engenharia de software*. Disponível em: <http://inf.unisul.br>. Acesso em: 13 maio 2007.

PAULA FILHO. *Engenharia de software*: fundamentos, métodos e padrões. São Paulo: LTC, 2003.

PIEKARSKI, A. E. T.; QUIN´AIA, M. A. *Reengenharia de software*: o que, por que e como. Disponível em: <http://www.unicentro.br/pesquisa/editora/revistas/exatas/v1n2/Reengenharia.pdf>. Acesso em: 22 jun. 2007.

PFLEEGER, S. L. *Engenharia de software*: teoria e prática. São Paulo: Prentice Hall, 2004.

PRESSMAN, R. S. *Engenharia de software*. São Paulo: Makron Books, 1996.

_____. *Engenharia de software*. São Paulo: Makron Books, 2006.

SOMMERVILE, I. *Software Engineering*. 4th ed. Addison-Wesley Publishers Ltd, 1992.

EAD

Análise de Requisitos de *Software*

4

1. OBJETIVOS

- Compreender os conceitos de requisito e estimativa de *software.*
- Entender a importância da definição de requisito e estimativa de *software*.
- Analisar os conceitos estudados.

2. CONTEÚDOS

- Análise de requisitos.
- Atividades de análise.
- Documentos e revisão de requisitos.
- Processo de comunicação.

3. ORIENTAÇÕES PARA O ESTUDO DA UNIDADE

Antes de iniciar o estudo desta unidade, é importante que você leia as orientações a seguir:

1) Durante esta unidade, você vai entrar em contato com a complexidade que envolve o trabalho de um analista de sistemas em propor um sistema computacional adequado à necessidade do cliente.
2) Um dos maiores mitos da indústria do *software* é o cliente acreditar que a tecnologia irá solucionar todos os seus problemas. Porém, a tecnologia não é mágica. Para que a inserção de um sistema baseado em computador seja eficaz em uma organização é importante que esta esteja organizada, as funções bem claras e as responsabilidades bem definidas.
3) O analista de sistemas, em sua vida profissional, irá se deparar com os mais diversos ambientes. Para que ele desenvolva bem seu trabalho, é preciso que ele tenha versatilidade para conviver com as mais diversas pessoas e situações. Para iniciar o estudo desta unidade, é importante que você tenha em mente que o analista é quem faz a interface entre estes dois mundos distintos: a tecnologia e um cliente com as mais diversas necessidades.
4) Além disso, no contexto nacional, no qual temos um grande número de micro e pequenas empresas desenvolvedoras de *software*, é muito comum você se deparar com o termo analista-programador. Isto é, um mesmo profissional desempenhando mais de um papel no processo de *software*. Para que este profissional seja eficiente em ambas as funções, é importante que a cada momento ele incorpore o papel referente a cada tarefa.
5) O conteúdo desta unidade será útil para que você perceba a complexidade que envolve a atividade do analista. Portanto, procure um local tranquilo para seus estudos, discuta com seus colegas sobre o tema. Tire dúvidas com seu tutor. Enfim, procure assimilar bem o aprendizado, aproveite esta oportunidade.

4. INTRODUÇÃO À UNIDADE

Na unidade anterior, você teve a oportunidade de aprender sobre gerência e planejamento de projeto e conhecer um pouco mais sobre a importância dessas atividades. Aliás, até na nossa vida diária planejar nossas ações e gerenciá-las é fundamental para que tenhamos bons resultados.

Nesta unidade, você terá a oportunidade de aprender os conceitos e as definições de requisitos e estimativas de *software*.

5. ANÁLISE DE REQUISITOS

Na maioria das vezes em que falamos sobre analistas e projetistas, temos um impasse – até que ponto vai a atividade de um e quando começa a atividade do outro.

O **analista** é o responsável por decidir **o que** fazer e o **projetista** decide **como** fazer. Essas duas funções juntas auxiliam, e "o que" e "o como" é parte dos **requisitos**. Os requisitos nada mais são do que um conjunto de necessidades a serem atendidas pelo sistema a ser desenvolvido.

Entender o que são requisitos de *software* é fundamental para o sucesso de um projeto de *software*. Independentemente da precisão do projeto e da implementação de um *software*, certamente, trará problemas ao cliente/usuário se sua análise de requisitos for mal realizada.

A **análise de requisitos** é uma tarefa que envolve um trabalho de descoberta, refinamento, modelagem e especificação das necessidades e desejos relativos ao *software* a ser desenvolvido.

Nessa fase, tanto o cliente como o desenvolvedor vão desempenhar um papel importante. Ao cliente caberá a formulação (de modo concreto) das necessidades em termos de funções e desempenho; enquanto isso, o desenvolvedor atuará como indagador, consultor e solucionador de problemas.

Essa etapa é de suma importância no processo de desenvolvimento de um *software*, sobretudo por estabelecer o elo entre a alocação do *software* no plano de sistema (realizada na etapa de Engenharia de Sistema) e o projeto do *software*.

Dessa forma, permite que o engenheiro de sistemas especifique as necessidades do *software* em termos de funções e de desempenho, estabeleça as interfaces com os demais elementos do sistema e especifique as restrições de projeto.

A análise de requisitos permite ao engenheiro de *software* (ou analista) 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, essa etapa proporciona a obtenção da informação e das funções que poderão ser traduzidas em projeto procedimental, arquitetônico e de dados.

Por meio dos requisitos é possível definir critérios de avaliação da qualidade do *software* a serem verificados quando concluído.

À medida que as aplicações para o *software* expandiram-se e tornaram-se cada vez mais complexas, maiores passaram a ser a dificuldade e os esforços gastos para fazer o levantamento dos requisitos.

6. ATIVIDADES DA ANÁLISE DE REQUISITOS

Todo processo de análise relacionada nessa etapa define dois tipos de requisitos:

- funcionais;
- não funcionais.

Mas antes de definirmos quais requisitos são funcionais, é preciso entender a análise das definições de requisitos.

Processo de definição e identificação de requisitos

Requisitos: condição necessária para atingirmos os objetivos.

Especificação: descrição minuciosa das características que um material, uma obra, ou um serviço deverão apresentar (PRESSMAN, 1996).

Portanto**, especificação** é diferente de **requisitos** e as duas atividades auxiliam a melhorar a descrição dos objetivos do projeto. Na literatura, você pode encontrar: Especificação de requisitos ou Especificação de projeto, tais expressões são sinônimas.

No começo do projeto, o analista estuda os documentos de especificação do sistema e o plano do *software*, como forma de entender o posicionamento do *software* no sistema e revisar o escopo do *software* utilizado para definir as estimativas do projeto.

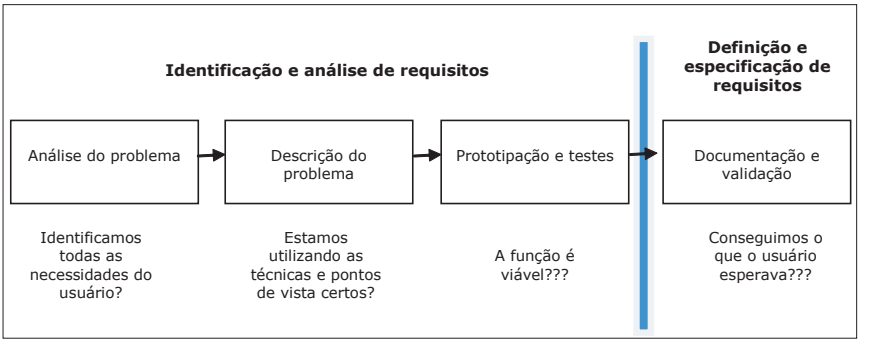
Quando um cliente solicita a construção de um novo sistema, certamente ele terá uma noção do que esse sistema fará, se vai substituir, adequar ou complementar um antigo, não importa, pois cada um tem um propósito, uma funcionalidade.

"Um requisito é uma característica do sistema ou a descrição de algo que o sistema é capaz de realizar, para atingir os seus objetivos" (PLEEDGER, 2005, p. 111).

Para iniciarmos o processo de definição de requisitos é preciso conversarmos com nosso cliente. Fazer perguntas, demonstrar sistemas similares, desenvolver protótipo de partes do sistema. Em seguida, registramos essas informações em um documento ou banco de dados.

Geralmente, os requisitos são escritos, em um primeiro momento, em uma linguagem que o cliente poderá entender e aprovar; posteriormente, eles serão reescritos em uma representação mais matemática ou visual para que os projetistas transformem os requisitos em um projeto de sistema. Esse processo de reescrever os requisitos em uma linguagem diferenciada é o **processo de modelagem**.

Observe, na Figura 1, a descrição do processo de requisitos.



Fonte: PFLEEGER, 2004, p. 112.
Figura 1 *Processo de definição de requisitos.*

A identificação de requisitos é considerada parte importante do processo, havendo inúmeras maneiras de fazer isso, pois dependerá da experiência e da preferência pessoal. Entretanto, uma forma apropriada é dividir o projeto em partes e definir quem teria as informações adequadas, auxiliando na definição dos requisitos para aquela fase do sistema.

Conversando, por meio de entrevista, é possível definir a amplitude de nosso sistema ou da parte que somos responsáveis.

Durante essa conversa, é fundamental que não reste nenhuma dúvida e, ao final, durante o processo de documentação dos requisitos, eles serão categorizados em:

- requisitos totalmente satisfeitos;
- requisitos altamente desejáveis, mas não necessários;
- requisitos possíveis, podendo ser eliminados.

É importante salientar que nenhum requisito especifica "como o sistema deverá ser implementado". O requisito informa o propósito do sistema.

Os requisitos identificam **o que** o sistema deve fazer e o projeto **como**.

7. TIPOS DE DOCUMENTOS DE REQUISITOS

Como o enfoque é o problema do nosso cliente, a identificação e a análise dos requisitos servem como propósitos distintos, porém relacionados.

A identificação dos requisitos possibilita-nos escrever um documento denominado **definição de requisitos**, conforme ilustra a fase de definição e especificação de requisitos na Figura 1, do tópico anterior. Esse documento deve ser escrito em linguagem natural, de forma que o usuário possa lê-lo, entendê-lo e interpretá-lo; representando um consenso entre o que o cliente deseja e o que é possível e será realizado.

Posteriormente, outro documento é gerado com as **especificações dos requisitos**. Esse documento redefine o anterior para uma linguagem mais técnica, apropriada para o desenvolvimento do projeto. Algumas vezes, um único documento é gerado e serve para os dois propósitos: comunicação com o cliente e projetista.

Contudo, é fundamental que haja uma correspondência direta entre cada requisito do documento de definição e de especificação de requisitos, pois é nesse momento que se inicia o processo de **gerência de configurações** utilizadas durante o ciclo de vida e que veremos com mais detalhes na próxima unidade.

Segundo Pfleeger (2004), **gerenciamento de configurações** é um conjunto de procedimentos que controlam:

a) os requisitos, os quais definem o que o sistema fará;
b) os módulos de projetos que serão gerados a partir dos requisitos;
c) o código do programa que implementa o projeto;
d) os testes que verificam a funcionalidade do sistema;
e) os documentos que descrevem o sistema.

Para Pfleeger (2004, p. 114),

> [...] a gerência de configuração fornece as 'linhas' que unem as partes do sistema entre si, unificando os componentes que tenham sido desenvolvidos separadamente. Essas linhas nos permitem coordenar as atividades de desenvolvimento [...].

Especificamente durante a identificação e análise de requisitos, o gerenciamento de configurações detalha a correspondência entre os elementos da definição de requisitos e a especificação de requisitos, de modo que a visão do cliente esteja associada à do desenvolvedor e possa ser acompanhada.

Tipos de requisitos: funcionais e não funcionais

Segundo Pfleeger (2004, p. 115):

> Os requisitos descrevem o comportamento de um sistema. À medida que o sistema atua nos dados ou nas instruções, objetos ou entidades se "movem" de um estado para outro, por exemplo: de vazio para cheio; de ocupado para desocupado; de envio para recebimento. Isto é, em um determinado estado, o sistema satisfaz um conjunto de condições; quando o sistema atua, ele pode mudar seu estado como um todo, modificando o de um objeto. Os requisitos expressam os estados e as transições do sistema e do objeto, de um para outro. Em particular, os requisitos descrevem atividades do sistema, tais como uma reação à inserção de dados, e o estado de cada entidade antes e depois de a atividade ocorrer. Por exemplo, em um sistema de folha de pagamento, os funcionários podem existir em pelo menos dois estados: funcionários que ainda não foram pagos e funcionários que já foram pagos. Os requisitos descrevem, como, na emissão de contracheques, um funcionário pode se "mover" do primeiro estado para o segundo.

Para descrevermos um requisito, podemos classificá-los em **funcionais e não funcionais**.

Requisitos funcionais

Os requisitos funcionais descrevem uma interação entre o sistema e o ambiente, ou seja, descrevem como o sistema deve funcionar, considerando determinado estado. E, também, declaram quais serviços que o sistema irá fornecer, como o sistema deve reagir a entradas específicas e como se comportará em determinadas situações.

Vejamos como exemplo o sistema de emissão de contracheque mencionado por Pfleeger. Se esse sistema for utilizado semanalmente, os requisitos funcionais deverão responder às questões relacionadas aos contracheques emitidos. Dessa forma, surgem

algumas questões, tais como: quais informações são necessárias para que um contracheque seja emitido? Sob quais condições o valor poderá ser alterado? O que causa a exclusão de um funcionário da folha de pagamento? etc.

É importante lembrar que as respostas a essas perguntas independem da implementação da solução para o cliente. Estamos descrevendo o que o sistema fará e quais os requisitos adicionais necessários para que ele funcione.

Vamos imaginar outra situação. Por exemplo, um sistema de matrícula escolar.

Logo no início, sabemos que um sistema de matrícula escolar deve permitir inclusão, alteração, exclusão e consulta, mas quais as perguntas necessárias para definição dos requisitos? Quais perguntas vocês fariam para o cliente que contratou seus serviços?

Neste momento, é importante que você reflita. Compare os questionamentos referentes ao sistema de matrícula escolar com as respostas referentes ao sistema de contracheques e tente traçar um paralelo entre eles. Sugiro que vá além do exemplo do contracheque, tente transpor o conteúdo abordado até o momento para um novo sistema e defina os requisitos.

Requisitos não funcionais

Os requisitos não funcionais colocam restrições no sistema, em vez de informar o que ele fará. E eles também podem ser chamados de restrições.

Informações como: o sistema deverá rodar em uma plataforma XYZ, as consultas serão respondidas em x espaço de tempo, o sistema permitirá um compartilhamento de dados entre diferentes escolas do mesmo grupo etc., são informações dos requisitos não funcionais.

Esses requisitos limitam nossa seleção com relação à linguagem e à modelagem a ser utilizada (estruturada, essencial, orientada a objeto etc.).

Tantos os requisitos funcionais como os não funcionais devem ser definidos e identificados com os clientes de maneira cuidadosa, pois definirão os próximos passos de implementação, testes e manutenção, além de ser importante na questão da documentação do projeto.

O documento de especificação de requisitos descreve todas as informações sobre o sistema e sua integração com o ambiente.

Segundo Pfleeger (2004) e Pressman (1996), esse documento é formado por itens específicos, como podemos observar no quadro a seguir.

Observe atentamente, no quadro ao lado, os questionamentos que são necessários para o desenvolvimento de um sistema.

Quadro 1 Documento de Resquisitos

DOCUMENTO DE REQUISITOS	
Tipo	Descrição do conteúdo
Ambiente físico	Qual será o local de funcionamento do equipamento? O funcionamento será efetuado em um ou em vários locais? Existe alguma restrição ambiental, tal como: temperatura, umidade ou interferência magnética (interferências como *softwares* da área médica, de controladores de voos etc. devem ser consideradas de forma diferenciada).
Interface	A entrada tem origem em outros sistemas? A saída vai para outro sistema? Existe uma maneira preestabelecida na qual os dados devem ser formatados? (CEP – sempre numérico, por exemplo). Existe alguma mídia definida que os dados devem utilizar?
Os usuários e fatores humanos	Quem utilizará o sistema? Haverá diversos tipos de usuários? Qual o nível de habilidade de cada tipo de usuário? Que tipo de treinamento será necessário para cada tipo de usuário? Que facilidade o usuário terá para entender e utilizar o sistema? Qual será a dificuldade para que o usuário utilize adequadamente o sistema?

DOCUMENTO DE REQUISITOS	
Tipo	Descrição do conteúdo
Funcionalidade	O que o sistema fará? Quando o sistema fará? Existem diversos modos de operação? Como e quando o sistema pode ser modificado ou aprimorado? Existem limitações quanto à velocidade de execução, ao tempo de resposta ou à saída?
Documentação	Que documentação é necessária? A documentação deve ser *on-line*, no formato de livro, ou ambos? A que público se destina cada tipo de informação?
Dados	Qual será o formato dos dados de entrada e saída? Com que frequência os dados serão enviados e recebidos? Que precisão devem ter os dados? Com que grau de precisão os cálculos serão realizados? Existem dados que devem ser mantidos por determinado tempo?
Recursos	Que materiais, pessoal ou outros recursos são necessários para construir, utilizar e manter o sistema? Que habilidade os desenvolvedores devem ter? Quanto espaço físico será ocupado pelo sistema? Quais os requisitos quanto à energia, ao aquecimento ou ao condicionamento de ar? Existe um cronograma definido para o desenvolvimento? Existe um limite de custo para o desenvolvimento ou para a aquisição de *hardware* ou de *software*?
Segurança	O acesso ao sistema ou às informações deve ser controlado? Como os dados de um usuário serão isolados dos outros usuários? Como os programas dos usuários serão isolados de outros programas e do sistema operacional? Com que frequência será realizado o *backup*? Onde serão armazenadas as cópias de *backup*? Devem ser tomadas precauções contra fogo, danos provocados pela água, ocorrência de roubo etc.?

DOCUMENTO DE REQUISITOS	
Tipo	Descrição do conteúdo
Garantia de qualidade	Quais os requisitos quanto à confiabilidade, disponibilidade, manutenibilidade, segurança e outros atributos de qualidade? Como as características do sistema devem ser demonstradas para os outros? O sistema deve detectar e isolar defeitos? Qual o tempo médio entre falhas que foi determinado? Existe um tempo máximo permitido para reinicializar o sistema depois de uma falha? Como o sistema pode incorporar modificações no projeto? Apenas a manutenção corrigirá os erros ou também incluirá o aprimoramento do sistema? Que medidas de eficiência serão aplicadas à utilização dos recursos e ao tempo de resposta? Com que facilidade o sistema se deslocará de um local para outro (acesso remoto) ou de um tipo de computador para outro?

Alguns passos podem ser seguidos para descobrir exatamente o que o cliente quer. Para tanto, é necessário que sejam realizadas várias entrevistas e reuniões. Basicamente, durante o processo de especificação de requisitos devemos:

a) analisar a situação atual;
b) fazer com que o usuário aprenda a entender o contexto, os problemas e os relacionamentos;
c) entrevistar usuários atuais e potenciais;
d) demonstrar como o novo sistema irá operar;
e) pesquisar documentação existente;
f) realizar um *brainstorming* com os usuários potenciais e usuais;
g) observar as estruturas e padrões;
h) documentar todo o processo e as reuniões.

8. REVISÃO DOS REQUISITOS

Todo requisito permite ao desenvolvedor explicar o entendimento de como os clientes querem que o sistema funcione. É

necessário informar aos projetistas quais funcionalidades e quais características o sistema resultante deve ter e informar a equipe de teste o que demonstrar para confirmar ao cliente que o sistema será entregue de acordo com o que foi solicitado.

Para verificar se os itens especificados serão alcançados é importante verificar sempre se:

a) os requisitos estão corretos;
b) os requisitos são consistentes;
c) os requisitos estão completos;
d) os requisitos são realistas;
e) cada requisito descreve algo que é importante para o cliente;
f) os requisitos podem ser verificados;
g) os requisitos podem ser rastreados.

Com relação ao sistema, após definirmos os requisitos funcionais e não funcionais, é importante verificarmos se:

a) as metas e os objetivos do *software* permanecem consistentes com as metas e os objetivos do projeto;
b) as interfaces necessárias para todos os elementos do sistema foram descritas;
c) o fluxo e a estrutura da informação são adequadamente definidos para o domínio da informação;
d) a modelagem e o diagrama estão claros;
e) há risco tecnológico do desenvolvimento;
f) os requisitos de *software* alternativos foram considerados;
g) a existência de inconsistências, redundâncias ou omissões;

O contato com o cliente foi completo.

9. DOCUMENTO DE DEFINIÇÃO DE REQUISITO

A documentação da definição de requisitos é fundamental em uma documentação e deverá ser descrita em uma linguagem

que os clientes entendam, pois a linguagem técnica será utilizada na modelagem do sistema.

Pfleeger (2004, p. 140) define os seguintes tópicos:

1) Em primeiro lugar, esboçamos o propósito geral do sistema. As referências a outros sistemas relacionados são incluídas, e incorporamos quaisquer termos e abreviações que possam ser úteis.
2) Em seguida, descrevemos o fundamento e os objetivos de desenvolvimento do sistema. Por exemplo, se um sistema deverá substituir uma abordagem existente, explicaremos porque o sistema existente é insatisfatório. Os métodos e procedimentos atuais são esboçados em detalhes suficientes para que possamos isolar esses elementos com os quais os clientes estão satisfeito daqueles que são insatisfatórios.
3) Se o cliente propôs uma nova abordagem para a solução do problema, esboçamos uma descrição da abordagem. Entretanto, lembre-se de que o objetivo dos documentos de requisitos é discutir o problema, e não a solução; o enfoque deve ser como o sistema satisfará as necessidades do cliente. Especificamente, se o cliente coloca qualquer restrição no desenvolvimento ou se existe qualquer suposição em especial a ser feita, o documento de definição deverá relacioná-la.
4) Uma vez que registramos essa visão geral do problema, descrevemos as características detalhadas do sistema proposto. Definimos os limites e as interfaces do sistema. As funções do sistema são explicitadas. Incluímos, também, uma lista completa de elementos e classes de dados e suas características. Detalhamos as relações entre dados e funções, assim como a entrada e saída de cada processo ou função. Os requisitos específicos de desempenho, tais como sincronismo, precisão e reação à falha, também são incluídos.
5) Finalmente, discutimos o ambiente em que o sistema irá operar. Incluímos os requisitos de suporte técnico, segurança e privacidade, e, também, quaisquer restrições especiais de *hardware* e *software* que deverão ser contempladas.

Após a definição dos requisitos e suas especificações, chegamos ao próximo passo, que é o de modelagem do sistema.

Após obter todos os requisitos, inicia-se o processo de modelagem. Pressman (2006, p. 19) define modelagem como sendo "a criação de modelos que permitem ao desenvolvedor e ao cliente, entender melhor os requisitos do *software* e o projeto que vai satisfazer estes requisitos".

Na Leitura Complementar 2, você verá que a modelagem juntamente com a definição e especificação de requisitos são documentos importantes para os desenvolvedores realizarem seu trabalho de forma eficiente e eficaz e cumprir os prazos e as estimativas.

10. PROCESSOS DE COMUNICAÇÃO

O desenvolvimento de um *software* é, na maior parte dos casos, motivado pelas necessidades de um cliente que deseja automatizar um sistema existente ou obter um novo sistema completamente automatizado. O *software*, porém, é desenvolvido por um desenvolvedor ou por uma equipe de desenvolvedores. Uma vez desenvolvido, o *software* será, provavelmente, utilizado por usuários finais, os quais não são, necessariamente, os clientes que originaram o sistema.

Isto significa que, de fato, três partes estão envolvidas no processo de desenvolvimento de um produto de *software*, são elas: o cliente, o desenvolvedor e os usuários. Para que o processo de desenvolvimento seja conduzido com sucesso, é necessário que os desejos do cliente e as expectativas dos eventuais usuários finais do sistema sejam precisamente transmitidos ao desenvolvedor.

Esse processo de transmissão de requisitos, do cliente e dos usuários ao desenvolvedor, invariavelmente, apresenta relativa dificuldade, considerando alguns aspectos, tais como: geralmente, os clientes não entendem de *software* ou do processo de desenvolvimento de um programa e o desenvolvedor, usualmente, não entende do sistema no qual o *software* vai executar.

Os dois aspectos mencionados provam que existe, efetivamente, um problema de comunicação a ser resolvido para que o processo de desenvolvimento seja bem sucedido. Dessa forma, compreendemos a importância desta etapa, que é obter as ideias do cliente sobre o problema a ser resolvido pelo sistema, as quais devem ser expressas na forma de um documento, se possível, que utilize ferramentas formais. A FAST pode auxiliar na comunicação do processo.

FAST

Há, na literatura, técnicas que facilitam a comunicação. Essas técnicas não são como mágicas, eliminando todos os problemas de comunicação, mas têm apresentado certa eficácia para diminuir os problemas de comunicação. Pressman (1995) propõe uma técnica denominada FAST (*Facilitaded Application Specification Techniques*). Esta técnica estimula a criação de uma equipe conjunta de clientes e desenvolvedores que trabalhem juntos para identificar o problema, propor elementos de solução, negociar diferentes abordagens e especificar um conjunto preliminar de requisitos de solução.

Uma Análise de Requisitos bem sucedida deve, normalmente, representar corretamente as necessidades do cliente e dos usuários, satisfazendo, às três partes envolvidas (incluindo o desenvolvedor). O que é verificado, em boa parte dos projetos de *software*, é que o cliente nem sempre consegue expressar perfeitamente quais são as suas reais necessidades, assim como os usuários têm dificuldades para exprimir as suas expectativas com relação ao que será desenvolvido. Um primeiro resultado desta etapa deve ser, sem dúvida, o esclarecimento a respeito do que são essas necessidades e expectativas.

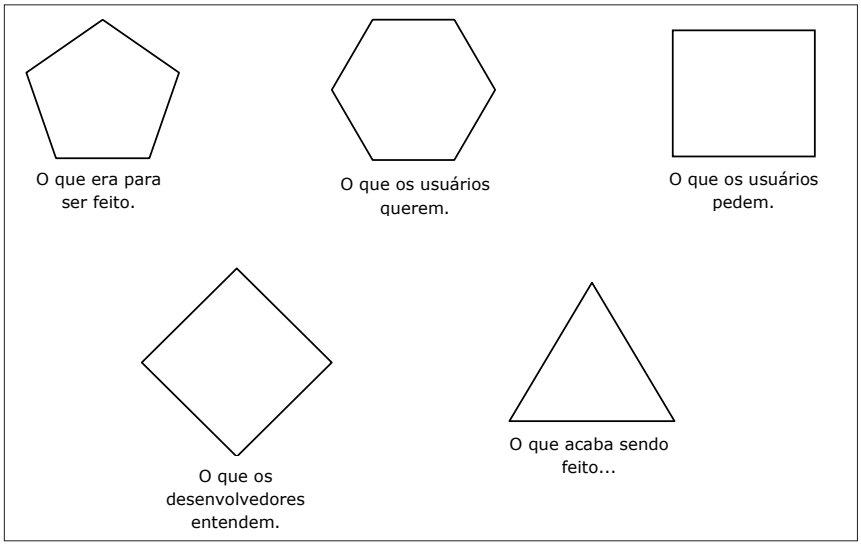
A obtenção bem sucedida de informações é um fator preponderante para o sucesso da Análise de Requisitos. A compilação das informações relevantes para o processo de desenvolvimento é uma tarefa bastante complexa, especialmente porque, muitas vezes, entram em jogo um conjunto de informações conflitantes. Além disso, quanto mais complexo for o sistema a ser desenvolvido, mais inconsistência haverá com relação às informações obtidas.

Neste contexto, o analista deve ter bom senso e experiência para extrair de todo o processo de comunicação as “boas” informações.

Assim, as dificuldades encontradas durante o levantamento de requisitos resumem-se em três itens, segundo Pressman (1995, p. 237), que são:

- **Obter informações pertinentes:** o reconhecimento do problema e a síntese da avaliação dependem da aquisição bem-sucedida de informações, provocada pela dificuldade de comunicação.
- **Cuidar da complexidade do problema:** o trabalho do analista é proporcional ao tamanho do problema, isto é, quanto maior o projeto e sua complexidade, maiores serão os esforços do analista.
- **Acomodar mudanças** (durante e depois do desenvolvimento): “Não importa onde se esteja no ciclo de vida do sistema, o sistema se modificará, e o desejo de mudá-lo persistirá ao longo de todo o ciclo”. (Primeira Lei da Engenharia de sistemas).

Paula Filho (2001) ilustra de forma bastante prática e interessante o problema de levantamento de requisitos, muitas vezes intensificados pela dificuldade de comunicação. Observe, na Figura 2, essa dificuldade.



Fonte: Adaptado de Paula Filho, (2001, p. 6)
Figura 2 *Evolução dos requisitos*

11. QUESTÕES AUTOAVALIATIVAS

Sugerimos que você procure responder, discutir e comentar as questões a seguir que tratam da temática desenvolvida nesta unidade. Se você encontrar dificuldade em respondê-las, procure revisar os conteúdos estudados para sanar suas dúvidas. A auto-avaliação pode ser uma ferramenta importante para você testar seu desempenho. Este é um momento ímpar para você fazer uma revisão desta unidade. Lembre-se de que no ensino a distância a construção do conhecimento se dá de forma cooperativa e colaborativa; portanto, compartilhe com seus colegas suas descobertas.

1) (ENADE, 2005) Requisitos de um sistema são frequentemente classificados como funcionais, não funcionais e de domínio. Qual a definição que melhor descreve requisitos não funcionais?
   a) São ferramentas automatizadas de apoio ao processo de desenvolvimento de sistemas.
   b) São requisitos que descrevem o que o sistema deve fazer, como deve reagir a determinadas entradas e como deve comportar-se em situações particulares.
   c) São requisitos que derivam do domínio da aplicação e que refletem características e restrições desse domínio.
   d) São requisitos que não estão diretamente relacionados com as funções específicas do sistema.
   e) São requisitos que especificam como deve ser testada uma parte do sistema, incluindo-se as entradas, os resultados esperados e as condições sob as quais os testes devem ocorrer.

2) Em relação aos princípios fundamentais da análise de requisitos, considere:

   Ajuda o analista a entender a informação, a função e o comportamento de um sistema, tornando a tarefa mais fácil e sistemática e tornando-se a base para o projeto, fornecendo ao projetista uma representação essencial do software, que pode ser mapeada num contexto de implementação.

   A afirmação anterior refere-se ao princípio:

   a) Do reconhecimento do problema.
   b) Da avaliação e síntese.
   c) Da especificação.
   d) Da revisão.
   e) Modelagem.

Fonte: prova aplicada em 03/2010 para o concurso do(a) TRT - 20ª REGIÃO (SE) - 2009, realizado pelo órgão/instituição Tribunal Regional do Trabalho da 20ª Região - Sergipe, área de atuação Jurídica, organizada pela banca FCC, para o cargo de Analista Administrativo - Informática, nível superior, área de formação Tecnologia da Informação.

Gabarito

Depois de responder às questões autoavaliativas, é importante que você confira o seu desempenho, a fim de que possa saber se é preciso retomar o estudo desta unidade. Assim, confira, a seguir, as respostas corretas para as questões autoavaliativas propostas anteriormente:

1) (d)

2) (e)

12. CONSIDERAÇÕES

Nesta unidade, você compreendeu a importância da atividade de levantamento de requisitos e pôde perceber que, quanto maior o esforço nesta fase, maiores as chances de que o *software* desenvolvido atenda às necessidades do cliente.

São levantadas nesta fase, quais as funções, o desempenho, as interfaces, a confiabilidade, enfim, todas as características que devem estar presentes no produto de *software*. Afinal, não basta ter um *software* correto, se ele não faz o que o cliente necessita.

Além disso, você aprendeu que há fatores que dificultam a obtenção dos requisitos. Não se esqueça de ler a Leitura Complementar 2. Lá você encontrará mais informações sobre o levantamento de requisitos.

Já na Unidade 5, você conhecerá os mecanismos da Engenharia de *Software* utilizados para gerenciar as mudanças, especialmente de requisitos, ocorridas durante o processo de desenvolvimento do *software*.

13. REFERÊNCIAS BIBLIOGRÁFICAS

PAULA FILHO, W. P. *Engenharia de Software*: fundamentos, métodos e padrões. 2. ed. Rio de janeiro: LTC, 2001.

PFLEEGER, S. L. *Engenharia de software*: teoria e prática. São Paulo: Prentice Hall, 2004.

PRESSMAN, R. S. *Engenharia de software*. São Paulo: Makron Books, 1996.

EAD

Gerenciamento de Configurações

5

1. OBJETIVO

- Compreender e discutir a importância do gerenciamento de configurações no desenvolvimento, teste e manutenção de produto de *software*.

2. CONTEÚDOS

- Gerenciamento de configurações – conceitos, contexto e utilidade.
- Linhas básicas.
- Processo de gerenciamento e configuração de *software*.
- Qualidade e gerenciamento de configuração.

3. ORIENTAÇÕES PARA O ESTUDO DA UNIDADE

Durante sua vida profissional, você irá se deparar com inúmeras variáveis, sendo as principais: as diversas aplicações para

software, as necessidades e prioridades dos clientes da indústria de *software* e as características distintas das equipes de desenvolvimento. Conviver com essas variáveis não é tão simples. Para que os modelos proposto pela Engenharia de *Software* sejam eficazes é importante que você saiba adaptá-los, considerando estas diversas variáveis.

Estas variáveis assumem valores distintos a cada novo projeto de *software*. Além disso, a instabilidade destas variáveis pode comprometer o projeto como um todo. Além do esforço para levantar os requisitos do *software* é vital que haja um acompanhamento no que diz respeito às mudanças que podem ocorrer nestes requisitos durante o processo de desenvolvimento.

Você verá que o gerenciamento de configuração não é uma tarefa simples de ser executada, mas é muito importante para manter a qualidade do *software* durante seu desenvolvimento. Se você tiver dúvidas durante o estudo desta unidade, interaja com seus colegas, discuta sobre o tema, peça auxílio ao seu tutor.

Sempre que necessário, consulte o glossário, para que você se recorde de cada conceito necessário ao melhor aprendizado desta unidade.

4. INTRODUÇÃO À UNIDADE

Na unidade anterior, você teve a oportunidade de compreender a importância dos conceitos de requisito e estimativa de *software.*

Nesta unidade, você terá a possibilidade de conhecer um pouco sobre o gerenciamento de configuração de um *software* e os itens que compõem esse processo.

Alterar um *software* é inevitável quando o estamos desenvolvendo, pois inúmeras ideias surgem, possibilitando sua melhoria, ou solicitações por parte do cliente ou, ainda, dos próprios gerentes de desenvolvimento, e a solução é alterar o projeto inicial para adequá-lo conforme as necessidades.

Geralmente, quando falamos em "projeto de desenvolvimento de *software*", estamos nos referindo a sistemas complexos, realizados por módulos por diversos desenvolvedores que podem estar, ou não, fisicamente distantes e conhecer, ou não, todos os membros da equipe de desenvolvimento.

É nesse cenário que problemas em modificações em rotinas, sistema ou até mesmo no escopo do projeto pode ser catastrófico se não for coordenada e gerenciada de maneira adequada.

Imagine que você está desenvolvendo um módulo, vai para casa descansar ao final de um dia de trabalho e, no dia seguinte, percebe que algumas rotinas foram alteradas. E o pior: alteradas por alguém que não conhecia direito o quanto aquelas rotinas influenciavam outras do próprio sistema ou que eram usadas em outro sistema de outro usuário.

Certamente, essa situação causaria algum desconforto entre os integrantes da equipe.

Antes de realizar qualquer mudança, elas devem ser analisadas; registradas antes de serem implementadas; relatadas aos que precisam tomar conhecimento delas ou controladas de modo que melhorem a qualidade geral do projeto e reduzam erros (PRESSMAN, 1996).

É nesse cenário de controlar as alterações que surge o **Gerenciamento de configurações de *software* (GCS)** ou ***Software configuration management* (SCM)**.

5. GERENCIAMENTO DE CONFIGURAÇÕES DE *SOFTWARE*

GCS é uma atividade abrangente que é aplicada em todo o processo de engenharia de *software*. Uma vez que uma mudança pode ocorrer a qualquer tempo, as atividades de GCS são desenvolvidas para:

1) Identificar mudanças.
2) Controlar mudanças.
3) Garantir que a mudança esteja sendo adequadamente implementada.
4) Relatar mudanças a outras pessoas que possam ter interesse nelas (PRESSMAN, 1996, p. 916)

GCS pode ser considerada como a arte de coordenar o desenvolvimento de *software* para minimizar os problemas de mudanças e alterações antecipadamente.

É importante salientar que GCS não é manutenção.

Segundo Pressman (2006 e 1996) e Pfleeger (2004), a manutenção é um conjunto de atividades da Engenharia de Software que acontece depois que o sistema foi entregue para o usuário, enquanto GCS é um conjunto de atividades da Engenharia de Software que acontece no momento que o projeto começa a ser desenvolvido e vai "até" ser retirado do mercado. É um conjunto de atividade contínua. (Figuras 1 e 2).

É fácil entender esse processo contínuo de controle. Portanto, sugiro que você leia novamente o que Pressman e Pfleeger apontam sobre esse assunto, percebendo a importância desse tema na sua vida profissional.

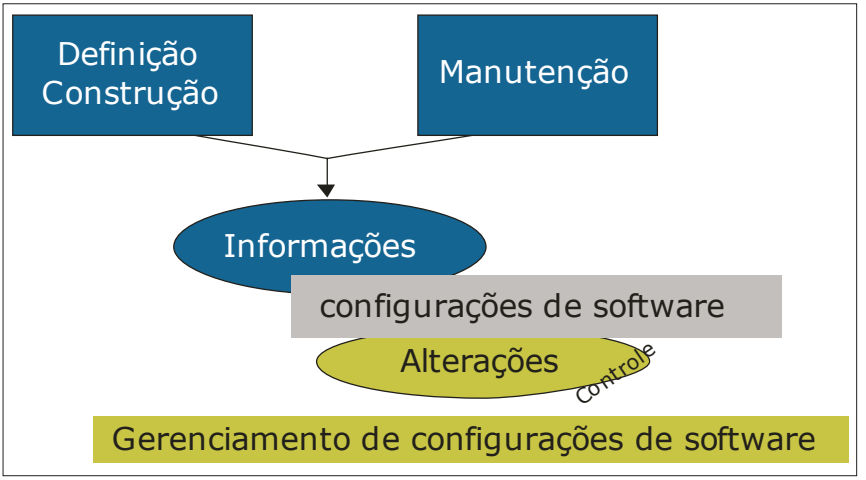


Figura 1 *Descrição ilustrativa sobre GCS*.

Temos definições de itens para a construção de um produto de *software* e a atividade de manutenção. Essas atividades geram informações que ficam armazenadas como item de configuração de *software*. Ao serem alteradas, precisam ser controladas, e esse processo é denominado gerenciamento de configuração de *software*.

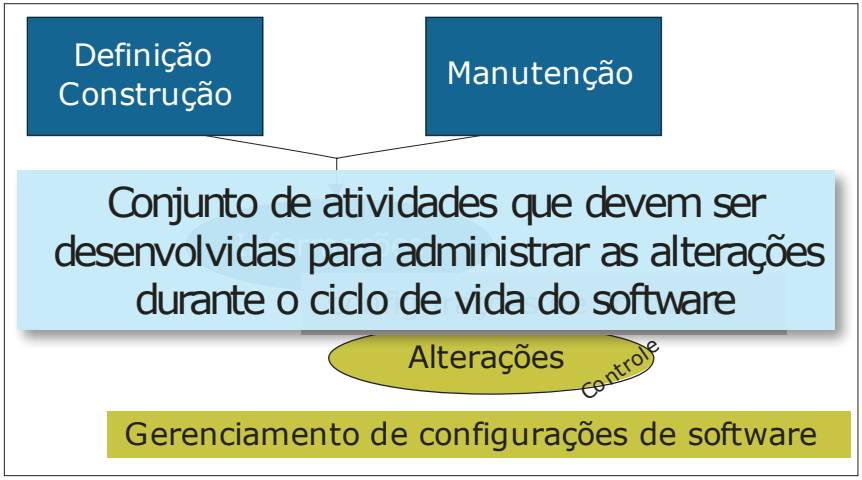


Figura 2 *Descrição ilustrativa sobre o que é GCS.*

> [...] uma meta primordial da metodologia de engenharia de software é melhorar a facilidade com que as mudanças podem ser acomodadas e reduzir a quantidade de esforços despendidos quando as mudanças são feitas (PRESSMAN, 1996, p. 917).

Como já vimos anteriormente, o resultado de todo processo de Engenharia de *Software* são informações que podem ser divididas em:

- programa de computador (executável e fonte, que, na verdade, é o resultado do projeto contratado);
- documentos que apoiam e esclarecem todo o processo e itens relacionados a ele;
- estrutura de dados (existentes e integradas que apoiam e são apoiadas tanto para o item 1 quanto para o 2).

Os itens gerados de todos esses componentes são chamados de **itens de configuração de *software* (ICS) ou *software configu-***

***ration* itens (SCI)** e a descrição e as informações relacionadas a eles deverão fazer parte da **documentação do Plano de Projeto**.

Seria mais prático e tranquilo se trabalhássemos por meio da garantia da não mudança do plano de projeto ou da especificação de *software*. Infelizmente, não é isso o que acontece. Mudanças ocorrem o tempo todo; aliás, isso é certo, se você está desenvolvendo um projeto de *software*, não importa em qual momento do ciclo de desenvolvimento você esteja, com certeza, mudanças serão implementadas por você e solicitadas pelos membros da equipe de desenvolvimento e/ou projetista e/ou gerente e/ou qualquer pessoa que conheça e "pense" no projeto. Isso é tão certo quanto dois mais dois são quatro!

Gerenciar as mudanças oferece-nos um ambiente estável de desenvolvimento, pois, alterações sem controle proporcionam um ambiente caótico e, na Unidade 1, vimos que a ES veio pra colocar ordem numa atividade que era caótica: a de desenvolvimento de *software* sem "ordem", "padrão" ou "gerenciamento adequado".

Há técnicas e conceitos que nos ajudam a coordenar essas mudanças, e é isso que veremos a partir de agora.

6. BASELINE – LINHAS BÁSICAS

Finalmente, trataremos de algo estável, em um ambiente tão instável quanto o de desenvolvimento de um projeto.

Segundo Rocha (2001), o Gerenciamento de Configuração é um processo que define as atividades e os procedimentos administrativos e técnicos a serem aplicados por todo o ciclo de vida do *software*, destinados a identificar e a definir os itens de *software* em um sistema e estabelecer suas linhas básicas. Além disso, deverá controlar as modificações e liberações dos itens; registrar e apresentar a situação dos itens e dos pedidos de modificação; garantir a conclusão, a consistência e a correção dos itens; controlar o armazenamento, a manipulação e a distribuição dos itens de *software*.

Todos os envolvidos em um projeto querem modificar algo, como, por exemplo, clientes querem modificar requisitos; administração, modificar a abordagem do projeto; desenvolvedores, modificar abordagens técnicas etc.

Com o passar do tempo e a familiaridade com o desenvolvimento de cada projeto, novas ideias, sugestões e mudanças surgem e o engenheiro de *software* enfrenta uma realidade "difícil": muitas das mudanças solicitadas são justificáveis, ou seja, merecem ser refeitas.

Uma **linha básica ou *baseline*** é um conceito de GCS que auxilia no controle das alterações sem impedir seriamente as mudanças justificáveis, ou seja, se uma mudança é justificável ela poderá ser realizada, mas por meio de uma linha básica. Esse conceito é utilizado também no planejamento e no gerenciamento de projetos para indicar o congelamento do documento plano do desenvolvimento do projeto de *software*.

Pressman (1996, p. 918) descreve-nos o mecanismo da linha básica por meio de uma analogia:

> Consideremos as portas da cozinha de um grande restaurante. Para eliminar colisões, uma porta é marcada como entrada e outra como saída. As portas têm quatro paradas que permitem que elas sejam abertas somente na direção apropriada.
>
> Se um garçom pegar um pedido na cozinha, colocá-lo numa bandeja e depois perceber que escolheu o prato errado, ele pode mudar para o prato certo rápida e informalmente antes de sair da cozinha.
>
> Se, entretanto, ele sair da cozinha, entregar o prato ao cliente e depois for informado de seu erro, ele deve seguir uma série de procedimentos: 1. Olhar a conta para determinar se ocorreu o erro; 2. Desculpar-se profundamente; 3. Retornar à cozinha pela porta ENTRADA; 4. Explicar o problema, e assim por diante.

Uma ***baseline*** é idêntica à porta da cozinha do restaurante. Antes de um item de configuração de *software* tornar-se uma linha básica, mudanças podem ser realizadas rápida e informalmente; após esse período, é necessário que as mudanças sejam feitas for-

malmente e por meio de procedimentos e etapas. Conforme demonstra a figura 3.

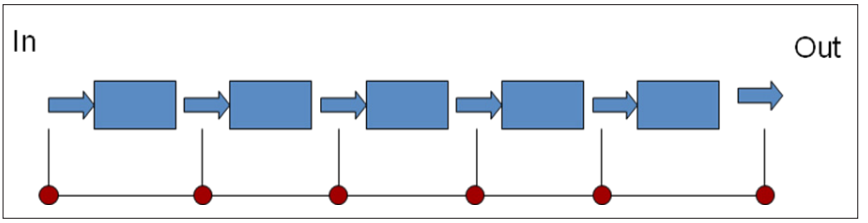


Figura 3 *Figura ilustrativa da definição das linhas básicas.*

Uma linha básica é um marco de referência. Antes de um item tornar-se uma linha básica, inúmeras alterações podem ocorrer, pessoas diferentes podem alterá-los (além do desenvolvedor), sugestões podem ocorrer; uma vez que o item foi considerado referência/marco, alterações só poderão ser realizadas por meio de solicitação e documentação formal.

Linhas básicas podem ocorrer em qualquer fase do projeto e, uma vez definidas informações sobre ela, é colocada no que chamamos de **repositório de *software*** ou **repositório de itens de configuração** ou **biblioteca de projeto**.

Quando alguém da equipe de engenharia de *software* desejar realizar uma mudança em um SCI definido com linha básica, esse item será copiado do BD (banco de dados) para a área particular do solicitante e um registro dessa solicitação será guardado em um arquivo chamado contábil. O solicitante poderá alterar o arquivo copiado até que todas as mudanças sejam completadas e, só então, ele voltará à base de dados inicial, já atualizada. Em algumas circunstâncias, o arquivo original é bloqueado e ninguém o pode utilizar até que todas as mudanças sejam executadas e o arquivo seja atualizado, revisto, aprovado e liberado novamente para uso de todos da equipe.

As linhas básicas podem ocorrer ao final de cada fase do desenvolvimento ou quando gerentes, projetistas, engenheiros ou desenvolvedores decidirem. Quando um item passou, é definido

como linha básica e pelo *baseline* ele é considerado *"baselined"*, ou é dito que o item "tornou-se uma linha básica"

Característica de um item *baselined*:

1) foi revisto formalmente e teve acordo das partes;
2) serve como base para o trabalho futuro;
3) armazenado no repositório dos itens de configuração (RIC);
4) alterado somente por meio de procedimentos formais de controle de mudança.

7. PROCESSO DE GERENCIAMENTO DE CONFIGURAÇÃO DE *SOFTWARE* (GCS)

Para controlarmos o processo de GCS, é necessário coordenarmos as alterações de algumas atividades, como veremos a seguir.

O processo consiste em diferentes etapas que descreveremos a seguir:

a) selecionar o item a ser gerenciado;
b) identificar os objetos;
c) controlar versão;
d) controlar mudanças;
e) controlar auditoria de configuração;
f) controlar relatório de *status* do sistema;
g) controlar a interface;
h) controlar fornecedores e informação.

A. Selecionar o item a ser gerenciado

Selecionar o item a ser gerenciado é uma das tarefas mais difíceis, pois eles devem ser escolhidos por meio de análise minuciosa e, novamente, é um processo que engloba técnica e criatividade.

Se não selecionar quais itens serão gerenciados e souber exatamente o porquê, poderá acontecer uma superdocumentação e um atraso nas funções de desenvolvimento, desnecessárias.

O **processo para execução dessa atividade** pode ser estruturado por:

a) selecionar o item;
b) descrever como o item se relaciona a outros no BD;
c) planejar as linhas de referências tendo como base o ciclo de desenvolvimento do projeto;
d) descrever "quando" pode ser alterado (descrever quais as justificativas que serão aceitas e em que circunstâncias);
e) descrever "como" pode ser alterado (se haverá necessidade ou não de suspender sua execução e utilização durante o período de mudança);
f) criar um documento e anexar ao plano do projeto;
g) inserir essas informações, de forma adequada, no repositório de itens de configuração.

B. Identificação de objetos na configuração de *software*

Para controlar e administrar os Itens de Configuração de *Softwares* (ICS), é necessário que cada um seja nomeado separadamente e, depois, organizado, utilizando uma abordagem orientada a objeto.

Cada item é considerado um objeto, por isso são necessários o nome, a descrição, uma lista de recursos e uma identificação.

Na qual:

1) **Nome**: nome dado ao objeto.
2) **Descrição**: lista de itens de dados que identifica o tipo de ICS (documento, programa, dados) que é representado pelo objeto; um identificador do projeto; informações sobre mudanças ou versões.
3) **Lista de recurso**: lista de entidades fornecidas, processadas, consultadas ou exigidas pelo objeto.
4) **Identificação**: identifica o objeto de configuração, também leva em consideração objetos relacionados que existem entre o objeto nomeado.

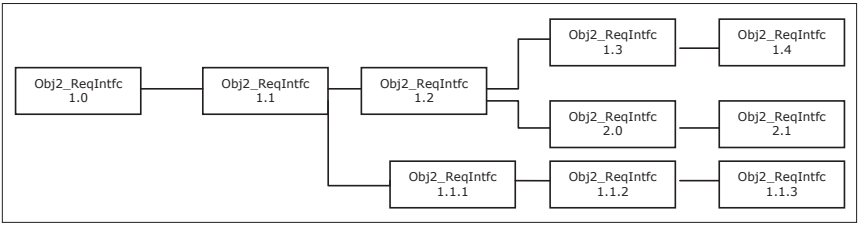
Exemplo de uma informação existente no ICS:

ITEM	ID	TIPO	NOME	VERSÃO	NOME COMPLETO
Especificação do projeto	ProjAA	EP	Obj1_escopo	V1.0	AAEP Obj1_escopo V10
Especificação de requisito	ProjAA	ER	Obj2_ReqIntfc	V2.0	AAER Obj2_ReqIntfc V20

Segundo Pfleeger (2004), os objetos podem ser:

- Básicos: unidade de texto criada por um engenheiro de *software* durante a análise, projeto, codificação ou teste. Um exemplo pode ser uma seção de especificação de requisitos ou uma listagem fonte para um módulo.
- Composto: uma coleção de objetos básicos e outros objetos compostos.

O controle das alterações desses objetos pode ser documentado por meio de gráficos, como podemos ver na Figura 4.



Fonte: Adaptado de Pressman (1996, p. 929)
Figura 4 *Representação gráfica do controle de objetos nos ICS também chamado de Gráfico de Evolução.*

A representação gráfica do controle de mudanças dos ICS, tratados como objeto, fará parte da **documentação do plano de projeto**.

C. Controle de versão

O controle de versão combina procedimentos e ferramentas para gerenciar diferentes versões de objetos de configurações.

Segundo Pressman (1996, p. 927),

> O gerenciamento de configurações permite que o usuário identifique configurações alternativas do sistema de software por meio da escolha de versões apropriadas. Isso é levado ao efeito ao associar atributos a cada versão de software e depois permitir que uma configuração seja especificada [e construída], descrevendo-se o conjunto de atributos desejados.

Como no item anterior, o controle de versões também pode ser representado graficamente, como veremos na Figura 5.

Uma das representações existentes para esse caso é a chamada **Árvore Delta**. Nela, são inseridas figuras delta, de acordo com sua relação e com a alteração para uma nova versão, por exemplo:

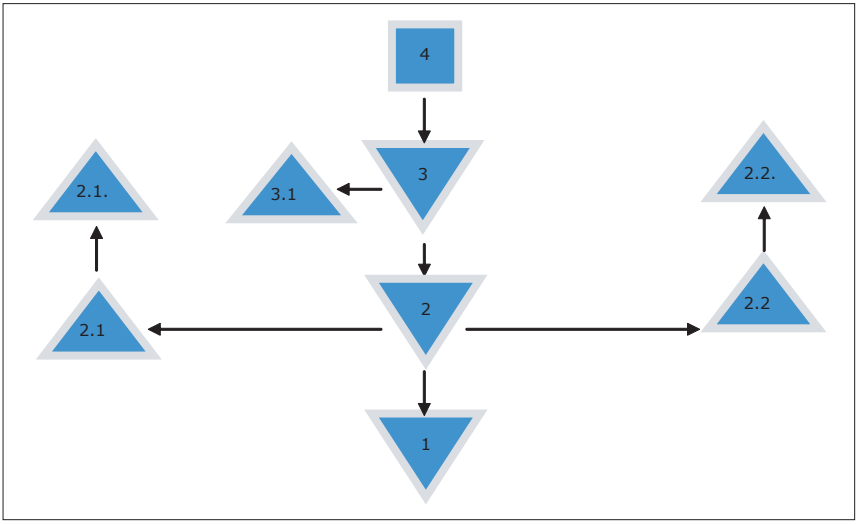


Figura 5 *Representação gráfica de Árvore Delta para o controle de versão.*

A representação gráfica do controle de versões dos ICS fará parte da **documentação do plano de projeto**.

D. Controle de mudanças

Não controlar as mudanças significa trabalhar em um ambiente caótico e, consequentemente, realizar trabalhos com qualidade questionável.

O controle de mudanças combina procedimentos humanos e ferramentas automatizadas para proporcionar um mecanismo de controle das mudanças.

Um pedido de mudança sempre será submetido a avaliações técnicas, analisados os potenciais efeitos colaterais, o impacto global sobre os outros objetos de configuração e funções do sistema e o custo projetado da mudança.

O resultado dessa avaliação é apresentado em um **documento** chamado **relatório de mudanças** e uma equipe pertencente à **Autoridade Controladora de Mudanças** (*Change Control Authority* – CCA) é que toma a decisão final sobre o *status* e a prioridade da mudança.

Uma ordem de mudança de engenharia (*Engineering Change Order* – ECO) é gerada para cada mudança aprovada. Na ECO, é descrita a mudança a ser realizada, as restrições que serão respeitadas e os critérios de revisão e auditoria.

Nesse momento, o processo a ser alterado passa por um *check-out*. Esse processo retira o item do repositório, cataloga-o no sentido de informar, a quem desejar, que ele está sendo alterado e, posteriormente, passa por um período de *check-in* (inserção do objeto no repositório). Observe as Figuras 6, 7 e 8.

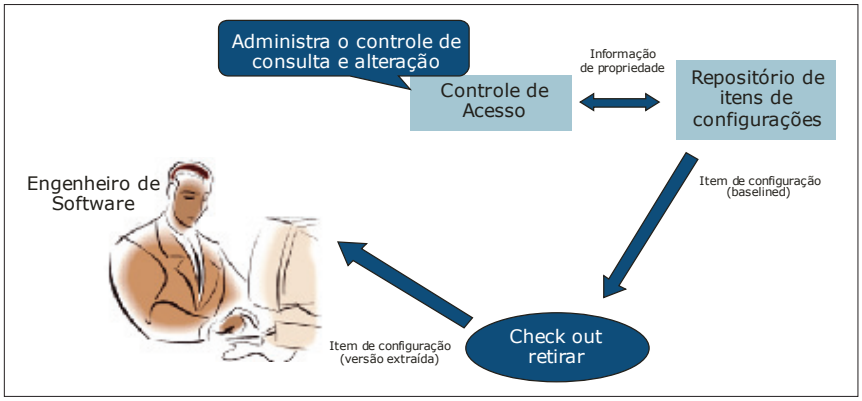


Figura 6 *Descrição ilustrativa sobre chek-out*.

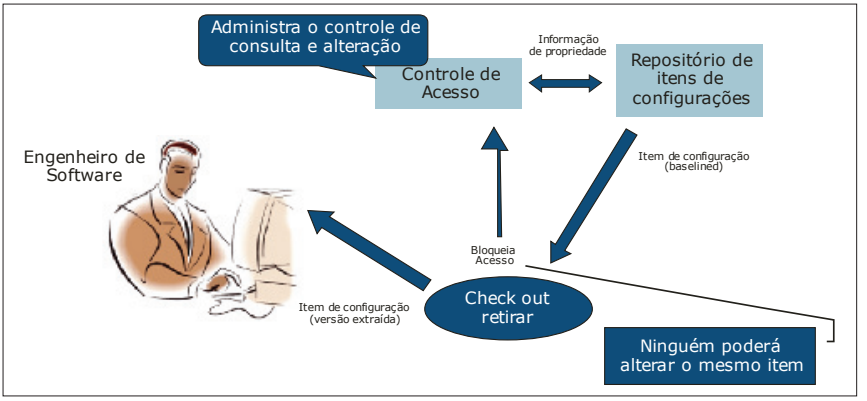


Figura 7 *Descrição figurativa do bloqueio do objeto no repositório*.

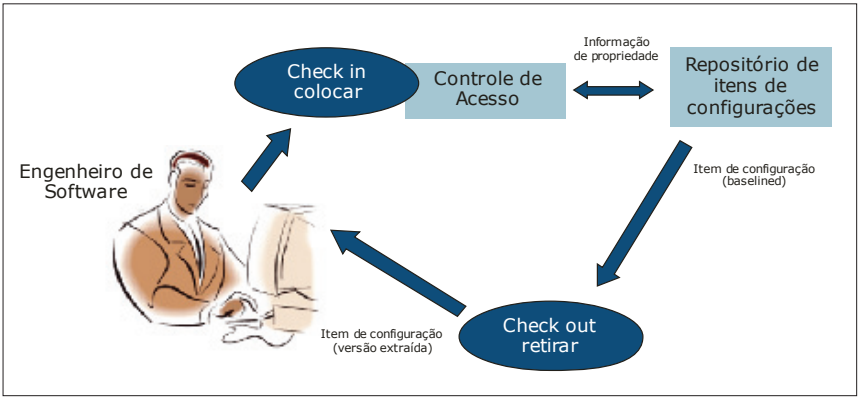


Figura 8 *Item alterado e inserido no repositório.*

Na Tabela 1, você encontrará a descrição do processo de mudança de acordo com Pressman (1996).

Tabela 1 Processo de controle de mudanças.

A. Necessidade de mudança é reconhecida.	
B. Usuário submete um pedido de mudança.	
C. Desenvolvedor avalia.	
D. Um *relatório de mudança* é gerado.	
E. Autoridade controladora de mudança toma uma decisão.	
1. A solicitação é colocada na fila para ser processada, uma ordem de mudança de engenharia é gerada.	2. O pedido de mudança é rejeitado.
1.1. Pessoas são designadas aos objetos de configuração.	2.1. O usuário é informado.

1.2. Os objetos (itens) de configuração passam por um *check-out* (registro de saída).	
1.3. É realizada a mudança.	
1.4. A mudança é revisada (passa por uma auditoria).	
1.5. Os itens de configuração que foram mudados passam por um *check-in* (registro de entrada).	
1.6. A linha básica para os testes é estabelecida.	
1.7. As atividades de garantia da qualidade e de testes são realizadas.	
1.8. As mudanças para inclusão no próximo lançamento (revisão) são "promovidas".	
1.9. A versão apropriada do *software* é reconstruída.	
1.10. As mudanças em todos os itens de configuração são revistas (passam por uma auditoria).	
1.11. As mudanças da nova versão são incluídas.	
1.12. A nova versão é distribuída.	

E. Auditoria de configurações

A identificação, o controle de versão e o controle de mudanças auxiliam o desenvolvedor de *software* a manter a ordem daquilo que, de outro modo, seria confuso e difícil de ser gerenciado.

Entretanto, a mais controlada ação de mudança é capaz de controlar e rastrear as mudanças até o ponto em que uma ECO é gerada.

Para garantirmos que a mudança foi adequadamente realizada, são necessárias:

- revisões técnicas formais;
- auditoria de configuração de *software.*

A **revisão técnica formal** deve ser realizada em todas as situações. Focaliza a exatidão técnica do objeto de configuração que foi alterado e avalia sua consistência com relação a outros ICS.

A **auditoria de configuração de *software*** complementa a revisão técnica formal ao avaliar um objeto de configuração quanto às características que, geralmente, não são consideradas durante a revisão.

Segundo Pressman (1996, p. 934):

- A auditoria pergunta e responde as seguintes questões:
- A mudança especificada na ECO foi feita? Outras modificações adicionais foram incorporadas?
- Uma revisão técnica formal foi realizada para avaliar a exatidão técnica?
- Os padrões de ES foram adequadamente seguidos?
- A mudança foi "realçada" no ICS? A data da mudança e o autor da mudança foram especificados? Os atributos do objeto de configuração refletem a mudança?
- Os procedimentos de GCS para anotar a mudança, registrá-la e relatá-la foram seguidos?
- Todos os ICS relacionados foram adequadamente atualizados?

F. Relato de *status* de configuração

A construção do **relato de *status* de configuração RSC** (*Configuration Status Report* – CSR) é uma tarefa que tem como objetivo responder às seguintes questões que relatam todo o desenvolvimento e a alteração:

a) O que aconteceu?
b) Quem fez?
c) Quando aconteceu?
d) O que mais é afetado?

Toda vez que um ICS recebe uma identificação nova é atualizado ou, quando uma auditoria é realizada, uma entrada de RSC é efetuada.

Essas informações, geralmente, são geradas de maneira *online* e podem ser acessadas pela administração e profissionais, para que saibam das mudanças realizadas.

G. Controle de interface

O controle de interface segue os mesmos padrões dos itens anteriores e coordena as mudanças de interfaces dos objetos que pertencem ao ICS.

As mudanças de interfaces solicitadas são coordenadas pelo processo de controle de mudanças e, por meio delas, é atualizada a base de dados que contém as informações sobre as interfaces/acesso a dados.

H. Controle de fornecedores de informação

Nem sempre toda informação e base de dados acessados por um projeto faz parte de um banco de dados pertencente ao cliente.

Algumas vezes, informações e banco de dados de terceiros são relacionados e denominados **base de dados de fornecedor**.

8. QUALIDADE E GERENCIAMENTO DE CONFIGURAÇÃO

A implementação da Gestão de Configuração de *Software* está totalmente relacionada à busca pela qualidade do produto desenvolvido. A atividade de Gerenciamento de Configuração é uma tarefa básica no modelo CMMI (*Capability Maturity Model Integration*) e é um processo de apoio nas normas ISO 12207 e ISO 15504, ou seja, esta atividade é considerada critério de diferenciação entre as empresas que desenvolvem *software*.

O Gerenciamento de Configuração é citado por Paula Filho (2003) como uma das disciplinas que sustentam os vértices do triangulo crítico da engenharia de *software*, que se resume a requisitos, prazo e custo.

No entanto, é visível a dificuldade das organizações em adotar conceitos e práticas relacionadas ao processo de GCS. Essa dificuldade, muitas vezes, está evidenciada pelo fato de que a implantação deste processo é uma atividade complexa que envolve definir uma abordagem adequada para o processo de GCS e a sele-

ção de ferramentas de apoio a esta abordagem. Por isso a implantação do GCS exige investimentos tanto em recursos financeiros como humanos que são escassos, especialmente para pequenas organizações, o que ocasiona, não raras vezes, um impedimento à adoção dessas práticas que poderiam contribuir para o seu crescimento e sobrevivência (CUNHA, 2004).

Segundo Paula Filho (2003), mesmo para projetos de porte muito pequeno, um mínimo de controle de versões é necessário para evitar o desperdício de trabalho. Com o controle de versões, é possível conservar as versões antigas dos artefatos, as quais contêm materiais que podem ser novamente aproveitados, mas é fundamental evitar que versões mais antigas venham, de forma inadvertida, a tomar o lugar de versões mais novas. Esse tipo de erro é comum mesmo em projetos individuais; e essa espécie de desperdício é facilmente evitada.

O GCS está relacionado à Gestão da Qualidade e contribui para a qualidade do projeto de *software* por meio do suporte às suas atividades relacionadas tanto aos aspectos gerenciais quanto técnicos.

9. QUESTÕES AUTOAVALIATIVAS

Sugerimos que você procure responder, discutir e comentar as questões a seguir que tratam da temática desenvolvida nesta unidade.

1) (ENADE, 2008) O gerenciamento de configuração de *software* (GCS) é uma atividade que deve ser realizada para identificar, controlar, auditar e relatar as modificações que ocorrem durante todo o desenvolvimento ou mesmo durante a fase de manutenção, depois que o software for entregue ao cliente. O GCS é embasado nos chamados itens de configuração, que são produzidos como resultado das atividades de engenharia de *software* e que ficam armazenados em um repositório. Com relação ao GCS, analise as duas asserções apresentadas a seguir.

   No GCS, o processo de controle das modificações obedece ao seguinte fluxo: começa com um pedido de modificação de um item de configuração, que leva à aceitação ou não desse pedido e termina com a atualização controlada desse item no repositório porque o controle das modificações dos itens de configuração baseia-se nos processos de check-in e check-out que fazem, respectivamen-

te, a inserção de um item de configuração no repositório e a retirada de itens de configuração do repositório para efeito de realização das modificações.

Acerca dessas asserções, assinale a opção correta.

a) As duas asserções são proposições verdadeiras, e a segunda é uma justificativa correta da primeira.
b) As duas asserções são proposições verdadeiras, e a segunda não é uma justificativa correta da primeira.
c) A primeira asserção é uma proposição verdadeira, e a segunda é uma proposição falsa.
d) A primeira asserção é uma proposição falsa, e a segunda é uma proposição verdadeira.
e) As duas asserções são proposições falsas.

2) Considere as seguintes afirmativas:

I – Seu escopo não deve abordar a definição do que será gerenciado, entretanto, deve definir o esquema a ser usado para identificar os itens de configuração.

II – Deve especificar as ferramentas usadas para o gerenciamento de configurações e os respectivos processos de uso, porém não deve entrar no mérito de definição de políticas a serem adotadas no controle de mudanças.

III – Deve estabelecer as responsabilidades pelos procedimentos de gerenciamento de configuração e ainda descrever a estrutura do banco de dados para o registro das informações de configuração.

Um plano de gerenciamento de configurações de sistemas de *software* em desenvolvimento deve estar de acordo com o que consta em:

a) II e III, apenas.
b) I, II, e III.
c) I, apenas.
d) II, apenas.
e) III, apenas.

Fonte: Prova aplicada em 04/2009 para o concurso do(a) PGE-RJ - 2009, realizado pelo órgão/instituição Procuradoria Geral do Estado do Rio de Janeiro, área de atuação Jurídica, organizada pela banca FCC, para o cargo de Técnico Superior de Análise de Sistemas e Métodos, nível superior, área de formação Tecnologia da Informação.

3) O processo Gerenciamento de configuração deve prover um modelo lógico da infra-estrutura de TI para identificar, controlar, manter e verificar as versões somente

a) dos componentes de hardware.
b) dos componentes de *software*.
c) dos itens de configuração.
d) dos itens de documentação.
e) das linhas de base (baselines).

Fonte: Prova aplicada em 09/2008 para o concurso do(a) TRT - 18ª Região (GO) - 2008, realizado pelo órgão/instituição Tribunal Regional do Trabalho da 18ª Região – Goiás, área de atuação Jurídica,

organizada pela banca FCC, para o cargo de Analista Judiciário - Tecnologia da Informação, nível superior , área de formação Tecnologia da Informação .

Gabarito

Depois de responder às questões autoavaliativas, é importante que você confira o seu desempenho, a fim de que possa saber se é preciso retomar o estudo desta unidade. Assim, confira, a seguir, as respostas corretas para as questões autoavaliativas propostas anteriormente:

1) (b)

2) (e)

3) (c)

10. CONSIDERAÇÕES

As questões estudadas nesta unidade estão relacionadas à Gerência de Configuração de *Software* (GCS) e têm como objetivo coordenar e auditar todas as mudanças ocorridas no sistema de desenvolvimento do *software* até o momento em que é retirado do mercado.

É importante salientar que não se trata de "manutenção" nem de procedimentos que ocorrem após a entrega do projeto ao cliente. GCS tem como objetivo responder, por meio de um processo estruturado, às seguintes questões:

a) O que mudou e quando?
b) Por que mudou?
c) Quem fez a mudança?
d) Podemos reproduzir essa mudança?

Podemos dizer que:

- O controle de versão é capaz de dizer o que mudou e quando mudou.
- O controle de mudanças é capaz de atribuir os motivos a

cada uma das mudanças.

- A auditoria responde às duas últimas perguntas: “Quem realizou a mudança?” e “podemos reproduzir a mudança?”, ou seja, cada pergunta está diretamente relacionada ao processo de controle de atualização e mudanças do projeto que é de vital importância para a **Engenharia de *Software***.

Na próxima unidade, abordaremos as questões relacionadas à garantia de qualidade de *software*.

11. REFERÊNCIAS BIBLIOGRÁFICAS

CUNHA, J. R. D. et al. *Uma abordagem para o processo de gerenciamento de configuração de software*. Disponível em: <http://www.inf.ufsc.br/resi/edicao04/artigo05.pdf>. Acesso em: 29 jun. 2007.

CUNHA, J. R. D. D.; PRADO A. F.; SANTOS A. C. Uma Abordagem para o Processo de Gerenciamento de Configuração de Software. *RESI – Revista Eletrônica de Sistemas e Informação,* 4. Ed., Vol.III, nº.I, Nov. 2004. Disponível em: < http://revistas.facecla.com.br/index.php/reinfo/article/view/145i>. Acesso em: 22 dez. 2009.

DIAS, A. F. *O que é gerência de configuração?* Disponível em:<http://www.pronus.eng.br/artigos_tutoriais/gerencia_configuracao/gerencia_configuracao.php>. Acesso em: 29 jun. 2007.

PAULA FILHO, W. P. *Engenharia de software*: fundamentos, métodos e padrões. Rio de Janeiro: LTC, 2003.

PFLEEGER, S. L. *Engenharia de software*: teoria e prática. São Paulo: Prentice Hall, 2004.

PRESSMAN, R. S. *Engenharia de software*. São Paulo: Makron Books, 1996.

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

EAD

Garantia de Qualidade de *Software*

6

1. OBJETIVOS

- Compreender os conceitos de Garantia de Qualidade de *Software*.
- Conhecer as atividades de Garantia de Qualidade de *Software*.
- Conhecer mecanismos de Garantia de Qualidade como a padronização e as revisões técnicas formais.
- Compreender os conceitos de confiabilidade, disponibilidade, manutenibilidade e segurança, relacionados à qualidade.

2. CONTEÚDOS

- Objetivos e conceitos de Garantia da Qualidade.
- Grupo de Garantia de Qualidade.
- Padronização.

- Revisões técnicas formais.
- Confiabilidade de *software*.

3. ORIENTAÇÕES PARA O ESTUDO DA UNIDADE

Antes de iniciar o estudo desta unidade, é importante que você leia as orientações a seguir:

1) Quando se fala sobre qualidade logo temos em mente os princípios de qualidade total, lembramos de Deming e Juran, pensamos também em normas ISO, entre outros. Enfim, ligamos a busca pela qualidade à indústria de manufatura. Porém, é importante ressaltar que os clientes de qualquer indústria estão cada vez mais exigentes, o mesmo ocorre com a indústria do *software*. Os sistemas computacionais estão apoiando as mais diversas atividades humanas, das mais simples às mais complexas. Há atividades em que os erros seriam desastrosos. Assim, é importante a preocupação com a confiabilidade dos *softwares* desenvolvidos. No entanto, não é possível inserir qualidade em um produto já desenvolvido, características de qualidade são inseridas durante todo o processo produtivo.
2) Para o estudo desta unidade, tenha em mente que na atual fase da indústria de *software* não é mais possível o desenvolvimento de sistemas de forma artesanal. Desenvolvedores que ainda trabalham assim terão poucas chances de se manterem no mercado.
3) Aproveite ao máximo os conceitos apresentados nesta unidade. Sempre que possível, interaja com seus colegas. Compartilhe suas dúvidas e aprendizado.
4) Aceite o desafio de construir uma comunidade interativa, pois, além de significativos ganhos para sua vida pessoal e profissional, essa interação irá colocá-lo em consonância com as novas exigências do mundo científico e profissional. Afinal, conhecer é exercer seu direito de cidadania.

4. INTRODUÇÃO À UNIDADE

Na Unidade 5, você teve a oportunidade de aprender alguns conceitos relacionados à atividade de Gerenciamento de Configuração de *Software*. E, dessa forma, conscientizou-se de que a necessidade de mudanças é uma constante no desenvolvimento de *software* e o gerenciamento de configuração objetiva administrar as mudanças durante o processo produtivo para que o foco na qualidade do produto seja mantido.

Nesta unidade, você conhecerá a Garantia da Qualidade de *Software*. Esta atividade juntamente com o Gerenciamento de Configuração são mecanismos da Engenharia de *Software* que apoiam o processo de *software*.

5. CONCEITOS E OBJETIVOS

Com os estudos realizados até o momento, você aprendeu que a Engenharia de *Software* tem como meta entregar produtos de *software* dentro do prazo e custo estimados e com a qualidade que o cliente necessita.

Por meio de um processo de desenvolvimento eficiente e de uma gestão de projetos eficaz, é possível diminuir os riscos e conseguir cumprir os prazos e custos estimados. Mas e a qualidade? Como ela pode ser assegurada?

Em primeiro lugar, é importante entender o que é qualidade. Informações detalhadas sobre o termo qualidade serão descritas na Unidade 10. Para esta unidade, os conceitos aqui expostos são suficientes.

Das inúmeras definições existentes, seguem algumas para **qualidade de *software***:

Segundo Paula Filho (2001), a qualidade de um produto é a conformidade com seus respectivos requisitos.

Conformidade a requisitos funcionais e de desempenho explicitamente declarados, a padrões de desenvolvimento claramente documentados e a características implícitas que são esperadas de todo *software* profissionalmente desenvolvido (PRESSMAN, 2006).

Pressman (2006) detalha sua definição anterior, enfatizando três pontos importantes, que são:

- os requisitos do *software* são a base pela qual a qualidade é medida. A falta de conformidade com os requisitos quer dizer que há falta de qualidade.
- as normas especificadas definem um conjunto de critérios de desenvolvimento que guia o modo pelo qual o *software* é submetido à engenharia. Se os critérios não são seguidos, ocorrerá, falta de qualidade.
- um conjunto de requisitos implícitos frequentemente não é mencionado (por exemplo, o desejo da facilidade de uso e boa manutenibilidade). Se o *software* satisfaz seus requisitos explícitos, mas deixa de satisfazer requistos implícitos, sua qualidade é suspeita.

Paula Filho (2001) faz uma analogia interessante sobre qualidade, vamos conhecer. Um carro popular pode ser de boa qualidade e um carro de luxo pode ser de má qualidade. O que define a qualidade é a comparação com os respectivos requisitos: o confronto entre a promessa e a realização de cada produto.

Agora que você entendeu o que significa qualidade, vejamos o significado de Garantia de Qualidade de *Software*.

Vejamos, a seguir, como alguns autores definem a **Garantia de Qualidade de *Software***.

A Garantia de Qualidade de *Software* consiste em um conjunto de funções para auditar e relatar que avalia a efetividade e completeza das atividades de controle de qualidade. A meta da Garantia de Qualidade é fornecer à gerência os dados necessários

para que fique informada sobre a qualidade do produto. (Pressman, 2006, p. 579). Garantia da Qualidade de *Software* é uma atividade preventiva em relação aos problemas da qualidade de produto que poderiam surgir (PAULA FILHO, 2001).

Para Sommerville (2005), garantia de Qualidade de *Software* é o estabelecimento de uma estrutura de procedimentos e de padrões organizacionais que conduzam ao *software* de alta qualidade.

A meta da Garantia de Qualidade é fornecer à gerência os dados necessários para que fique informada sobre a qualidade do produto, ganhando, assim, compreensão e confiança de que a qualidade do produto está satisfazendo as suas metas (PRESSMAN, 2006). A qualidade de um produto é a conformidade com seus respectivos requisitos (PAULA FILHO, 2001).

"Todo programa faz alguma coisa direito, apenas pode não ser a coisa que desejamos que ele faça" (PRESSMAN, 2006, p. 724).

Características de qualidade não podem ser inseridas em um produto depois de pronto, e com o *software* não é diferente. A qualidade do produto de *software* pode ser vista como uma consequência da qualidade do processo usado para desenvolvê-lo.

Você, certamente, encontrará em livros a sigla SQA, do inglês *Software Quality Assurance*, que é a tradução para Garantia de Qualidade de Software. Para efeito de praticidade, de agora em diante, nesta Unidade, usaremos somente a sigla GQS para nos referirmos a esta atividade.

Segundo Paula Filho (2001), os objetivos da Garantia de Qualidade de *Software* (GQS) são:

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

b) Dar suporte ao trabalho das equipes projetistas de *software* garantindo a qualidade de atividades e resultados de cada projeto.

c) Verificar as atividades e os resultados dos projetos de *software*.

d) Fornecer à gerência informação atualizada sobre o *status* dos projetos e sua conformidade com os padrões da qualidade da organização.

e) Mais especificamente, a GQS tem como objetivos:

f) seguir os padrões e as recomendações em vigor na organização;

g) usar métodos técnicos apropriados de engenharia de *software*, conforme os padrões em vigor;

h) compatibilizar entre si os projetos do Plano de Qualidade do *Software* com o Plano de Desenvolvimento de *Software* de cada projeto;

i) realizar revisões formais e auditorias requeridas pelo Plano de Qualidade do *Software*, para todos os projetos, por equipes independentes em relação ao projeto em questão;

j) elaborar a documentação requerida pelo Plano de Qualidade do *Software*, durante o desenvolvimento do *software*;

k) obedecer aos procedimentos e mecanismos previstos pelo Gerenciamento de Configuração do *Software*;

l) realizar os testes previstos pelo Plano da Qualidade do *Software*, priorizando os testes de áreas de maior risco;

m) completar e aprovar todas as atividades do Plano de Desenvolvimento, antes do início das próximas atividades;

n) identificar, o mais breve possível, os problemas da não conformidade das atividades e resultados dos projetos e tratá-los até a sua completa resolução;

o) auditar por profissionais externos todo projeto, quando necessário;

p) obedecer aos padrões e recomendações em vigor na organização das próprias atividades de GQS.

De acordo com o **Processo Praxis** padrão, as atividades de GQS são:

- **Planejamento de qualidade:** é pertinente a um projeto específico, e é executado pelo gerente do projeto, ou por quem designar. Seu principal produto é o Plano de Qualidade de *Software*, que deve estar em acordo com o Plano de Desenvolvimento do projeto em questão. Em grandes projetos, é importante a existência de um Plano de Qualidade separado do Plano de Desenvolvimento. Em projetos pequenos, características de qualidade podem ser incorporadas ao Plano de Desenvolvimento de *Software*.
- **Auditoria da qualidade:** é realizada pelo Grupo de GQS. Esta atividade é responsável por atingir os objetivos traçados anteriormente, tais como conformidade com padrões, rastreamento de projetos, compatibilidade do processo com o plano etc. Os resultados de cada auditoria devem ser documentados em relatório.
- **Atividade de verificação:** nesta atividade, são verificadas as atividades de outros grupos de suporte (como grupo de gerenciamento de configuração, grupo de engenharia, grupo de treinamento etc.) para verificar se eles estão procedendo de acordo com as políticas, os processos e os padrões adotados pela organização. Todas as informações devem ser reportadas à gerência da organização.

O **Processo Praxis** objetiva dar suporte a projetos didáticos em disciplinas de Engenharia de *Software*. A sigla PRAXIS significa PRocesso para Aplicativos eXtensíveis InterativoS, refletindo uma ênfase no desenvolvimento de aplicativos gráficos interativos, baseado na tecnologia orientada a objetos.

Durante as fases do desenvolvimento do *software*, podemos notar que alguns defeitos são introduzidos, inconscientemente, pelas pessoas. Afinal, como diz o ditado: errar é humano. A GQS é a atividade responsável por remover esses erros, ou parte deles,

à medida que são inseridos. Assim, a GQS é uma atividade que acompanha todo o processo produtivo do *software*.

Segundo Paula Filho (2001), defeitos que não são removidos durante as fases de desenvolvimento de *software* serão detectados posteriormente, o que fará com que a correção seja mais onerosa, porque quanto mais tarde um defeito é corrigido, mais cara é a sua correção. Geralmente, quanto melhor a qualidade do processo, menor é o tempo de desenvolvimento. Processos mais qualificados permitem a detecção e a eliminação precoce de defeitos.

Observe, a seguir, a descrição das tarefas de GQS, segundo Pressman (1995).

1) **Aplicação de métodos técnicos:** a qualidade não é inserida depois de o produto estar pronto, portanto, a GQS inicia-se com um conjunto de métodos e ferramentas técnicas que ajudam o analista a conseguir uma especificação de elevada qualidade e o projetista a desenvolver um projeto de elevada qualidade também.
2) **Revisão técnica formal:** assim que cada produto da engenharia de *software* estiver pronto, a sua qualidade deve ser avaliada por meio de revisões.
3) **Atividade de teste de *software*:** o grupo de GQS deve trabalhar com a equipe de testes.
4) **Aplicação de padrões:** se a organização segue algum padrão formal é necessária a atividade de GQS para garantir que eles sejam seguidos.
5) **Controle de mudanças:** colabora com o gerenciamento da configuração de *software*, fazendo com que a qualidade do *software* seja assegurada independentemente das mudanças que serão feitas.
6) **Medição:** a GQS deve coletar medidas durante o processo de *software* para rastrear a qualidade do *software* e avaliar o impacto das mudanças.
7) **Manutenção de registros:** a atividade de GQS oferece procedimentos de coleta e disseminação de informação. Os resultados de revisões, auditorias, controle de

mudanças, testes e outras atividades de GQS devem ser registradas no histórico de um projeto e levados ao conhecimento dos interessados.

Essas atividades de manutenção de registros são úteis não somente para fomentar a comunicação entre os envolvidos em um projeto, mas também para manter dados históricos de projetos atuais para serem usados no planejamento de projetos futuros, diminuindo, assim, os riscos de estimativas erradas. Lembre-se de que a Engenharia de *Software* não é uma ciência exata, ela é baseada, fundamentalmente, em dados de projetos do passado, com a filosofia: repita os acertos, modifique o que deu errado.

6. GRUPO DE GARANTIA DE QUALIDADE DE SOFTWARES (GQS)

Geralmente, somos mais eficazes para encontrar erros inseridos por outras pessoas do que os nossos próprios erros. Não é assim? Na GQS não é diferente, as revisões mais eficientes são aquelas realizadas por revisores distintos ao processo, ou seja, pessoas que não participaram do processo de autoria. Assim como na fase de testes, a GQS é mais eficaz quando coordenada por um grupo distinto ao grupo que construiu o *software*.

Para Pressman (2006), o grupo de GQS atua como representante residente do cliente, ou seja, deve olhar o *software* pelo ponto de vista do cliente. O grupo de GQS tem responsabilidade pelo planejamento, supervisão, registro, análise e relato da garantia da qualidade e o objetivo de GQS é auxiliar a equipe de desenvolvimento a conseguir um produto final de alta qualidade. Além disso, é fundamental que o grupo de GQS não seja a mesma equipe responsável pelo gerenciamento do projeto para não comprometer prazos, custos e recursos.

Uma tarefa difícil dos gerentes de *software* é alocar pessoas para trabalhar com GQS (Garantia de Qualidade de *Software*).

Essa prática de iniciar novas contratações em SQA é uma solução parcial que pode ser efetiva apenas se há pessoas experientes no mercado. Vale ressaltar que recrutar pessoas para trabalhar em SQA não é uma tarefa fácil, porque os profissionais de *software*, geralmente, preferem atribuições de desenvolvimento e a gerência, certamente, quer atribuir aos melhores projetistas o trabalho de projeto. O esquema de rotatividade também pode ser efetivo, mas, infelizmente, o desenvolvimento de *software* é adepto a transferir pessoas para trabalhar em GQS e não "as pegar" de volta.

Uma solução efetiva é requerer que todos os novos gerentes de desenvolvimento sejam promovidos para trabalharem em GQS. Isso poderia significar que potenciais gerentes poderiam passar entre seis meses e um ano em GQS, antes de serem promovidos à gerência. Essa é uma medida extrema, mas pode ser efetiva. Para o trabalho de GQS ser efetivo, deve haver bons profissionais na equipe e um completo apoio da gerência, no sentido de investimento (BUENO, 2006).

7. PADRÕES DE *SOFTWARE*

Como você pode perceber, a manutenção de padrões e procedimentos formais é um dos objetivos da GQS.

De acordo com Sommerville (2005, p.460), há dois tipos de padrões que devem ser estabelecidos como parte do processo de garantia da qualidade, observe a seguir:

1) **Padrões de produto:** são padrões que se aplicam ao produto de *software* em desenvolvimento. Eles incluem padrões de documentos, como a estrutura do documento de requisitos a ser produzido; padrões de documentação, como cabeçalhos de comentários para a definição de uma classe de objeto e padrões de codificação, que definem como uma linguagem de programação deve ser utilizada.

2) **Padrões de processo:** são os padrões que definem os processos a serem seguidos durante o desenvolvimento de *software*. Eles podem incluir definições de especificações, processos de projeto e validação, e uma descrição dos documentos que devem ser gerados no curso desses processos.

Para Sommerville (2005), há uma série de razões pelas quais os padrões de *software* são considerados importantes, tais como:

- Fornecer um agrupamento das melhores práticas ou, pelo menos, das mais adequadas. Esse conhecimento é adquirido depois de um grande número de tentativas e erros. Construir esse conhecimento dentro de um padrão evita a repetição de erros cometidos anteriormente. Os padrões registram sabedoria e valor para a organização.
- Fornecer infraestrutura em torno da qual o processo de GQS pode ser implementado. Considerando que os padrões incluem as melhores práticas, o controle de qualidade simplesmente tem como objetivo garantir que os padrões foram seguidos de maneira adequada.
- Ajudar em termos de continuidade, quando uma tarefa é realizada por uma pessoa ela pode ser assumida e continuada por outra pessoa. Os padrões asseguram que todos os engenheiros de uma organização adotem as mesmas práticas. Consequentemente, o esforço de aprendizado é reduzido quando se inicia um novo trabalho.

A padronização permite que a qualidade da atividade não seja dependente de uma pessoa, mas que esta qualidade esteja institucionalizada (enraizada) na organização. Isso também diminui os riscos causados pela rotatividade de funcionários.

O desenvolvimento de padrões é um processo complexo. Os padrões mais aceitos são, obviamente, os desenvolvidos por organizações nacionais ou internacionais com competências inquestionáveis, tais como: US DoD (Departamento de Defesa dos Estados Unidos, ANSI, IEEE, ISO, ABNT etc.).

A padronização pode ser considerada burocrática e desnecessária por muitos engenheiros de *software*. Para evitar estes pensamentos, é necessário, conforme Sommerville (2005):

- Envolver os engenheiros de *software* no desenvolvimento de padrões para o produto. Eles devem compreender os benefícios dos padrões e se comprometer com eles. Para tanto, é fundamental que os documentos de padronização, não só definam as regras, mas também explicitem as razões pelas quais serão utilizados.
- Revisar e modificar os padrões regularmente é importante para que os engenheiros de *software* reflitam as constantes evoluções tecnológicas. Um manual de padrões é necessário e deve evoluir com as circunstâncias e tecnologias mutáveis.
- Fornecer ferramentas *CASE* para apoiar padrões sempre que possível. Os padrões que não possuem apoio de ferramentas são alvos de queixas, devido ao difícil trabalho de implementá-los.

Muitas vezes, pode ser necessária a adaptação dos padrões à realidade da organização que a utiliza. Mas essas adaptações devem ser realizadas de maneira criteriosa e acompanhada pelo gerente de projeto e responsáveis pela qualidade.

8. REVISÕES TÉCNICAS FORMAIS

Revisão e qualidade

Revisões Técnicas Formais são consideradas filtros da Engenharia de *Software* que têm como propósito encontrar problemas de qualidade.

Além de um mecanismo utilizado pela GQS, as revisões técnicas formais também são consideradas um meio utilizado pelos gerentes de projeto para fazer o rastreamento de controle do projeto em relação ao que foi descrito no Plano de Desenvolvimento. A qualidade do projeto é avaliada por meio de várias revisões desse tipo.

Segundo Pressman (2006), a atividade central que promove a avaliação da qualidade é a Revisão Técnica Formal (*Formal Technical Review – FTR*). Essa revisão é um encontro realizado pelo pessoal técnico com o propósito único de descobrir problemas de qualidade. Note que há algumas restrições que devem ser observadas em relação ao pessoal técnico, tais como:

- na reunião, devem estar envolvidas entre três e cinco pessoas;
- para participar da reunião é fundamental que haja uma preparação, a qual não deve ser superior a duas horas de trabalho de cada pessoa envolvida;
- a reunião de revisão deve durar menos de duas horas.

A revisão, realizada pelo pessoal técnico, deve concentrar-se em uma parte específica do projeto, por exemplo, ao invés de revisar todo o *software*, é preciso revisar módulos separadamente, pois, ao limitar o foco, há maior probabilidade de encontrar erros.

Dessa forma, podemos dizer que o foco da revisão técnica formal é um produto de trabalho (por exemplo, uma parte da especificação de requisitos, o código-fonte de um componente etc.). Uma vez desenvolvido esse produto, a revisão pode ser realizada. O gerente de projeto envia cópias do produto aos revisores que devem se familiarizar e analisar o produto em questão. A reunião é marcada e solicita-se a presença do produtor (pessoa responsável por desenvolver o produto, por exemplo, um analista ou programador), dos revisores (representantes de grupos interessados, por exemplo, o grupo de GQS de gerenciamento de configuração) e do gerente de projeto. Durante a reunião, o produtor apresenta o produto de trabalho em questão, os revisores informam sobre as análises já realizadas e o gerente verifica se a execução dos produtos está de acordo com o planejamento do projeto. Quando erros são encontrados, esses devem ser registrados. Ao final da reunião, de acordo com Pressman (2006), três situações são possíveis:

1) O produto é aceito sem modificações.

2) O produto é rejeitado, devido a erros graves.
3) O produto é aceito mediante a condição de que os erros encontrados serão corrigidos.

Por ser considerado um procedimento formal, essa revisão deve ser registrada pela equipe responsável pela revisão, que deverá confeccionar um relatório bem resumido com, aproximadamente, uma página, informando o que foi revisado, quem participou da revisão, quais foram as descobertas e as conclusões. Ao final, o relatório deve ser entregue aos interessados. Os interessados são os desenvolvedores envolvidos na produção e no teste do item avaliado. Além disso, é importante que se crie um mecanismo que garanta que as correções serão, de fato, implementadas.

Um benefício óbvio de revisões técnicas formais é a descoberta de erros antes que eles se propaguem para as próximas fases do processo de *software*. Segundo Bueno (2006), estudos de algumas indústrias indicam que as atividades de projeto introduzem entre 50% e 60% de todos os erros durante o processo de *software*. Entretanto, técnicas de revisões formais têm atingido um percentual de 75% na descoberta desses erros, reduzindo o custo dos passos subsequentes.

As revisões técnicas formais podem ser executadas durante todo o processo de *software*, podendo reunir o grupo de GQS aos respectivos envolvidos na execução de cada uma das fases do processo. Para ilustrar, a seguir apresentamos as fases de desenvolvimento e algumas questões que podem ser levantadas e respondidas nas revisões em cada fase (PRESSMAN, 1995):

1) **Engenharia de sistema**.

   As funções principais são definidas de forma clara e não ambígua?

   As interfaces entre os elementos são definidas?

2) **Planejamento do projeto:** a revisão pode avaliar os riscos.

   A terminologia é clara?

   Os riscos foram definidos?

   O cronograma é consistente?

3) **Análise**.

   A divisão do problema em partições é completa?

   Os requisitos são consistentes com os prazos, recursos e orçamento?

4) **Projeto**.

   A manutenibilidade foi levada em consideração?

   O projeto é modular?

5) **Codificação**.

   O projeto foi adequadamente traduzido?

   O código foi documentado?

6) **Plano de teste e Testes**.

   O cronograma de teste foi explicitamente definido?

   O tratamento de erros está testado?

7) **Manutenção**.

   Foram considerados possíveis efeitos colaterais associados à mudança?

   A solicitação de mudança foi documentada?

9. CONFIABILIDADE DE *SOFTWARE*

Características como confiabilidade, disponibilidade e segurança são fundamentais para a qualidade global do produto. Portanto, para que o *software* tenha qualidade é necessário que ele seja confiável, disponível, manutenível e seguro.

Para Pressman (1995, p. 768) “se um programa deixar de funcionar repetida e frequentemente, pouco importa se outros fatores de qualidade são aceitáveis”.

Confiabilidade de *software* pode ser definida, segundo Pfleeger (2004), como a probabilidade de um sistema operar sem falhas, sob certas condições, em um determinado intervalo de tempo.

Agora, você precisa saber o que é uma **falha**.

Para Pressman (1995), falha é a não conformidade aos requisitos do *software*.

Falhas no hardware podem ser causadas devido ao desgaste físico e podem ser resultantes de problemas de projeto, implementação ou mudanças de requisitos.

Gráficos com níveis de falhas de *software* e de *hardware* foram apresentados no estudo da Unidade 1 desta disciplina.

As falhas podem ser classificadas em níveis de gravidade e variar de simples aborrecimentos a grandes prejuízos. A correção de falhas pode levar segundos ou semanas e, até mesmo, meses de trabalho. Além disso, a correção de uma falha pode inserir novos erros, que poderão gerar novas falhas.

Pfleeger (2004) classifica as falhas em:

1) **Catastrófica**: uma falha que pode causar destruição ou perda do sistema.
2) **Crítica**: uma falha que pode causar graves prejuízos ou danos ao sistema principal, que resulta na perda de uma missão.
3) **Marginal**: uma falha que pode causar um prejuízo ou um dano menor ao sistema, que resulte em atraso, perda de disponibilidade ou degradação da missão.
4) **Pequena**: uma falha que não é séria o suficiente para causar prejuízos ou danos ao sistema, mas que resulta em manutenção ou reparo não programado.

Uma métrica de confiabilidade bastante citada é a defeitos/KLOC (mil linhas de código). Alguns pesquisadores recomendam que a confiabilidade de um sistema pode ser medida por meio do tempo médio entre a ocorrência de falhas (Mean Time Between Failure - MTBF).

As falhas influenciam não apenas na confiabilidade do *software*, mas também na sua disponibilidade e na manutenção. Observe, a seguir, a definição dos conceitos segundo Pfleeger (2004).

- **Disponibilidade:** probabilidade de um sistema estar operando com sucesso, de acordo com as especificações, em um determinado momento (se o sistema está disponível no momento em que você precisa dele, se ele está sempre pronto a ser executado).
- **Manutenibilidade:** probabilidade de que uma manutenção possa ocorrer dentro de um prazo especificado.

Pfleeger (2004), para explicar os termos, faz uma analogia com um automóvel. Um automóvel deve ser confiável, permitindo ser utilizado por longos períodos em grandes viagens. Ele deve estar disponível, isto é, deve estar apto a ser utilizado quando o dono precisar. Além disso, o automóvel deve ter alta manutenibilidade, ou seja, não ter sido construído por uma fábrica que já tenha fechado, pois, assim, demoraria muito tempo para se encontrar a peça de reposição.

Além dos conceitos definidos por Pfleeger (2004), Pressman (2006) trata a **segurança de *software*** como uma atividade de garantia de Qualidade de *Software*, que foca na identificação e avaliação de riscos potenciais, que podem afetar o *software* negativamente e causar a falha de todo o sistema. Se esses riscos forem identificados logo no início do processo de *software*, os desenvolvedores podem especificar características do projeto que eliminem ou controlem esses riscos em potencial.

A tarefa de identificação de riscos de segurança inicia-se na fase de levantamento de requisitos. O *software* deve ser analisado no contexto de todo o sistema, para que os riscos sejam identificados. Depois de identificados e analisados os riscos, requisitos de segurança podem ser especificados para o *software*. A especificação pode conter uma lista de eventos indesejáveis e as respostas ideais do sistema a cada um destes eventos.

Apesar de serem conceitos relacionados, Pressman (2006) diferencia os conceitos de confiabilidade e segurança. Vamos conhecer essa diferenciação.

- **Confiabilidade:** usa a estatística para determinar a probabilidade de que uma falha de *software* ocorra, mas esta falha pode ou não resultar em um risco ou infortúnio.
- **Segurança:** examina os modos pelos quais as falhas resultam em condições que podem levar ao infortúnio, as falhas não são consideradas em um vazio, mas são avaliadas no contexto de todo o sistema e o ambiente.

10. OUTROS FATORES DE QUALIDADE

Além da confiabilidade, disponibilidade, manutenibilidade e segurança, há vários outros fatores que influenciam na qualidade do produto de *software*.

Observe, a seguir, alguns dos fatores que influenciam na qualidade do produto de *software* segundo Pressman (1995):

- **Corretitude:** medida que o programa satisfaz sua especificação e cumpre com os objetivos do cliente.
- **Confiabilidade:** precisão com que o programa executa a função pretendida.
- **Eficiência:** recursos exigidos para que o programa execute a função.
- **Integridade:** controle de acesso a pessoas não autorizadas.
- **Usabilidade:** esforço para usar o *software*.
- **Manutenibilidade:** esforço exigido para localizar e corrigir um erro.
- **Flexibilidade:** esforço para modificar o *software*.
- **Testabilidade:** esforço necessário para testes.
- **Portabilidade**: esforço para transferir de um ambiente para outro.
- **Reusabilidade**: medida que o programa pode ser reusado em outras aplicações.
- **Interoperabilidade:** esforço para acoplar um sistema a outro.

11. QUESTÕES AUTOAVALIATIVAS

Sugerimos que você procure responder, discutir e comentar as questões a seguir:

1) (POSCOMP, 2005) Considere as seguintes informações sobre o objetivo da atividade de validação de software:

   I – Verificar se o produto está sendo corretamente construído.

   II – Verificar se o produto está sendo corretamente avaliado.

   III – Verificar se o produto correto está sendo construído.

   Quais são as afirmações verdadeiras?

   a) Somente a afirmação (II).
   b) Somente a afirmação (III).
   c) Somente as afirmação (I) e (II).
   d) Somente as afirmação (II) e (III).
   e) Afirmação (I), (II) e (III).

2) Para se garantir a qualidade dos processos, vários passos devem ser tomados, entre eles:

   I – Gerenciar os requisitos, identificando quais são as principais necessidades do *software*, considerando tanto os requisitos funcionais quanto os não funcionais.

   II – Acompanhar o projeto de *software* para que se possa ter uma visão bem realista do progresso do projeto, sendo possível tomar ações eficazes quando o desempenho *software* se desviar de forma significativa dos planos do projeto.

   III – Gerenciar a configuração do *software* para estabelecer e manter a integridade dos produtos do projeto ao longo do ciclo de vida do *software* para dar maior segurança ao desenvolvedor e permitir maior controle de desenvolvimento.

   IV – Desenvolver um processo padrão para ser gerenciado e revisado, Identificar os pontos fortes e fracos do processo de desenvolvimento e planejar atividades de melhoramento.

   É correto o que se afirma em

   a) II e IV, apenas.
   b) I, II e III, apenas.
   c) II, III e IV, apenas.
   d) I e III, apenas.
   e) I, II, III e IV.

Fonte: Prova aplicada em 12/2009 para o concurso do(a) TRE-AM - 2010, realizado pelo órgão/instituição Tribunal Regional Eleitoral do Amazonas, área de atuação Jurídica, organizada pela banca FCC, para o cargo de Analista Judiciário - Tecnologia da Informação, nível superior, área de formação Tecnologia da Informação.

3) A estratégia de qualidade aplicada à arquitetura tradicional de *software* deve garantir para as etapas de Engenharia de Sistemas, Requisitos e Projetos, respectivamente, os testes de:

a) Sistema, validação e integração.
b) Sistema, integração e unidade.
c) Integração, validação e sistema.
d) Validação, integração e unidade.
e) Sistema, unidade e integração.

Fonte: Prova aplicada em 04/2009 para o concurso do(a) PGE-RJ - 2009, realizado pelo órgão/instituição Procuradoria Geral do Estado do Rio de Janeiro, área de atuação Jurídica, organizada pela banca FCC, para o cargo de Técnico Superior de Análise de Sistemas e Métodos, nível superior, área de formação Tecnologia da Informação .

Gabarito

Depois de responder às questões autoavaliativas, é importante que você confira o seu desempenho, a fim de que possa saber se é preciso retomar o estudo desta unidade. Assim, confira, a seguir, as respostas corretas para as questões autoavaliativas propostas anteriormente:

1) (b)

2) (e)

3) (a)

12. CONSIDERAÇÕES

Chegamos ao final do estudo da Unidade 6. Durante esta unidade, você teve a oportunidade de aprender os conceitos de qualidade e de garantia da qualidade de *software* e entender a importância da GQS na busca e manutenção da qualidade do produto de *software*.

Você pôde estudar quais são os objetivos e atividades de GQS e conscientizar-se da importância das definições de padrões e de como são realizadas as Revisões Técnicas Formais.

Além disso, compreendeu que um *software* com qualidade deve ser um *software* confiável, disponível, de fácil manutenção e seguro.

Na próxima unidade, você conhecerá os principais aspectos relacionados a testes de *software* e sua importância para a qualidade do produto de *software*.

13. REFERÊNCIAS BIBLIOGRÁFICAS

BUENO, C. F. S; CAMPELO, G. B. *Qualidade de Software*. Departamento de Informática. Universidade Federal de Pernambuco - Recife, PE, 2006. Disponível em: <http://www.cin.ufpe.br/~mrsj/Qualidade/Qualidade%20de%20Software.pdf>. Acesso em: 04 jan. 2010.

PAULA FILHO, W. P. *Engenharia de software*: fundamentos, métodos e padrões. 2. ed. Rio de janeiro: LTC, 2001.

PFLEEGER, S. L. *Engenharia de software teoria e prática*. 2. ed. São Paulo: Prentice Hall, 2004.

PRESSMAN, R. S. *Engenharia de software*. São Paulo: Makron Books, 1995.

_____. *Engenharia de software*. 6. ed. São Paulo: McGraw-Hill, 2006.

SOMMERVILE, I. *Engenharia de software*. 6. ed. São Paulo: PearsonAddison Wesley, 2005.

EAD

Teste de *Software*

7

1. OBJETIVOS

- Compreender a importância dos testes de *software*.
- Conhecer e discutir alguns conceitos básicos relacionados à atividade de teste.

2. CONTEÚDOS

- Estratégias de teste.
- Técnicas ou métodos de teste.
- *Workbench* de testes.

3. ORIENTAÇÕES PARA O ESTUDO DA UNIDADE

Antes de iniciar o estudo desta unidade, é importante que você leia as orientações a seguir:

1) Nos estudos das unidades anteriores, você compreendeu que qualidade é fator primordial para a produção de *software*.
2) É importante saber administrar o tempo para que o projeto tenha sucesso. O tempo é recurso crítico na produção de *software*. O desenvolvedor deve se acostumar a trabalhar com escassez deste recurso. É necessário administrar o tempo a fim de encaminhar bem o projeto. No entanto, não é raro nos depararmos com projetos atrasados. Quando isto ocorre, a atividade que é mais deixada de lado é o teste. Para o estudo desta unidade, você tem que ter em mente que, apesar do fator tempo ser muitas vezes uma restrição, é importante que o teste seja uma atividade considerada em um projeto de *software*. Com maior ou menor ênfase, dependendo das características da aplicação, é necessário considerar e planejar os testes, para que eles sejam eficazes na descoberta de erros. Lembre-se de que produtos com erros são produtos sem qualidade.

4. INTRODUÇÃO À UNIDADE

Na unidade anterior, você conheceu as atividades de Garantia de Qualidade de *Software* (GQS) e sua importância para inserir e manter qualidade no produto de *software*. Você aprendeu, também, que uma das atividades de GQS é o acompanhamento dos testes, visando à detecção de falhas e suas correções.

Nesta unidade, estudaremos alguns conceitos básicos relacionados às atividades de testes de *software*.

Observe que é fundamental, mesmo depois de todo cuidado durante o desenvolvimento do *software*, com revisões periódicas e inspeções para remoção de defeitos, realizar a etapa de teste para detectar as falhas que escaparam das revisões e para avaliar o grau de qualidade de um produto e seus componentes.

O fluxo de teste é extremamente importante para o desenvolvimento de *software*. Segundo Pressman (1996), o custo de correção de um problema na fase de manutenção pode chegar até a 60% do custo total do projeto. Em outras palavras, realizar testes é fundamental não apenas para atestar a qualidade do produto, mas também para diminuir custos totais do projeto.

Segundo Paula Filho (2005), na fase de teste deseja-se conseguir detectar a maior quantidade possível de defeitos que não foram percebidos pelas revisões, considerando os limites de custos e prazos.

5. ESTRATÉGIAS DE TESTES

Segundo Pressman (2007) e Paula Filho (2005), teste é considerado um conjunto de atividades que deve ser planejado antecipadamente e realizado de forma sistemática.

O planejamento e a realização das atividades de teste fazem parte do que definimos por **estratégia de teste de *software***.

Dessa forma, é fundamental que você compreenda que a Engenharia de Software "coloca ordem" em diversas atividades do desenvolvimento de um produto de *software*, esse *colocar ordem* implica planejar antecipadamente atividades e gerenciá-las. No caso de teste, isso não é diferente. Temos que montar uma estratégia antes de iniciarmos qualquer atividade relacionada ao teste de um *software*.

Segundo Pressman (1996), uma estratégia ou política de teste de *software* define técnicas de projeto de casos de teste e métodos de teste específico para **cada etapa do processo de engenharia de *software***.

Nenhum produto é testado apenas ao final de seu desenvolvimento, é por isso que para cada etapa do processo de engenharia de *software* é necessária uma estratégia específica de teste.

Por empirismo, sabemos que:

- a atividade inicia-se no nível de módulos e prossegue "para fora", na direção da integração de todo o sistema. Em sua vida acadêmica, sempre que você fez um programa e que continha módulos ou procedimentos, normalmente, você os testava separadamente e, depois, sua conexão. É a isso que esse item se refere.
- diferentes técnicas de teste são apropriadas a diferentes pontos de tempo. Em cada fase do projeto, testes devem ser aplicados e há técnicas específicas ou que podem trazer melhores resultados em pontos específicos do projeto.

Uma estratégia de teste deve:

a) acomodar testes de baixo nível e teste de alto nível;
b) oferecer orientação ao profissional;
c) oferecer um conjunto de marcos de referência ao administrador do projeto e de cada fase do sistema;
d) ser mensurável; há testes estatísticos que são utilizados para testar o desempenho e a confiabilidade do programa para checar como ele trabalha sob determinadas condições operacionais.

Vale ressaltar que a atividade oficial de teste não é a mesma realizada pela equipe de desenvolvimento.

A equipe de desenvolvimento realiza atividades de revisão e depuração e, após atestar que o projeto está adequado, é que a de teste entra em ação.

Para grandes projetos, uma equipe de teste é especialmente contratada. Ela é formada por um **grupo independente** de pessoas para atestar a qualidade da avaliação e verificação que está sendo realizada.

Geralmente, chega a um ponto em que os desenvolvedores e analistas "não enxergam mais erros" e isso não significa que eles não existem. É nesse momento que novos agentes são chamados,

esses agentes não participaram anteriormente do processo de desenho e implementação do projeto.

Segundo Sommerville (2007), um **grupo independente de teste** (*Independent Test Group* – ITG) apesar de ser, como o próprio nome diz, independente, faz parte da equipe de projeto de desenvolvimento de *software*, no sentido que está envolvido durante o processo de especificação e continua envolvido planejando e especificando estratégias e políticas de teste ao longo do projeto.

É importante mencionar que embora as atividades de teste e depuração sejam atividades diferentes e independentes, a depuração deve ser sempre acomodada em qualquer estratégia de teste.

Para que você compreenda as estratégias de teste, é fundamental que dois conceitos estejam claros: validação e verificação.

> Validação e Verificação (V&V) é o nome dado aos processos de verificação e análise que asseguram que o software cumpra com as especificações e atenda às necessidades dos clientes (SOMMERVILLE, 2005, p. 358).

- **Validação:** refere-se ao conjunto de atividades que garante que o *software* construído é rastreável às exigências do cliente.

Responde à pergunta: Estamos fazendo o produto **certo**?

- **Verificação:** refere-se ao conjunto de atividades que garante que o *software* está sendo construído da maneira correta.

Responde à pergunta: estamos fazendo o **certo** produto?

Ambas as definições estão diretamente relacionadas à garantia de qualidade de *software,* como podemos observar na Figura 1.

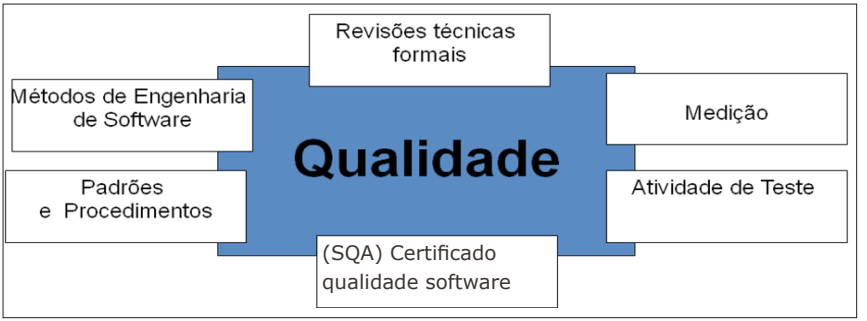


Figura 1 *Relação das atividades relacionadas à garantia de qualidade do processo de software.*

As principais referências sobre teste são os definidos pelo padrão IEEE (*Standards Collection – Software Engineering*), disponível em: <http://standards.ieee.org/software/>. Nesta unidade, utilizaremos esse padrão, referenciados pelos autores Sommerville (2005 e 2007) e Paula Filho (2005). Entretanto, não deixem de acessar o *site* citado anteriormente para conhecer um pouco mais sobre padrões IEEE relacionados aos assuntos de Engenharia de *Software*.

6. TÉCNICAS OU MÉTODOS DE TESTES

Para serem eficazes, é necessário que os testes sejam cuidadosamente planejados e estruturados (estratégia/política de teste).

Durante a realização dos testes, seus resultados devem ser cuidadosamente relatados (documentados) e inspecionados, pois nem sempre os resultados previstos são os obtidos, e nem sempre é tão óbvio quando se detecta um erro. Lembre-se de que os erros óbvios foram diagnosticados e adequados durante o processo de depuração realizado pelos programadores.

Segundo Paula Filho (2005, p. 184):

> [...] os testes são indicadores de qualidade do produto, mais do que meios de detecção e correção de erros. Quanto maior o número de

> defeitos detectados em um software, provavelmente maior também o número de defeitos não detectados. A ocorrência de um número anormal de defeitos em uma bateria de testes indica uma provável necessidade de redesenho dos itens testados.

Em se tratando de testes, é importante lembrar que:

- se erros graves forem encontrados com regularidade, então a qualidade e a confiabilidade do *software* são suspeitas;
- se erros facilmente corrigíveis forem encontrados, então a qualidade e a confiabilidade do *software* estão aceitáveis **ou** os testes são inadequados para revelar erros graves nesse *software* de maneira particular;
- se não forem encontrados erros, então a configuração de testes não foi suficientemente elaborada e os erros estão escondidos no *software*.

Com base nas explicações anteriores, podemos destacar que:

a) a atividade de teste é o processo de executar um programa com a intenção de descobrir um erro;
b) se a atividade de teste for conduzida com sucesso, ela descobrirá erros no *software;*
c) um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um erro ainda não descoberto;
d) um teste bem-sucedido é aquele que revela um erro ainda não descoberto.

Na Figura 2, podemos observar a estrutura das atividades de teste conforme discutido anteriormente.

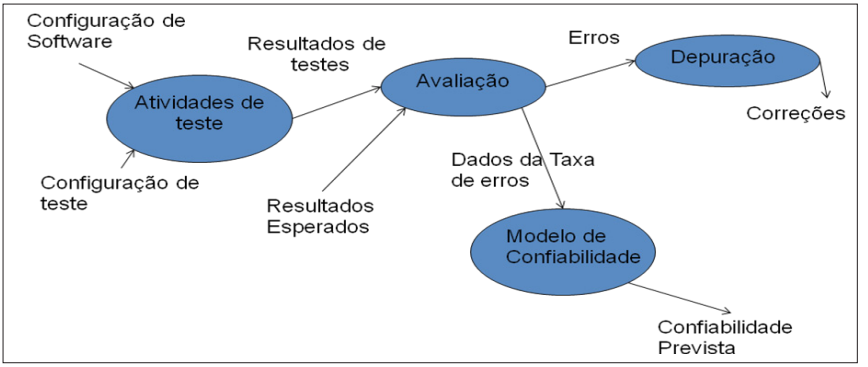


Figura 2 *Atividades de teste.*

As políticas de configuração de teste agregada à configuração e documento do *software* gerará uma política de atividades de teste, e, após executada, serão verificados os resultados obtidos com os esperados e realizada uma avaliação sobre esses dados. A partir desse momento, a taxa de erros entre esperados e obtidos será analisada por meio de uma política de confiabilidade (também criada anteriormente no momento da definição da estratégia de teste), e, caso a taxa de erro encontrada não seja adequada para o modelo de confiabilidade descrito, os erros serão novamente analisados, depurados e corrigidos.

Há, basicamente, duas maneiras de se construírem testes para detecção de defeitos.

- Teste da caixa preta.
- Teste da caixa branca.

A. Teste da caixa preta

Teste de caixa preta, também chamados de testes funcionais, é uma abordagem em que os testes são derivados da especificação de programas ou de componentes. Analisa o sistema como uma caixa preta, na qual seu comportamento será avaliado apenas por meio da análise das entradas e saídas relacionadas, ou seja, refere-se aos testes que são realizados nas interfaces do *software.*

Esse tipo de teste, geralmente, é usado para demonstrar que as funções dos *softwares* são operacionais, que a entrada é adequadamente aceita, a saída é corretamente produzida e a integridade das informações externas é mantida.

Essa técnica examina aspectos de um sistema **sem** se preocupar muito com a estrutura lógica do *software*, pois se concentra nos requisitos funcionais.

O teste de caixa preta procura descobrir erros nas seguintes categorias:

- funções incorretas ou ausentes;
- erros de interface;
- erros nas estruturas de dados ou no acesso a BD externos;
- erros de desempenho;
- erros de inicialização e término.

B. Teste da caixa branca ou de estrutura

O teste de caixa branca ou estrutura tem como objetivo determinar defeitos na estrutura interna do produto com técnicas que exercitem possíveis caminhos e erros de execução.

Nessa técnica, são testados caminhos lógicos por meio do *software*, fornecendo-se casos de testes que põem à prova conjuntos específicos de condições e/ou laços definidos no sistema.

Devemos ter em mente que testes exaustivos apresentam certos problemas logísticos, porque, mesmo para pequenos programas, o número de caminhos lógicos possíveis pode ser muito grande, mas, mesmo assim, essa técnica nunca deverá ser descartada, pois, como dissemos – cada caso de teste tem sua funcionalidade em momentos específicos definidos pela estratégia de teste escolhida.

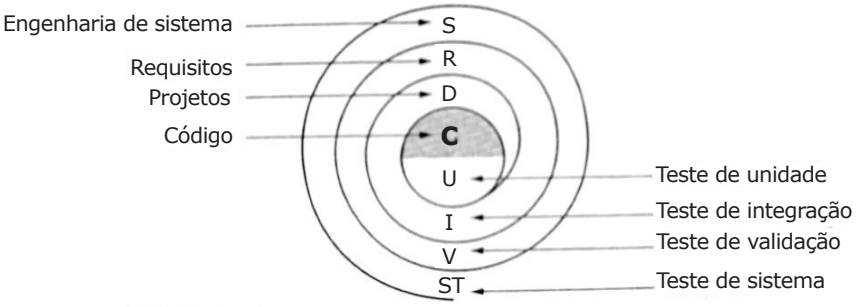
Estratégia de teste, também chamada de bateria de testes

As políticas de teste podem ser criadas seguindo as seguintes estratégias:

a) Teste de unidade.
b) Teste de integridade.
c) Teste de validação.
d) Teste de sistema.

No padrão nomenclatura IEEE, a bateria de teste de validação é chamada de teste de sistema, reservando-se o termo "teste de validação" àquele realizado pelo cliente como parte de um procedimento de aceitação do produto. Por efeitos didáticos, resolvemos definir, separadamente, o que é o teste de validação e o teste de sistemas. Entretanto, observe que os conteúdos desses dois grupos são semelhantes.

Observe nas Figuras 3 e 4 a estrutura e a relação das estratégias de testes com as fases do ciclo de vida.



Fonte: PRESSMAN, 1996, p. 840.
Figura 3 *Estrutura das estratégias de teste.*

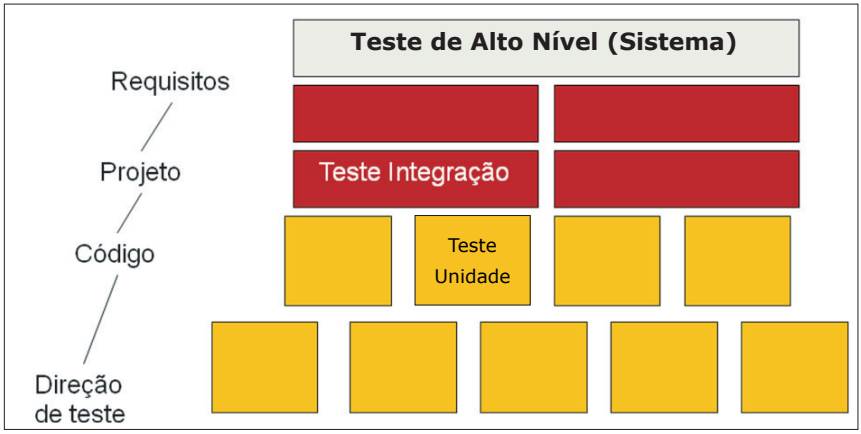


Figura 4 *Estratégia de teste*.

Observe a Figura 4. Na estratégia de teste, ao testarmos a unidade de um sistema, é analisado o código desenvolvido; no teste integração, o código já foi verificado, porém são analisados se diferentes módulos estão integrados de forma a atingir aos objetivos do projeto; no teste de alto nível são verificados se todos os requisitos funcionais e não funcionais foram satisfeitos.

A. Teste de unidade

O objetivo do teste de unidade é verificar os elementos que podem ser testados logicamente. Geralmente, concentra-se em cada unidade do *software*, de acordo com o que é implementado no código-fonte, cada módulo é testado individualmente, garantindo que funcione adequadamente. Para esse tipo de teste, são utilizadas as técnicas de caixa branca.

B. Teste de integridade

O objetivo do teste de integração é **verificar** as interfaces entre as partes de uma arquitetura do produto. É uma técnica sistemática para a construção da estrutura de programa, realizando-se, ao mesmo tempo, testes para descobrir erros associados a interfaces e para verificar erros de integração.

A partir dos módulos testados ao nível de unidade, é construída uma estrutura de programa que foi determinada pelo projeto.

Utilizam-se, especialmente, as técnicas de teste de caixa preta.

Os testes de integração podem ser compostos por:

- **Integração incremental:** o programa é construído e testado em pequenos segmentos, em que os erros são mais fáceis de serem isolados e corrigidos; as interfaces têm maior probabilidade de serem testadas completamente, e uma abordagem sistemática ao teste pode ser aplicada.
- **Integração não incremental:** abordagem *big-bang*. O programa completo é testado como um todo, mas prepare-se, o resultado é o caos. Quando os erros são encontrados, é difícil a correção, porque o isolamento das causas é complicado devido à vasta amplitude por estar sendo testado o programa completo.
- **Integração *top-down*:** os módulos são integrados movimentando-se de cima para baixo, por meio da hierarquia de controle. Inicia-se em um módulo de controle principal e, a partir dele, os outros são incorporados à estrutura de uma maneira *depth-first* (primeiro pela profundidade) ou *breadth-first* (primeiro pela largura).

C. Teste de validação

Ao término da atividade de teste de integração, o *software* está completamente montado como um pacote, erros de interface foram descobertos e corrigidos e uma série final de testes de *software*, ou seja, os testes de validação, podem ser inicializados.

O teste de validação tem como objetivo **validar** o produto, ou seja, verificar os requisitos estabelecidos como parte da análise de requisitos de *software*.

É considerado um teste de alto nível, pois verifica se todos os elementos combinam-se adequadamente e se a função/desempenho global do sistema é conseguida.

Nos testes de validação:

- a validação é bem-sucedida quando o *software* funciona de uma maneira razoavelmente esperada pelo cliente;
- a especificação de requisitos do *software* contém os critérios de validação que formam a base para uma abordagem ao teste de validação;
- a validação é realizada por meio de uma série de testes de caixa preta que demonstram a conformidade com os requisitos.

O plano de teste de validação garante que:

a) todos os requisitos **funcionais** sejam satisfeitos;
b) todos os requisitos de desempenho sejam conseguidos;
c) a documentação (projeto, programador e usuário) esteja correta;
d) requisitos não funcionais (portabilidade, compatibilidade, remoção de erros, manutenibilidade etc.) sejam cumpridos;

A revisão da configuração – verifica se os elementos de configuração de *software* foram adequadamente catalogados e desenvolvidos a ponto de apoiar a fase de manutenção.

Quando **um projeto de software é personalizado para vários clientes,** podemos realizar os testes de aceitação por meio das técnicas denominadas **Alfa e Beta**.

- **Teste Alfa**: o cliente testa o produto **nas instalações do desenvolvedor**. O *software* é utilizado em um ambiente controlado com o desenvolvedor "olhando sobre os ombros" do usuário e registrando erros e problemas.
- **Teste Beta**: o teste é realizado nas **instalações do cliente** pelo usuário final do *software*. O desenvolvedor não

está presente. O cliente registra os problemas que são encontrados e relata-os ao desenvolvedor em intervalos regulares.

Quando um produto de *software* é desenvolvido de forma personalizada (para um cliente), não se utiliza o teste alfa e nem o beta. Os testes de aceitação são realizados pelo usuário final para capacitá-lo e para validar todos os requisitos e variam de um teste informal a uma série de testes planejados que podem ser executados nas dependências do usuário ou do desenvolvedor, de acordo com o estipulado entre as partes.

D. Teste de sistema

Nessa categoria de teste, o *software* e outros elementos do sistema são testados como um todo. Envolve vários testes diferentes, com o propósito primordial de pôr completamente à prova o sistema baseado em computador.

- **Teste de recuperação:** um sistema deve ser capaz de tratar determinados erros e voltar à função global sem grandes problemas. Esse teste força o sistema a provocar esses erros para verificar como se comportaria.
- **Teste de regressão:** executam novos testes previamente executados para assegurar que alterações realizadas em partes do produto não afetem as outras partes já testadas. Alterações realizadas, especialmente na fase de manutenção, podem introduzir erros nas partes previamente testadas.
- **Teste de segurança:** tenta verificar se todos os mecanismos de proteção embutidos em um sistema o protegerão, de fato, de acessos indevidos.

A grande dúvida quando começamos a conhecer e a perceber a importância dos testes é: quando realizamos testes? Como saber se já testamos o suficiente?

Claro que uma resposta definitiva é complicada de oferecer, mas, segundo Sommerville (2007), há uma resposta pragmática que podemos considerar: é que jamais será completada uma atividade de teste; na verdade, o que acontece é que se transfere essa responsabilidade do projetista para a equipe de teste e desta para o cliente. Uma outra resposta aceitável: pode ser que você deverá interromper suas atividades de teste quando seu tempo provisionado para isso se esgotar ou acabar o dinheiro previsto para essa etapa.

Você conheceu algumas atividades de teste, e é fundamental que você reconheça a importância dessas atividades no projeto de software. Portanto, sugiro que você pesquise algumas técnicas que não foram abordadas neste material. Como, por exemplo:
- Particionamento de Equivalência ou Partição de Equivalência.
- Teste de caminho (básico).
- Análise do valor limite.
- Testes top-down e bottom-up.
- Teste de estresse.
- Testes orientados a objetos.
- Teste de classe de objetos.
- Teste de integração de objetos.

Procure relacionar essas técnicas com as estratégias abordadas. Boa pesquisa!

7. *WORKBENCH* DE TESTES

Como a fase de teste é muito trabalhosa, as primeiras ferramentas de software desenvolvidas foram as que apoiam essa atividade. A escolha da ferramenta é realizada com base nas características da aplicação a ser testada e na disponibilidade de ferramentas. Assim, as ferramentas a serem utilizadas devem ser inseridas no plano de testes como recursos técnicos a serem disponibilizados.

Para concluirmos nossos estudos sobre testes de *software,* é importante que você conheça um pouco sobre *workbench* de teste.

Primeiro, o que é *workbench*? Se pesquisarmos em algum dicionário de inglês, encontraremos que é o nome dado a mesas que servem como apoio ao desenvolvimento de atividades com madeira, metal e outros materiais.

Em nosso caso, essa expressão é utilizada para definir algumas ferramentas que foram desenvolvidas, que oferecem uma gama de recurso e reduz significativamente o custo do processo de teste, pois podem ser integradas às bancadas de testes existentes.

Segundo Sommerville (2005, p. 394), há algumas ferramentas que podem ser incluídas em *workbench* de teste, tais como:

> **Gerenciador de teste:** gerencia a execução dos testes de programa. O gerenciador de testes mantém o acompanhamento dos dados de teste, dos resultados esperados e dos recursos de programa testados.
>
> Gerador de dados de teste: gera dados de teste para o programa ser testado.
>
> **Oráculo:** gera previsões dos resultados esperados para o teste [...]. As diferenças entre saída e entrada são evidenciadas.

Comparador de arquivos: compara os resultados dos testes de programas com os resultados dos testes precedentes e relata a diferença entre eles. A diferença nesses resultados indica problemas em potencial com a nova versão do sistema.

> **Gerador de relatórios:** fornece uma definição de relatório e recursos de geração para os resultados de testes.
>
> **Analisador dinâmico:** adiciona o código a um programa para contar o número de vezes que cada declaração foi executada. Depois que os testes foram executados, é gerado um perfil de execução que mostra a freqüência com que cada declaração de programa foi executada.
>
> **Simulador:** diferentes tipos de simuladores podem ser fornecidos. Os simuladores-alvos simulam a máquina em que o programa deverá ser executado. Os simuladores de interface com o usuário são programas orientados por *scripts*, que simulam múltiplas interações com o usuário. A utilização de E/S significa que o *timing* de seqüências de transações pode ser repetido.

Devido ao tempo de se desenvolver *workbenches* de testes completos, é necessário esclarecer que esse recurso só é utilizado

quando sistemas são complexos e grandes. Independentemente de sua utilização, é importante conhecermos a existência desse recurso.

8. QUESTÕES AUTOAVALIATIVAS

Teste seus conhecimentos, realizando as questões a seguir:

1) No contexto da estratégia para teste de um projeto, os estágios de teste desempenham um papel importante. O teste que é aplicado a componentes do modelo de implementação para verificar se os fluxos de controle e de dados estão cobertos e funcionando conforme o esperado, é o teste:

   a) Do desenvolvedor.
   b) Independente
   c) De integração
   d) De sistema
   e) Unitário

Fonte: Prova aplicada em 03/2010 para o concurso do TRT - 20ª REGIÃO (SE) - 2009, realizado pelo órgão/instituição Tribunal Regional do Trabalho da 20ª Região - Sergipe, área de atuação Jurídica, organizada pela banca FCC, para o cargo de Analista Administrativo - Informática, nível superior, área de formação Tecnologia da Informação.

2) (POSCOMP, 2003) Em relação ao teste de software, qual das afirmações a seguir é INCORRETA:

   a) Os dados compilados quando a atividade de teste é levada a efeito proporcionam uma boa indicação da confiabilidade do software e alguma indicação da qualidade do software como um todo.
   b) Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um erro ainda não descoberto.
   c) Um teste bem sucedido é aquele que revela um erro ainda não descoberto.
   d) A atividade de teste é o processo de executar um programa com a intenção de demonstrar a ausência de erros.
   e) O processo de depuração é a parte mais imprevisível do processo de teste pois um erro pode demorar uma hora, um dia ou um mês para ser diagnosticado e corrigido.

3) (ENADE, 2005) Julgue os seguintes itens referentes a teste de software.

   I – A técnica de teste funcional, que estabelece os requisitos de teste com base em determinada implementação, permite verificar se são atendidos os detalhes do código e solicita a execução de partes ou de componentes elementares do programa; a técnica de teste estrutural aborda o software de um ponto de vista macroscópico e estabelece os requisitos de teste, com base em determinada implementação.

   II – Na fase de teste de unidade, o objetivo é explorar-se a menor unidade de projeto, procurando-se identificar erros de lógica e de implementação de cada módulo; na fase de teste de integração, o objetivo é descobrir erros as-

sociados às interfaces entre os módulos quando esses são integrados, para se construir a estrutura do software, estabelecida na fase de projeto.

III – Critérios com base na complexidade, em fluxo de controle e em fluxo de dados, são utilizados pela técnica estrutural de teste.

Assinale a opção correta.

a) Apenas um item está certo.
b) Apenas os itens I e II estão certos.
c) Apenas os itens I e III estão certos.
d) Apenas os itens II e III estão certos.
e) Todos os itens estão certos.

Gabarito

Depois de responder às questões autoavaliativas, é importante que você confira o seu desempenho, a fim de que possa saber se é preciso retomar o estudo desta unidade. Assim, confira, a seguir, as respostas corretas para as questões autoavaliativas propostas anteriormente:

1) (e)

2) (d)

3) (d)

9. CONSIDERAÇÕES

Como podemos observar, planejamento de testes e manutenção fazem parte dos conceitos fundamentais da idealização de um sistema.

Realizar testes está muito além de apenas compilar. Está relacionado ao perfeito funcionamento, a longo prazo, de um sistema.

Entretanto, embora projetemos sistemas para terem uma vida relativamente longa, manutenções sempre serão necessárias. Muitas serão solicitadas pelos usuários e temos que avaliar a pertinência desses pedidos, outras serão oferecidas pelo próprio fabricante do produto com o objetivo de atualizar e melhorar a eficiência do produto de *software*.

Não importa como essas mudanças ou casos de testes são solicitados, o importante é que tanto um quanto o outro fazem parte de atividades bem planejadas e, especialmente, quando executadas, devem ser adequadamente documentadas.

Na unidade seguinte, você aprenderá sobre manutenção de *software*.

10. REFERÊNCIAS BIBLIOGRÁFICAS

IEEE – Standards Collection – Software Engineering (2007). Disponível em: <http:// standards.ieee.org/ software/>. Acesso em: 25 jun. 2007.

PAULA FILHO. *Engenharia de software*: Fundamentos, métodos e padrões. São Paulo: LTC, 2003.

PRESSMAN, R. S. *Engenharia de software*. São Paulo: Makron Books, 2006.

SOMMERVILE, I. *Engenharia de software.* 6. ed. São Paulo: Pearson, 2005.

_____. *Engenharia de software*. 8. ed. São Paulo: Pearson, 2007.

EAD

Manutenção de *Software*

8

1. OBJETIVOS

- Conhecer e discutir alguns conceitos básicos relacionados à atividade de manutenção.
- Compreender as características e as dificuldades da manutenção nos sistemas.
- Identificar os custos e os esforços gastos na manutenção.
- Compreender conceitos que estão intimamente relacionados à manutenibilidade de *software*, tais como: à documentação, ao projeto de *software*, à modularidade, à coesão e ao acoplamento.

2. CONTEÚDOS

- Mudanças no *software*.
- Manutenibilidade.
- Características de manutenção.

- Manutenção estruturada e não estruturada.
- Custos e esforços da manutenção.
- Problemas associados à manutenção.
- Documentação de *software*.
- Projeto de *software.*

3. ORIENTAÇÕES PARA O ESTUDO DA UNIDADE

Antes de iniciar o estudo desta unidade, é importante que você leia as orientações a seguir:

1) Até agora você pôde aprender que o processo de desenvolvimento de *software* é composto de várias fases, nas quais é necessário incorporar sempre ao *software* os requisitos definidos pelo cliente. Agora chegamos à última etapa do ciclo de vida do sistema. Etapa esta que pode ser mais longa do que todo o seu processo de desenvolvimento.
2) É fundamental que no estudo desta unidade, você tenha em mente que o trabalho do desenvolvedor não termina quando o *software* é implantado. Pelo contrário, dependendo do projeto, o maior esforço é despendido na sua manutenção.
3) Sempre que surgirem dúvidas em relação aos conceitos, consulte o *Glossário de Conceitos*, compartilhe suas dúvidas com os colegas e, sempre que necessário, peça auxílio a seu tutor.
4) Adquira o hábito da leitura e da pesquisa. Leia livros, revistas e periódicos, pesquise em *sites* e, caso encontre alguma novidade relacionada ao conteúdo estudado, compartilhe com seus colegas de curso.

4. INTRODUÇÃO À UNIDADE

Na unidade anterior, você conheceu as particularidades dos testes de *software* e verificou a importância dos testes para a qualidade do produto de *software*.

Nesta unidade, estudaremos alguns conceitos básicos relacionados às atividades de manutenção de um *software*.

5. MUDANÇAS NO *SOFTWARE*

Vamos iniciar esta unidade lembrando-nos da chamada Crise do *Software*. O que mesmo a originou, por volta da década de 1960? Você se lembra? A manutenção de *software* teve alguma relação com o surgimento da Crise do *Software*? Se você respondeu que sim, parabéns, você acertou.

A palavra manutenção é bastante associada ao início da Crise do *Software*, em que a dificuldade de manutenção em códigos "alienígenas" fazia com que os desenvolvedores perdessem muito tempo corrigindo e modificando *softwares* e não conseguissem suprir a demanda por novos sistemas.

Ah! Você deve estar se perguntando: o que é código alienígena? O termo alienígena é usado para indicar *softwares* feitos sem qualquer padronização, disciplina ou documentação. O desenvolvedor demorava tanto tempo para entender o significado das instruções (quando o código era feito por outra pessoa) que dava a impressão de que o código tinha vindo de outro planeja. Entendeu?

A manutenção de *software* é o processo geral de modificações de um sistema depois que ele começa a ser utilizado. Ela pode ser simples – corrigir erros de código, de forma complexa com a finalidade de acomodar novos requisitos.

Para O'Brien (2004), manutenção de sistemas é uma forma de monitoração, avaliação e modificação de um determinado sistema de informação para correções de determinados erros ou implementação de melhorias.

O desenvolvimento do sistema está completo quando ele pode ser considerado operacional, isto é, quando o sistema está

sendo utilizado pelos usuários em um ambiente real de produção. Para Pfleeger (2004), qualquer trabalho efetuado para modificar o sistema, depois que ele estiver em operação, é considerado manutenção.

Geralmente, a manutenção não está relacionada a grandes alterações do *software*, elas estão mais relacionadas à acomodação de novos componentes de sistemas.

A primeira lei da Engenharia de *Software* diz que: "Não importa onde se esteja no ciclo de vida do sistema, o sistema se modificará, e o desejo de mudá-lo persistirá ao longo de todo o ciclo", em outras palavras, a indústria de *software* é composta de inúmeras variáveis e controvérsias. Mas todos concordam que a mudança é uma constante na Engenharia de *Software*.

Você deve estar lembrado dos mitos do *software*, em que você aprendeu que estes mitos desencadeiam desinformação e confusão. Entre os mitos relacionados com a manutenção de *software*, Pressman (1996, p. 29) descreve o principal deles, o de se acreditar que "assim que o programa (*software*) estiver sido criado e colocado em operação, o trabalho do desenvolvedor está acabado". Esse mito anula ou desconsidera manutenção de *software* como parte do seu ciclo de vida. Segundo Pressman (1996, p. 29), os dados na indústria do *software* indicam que entre 50% e 70% de todo o esforço gasto no desenvolvimento de um *software* serão despendidos depois que ele for entregue pela primeira vez ao cliente.

Portanto, é um grande erro acreditar que, depois que *software* for criado e entregue ao cliente, o trabalho do desenvolvedor de *software* acabou. Dessa forma, podemos dizer que o trabalho em relação ao *software* não termina quando o *software* está pronto, pelo contrário, o trabalho continua, pois, além de necessitar de correções de eventuais erros que possam surgir, novas tecnologias também surgirão e o cliente, certamente, exigirá que elas sejam aplicadas ao *software*. Assim, são necessárias modificações para

que o *software* esteja sempre atualizado e com suas funções de acordo com as exigências dos clientes ou usuários. Pode-se dizer, então, que a manutenção de um *software* é um "mal necessário e inevitável", sendo de extrema importância para as organizações que desenvolvem ou utilizam os sistemas de informação.

Pressman (1996, p. 29) cita, ainda, o mito de que "a única coisa a ser entregue em um projeto bem sucedido é o programa funcionando". Na realidade, porém, o *software* funcionar é somente uma parte de uma configuração que inclui, também, vários outros elementos gerados durante o processo de desenvolvimento do *software*, como, por exemplo, o planejamento, a Especificação dos Requisitos do Sistema e o *Software*, manuais de utilização do sistema, entre outros. A documentação gerada é o alicerce de um desenvolvimento bem-sucedido, que fornece um guia para a tarefa de manutenção do *software*.

Assim que o sistema começa a ser utilizado, ele passa por um estágio de contínuas mudanças, mesmo que tenha sido projetado a partir das melhores técnicas existentes de projeto e codificação. Com a velocidade com que surgem novas tecnologias, é inevitável que o sistema se torne obsoleto com o passar do tempo.

Para Pressman (1996), a manutenção ocorre porque não é razoável presumir que todo o teste de *software* realizado descobrirá todos os erros possíveis em um grande sistema de *software*.

Se a atividade de teste for conduzida com sucesso, ela descobrirá erros no *software* e, consequentemente, diminuirá a necessidade de manutenção corretiva. No entanto, muitas vezes, devido à pressão por prazos de entrega e ao custo envolvido em atividade de teste, ela não é executada com a dedicação necessária, aumentando, assim, a necessidade de intervenções após a entrega do *software* ao cliente.

Essa fase pode ser considerada a mais "problemática" do ciclo de vida de um *software*, pois os sistemas devem continuar rodando e, nesse momento, as alterações são inevitáveis.

Embora na prática não haja muita diferença entre os tipos de manutenção, podemos dividi-las nas seguintes categorias:

- Corretivas: manutenção para corrigir defeitos no *software*.
- Adaptativas: adaptar o *software* a um ambiente operacional diferente.
- Perfectivas: mudanças para fazer acréscimos a funcionalidades do sistema ou modificá-lo. Geralmente, pode atender pedidos do usuário para:
  a) modificar funções existentes;
  b) incluir novas funções;
  c) efetuar melhoras em geral;
  d) melhorar a manutenibilidade ou confiabilidade futura;
  e) fornecer uma base melhor para futuros melhoramento.

6. MANUTENIBILIDADE

Um dos critérios da qualidade de produto de *software* é a sua manutenibilidade.

Segundo Pressman (1996, p. 884), manutenibilidade pode ser definida, qualitativamente, como a facilidade com que um *software* pode ser entendido, corrigido, adaptado e/ou expandido. A manutenibilidade é meta primordial e, também, orienta o processo de engenharia de um *software*.

A manutenibilidade, segundo a norma ISO/IEC 9126-1 (1999), pode ser melhor entendida por meio da análise de suas sub-características:

1) **Analisabilidade**: capacidade de permitir o diagnóstico de deficiências ou causas de falhas no *software*, ou a identificação de partes a serem modificadas.
2) **Modificabilidade**: capacidade de permitir que uma modificação especificada seja implementada.

3) **Estabilidade**: capacidade de evitar efeitos inesperados decorrentes de modificações no *software*.
4) **Testabilidade**: capacidade de permitir que o *software*, quando modificado, seja validado.
5) **Conformidade relacionada à manutenibilidade**: capacidade de estar de acordo com normas ou convenções relacionadas à manutenibilidade.

As organizações empresariais, de modo geral, estão, constantemente, submetidas a mudanças que requerem balanço e equilíbrio. Essas organizações atuais mantêm uma dinâmica interação com o ambiente, sejam fornecedores, clientes, concorrentes, órgãos governamentais, entidades sindicais, enfim, vários outros agentes externos. Elas influem sobre o meio-ambiente e, também, são diretamente influenciadas. A organização, para manter-se competitiva, deve estar atenta às mudanças ambientais e tentar absorver tais mudanças sem afetar suas metas e objetivos primários.

Conseguir acompanhar o ritmo das mudanças é um critério indispensável para garantir competitividade. As organizações conseguem e mantêm vantagem competitiva por meio da melhoria, da inovação e do aperfeiçoamento (PORTER, 1993).

Obviamente, os sistemas de informação computacionais que apoiam a operacionalidade, a gerência e a estratégia dessas organizações, devem se adaptar, também, a essas mudanças. Assim, a manutenção também realiza modificações por motivo de mudanças que aconteçam na organização ou no ambiente de negócios. Um exemplo disso são reestruturações que ocorrem na legislação tributária, que exigem grandes mudanças nos sistemas de informação que as empresas possuem.

Enfim, os sistemas computacionais que apoiam estas organizações devem ter a flexibilidade necessária para acompanhar as mudanças. Portanto, quanto melhor a manutenibilidade dos sistemas, melhor para a expansão e produtividade na indústria de *software*.

7. CARACTERÍSTICAS DE MANUTENÇÃO

Segundo Sommerville (2005, p. 519):

> Os custos de manutenção, com uma proporção dos custos de desenvolvimento variam de um domínio de aplicação para outro. Para sistemas de aplicação de negócio (...) os custos de manutenção eram amplamente comparáveis aos custos de desenvolvimento de sistemas. Para os sistemas embutidos de tempo real, os custos de manutenção podem ser até quatro vezes maiores do que os custos de desenvolvimento. Os elevados requisitos de confiabilidade e desempenho desses sistemas podem exigir que os módulos sejam estreitamente vinculados e, como conseqüência, difíceis de serem modificados.

Investir em esforço ao projetar e implementar um sistema é considerada a melhor opção para diminuir os custos de manutenção, pois é muito mais caro inserir novas mudanças no sistema (adicionar funcionalidades) depois da entrega do que durante o desenvolvimento.

> Boas técnicas de engenharia de software, como a especificação precisa, o uso do desenvolvimento orientado a objetos e do gerenciamento de configurações, contribuem para a redução do custo de manutenção (SOMMERVILLE, 2005, p. 519).

Há alguns problemas clássicos que podem interferir em uma boa manutenção após a entrega do sistema para o cliente, como por exemplo:

1) é impossível ou difícil traçar a evolução do *software* devido às várias versões existentes;
2) as alterações realizadas não são adequadamente documentadas;
3) é difícil rastrear o processo pelo qual o sistema foi criado;
4) é difícil entender programas de outras pessoas, especialmente quando estas não estão por perto para explicar;
5) documentação não existe, é incompreensível ou está desatualizada;
6) a maioria dos *softwares* não foi projetada para apoiar alterações;
7) a manutenção não é vista como trabalho glamouroso e sim como um retrabalho.

A manutenção varia de acordo com o tipo do *software*, dos processos de desenvolvimento e do pessoal envolvido no processo. Algumas vezes, surge, até mesmo, de conversas informais, o que pode levar a um processo de **manutenção informal** (Figura 1); entretanto, o mais adequado e utilizado em grandes projetos é a **manutenção formal**, com documentos específicos descrevendo etapas e características de cada fase ou etapa.

Basicamente, podemos dividir o processo de manutenção em:

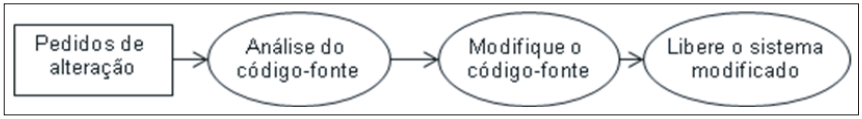
a) O usuário, a gerência ou clientes solicitam a mudança.

b) Se as mudanças forem aceitas, um novo *release* do sistema será planejado. Durante esse planejamento, todas as solicitações e implicações são analisadas e o procedimento de alteração idealizado. Nessa etapa, os novos requisitos do sistema que foram propostos são analisados e validados.

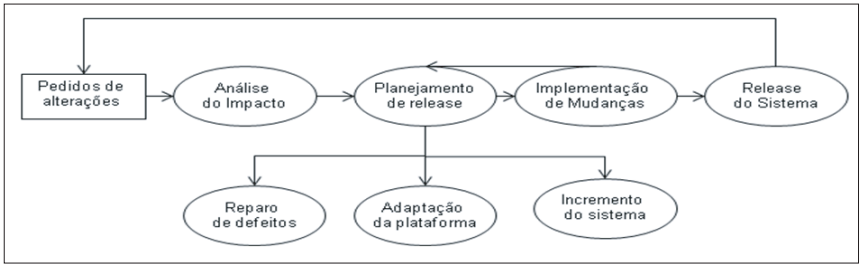
c) Mudanças são implementadas e validadas em uma nova versão do sistema.

d) Nova versão do sistema é liberada.

Essas etapas podem ser compreendidas melhor nas Figuras 1 e 2.

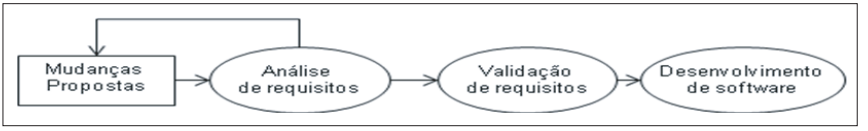


Fonte: SOMMERVILLE, 2005, p. 522.
Figura 1 *Processo de reparos de emergência.*



Fonte: SOMMERVILLE, 2005, p. 522.
Figura 2 *Visão geral do processo de mudança.*

Claro que há mudanças que implicam bom funcionamento imediato do sistema e essas mudanças são realizadas com menos burocracia, mas o processo de análise e validação das alterações é sempre fundamental para que uma versão adaptada do sistema seja liberada para o cliente e não cause maiores danos. Entretanto, para assegurar que os requisitos, o projeto de *software* e o código se tornem inconsistentes a essas mudanças, assim como as categorizadas informais, devem sempre ser documentadas e a documentação geral do projeto de *software* atualizado. Figura 4.



Fonte: SOMMERVILLE, 2005, p. 522.
Figura 3 *Implementação de mudanças.*

Gerentes odeiam surpresas, portanto, uma boa equipe de desenvolvimento sempre procurará prever que mudanças no sistema possivelmente serão solicitadas, quais partes podem causar maior dificuldade para alterá-la.

Segundo Sommerville (2005), considerando as previsões de mudanças, podemos dividi-las em três e a cada uma delas associar determinadas perguntas:

1) **Previsão de facilidade de manutenção:** que partes do sistema serão mais dispendiosas para se manter?
2) **Previsão de mudanças no sistema**: que partes do sistema provavelmente serão mais afetadas pelos pedidos de mudanças? Quantos pedidos de mudanças podem ser esperados?
3) **Previsão de custo de manutenção:** quais serão os custos de manutenção durante o tempo de vida útil do sistema? Quais serão os custos de manutenção do sistema no próximo ano?

Prever as respostas dessas perguntas garantirá um maior controle das consequências que podem ser geradas, no sistema e na documentação, a partir da solicitação de mudanças.

8. MANUTENÇÃO ESTRUTURADA E NÃO ESTRUTURADA

Segundo Pressman (1996), até recentemente, a manutenção era uma fase negligenciada da engenharia de *software*, e a literatura sobre manutenção ainda é um pouco precária quando comparada a outras fases do desenvolvimento.

Para uma melhor compreensão das características da manutenção pode-se considerar: as atividades que são exigidas na manutenção; a falta ou impacto de uma abordagem de engenharia sobre a eficiência e a eficácia dessas atividades, e o custo e problemas que surgem frequentemente quando a manutenção é colocada em prática.

Muitas vezes, certas requisições de manutenção estão relacionadas a problemas que devem ser resolvidos imediatamente. O caminho a ser seguido em situações de emergência é o mais simples possível. Inicia pela análise do código fonte, segue com a modificação do código e finaliza com a liberação do sistema já reparado. Contudo, um grande problema esconde-se por trás da rapidez com que a manutenção foi executada, visto que a documentação raramente é atualizada.

Há, ainda, situações em que o único elemento que se tem disponível sobre um determinado *software* que precisa ser modificado é seu código fonte. Frequentemente, a documentação interna do código é ruim ou até inexistente, o que faz da manutenção um processo de difícil avaliação do código.

Nos casos em que a manutenção é feita sem seguir uma abordagem sistemática para implementação das modificações, é considerada **manutenção não estruturada**. Isso acelera o processo de envelhecimento do sistema e futuras alterações ficam cada vez mais difíceis de serem desenvolvidas.

A manutenção é, portanto, considerada **estruturada** quando o *software* a ser modificado possui uma configuração completa, isto é, o código fonte e seu processo de desenvolvimento estão docu-

mentados. A manutenção estruturada inicia-se com a análise dessa documentação e, em seguida, são avaliados os impactos das mudanças ou correções solicitadas antes de serem feitas. Depois, são implementadas modificações no código, utilizando uma abordagem planejada e a documentação do *software* é atualizada. Antes de ser liberado, são feitos testes completos para detecção de erros que possam ter sido inseridos durante a modificação no *software*.

Portanto, a manutenção estruturada caracteriza-se por uma sequência objetiva de etapas para modificar o *software*. No entanto, manutenção estruturada não significa manutenção livre de problemas, apenas o esforço despendido diminui e a qualidade global de uma mudança (ou correção) é aumentada.

O processo de manutenção não deve ser visto como um processo separado e sim como um ciclo, ou uma iteração, do processo de desenvolvimento. Essa é a manutenção estruturada que possui todas as etapas convencionais do desenvolvimento, desde a formulação e validação de novos requerimentos até a implementação dos componentes e testes do sistema.

9. CUSTOS E ESFORÇO DESPENDIDO NA MANUTENÇÃO

Os custo com manutenção de um sistema representam grande parte do orçamento das organizações. De acordo com Pfleeger (2004), os problemas enfrentados pela manutenção contribuem para os seus altos custos.

Para Pressman (1996), na década de 1970, o índice do orçamento despendido em manutenção era em torno de 35% a 40% em uma organização. Conforme Pfleeger (2004), na década de 1980, esse índice chegou a 60% do custo total do ciclo de vida de um sistema, ou seja, desde o desenvolvimento, incluindo manutenção, até o eventual descarte ou substituição.

Considerando o contexto apresentado anteriormente, Pfleeger (2004, p. 391) afirma que “as estimativas atuais sugerem que,

no ano 2000, os custos de manutenção aumentaram 80% do custo referente ao tempo de vida de um sistema".

Entretanto, uma forma eficaz de diminuir custos de manutenção, segundo Sommerville (2005), é investir em esforço para projetar e implementar um sistema. E, para isso, há algumas técnicas da engenharia de *software* que podem contribuir para a redução de custos, que são: a especificação precisa, a utilização do desenvolvimento da orientação a objeto e o gerenciamento e a configuração.

Os custos também diminuem a partir do momento em que se empregam mais esforços no desenvolvimento de um sistema visando facilitar a sua manutenção, a qual se torna mais onerosa quando se trata de um sistema pronto, sendo necessário acrescentar novas funcionalidades.

Alguns fatores influenciam no esforço necessário à manutenção, segundo Pfleeger (2004, p. 391), são eles:

1) **Tipo de Aplicação:** um sistema altamente sincronizado e, em tempo real, possuem maior dificuldade para serem modificados do que sistemas que não necessitam de sincronia para seu bom funcionamento. Deve-se tomar cuidado para que uma mudança não afete a sincronia de outros componentes.
2) **Novidade do sistema:** ao se realizar novas aplicações em um sistema ou novos meios de utilizar funções comuns, a equipe de manutenção não pode confiar na experiência e no conhecimento para corrigir os erros ou defeitos. É necessário muito tempo para conhecer o projeto e localizar o problema. Em alguns casos, é até mesmo necessário realizar novos testes.
3) **Rotatividade e disponibilidade de pessoal da manutenção:** a manutenção é prejudicada quando há uma rotina de substituições dos membros da equipe por outros grupos, pois é necessário um tempo considerável para compreender um sistema antes de modificá-lo.

4) **Duração da vida útil do sistema:** sistemas projetados para durar um longo período de tempo necessitarão de mais manutenção, do que um projeto que possui vida útil mais curta, por isso correções e atualizações na documentação são importantes.
5) **Dependência de um ambiente que se modifica:** mudanças no ambiente requerem, consequentemente, mudanças no *software*.
6) **Características de um *hardware*:** a utilização de componentes e suporte técnico de *hardware* não confiáveis dificultam a descoberta da fonte de um problema.
7) **Qualidade do projeto:** caso o sistema não seja formado por componentes coesos e independentes, mudanças em um dos componentes podem acarretar problemas não previstos nos demais componentes.
8) **Qualidade do código:** se o código não possuir estruturação ou seguir princípios da arquitetura, pode ser mais difícil encontrar defeitos. Além disso, a própria linguagem de programação utilizada influencia na facilidade de manutenção.
9) **Qualidade da documentação:** quando não se documenta um código ou projeto, ou quando a documentação estiver errada ou desatualizada, a busca para soluções de um problema torna-se difícil.
10) **Qualidade dos testes**: a realização de testes com dados incompletos ou que não tem previsão de uma possível alteração, podem dificultar, também, o trabalho de manutenção.

10. PROBLEMAS ASSOCIADOS À MANUTENÇÃO

De acordo com Pressman (1996), quase todos os problemas associados à manutenção relacionam-se com deficiências na forma em que o *software* foi desenvolvido e planejado. O que significa que o cliente gasta com a compra do *software* e gasta muito mais após a sua obtenção.

Outros problemas como o mau uso e a falta de disciplina e controle das atividades existentes da engenharia de *software* também estão relacionados ao desenvolvimento do *software*.

Segundo Pressman (1996), muitos são os problemas que estão relacionados com a manutenção do *software*, veja quais são a seguir:

1) Rastrear a evolução dos *softwares*, muitas vezes, é difícil ou quase impossível por meio de lançamentos ou versões, pois as mesmas não são devidamente documentadas.
2) Rastrear o processo pelo qual o *software* foi criado, constantemente, torna-se difícil ou impossível.
3) Entender o programa desenvolvido por alguém desconhecido é muito difícil, pois o desenvolvedor não está por perto para explicar e a dificuldade aumenta à medida que os elementos de uma configuração diminuem.
4) A falta de existência de uma documentação ou a falta de compreensão dela também é um problema. Para o reconhecimento de um *software* a documentação deve ser o primeiro passo, embora ela deva ser compreensível e consistente.
5) Um grande problema é que os *softwares* não são projetados para receber mudanças. A menos que se utilize um método de projeto que acomode mudanças, como classes de objetos ou independência funcional, a manutenção no *software* torna-se difícil e, provavelmente, ocorrerão erros.
6) A manutenção não é considerada um trabalho de expressivo reconhecimento, essa percepção vem do grande número de frustrações sofridas durante o trabalho realizado por ela.

Os problemas descritos na manutenção do *software* podem ser associados ao grande número de programas existentes na atualidade e que foram desenvolvidos sem considerar a engenharia de *software*. Entretanto, a engenharia de *software* possui soluções, no mínimo, parciais para os problemas associados à manutenção.

A manutenibilidade não pode ser medida diretamente, para tanto são utilizadas métricas indiretas. Uma métrica simples, orientada ao tempo, é o tempo médio para a mudança ou MTTC

(*mean-time-to-change*) (PRESSMAN, 1996). Essa métrica trata-se do tempo gasto em analisar o pedido de mudança, projetar e implementar a mudança, testá-la e distribuir a mudança a todos os usuários. Os *softwares* com boa manutenibilidade terão MTTC menor do que os que forem difíceis de serem mantidos.

Antes de continuar seus estudos, pare um pouquinho e, com base em todo o que você aprendeu até agora sobre Engenharia de *Software*, reflita: qual a relação entre documentação e manutenção de *software*?

11. DOCUMENTAÇÃO DO *SOFTWARE*

Segundo Souza (2005), um documento é qualquer artefato externo ao código fonte que tem como objetivo transmitir informação de uma maneira persistente. O termo documentação inclui documentos e comentários de código fonte.

Pfleeger (2004) define documentação como um conjunto de descrições que explicam a um leitor o que os programas fazem e como o fazem. A documentação interna é o material escrito direto no código e toda a documentação restante é considerada documentação externa.

A documentação de *software* tem a finalidade de comunicar a informação sobre o *software* ao qual ele pertence, por exemplo, especificações, modelos, documentos, código fonte e entre outros.

A falta da documentação de *software* foi a principal causa da Crise do *Software*. Assim, podemos dizer que a documentação tem grande importância no entendimento do programa. Mais de 50% do trabalho de evolução do *software* é dedicado ao entendimento do programa e a tarefa mais difícil da manutenção de *software* é o entendimento da estrutura do sistema (SOUZA, 2005).

A manutenção não pode ser realizada enquanto o código fonte não for compreendido e, dessa forma, os documentos gerados em fases anteriores à codificação, como especificação de requisitos e documentação de projeto constituem a base para o entendimento. Entretanto, a estrutura e a documentação do código fonte também são relevantes na manutenibilidade do sistema.

A falta de documentação pode representar um risco para os projetos de *software* em especial na fase de manutenção. É impossível garantir que o conhecimento dos desenvolvedores experientes, permanecerá na equipe quando esta for dissolvida e a falta de documentação ou a documentação incorreta afetará a compreensão do sistema, sobretudo na atividade de análise de código.

Em pesquisa realizada por Souza (2005), junto a profissionais experientes em manutenção, foi constatado que os artefatos de *software* que mais auxiliam no entendimento do sistema na fase de manutenção, são: "Código Fonte" com 93,8% de indicação, "Comentários do Código Fonte" (77,3%), "Modelo Lógico de Dados" (71,9%), "Descrição dos Requisitos" (60,3%), "Modelo de Dados" (59,4%) e "Dicionário de Dados" (47,6%). A importância do código fonte, juntamente com os comentários nele existentes, reforça a importância da adoção de padrões de codificação e de organização interna do código.

No que diz respeito à documentação interna, Pfleeger (2004) sugere a colocação de comentários (cabeçalho) no início de cada componente (função, procedimento, módulo ou classe), como: nome componente, nome do desenvolvedor, seu objetivo no contexto do projeto geral, quando foi escrito e revisado, entre outros. Além do cabeçalho, devem ser colocados comentários no corpo do código fonte, mas quando o código é bem estruturado, as linhas de código bem formatadas, os rótulos (nomes das variáveis e dos dados) fáceis de distinguir, o número de comentários adicionais é pequeno.

As especificações de requisitos e modelos criados em fases anteriores à codificação são consideradas documentações exter-

nas. Mas há, ainda, a documentação externa do programa. Essa documentação corresponde a um relatório com os itens: quem, o que, por que, quando, onde e como, semelhante ao cabeçalho do código fonte, mas com riqueza de detalhes. A documentação externa de cada componente faz parte da documentação do sistema.

12. PROJETO DE *SOFTWARE*

O projeto é o núcleo técnico da engenharia de *software* e é aplicado independentemente do paradigma de desenvolvimento usado. No projeto os requisitos do cliente, são traduzidos em uma representação de *software* que pode ser avaliado quanto à qualidade, antes que a codificação se inicie.

Na Unidade 2, você teve a possibilidade de aprender sobre os Paradigmas da Engenharia de *Software*, dentre eles o Ciclo de Vida Clássico. Neste paradigma, uma vez feito o levantamento dos requisitos do *software*, ou seja, depois de saber "o que" é necessário para que o *software* cumpra com as necessidades do cliente, passa-se o foco para o "como" – a fase de PROJETO. Nesta fase, o projetista vai estudar as técnicas que devem ser usadas para concretizar as necessidades do cliente. Portanto, a palavra projeto utilizada neste tópico, trata-se desta fase do processo de *software*.

O projeto é considerado o alicerce da qualidade do *software* e serve de base para todos os passos de engenharia, inclusive a manutenção. Segundo Pressman (1996), o projeto é o lugar em que a qualidade é fomentada durante o desenvolvimento de *software* e é a única forma para traduzir os requisitos do cliente em um produto ou sistema acabado.

No projeto é que se tomam decisões que afetarão fortemente o sucesso da implementação e a facilidade com que o *software* será mantido. Negligenciando a fase de projeto, corre-se o risco de se construir um sistema instável, que falhará quando peque-

nas mudanças forem feitas. Algumas características inseridas no *software* na fase de projeto como modularidade, coesão e acoplamento são cruciais para a manutenibilidade dele.

Modularidade

O conceito de modularidade já foi institucionalizado na indústria de *software* como uma das características que facilitam o entendimento, o desenvolvimento e a manutenção do *software*. Pressman (1996) define módulo como componentes de *software*, separadamente nomeados e endereçáveis.

Segundo Pressman (1996), o projeto deve ser modular, ou seja, o *software* deve ser logicamente dividido em componentes que executem funções e subfunções específicas. Além disso, o projeto deve levar à construção de módulos com características funcionais independentes e com interfaces pouco complexas entre eles e o ambiente externo.

Um dos lemas da engenharia de *software* é "dividir para conquistar", quando aplicado ao projeto significa que dividindo um problema complexo em módulos (decomposição), divide-se, também, a complexidade em módulos de complexidades reduzidas e, portanto, mais fáceis de serem compreendidos e desenvolvidos. Isso, porém, não significa que se dividir o *software* indefinidamente em módulos o esforço para desenvolvê-lo será também indefinidamente menor, pois aumenta o esforço para construir as interfaces entre os módulos.

O fator de o projeto ser modular não impede que a implementação seja monolítica (compostos de apenas um módulo), mas *softwares* monolíticos são difíceis de serem construídos e entendidos. A dificuldade no entendimento do código fonte refletirá na dificuldade de dar manutenção a ele.

A divisão do projeto em módulos o torna mais administrável, permitindo uma melhor arquitetura e inteligibilidade do *software*, melhor controle hierárquico entre as funções que o *software* deve desempenhar.

Os módulos devem ser construídos de forma que as informações (dados e instruções) neles contidas sejam inacessíveis por outros módulos que não tenham necessidade de tais informações. Assim, é possível a definição de módulos independentes que se comunicam entre si, fornecendo somente as informações necessárias. Essa ocultação de informações oferece benefícios durante a manutenção, pois como os dados e procedimentos são ocultos das outras partes, erros introduzidos em um módulo durante a modificação são menos prováveis de serem propagados a outros módulos.

Os módulos devem ter alta coesão e baixo acoplamento.

Vejamos, agora, estes dois novos conceitos.

Coesão

Coesão é uma extensão do conceito de ocultação de informações. Segundo Pressman (1996), um módulo coeso executa apenas uma única função no procedimento de *software*, com isso exigi-se pouca relação com os procedimentos executados em outras partes do programa.

Quanto maior a coesão, melhor, pois uma elevada coesão é sempre buscada pelos desenvolvedores, mas a coesão de nível médio é também aceitável. É importante, porém, evitar baixos níveis de coesão no momento em que os módulos estão sendo projetados. A coesão facilita o entendimento do *software* e faz com que a manutenção seja localizada, ou seja, uma solicitação de mudança em uma função específica do *software* exigirá alteração no seu módulo correspondente.

O fato de a modularidade ser reconhecidamente importante na engenharia de *software*, não significa que se deva dividir aleatoriamente o projeto ou o *software* em partes. Para que se tenha coesão, é necessária uma divisão objetiva e cuidadosa em módulos que executem claramente funções específicas.

A coesão elevada tem como principal característica um módulo que executa uma tarefa procedimental distinta, o que facilita

a manutenção, pois a execução de uma tarefa em cada módulo faz com que haja uma diminuição de erros ou falhas no projeto. Isso não acontece com a coesão de nível baixo, em que o módulo realiza várias tarefas, mesmo que seja constituído por entidades funcionais independentes, poderia ser mais bem executado com módulos separados. Tarefas combinadas em apenas um módulo têm como consequência o aumento de erros quando realizada modificações.

Assim, o esforço gasto na construção de módulos coesos é recompensado pelo maior entendimento do *software* construído e pela diminuição de custos e facilidades na execução de futuras manutenções.

Acoplamento

Segundo Pressman (1996), acoplamento é uma medida da interconexão dos módulos de uma estrutura de *software* e depende da complexidade da interface entre os módulos, ou seja, do fluxo de informações entre eles.

Quanto menor o acoplamento, melhor. Uma das formas de facilitar a manutenção é alcançar o acoplamento no nível baixo, pois a conectividade simples entre os módulos tem como resultado um *software* de fácil compreensão e com menor chance de efeitos de propagação de erros.

Projetos com módulos que possuem elevado acoplamento devem ser evitados, pois causam problemas, como perda de muito tempo para encontrar o problema ou erro no projeto, e, consequentemente, gasto desnecessário de dinheiro. Isso não ocorre quando se tem um acoplamento de nível baixo, que é de grande auxílio durante a manutenção de *software*, tornando-o mais receptivo a modificações, facilitando a introdução de novas funcionalidades e correções no *software*.

Segundo Pressman (1996), a orientação a objetos baseia-se em conceitos que começamos a aprender no jardim de infância

como: objetos, atributos, classes e membros, o todo e suas partes. Portanto, torna o desenvolvimento de software mais compatível ao entendimento humano, pois introduz uma maneira de pensar os problemas utilizando modelos organizados a partir de conceitos do mundo real.

O conceito de modularidade, ocultamento de informações, coesão e acoplamento são, também, muito empregados na orientação a objeto. A facilidade de manutenção é uma das características que contribuiu para a ascensão da orientação a objeto como um paradigma de destaque no desenvolvimento de *software*.

13. QUESTÃO AUTOAVALIATIVA

Responda as questões a seguir:

1) São estratégias possíveis de serem adotadas por uma organização para evolução de seus sistemas legados as apresentadas a seguir, **EXCETO**:
   a) descartar o sistema completamente.
   b) substituir todo ou parte do sistema por um novo sistema.
   c) deixar o sistema sem alterações e continuar com a manutenção regular.
   d) fazer reengenharia do sistema para aprimorar sua facilidade de manutenção.
   e) realizar engenharia reversa do código do sistema com o objetivo de aumentar sua complexidade ciclomática e torná-lo crítico à empresa.

Fonte: Prova aplicada em 05/2008 para o concurso da Petrobrás - 2008, realizado pelo órgão/instituição Petróleo Brasileiro SA, área de atuação Outras, organizada pela banca CESGRANRIO, para o cargo de Técnico em Informática, nível médio, área de formação Tecnologia da Informação.

Gabarito

Depois de responder às questões autoavaliativas, é importante que você confira o seu desempenho, a fim de que possa saber se é preciso retomar o estudo desta unidade. Assim, confira, a seguir, as respostas corretas para as questões autoavaliativas propostas anteriormente:

1) (e)

14. CONSIDERAÇÕES

Como podemos observar, a manutenção faz parte do ciclo de vida dos sistemas baseados em computador. A facilidade em dar manutenção é critério fundamental para alongar a vida útil de sistemas baseados em computador e protelar a morte desses sistemas.

Já a dificuldade em dar manutenção em *softwares* é um "fantasma" que acompanha a indústria de *software* desde os seus primórdios. A Engenharia de *Software* já indica os caminhos da manutenibilidade, por meio da documentação, das técnicas de modularidade de projetos entre outras. Mas ainda não é o suficiente.

A próxima unidade tratará das ferramentas que apoiam o todo o processo de desenvolvimento de *software*, inclusive a manutenção.

15. REFERÊNCIAS BIBLIOGRÁFICAS

IEEE. *Standards Collection*: Software Engineering (2007). Disponível em: <http:// standards.ieee.org/ software/>. Acesso em: 25 jun. 2007.

International Standard Organization (ISO). ISO/IEC 9126-1-1nformation technology. Part 1: quality model. Software product quality. Geneva: ISO, 1999. 40 p.

O'Brien, J. A. *Sistemas de Informação*: as decisões gerenciais na era da internet. São Paulo: Editora Saraiva, 2004.

PAULA FILHO. *Engenharia de software*: fundamentos, métodos e padrões. São Paulo: LTC, 2003.

PORTER, M. E. *A Vantagem competitiva das nações*. Rio de Janeiro: Campus. 1993.

PRESSMAN, R. S. *Engenharia de software*. São Paulo: Makron Books, 1996.

_____. *Engenharia de software*. São Paulo: Makron Books, 2006.

SOMMERVILE, I. *Engenharia de software.* 6. ed. São Paulo: Pearson, 2005.

_____. *Engenharia de software*. 8. ed. São Paulo: Pearson, 2007.

SOUZA, S. C. B. de; ANQUETIL, N.; OLIVEIRA, K. M. Documentação Essencial para Manutenção de Software. In: *IV Simpósio Brasileiro de Qualidade de Software*, 2005, Porto Alegre, v. 1. p. 15-20.

EAD

Ferramentas CASE

9

1. OBJETIVOS

- Compreender as ferramentas que automatizam o trabalho do engenheiro de *software*.
- Conhecer os conceitos, objetivos e contexto histórico destas ferramentas.
- Entender como as ferramentas CASE podem aumentar a produtividade da indústria de *software*.
- Conhecer como são classificadas as ferramentas CASE.
- Escolher uma ferramenta CASE.
- Identificar impactos da utilização da ferramenta CASE e os benefícios que essas ferramentas trazem à indústria do *software*.

2. CONTEÚDOS

- Conceitos e objetivos de CASE.
- Contexto histórico.

- Produtividade.
- Classificação.
- Modelo de adoção de CASE.
- Impactos das CASE.

3. ORIENTAÇÕES PARA O ESTUDO DA UNIDADE

Antes de iniciar o estudo desta unidade, é importante que você leia as orientações a seguir:

1) No decorrer desta disciplina, você verificou que o desenvolvimento de *software* envolve várias atividades. Estas atividades precisam ser bem planejadas para que se consiga cumprir os prazos e os custos definidos inicialmente e que o sistema desenvolvido cumpra com os requisitos. A fim de aumentar a produtividade da indústria de *software*, nós desenvolvedores criamos ferramentas que automatizam também o nosso trabalho. Em outras palavras, criamos *softwares* para nos ajudar a desenvolver outros *softwares*.
2) Para uma melhor absorção do conteúdo aqui exposto, interaja com seus colegas e com seu tutor. No *Caderno de atividades e interatividades* (CAI) você encontrará todas as interatividades e atividades propostas. Participe! Lembre-se de que é fundamental que você entregue as atividades nas datas previstas.
3) Sua formação é essencial, pois ela determinará posturas e escolhas no desenvolvimento de sua prática. Invista em você, faça da pesquisa e da interação com seus colegas de curso e com seu tutor hábitos que poderão ajudá-lo a ampliar e a aprofundar seus conhecimentos.

4. INTRODUÇÃO À UNIDADE

Na Unidade 8, você compreendeu que o trabalho do engenheiro de *software* vai muito além de um projeto bem-sucedido e

do *software* entregue ao cliente. Pelo contrário, mudanças serão necessárias durante a vida do *software* e, muitas vezes, o trabalho de manter o *software* é muito maior do que o esforço para desenvolvê-lo.

Nesta unidade, você conhecerá como as ferramentas podem automatizar o trabalho do engenheiro, minimizando seu esforço e aumentando a produtividade da indústria do *software*.

Vamos lá!

5. CONCEITOS E OBJETIVOS DAS CASES

Na Unidade 1, você conheceu algumas definições da Engenharia de Software e viu que a Engenharia de Software é fundamentalmente composta por três elementos, sendo eles: métodos, ferramentas e procedimentos.

As ferramentas proporcionam um apoio automatizado ou semiautomatizado aos métodos. Quando as ferramentas são integradas de forma que a informação criada por uma ferramenta possa ser usada por outra, é estabelecido um sistema de suporte ao desenvolvimento de *software* chamado – Engenharia de Software Auxiliada por Computador, ou CASE, do inglês *Computer-Aided Software Engineering* (PRESSMAN, 1995).

As ferramentas proporcionam um apoio automatizado aos métodos. Hoje em dia, já existem ferramentas capazes de sustentar cada um dos métodos já citados nas unidades anteriores, automatizando várias fases do processo de desenvolvimento de *software*.

A seguir, você pode observar algumas definições, de diferentes autores, para essas ferramentas:

- CASE refere-se a uma ampla gama de diferentes tipos de programas utilizados para apoiar as atividades do processo de *software*, como análise de requisitos, modelagem de sistema, depuração e testes (SOMMERVILLE, 2005).

- CASE é uma ferramenta ou conjunto de técnicas facilitadoras de desenvolvimento de *software* moderno, que utiliza técnicas para ajudar no trabalho dos engenheiros de *software* (REZENDE, 2005).
- Uma ferramenta CASE é um instrumento ou sistema automatizado utilizado para realizar uma tarefa da melhor maneira possível. Essa melhor maneira pode significar que a ferramenta nos torna mais precisos, eficientes e produtivos ou que exista melhora na qualidade do produto resultante (PFLEEGER, 2004).

De acordo com Sommerville (2005), os engenheiros fazem os produtos funcionarem, aplicando teorias, métodos e ferramentas nas situações apropriadas de modo seletivo. O *software* não é apenas um programa de computador, é, também, toda a documentação associada e os dados necessários para fazer com que esses programas funcionem corretamente.

A Engenharia de Software não cuida apenas do desenvolvimento de um *software*, mas também do desenvolvimento de novas ferramentas ou da melhoria das ferramentas que já existem para suporte e apoio ao *software*.

Então, ferramentas CASE são *softwares* que auxiliam a desenvolver novos *softwares*? Aproveite as definições anteriormente expostas e crie sua própria definição

Você pode perceber que o objetivo de qualquer indústria é satisfazer às necessidades de seus clientes entregando produtos com qualidade e aumentando, assim, a produtividade de seus processos de produção. Na indústria de *software*, esses objetivos não são diferentes.

Assim, a Engenharia de Software tem como objetivo o aperfeiçoamento da qualidade dos *softwares* desenvolvidos e o aumento da produtividade dos engenheiros que os desenvolve, visando sistematizar sua manutenção, de modo que aconteça dentro de

prazos estimados, com progresso controlado e usando métodos, tecnologias e processos em continuo aprimoramento (REZENDE, 2005).

Você se lembra da Crise do *Software*? Crise do *Software* é o termo que resume todos os problemas que permeiam o desenvolvimento de *software*. Tais problemas são focados, especialmente, no não cumprimento de prazos e custos estabelecidos e na entrega de *software* com a qualidade inferior à esperada. Um dos sintomas da crise, que permeiam até os dias atuais, é a dificuldade de suprir a demanda por novos *softwares*. As ferramentas CASE são promessas da Engenharia de *Software* para automatizar tarefas, diminuir o tempo de desenvolvimento e, assim, atender melhor a crescente demanda por novos *softwares*.

Em outras palavras, a preocupação em desenvolver ferramentas que automatizam o trabalho de engenheiros de *software* é uma das tentativas de aumentar a produtividade da indústria de *software* e, portanto, uma das armas para enfrentar a crise.

6. CONTEXTO HISTÓRICO

Pressman (1995) lembra-nos o velho ditado sobre os filhos do sapateiro, ele passa a maioria do tempo fazendo sapatos para os outros e seus próprios filhos não têm sapatos feitos por ele.

Esse ditado é análogo ao ditado popular “em casa de ferreiro o espeto é de pau”.

Segundo Pressman (1995), nos últimos 20 anos, a grande maioria dos engenheiros de *softwares* tem sido como os filhos do sapateiro, constroem *softwares*, sistemas complexos que automatizam o trabalho para os outros, e, para si mesmos, não têm usado quase nada para automatizar o ambiente de trabalho. Recentemente, a engenharia de *software* era fundamentalmente uma atividade manual em que ferramentas eram usadas somente em últimos estágios do processo.

É inevitável, para compreender a origem das ferramentas CASE, fazer uma relação com a origem das ferramentas CAD já consagradas na automatização das mais diversas engenharias. O posicionamento que as ferramentas CASE ocupa hoje na Engenharia de Software é a mesma que as CAD ocupavam nas décadas de 1970. As ferramentas CASE estão para a Engenharia de *Software* assim como o CAD está para a Engenharia Civil. As ferramentas CAD implementam práticas de engenharia que foram experimentadas e provadas há muito tempo, já as CASE apresentam um conjunto de ferramentas semi-automatizado e automatizado que estão implementando uma cultura de engenharia que é novidade para muitas pessoas.

Por volta da década de 1950, os engenheiros mecânicos e elétricos utilizavam ferramentas manuais na criação de seus projetos, como réguas de cálculo e calculadoras mecânicas, livros e tabelas que continham as fórmulas e algoritmos que precisavam para efetuar a análise de um problema de engenharia. Um bom trabalho era feito, mas era feito manualmente.

Passou-se uma década e o mesmo grupo de engenharia começou a experimentar a engenharia baseada por computador, mas ainda com a resistência de alguns integrantes do grupo, pois eles não confiavam no resultado.

No ano de 1975, todas as fórmulas que os engenheiros necessitavam estavam embutidas em um grande conjunto de programas de computador, usados para analisar uma ampla variedade de problemas de engenharia, e, assim, tornou-se inevitável o uso de tais ferramentas por essas pessoas, que começaram a confiar nos resultados desses programas. Assim, uma ponte entre o trabalho de engenharia e de manufatura estava em construção, criando o primeiro elo entre o Projeto Auxiliado por Computador (*Computer-Aided Design* – CAD) e a Manufatura Auxiliada por Computador (*Computer-Aided Manufacturing* – CAM) (PRESSMAN, 1995).

Como você pode perceber, os engenheiros de *software* passaram anos desenvolvendo soluções para outros profissionais,

sem se preocupar ou sem tempo para pensar em suas próprias necessidades. No entanto, com o passar dos anos e a institucionalização da produção de *software* como indústria, estas necessidades se afloraram e, finalmente, os engenheiros de *software* ganharam sua primeira ferramenta auxiliada por computador, a Engenharia de *Software* Auxiliada por Computador (*Computer–Aided Software Engineering* – CASE).

Atualmente, as ferramentas CASE fazem parte da caixa de ferramenta do engenheiro de *software*, que lhe proporcionam a capacidade de automatizar as atividades manuais e de melhorar a informação da engenharia (PRESSMAN, 1995).

7. PRODUTIVIDADE EM *SOFTWARE*

A definição do que é produtividade na indústria de *software* ainda é controversa, pois a produtividade em *software* não é uma medida direta. Assim como nas outras engenharias, a engenharia de *software* propõe algumas métricas para deter dados tangíveis sobre o processo e o produto de *software*, como, por exemplo, Linhas de Código (*Line-of-Code* - LOC) e Pontos-por-Função (*Function-Point* – FP), já mencionados no estudo da Unidade 3. A produtividade em *software*, porém, não é uma medida exata, há vários fatores que influenciam na produtividade.

Segundo Pressman (1995), há, no mínimo, cinco fatores que interferem na produtividade do *software*, são eles:

a) **Fatores humanos**: o tamanho e a experiência da organização de desenvolvimento.
b) **Fatores do problema**: a complexidade do problema a ser resolvido e o número de mudanças nos requisitos ou restrições de projeto.
c) **Fatores do processo**: técnicas de análise e projeto que são usadas, como: linguagens e ferramentas CASE disponíveis e técnicas de revisão.

d) **Fatores do produto**: confiabilidade e desempenho do sistema baseado em computador.

e) **Fatores relacionados ao recurso**: disponibilidade de ferramentas CASE, recursos de *hardware* e *software*.

Portanto, a disponibilidade de ferramentas CASE é considerada relevante, influenciando em até 40%, segundo Pressman (1995), na busca pelo aumento da produtividade do processo de *software*.

Para Sommerville (2005), as ferramentas de tecnologia CASE possuem facilidades gráficas para o planejamento, projeto e construção de sistemas. Elas podem ser utilizadas para gerar um esboço do programa, a partir de um projeto. Isso inclui o código, implementar interfaces, e em vários casos, o desenvolvedor precisa apenas acrescentar pequenos detalhes da operação de cada componente do programa. Elas também podem incluir geradores de códigos, que, automaticamente, originarão código-fonte com base no modelo de sistema e, também, algumas orientações de processo, que fornece conselhos ao engenheiro de *software* sobre o que fazer em seguida.

8. CLASSIFICAÇÃO DAS CASE

De acordo com Sommerville (2005), em uma perspectiva de processo, isto é, quanto às fases do processo que a ferramenta automatiza, as CASE podem se dividir em três categorias:

1) ***Front End* ou *Upper-CASE*:** são aquelas ferramentas que dão apoio à análise e ao projeto, isto é, às fases iniciais do desenvolvimento do *software*.

2) ***Back End* ou *Lower-CASE*:** são aquelas ferramentas destinadas a dar apoio à implementação e aos testes, como depuradores, sistemas de análise de programa, geradores de casos de testes e editores de programas.

3) ***I-CASE* ou *Integrated CASE*:** são as ferramentas que têm como objetivo unir a *Upper-CASE* à *Lower-CASE*, isto é, cobrem todo o ciclo de vida do *software*.

Já Pressman (1995), em uma perspectiva de função, ou seja, de acordo com sua função específica que automatiza, as CASE podem ser classificadas em oito ferramentas. Observe a seguir a descrição de cada uma.

1) **Ferramenta de planejamento de sistemas comerciais:** estas ferramentas constituem uma "meta-modelo", com base na qual sistemas de informação específicos são derivados. Em vez de se concentrar nos requisitos de uma aplicação específica, a informação comercial é modelada à medida que circula entre as várias entidades organizacionais dentro de uma empresa. O objetivo dessa categoria é melhorar a compreensão de como a informação circula entre as várias unidades organizacionais.
2) **Ferramenta de gerenciamento de projetos:** a maioria dessas ferramentas concentra-se em um elemento específico do gerenciamento de projetos ao invés de oferecer um suporte abrangente à atividade de gerenciamento. Ao ser usado um conjunto selecionado de ferramenta CASE, o gerente de projetos pode gerar estimativas úteis de esforço, custo e duração de um projeto de *software*, definir uma estrutura de divisão de trabalho, planejar uma programação viável de projeto e acompanhar projetos em base contínua. Além disso, essas ferramentas podem ser usadas para compilar métricas e rastrear os requisitos.
3) **Ferramenta de apoio:** abrange as ferramentas de aplicação e de sistemas que complementam o processo de Engenharia de *Software*. Dentro dessa ampla categoria, estão incluídas as ferramentas de documentação, de rede e de *software* básico, ferramenta de garantia de qualidade e ferramenta de gerenciamento de banco de dados e de configuração.
4) **Ferramentas de análise e projeto:** possibilitam que o engenheiro de *software* crie um modelo do sistema que será construído. Essas ferramentas também auxiliam na criação do modelo e na avaliação da qualidade do modelo.
5) **Ferramenta de programação:** abrangem compiladores, editores e depuradores que se encontram à disposição

para apoiar a maioria das linguagens de programação convencionais. Estão nessa categoria as linguagens de quarta geração, os geradores de aplicações e as linguagens de consulta a banco de dados.

6) **Ferramentas de integração e teste:** auxiliam na aquisição de dados de testes, na análise do código-fonte, no planejamento, no gerenciamento e controle de testes.
7) **Ferramenta de prototipação:** dão suporte à criação de modelos para prototipação.
8) **Ferramentas de manutenção:** auxiliam na execução de engenharia reversa, análise e reestruturação de código e na reengenharia.

Veja que há várias classificações para as ferramentas CASE. Elas podem ser classificadas por sua função, por seus papéis como instrumentos para os gerentes e para o pessoal técnico, pelo uso que elas possuem nas várias etapas do processo de Engenharia de *Software*, pela arquitetura de ambiente, *hardware* ou *software* que suporta ou, até mesmo, pela origem ou custo delas.

A Engenharia de *Software* Auxiliada por Computador pode ser tão simples quanto uma única ferramenta que suporte uma atividade de Engenharia de *Software* específica ou tão complexa quanto um ambiente completo que abrange ferramentas, banco de dados, pessoas, *hardware*, rede, sistemas operacionais, padrões e uma infinidade de outros componentes (Pressman, 1995). Enfim, todas as ferramentas "*software*s" que, de alguma forma, auxiliam nos trabalhos de um engenheiro de *software*, podem ser considerada como CASE.

Há várias classificações de ferramentas CASE que apoiam às diversas fases do processo de *software*. Para cada uma dessas classificações, há, no mercado, inúmeras ferramentas disponíveis. Entre elas, podemos citar: *Poseidon*, *Rational*, *ErWin*, *Oracle Designer*, *Genexus*, *Clarify*, *Dr.Case*, *Multicase*, *Paradigm*, *PowerDesigner*, *Together*, *Cognos*, *CoolGen*, *Smart*, *Theseus*, *BPWin*, *Arena*, *Visio*, *Brio*, *Microstrategy*.

9. MODELO SEI DE ADOÇÃO DE CASE

O SEI (*Software Engineering Institute* – Instituto de Engenharia de *Software*), *Carnegie Mellon University, Pittsburgh*, *Pennsylvania*, USA, desenvolveu um processo de adoção de ferramentas CASE. O modelo tem como postulados seis estágios para um processo de adoção de ferramentas CASE, conforme você pode acompanhar na Figura 1.

Maiores informações sobre o modelo SEI (*Software Engineering Institute*) podem ser encontrados no site do Instituto Nacional de Pesquisas Espaciais. Disponível em: <www2.dem.inpe.br/ijar/Guiacase.doc>. Acesso em: 05 jul. 2010.

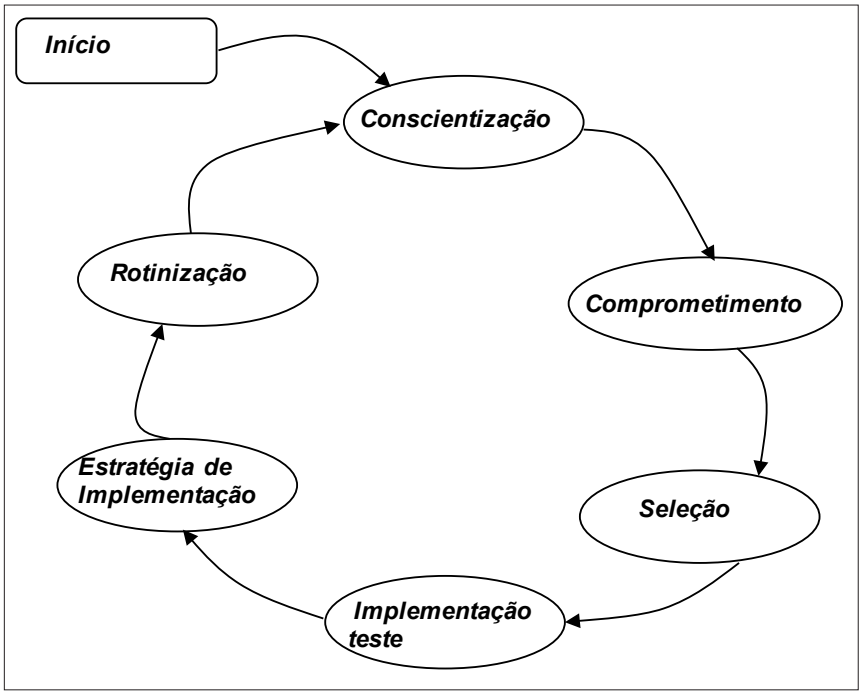


Figura 1 *Modelo de adoção de CASE*

Os seis estágios descritos na Figura 1 representam um ciclo em que cada estágio fornece a entrada para o próximo estágio. Dependendo da maturidade de uma organização antes do esforço de

adoção, ou seja, antes de adquirir de fato uma ferramenta CASE, alguns dos estágios preliminares podem já ter sido completados.

1) **Conscientização**: é uma busca preliminar de informação sobre ferramentas CASE antes de se fazer qualquer comprometimento para adotar a tecnologia CASE. É necessário consultar literaturas, participar de *workshops* e seminários, e solicitar informações de outras organizações que já adotaram tecnologia CASE. Conscientização da força e da fraqueza da tecnologia CASE.
2) **Comprometimento**: é o processo de decisão para adotar ferramentas CASE. Comprometimento do gerente e daqueles que utilizarão ou que sofrerão o impacto da ferramenta. Um erro comum nessa fase é menosprezar a importância do comprometimento daqueles cujas atividades serão afetadas pela incorporação da nova tecnologia. A relutância da gerência em introduzir as mudanças organizacionais necessárias para o sucesso da implementação da CASE representa a maior barreira para aumentar a produtividade do *software*. A forma mais fácil de convencer a gerência é provar que existem substanciais ganhos em produtividade e qualidade a serem atingidos pela implementação da tecnologia CASE. Essa evidência, porém, não está prontamente disponível.
3) **Seleção**: esta estratégia deve ter como foco as necessidades de curto e de longo prazos da organização. Baseada nas necessidades identificadas, começa-se a escolha de uma ferramenta individual. Uma abordagem de seleção de ferramenta apropriada inclui:
    - elaborar uma pequena lista das opções de ferramentas;
    - determinar como a nova ferramenta irá interagir com outras ferramentas no ambiente;
    - analisar as ferramentas candidatas de acordo com os critérios técnicos e não técnicos.
4) **Implementação (Teste)**: uma vez selecionada uma ferramenta, é importante testá-la em um projeto piloto. Muitas organizações pulam esse passo, pelo fato de ele

exigir significantes recursos, incluindo pessoal, tempo e dinheiro. Entretanto, somente em um projeto piloto, executado em condições reais, determinará o que a ferramenta oferece, como ela funciona, como ela efetivamente executa uma tarefa e quais são seus defeitos (aspectos verificados em uma avaliação experimental e não somente em demonstração de vendedores). As ferramentas demonstram melhor suas capacidades com dados reais e não em ambientes arranjados por tutoriais de vendedores. Se a ferramenta funciona bem no projeto piloto, poderá, também, suportar seu uso contínuo (cuidado para não gerar expectativas irreais). Durante o período de teste, é importante executar uma análise objetiva no completo ciclo de desenvolvimento, com simulações reais do tamanho do *database* e múltiplos usuários. Este tipo de teste dará uma ideia melhor das funções específicas fornecida pelas ferramentas individuais e uma ideia de como as várias ferramentas trabalharão juntas. Um projeto piloto bem-sucedido é o mínimo para se estabelecer um compromisso organizacional seguro.

5) **Estratégia de implementação**: o primeiro desafio desse estágio envolve a integração da nova ferramenta com o ambiente existente, podendo afetar muitos elementos no ambiente, incluindo o pessoal, processos e métodos existentes. O pessoal afetado pode ser resistente às mudanças. As características de uma ferramenta podem requerer mudanças nos métodos e processos. A ferramenta em si pode requerer personalização para suportar as necessidades organizacionais. Estratégias de implementação pobres podem levar ao desuso da ferramenta. É importante reconhecer que a tarefa de implementação de uma ferramenta não se completa com uma seleção cuidadosa e a subsequente demonstração de suas possibilidades em um projeto piloto. Deve, também, ser reconhecido que o uso de uma ferramenta pode ser contraprodutivo. Uma boa estratégia de implementação é algo que encoraja o uso da ferramenta.

6) **Rotinização**: representa o início da fase de manutenção para uma ferramenta CASE. Muitos esforços na adoção de ferramentas têm falhado por causa da inabilidade de incorporar a ferramenta no dia a dia das atividades e no planejamento da organização. É necessário, portanto, a "doutrinação" de novos funcionários dentro do sistema e a contínua atualização dos perfis dos funcionários existentes com foco no treinamento, nos recursos necessários ao suporte da ferramenta, complexidade da ferramenta, avanços da ferramenta e sistemas.

Agora que você conhece os seis estágios para o processo de adoção de uma ferramenta CASE, é importante saber que, quando uma empresa que desenvolve *software* se dispõe a adquirir uma ferramenta CASE, esta empresa se coloca, nesse momento, no papel de cliente da indústria de *software*. Interessante, não? E, dessa forma, deixamos por um momento o papel de desenvolvedor e passarmos a ser clientes. A aversão à mudança, a necessidade de treinamento, os problemas de implantação, as necessidade de suporte, que, normalmente, nossos clientes sofrem, nós, desenvolvedores, é que sofremos na implantação de uma CASE.

10. IMPACTO DAS CASE

Você pode ter percebido, no decorrer desta unidade, que muitas são as promessas das ferramentas CASE, tanto para o aumento da produtividade do processo quanto para a melhoria da qualidade do produto de *software*. Mas será que a Engenharia de *Software* já está obtendo tantos benefícios das CASE, assim como as demais engenharias obtêm benefícios das ferramentas CAD? Essa pergunta é difícil de responder, mas é fundamental analisarmos algumas situações. E, para realizarmos essa análise, contamos com informações contidas no *Guia para adoção de CASE* (disponível no *site*: <www2.dem.inpe.br/ijar/Guiacase.doc>. Acesso em: 21 de mar. 2010). Observe a seguir.

1) Idealmente, as CASE têm como promessas:
   a) encorajar um ambiente interativo;
   b) reduzir custos de manutenção;
   c) melhorar a qualidade do produto de *software*;
   d) agilizar o processo de desenvolvimento;
   e) aumentar a produtividade.
2) Na realidade, tem-se observado os seguintes benefícios decorrentes da utilização de ferramentas CASE:
   a) CASEs de gerenciamento de configuração e documentação são, geralmente, mais aceitas como mecanismo de melhoria do *software*;
   b) benefícios controversos de CASEs de análise e projeto, engenharia reversa e ferramentas de geração de códigos disponíveis comercialmente;
   c) ganhos variando de 10% a 30 % resultante do uso de CASE na análise e projeto;
   d) ganhos verdadeiros ocorrem somente depois de um ou dois anos de experiência;
   e) ganhos variáveis de produtividade;
   f) modestos ganhos de qualidade;
   g) documentação melhorada (aumento da manutenibilidade);
   h) melhoria na comunicação;
   i) imposição de metodologia e padrões.
3) Há alguns fatores que influenciam na análise do impacto das CASE, tais como:
   a) existência de dados fantasiosos sobre CASE;
   b) natureza diversificada das ferramentas e dos processos organizacionais (algumas ferramentas se acomodam a alguns processos e a outros não);
   c) diferenças nas medições de qualidade e produtividade interferem na avaliação do impacto das CASEs;
   d) tamanho do projeto (normalmente, limitadas para projetos grandes), características dos clientes e usuários também influem no impacto da ferramenta;

e) diversidade de aplicações dificultam comparações;
f) disponibilidade de dados históricos influem na eficiência da CASE;
g) tempo de utilização da ferramenta.

4) Como os benefícios das CASE são ainda relativos, as empresas que se dispuserem a adquirir uma ferramenta para automatizar o processo de *software*, devem considerar alguns itens com relação a CASE em questão, como, por exemplo:
   a) custo (investimentos) de adotar a tecnologia CASE;
   b) consistência entre os processos e métodos suportados pelas ferramentas CASE e os processos e métodos utilizados na organização;
   c) mecanismos de suporte necessários para ferramentas CASE (por parte do fornecedor);
   d) limites da ferramenta quanto ao tamanho do projeto;
   e) complexidade da adoção e usabilidade mínima. Complexidade dos processos de adoção das ferramentas CASE;
   f) capacidade de acomodar mudanças, uma vez que os requisitos se modificam;
   g) permissão da engenharia reversa dos *softwares* desenvolvidos sem usar uma ferramenta CASE.

O sucesso ou falha do esforço de adoção da CASE depende muito da habilidade de uma organização para gerenciar custos de curto e de longo prazo.

1) Portanto, devem ser consideradas algumas implicações de curto prazo:
   a) um potencial decaimento na produtividade;
   b) insatisfação de parte dos funcionários ao adotar a nova tecnologia;
   c) mudanças nos processos e métodos;
   d) treinamento potencialmente extensivo;
   e) custos significativos.

2) Além disso, há implicações de longo prazo a serem consideradas:
   a) custo da manutenção das ferramentas CASE (reestruturação, versionamento);
   b) lançamento frequente de nova tecnologia;
   c) custos contínuos para treinamento de novos funcionários e atualização dos funcionários existentes e treinados.

Tendo em vista a crescente demanda por novos sistemas computadorizados, a engenharia de *software* empenha-se em criar métodos e ferramentas que aumente a produtividade dos processos de desenvolvimento de *software*. As ferramentas CASE são consideradas grandes promissoras na busca do desafio de aumentar a qualidade e a produtividade dos processos de *software*.

Apesar de serem consideradas, muitas vezes, ferramentas de custo elevado e de utilização complexa, uma vez inserida e institucionalizada no processo de desenvolvimento de *software*, certamente, trarão benefícios ao projeto por meio do aumento da produtividade.

Assim, a Engenharia de *Software* oferece métodos e técnicas para desenvolver ferramentas automatizadas para auxiliar no trabalho de profissionais das mais diversas áreas de atuação. Tais ferramentas – *softwares* – são bem aceitas, prova disso é a crescente busca por novos, e cada vez mais complexos, sistemas.

Apesar de ainda ser um conceito relativamente novo no mercado, as ferramentas CASE já estão auxiliando os engenheiros de *softwares* nas diversas fases de seu trabalho. As ferramentas CASE são consideradas como uma alternativa promissora da Engenharia de *Software* em aumentar a produtividade da indústria do *software*, para tanto, essa engenharia empenha-se na construção de ferramentas cada vez mais sofisticadas para auxiliar no trabalho de seus profissionais.

11. QUESTÕES AUTOAVALIATIVAS

Sugerimos que você procure responder, discutir e comentar as questões a seguir

1) Ferramentas CASE é uma classificação que abrange todas ferramentas baseada em computadores que auxiliam atividades de engenharia de *software*, desde análise de requisitos e modelagem até programação e testes. Nesse contexto, é correto afirmar que:
   a) a maior quantidade de códigos de programação é compensada pela melhoria e redução de custos na manutenção.
   b) as ferramentas de codificação são classificadas na categoria *Upper Case.*
   c) as ferramentas de análise, projeto e implementação estão classificadas na categoria *Integrated Case.*
   d) geralmente dispensam capacitação específica dos recursos da empresa.
   e) um dos componentes indispensáveis de uma ferramenta CASE é a modelagem visual, ou seja, a possibilidade de representar, através de modelos gráficos, o que está sendo definido.

Fonte: Prova aplicada em 12/2009 para o concurso do TRE-AM - 2010, realizado pelo órgão/instituição Tribunal Regional Eleitoral do Amazonas, área de atuação Jurídica, organizada pela banca FCC, para o cargo de Analista Judiciário - Tecnologia da Informação, nível superior, área de formação Tecnologia da Informação .

Gabarito

Depois de responder às questões autoavaliativas, é importante que você confira o seu desempenho, a fim de que possa saber se é preciso retomar o estudo desta unidade. Assim, confira, a seguir, as respostas corretas para as questões autoavaliativas propostas anteriormente:

1) (e)

12. CONSIDERAÇÕES

Chegamos ao final do estudo da Unidade 9, durante o qual você teve a possibilidade de aprender o que são ferramentas CASE e como elas podem auxiliar os engenheiros de *software* na automatização de seu trabalho.

Você pôde perceber, também, que essas ferramentas são complexas, mas bastante promissoras para aumentar a produti-

vidade dos processos de *software*. Além disso, pelo fato de aumentar a precisão dos processos, essas ferramentas auxiliam na construção de produtos de *software* com maior qualidade.

As ferramentas CASE são ainda novas, mas a Engenharia de *Software* está se empenhando para desenvolver ferramentas cada vez mais adequadas aos profissionais de *software*.

Na próxima e última unidade, você terá a oportunidade de conhecer mais alguns conceitos sobre a tão almejada qualidade de produto e qualidade de processo de *software*.

13. E-REFERÊNCIA

Lista de figuras

Figura 1 – Modelo de adoção de CASE. Disponível em: <www2.dem.inpe.br/ijar/Guiacase.doc>. Acesso em: 21 mar. 2010.

14. REFERÊNCIAS BIBLIOGRÁFICAS

PFLEEGER, S. L. *Engenharia de Software teoria e Prática*. 2. ed. São Paulo: Prentice Hall, 2004.

PRESSMAN, R. S. *Engenharia de software*. São Paulo: Makron Books, 1995.

REZENDE, D. A. *Engenharia de Software e Sistemas Operacionais*. Rio de Janeiro: Brasporte, 2005.

SOMMERVILE, I. *Software engineering.* 4th ed. Addison: Wesley, 1994.

_____. *Engenharia de Software*. 6. ed. São Paulo: PearsonAddison Wesley, 2005.

EAD

Qualidade de *Software*

10

1. OBJETIVOS

- Compreender e refletir a importância da qualidade de *software*.
- Perceber a existência e as peculiaridades das técnicas e estratégias que focam na qualidade do *software*.
- Compreender que a qualidade do produto é dependente da qualidade do processo.
- Conhecer modelos para a melhoria da qualidade do *software*.

2. CONTEÚDOS

- Definições de qualidade e qualidade de *software.*
- Qualidade de processo e qualidade de produto.
- Fatores de qualidade de *software*.
- Garantia da qualidade de *software.*

- Métrica de qualidade.
- Estimativa de *software*.
- Qualidade de processo de *software*.
- Modelo CMMI.

3. ORIENTAÇÕES PARA O ESTUDO DA UNIDADE

Antes de iniciar o estudo desta unidade, é importante que você leia as orientações a seguir:

1) No decorrer desta disciplina foram citadas inúmeras vezes a questão da qualidade. A qualidade é uma das metas da Engenharia de *Software*. De forma resumida, podemos dizer que o *software* que não faz o que o cliente quer, é um *software* sem qualidade.
2) Nesta unidade, você verá que a qualificação dos processos de produção é o responsável pela inserção da qualidade ao produto final. As normas de qualidade para o processo de *software* nada mais são do que o agrupamento dos conceitos de Engenharia de *Software* vistos durante o estudo desta disciplina.
3) No decorrer desta unidade tenha em mente as seguintes reflexões:
   - Os investimentos em qualidade é uma realidade nas indústrias para que elas mantenham a sua competitividade. Você acredita que investimentos em qualidade são, também, uma questão de sobrevivência na indústria do *software*?
   - Você acha que a qualidade do produto pode ser considerada consequência direta da qualidade do processo de *software*?
4) Sugerimos que acesse o *site* <http://www.sei.cmu.edu/> para você conhecer mais sobre o CMMI e outros assuntos relacionados ao SEI – *Series in software engineering* (Série da engenharia de software).

4. INTRODUÇÃO À UNIDADE

Na unidade anterior, você teve a oportunidade de estudar e discutir a importância da manutenção de *software* para a indústria de *software.*

Nesta unidade, discutiremos os pontos relacionados à qualidade de produto e de processo de *software*. Na verdade, abordaremos alguns conceitos definidos pelo Instituto de Engenharia de *Software* (SEI - *Software Engineering Institute).*

Não conseguiríamos falar em desenvolvimento de um produto de *software* sem mencionar fatores relacionados à sua qualidade.

Entretanto, qualidade pode ser um conceito subjetivo – o que é qualidade para uma pessoa, pode não ser para outra.

Dessa forma, para se medir a qualidade de um *software* conceitos subjetivos não podem ser considerados. É necessário, então, que se tenha uma definição precisa do que é um *software* de qualidade ou, pelo menos, quais são as propriedades que devem caracterizar um *software* desenvolvido segundo os princípios da Engenharia de *Software* para que se possa atestar sua falta, parcial ou total de qualidade.

Mas para chegar ao conhecimento de um modelo de certificação de qualidade, temos que, primeiro, conhecer o que é qualidade nesse contexto de Engenharia de *Software*.

5. DEFINIÇÃO DE *SOFTWARE* DE QUALIDADE

Quando falamos em certificação de qualidade em empresas, geralmente, pensamos em certificações ISO, especificamente as relacionadas à ISO 9000 e suas derivações.

Certificações ISO possuem restrições, listas de atributos e níveis que as empresas devem obter, relacionadas à qualidade, para que possam ser certificadas.

Segundo a **norma ISO 9000** (versão 2000), foi definido que a qualidade é o grau em que um conjunto de características inerentes a um produto, processo ou sistema cumpre os requisitos inicialmente estipulados para estes (Disponível em: < http://www.pucrs.br/edipucrs/online/projetoSI/6-Engenharia/qualidadeSoftware.htm>. Acesso em: 6 abr. 2010).

É importante mencionar que a qualidade deve ser avaliada de forma diferente quando analisamos um produto de *software* ou um processo de desenvolvimento de *software*. São diretrizes diferentes para contextos diferentes.

Antes de continuarmos, pense um pouco: o que é qualidade para você? Será que você e seus amigos têm a mesma ideia sobre qualidade? Quando você acha uma pessoa simpática é sempre unanimidade? Dizem que "a qualidade está nos olhos de quem avalia."...

A definição do termo qualidade depende do âmbito em que ocorre, possuindo diversas interpretações. Por esse motivo, a qualidade continua sendo mal entendida com decorrências que prejudicam as organizações que desejam implementá-la. Uma melhor compreensão do termo é fundamental para que a qualidade possa assumir função estratégica na competitividade. Na formulação de um conceito de qualidade, é essencial a clareza de uma definição, mas o fundamental é que o seu significado esteja perfeitamente entendido e que seja uma linguagem comum por toda a organização.

Veremos, agora, o que significa qualidade para os dois grandes nomes da qualidade total: Juran e Deming.

Pesquisando sobre a bibliografia destas duas personalidades – Edwards Deming e Joseph Moses Juran – você conhecerá a origem da qualidade e como foi a sua repercussão pelo mundo.

Juran (1992,) admite a existência de várias definições para o termo qualidade, descrevendo o termo como o desempenho do produto. Esse desempenho resulta das características do produto que levam à satisfação do cliente e interferem na decisão de com-

pra. O autor ainda descreve a qualidade como a ausência de deficiências, as deficiências do produto afetam os custos, por falhas no uso, repetição de trabalho e desperdício. Unindo o desempenho do produto à ausência de deficiências, pode-se descrever a qualidade como a adequação ao uso.

Deming (1990) também reconhece a amplitude do conceito de qualidade e informa que ela só pode ser definida de acordo com quem a avalia. Ele afirma que qualidade não significa luxo, qualidade é qualquer coisa que o cliente necessita ou deseja. E como as necessidades e desejos dos clientes estão sempre mudando, a solução para a definição da qualidade em termos de cliente é redefinir os requisitos constantemente. Deming enfatiza uma sistemática para solução de problemas de qualidade conhecida por Ciclo *Deming* ou PDCA – *Plan, Do, Check, Action*. Este método de análise sistemática é a base para o processo de melhoria contínua em todos os níveis da organização que deseja qualidade.No mundo de Deming, você nunca chega à perfeição, ou seja, você sempre pode fazer melhor e nunca deve interromper o processo de aperfeiçoamento.

No que diz respeito à área de desenvolvimento de *software*, uma das definições mais citadas é a de Crosby, também derivada das definições anteriores. Para Crosby (1991), qualidade é a conformidade com os requistos e é medida pelo custo da não qualidade. Para ele, qualidade é um estado binário: ou há conformidade (qualidade) ou há não conformidade (não qualidade).

Com os avanços da informática, cada vez mais os computadores passam a integrar a rotina diária e a produção de *software* vem sofrendo um constante aumento. Os sistemas baseados em computador estão sendo usados nas mais diversas áreas e, na maioria das situações, não admitem erros.

Considerando a importância do *software* na atualidade, aliada às crescentes exigências dos usuários, tem-se verificado um notável aumento nos interesses pela qualidade do *software*.

Apesar de várias as definições de qualidade, precisamos verificar o que é qualidade nesse contexto, para entendermos porque as empresas procuram tanto um *software* com qualidade.

Para Pressman (2006), **qualidade de *software*** tem como objetivo garantir a qualidade final do *software* a ser desenvolvido. Essa garantia é estabelecida por meio da definição e normatização dos processos de desenvolvimento. É importante lembrar que a qualidade de *software* é uma área dentro da engenharia de *software*.

Apesar dos diversos modelos aplicados na questão "qualidade de *software*" atuarem durante todo o processo de desenvolvimento, o foco principal está na satisfação do cliente ao receber o produto pronto. É importante garantir que o sistema cumpra com todas as especificações acordadas anteriormente, entre a empresa desenvolvedora e o cliente e, nesse momento, percebemos, novamente, a importância de uma definição de requisitos bem elaborada.

Entretanto, é simplista dizer que avaliação de qualidade de um *software* só poderá ser realizada depois que o *software* for entregue para o cliente, pois deve-se garantir a qualidade desde o início da construção do *software*, controlando sua implementação passo a passo e medindo sua qualidade antes que ele saia da "fábrica".

Você, provavelmente, ouviu falar que a qualidade das partes garante a qualidade do todo, é mais ou menos isso o que acontece: ao garantirmos a qualidade das partes do *software*, estamos garantindo a qualidade total do produto.

Portanto, há fatores internos e externos que estão relacionados à qualidade de um *software*.

- **Fatores de qualidade externos:** são aqueles que podem ser detectados, especialmente, pelo cliente ou por eventuais usuários. A partir da observação de alguns fatores,

como, por exemplo: o desempenho, a facilidade de uso, a correção, a confiabilidade, a extensibilidade, dentre outros. O cliente, de acordo com seu ponto de vista, pode concluir sobre a qualidade do *software*.

- **Fatores de qualidade internos:** são aqueles relacionados à visão de um programador, particularmente, do programador que assumirá as tarefas de manutenção do *software*. Nessa classe, encontram-se fatores como: modular idade, legibilidade, portabilidade, manutenibilidade, dentre outros.

Em ambas as visões, os requisitos de *software* formam a base em que a qualidade é avaliada. Pode-se afirmar que a falta de conformidade com os requisitos é falta de qualidade. Alem dos requisitos, há padrões específicos que definem os critérios de desenvolvimento que servem como diretrizes para o *software* que está sendo produzido. Se os critérios não são seguidos, resulta na falta de qualidade (PRESSMAN, 1996).

Mesmo observando que são diferentes entre si, a garantia da qualidade de *software*, tanto de fatores externos como de fatores internos é que garantirá um bom produto de *software*.

Segundo Staa (1987), pode-se dizer que um *software* de boa qualidade é aquele que:

a) produz resultados úteis e confiáveis na oportunidade certa;
b) é ameno ao uso, mensurável, corrigível, modificável e evolutível;
c) opera em máquinas e ambientes reais;
d) é desenvolvido de forma econômica e no prazo estipulado;
e) opera com economia de recursos.

A qualidade de *software* é um conceito muito mais amplo do que um *software* correto e bem documentado, requerendo metodologias e técnicas de desenvolvimento específicas.

6. QUALIDADE DE PROCESSO E QUALIDADE DE PRODUTO

Raramente, a qualidade pode ser incorporada ao produto final após o processo de desenvolvimento terminado. Portanto, a qualidade do produto de *software* é um objetivo do processo de desenvolvimento. Assim, ao desenvolver um produto, deve-se ter previamente estabelecidas as características de qualidade que se deseja alcançar. Se o processo de desenvolvimento de *software* considerar as características de qualidade, há uma grande tendência de que o produto final apresente tais características.

Apesar de serem conceitos relacionados e, na maioria das vezes, dependentes, pode-se fazer um estudo separado dos conceitos de qualidade de produto e qualidade de processo de *software*.

A qualidade do produto diz respeito a certas características que contribuem para a sua utilidade. Se um produto tem de ser apropriado para um uso específico, ele, deve conter alguns atributos importantes, denominados fatores de qualidade (PRESSMAN, 1996).

Com relação aos produtos de *software*, os fatores de qualidade focalizam três aspectos importantes: sua adaptabilidade diante de novos ambientes, suas características operacionais e sua habilidade de suportar mudanças.

7. FATORES DE QUALIDADE DE *SOFTWARE*

Os fatores que afetam a qualidade de *software* podem ser categorizados em:

- fatores que podem ser medidos diretamente (erros de execução);
- fatores que podem ser medidos apenas indiretamente (usabilidade do *software*).

Observe, a seguir, alguns fatores a serem considerados de acordo com Pressman (2006), relacionados à qualidade do produto de *software*.

Corretitude – Ele faz aquilo que eu quero?

Capacidade dos produtos de *software* de realizar suas tarefas de forma precisa, conforme definido nos requisitos e na especificação.

Confiabilidade – Ele se comporta com precisão o tempo todo?

Capacidade do sistema de funcionar mesmo em condições anormais. É um fator diferente da corretitude. Um sistema pode ser correto sem ser confiável, ou seja, seu funcionamento ocorrerá apenas em determinadas condições.

Flexibilidade – Posso mudá-lo?

Facilidade com a qual se introduz modificações nos produtos de *software*. Todo *software* é considerado, em princípio, "flexível" e, portanto, passível de modificações. No entanto, esse fator nem sempre é muito bem entendido, em especial quando se trata de pequenos programas.

Por outro lado, para *softwares* de grande porte, esse fator tem uma importância considerável, e pode ser atingido a partir de dois critérios importantes:

- **Simplicidade de projeto:** quanto mais simples e clara a arquitetura do *software*, mais facilmente as modificações poderão ser realizadas.
- **Descentralização:** implica maior autonomia dos diferentes componentes de *software*, de modo que a modificação ou a retirada de um componente não implique uma reação em cadeia que altere todo o comportamento do sistema, podendo até introduzir erros antes inexistentes.

Reusabilidade – Serei capaz de reutilizar parte do *software*?

Capacidade dos produtos de *software* serem reutilizados, totalmente ou em parte, para novas aplicações.

Atualmente, esse conceito é extremamente utilizado no desenvolvimento organizado de produtos de *software*.

A necessidade da reusabilidade vem da constatação de que muitos componentes de *software* obedecem a um padrão comum, o que permite, então, que essas similaridades sejam exploradas para a obtenção de soluções para diferentes classes de problemas.

Esse fator possibilita, especialmente, a economia e um nível de qualidade satisfatório na produção de novos *softwares* e, dessa forma, menos programas precisam ser escritos, o que significa menos esforço e menor risco de ocorrência de erros.

Compatibilidade – Serei capaz de compor uma interface com outro sistema?

Quando falamos em compatibilidade, estamos nos referindo à facilidade que produtos de *software* têm ao serem combinados com outros. Esse é um fator relativamente importante, pois um produto de *software* é construído (e adquirido) para trabalhar com outros *softwares*.

Eficiência **– Ele rodará no *hardware* de destino tão bem quanto possível?**

A eficiência está relacionada com a utilização racional dos recursos de *hardware* e de sistema operacional da plataforma na qual o *software* será instalado.

Portabilidade – Serei capaz de utilizá-lo em outra máquina?

A portabilidade consiste na capacidade de um *software* em ser instalado para diversos ambientes de *software* e *hardware*. A portabilidade nem sempre é uma característica facilmente atingi-

da, devido, especialmente, às diversidades existentes nas diferentes plataformas em termos de processador, composição dos periféricos, sistema operacional etc.

Usabilidade – Ele foi projetado para o usuário?

Esse fator é certamente detectado pelos usuários do *software*. Quanto mais fácil de ser utilizado, melhor ele será considerado.

Manutenibilidade – Posso consertá-lo?

A manutenibilidade está relacionada ao esforço exigido para localizar e reparar erros em um programa, além de adequá-lo a novas versões e atualizá-lo de forma eficaz e eficiente.

Testabilidade

Relacionado ao esforço despendido para testar um *software*, a fim de garantir que execute todas as funções para as quais foi projetado.

Integridade – Ele é seguro?

Se o sistema pode ser facilmente acessado por pessoas não autorizadas.

Os itens anteriores citados podem, de acordo com referências existentes, serem medidos e, por meio do resultado, ser definido seu fator de qualidade.

8. GARANTIA DE QUALIDADE DE *SOFTWARE*

A garantia da qualidade é uma atividade fundamental para qualquer negócio que gere produtos ou serviços, como vimos na Unidade 6.

Em resumo, a garantia de qualidade de *software* engloba algumas atividades como:

a) atividade de teste – que vimos na Unidade 7;
b) padronizações e procedimentos formais – são aplicados ao processo de engenharia de *software* para o desenvolvimento de *software* com qualidade;
c) controle de mudança;
d) medição;
e) manutenção.

Algumas dessas atividades veremos a seguir; outras, sugiro que as pesquisem, porque é fundamental conhecê-las, mas, devido à carga horária da disciplina, não teremos tempo para abordá-las como gostaríamos.

Na unidade 6, você conheceu os conceitos relacionados à Garantia de Qualidade de *Software*.

9. MÉTRICA DE QUALIDADE DE *SOFTWARE*

A possibilidade de estabelecer uma medida da qualidade é um aspecto importante para a garantia de um produto de *software.*

Mas como medir, por exemplo, o quanto um *software* será fácil ou não de dar manutenção? Ou como garantir a segurança de um *software*?

É nesse contexto de "como medir" que um novo conceito na área de Engenharia de *Software* é inserido: conceito de métrica.

Uma vez que as medidas quantitativas (mensuráveis) têm-se provado eficientes em vários ramos da ciência, cientistas de computação têm trabalhado arduamente para aplicar métodos similares no desenvolvimento de *software*.

Para Cavano e MacCall (*apud* PRESSMAN, 1996, p. 753):

> A determinação da qualidade é fundamental nos eventos cotidianos – concursos de degustação de vinhos, eventos esportivos, concursos de talento, etc. Nessas situações, a qualidade é julgada na maneira mais fundamental e direta: uma comparação lado a lado dos objetos sob condições idênticas e com conceitos predetermina-

> dos. O vinho pode ser julgado de acordo com a clareza, cor, buquê, sabor, etc. Porém, esse tipo de julgamento é muito subjetivo; para ter qualquer valor absoluto, ele deve ser feito por um especialista.
>
> A subjetividade e a especialização também se aplicam na determinação da qualidade de *software*. Para ajudar a resolver esse problema, uma definição mais precisa de qualidade de software é necessária, bem como uma forma de derivar medições quantitativas de qualidade de software para análise objetiva... Uma vez que não existe essa coisa de conhecimento absoluto, ninguém deve esperar medir qualidade de software exatamente, porque cada medição é parcialmente imperfeita. Jacob Bronowsky descreveu esse paradoxo do conhecimento desta maneira: "Ano a ano deparamo-nos com instrumentos cada vez mais precisos com os quais podemos observar a natureza com mais precisão. E quando olhamos para as observações ficamos desconsertados ao ver que elas ainda são vagas, e achamos que elas continuam tão incertas como sempre.

De acordo com a citação anterior, percebemos que medir qualidade não é fácil, mas nem por isso especialistas deixaram de tentar e de criar modelos para que chegássemos, o mais próximo possível, de uma medição eficaz.

É importante salientar que não medimos diretamente a qualidade de software, mas a manifestação dessa qualidade durante sua execução.

Observe a seguir algumas técnicas de medição.

Métricas de dimensão e complexidade

Atualmente, têm sido propostas inúmeras métricas para medir a dimensão e a complexidade de um programa. Optou-se por apresentá-las em conjunto, pois, em sua maioria, a mesma métrica é apresentada como responsável pela quantificação ora da dimensão, ora da complexidade.

A. Linhas de Código Fonte (LOC - "Lines Of Code")

É uma métrica de dimensão que, apesar de criticada, é, ainda, a mais utilizada.

Embora aparentemente simples, essa métrica obriga a uma definição inequívoca das regras de contagem de linhas, nomeadamente no tocante ao tratamento a dar às linhas em branco e de comentário, instruções não executáveis, diretivas de compilação, múltiplas instruções por linha ou múltiplas linhas por instrução, bem como no caso de reutilização de código.

B. Métricas de Halstead

É um conjunto de métricas proposto por **Maurice Halstead** que se baseia na teoria da informação e que foi designado por "*Software Science*".

Maurice Halstead, em 1972, iniciou estudos sobre algoritmos tentando testar empiricamente a hipótese de que os operadores (comandos e palavras reservadas) e os operandos (itens de dados) em um programa deviam relacionar-se com a quantidade de erros nos algoritmos.

Essa métrica utiliza medidas primitivas para desenvolver expressões para o comprimento global do programa, o volume mínimo potencial para um algoritmo, o volume real (medido em bits), o nível do programa e outras características como esforço do desenvolvimento, tempo de desenvolvimento e, até mesmo, o número projetado de falhas no *software*.

Utiliza-se de cálculos matemáticos para obter o valor métrico da qualidade e complexidade do *software*.

Pressman (1996) defende que a métrica de comprimento de Halstead é objetiva e melhor que a LOC.

C. Métricas de McCabe

A primeira e mais conhecida métrica proposta por Thomas McCabe é a *métrica de complexidade ciclomática*.

Essa métrica pressupõe que a complexidade depende do número de decisões (complexidade ciclomática). Ela é adimensional e corresponde ao número máximo de percursos linearmente independentes por meio de um programa.

Os caminhos podem ser representados por meio de um grafo orientado em que os nós representam uma ou mais instruções sequenciais e os arcos orientados indicam o sentido do fluxo de controle entre várias instruções. Observe o gráfico de fluxo de controle na Figura 1.

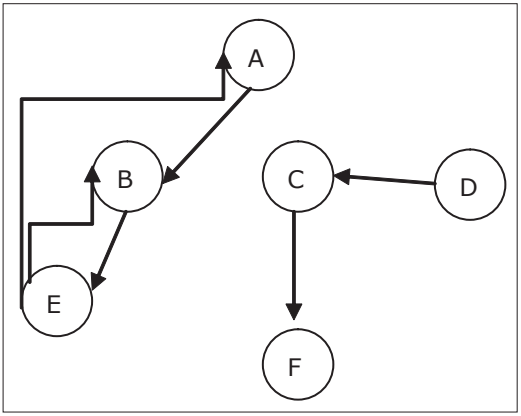


Figura 1 *Complexidade do gráfico de fluxo de controle.*

Vale a pena você pesquisar e conhecer outras métricas, como, por exemplo:

Métrica dos nós: proposta por Woodward.

Métrica dos fluxos de informação: proposta por Henry e Kafura.

Métrica voltada para orientação a objeto.

10. ESTIMATIVA DE *SOFTWARE*

Até o momento estávamos estudando métricas de *softwares*, por meios de indicadores.

As métricas de *software* têm como princípios especificar as funções de coleta de dados de avaliação e desempenho, atribuir essas responsabilidades a toda a equipe envolvida no projeto, reunir dados de desempenho pertencentes à complementação do *software*, analisar os históricos dos projetos anteriores para de-

terminar o efeito desses fatores e utilizar os efeitos para pesar as previsões futuras.

Agora, iniciaremos o estudo de estimativa. Estimar e medir possuem definições diferentes, mas suas atividades são integradas/interdependentes.

Estimar significar valorar esforço, prazos e custos no desenvolvimento de *software*.

Quando falamos em estimativa de *software*, é fundamental que fique claro que é difícil afirmar se é possível desenvolver o produto desejado pelo cliente antes de conhecer os detalhes do projeto.

Por isso, uma boa definição de requisitos e encontros periódicos com os clientes é fundamental para se estimar o tempo de desenvolvimento, custo, tamanho do projeto etc.

O desenvolvimento de um *software* é um processo gradual de refinamento e devemos sempre lembrar que:

a) a incerteza da natureza do produto contribui para a incerteza da estimativa;
b) requisitos e escopo mudam;
c) defeitos, geralmente, são encontrados e demandam retrabalho;
d) a produtividade varia de pessoa para pessoa.

O **processo de estimativa** envolve cinco etapas básicas:

a) estimar o tamanho do produto;
b) estimar o tempo;
c) estimar o esforço;
d) estimar o custo (envolve quatro fatores);
e) estimar o prazo.

É viável fornecer estimativas dentro de uma faixa de valores permitidos e, com o passar do tempo e a partir do momento em que se conhece mais e mais o projeto, refinar esse limite.

Estimativa de tamanho

É a dimensão do *software* a ser produzido.

A estimativa de tamanho pode ser avaliada por técnicas de números de linhas de código, números de pontos de função, números de requisitos, pontos de casos de uso etc.

Estimativa de tempo

Após desenvolver uma estimativa do **volume de trabalho** a ser realizado, não é fácil estimar a **extensão de tempo** que o projeto exigirá, uma vez que a preocupação com a relação **tempo/pessoa** é de extrema complexidade e importância. Então, é preciso tomar muito cuidado, são necessárias técnicas de estimativas, base histórica e conhecimento sobre o projeto estimado para um bom andamento do projeto.

Estimativa de esforço

- É a **técnica mais comum** para se levantar os custos de qualquer projeto de desenvolvimento de engenharia.
- A estimativa de esforço inicia-se com um delineamento das funções do *software* obtidas a partir do escopo do projeto.
- O planejador estima o esforço (por exemplo, pessoas/mês) que seria exigido para se concluir cada tarefa de engenharia de *software* para cada função de *software*. Taxas de mão de obra (isto é, custo/esforço unitário) são aplicadas em cada tarefa de engenharia de *software*.

Estimativa de custo

A estimativa de custo tem como objetivo calcular de maneira antecipada todo e qualquer custo que esteja associado ao sistema, tais como: construção, instalação, operação e manutenção.

Custo da construção

O custo da construção é importante, pois permite sabermos o total das pessoas envolvidas no desenvolvimento do sistema, tais como: burocratas, diretores, membros da comunidade usuária, consultores e programadores, membros da auditoria, do controle de qualidade ou da equipe de operações.

Custo da instalação do sistema

É um projeto simples que podemos entregar em disquetes ou em CD-ROMs e a instalação fica por conta do próprio usuário. Em caso de sistemas grandes, porém, o processo de instalação é mais complexo e envolve outros fatores como: custo de treinamento do usuário, custo de conversão de banco de dados, custo de instalação do fornecedor, custo da aprovação legal etc.

Custo operacional

O custo operacional inicia-se após a instalação do sistema. É quando haverá um custo para o usuário manter sua operação. Contudo, isso também representará uma área em que seu novo sistema economizará dinheiro, pois ele, presumivelmente, será mais barato que o atual sistema. Os custos mais comuns são: custos de *hardware* e suprimentos, custos de *software*, custo de pessoal, custo de manutenção etc.

Custo de manutenção ou falhas

Está relacionado às diversas formas de erros que tornam o sistema completamente indisponível, até que o erro seja corrigido, enquanto, em outros casos, o sistema continua funcionando, porém uma ou mais de suas saídas podem estar incorretas. Esse custo é importante pelo fato de não termos sistemas perfeitos.

Estimativa de prazo

Corresponde, geralmente, a datas fornecidas pelo cliente e, sempre que possível, devem ser respeitadas.

Fator humano

Quando os objetivos para o desenvolvimento de sistemas não são claros, os desenvolvedores começam a deduzir e a criar o produto dentro de suas próprias visões, levando a sistemas inadequados para a função do negócio a ser atendida e, consequentemente, as métricas falham, gerando uma **expectativa divergente** entre o cliente e os técnicos responsáveis, isto é, uma *estimativa irreal*.

As pessoas são sensíveis aos estímulos externos e, por meio destes, são influenciadas, modificando suas atitudes e pensamentos. Um analista, ou um grupo de analistas, disposto a estimar o tempo e o custo de um projeto não poderia deixar de dar a devida relevância a tais fatores.

Engenharia humana

Para tentar amenizar a dificuldade e estabelecer critério para a estimativa em relação às pessoas, surge o conceito de "Engenharia Humana", que consiste em aplicar conceitos de psicologia para se projetar uma interação **homem-computador** de alta qualidade. Do ponto de vista do especialista em engenharia humana, o homem e a máquina são partes integrantes de todo um sistema. Ele vê o homem como um elo de coleta e *processamento de dados*.

As estimativas jamais poderão ser precisas e exatas, pois não são apenas fatores técnicos, "contáveis" e palpáveis que fazem parte de um projeto, mas há, também, pessoas, sentimentos, políticas, crenças, ambiente e outros fatores que não se podem estimar, portanto são absolutamente variáveis.

Lembrem-se: estimar é necessário sim, mas com forte embasamento teórico e prático. Estimar não é adivinhar.

11. QUALIDADE DE PROCESSO DE *SOFTWARE*

Com o passar do tempo, uma das maiores evoluções no estudo da qualidade, descoberta ao longo dos anos, foi a verificação

de que a qualidade do produto é importante, mas a qualidade do processo de produção tem uma importância ainda maior.

Um processo de *software* é um conjunto de ferramentas, métodos e práticas utilizados para produzir o *software*. O processo de *software* é representado por um conjunto sequencial de atividades, objetivos, transformações e eventos que encapsulam estratégias para o cumprimento da evolução de *software* (PRESSMAN, 1996).

Em meados da década de 1980, surgiu um movimento que focalizou os processos de desenvolvimento de *software*. As falhas nos processos de gerência e a manutenção do desenvolvimento de *software* foram reconhecidas como inibidores principais no crescimento da qualidade e produtividade (LAMPRECH, 1994). O processo utilizado no desenvolvimento de um projeto tem grande reflexo na produtividade e na qualidade do *software* desenvolvido. Os estudos recentes sobre qualidade são voltados para o melhoramento do processo de desenvolvimento de *software*, pois garantir a qualidade do processo é um grande passo para garantir, também, a qualidade do produto. Após anos de estudos e promessas para solucionar problemas de tecnologia com mais tecnologia, surge um movimento de compreensão de que os problemas de desenvolvimento de *software* não são apenas tecnológicos, mas gerenciais e organizacionais.

Um processo de *software* bem definido e documentado, utilizado para integrar pessoas, tarefas, ferramentas e métodos, pode prover uma base essencial para garantir a qualidade do produto final. O processo utilizado para desenvolver e manter o *software* afeta significativamente o custo, a qualidade e o prazo de entrega do produto. O impacto é tão significante que a melhoria do processo de *software* é vista por algumas pessoas como a mais importante forma para melhorar o produto de *software*.

Um processo disciplinado faz com que o desenvolvimento e a manutenção sejam indistinguíveis em um ciclo de evolução do produto. Para que o desenvolvimento e a manutenção sejam fei-

tos de maneira ordenada, as organizações devem ter seus processos como sendo passíveis de controle, melhorias e medidas. Isso requer programas voltados para o aumento da produtividade e da qualidade. Antes de qualquer tentativa de uma organização aperfeiçoar seu processo, deve-se conhecer como o *software* é desenvolvido. O conhecimento de onde o esforço deve ser concentrado é essencial. Sem o conhecimento do tipo de processo existente e os produtos característicos produzidos durante o desenvolvimento, não há um meio de aperfeiçoar ou quantificar a qualidade do produto (KAN, 1995).

Alguns autores na área de Engenharia de *Software* recomendam que o processo de desenvolvimento seja uma atividade que deve ser controlada, medida e melhorada, e esse é um passo importante para tornar o processo realmente efetivo. Para isso, é necessário que se tenha o controle do relacionamento entre as atividades, as ferramentas, o pessoal, os métodos e, especialmente, as características do produto de *software*, que é o objetivo final de todo o processo.

A importância da qualidade no setor de *software* é indiscutível. Assim, muitas instituições preocuparam-se em criar modelos para auxiliar na busca da tão almejada qualidade tanto do produto quanto do processo de *software*.

Com a intensa dinâmica que rege os processos de *software*, a definição explícita de um processo já não é uma tarefa trivial, gerenciá-los, então, é ainda mais difícil. Para amenizar esta dificuldade, foi necessária a definição de modelos de apoio à melhoria do processo de *software*.

Entre os modelos disponíveis, os que se destacam são: a norma ISO/IEC 12207 (Processos de Ciclo de Vida de *Software*) [ISO 95]; o projeto SPICE (*Software Process Improvement and Capability Determination*) ou futura norma ISO/IEC 15504 [ISO 98]; e o CMMI (*Capability Maturity Model Integration*).

12. MODELO CMMI

Há inúmeras abordagens de avaliação e certificação de qualidade de processo de *software*.

Algumas mais outras menos utilizadas e conhecidas no Brasil; entretanto, por não serem muito utilizadas no Brasil ou divulgadas não significa que sejam menos importantes.

Neste tópico, abordaremos um modelo conhecido como cmmi – *capability maturity model integration* (modelos integrados de maturidade e capacidade).

Para entendermos o modelo CMMI, vamos, inicialmente, conhecer um pouco da sua história.

CMMI é uma evolução do CMM e procura estabelecer um modelo único para o processo de melhoria corporativa, integrando diferentes modelos e disciplinas de desenvolvimento de *software* como: gerência de projetos de desenvolvimento de *software*, engenharia de *software*, engenharia de sistemas etc (WIKIPEDIA, 2007).

Com efeito, quando falamos em modelos, estamos nos referindo ao objetivo pelo qual estabelecemos, com base em estudos, históricos e conhecimento operacional, um conjunto de "melhores práticas" que serão utilizadas para um fim específico; nesse caso, desenvolvimento de projeto e de *software*.

A causa mais comum do insucesso dos projetos de desenvolvimento de *software* é a má utilização ou a completa indiferença aos métodos e ferramentas orientados à concepção.

Há tantas ferramentas, métodos e práticas que podem nos auxiliar a desenvolver sistemas cada vez com mais qualidade e menores custos.

É possível que os projetos possam ter bons resultados em empresas de desenvolvimento de *software* em que a utilização de

metodologias consistentes não é prática adotada; mas, em geral, esses bons resultados são muito mais uma consequência de esforços individuais do que propriamente causados pela existência de uma política e de uma infraestrutura adequada à produção de *software*.

É nesse cenário que entra a utilização de um modelo.

Como o CMMI é uma evolução do CMM, vamos conhecer um pouco mais do CMMI, e, para conhecer sua origem (CMM), sugiro que pesquisem.

O **modelo CMMI –** *Capability Maturity Model Integragion* – foi definido pelo SEI – *Software Engineering Institute* – na *Carnegie Mellon University*, Pensilvânia (EUA) – com o objetivo de estabelecer conceitos relacionados aos níveis de maturidade das empresas de desenvolvimento de *software*, com respeito ao grau de evolução em que elas se encontram nos seus processos de desenvolvimento.

O modelo estabelece, também, quais as providências necessárias para as empresas aumentarem, gradualmente, seu grau de maturidade, melhorando, por consequência, sua produtividade e a qualidade do produto de *software.* Esse modelo (CMM e, posteriormente, CMMI) é considerado como referência.

Como dissemos anteriormente, o CMMI procura estabelecer um modelo único para o processo de melhoria corporativa, integrando modelos e disciplinas, tais como: engenharia de *software* e sistemas, fontes de aquisição e desenvolvimento integrado do produto. A seguir, descreveremos algumas dessas disciplinas para que você consiga ter um pouco mais de entendimento sobre CMMI.

Um **processo de desenvolvimento de *software*** corresponde ao conjunto de atividades, métodos, práticas e transformações que uma equipe utiliza para desenvolver e manter um *software* e seus produtos associados (planos de projeto, documentos de projeto, código, casos de teste e manuais de usuário). Uma empresa é

considerada em um maior grau de maturidade quanto mais evoluído for seu processo de desenvolvimento de *software*.

A **capabilidade** de um processo de *software* está relacionada aos resultados obtidos pela sua utilização em um ou em vários projetos. Essa definição permite estabelecer uma estimação de resultados em futuros projetos.

O **desempenho** de um processo de *software* representa os resultados que são fluentemente obtidos pela sua utilização. A diferença básica entre esses dois conceitos (**capabilidade e desempenho**) está no fato de que, enquanto o primeiro está relacionado aos resultados "esperados", o segundo relaciona-se aos resultados que foram efetivamente obtidos.

A **maturidade** de um processo de *software* estabelece os meios pelos quais ele é definido, gerenciado, medido, controlado e efetivo, implicando um potencial de evolução da capabilidade. Em uma empresa com alto grau de maturidade, o processo de desenvolvimento de *software* é bem entendido por todo o *staff* técnico, devido à existência de documentação e políticas de treinamento, que é continuamente monitorado e aperfeiçoado por seus usuários.

À medida que uma organização cresce em maturidade, ela institucionaliza seu processo de desenvolvimento de *software* por meio de políticas, normas e estruturas organizacionais, as quais geram uma infraestrutura e uma cultura de suporte aos métodos e procedimentos de desenvolvimento.

CMMI é uma certificação que empresas de Tecnologia da Informação podem obter para garantir a qualidade do processo e, consequentemente, do seu *software*. Essa certificação como a ISO abre novos mercados e novas parcerias

O primeiro passo para a implantação do modelo e certificação CMMI é a identificação, a qual é realizada por meio de um método definido pelo SEI e conduzido por um avaliador credenciado. Geralmente, realizado por empresas que possuem *know-how* do processo

de certificação e por si só é certificada para dar esse tipo de suporte, partindo do estágio em que a empresa se encontra no presente.

A empresa é avaliada e é estabelecido o nível de maturidade a ser alcançado, visando ajudá-la no desenvolvimento e na manutenção dos projetos de *software*, como também melhorar a capacidade de seus processos.

Após a verificação do estágio da empresa, será analisada qual a próxima etapa a ser alcançada e quais as competências que devem ser adquiridas nesse processo.

Essa fase é importante, pois permite alcançar o sucesso e, consequentemente, a melhoria na qualidade dos serviços e produtos fornecidos pela área de tecnologia da empresa e assim sucessivamente a cada fase/nível.

Representações da CMMI

- **Representação contínua:** possibilita à organização utilizar a ordem de melhoria que atender aos objetivos de negócio da empresa. É caracterizado por níveis de capacidade (*Capability Levels*).
- **Representação por estágios**: é uma sequência pré-definida para melhoria baseada em estágios. Cada estágio é considerado base para o próximo. É criado por níveis de maturidade.

A. Representação por estágios: níveis de maturidade no processo de desenvolvimento de *software*

A representação por estágio de maturidade é o mais utilizado.

Os níveis de maturidade tiveram sua origem no modelo CMM. E esses níveis garantem a certificação CMMI descrita anteriormente.

São definidos cinco níveis de maturidade, que dizem respeito ao processo de desenvolvimento de *software* adotado no nível

das empresas, estabelecendo uma escala ordinal que conduz as empresas ao longo de seu aperfeiçoamento.

Um nível de maturidade é um patamar de evolução de um processo de desenvolvimento de *software*, correspondendo a um degrau na evolução contínua de cada organização.

Cada nível corresponde a um conjunto de objetivos que, uma vez atingidos, estabilizam um componente fundamental do processo de desenvolvimento de *software*, tendo como consequência direta o aumento da capabilidade da empresa.

A Figura 2 apresenta os cinco níveis de maturidade propostos pelo modelo CMMI, no qual se pode observar, também, o estabelecimento de um conjunto de ações que permitirão a uma empresa subir de um degrau para o outro nessa escala.

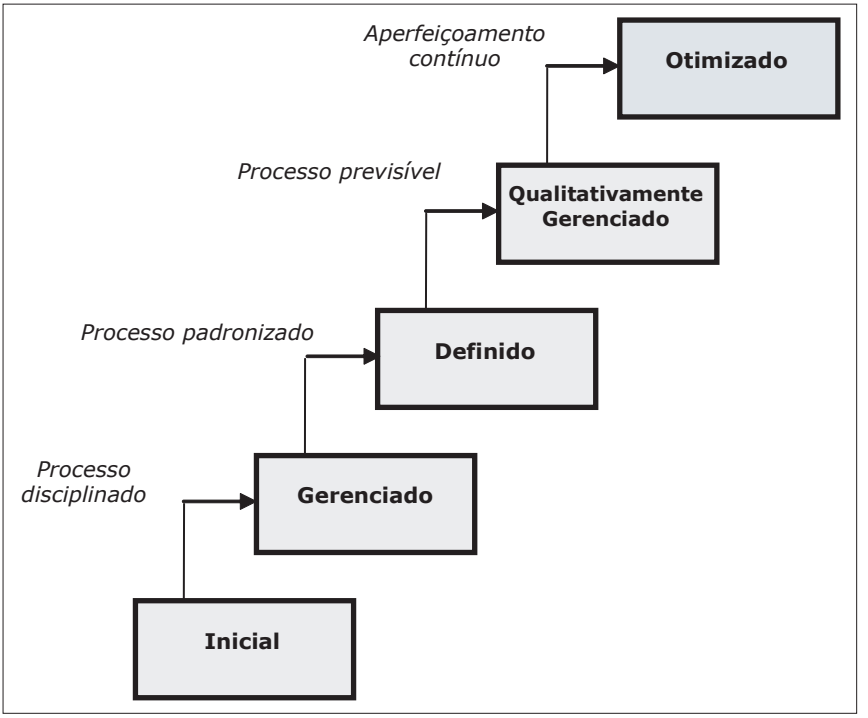


Figura 2 *Níveis de maturidade – CMMI.*

Observe, a seguir, a descrição de cada nível e suas principais características.

1. Nível inicial (initial)

No **nível inicial**, o desenvolvimento de *software* é realizado de forma totalmente "ad hoc", sem uma definição de processos.

Caso ocorra algum problema durante a realização de um projeto, a organização tem a tendência de abandonar totalmente os procedimentos planejados e passa a um processo de codificação e testes; dessa forma, o produto obtido pode apresentar um nível de qualidade suspeito. Veja a seguir as principais características deste nível.

Características do *initial*

Completa falta de planejamento. Os funcionários estão focados, basicamente, em atividades corretivas que surgem a todo o momento. Não existe controle dos processos

A capabilidade de uma empresa caracterizada como nível 1 é totalmente imprevisível, uma vez que o processo de desenvolvimento de *software* é instável, sujeito a mudanças radicais frequentes, não apenas de um projeto a outro, mas também durante a realização de um mesmo projeto.

Nesse nível, estimação de custos, prazos e qualidade do produto é algo totalmente fora do contexto e da política de desenvolvimento, que, vale lembrar, "é inexistente".

Embora não se possa "assegurar" a esse nível o fracasso de um projeto, é possível dizer que o sucesso é, geralmente, resultado de esforços individuais, variando com as habilidades naturais, o conhecimento e as motivações dos profissionais envolvidos no projeto.

2. Nível gerenciado (managed)

Nesse nível, políticas de desenvolvimento de *software* e tarefas de suporte a essas políticas são estabelecidas, e o planeja-

mento de novos projetos, geralmente, são baseados em experiências obtidas em projetos anteriores.

Para uma empresa atingir esse nível, é imprescindível institucionalizar o gerenciamento efetivo dos seus projetos de *software*, de modo que o sucesso dos projetos anteriores possa ser repetido nos projetos em curso.

Nesse nível, os requisitos do *software* e o trabalho a ser realizado para satisfazê-los são planejados e supervisionados ao longo da realização do projeto. São definidos, também, padrões de projeto e cabe à instituição garantir sua efetiva implementação.

A capabilidade de uma empresa situada nesse nível pode ser caracterizada como disciplina, em razão dos esforços de gerenciamento e acompanhamento do projeto de *software*. Veja a seguir as principais características deste nível.

Características do *managed*

Processos básicos de gerenciamento de projeto são estabelecidos para planejar e acompanhar custos, prazos e funcionalidades. Compromissos são firmados e gerenciados. A disciplina de processo permite repetir sucessos de projetos anteriores em aplicações similares. Tipicamente, possui gerenciamento de projetos estabelecido; procedimentos técnicos escritos; acompanhamento de qualidade; gerência de configuração inicial; atividades básicas de medição e análise. O sucesso depende, basicamente, do gerenciamento do projeto.

3. Nível definido (defined)

No **nível definido**, o processo de desenvolvimento de *software* é consolidado tanto do ponto de vista do gerenciamento quanto das tarefas de engenharia a realizar. Esse processo é realizado por meio de documentação, padronização e integração no contexto da organização, que adota essa versão para produzir e manter o *software*.

Os processos definidos nas organizações situadas nesse nível são utilizados como referência para os gerentes de projeto e os membros do *staff* técnico, os quais são baseados em práticas propostas pela Engenharia de *Software*.

Programas de treinamento são promovidos ao nível da organização, como forma de difundir e padronizar as práticas adotadas no processo definido.

As características particulares de cada projeto podem influir no aprimoramento de um processo de desenvolvimento, e, para cada projeto em desenvolvimento, este pode ser instanciado.

Um processo de desenvolvimento bem definido deve conter padrões, procedimentos para o desenvolvimento das atividades envolvidas, mecanismos de validação e critérios de avaliação.

A capabilidade de uma empresa no nível 3 é caracterizada pela padronização e consistência, uma vez que as políticas de gerenciamento e as práticas da Engenharia de Software são aplicadas de forma efetiva e repetidas. Veja a seguir as principais características deste nível.

Características do *defined*

Atividades de gerenciamento básico e as de Engenharia de Software são documentadas, padronizadas e integradas em processos-padrão. Todos os projetos de desenvolvimento ou manutenção de softwares utilizam uma versão de um desses processos adaptada às características específicas de cada projeto. Possui processos gerenciais e técnicos bem definidos, possibilidade de avaliação do processo; ferramentas e metodologias padronizadas; medições iniciais de desempenho; inspeções e auditorias rotineiras; testes padronizados; gerência de configuração; evolução controlada dos processos técnicos e gerenciais.

4. Nível qualitativamente gerenciado (quantitatively managed)

No **nível qualitativamente gerenciado**, é realizada a coleta de medidas do processo e do produto obtido, o que permitirá um controle sobre a produtividade (do processo) e a qualidade (do produto).

É definida uma base de dados para coletar e analisar os dados disponíveis dos projetos de *software*. Medidas consistentes e bem definidas são, então, uma característica das organizações situadas nesse nível, as quais estabelecem uma referência para a avaliação dos processos de desenvolvimento e dos produtos.

Os processos de desenvolvimento exercem um alto controle sobre os produtos obtidos; as variações de desempenho do processo podem ser separadas das variações ocasionais (ruídos), especialmente, no contexto de linhas de produção definidas. Os riscos relacionados ao aprendizado de novas tecnologias ou sobre um novo domínio de aplicação são conhecidos e gerenciados cuidadosamente.

A capabilidade de uma organização situada nesse nível é caracterizada pela previsibilidade, uma vez que os processos são medidos e operam em limites conhecidos. Veja a seguir as principais características deste nível.

Características do *quantitatively managed*

Características: métricas detalhadas do processo de software e da qualidade do produto são coletadas. Tanto o processo quanto o produto de software são quantitativamente compreendidos, avaliados e controlados. Relatórios estatísticos são gerados. Tipicamente, encontra-se estabelecido e em uso rotineiro um programa de medições, a qualidade é planejada por um grupo dedicado, sendo rotineiramente avaliada e aprimorada.

5. Nível otimizado (optmizing)

No **nível otimizado**, a organização promove contínuos aperfeiçoamentos no processo de desenvolvimento, utilizando uma realimentação quantitativa do processo e aplicando novas ideias e tecnologias. Os aperfeiçoamentos são definidos a partir da identificação dos pontos fracos e imperfeições do processo corrente e do estabelecimento das alterações necessárias para evitar a ocorrência de falhas. Análises de custo/benefício são efetuadas sobre o processo de desenvolvimento com base em dados extraídos de experiências passadas.

Quando os problemas relacionados à adoção de um dado processo de desenvolvimento não podem ser totalmente eliminados, os projetos são cuidadosamente acompanhados para evitar a ocorrência de problemas inerentes do processo. Veja a seguir as principais características deste nível.

Características do *optmizing*

Características: a melhoria contínua do processo é estabelecida por meio de sua avaliação quantitativa e da implantação planejada e controlada de tecnologias e ideias inovadoras. Projetos-piloto são realizados para a absorção e internalização de novas tecnologias. Tipicamente, um alto nível de qualidade e de satisfação dos clientes é alcançado rotineiramente, com grande foco na melhoria contínua.

Segundo Sousa (2007, p. 168):

> Deve-se lembrar de que não basta saber onde se deseja chegar; é preciso traçar o caminho que se irá trilhar para atingir o objetivo. Nesta tarefa, a metodologia CMMI também socorre, dividindo cada estágio em áreas de processo e para cada uma delas são definidos dois conjuntos de metas: as específicas e as genéricas.
>
> A essas metas, a definição do modelo recomenda práticas genéricas divididas em um conjunto de características comuns que por sua vez se divide em quatro categorias. São elas:
>
> **Comprometimento com a execução** – Agrupa práticas relacionadas à definição de políticas e responsabilidades, descrevendo ações para assegurar que o processo se estabeleça e seja duradouro;
>
> **Habilitação para execução** – Agrupa práticas contendo pré-condições para o projeto, de forma a permitir a implementação adequada do processo;
>
> **Direcionamento a implementação** – Agrupa práticas relacionadas ao gerenciamento do desempenho do processo;
>
> **Verificação da implementação** – Agrupa práticas para revisão junto à alta gerência e avaliação objetiva da conformidade com processos, procedimentos e padrões.
>
> É necessário à Empresa focar seus esforços na definição das metas específicas/genéricas para a realização do trabalho.
>
> **As metas específicas**, na maioria das vezes, estão focadas no negócio da empresa e buscam alinhar a metodologia CMMI às necessidades próprias; por sua vez as metas comuns focam em aspectos inerentes a qualquer empresa e devem ser considerados para a correta implementação da metodologia, de forma a garantir a maximização dos resultados.
>
> As categorias acima descritas deverão ser consideradas em qualquer estágio com o qual a empresa se identifique dentro da metodologia exposta. Elas buscam direcionar as ações de forma a garantir que o ciclo de evolução seja completado, possibilitando a implementação de uma evolução contínua dos processos e do produto como um todo.

> A metodologia CMMI é um guia para as pessoas de TI que já estão cansadas de agir como bombeiros, trabalhando arduamente e sem encontrar nenhum reconhecimento pelos usuários. Não é de forma alguma um processo simples de ser realizado, exige uma mudança de cultura voltada para o planejamento, a qualidade e o controle dos processos de desenvolvimento dos *softwares*.

B. Representação contínua: níveis de capacidade das áreas de processo

Nessa representação, os processos são agrupados em áreas específicas. Os níveis de capacidade sugerem uma melhoria contínua de processos em cada área da organização. As áreas de processo do CMMI no modelo de representação contínua são agrupadas nas categorias: Gerência de processos, Gerência de projeto, Engenharia e Suporte.

> [...] os perfis de capacitação representam caminhos de melhoria, indicando a evolução para cada uma das áreas. Em cada área de processo, os objetivos e as práticas são listados, seguidos por objetivos e práticas genéricos (KOSCIANSKI, 2007, p. 109).

Cada nível de capacidade é construído sobre o outro, seguindo uma ordem de melhoria de processo.

Na Tabela 1, é possível visualizar as áreas de processo na representação contínua, agrupadas em categorias:

Tabela 1 *Áreas de processo na representação contínua*

GERÊNCIA DE PROCESSOS	Foco no processo
	Definição de processos
	Treinamento
	Desempenho de processo
	Inovação e implantação
GERÊNCIA DE PROJETO	Planejamento de projeto
	Controle e monitoramento de projeto
	Gerência de acordos com fornecedores
	Gerência de projeto integrada
	Gerência de riscos
	Integração de equipe
	Integração de fornecedores
	Gerência quantitativa de projeto

ENGENHARIA	Gerência de requisitos
	Gerência de desenvolvimento
	Solução técnica
	Integração de produto
	Verificação
	Validação
SUPORTE	Gerência de configuração
	Garantia de qualidade de produto e processo
	Medida e análise
	Análise de decisão e resolução
	Ambiente organizacional para integração
	Resolução e análise de causas

Fonte: Koscianski (2007)

Na representação contínua, o modelo estabelece seis níveis de capacidade para as áreas de processo: incompleto, realizado, gerenciado, definido, gerenciado quantitativamente e otimizado, sendo os níveis:

1) Nível 0 – Incompleto: uma área de processo é considerada incompleta quando ela não é executada ou é parcialmente executada. Isto é, um ou mais objetivos específicos da área não é executado.
2) Nível 1 – Executado: Uma área de processo é executada quando o objetivo genérico e todos os objetivos específicos da área são satisfeitos.
3) Nível 2 – Gerenciado: Uma área de processo gerenciada é uma área executada que também é planejada, está de acordo com uma política, emprega pessoas com habilidades técnicas para produzir adequadamente seus produtos, envolve as pessoas que tem interesses no projeto, é monitorado, controlado e revisado.
4) Nível 3 – Definido: Uma área de processo definida é uma área gerenciada e que está de acordo um conjunto de processos padrões da organização.
5) Nível 4 – Quantitativamente Gerenciado: Uma área de processo quantitativamente gerenciada é uma área definida que é controlada utilizando estatísticas e outras técnicas quantitativas. Os objetivos quantitativos para a qualidade e desempenho do processo são estabelecidos e usados como critério no gerenciamento do processo.

6) Nível 5 – Otimizado: Uma área de processo otimizada é uma área quantitativamente gerenciada que é modificada e adaptada para atingir os objetivos de negócio. Um processo otimizado focaliza em melhorais tecnológicas incrementais e inovadoras.

O nível de capacidade da abordagem contínua focaliza na crescente habilidade da organização para desenvolver, controlar e melhorar seu desempenho na área de processo em questão. Cada área de processo possui características relativas a mais de um nível no modelo em estágio. Assim, uma área de processo que, no modelo em estágios, pertence exclusivamente ao Nível 2, no modelo contínuo pode ter características que a coloquem em outros níveis. Assim, no modelo contínuo, cada área é classificada separadamente.

Utilização do modelo CMMI

Durante o estudo desta disciplina, vimos diferentes áreas relacionadas ao projeto de desenvolvimento de *software*. Nas tabelas a seguir, procuramos relacionar essas áreas com os níveis de maturidade do modelo CMMI.

Pelas tabelas, é possível ter uma ideia sobre quais as etapas recebem destaque de padronização e maturação para conseguir a certificação em cada um dos níveis.

Vale ressaltar que a certificação, seja em qualquer um dos níveis, após ser conseguida, pode ser perdida ou a empresa pode entrar em sindicância.

Após conseguida uma certificação, de tempo em tempo, a empresa passa por novas avaliações para diagnosticar se manteve seu nível de maturidade e/ou melhorou.

Caso a empresa não mantenha seu nível de maturidade, ela poderá entrar em sindicância e, posteriormente, será decidido se os itens observados farão com que a empresa perca a certificação de determinado nível ou se será concedido um novo período de adaptação para que melhore e seja capaz de adequar “novamente” ao nível antigo/desejado.

Tabela 2 *Níveis de maturidade*.

ÁREAS-CHAVE POR NÍVEL DE MATURIDADE	
NÍVEL	ÁREAS-CHAVE
REPETÍVEL (2)	• Gerenciamento de requisitos. • Planejamento do processo de desenvolvimento. • Acompanhamento do projeto. • Gerenciamento de subcontratação. • Garantia de qualidade. • Gerenciamento de configuração.
DEFINIDO (3)	• Ênfase ao processo na organização. • Definição do processo na organização. • Programa de treinamento. • Gerenciamento integrado. • Engenharia de *software*. • Coordenação intergrupo. • Atividades de revisão.
GERENCIADO (4)	• Gerenciamento da qualidade de *software*. • Gerenciamento quantitativo do processo.
OTIMIZADO (5)	• Prevenção de falhas. • Gerenciamento de mudança de tecnologia. • Gerenciamento de mudança do processo.

Tabela 3 *Fatores comuns*.

FATORES COMUNS	
FATOR COMUM	OBJETIVOS
PASSÍVEL DE REALIZAÇÃO	Descreve as ações a realizar para definir e estabilizar um processo.
CAPACIDADE DE REALIZAÇÃO	Define pré-condições necessárias no projeto ou organização para implementar o processo de modo competente.
ATIVIDADES REALIZADAS	Descreve os procedimentos necessários para implementar uma área-chave.
MEDIDAS E ANÁLISES	Indica a necessidade de realização de atividades de medição e análise das medidas.
VERIFICAÇÃO DA IMPLEMENTAÇÃO	Corresponde aos passos necessários para assegurar que todas as tarefas foram realizadas adequadamente.

As etapas de certificação que relacionam as duas tabelas anteriores podem ser descritas **resumidamente**, veja a seguir:

1) seleção de uma equipe, cujos elementos serão treinados segundo os conceitos básicos do modelo CMMI, os quais deverão ter conhecimentos de engenharia de *software* e gerenciamento de projetos;
2) selecionar profissionais representantes da organização, para os quais será aplicado um questionário de maturidade;
3) analisar as respostas obtidas do questionário de maturidade, procurando identificar as áreas principais;
4) realizar uma visita (da equipe) à organização para a realização de entrevistas e revisões de documentação relacionadas ao processo de desenvolvimento; as entrevistas e revisões deverão ser conduzidas pelas áreas principais do CMM e pela análise do questionário de maturidade;
5) ao final da visita, a equipe elabora um relatório enumerando os pontos fortes e os fracos da organização em termos de processo de desenvolvimento de *software*;
6) finalmente, a equipe prepara um perfil de áreas principais que indica as áreas onde a organização atingiu/não atingiu os objetivos de cada área

13. QUESTÕES AUTOAVALIATIVAS

Sugerimos que você procure responder, discutir e comentar as questões a seguir:

1) (POSCOMP, 2007) Para atingir usabilidade, o projeto da interface de usuário para qualquer produto interativo, incluindo software, necessita levar em consideração um número de fatores.

   Marque, nas alternativas abaixo, o fator que NÃO deve ser considerado na análise de usabilidade de um projeto de interface de usuário.

   a) Capacidades cognitivas e motoras de pessoas em geral.
   b) Características únicas da população usuária em particular.
   c) Fatores que levem em consideração as restrições de uso de um grupo em particular não suportado pelo produto.
   d) Requisitos das atividades dos usuários que estão 'sendo suportadas pelo produto.
   e) Nenhuma das anteriores.

2) As abordagens do CMMI envolvem a:
   a) Avaliação da maturidade da informatização da organização ou a capacitação das suas áreas de projeto, o estabelecimento de requisitos e a aquisição de recursos computacionais.
   b) Implementação da maturidade da organização ou a capacitação das suas áreas de racionalização, o estabelecimento de requisitos e a modificação da estrutura.
   c) Avaliação da maturidade das interfaces da organização e a vinculação das suas áreas de processo ao estabelecimento de prioridades para a capacitação de pessoal.
   d) Avaliação da mentalidade estratégica da organização para capacitação das suas áreas de risco, estabelecimento de ações emergenciais e implementação de ações de melhoria.
   e) Avaliação da maturidade da organização ou a capacitação das suas áreas de processo, o estabelecimento de prioridades e a implementação de ações de melhoria.

Fonte: Prova aplicada em 01/2010 para o concurso do SUSEP - 2010, realizado pelo órgão/instituição Superintendência de Seguros Privados, área de atuação Controle e Gestão, organizada pela banca ESAF, para o cargo de Analista Técnico, nível superior.

Gabarito

Depois de responder às questões autoavaliativas, é importante que você confira o seu desempenho, a fim de que possa saber se é preciso retomar o estudo desta unidade. Assim, confira, a seguir, as respostas corretas para as questões autoavaliativas propostas anteriormente:

1) (c)

2) (e)

14. CONSIDERAÇÕES

Nesta unidade foi abordado apenas o modelo CMMI sugerimos que você pesquise e conheça o uso de normas ISO e outro modelo chamado SPICE.

Lembre-se de que a Engenharia de *Software* é uma disciplina fundamental para quem deseja trabalhar na área de desenvolvimento. Portanto, não deixe de pesquisar esses novos modelos. Quem sabe por meio de uma pesquisa complementar dessas não surja uma nova ideia de oportunidade profissional.

15. REFERÊNCIAS BIBLIOGRÁFICAS

CROSBY, P. B. *Qualidade é investimento*. Rio de Janeiro: Editora José Olympio, 1991.

DEMING, W. E. *Qualidade*: a revolução da administração. Rio de Janeiro: Marques-Saraiva, 1990.

JURAN, J. M. *A qualidade desde o projeto* - novos passos para o planejamento da qualidade de produtos e serviços. São Paulo: Pioneira, 1992.

KAN, S. H. *Metrics and models in software quality engineering*. Massachusetts: Addison-Wesley, 1995.

KOSCIANSKI, A.; SANTOS, M. *Qualidade de Software*. 2. ed. São Paulo: Editora Novatec, 2007.

LAMPRECH, J.L. *ISO 9000 e o setor de serviços*. Rio de Janeiro: Editora Qualitymark, 1997.

PANORAMA DO SETOR DE *SOFTWARE* NO BRASIL. Disponível em: <http://www.mct.gov.br/index.php/content/view/6358.html>. Acesso em: 2 jul. 2007.

PRESSMAN, R. S. *Engenharia de software*. 3. ed. São Paulo: Makron Books, 1996.

_____. *Engenharia de software*. 6. ed. São Paulo: McGraw-Hill, 2006.

QUALIDADE E PRODUTIVIDADE NO SETOR DE *SOFTWARE* BRASILEIRO. Disponível em: <http://www.mct.gov.br/index.php/content/view/2833.html>. Acesso em: 2 jul. 2007.

STAA, A. *Engenharia de programas*. Rio de Janeiro: LTC - Livros Técnicos, 1987.

16. CONSIDERAÇÕES FINAIS

Chegamos ao final desta disciplina e esperamos que você tenha entendido os reais objetivos da Engenharia de *Software*. Você pôde conhecer os principais conceitos do que pode ser chamada de ciência do desenvolvimento de *software*.

De forma resumida, a grande meta da Engenharia de *Software* é aumentar a qualidade de seus produtos e a produtividade de seus processos.

Há, ainda, muitas controvérsias sobre este assunto, o que é natural numa disciplina tão nova e inserida em um ambiente dinâmico que é o da indústria de *software*.

Todos os assuntos abordados (paradigmas de *software*, requisitos, documentação, qualidade, teste, manutenção, dentre outros) foram uma tentativa para conscientizá-lo sobre o quão

importante e bem estruturado deve ser o processo de desenvolvimento de *software*.

Não deixe de pesquisar mais sobre o assunto, visitar os *sites* indicados e conhecer os assuntos relacionados a esta disciplina. Enriqueça seus conhecimentos e nunca pare de estudar. Um dos grandes pré-requisitos desta nossa área é a atualização contínua.

Lembre-se de que seu crescimento pessoal e profissional está relacionado à sua capacidade e vontade de sempre ir além do conteúdo da disciplina.

Abraços a todos.

Ana Paula e Jaciara.