Sistema Aberto de Educação

SABE
Sistema Aberto de Educação

Guia de Estudo

Engenharia de Software I

unis
CENTRO UNIVERSITÁRIO DO SUL DE MINAS

Instituição Credenciada pelo MEC
Centro Universitário do Sul de Minas

CREDENCIADO PELO MEC PORTARIA 4.385/05

SABE – Sistema Aberto de Educação

Av. Cel. José Alves, 256 - Vila Pinto
Varginha - MG - 37010-540
Tele: (35) 3219-5204 - Fax - (35) 3219-5223

Instituição Credenciada pelo MEC – Portaria 4.385/05

Centro Universitário do Sul de Minas - UNIS/MG
Unidade de Gestão da Educação a Distância – GEaD

Mantida pela
Fundação de Ensino e Pesquisa do Sul de Minas - FEPESMIG

Varginha/MG

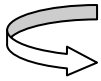
005.1
S729E SOUZA, Rafael Rodrigues de.
Guia de Estudo – Engenharia de *Software* I. Rafael Rodrigues de Souza. Varginha: GEaD-UNIS/MG, 2008.
131p.

1. *Software* 2. Análise de Sistemas 3. Ciclo de Vida do *Software* I. Título.

Autor(a)
Rafael Rodrigues de Souza – rafaelvga@yahoo.com.br

Mestre em Administração e Desenvolvimento Organizacional (2007) pela FACECA, Bacharel em Ciência da Computação (2001) pelo UNIS-MG, Técnico em Processamento de Dados (1996) pelo Colégio Batista. Atua como docente em cursos de graduação e pós-graduação no UNIS, FACECA e UNINCOR. É sócio da VSoftware, empresa especializada no desenvolvimento de *softwares* gerenciais utilizando as linguagens Visual Basic e C#.

TABELA DE ÍCONES

	**REALIZE**. Determina a existência de atividade a ser realizada. Este ícone indica que há um exercício, uma tarefa ou uma prática para ser realizada. Fique atento a ele.
	**PESQUISE**. Indica a exigência de pesquisa a ser realizada na busca por mais informação.
	**PENSE**. Indica que você deve refletir sobre o assunto abordado para responder a um questionamento.
	**CONCLUSÃO**. Todas as conclusões, sejam de idéias, partes ou unidades do curso virão precedidas desse ícone.
	**IMPORTANTE**. Aponta uma observação significativa. Pode ser encarado como um sinal de alerta que o orienta para prestar atenção à informação indicada.
	**HIPERLINK**. Indica um link (ligação), seja ele para outra página do módulo impresso ou endereço de Internet.
	**EXEMPLO.** Esse ícone será usado sempre que houver necessidade de exemplificar um caso, uma situação ou conceito que está sendo descrito ou estudado.
	**SUGESTÃO DE LEITURA**. Indica textos de referência utilizados no curso e também faz sugestões para leitura complementar.
	**APLICAÇÃO PROFISSIONAL.** Indica uma aplicação prática de uso profissional ligada ao que está sendo estudado.
	**CHECKLIST ou PROCEDIMENTO.** Indica um conjunto de ações para fins de verificação de uma rotina ou um procedimento (passo a passo) para a realização de uma tarefa.
	**SAIBA MAIS.** Apresenta informações adicionais sobre o tema abordado de forma a possibilitar a obtenção de novas informações ao que já foi referenciado.
	**REVENDO.** Indica a necessidade de rever conceitos estudados anteriormente.

SUMÁRIO

APRESENTAÇÃO

Olá!!!

Você vai usar este guia na disciplina Engenharia de Software I, do curso de Bacharelado em Sistemas de Informação pelo Centro Universitário do Sul de Minas – UNIS-MG. Nesta disciplina, você vai conhecer ferramentas, metodologias e técnicas que podem ser utilizadas na análise, documentação e no projeto de um *software*.

Você verá também que existem várias formas de se conhecer o funcionamento de uma empresa, saber qual o problema que o *software* irá resolver e projetar uma solução para atender a esta situação. No decorrer deste guia, estes métodos vão ser apresentados e faremos também estudos de caso para que consigamos aplicar o conteúdo em situações encontradas com freqüência no dia-a-dia.

Espero que você se dedique bastante, faça pesquisas, coloque suas dúvidas no fórum, participe ativamente para juntos conseguirmos chegar ao final do semestre com a sensação e a certeza de termos aprendido bastante, vai ser bem interessante, você vai ver.

Vamos trabalhar!!!

Rafael Rodrigues de Souza
Professor de Engenharia de Software I

EMENTA

Processo de desenvolvimento de *software* segundo o modelo estruturado. Ciclo de vida de desenvolvimento de *software*. Qualidade de *software*. Técnicas de planejamento e gerenciamento de *software*. Gerenciamento de configuração de *software*. Engenharia de requisitos. Métodos de análise e de projeto de *software*. Garantia de qualidade de *software*. Verificação, validação e teste. Manutenção. Documentação. Padrões de desenvolvimento. Ambientes de desenvolvimento de *software*

1 INTRODUÇÃO

Para entender o conteúdo, você precisa ter uma base teórica sobre alguns conceitos que vamos usar durante a disciplina. A maioria desses termos é muito simples, mas, para não correr o risco de algum deles atrapalhar o seu entendimento, vou apresentar a definição de cada um. Sei que você deve estar ansioso para saber o que é em si a Engenharia de Software, mas precisamos antes ter esta base teórica.

1.1 CONCEITOS BÁSICOS

Software: É o produto que os profissionais como programadores, engenheiros, analistas e outros constroem e depois mantêm ao longo do tempo. Abrange os programas de computador, a documentação e os dados usados na execução correta destes programas. Existem vários tipos de *software* que podem fazer as mais variadas tarefas.

	Software de controle administrativo, controle de estoque, editores de texto, planilhas eletrônicas, dentre outros podem ser citados como exemplos.

Sistema: É um grupo de componentes que trabalham em conjunto tentando alcançar um objetivo comum; esta definição é bem abrangente e pode ser usada para qualquer tipo de sistema. Existem outros tipos de sistema como o biológico (do corpo humano), sistema solar, sistema de informação e outros. Um sistema de informação, que é o que importa para nós, é um conjunto de programas de computador, equipamentos, pessoas, procedimentos e outros componentes que são usados para gerar informação. Não faça confusão entre *software* e sistema; um *software* faz parte de um sistema, mas ele sozinho nem sempre é um sistema,

entendeu o porquê? Eu posso ter diversos *softwares*, pessoas, equipamentos e chamar tudo isto de sistema, ok?

Programa de computador: Um programa de computador é um conjunto de instruções que descrevem uma tarefa a ser realizada por um computador. O termo pode ser uma referência ao código fonte, escrito em alguma linguagem de programação, ou ao arquivo que contém a forma executável deste código fonte. Um programa na realidade é um *software* sem sua especificação, documentação, manual e outros componentes.

Usuário: São pessoas que usam algum tipo de serviço, neste caso, serviços de informação. Os usuários em sistemas de informação são aqueles que usam a tecnologia para fazer algum trabalho.

Engenharia: É a atividade na qual os conhecimentos científicos e técnicos e a experiência prática são usados para explorar os recursos naturais, para o projeto, construção e operação de objetos úteis.

	Agora faça o seguinte, volte em cada um dos conceitos e veja se consegue me explicar o que leu, utilizando as suas palavras. Se conseguir ótimo, sinal que você não tem nenhuma dúvida; se não conseguir, use o fórum, mande-me suas dúvidas, vamos discutir o assunto.

Agora que você já tem conhecimento dos conceitos básicos, vamos entender do que trata a Engenharia de Software. É uma disciplina que se ocupa com todos os aspectos envolvidos na produção de *software*, desde os estágios iniciais, nos quais se busca conhecer o que o sistema precisa fazer (especificação), passando por sua construção até a manutenção deste sistema em funcionamento.

1.2 ENGENHARIA DE SOFTWARE

1.2.1 Conceito

Vamos agora unir os conceitos de Engenharia com o conceito de Software, os dois foram definidos no item anterior. Podemos definir então que **Engenharia de Software** é uma área da informática voltada para a especificação, desenvolvimento e manutenção de sistemas de *software* usando tecnologias e práticas de sistemas de informação, gerência de projetos e outras disciplinas, buscando organização, produtividade e qualidade.

Você consegue ver qual a diferença entre o desenvolvimento de um produto feito de forma artesanal e o desenvolvimento de um produto de acordo com a engenharia? Qual a diferença entre o trabalho de um artesão e o de um engenheiro? Qual vantagem e desvantagem em cada um?

Não responda ainda, vá ao fórum e coloque seus comentários lá, escreva o que você está pensando.

1.2.2 Objetivos

A Engenharia de Software (ES) surgiu em meados dos anos 70 na tentativa de contornar a crise do *software* (será visto depois) e dar um tratamento de engenharia, mais sistemático e controlado, ao desenvolvimento de sistemas de *softwares* complexos.

Podemos chamar de um "sistema de *software* complexo", quando ele é feito por vários componentes abstratos de *software* (estruturas de dados e algoritmos) agrupados na forma de procedimentos, funções, módulos, objetos interconectados entre si, compondo a arquitetura do *software*, que deverão ser executados em sistemas computacionais.

A ES possui diversos objetivos, vamos ver alguns deles:

- Aplicação de teoria, modelos, técnicas e ferramentas para o desenvolvimento de *software* de forma organizada.

- Aplicação de métodos, técnicas e ferramentas para gerenciar e controlar o desenvolvimento de *software*.

- Produção da documentação formal usada na comunicação entre os membros da equipe de desenvolvimento e os usuários.

- Melhorar a qualidade, aumentar a produtividade do desenvolvimento e aumentar satisfação do usuário do *software*.

Agora que você já conhece um pouco do universo que envolve a engenharia de *software,* vamos começar a aprofundar o conteúdo, conhecendo técnicas, conceitos e ferramentas importantes para atingirmos os objetivos que queremos alcançar.

2 PERSPECTIVA HISTÓRICA

Para você entender a situação atual do desenvolvimento de *software*, seus problemas e complicações, é necessário ter uma visão do que vem acontecendo desde o seu surgimento. Vamos fazer o seguinte, vou comentar quais são os principais tipos de *software*, mostrar como evoluiu e depois o que chamamos de crise do *software*.

Pode parecer bobagem, mas para entender o presente você tem que saber o que aconteceu no passado que gerou a situação atual, isto é bem importante. Domine o conteúdo todo, só assim será um profissional completo.

2.1 Tipos de *Software*

Se você procurar em livros ou até mesmo na internet vai achar várias classificações de *software*. Esta que vou usar é do Pressman, um dos papas deste conteúdo, ele diz que os *softwares* de uma forma geral podem ser classificados de acordo com seu objetivo e que, à medida que o *software* cresce em termos de complexidade e abrangência, vai ficando mais difícil de ter clareza desta divisão. Mas ainda assim, ele os agrupou em sete categorias:

***Software* de sistemas**: São programas escritos para apoiar outros programas. A área do *software* básico tem algumas características bem marcantes: forte interação com o *hardware* de computador; intenso uso por múltiplos usuários; operações simultâneas; compartilhamento de recursos e sofisticada administração da execução.

	Componentes de sistemas operacionais, software de rede, processadores de telecomunicações e acionadores de dispositivos.

***Software* de aplicação**: é um programa de computador que é feito para desempenhar tarefas práticas, em geral está ligado ao processamento de dados,

como o trabalho em escritório ou empresa. A sua natureza é então diferente da de outros tipos de *software*, como sistemas operacionais, jogos, entre outros.

	Processamento de transações em uma loja, controle de processo de fabricação de um produto, controles administrativos, financeiros e outros.

***Software* científico e de engenharia:** Tem sido caracterizado por algoritmos de processamento de grandes quantidades de números. As aplicações vão desde a astronomia até a vulcanologia da análise de fadiga mecânica de automóveis e à dinâmica orbital de naves espaciais, ou seja, trabalhos muito diferentes. Eles têm em comum a aplicação da engenharia ou a resolução de algum problema científico.

	Simulação de resistência de materiais, simulação de sistemas planetários, cálculos estruturais de engenharia, dentre outros.

***Software* embutido:** São usados para controlar produtos e sistemas para os mercados industriais e de consumo. O *software* embutido (*embedded software*) fica na memória de algum dispositivo e executa atividades limitadas e particulares.

	Funções digitais em automóveis, tais como controle, mostradores no painel, sistemas de freio; controle de teclado para fornos de microondas ou qualquer software dentro de algum equipamento.

***Software* para linhas de produtos:** Normalmente desenvolvido para ser usado por muitos clientes diferentes, o *software* para linhas de produtos pode focalizar um mercado limitado e especial ou dirigir-se ao mercado de consumo de massa.
Perceba que alguns *softwares* que aparecem na categoria Aplicação também podem aparecer em linhas de produtos, porém este segundo tenta atender a uma grande quantidade de usuários, já o primeiro um grupo de usuários específico.

	Processamento de texto, planilhas, gráficos, aplicações financeiras pessoais e empresariais.

Aplicações Web: Aplicações para web, “ApsWeb”, cobrem uma ampla gama de aplicações. Pode ser pouco mais que um conjunto de arquivos ligados por hipertexto ou ambientes complexos integrados ao banco de dados da empresa.

	Sites institucionais, e-commerce, intranet e outros.

***Software* de inteligência artificial:** Usa algoritmos não numéricos para resolver problemas complexos que não sejam facilmente resolvidos pela computação tradicional. Normalmente eles tentam simular o comportamento humano ou a inteligência para resolver alguma situação.

	Robótica, reconhecimento de voz, jogos entre outros.

O desempenho dos computadores aumenta a todo o momento, sempre temos notícia de computadores mais rápidos, não é mesmo? Isso aconteceu no passado também, diminuiu-se o tamanho dos computadores, o preço foi caindo e conseqüentemente esses fatos ajudaram a acelerar o desenvolvimento do *software*, que pode ter recursos mais avançados já que tem melhores computadores à disposição.

Você conseguiu reconhecer as divisões que existem nos *softwares* que você usa no seu dia-a-dia? Pense um pouco no que você usa e tente encaixar em uma das definições que eu passei. Ficou em dúvida? Use o fórum!!!

2.2 Evolução do *Software*

À medida que os computadores foram evoluindo, como já disse antes, os programadores passaram a ter *hardware* cada vez mais potente para usar, o *software* foi evoluindo em conjunto com os computadores até chegar à situação atual. Veja abaixo uma divisão aproximada dessa evolução:

(1950 - 1965) *Hardware*: contínuas mudanças; *software*: arte "secundária"; *hardware* de propósito geral; *software* específico para cada aplicação; não havia documentação.

(1965 - 1975) Multiprogramação e sistemas multiusuários; técnicas interativas; sistemas de tempo real; 1ª geração de SGBD's; produto de *software*; bibliotecas de *software*; crescimento de sistemas baseados em computador; manutenção quase impossível; crise de *software*.

(1975 - hoje) Sistemas distribuídos; redes locais e globais; uso generalizado de microprocessadores - produtos inteligentes; *hardware* de baixo custo; impacto de consumo; CRISE DE *SOFTWARE*.

(Quarta era do *software*: hoje, estamos na transição) Tecnologias orientadas a objetos; Sistemas especialistas e *software* de inteligência artificial usados na prática; Computação Paralela; Internet.

2.3 CARACTERÍSTICAS DO *SOFTWARE*

Para ter a compreensão do que é *software*, é importante que você conheça as características que o fazem ser diferente das outras coisas que nós construímos. Quando um equipamento é construído, o processo criativo humano (análise, projeto, construção e teste) é imediatamente traduzido numa forma física. Se construirmos um novo computador, nossos rascunhos iniciais, desenhos de projeto e protótipo

evoluem para um produto físico (chips VLSI, placas de circuito, fontes de energia etc.).

	Perceba que um elemento físico, como um computador, você pode pegar, analisar, sentir e, em pouco tempo, ter uma primeira impressão. Com um elemento lógico como é o software, isto não é fácil assim.

Veja então as principais características do *software*:

1. O *software* é desenvolvido ou projetado por engenharia, não manufaturado no sentido clássico

Existem algumas semelhanças no desenvolvimento de *software* e na produção de um equipamento, mas as duas atividades são bem diferentes. Nelas, a alta qualidade pode ser obtida se você fizer um bom projeto, mas a fase de fabricação do *hardware* pode introduzir problemas de qualidade que não existem (ou são facilmente corrigidos) para o *software*. Ambas as atividades dependem de pessoas, mas a relação entre as pessoas envolvidas e o trabalho executado é totalmente diferente. As duas exigem a construção de um "produto", no entanto a forma de se fazer cada um é muito diferente.

Os custos do *software* estão concentrados no trabalho de engenharia. Isso significa que os projetos de *software* não podem ser feitos como se fossem projetos de manufatura.

Nos anos 90, o conceito de "fábrica de *software*" surgiu e foi discutido na literatura. Torna-se importante observar que este termo não quer dizer que a manufatura de *hardware* e o desenvolvimento de *software* sejam iguais. Ao contrário, o conceito de fábrica de *software* recomenda o uso de ferramentas automatizadas no desenvolvimento de *software*.

2. O *software* não se "desgasta"

Para que fique claro o que quero dizer com relação a não se desgastar, vamos continuar fazendo o nosso comparativo entre o *hardware* e o *software*.

Veja a Figura 1, ela mostra o índice de falhas em um equipamento como uma função do tempo para o *hardware*. A relação, muitas vezes chamada "curva da banheira", indica que o *hardware* exibe índices de falhas elevados logo no começo de seu ciclo de vida (essas falhas freqüentemente são atribuídas a defeitos de projeto e manufatura); os defeitos são corrigidos e o índice de falhas cai para um nível estável (esperançosamente, muito baixo) durante certo período de tempo. O tempo passa, e o índice de falhas sobe novamente à medida que os componentes de *hardware* sofrem os efeitos de poeira, vibração, temperaturas extremas e muitos outros males ambientais.
Colocado de maneira simples, o *hardware* começa a se desgastar com o passar do tempo.

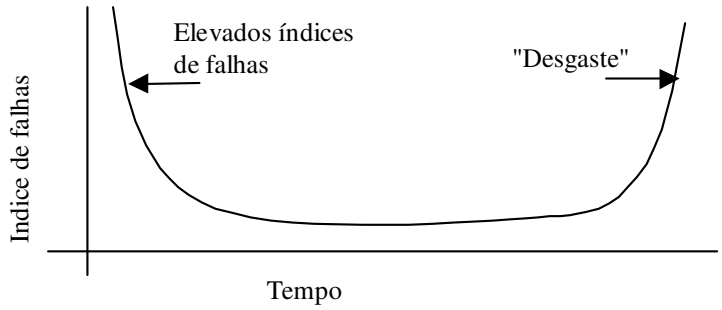


Figura 1: Curvas de Falhas Para o *Hardware*

O *software* é totalmente diferente, por não ter componentes físicos, ele não se desgasta. O que deveria acontecer na teoria é o que está representado na Figura 2, à medida que o tempo vai passando, os erros são corrigidos e o número de problemas vai diminuindo, parece bem lógico, não é? Só que não é isso que realmente acontece, o *software* não se desgasta, mas se deteriora!

Durante a sua vida, o *software* passará por modificações. Enquanto as modificações são feitas, é provável que alguns novos defeitos sejam introduzidos, causando um dente na curva de falhas. Antes que a curva possa voltar ao seu estado original da taxa de falhas estável, outra modificação é solicitada, causando um novo dente na curva. A taxa de falhas mínima começa a subir lentamente – o *software* está se deteriorando devido a modificações, fazendo com que a curva do índice de falhas seja parecida com a mostrada na Figura 3:

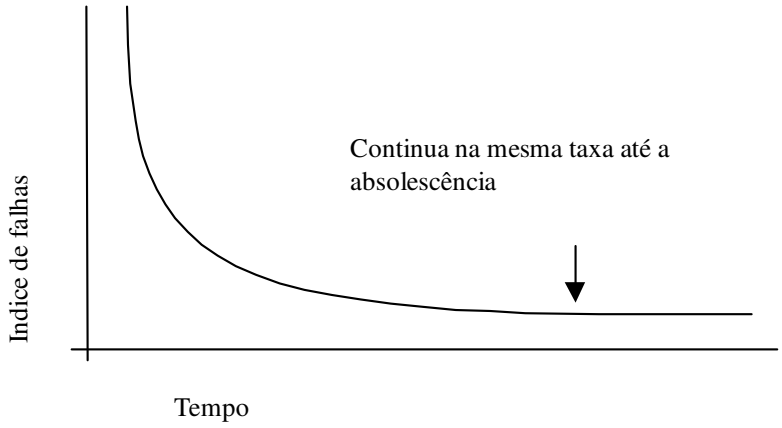


Figura 2: Curva de Falhas do Software (Idealizada)

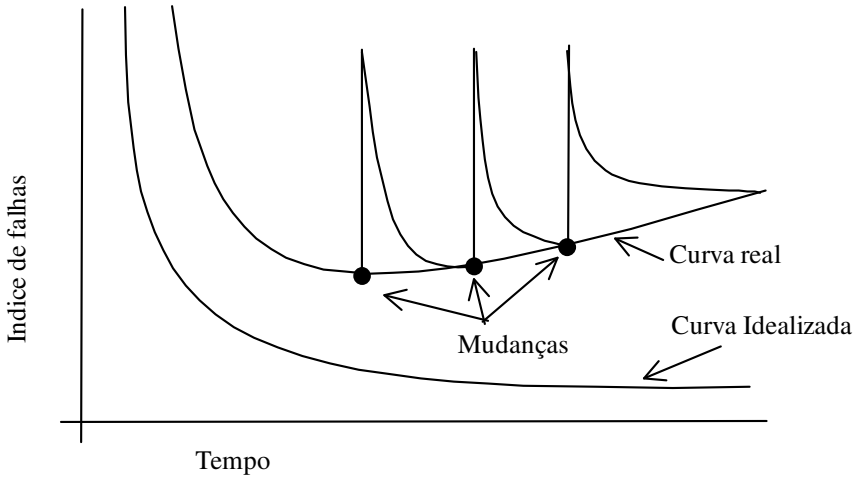


Figura 3: Curva de Falhas Real Para o *Software*

Outra característica do uso mostra a diferença entre o *hardware* e o *software*. Quando se desgasta, um componente de *hardware* é substituído por uma "peça de reposição. Não existem peças de reposição para o *software*. Toda falha deste indica um erro de projeto ou na forma como ele foi programado. Portanto a sua manutenção envolve consideravelmente mais complexidade do que a manutenção de *hardware*.

3. A maioria dos *softwares* é feito sob medida em vez de ser montado a partir de componentes existentes

Vamos considerar a montagem de um computador, o técnico escolhe os componentes para montar o equipamento de que precisa, junta esses componentes que já foram construídos anteriormente e pronto.
Infelizmente, os projetistas de *software* não têm como fazer o que acabei de descrever. Com poucas exceções, não existem catálogos de componentes de *software*. Muita coisa já foi escrita sobre reusabilidade de *software*, mas só agora começam a aparecer as primeiras tentativas bem sucedidas, vamos torcer para que no futuro a junção de vários componentes prontos seja possível na construção de um novo *software*. Seria uma maravilha.

Conclusão	O desafio dos engenheiros de software é permitir que a criação saia de um processo artesanal e chegue o mais próximo possível de um processo de manufatura, do qual se pode ter controle, com o qual se pode realizar testes confiáveis e chegar a um produto de qualidade. No entanto, não se pode comparar a produção de um software com a produção de um equipamento qualquer. Embora existam algumas semelhanças, são totalmente diferentes e deve-se respeitar essas particularidades.

2.4 Mitos de *Software*

Existem muitos conceitos errôneos, que geram desinformação e confusão, idealizados pelos envolvidos no desenvolvimento de *software*. Estes são os mitos:

MITOS DA GERÊNCIA: Os gerentes com responsabilidade sobre o assunto estão freqüentemente sob pressão para obedecer a orçamentos, manter cronogramas no prazo e melhorar a qualidade. Um gerente freqüentemente se agarra na credibilidade de um mito de *software*, se isso puder diminuir a pressão.

MITO: Um livro que está cheio de padrões e procedimentos já não fornece ao meu pessoal tudo o que ele precisa saber?
REALIDADE: O livro de padrões pode muito bem existir, mas será que ele é usado? Os profissionais sabem de sua existência: Ele reflete as modernas práticas de Engenharia de Software? É completo? Está voltado para melhorar o prazo de entrega, mantendo o foco na qualidade? Em muitos casos a resposta é não.

MITO: Meu pessoal tem ferramentas de desenvolvimento de *software* que estão no estado-da-arte, afinal, compramos para eles os computadores mais novos.

REALIDADE: É necessário muito mais do que o último modelo de computador de grande porte para fazer o desenvolvimento de *software* de alta qualidade. Ferramentas CASE são mais importantes que o HW para conseguir boa qualidade e produtividade, todavia a maioria dos desenvolvedores ainda não as utilizam efetivamente.

MITO: Se nos atrasarmos no planejamento, podemos adicionar mais programadores e ficar em dia.

REALIDADE: O desenvolvimento de *software* não é um processo mecanizado como o de fabricação. Adicionar pessoas a um projeto de *software* atrasa-o ainda mais, pois precisa gastar tempo orientando os recém-chegados. Pessoas podem ser adicionadas, mas de forma planejada e bem coordenada.

MITO: Se eu decidir terceirizar um projeto de *software,* vou poder relaxar e deixar que aquela firma o elabore.

REALIDADE: Se uma organização não sabe como gerir e controlar projetos de *software* inteiramente, certamente vai ter problemas quando terceirizar esses projetos.

MITOS DO CLIENTE: Em muitos casos, o cliente acredita em mitos de *software* porque os gerentes e profissionais de *software* fazem pouco para corrigir essa desinformação. Mitos levam a falsas expectativas e, em última análise, à insatisfação com o desenvolvedor.

MITO: O estabelecimento geral de objetivos é suficiente para iniciar a escrita de programas – podemos fornecer os detalhes posteriormente.

REALIDADE: Uma definição inicial malfeita é a principal causa de esforços malsucedidos de *software*. Uma descrição formal e detalhada do domínio da informação, da função, do comportamento, do desempenho, das interfaces, das restrições de projeto e dos critérios de validação é essencial. Essas características podem ser determinadas somente depois de intensa comunicação entre o cliente e o desenvolvedor.

MITO: Os requisitos de projeto mudam continuamente, mas as mudanças podem ser facilmente acomodadas porque o *software* é flexível.

REALIDADE: É verdade que os requisitos de *software* mudam, mas o impacto da mudança varia com a época em que é introduzida. Se uma atenção séria for dada à definição inicial, solicitações de mudanças que ocorrem no início do desenvolvimento podem ser acomodadas facilmente. Quando mudanças são solicitadas durante o projeto de *software*, o impacto no custo cresce rapidamente.

Mudanças na função, no desempenho, na interface ou em outra característica durante a implementação (código e teste) têm um impacto significativo no custo. Mudanças solicitadas depois de o *software* entrar em produção podem ficar bem mais caras do que se fossem solicitadas antes.

MITOS DO PROFISSIONAL:

MITO: Quando escrevemos um programa e o fazemos funcionar, nosso trabalho está completo.

REALIDADE: Alguém um dia disse "*quanto mais cedo você começar a escrever código, mais vai demorar para acabar*". Entre 60 e 80% de todo o esforço despendido em *software* vai ser despendido depois de ele ser entregue ao cliente pela primeira vez.

MITO: Até que eu esteja com o programa "rodando" não tenho como avaliar sua qualidade.

REALIDADE: Um dos mecanismos mais eficazes de garantia de qualidade de *software* pode ser aplicado a partir do início de um projeto – a revisão técnica formal. Revisões de *software* são como um "filtro de qualidade" que se descobriu ser mais eficaz do que testes para encontrar alguns tipos de defeitos de *software*.

MITO: O único produto de trabalho que pode ser entregue para um projeto de *software* bem sucedido é o programa executável.

REALIDADE: Um programa executável é apenas uma parte de uma configuração de *software* que inclui vários elementos. A documentação fornece a base para uma engenharia bem-sucedida e, mais importante, orienta para o suporte ao *software*.

MITO: A engenharia de *software* vai nos fazer criar documentação volumosa e desnecessária que certamente nos atrasará.

REALIDADE: A engenharia de *software* não se relaciona à criação de documentos. **Refere-se à criação de qualidade**. Melhor qualidade leva à redução de trabalho. E menor retrabalho em tempos de entrega mais rápidos.

2.5 Crise de *Software*

O desenvolvimento de *software* é uma tarefa extremamente complexa, entender bem a situação à que se destina, fazer um bom levantamento das necessidades, detectar as falhas nos procedimentos, enfim, todas as tarefas que compõem a Engenharia de Software é alvo de estudos há vários anos.

Apesar do esforço existente para se obter produtos com qualidade e em grande número, não é exatamente isso que acontece. O conjunto de problemas que ocorre gerou o que chamamos de Crise de *Software*. Vejamos quais são os principais fatores:

	Quantas vezes você já ouviu pessoas reclamando de falhas e problemas em *softwares*? Quantas vezes você já precisou de algum serviço, mas não obteve porque o sistema estava "fora do ar"?

1) Estimativas de prazo e de custo freqüentemente são imprecisas.
"Não dedicamos tempo para coletar dados sobre o processo de desenvolvimento de *software*".

Esse problema tem duas principais causas: na primeira, a empresa opta por trocar seu *software* ou por começar o uso de um, caso ainda não tenha. Quando isso ocorre, ela precisa agir com rapidez, sendo assim, pressiona para que ocorra em tempo e custo recorde. Cabe, neste caso ao responsável pelo projeto, negociar um prazo e custo viável para todos os envolvidos, tentando evitar esta imprecisão.

O segundo principal problema ocorre por falta de conhecimento ou omissão do responsável pelo levantamento do *software*. Normalmente a complexidade é maior do que o que foi avaliado ou possui mais detalhes do que foi detectado, gerando assim atraso no prazo de entrega e conseqüentemente fazendo com que o custo se eleve.

2) A produtividade das pessoas da área de *software* não tem acompanhado a demanda por seus serviços.
A todo momento, demanda por novas funcionalidades são geradas, o que ocorre com muita freqüência é que para se implementar qualquer solicitação gasta-se muito tempo. Por este motivo, as linguagens de programação estão tendo muita atenção por parte dos fabricantes para que se consiga desenvolver *softwares* no menor tempo possível.

Outra situação é que ocorre muito retrabalho na codificação do programa, grande parte gerada por análise imprecisa ou incorreta, acarretando assim uma sobrecarga de trabalho.

3) A qualidade de *software* às vezes é menos que adequada. Só recentemente começam a surgir conceitos quantitativos sólidos de garantia de qualidade de *software*.

4) O *software* existente é muito difícil de manter.
Manutenção devora o orçamento destinado ao *software*. A facilidade de manutenção não foi enfatizada como um critério importante.

Muito bem, neste momento você tem que estar bem tranqüilo com relação ao entendimento do que é um *software*, o que faz a Engenharia de Software e tudo mais que escrevi aqui. Se algo não ficou claro, use o fórum, mande e-mail, comunique-se de qualquer forma, mas não fique com dúvida, ok?

3 Ciclos de Vida de *Software*

O *software*, como qualquer produto, segue uma seqüência lógica de passos até ficar pronto, a diferença é que existem várias abordagens que podemos usar para ir desde a análise do problema até o resultado final. Ciclo de vida é o nome dado para este passo a passo feito na produção de um sistema. Existem diversos, cada um se encaixa em uma situação diferente e só o conhecimento de todos pode ajudar você a decidir qual é o melhor. Vamos ver quais são os principais:

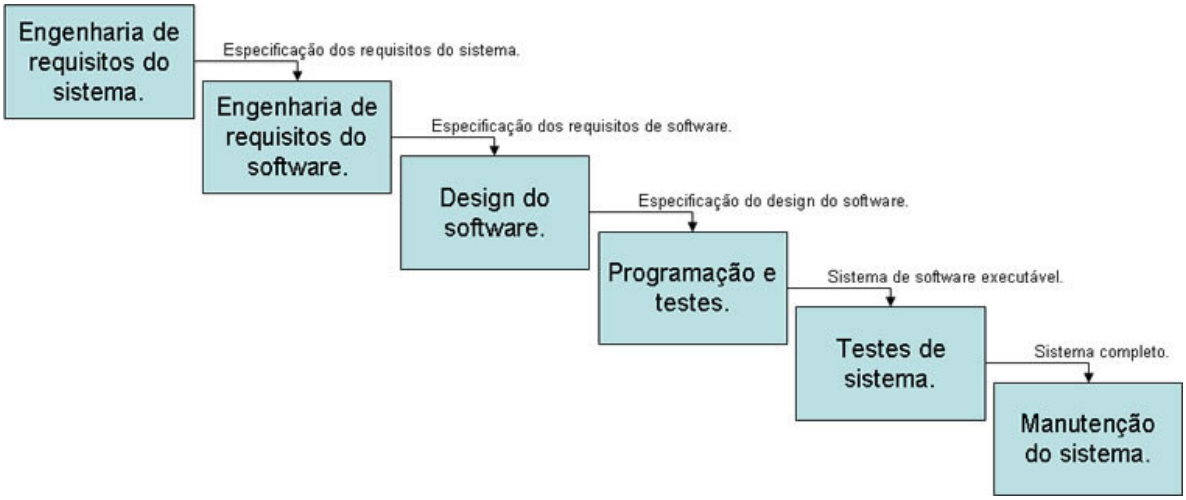
3.1 Ciclo de Vida Clássico (Cascata)

É o modelo mais antigo e o mais amplamente usado da engenharia de *software*. É um método seqüencial, em que o resultado de uma fase vai ser a entrada para a outra fase.

O modelo também é conhecido como Cascata e abordagem “top-down”, foi proposto por Royce em 1970. Até meados da década de 1980, foi o único modelo com aceitação geral.

Foi modelado de acordo com o ciclo da engenharia convencional e abrange as seguintes fases:

- Levantamento de Requisitos
- Análise de Requisitos
- Projeto
- Implementação
- Testes
- Manutenção



Análise e definição dos requisitos

Nesta etapa, são estabelecidos os requisitos do produto que se deseja desenvolver, o que consiste normalmente dos serviços que se devem fornecer, das limitações e dos objetivos do *software*. Sendo isso estabelecido, os requisitos devem ser definidos de uma maneira apropriada para que sejam úteis na etapa seguinte.

Esta etapa inclui também a documentação e o estudo de viabilidade do projeto com o fim de determinar o processo de início de desenvolvimento do sistema; pode ser vista como uma concepção de um produto de *software* e também como o início do seu ciclo de vida.

Projeto do sistema

O projeto do sistema é um processo de vários passos que se centraliza em quatro atributos diferentes do sistema: estrutura de dados, arquitetura do *software*, detalhes dos procedimentos e detalhamento das interfaces. O projeto representa os requisitos de uma forma que permite a codificação do produto (é uma prévia etapa de codificação). Da mesma maneira que a análise dos requisitos, o projeto é documentado e transforma-se em uma parte do *software*.

Implementação

Esta é a etapa em que são criados os programas. Se o projeto possui um nível de detalhe elevado, a etapa de codificação pode ser facilmente feita. A princípio, sugere-se incluir um teste unitário dos módulos nesta etapa; nesse caso, as unidades de código produzidas são testadas individualmente antes de passar à etapa de integração e ao teste global (que veremos do que trata mais adiante).

Teste do sistema

Concluída a codificação, começa a fase de teste do sistema. O processo de teste centraliza-se em dois pontos principais: as lógicas internas do *software* e as funcionalidades externas. Esta fase decide se foram solucionados erros de "comportamento" do *software* e assegura que as entradas definidas produzam resultados reais que coincidam com os requisitos especificados.

Manutenção

A fase de manutenção é o momento quando alterações ou correções são feitas no sistema; sua principal característica é estar presente durante todo o tempo de existência do sistema, ou seja, não tem fim. A manutenção pode ser:

- Corretivas: Erros podem ser encontrados durante o funcionamento do sistema.

- Adaptativa: O usuário pode propor alguma mudança a fim de acomodar mudanças de procedimentos, como, por exemplo, nova versão do sistema operacional etc.

- Preventiva: Modificação do sistema em função de mudanças necessárias para manter sua confiabilidade e eficiência, como, por exemplo, reorganizar o banco de dados etc.

Melhorias e correções podem ser consideradas como parte do desenvolvimento.

As etapas descritas são as principais, porém existem subetapas dentro de cada etapa, que podem ser bem diferentes de um projeto para outro. Também é possível que certos projetos de *software* exijam a incorporação de uma etapa extra ou a separação de uma etapa em outras etapas.

Com certeza, todas essas variações do modelo Cascata possuem o mesmo conceito básico: a idéia de que uma etapa fornece saída que serão usadas como entradas para a etapa seguinte. Portanto, o processo de desenvolvimento de acordo com o modelo Cascata é simples de conhecer e controlar.

Desvantagens do Ciclo de Vida Clássico
- Execução Seqüencial das fases
- Implementação de Unidades
- Testes de unidades (abordagem *bottom-up*)
- Um erro grave pode ser detectado apenas no final da construção do sistema.

Atividades de Proteção: as fases e etapas descritas são complementadas por uma série de atividades de proteção que ajudam chegar ao final do projeto como um todo obtendo sucesso, são elas:

Revisões: são feitas para garantir que a qualidade seja mantida à medida que cada etapa é concluída.

Documentação: é desenvolvida e controlada para garantir que informações completas sobre o *software* estejam disponíveis para uso quando for preciso.

Controle das Mudanças: é o controle do que vai ser alterado, garantindo que as mudanças vão ser aprovadas e acompanhadas, ou seja, nenhuma mudança deve ser feita sem documentar, o *software* e a documentação têm que estar em sincronia.

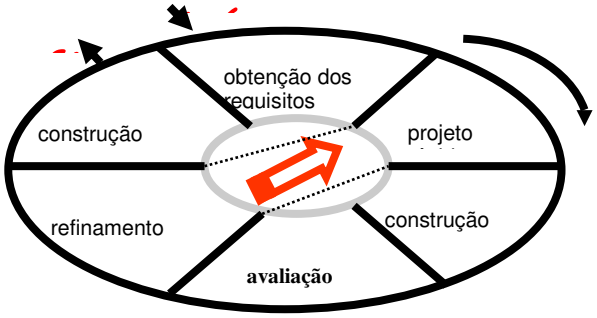
Conclusão	O modelo clássico é simples e a seqüência dele é quase que natural, por este motivo ele é o mais usado. Embora possua fragilidades, ele é significativamente melhor do que uma abordagem casual ao desenvolvimento de *software.*

3.2 Prototipação

Muitas vezes, a troca de informações entre o cliente e os desenvolvedores não é suficiente, isto geralmente acarreta em erros e deficiências no projeto de *software* e na insatisfação dos clientes. Para que isso não aconteça, podemos criar um protótipo, que tem como objetivo prevenir esses problemas.

Protótipo é a primeira versão desenvolvida do *software*, tem como finalidade avaliar a questão da interface com o usuário, validar requisitos e apresentar a viabilidade do sistema.

Durante a criação do protótipo, clientes e desenvolvedores ficam em constante interação, facilitando o levantamento de requisitos e funcionalidades do sistema.



Um dos usos mais freqüentes do protótipo é quando o cliente definiu um conjunto de objetivos gerais para o *software*, mas não identificou requisitos de entrada, processamento e saída com detalhes.

Atividades da Prototipação

1. Obtenção dos Requisitos: desenvolvedor e cliente definem os objetivos gerais do *software*, identificam quais requisitos são conhecidos e as áreas que necessitam de definições adicionais.

2. Projeto Rápido: representação dos aspectos do *software* que são visíveis ao usuário (abordagens de entrada e formatos de saída).

3. Construção Protótipo: implementação do projeto rápido usando qualquer ferramenta ou linguagem de programação que permita fazer um protótipo sem gastar muito tempo.

4. Avaliação do Protótipo: cliente e desenvolvedor avaliam o protótipo.

5. Refinamento dos Requisitos: cliente e desenvolvedor refinam os requisitos do *software* a ser desenvolvido.

Ocorre um processo de iteração que pode levar à primeira atividade até que as necessidades do cliente sejam satisfeitas e o desenvolvedor compreenda o que precisa ser feito.

Prototipagem evolutiva

Recentemente, a separação entre o desenvolvimento do protótipo e o desenvolvimento do sistema propriamente deu origem a prototipagem evolutiva.

Na prototipagem evolutiva é desenvolvido rapidamente um protótipo, que vai sendo modificado de acordo com comentários dos envolvidos (*stakeholders*) até se chegar ao sistema final.

O desenvolvimento do protótipo deve seguir os mesmos critérios de qualidade de qualquer outro sistema. Para que a evolução do protótipo para o sistema final pelo processo de prototipagem evolutiva chegue no final e seja um sistema robusto, é necessário um planeamento e um desenvolvimento cuidadoso desde o início.

Os processos de especificação, *design* e implementação são intercalados de forma repetida e o sistema vai sendo assim desenvolvido de um forma incremental.

São utilizadas técnicas de rápido desenvolvimento de *software*.

Em sistemas de grande dimensão, podem ocorrer problemas no âmbito da gestão do desenvolvimento e contratual. Se os sistemas evoluem muito rapidamente, pode originar que não seja produzida a documentação devida. Os contratos entre a entidade que desenvolverá o sistema e os clientes normalmente têm o documento de requisitos por base. Visto o documento de requisitos não existir ou não ser detalhado torna-se difícil elaborar o contrato.

Resumindo, a prototipagem evolutiva permite o desenvolvimento de pequenos e médios sistemas rapidamente, que se adaptam às necessidades do cliente. Essa forma de prototipagem é comum hoje em dia no desenvolvimento de aplicações web.

Alguns problemas com a prototipação: o cliente não sabe que, durante o desenvolvimento do protótipo de *software* que ele vê, não foram consideradas a qualidade na programação e a facilidade de manutenção (manutenibilidade) a longo prazo. Não aceita bem a idéia de que a versão final do *software* ainda vai ser construída, e tenta "forçar" a utilização do protótipo como produto final.

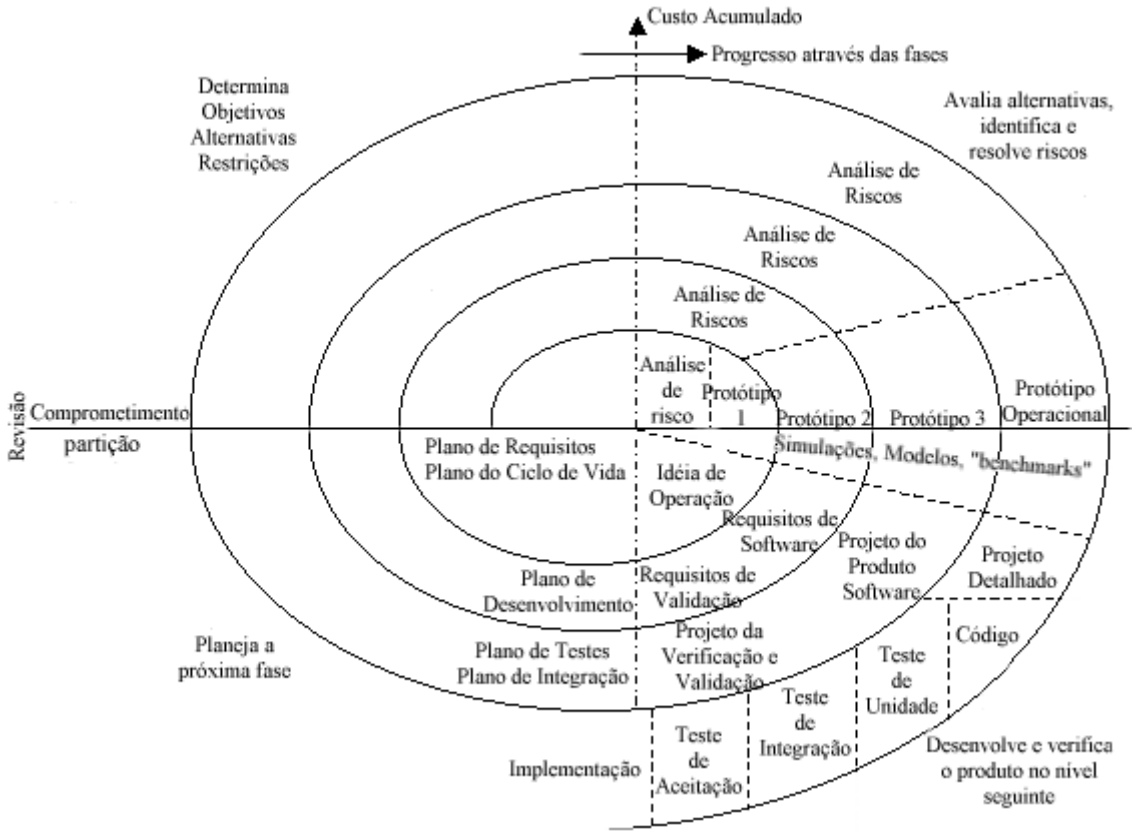
Conclusão	Ainda que possam ocorrer problemas, a prototipação é um ciclo de vida eficiente. A chave é definirem-se as regras do jogo logo no começo; o cliente e o desenvolvedor devem concordar que o protótipo vai ser construído para servir como um mecanismo para ajudar a definir os requisitos e não será o produto final.

3.3 Espiral

Este modelo foi originalmente proposto por Boehm em 1988. Uma maneira simplista de analisar este modelo é considerá-lo como um modelo cascata no qual antes de cada fase é feita uma análise de risco e sua execução é feita evolucionariamente (ou incrementalmente).

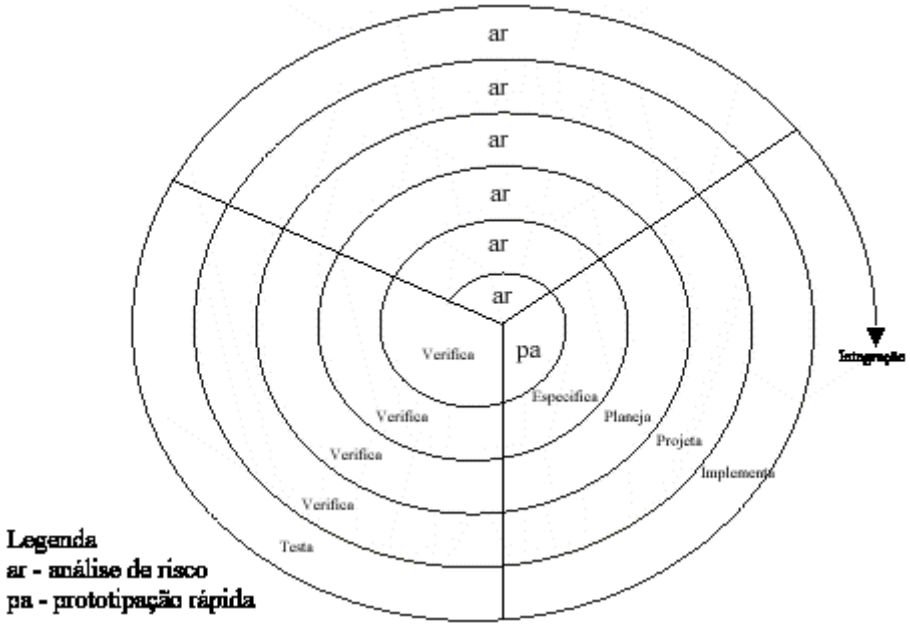
A dimensão radial representa o custo acumulado atualizado, e a dimensão angular representa o progresso por meio da espiral. Cada setor da espiral corresponde a uma tarefa (fase) do desenvolvimento. No modelo original, foram propostas quatro tarefas (fases ou quadrantes).
Um ciclo começa com a determinação de objetivos, alternativas e restrições (primeira tarefa) quando ocorre o comprometimento dos envolvidos e o estabelecimento de uma estratégia para alcançar os objetivos.
Na segunda tarefa, "Avaliação de alternativas, identificação e solução de riscos", executa-se uma análise de risco. Prototipação é uma boa ferramenta para tratar riscos. Se o risco for considerado inaceitável, o projeto pode ser encerrado. Na terceira tarefa, acontece o desenvolvimento do produto. Nesse quadrante, pode-se

considerar o modelo cascata. Na quarta tarefa, o produto é avaliado e se prepara para iniciar um novo ciclo.



Modelo Espiral Completo

A manutenção de um *software* utilizando este modelo de ciclo de vida é tratada da mesma forma que o desenvolvimento.



Modelo simplificado de Boehm

Variações do modelo espiral consideram entre três e seis tarefas ou setores da espiral. Um exemplo são as regiões:

- comunicação com o cliente;
- planejamento;
- análise de risco;
- engenharia;
- construção e liberação e
- avaliação do cliente.

3.4 Mudança na natureza de desenvolvimento de *software*

Todo processo de desenvolvimento de *software* contém 3 fases genéricas, independentes do modelo de engenharia de *software* escolhido:

1. DEFINIÇÃO, 2. DESENVOLVIMENTO e 3. MANUTENÇÃO.

DEFINIÇÃO: "*o que"* será desenvolvido.

Análise do Sistema: define o papel de cada elemento num sistema baseado em computador, atribuindo em última análise, o papel que o *software* desempenhará.
Planejamento do Projeto de Software: assim que o escopo do *software* é estabelecido, os riscos são analisados, os recursos são alocados, os custos são estimados e, tarefas e programação de trabalho definidas.

DESENVOLVIMENTO: "como" o *software* vai ser desenvolvido.
Projeto de *Software*: traduz os requisitos do *software* num conjunto de representações (algumas gráficas, outras tabulares ou baseadas em linguagem) que descrevem a estrutura de dados, a arquitetura do *software*, os procedimentos algorítmicos e as características de interface.

MANUTENÇÃO: concentra-se nas "mudanças" que ocorrerão depois que o *software* for liberado para uso operacional.

CORREÇÃO: mesmo com as melhores atividades de garantia de qualidade de *software*, é provável que o cliente descubra defeitos no *software*. A manutenção *corretiva* muda o *software* para corrigir defeitos.

Adaptação: com o passar do tempo, o ambiente original (por exemplo, a CPU, o sistema operacional e os periféricos) para o qual o *software* foi desenvolvido provavelmente mudará. A manutenção adaptativa muda o *software* para acomodar mudanças em seu ambiente.

Melhoramento Funcional: à medida que o *software* é usado, o cliente/usuário reconhecerá funções adicionais que oferecerão benefícios.

Atividades de Proteção: as fases e etapas descritas devem ser complementadas por uma série de atividades de proteção que irão ajudar a melhorar a qualidade do *software* como um todo.

Revisões: efetuadas para garantir que a qualidade seja mantida à medida que cada etapa é concluída.

Documentação: desenvolvida e controlada para garantir que informações completas sobre o *software* estejam disponíveis para uso posterior.

Controle das Mudanças: é instituído de forma que as mudanças possam ser aprovadas e acompanhadas.

Conclusão	ENGENHARIA DE SOFTWARE pode ser vista como uma abordagem de desenvolvimento de *software* elaborada com disciplina e métodos bem definidos.

4 ANÁLISE ESTRUTURADA

4.1 Introdução

As técnicas estruturadas surgiram no sentido *bottom-up*, isto é, da programação para a análise.

A programação estruturada, introduzida no final da década de 60, teve como principal propósito a construção de programas mais fáceis de serem lidos e compreendidos. Isso foi conseguido com o abandono do comando "GOTO" e uso, na construção dos programas, de apenas três estruturas básicas de controle (seqüência, seleção e repetição).

O projeto estruturado, introduzido em meados da década de 70, preocupou-se com a organização dos programas. As recomendações do projeto estruturado para organizar um sistema em módulos de tamanho reduzido, cada qual executando uma única função específica, e a maneira como os módulos devem ser interligados, contribuíram para a manutenibilidade do sistema.

No final da década de 70, a análise estruturada possibilitou especificar os requisitos lógicos do sistema em um modelo gráfico de alto nível, capaz de ser compreendido pelos usuários e de ser mapeado para a arquitetura do projeto. O modelo gráfico introduzido pela análise estruturada representa os dados utilizados por um sistema, os fluxos que transportam e os processos que os transformam.

4.2 Características da Metodologia

A análise estruturada possui as seguintes características:

- utiliza linguagem gráfica com suporte de texto;
- fornece uma visão *top-down* e particionada do sistema;
- possibilita eliminar redundâncias e
- os instrumentos conseguem ser transparentes ao leitor.

4.3 Princípios Utilizados na Solução de Problemas

Os seguintes princípios, quando utilizados, auxiliam o analista de sistemas a lidar com suas limitações, na solução de problemas:

Abstração: utilizado para separar aspectos que estão ligados a uma determinada particularidade da realidade. Possibilita a construção de modelos genéricos e simplificados do mundo real.

- **Rigor e Formalidade:** fornece uma abordagem metódica e rigorosa para resolver um problema, ao contrário de uma abordagem *ad-hoc*, que não permite o estabelecimento de controles.
- **Dividir-para-Conquistar:** um problema grande e complexo pode ser dividido em um conjunto de problemas menores e independentes, mais fáceis de serem compreendidos e resolvidos.
- **Organização Hierárquica:** os componentes da solução de um problema podem ser organizados em uma estrutura hierárquica. Assim, a solução de um problema pode ser compreendida e construída nível a nível. A cada nível são acrescentados mais detalhes.

4.4 Modelos da Metodologia

A análise estruturada utiliza os seguintes modelos para especificar os requisitos lógicos do sistema:

- **Diagrama de Fluxo de Dados (DFD):** representa um sistema de informações como uma rede de processos, interligados entre si por fluxos e depósitos de dados.
- **Dicionário de Dados (DD):** contém a definição dos dados utilizados no DFD.
- **Especificação da Lógica dos Processos:** especifica a lógica dos processos representados no DFD.

5 ENGENHARIA DE REQUISITOS

É o processo que envolve todas as atividades exigidas para criar e manter o documento de requisitos de sistema, ou seja, de descobrir, analisar, documentar e verificar as funções e restrições do sistema a ser criado.
O processo de engenharia de requisitos é composto por quatro atividades de alto nível, sendo elas: Identificação, Análise e negociação, Especificação/ documentação e Validação.

Imagine o seguinte, seu chefe lhe chamou na sala dele e pediu para que você conversasse com o almoxarife para saber como ele trabalha e, depois disto, fizesse um documento detalhando para o programador um *software* para controlar todo o almoxarifado. E agora??? O que fazer???
É exatamente isso que vou lhe mostrar, como começar a documentar um *software*, por onde começar, quais são os envolvidos e tudo mais, mãos à obra!!!

Requisito

A definição de Requisito depende muito do contexto no qual a palavra é usada, por isto vou dar duas definições, uma do dicionário Aurélio, que é uma definição mais ampla e outra do IEEE, um órgão que determina regras e padrões sobre assuntos relacionados à engenharia elétrica e informática:

- **Requisito (Aurélio)**
  - Condição necessária para a obtenção de certo objetivo, ou para o preenchimento de certo fim.
- **Requisito (IEEE)**
  - Uma condição ou capacidade necessitada por um usuário para resolver um problema ou alcançar um objetivo.
  - Uma condição ou capacidade que deve ser satisfeita por um sistema para satisfazer um contrato ou um padrão.

Então, de acordo com as definições, podemos dizer que requisitos são descrições de como o sistema deverá se comportar, condições sobre operação de sistema ou especificações de uma propriedade ou atributo de sistema.

	IEEE - Institute of Electrical and Electronic Engineers http://www.ieee.org IEEE – Conselho Brasil http://www.ieee.org.br/ IEEE – Software Engineering Standards Zone http://standards.ieee.org/software/

5.1 Levantamento de requisitos

O "usuário"

Nesse exemplo que dei acima, você já viu de quem depende uma boa parte do *software*. Sim, dele mesmo, o usuário, aquele que vai usar o *software* depois de pronto, lançando informações e tirando resultados. Por que ele é tão importante? Porque ele é quem vai lhe passar todas as informações a respeito de como deve funcionar o *software*.

Perceba que uma das diversas atividades do analista é manter um diálogo constante com o usuário. É importante falar a língua dele, fazer com que ele se sinta igual a você, não fique acanhado de conversar, para isto é importante conhecê-lo. É necessário saber exatamente com que tipo de usuário ou usuários você está lidando, isso mesmo, eles também são classificados. Por que saber o tipo do usuário? Para que você se comporte de forma que consiga ganhar sua confiança, não seja mal interpretado e se prepare melhor para conversar com ele.

Eles podem ser classificados de três maneiras:

- pelas funções na organização;
- por nível de experiência no uso do computador e

- por características de personalidade.

Os usuários classificados pelas funções na organização podem ser agrupados em:

- Operativos: normalmente têm visão local, são os executores das funções no sistema e têm visão física do sistema.

- Supervisores: podem ou não ter visão local, normalmente conhecem a operação e, muitas vezes, agem como intermediários entre os usuários e os níveis mais elevados da direção.

- Executivos: têm visão global, não têm muita experiência operativa e possuem preocupações estratégicas.

Os usuários classificados por nível de experiência no uso do computador podem ser agrupados em:

- Amadores: extremamente leigos em computadores.

- Novatos: participaram (ou participam) de maneira mínima de algum trabalho ou até mesmo de um projeto maior que envolva o computador e consideram-se entendidos no assunto.

- Conhecedores: Possuem experiência com outros *softwares*, sabem dimensionar as necessidades por meio da utilização de *hardware* e *software* e podem ser um bom ponto de apoio no momento de entender o que deve ser feito.

Os usuários classificados por características de personalidade podem ser divididos em:

- Conservadores: costumam não expor as idéias ou informações sobre a empresa de maneira clara, ou por desinteresse ou até mesmo por não concordar com as inovações que irão acontecer.

- Indiferentes: não aprovam ou reprovam a mudança que está acontecendo, mas, muitas vezes, prejudicam o levantamento de dados por falta de iniciativa ou motivação.

- Extrovertidos: normalmente muito comunicativos, mas, muitas vezes, interferem no trabalho do analista, ou ainda dão uma atenção aos assuntos menos relevantes.

Como você viu, existem vários tipos de usuários, e a percepção que se tem deles pode lhe ajudar na hora de filtrar o que é importante e o que não é, por exemplo, um usuário extrovertido pode falar sobre itens do sistema que não tem tanta importância, já o usuário conservador pode não passar todas as informações necessárias.
Enfim, essa classificação é para que você possa observar o tipo de usuário e interagir com ele da forma mais adequada.

Uma vez caracterizado e conhecido quem é o usuário, é hora de iniciar a conversação com ele. Você pode usar diversas técnicas para lhe ajudar nesse processo, veja algumas:

- Entrevistas
- Observação Pessoal
- Questionários

5.1.1 Entrevistas

Você já deve ter passado por alguma entrevista, seja para emprego ou para qualquer outra finalidade. Na Engenharia de Software também a utilizamos para tentar tirar do usuário a maior quantidade possível de informações. A entrevista é uma técnica baseada na conversa direta com os usuários reais do sistema a ser analisado.
Essa técnica é ideal para ajudar na primeira etapa do levantamento dos dados. Nessa etapa os usuários têm a liberdade de apresentar toda a situação, todos os seus problemas, todas as suas necessidades, ou seja, têm a liberdade para falar o que bem entenderem sobre as questões ligadas ao seu trabalho e sobre como acreditam que um sistema de informação possa auxiliá-los na melhoria da situação atual da empresa.

As entrevistas podem ser feitas em duas fases ou momentos: um primeiro exploratório e um segundo analítico.

Essas etapas são agendadas com o usuário e, provavelmente, vão acontecer várias vezes, até que fique claro o que o *software* deve fazer.

No primeiro momento, o entrevistado irá falar sobre o assunto de uma forma geral. Nesta hora, você não deve se preocupar com os detalhes ainda, eles virão depois, é momento de você conhecer a situação, tentar ter uma visão geral. Para isto, você deve ficar atento às expressões utilizadas pelos usuários que:
- possam transparecer que ele sente necessidade de informações:
“ ... eu gostaria de gerar relatórios por fornecedores, mas não temos nenhum registro dos nossos fornecedores...”

- caracterizam problemas que ele vem sentindo:
“ ... para fechar o fluxo de caixa, eu preciso calcular sempre o valor atual do ICMS...”

- caracterizam objetivos de evolução da sua organização:
“ ... hoje em dia, só trabalhamos com regime de diárias, mas pretendemos incorporar o regime de mensalistas em seis meses...”

Um item muito importante é você saber o que vai melhorar na empresa com relação à informação, ou seja, o que o *software* pode fazer, qual informação ele dará que será útil. Isso pode ser percebido pela presença de certas palavras nas sentenças formuladas pelo usuário, como:

- verbo "saber", em expressões do tipo "eu nunca sei..." ou "eu gostaria de saber...".

- sentenças como: "eles me perguntam..." ou "o chefe me questiona sobre...".

Essas sentenças são pistas de que alguma informação é necessária;

- os verbos "planejar" e "controlar", por exemplo,

- "nossa meta é ...", "os estoques estão fora de controle...":

Sentenças como calcular, montar relatório, montar gráfico, registrar, consultar.

Atenção, anote tudo. Depois de algum tempo, você provavelmente não vai mais se lembrar do que foi falado com todos os detalhes.

A fase seguinte (analítica) trata do levantamento dos dados, isto é, da análise em si. Até aqui você estava somente conhecendo a respeito da empresa de forma abrangente.

Nessa fase, o analista interpreta os resultados obtidos e estimula os entrevistados (os clientes) a fornecer detalhes sobre as respostas que eles esperam obter do sistema, bem como dos problemas para os quais esperam obter uma solução e suas

causas. Um aspecto importante é que o entrevistador não deve solicitar detalhes sobre qualquer informação que estiver sendo transmitida nessa fase do levantamento dos dados. Em hipótese nenhuma, o entrevistado deverá ser estimulado, nessa fase, com recomendações de possíveis relatórios ou consultas que o analista presume que sejam importantes. Isso pode fazer com que ele peça itens que nem sabe se irá usar.

Outro fator importante é planejar a entrevista. Uma entrevista não pode ser apenas um diálogo conduzido pelo usuário, mas deve ser um diálogo direcionado pelo analista.

Planejamento

Todas as entrevistas devem ser planejadas. O planejamento de uma entrevista envolve os seguintes passos:

1. *Estudar material existente sobre os entrevistados e suas organizações*.

   Procure dar atenção especial à linguagem usada pelos membros da organização, procurando estabelecer um vocabulário comum a ser usado na elaboração das questões da entrevista. Este passo visa otimizar o tempo gasto nas entrevistas, evitando-se perguntas sem relevância para o *software*.

2. *Estabelecer objetivos*.

   De maneira geral, há algumas áreas sobre as quais um engenheiro de *software* desejará fazer perguntas relativas ao processamento de informação e ao comportamento na tomada de decisão tais como: fontes de informação, formatos da informação, freqüência na tomada de decisão, estilo da tomada de decisão, etc.

3. *Decidir quem entrevistar*.

   É importante incluir na lista de entrevistados pessoas-chave de todos os níveis da organização afetados pelo sistema. A pessoa de contato na

organização pode ajudar nesta seleção. Quando necessário, use amostragem.

4. *Preparar a entrevista*.

Uma entrevista deve ser marcada com antecedência e deve ter duração entre 45 minutos e uma hora.

5. *Decidir sobre os tipos de questões e a estrutura da entrevista*.

O uso de técnicas apropriadas de questionamento é o "coração" de uma entrevista. Tente definir os rumos das questões fazendo com que ao final você tenha as informações de que precisa.

6. *Decidir como registrar a entrevista*.

Entrevistas devem ser registradas para que informações obtidas não sejam perdidas logo em seguida. Os meios mais naturais de se registrar uma entrevista incluem anotações e o uso de gravador.

Vale lembrar que a técnica de entrevistas é utilizada em grupos relativamente pequenos de usuários, geralmente ligados à gerência do processo (Usuários Executivos), isto é, os usuários que estão relacionados diretamente com as negociações do sistema.

Conclusão	Entrevista é uma técnica usada para tentar entender o que o software irá fazer; é realizada com os usuários e pode ser feita várias vezes. Normalmente acontece em duas etapas: exploratória, que busca o entendimento geral do problema e analítica, que tenta levantar dados para conseguir atender às necessidades do usuário.

5.1.2 Questionários

A técnica de questionários determina a elaboração de questionamentos feitos por meio de pequenas séries de perguntas para explorar a opinião do usuário sobre algum assunto.

Podemos classificar essas perguntas como *"abertas*" ou *"fechadas*". As perguntas do tipo "*abertas"* são aquelas que requisitam a participação do usuário já que ele escreverá com suas palavras a resposta. As questões nas quais o usuário tem um conjunto de possíveis respostas preestabelecidas (questões de múltipla escolha) são do tipo *"fechadas"*. Nesse caso, o usuário pode ser especulado de maneira a expressar sua opinião concordando ou discordando de afirmativas propostas.

A figura 2-a traz um exemplo de questões abertas nas quais o usuário deve preencher o campo em branco registrando sua opinião. Esse tipo de questão é muito utilizado para questionários em que o objetivo é explorar a opinião pessoal do usuário, como por exemplo:

- saber o que ele espera do novo sistema;
- saber quais as dificuldades do sistema (computacional ou não) antigo;
- saber quais informações são necessárias.

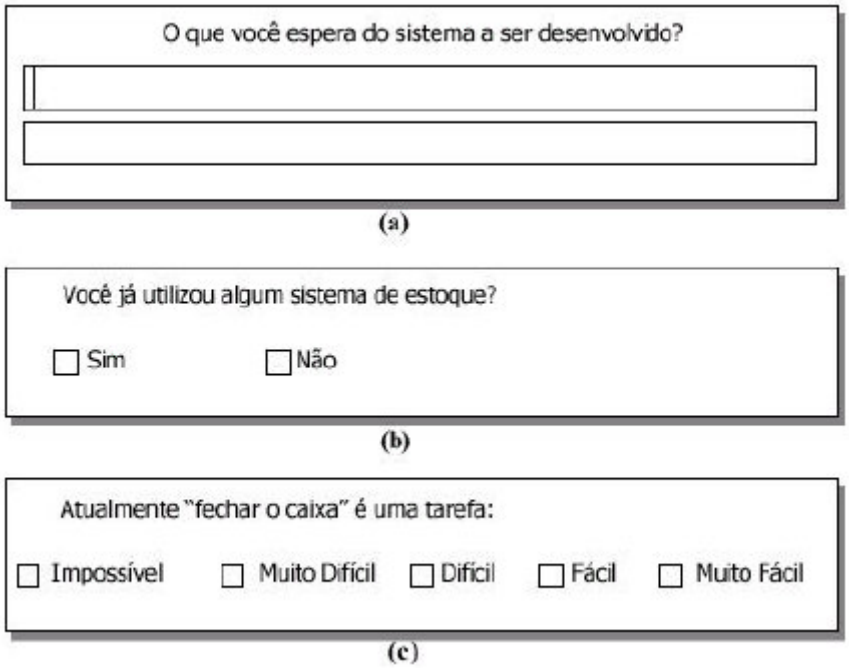


Figura 2 - Exemplos de Perguntas. (a) Tipo Open-ended. (b) Tipo Closed. (c) Tipo Closed.

Na Figura 2-b e Figura 2-c, as questões ilustradas são do tipo “fechadas”. Esse tipo de questão facilita a análise dos resultados, pois é facilmente mapeada para números, embora restrinja as possibilidades de resposta dos usuários. Como uma alternativa para enriquecer as opções de resposta, utiliza-se as escalas proporcionais.

As escalas proporcionais de avaliação são instrumentos que facilitam a conversão das respostas dos usuários para números, direcionando a opinião do usuário em intervalos escalares. Esse tipo de recurso torna o resultado mais fiel à realidade dos usuários, pois as respostas são mais elaboradas do que um simples “Sim ou Não”.

Veja abaixo alguns tipos de perguntas fechadas que utilizam escala:

Escala Comparativa. As escalas comparativas induzem o usuário a comparar elementos de uma solução de interface proposta com outros da mesma categoria, como pode ser observado na Figura 3.

Com relação às marcas de café *A* e *B*, qual sua opinião sobre a marca *B*, comparativamente à marca *A*, observando-se os seguintes atributos:				
Atributos	***Pior que A***	***Igual a A***	***Melhor que A***	***Não sei***
Pureza	_____	_____	_____	_____
Sabor	_____	_____	_____	_____
Aroma	_____	_____	_____	_____
Qualidade	_____	_____	_____	_____
Textura	_____	_____	_____	_____
Torrefação	_____	_____	_____	_____
Embalagem	_____	_____	_____	_____
Marca				

Figura 3- Escala Comparativa.

Escala de Likert. As escalas de Likert indicam o grau de concordância ou discordância dos usuários com as afirmações dadas, conforme o exemplo apresentado na Figura 4.

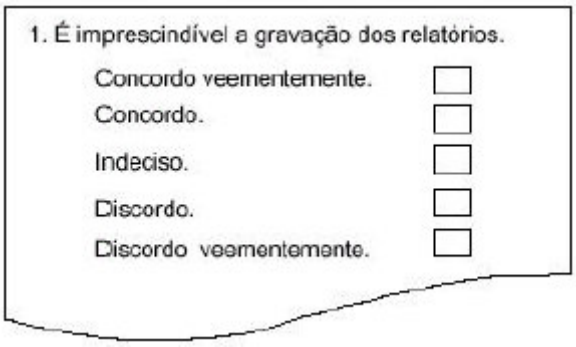


Figura 4 – Escala de Likert

Escala diferencial semântica. Este tipo de escala permite ao usuário assinalar sua opinião num subconjunto de alternativas entre pontos extremos de adjetivos opostos. A ilustração seguinte exemplifica esta escala.

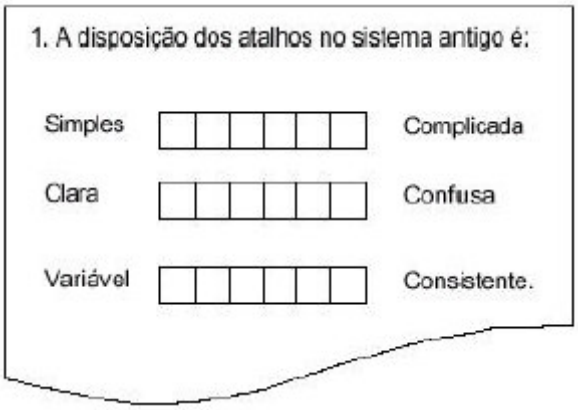


Figura 5 – Escala de diferencial semântica.

Então, a partir do uso de determinado tipo de escala ou de uma combinação de tipos podemos, utilizando simples questionamentos, explorar informações importantes para o levantamento de dados.

A composição dos questionamentos é um ponto importante e deve ser considerado. Primeiramente, você deve ter cuidado no planejamento das perguntas. As perguntas necessitam estar de acordo com os objetivos do questionário, isto é, estarem relacionadas com o que se quer avaliar. O segundo ponto é a melhor forma de questionar os usuários por meio do questionário, de modo que as perguntas não fiquem confusas, pois a precisão das perguntas vai refletir diretamente nos resultados.

Nós podemos analisar os resultados por meio de uma análise estatística das respostas individuais, que serão unidas, com o propósito de compor uma resposta capaz de espelhar a opinião de todo um grupo de usuários, ou podemos efetuar análises mais simples, envolvendo o cálculo de porcentagens baseados nos resultados obtidos.
Uma das grandes vantagens dos questionários é que essa técnica pode ser utilizada em grupos pequenos, médios e grandes de usuários que refletem todas as camadas de usuários reais do sistema. Os questionários podem ser feitos com formulários em folha de papel, via e-mail ou formulário eletrônico.

5.1.3 *Brainstorming*

O *brainstorming* (ou "tempestade cerebral") é uma atividade desenvolvida para explorar a potencialidade criativa do indivíduo. A técnica propõe que um grupo de pessoas - de uma até dez - se reúna e os componentes desse grupo se utilizem das diferenças em seus pensamentos e idéias para que possam chegar a uma solução para determinado problema com qualidade, gerando assim idéias inovadoras que levem o projeto adiante.

É preferível que as pessoas que se envolvam nesse método sejam de setores e competências diferentes, assim experiências diversas podem colaborar com a "tempestade de idéias" que se forma ao longo do processo de sugestões e discussões. Nenhuma idéia é descartada ou julgada como errada ou absurda.

Todas as idéias são ouvidas e trazidas até o processo de *brainwrite*, que constitui na compilação ou anotação das sugestões apresentadas em uma reunião com os participantes da sessão de *brainstorming*, e assim as idéias vão evoluindo até a chegada da solução efetiva.

Quando se necessita de respostas rápidas a questões relativamente simples, o *brainstorming* é uma das técnicas mais populares e eficazes. Ela é difundida e inserida em diversas outras áreas como educação, negócios, informática, internet e outras situações mais técnicas.

Como fazer

- Comece com um tema ou assunto e peça para os presentes expressarem todas as idéias que lhes venham à mente espontaneamente.

- Nenhuma idéia apresentada deve ser criticada.

- Todas as idéias verbalizadas devem ser escritas de maneira que fique visível para todos, estimulando novas idéias.

- Assim que terminar as verbalizações, todas as idéias escritas e expostas devem ser repassadas e analisadas para seleção das mais adequadas à situação.

- A seguir, selecione a melhor idéia ou conjunto de idéias que resolva o problema.

Viu como é simples? Esta técnica é muito utilizada e eficaz, experimente!!!

	Qual a importância de se identificar os requisitos do sistema antes de fazê-lo? Você acha que saber o que se quer de um software antes de começar a desenvolvê-lo pode ajudar a determinar o sucesso dele?

5.2 Estudo de Viabilidade

5.2.1 Custo X Benefício

Análise Custo-Benefício é uma coisa que todos nós fazemos a vida inteira, do momento em que nascemos até o último dia. O bebê chora mais ou chora menos para mamar dependendo da fome ou do cansaço, dos seus critérios e da sua personalidade – ele decide por análise de custo-benefício. O doente colabora mais ou menos com o tratamento em função da sua motivação, dos seus valores para lutar ou desistir – analisa os custos e os benefícios. **Todos nós sabemos fazer análise custo-benefício.** Uma definição de análise custo-benefício, em poucas palavras: **Analisar custo-benefício é avaliar se compensa fazer determinada coisa.** Parte principal, palavra-chave do conceito da análise custo-benefício: **compensa**? E **como se decide se algo compensa**?

Resposta: **Analisando e comparando com as alternativas possíveis** – sempre se tem pelo menos uma: “não fazer nada”.

O “custo” é o “esforço” necessário. Geralmente se pensa no gasto em dinheiro, mas pode ser o tempo necessário, o trabalho, os obstáculos a superar, o esforço necessário e por aí vai.
O “benefício” é a “recompensa”. Geralmente também se pensa no lucro em dinheiro, mas pode ser uma melhor condição de segurança, uma informação retornada mais rapidamente, satisfação, prazer, dentre outros.

Observe que o custo e o benefício podem envolver tanto aspectos facilmente medidos quanto aspectos impossíveis de serem medidos. E que mesmo os quantificáveis podem ter significados, impacto ou viabilidade diferente para diferentes pessoas, ou empresas.

Um exemplo de análise custo-benefício

A XYZ Corporation usa uma tecnologia para inscrever indivíduos em congressos e convenções por meio do telefone. Um dos proprietários gostaria de implementar um novo modelo baseado na internet no qual qualquer pessoa de todo o mundo seria capaz de se inscrever para uma convenção utilizando a *web*. No entanto, ao mesmo tempo em que parece ser uma grande idéia, a XYZ gostaria de saber qual é o custo de implementação e de uso do novo *software* em comparação com o atual sistema em uso e quais serão os benefícios ao longo dos próximos três anos.

Descobrir os Custos

Em primeiro lugar, você deve descobrir os custos envolvidos na implementação, o custo da aquisição de um novo servidor para executar os *softwares* adequadamente e outros custos envolvidos.

Licença para 10 computadores a US $ 500 cada	R$ 5.000
Custo do novo servidor	R$ 2.000
Custo de instalação	R$ 1.500
Custo de consultoria - honorários	R$ 1.000

Você também vai calcular quaisquer outros custos associados à execução de seu novo sistema, como treinamento, e também o tempo que os funcionários deixarão de trabalhar para aprender a usar a nova ferramenta.

Formação custos para 10 empregados x 20 horas	R$ 4.000
Custo de trabalho perdidos por 10 empregados x 20 horas	R$ 4.000

Agora, basta somar todas as despesas com a compra e implementação de seu novo *software*. Neste caso, o custo total será de R $ 17.500.

Descobrir os benefícios

Pense sobre os benefícios da aplicação desse sistema. Por exemplo, você poderia chegar à conclusão de que será capaz se inscrever duas vezes mais pessoas no mesmo período de tempo. Então, se normalmente a empresa tem um lucro de R$ 500 com inscrições durante 40 horas, com esse sistema implementado, seria capaz de lucrar R$ 1000 (R$ 500 a mais) durante o mesmo período 40 horas.

Comparar os custos com os benefícios

É fácil calcular que, se você duplicar o seu lucro em 40 horas ou uma semana de trabalho a 8 horas diárias, a implementação desse sistema que custa R$ 17500 serão pagos em exatamente 35 semanas, levando em consideração que seu lucro com as inscrições seria 100% maior. Claro que este é um simples caso de estudo, não levei em consideração o custo envolvido na disponibilização do sistema *on-line* e também não considerei que o sistema ficaria disponível 24 horas por dia e não somente oito.

Outros fatores que você deve considerar é com relação à melhoria do serviço, disponibilidade e tudo mais que é mais difícil de se transformar em número. Apesar de ser um exemplo superficial, serve para ilustrar como pode ser feita e como usar uma análise custo / benefício para se tomar decisões.

Outro aspecto da análise custo-benefício leva em consideração o custo incremental aos benefícios adicionados (melhor função e desempenho):

- Em certas situações, o custo sobe proporcionalmente aos benefícios até certo ponto. Depois o benefício adicional é excessivamente caro.

- Em outras situações, o custo sobe proporcionalmente e se mantém nesta subida estável, antes de se elevar drasticamente, para os benefícios subseqüentes.

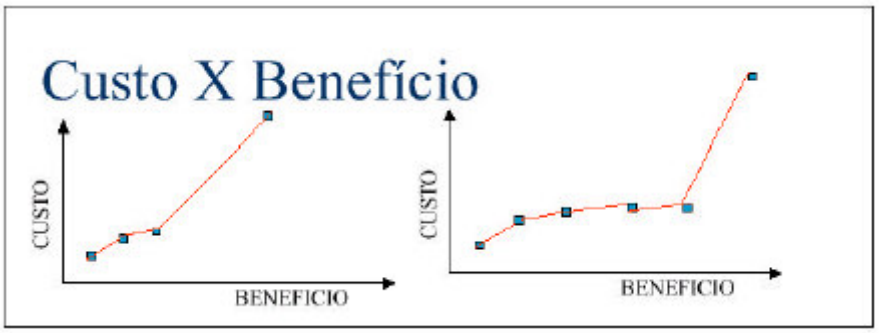


Figura 6 – Custo X Benefício

6 ANÁLISE TÉCNICA

Durante a análise técnica, o analista avalia os méritos técnicos da concepção do sistema, e, ao mesmo tempo, coleta informações adicionais sobre o desempenho, confiabilidade, manutenibilidade e capacidade de reutilização.

Um modelo é criado, tendo como base observações do mundo real, ou uma aproximação, fundamentada nas metas do sistema. O modelo deve incluir todos os fatores relevantes e ser confiável em termos de capacidade de repetir resultados. O projeto do modelo deve ser flexível a modificações e expansão.

A viabilidade técnica é a área mais difícil de ser avaliada, uma vez que os objetivos, as funções e o desempenho são um tanto quanto vagos. Algumas considerações são: riscos de desenvolvimento, disponibilidade de recursos e tecnologia. A cautela deve prevalecer na etapa de viabilidade técnica.

Como se pode perceber, essa análise é bem abstrata, não tem um padrão ou modelo a seguir, ela é nada mais que você julgar se existem as tecnologias disponíveis para resolver determinado problema e quais são elas. Após isto, devem ser considerados os pontos positivos e negativos de cada uma das possibilidades e então fazer a opção mais adequada.

6.1 DFD - DIAGRAMA DE FLUXO DE DADOS

O Diagrama de Fluxo de Dados é um modelo que você pode usar para representar como as informações trafegam dentro da empresa ou em determinada situação.
Antes de se fazer um DFD é preciso enxergar todos os processos, as informações que são lançadas, produzidas ou recebidas, os armazenamentos de dados que são feitos para só então conseguir fazer a representação de uma situação com sucesso.

Existem 4 componentes que podemos usar na montagem de um diagrama de fluxo de dados, são eles:

- Processos (bolhas)
- Entidades externas (pessoas/ fornecedores / clientes)
- Fluxo de Dados (a informação trafegando dentro do sistema)
- Depósito de Dados (arquivos)

6.1.1 Processo

Representa as transformações ocorridas com os dados. Corresponde a uma função ou atividade do sistema de informação.

Pode ser:
transformação do conteúdo que foi lançado ou recebido no conteúdo de um dado de saída;

modificação ou criação de dados armazenados, a partir do conteúdo de dados de entrada;

transformação de dados armazenados anteriormente no conteúdo de um dado de saída.

Se fosse para resumir tudo isto e facilitar sua vida, iria dizer que é uma situação na qual se pega um dado, faz algum procedimento com ele e gera uma saída podendo ser na tela, em um relatório ou até mesmo em um banco de dados.

O nome do processo deve estar relacionado com uma atividade ou função do negócio. Devem ser evitados nomes muito físicos (gravar, imprimir), muito técnicos (deletar, becapear) ou nomes muito genéricos (processar). A figura abaixo mostra a representação gráfica de um processo.

Nome do Processo = verbo no infinitivo + substantivo + (qualificador)

Figura 7 - Representação Gráfica do Processo

6.1.2 Fluxo de Dados

É utilizado para representar a movimentação de dados pelo sistema. Toda vez que tiver uma informação saindo de algum lugar e indo para outro, é o fluxo que você irá usar. Os dados transportados pelos fluxos são associados a um valor variável ou a um conjunto de valores variáveis.

O nome do fluxo deve ser um substantivo que facilite a identificação do dado ou pacote de dados transportado. Todo fluxo de dados possui direção (origem e destino). A figura abaixo mostra a representação gráfica de fluxos de dados de entrada e de saída.

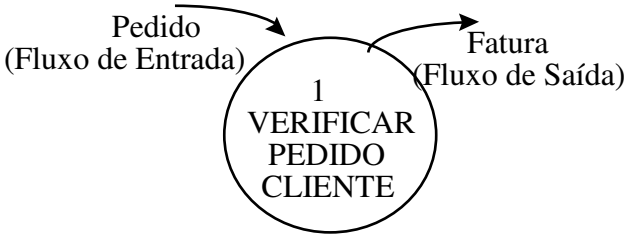


Figura 8 - Representação Gráfica de Fluxo de Dados

Conforme representado na Figura 9, fluxos de dados podem convergir ou divergir em um DFD, representando múltiplas fontes, múltiplos destinos ou combinação/separação de conteúdo.

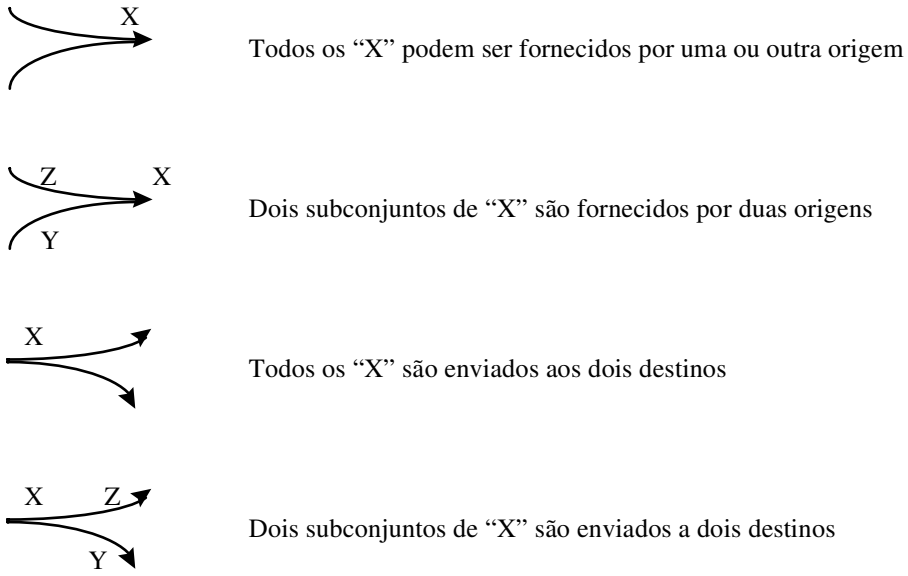


Figura 9 - Convergência e Divergência de Fluxo de Dados

6.1.3 Depósito de Dados

Representa um conjunto de dados armazenados. Pode ser considerado como uma área destinada a armazenar dados, situada entre processos que são executados em tempos diferentes.

Um depósito de dados pode ser visto fisicamente como um *pen drive*, uma área em disco, um caderno, um fichário, uma relação, uma ficha, ou seja, qualquer forma de armazenamento. A Figura 10 ilustra a representação gráfica de depósitos de dados.

PEDIDOS

Substantivo no plural

Figura 10 - Representação Gráfica de Depósito de Dados

A forma de representar os acessos aos depósitos de dados é ilustrada na Figura 11.

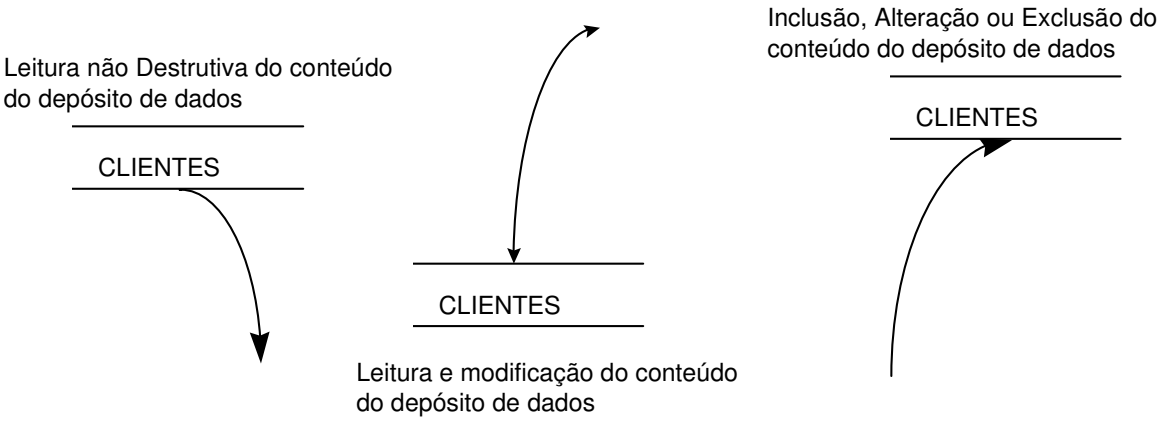


Figura 11 – Representação de acesso aos Depósitos de Dados

6.1.4 Entidade Externa

É uma fonte ou destino de dados que se comunica com o sistema fornecendo ou recebendo dados.

As entidades externas representam as interfaces do sistema, ficam fora do controle do sistema, mas interagem com ele. Por esta razão, o DFD não representa fluxos de dados entre entidades externas. A Figura 12 ilustra a representação gráfica de entidade externa.

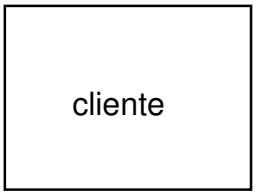
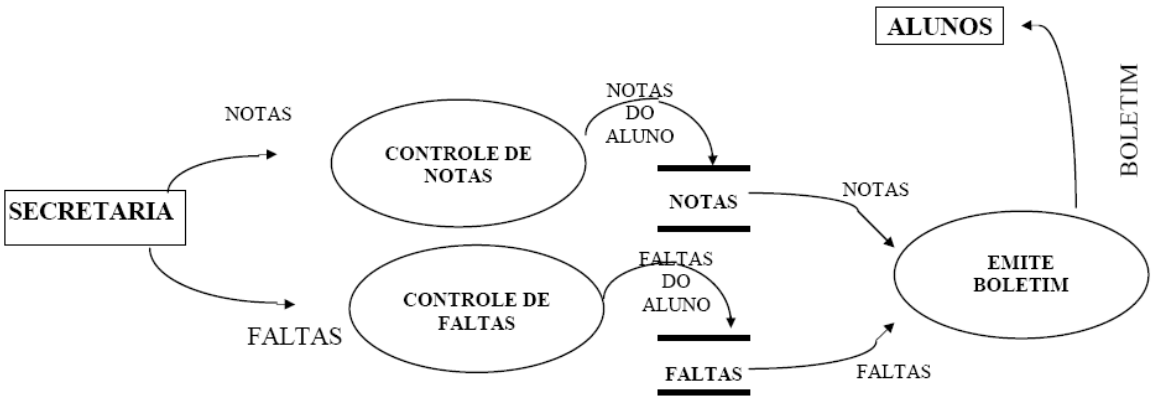


Figura 12 - Representação Gráfica da Entidade Externa

Você pode usar para lhe auxiliar uma tabela como a mostrada abaixo para fazer a anotação das entidades, processos e depósitos, isto irá ajudar muito na montagem. Veja, no exemplo, a tabela e, logo a seguir, o DFD desta tabela:

Entidades	Processos	Depósitos
Secretaria	Controle de Notas	Notas
Alunos	Controle de Faltas	Faltas
	Emissão Boletim	



Veja que é um diagrama que usa apenas quatro símbolos diferentes e, apesar disso, consegue representar o fluxo de informações sem deixar confusão. Ele é simples,

prático e bem útil na documentação de um *software* e pode ser usado até mesmo em reuniões com clientes e usuários exatamente porque é simples.

6.1.5 Organização do DFD em Níveis

Para evitar DFDs muito complexos, com um número muito elevado de processos (mais do que 9) e permitir uma visão particionada do sistema, os diagramas são organizados em níveis hierárquicos, nos quais os diagramas de nível inferior detalham processos de diagramas do nível superior.

O diagrama de mais alto nível hierárquico é denominado Diagrama de Contexto, o qual representa o sistema como um único processo e suas interações com o ambiente (o sistema, entidades externas e fluxos de entrada e saída).

O diagrama imediatamente subordinado ao Diagrama de Contexto é denominado Diagrama Zero, que representa as principais funções do sistema e as interfaces entre elas. Os processos nesse diagrama recebem os números 1, 2, 3, etc.
A seguir, os diagramas vão detalhando processos que já estão representados em diagramas anteriores. No exemplo abaixo, você pode perceber que, no diagrama Zero, temos o processo 1 e 2. Depois disso, são feitos diagramas para detalhar os processos 1 e 2. Se fosse necessário, poderíamos ter mais do que 2 processos, o importante é você dividir e organizar o DFD.

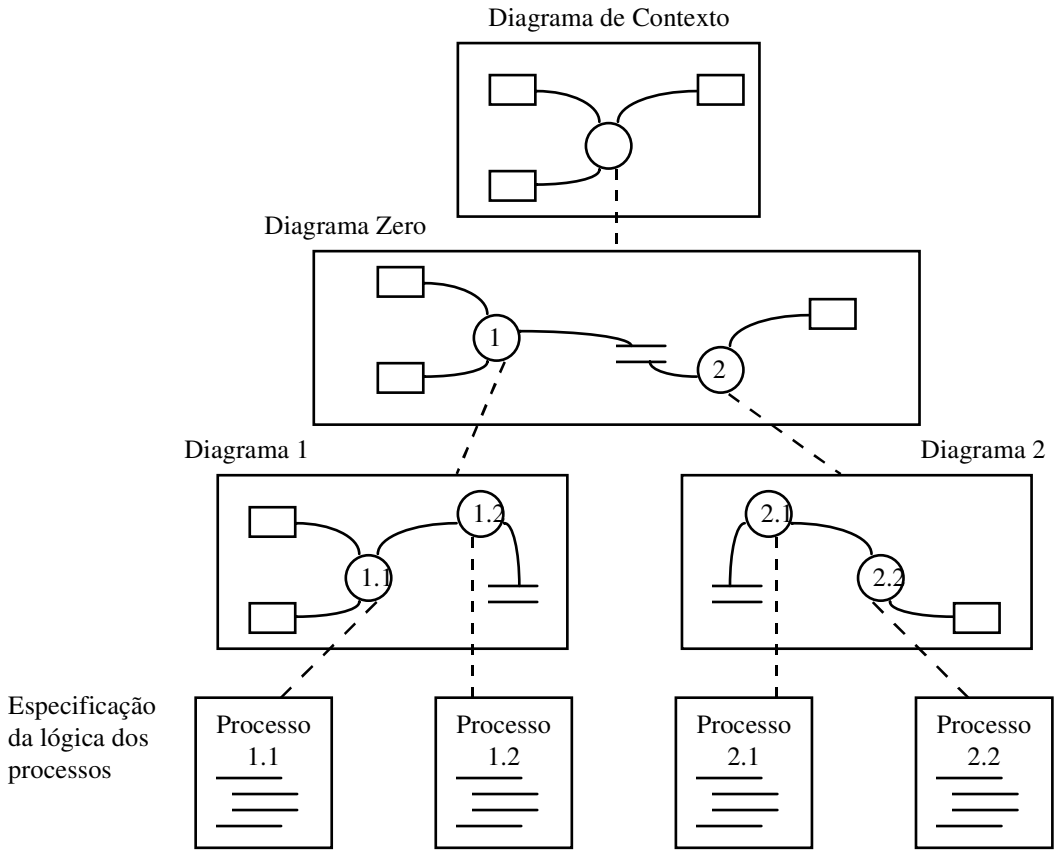


Figura 13 - DFD Organizado em Níveis Hierárquicos

6.1.6 Recomendações para Construção de DFD

a) Escolha nomes significativos para todos os objetos do DFD. Utilize nomes do próprio ambiente do usuário.
b) Os processos devem ser numerados de acordo com o diagrama ao qual pertencem.
c) Evite desenhar DFDs complexos.
d) Cuidado com as bolhas sem entrada ou sem saída.
e) Cuidado com os processos e fluxos não nomeados.
f) Cuidado com os depósitos de dados que só possuem fluxos de entrada ou de saída.

g) Fique atento ao princípio da conservação dos dados.
h) Os fluxos que entram e saem em um nível devem entrar e sair no nível inferior.
i) Mostre um depósito de dados no nível mais alto em que ele faz interface entre dois ou mais processos. Passe a representá-lo em todos os níveis inferiores que detalham os processos da interface.
j) Não perca tempo procurando um bom nome para um processo que só pode chamar-se “processar dados". Livre-se dele.
k) Só represente fluxos de rejeição nos diagramas de mais baixo nível.
l) Não represente no DFD fluxos de controle ou de material.
m) Só especifique a lógica dos processos primitivos, ou seja, dos processos não explodidos em outros diagramas.

	Seguindo essas dicas e com um pouco de treino, a chance de sucesso na representação é muito grande. Tente, encontre um fluxo de informação e faça a representação usando o DFD.

6.2 DD - DICIONÁRIO DE DADOS

6.2.1 Introdução

Você se lembra do DFD no qual tínhamos os depósitos de dados? Então, o dicionário de dados serve para detalhar cada um dos depósitos de dados. Como você já deve ter visto em banco de dados, existem as tabelas nas quais armazenamos os dados; essa ferramenta serve também para documentar as tabelas de um banco de dados.

Observe que os dados, na maioria das vezes, são armazenados em tabelas, mas isto não é uma regra, podem estar em arquivo texto, binário ou de outro tipo. O que quero dizer com isto? Que nem sempre o DD representa uma tabela, sempre representa dados armazenados, mas isto pode estar em qualquer formato desde que esteja estruturado.

6.2.2 Organização e Simbologia

Com relação à organização, é feita em uma única lista, que deve estar em ordem alfabética, sem separação entre os tipos de dados e deve conter:

- depósitos de dados;
- entidades e relacionamentos com atributos (correspondem aos registros dos depósitos de dados);
- fluxos de dados;
- estruturas de dados que compõem os registros dos depósitos de dados;
- fluxos de dados ou uma outra estrutura de dados;
- elementos de dados que compõem os registros dos depósitos de dados, fluxos de dados e as estruturas de dados.

6.2.3 Depósitos de Dados e Entidades

Deve ser feito no início do dicionário uma relação de todos os depósitos de dados existentes, isto nos dá uma visão geral e permite que antes de começar o detalhamento se tenha uma explicação do que vai ser armazenado em cada item.

A definição de um depósito de dados deve conter:

- significado do depósito;
- volume inicial de dados armazenados;
- a taxa de crescimento e
- nome do registro, o qual corresponderá a uma entidade ou a um relacionamento.

Os nomes dos depósitos de dados serão sempre escritos no PLURAL e com letras MAIÚSCULAS.

nome do depósito de dados = * significado *

* informações sobre volume *

* informações sobre taxa de crescimento *

registro do depósito de dados

Exemplo:

LIVROS = * cadastro de livros da livraria *

* volume inicial: 300 ocorrências *

* taxa de crescimento: 5 ocorrências por mês

{livro}

6.2.4 Itens de dados

Dentro de cada depósito, temos os itens de dados, que na maioria das vezes serão os campos de uma tabela. Vamos ver agora quais são os símbolos usados para representar a definição dos dados:

SÍMBOLO	SIGNIFICADO
=	é composto por
+	E
( )	dado ou estrutura de dado opcional
[|]	dados alternativos entre si - (somente um ou o outro item)
n{ }m	repetição de dados ou estruturas n - quantidade mínima de repetições; se não colocar, pode ser zero, não há limite inferior m – Quantidade máxima de repetições; se omitido, não há limite superior
*	delimitador de comentários - coloque no início, faça os comentários e coloque de novo no final
@ ou _____	indicador de chave primária de depósito de dados, pode usar o arroba antes do nome do campo ou sublinhar o campo.

Pode parecer em um primeiro momento difícil, mas veja os exemplos abaixo que vai perceber que é bem tranqüilo.

a) O registro do depósito de dados de clientes é composto, obrigatoriamente, de código-cliente, nome-cliente, endereço-cliente e, opcionalmente, de telefone-

cliente, sendo que o código-cliente é a chave do registro. Veja cada um dos símbolos e volte à tabela para conferir.

cliente = *cliente da livraria*
código-cliente + nome-cliente + endereço-cliente + (telefone-cliente)

b) O cliente pode não possuir telefone ou possuir mais de um telefone.

cliente = *cliente da livraria*
código-cliente + nome-cliente + endereço-cliente + {telefone-cliente}

c) O cliente possui no mínimo um telefone e no máximo três.

cliente = *cliente da livraria*
código-cliente + nome-cliente + endereço-cliente + 1{telefone-cliente}3

d) O cliente pode não possuir telefone ou possuir no máximo três.

cliente = *cliente da livraria*
código-cliente + nome-cliente + endereço-cliente + {telefone-cliente}3

e) O cliente possui no mínimo um telefone, podendo possuir vários outros.

cliente = *cliente da livraria*
código-cliente + nome-cliente + endereço-cliente + 1{telefone-cliente}

f) O cliente deve possuir um telefone, podendo ser comercial ou residencial.

cliente = *cliente da livraria*
código-cliente + nome-cliente + endereço-cliente + [telefone-comercial | telefone-residencial]

g) O cliente pode possuir telefone comercial, residencial ou ambos.

cliente = *cliente da livraria*

código-cliente + nome-cliente + endereço-cliente + [telefone-comercial | telefone-residencial | telefone-comercial + telefone-residencial]

6.2.5 Elementos de Dados

Os elementos de dados devem também ser detalhados item a item para que no momento da criação dos arquivos de dados durante a fase de implementação do sistema, não fiquem dúvidas do tipo de dados, para que serve o campo e tudo mais.

A definição de um elemento de dados contém:

- o significado do dado;
- as informações sobre o tipo de dado;
- o tamanho;
- o formato;
- o número de casas decimais;
- a unidade de medida;
- o domínio de valores e
- o significado dos valores do domínio e sinônimos.

Exemplos:

código-departamento = **
* domínio de valores: [10] | 20 | 30] *
* significado dos valores: 10 - Administração
20 - Industrial
30 - Vendas *

preço-item = * preço de venda do livro *
* tipo: numérico *
* tamanho: 10 posições *
* número de casas decimais: 2 *
* unidade de medida: $ *
* sinônimo: preço-venda *

Dados que possuem as mesmas características, ou que sejam sinônimos (um mesmo dado que é conhecido e tratado por mais de um nome), podem ser definidos utilizando-se referências cruzadas.

Exemplos:

data = **
formato: dd/mm/aa
data-fatura = *data de emissão de uma fatura*
data
data-pedido = *data de emissão de um pedido
data

6.2.6 Estruturas de Dados

Uma estrutura de dados é um conjunto de dados composto por mais de um elemento de dados, serve para simplificar a documentação e evitar a repetição, por exemplo:

cliente-pedido = * cliente que encaminhou o pedido *
nome-cliente + [endereço-cliente | telefone-cliente]

Cliente-pedido é composto por um conjunto de outros itens.

A definição de uma estrutura de dados contém o significado e a composição do dado. No dicionário de dados, os nomes desses dados são escritos com letras MINÚSCULAS,

estrutura = *definição *
elementos

Exemplos:

item-pedido = * item do pedido de um cliente *
nome-item + quantidade-item

pedido-livro = * pedido de livros dos clientes da livraria *
cliente-pedido + 1{item-pedido}

Para que a composição dos fluxos e as estruturas de dados sejam apresentadas de maneira *top-down* (na qual o mais geral vem em cima e o item mais detalhado vem abaixo), o que facilita o seu entendimento, deve-se procurar agrupar os elementos de dados em estruturas intermediárias.

Exemplo:

pedido-livro = * pedido de livros dos clientes da livraria* (especificação ruim)
nome-cliente + endereço-cliente + (telefone-cliente) + 1{nome-item + quantidade-item}

pedido-livro = *pedido de livros dos clientes da livraria* (especificação boa)
cliente-pedido + 1 {item-pedido}

Quando dois ou mais dados possuírem a mesma composição, suas definições podem ser feitas por meio de referências cruzadas.

Exemplo:

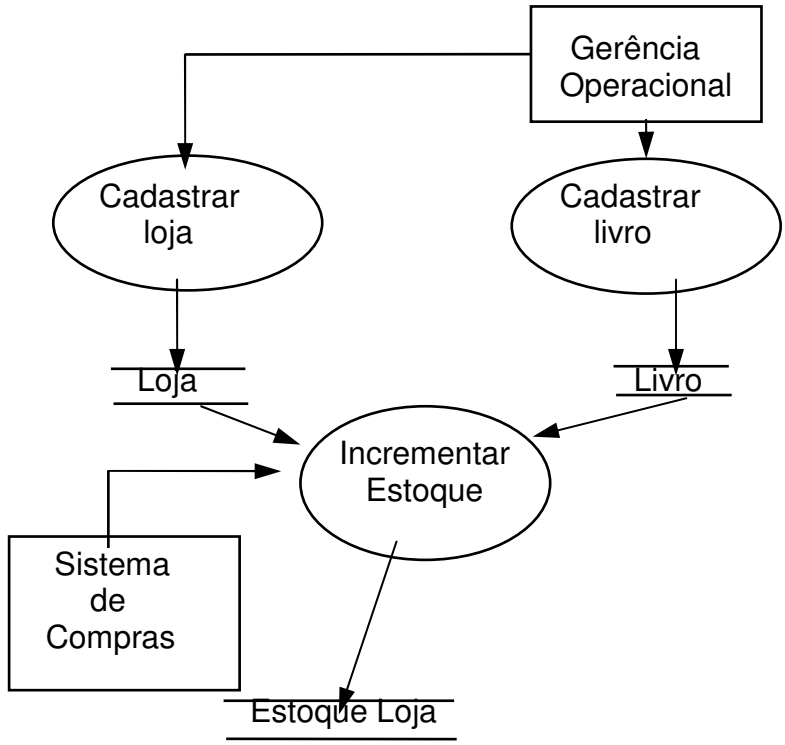
endereço = **
rua + complemento + bairro + cep + estado

endereço-cliente = **
endereço

endereço-fornecedor = **
endereço

O dicionário de dados deve ser um sistema que descreve o sistema de nosso interesse. Hoje em dia, existem no mercado diversos sistemas automatizados de dicionário de dados, com inúmeros recursos disponíveis. Além disso, com a evolução dos "*softwares*" do tipo CASE (Computer Aided Software Engeneering"), que servem de apoio ao desenvolvimento de sistemas, já se pode ter um auxílio bastante grande na tarefa de documentação. Nos CASEs, já costuma vir embutido um dicionário de dados

Para que você veja de forma mais clara o assunto, dê uma olhada no DFD abaixo e logo a seguir, no DD dos depósitos. Veja se ficou claro.



Para descrever os depósitos de dados teremos:

ender-loja = RUA
NÚMERO
COMPLEMENTO
NOME-BAIRRO
[CEP]
SIGLA-UF% da unidade de Federação %
TELEFONE % telefone para contato, inclui DDD%

loja = NUM-LOJA
ender-loja

livro = @COD-LIVRO
TÍTULO-LIVRO
NOME-AUTOR-PRINCIPAL
NUM-EDIÇÃO
PREÇO-UNITÁRIO

estoque-loja = @NUM-LOJA

@COD-LIVRO
QTDE-EM-ESTOQUE
NÍVEL-DE-REPOSIÇÃO

Não se esquecer de que, como um depósito de dados representa um conjunto de dados como se fora um "arquivo" (note que não estamos nos referindo a arquivos físicos do sistema, pois ainda não estamos nesta fase do desenvolvimento), podemos imaginar os depósitos de dados como sendo feitos de "registros" com as estruturas anteriormente definidas. Vale lembrar que, para acessar cada "registro" individualizado de cada depósito de dados, foi usado o conceito de chave de acesso, sendo usado o símbolo "@" para indicar isso.
Resulta daí que a chave para acesso ao depósito de dados livro é COD-LIVRO; a chave do depósito loja é NUM-LOJA, enquanto que a chave para o depósito estoque-loja é composta pela concatenação de NUM-LOJA e COD-LIVRO, pois somente assim poderemos saber qual a quantidade de um determinado livro em uma determinada loja.

Uma forma diferente e mais sintética de apresentar a composição de um depósito de dados ou de um fluxo de dados é por meio de um conjunto de campos (dados elementares ou estruturas de dados) separados por vírgulas, em que os campos que formam a chave dos depósitos estão sublinhados.

loja = {NUM-LOJA, RUA, NÚMERO, COMPLEMENTO, NOME-BAIRRO, CEP, SIGLA-UF, TELEFONE}

livro = {COD-LIVRO, TÍTULO-LIVRO, NOME-AUTOR-PRINCIPAL, NUM-EDIÇÃO, PREÇO-UNITÁRIO}

estoque-loja = {NUM-LOJA, COD-LIVRO, QTDE-EM-ESTOQUE, NIVEL-DE-REPOSIÇÃO}

Conclusão	Muito bem, resumindo, o dicionário de dados pode ser visto como um depósito central que descreve e define o significado de toda a informação usada na construção de um *software.*

7 GERÊNCIA DE CONFIGURAÇÃO

Durante o ciclo de vida do desenvolvimento, o *software* passa por uma série de modificações, desde a etapa de análise até a implantação.

A Gerência de Configuração de Software (CGS) vem definir critérios que permitem realizar essas modificações mantendo a consistência e a integridade do *software* com as especificações. Ela permite minimizar os problemas que acontecem durante o processo de desenvolvimento, por meio de um controle sistemático sobre as modificações. Não é objetivo da GCS evitar modificações, mas permitir que elas aconteçam sempre que possível, sem criar mais falhas e situações inesperadas.

Em geral a GCS é aplicada apenas quando existe um processo de desenvolvimento bem definido, com atividades agrupadas em fases, constituídas por objetivos bem definidos e documentados. Nesse contexto, ela atua como um suporte no qual as fases do desenvolvimento são conduzidas e os produtos controlados.

Aplicar um plano de gerência de configuração consiste em estabelecer normas, ferramentas e modelos para gerenciar de maneira satisfatória os itens de configuração de um sistema.

Podemos entender como item de configuração cada um dos elementos de informação que são criados durante o desenvolvimento de um produto de *software*, ou que para este desenvolvimento sejam necessários, que são identificados de maneira única e cuja evolução é passível de rastreamento.

Nessa definição, tanto os documentos como os arquivos-fonte que compõem um produto de *software* são Itens de Configuração (IC), assim também as ferramentas de *software* necessárias para o desenvolvimento.

Se o item de configuração for composto exclusivamente de *software*, ele é então chamado de Item de Configuração de *Software* (ICSW).

Qualquer IC constitui uma unidade funcional que possui um ciclo de desenvolvimento e acompanhamento de GCS próprios. Qualquer sistema em

desenvolvimento deve ser particionado em itens de configuração, e o seu desenvolvimento é visto como o desenvolvimento de cada um dos ICs.

Cada IC, por sua vez, pode ser particionado em outro conjunto de ICs e assim sucessivamente, até gerar um conjunto de ICs não particionáveis, que são desenvolvidos segundo um ciclo de vida definido anteriormente.

A configuração de um sistema de *software* passa a ser definida pela configuração do conjunto dos ICSW.

Em cada fase do processo de desenvolvimento, um conjunto bem definido de itens de configuração deve ser estabelecido. Esse conjunto representa um estágio do desenvolvimento, que pode sofrer modificações apenas mediante um mecanismo formal de alterações. Esses conjuntos são chamados de *Baselines*, ou Configurações Base do sistema.

Em princípio, *baselines* poderiam ser estabelecidas em qualquer ponto do desenvolvimento. Mas, a grande vantagem do conceito está em fazer coincidir o estabelecimento de *baselines* com os finais de fase do ciclo de vida do produto.

O desenvolvimento com configurações base pode, então, ser resumido nos seguintes pontos:

- Caracterização do ciclo de vida, identificando-se as fases que o desenvolvimento do *software* irá passar e, dentro delas, as atividades que vão ser realizadas e os produtos a serem desenvolvidos.

- Definição do conjunto de *baselines*. Para cada *baseline* planejada, devemos estabelecer quais serão os ICs que irão compor e quais as condições impostas para seu estabelecimento.

- *Baselines* representam marcos no processo de desenvolvimento: uma nova *baseline* é estabelecida no final de cada fase do ciclo de vida do *software*.

- Durante cada fase, o desenvolvimento dos ICs referentes a ela está sob controle de seus desenvolvedores, e isto deve ser feito com total liberdade, podendo os ICs serem criados e modificados com bastante facilidade.

- Durante cada fase, a modificação de uma configuração-base anteriormente estabelecida somente pode ser feita de forma controlada, mediante um processo bem definido, tentando garantir com isto que as conseqüências das mudanças feitas sejam conhecidas.

- Ao ser criada, cada *baseline* incorpora totalmente a anterior. Em qualquer instante do desenvolvimento, a última *baseline* estabelecida representa o estado atual do desenvolvimento como um todo, com isto os interessados conseguem saber como está o andamento do projeto.

- O estabelecimento de cada *baseline* é feito somente após ser aprovada por procedimentos de consistência interna, verificação e validação.

Dessa forma, é possível ter um controle sistemático sobre todos os itens de configuração abordados em cada fase do ciclo de vida do *software*, assim como é possível avaliar mais facilmente o seu grau de evolução.

7.1 GC do Ponto de Vista das Ferramentas de Apoio

Do ponto de vista das ferramentas existentes, a GC é formada pelas seguintes atividades:

Controle de Versão

É a espinha dorsal de toda a gerência de configuração, apoiando as atividades de controle de mudança e integração contínua. Fornece os seguintes serviços:

- identificação, armazenamento e gerenciamento dos itens de configuração e de suas versões durante todo o ciclo de vida do *software*;
- histórico de todas as alterações efetuadas nos itens de configuração;
- criação de rótulos e ramificações no projeto;
- recuperação de uma configuração em um determinado momento desejado do tempo.

Controle de Mudança

Fornece um serviço complementar ao oferecido pelo sistema de controle de versão. O foco desse tipo de ferramenta é nos procedimentos pelos quais as mudanças de um ou mais itens de configuração são propostas, avaliadas, aceitas e aplicadas.
Oferece serviços para identificar, rastrear, analisar e controlar as mudanças nos itens de configuração.

Integração Contínua

Para as necessidades da GC, bastaria um controle de construção de *software* que cuidasse da identificação, empacotamento e preparação de uma *baseline* para a entrega a um cliente externo ou interno, tornando-a um *release* ou uma *build* respectivamente.
A idéia de utilizar uma integração contínua, entretanto, vai um pouco mais além. O objetivo é garantir que as mudanças no projeto sejam construídas, testadas e relatadas tão logo quanto possível depois de serem introduzidas.
Em projetos de *software*, a construção deste é feita pela recuperação da configuração correta no sistema de controle de versão e a construção dos arquivos executáveis e de instalação do produto. Esse processo é executado geralmente após cada mudança publicada no sistema de controle de versão ou em intervalos de tempo pré-definidos.

Geralmente, são combinadas duas ferramentas separadas: uma que faz a construção do *software* e outra que monitora alterações no controle de versão e dispara a primeira para a construção.

A Gerência de Configuração é essencial para manter o desenvolvimento de *software* controlável. Ainda é grande o número de empresas que não utilizam nenhum tipo de GC ou que utilizam apenas o controle de versão nos seus projetos. As causas para essa situação são o desconhecimento da amplitude e importância da GC e o desconhecimento das ferramentas de apoio existentes.

Gerência de Configuração é uma atividade que deve ser usada em todos os projetos de desenvolvimento de *software*. A existência de várias opções de ferramentas *open source* torna a implantação da GC mais fácil principalmente para micro e pequenas empresas. Porém, é necessário algum esforço para a adequação ao processo e treinamento específicos.

8 MÉTRICAS PARA MEDIÇÃO DE *SOFTWARE*

8.1 Introdução

Atualmente, o *software* é um dos componentes que mais pesa no orçamento das organizações, e a maioria delas reconhece que é importante controlar gastos nessa área, poder verificar a qualidade do produto desenvolvido ou comprado e analisar seu desempenho.

O desejo de todo gerente é que possa ser medido com maior precisão o tempo que vai ser usado no desenvolvimento de determinado projeto, as métricas são usadas para isto, para calcular quanto tempo ou qual a complexidade de algum *software*.

8.2 Métricas de Processo

Até pouco tempo, a única maneira para estimar o processo de produção de *software* era a experiência dos técnicos envolvidos no próprio projeto, o que é bastante subjetivo e muito sujeito a erros. Isso levava a criação de produtos incompletos, podia aumentar o custo de implementação do *software* e também causava atrasos na entrega do *software*.

Viu-se, então, que era necessário definir medidas de qualidade para controlar a evolução do desenvolvimento do *software*. Foram criadas diversas ferramentas para aumentar a produtividade, indo das ferramentas CASE a novas técnicas de Programação OO que podem, efetivamente, trazer ganhos de produção, porém, não existiam métodos para medir esses ganhos.

Por muito tempo, os engenheiros de *software* enfrentaram diversos problemas para determinar qual seria o tamanho do projeto de *software*. A partir da década de 70, surgiram várias propostas para tentar solucionar esse problema, como definir métricas de processo.

As Métricas de Processo de Desenvolvimento de *Software* são um conjunto de teorias e práticas relacionadas com as medidas, permitindo fazer uma estimativa de custo, desempenho e cronograma de um projeto.

8.3 Classificação

As métricas de *software* podem ser divididas em duas grandes categorias:

- Medidas Diretas
- Medidas Indiretas

8.4 Medidas Diretas

As medições diretas do processo de engenharia de *software* avaliam o custo e o esforço gastos na manutenção e no desenvolvimento do produto. Avalia ainda:
- Número total de linhas de código gerado
-Total de erros registrados em um determinado intervalo de tempo
-Tamanho de memória ocupado pelo programa
- Velocidade de execução

As medições diretas são de obtenção simples, desde que sejam definidos os padrões para isto. Em contrapartida, outras características como funcionalidade, eficiência, complexidade, capacidade de manutenção, qualidade são bem mais difíceis de medir.

8.4.1 Pontos de Função

A análise de Pontos de Função é uma abordagem direta para a medição de *software*, que força o praticante a pensar em termos de *requisitos lógicos dos usuários* e a dominar a terminologia básica.

Com treinamento certificado pelo IFPUG (International Function Point Users Group), experiência com a "mão na massa" e requisitos bem documentados, contadores de pontos de função podem tornar-se rapidamente capazes no método. Desenvolvedores e contadores iniciantes freqüentemente “tropeçam”, por não entenderem que os mesmos termos utilizados na contagem de pontos de função e na tecnologia da informação possuem significados diferentes em cada contexto. Preste muita atenção nas definições para não confundir o termo dentro da contagem de ponto de função com o termo usado no dia-a-dia.

@	International Function Point Users Group http://www.ifpug.org Brazilian Function Point Users Group http://www.bfpug.com.br

Um dos desafios na implementação da análise de Pontos de Função é tornar o método compreensível para os desenvolvedores. Por serem baseados em requisitos funcionais dos usuários (o *que* o *software* faz), independentemente da implementação física (*como* o *software* faz o que faz), pontos de função forçam os desenvolvedores a pensar em termos lógicos.

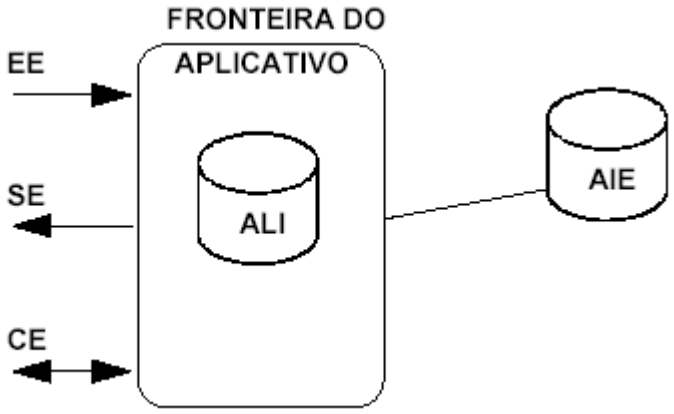
A análise de pontos de função (APF) é um método simples que determina o tamanho funcional de um sistema de informação ou projeto. O tamanho funcional pode ser usado para diferentes propósitos, por exemplo, orçamento, cálculo do prazo e qualquer outra finalidade.

Os Cinco Componentes dos Pontos de Função

O Manual de Práticas de Contagem do International Function Point Users Group (IFPUG) identifica cinco tipos de funções que são contados e avaliados:

1. **Arquivo Lógico Interno (ALI)**

2. **Arquivo de Interface Externa (AIE)**
3. **Entrada Externa (EE)**



4. **Saída Externa (SE)**
5. **Consulta Externa (CE)**

Arquivo Lógico Interno (ALI)

Um ALI é uma entidade lógica e persistente, ou seja, um depósito de dados, no qual os dados serão mantidos. Os ALI se baseiam em requisitos lógicos dos usuários e são independentes da implementação ou dos meios de armazenamento, como tabelas ou bancos de dados. Um ALI é contado com base em uma avaliação do número de campos de dados do usuário e do número de tipos de elementos de registros lógicos nele contidos.

Arquivo de Interface Externa (AIE)

Um AIE é uma entidade lógica e persistente, que é requerida para referência ou validação pelo *software* sendo contado, mas que é mantido por outro aplicativo de *software*. Simplificando, qualquer informação que você precisar importar de outro sistema ou acessar em outro sistema é um AIE (um arquivo de interface externa deve ser um arquivo lógico interno para um outro aplicativo). De forma semelhante a um ALI, um AIE é avaliado com base no número de campos de dados do usuário e

no número de tipos de elementos de registros lógicos. Arquivos de interface externa também são parte dos requisitos lógicos dos usuários.

Entrada Externa (EE)

Uma entrada externa é um processo lógico do negócio que mantém os dados em um ou mais arquivos lógicos internos, normalmente é uma tela onde o usuário digita informações. Uma entrada externa é contada com base no número de campos de dados do usuário envolvido e na soma dos ALI e AIE participantes do processo. Um exemplo de EE seria “Incluir empregado” em um aplicativo de recursos humanos.

Saída Externa (SE)

Uma saída externa é um processo lógico do negócio que gera dados para um usuário ou para outro aplicativo externo ao *software*, pode ser um relatório, um arquivo de exportação ou qualquer processo que faça uma saída de dados. Exemplos típicos de saídas externas incluem relatórios de usuários, arquivos para a Receita Federal como, por exemplo, o Sintegra. As saídas externas também fazem parte dos requisitos lógicos dos usuários.

Consulta Externa (CE)

Uma consulta externa é o último tipo de função lógica de usuário contada por meio de pontos de função. Uma CE consiste em um par gatilho-resposta (ou pergunta-resposta) em que uma pergunta ou solicitação de dados, vinda de fora (tipicamente a partir de um usuário ou de outro aplicativo), entra no aplicativo; os dados são recuperados para atender à solicitação e enviados para fora. Mais uma vez, as Consultas Externas fazem parte dos requisitos lógicos dos usuários de um aplicativo de *software*.

Este foi o primeiro passo; veja quais são todos os passos até chegarmos no cálculo do tamanho do *software*:

- Passo 1: Identificar as funções do sistema que são relevantes para o usuário.

- Passo 2: Determinar a complexidade funcional de cada função.
- Passo 3: Calcular a contagem dos pontos de função não ajustados do sistema (Pontos de Função Brutos).
- Passo 4: Indicar os requerimentos gerais para o sistema usando as 14 características gerais do sistema.
- Passo 5: Calcular a contagem dos pontos de função ajustados do sistema.

Cada tipo de função que acabei de comentar (ALI, AIE, EE, SE, CE) é classificado (passo 2), recebendo complexidade Baixa, Média ou Alta com base nas tabelas que estão logo abaixo.

Número de Registros Lógicos	Número de Itens de Dados Referenciados		
	De 1 a 19	De 20 a 50	51 ou mais
Apenas 1	Simples	Simples	Média
De 2 a 5	Simples	Média	Complexa
6 ou mais	Média	Complexa	Complexa

Tabela para identificar a complexidade dos ALI e AIE

Número de Arquivos Referenciados	Número de Itens de Dados Referenciados		
	De 1 a 4	De 5 a 15	16 ou mais
0 ou 1	Simples	Simples	Média
2	Simples	Média	Complexa
3 ou mais	Média	Complexa	Complexa

Tabela para identificar a complexidade das Entradas Externas

Número de Arquivos Referenciados	Número de Itens de Dados Referenciados		
	De 1 a 5	De 6 a 19	20 ou mais
0 ou 1	Simples	Simples	Média
2 ou 3	Simples	Média	Complexa
4 ou mais	Média	Complexa	Complexa

Tabela para identificar a complexidade das Consultas e Saídas Externas

Essa classificação vai ser usada na montagem da tabela abaixo para fazer a primeira parte da contagem (passo 3). Por exemplo, o *software* tem 15 telas, 3 de complexidade alta, 7 de complexidade média e 5 de complexidade baixa; você vai montar a tabela preenchendo-a com estes dados e calculando um total para cada linha. Depois disto, some o total de cada linha para achar um total geral, que é o que chamamos de ponto de função bruto.

Tipo de Função	Baixa	Média	Alta	Total
Arquivo Lógico Interno	3 x 7	2 x 10	2 x 15	21+20+30=71
Arquivo de Interface Externa	x 5	x 7	x 10	0
Entrada Externa	5 x 3	7 x 4	3 x 6	15+28+18=61
Saída Externa	x 4	x 5	x 7	0
Consulta Externa	x 3	x 4	x 6	0
			Total Geral	132

No exemplo acima, não preenchi todas as linhas. Pode acontecer, mas é bem difícil, de um *software* que não ter saída externa nem consulta externa, mas como era somente para demonstrar o uso da tabela, não me preocupei com isto. Veja abaixo a tabela limpa.

Tipo de Função	Baixa	Média	Alta
EE	x 3	x 4	x 6
SE	x 4	x 5	x 7
CE	x 3	x 4	x 6
ALI	x 7	x 10	x 15
AIE	x 5	x 7	x 10

O penúltimo passo na contagem de pontos de função envolve o ajuste da contagem por meio de um Fator de Ajuste de Valor (FAV), que avalia restrições de negócios adicionais do *software* não consideradas pelos cinco tipos de funções.

As quatorze características gerais de sistema do APF

1. Comunicação de Dados
2. Processamento de Dados Distribuído
3. Desempenho
4. Utilização do Equipamento (Restrições de Recursos Computacionais)
5. Volume de Transações
6. Entrada de Dados *On-line*
7. Eficiência do Usuário Final (Usabilidade)
8. Atualização *On-line*
9. Processamento Complexo
10. Reusabilidade
11. Facilidade de Implantação
12. Facilidade Operacional (Processos Operacionais, tais como Inicialização, Cópia de Segurança, Recuperação, etc.)
13. Múltiplos Locais e Organizações do Usuário
14. Facilidade de Mudanças (Manutenibilidade)

Cada uma dessas características é avaliada em uma classificação de 0 (sem influência) a 5 (muita influência), baseada em critérios claros. A soma total, também denominada de Nível de Influência, tem um valor entre 0 (14 x 0) e 70 (14 x 5). Veja a descrição de cada um dos quatorze itens que lhe ajudará a identificar qual o nível de influência de cada item. Pode parecer que são muitos passos, na primeira vez que você fizer vai achar complexo, depois vê que é bem tranqüilo.

Comunicação de dados: os aspectos relacionados aos recursos utilizados para a comunicação de dados do sistema deverão ser descritos de forma global. Descrever se a aplicação utiliza protocolos diferentes para recebimento/envio das informações do sistema.

Processamento de Dados Distribuído: Esta característica refere-se a sistemas que utilizam dados ou processamento distribuído, valendo-se de diversas CPUs.

Desempenho: Trata-se de parâmetros estabelecidos pelo usuário como aceitáveis, relativos a tempo de resposta.

Utilização do Equipamento: Trata-se de observações quanto ao nível de utilização de equipamentos requeridos para a execução do sistema. Este aspecto é observado com vista a planejamento de capacidades e custos.

Volume de transações: Consiste na avaliação do nível de influência do volume de transações no projeto, desenvolvimento, implantação e manutenção do sistema.

Entrada de dados *on-line*: A análise desta característica permite quantificar o nível de influência exercida pela utilização de entrada de dados no modo *on-line* no sistema.

Usabilidade: a análise desta característica permite quantificar o grau de influência relativo aos recursos implementados com vista a tornar o sistema amigável, permitindo incrementos na eficiência e satisfação do usuário final.

Atualizações *on-line*: Mede a influência no desenvolvimento do sistema face à utilização de recursos que visem à atualização dos Arquivos Lógicos Internos, no modo *on-line*.

Processamento complexo: a complexidade de processamento influencia no dimensionamento do sistema, e, portanto, deve ser quantificado o seu grau de influência, com base nas seguintes categorias:

Reusabilidade: a preocupação com o reaproveitamento de parte dos programas de uma aplicação em outras aplicações implica em cuidados com padronização.

Facilidade de implantação: a quantificação do grau de influência desta característica é feita observando-se o plano de conversão e implantação e/ou ferramentas utilizadas durante a fase de testes do sistema.

Facilidade operacional: a análise desta característica permite quantificar o nível de influência na aplicação, com relação a procedimentos operacionais automáticos que reduzem os procedimentos manuais, bem como mecanismos de inicialização, salvamento e recuperação, verificados durante os testes do sistema.

Múltiplos Locais e Organizações do Usuário: consiste na análise da arquitetura do projeto, observando-se a necessidade de instalação do sistema em diversos lugares.

Facilidade de mudanças: focaliza a preocupação com a influência da manutenção no desenvolvimento do sistema.

Você deve calcular o nível de influência. Para encontrar o valor, basta multiplicar cada uma das 14 características pelo seu nível de influência. Veja a tabela que pode lhe auxiliar nesse cálculo.

Característica	Influência (0 a 5)
1. Comunicação de Dados	
2. Processamento de Dados Distribuído	
3. Desempenho	
4. Utilização do Equipamento (Restrições de Recursos Computacionais)	
5. Volume de Transações	
6. Entrada de Dados *On-line*	
7. Eficiência do Usuário Final (Usabilidade)	
8. Atualização *On-line*	
9. Processamento Complexo	
10. Reusabilidade	
11. Facilidade de Implantação	
12. Facilidade Operacional (Processos Operacionais, tais como Inicialização, Cópia de Segurança, Recuperação, etc)	
13. Múltiplos Locais e Organizações do Usuário	
14. Facilidade de Mudanças	

(Manutenibilidade)	
Nível de influência (soma de todos os itens)	

Usando o Nível de Influência das características gerais do sistema (passo 4), o tão falado *Fator de Ajuste de Valor* é calculado:

Fator de Ajuste = 0,65 + 0,01* Nível de Influência

A contagem dos pontos de função ajustados é calculada da seguinte forma:

Contagem dos Pontos de Função Ajustados = Contagem dos Pontos de Função Não Ajustados x Fator de Ajuste

Ufa, chegamos lá, um pouco trabalhoso não é? Mas o que acha melhor, realizar este trabalho ou "chutar" o tempo que vai ser gasto no projeto? Espero que escolha a primeira opção.

Conclusão	O cálculo funcional de tamanho e pontos de função oferece um processo objetivo e replicável para avaliar o tamanho lógico do *software* com base em requisitos funcionais dos usuários.

8.5 Medidas Indiretas

As medidas indiretas tentam medir características como:

- Eficiência
- Manutenibilidade (facilidade de manutenção do programa)
- Funcionalidade
- Complexidade

8.5.1 Métricas Orientadas ao Tamanho

São medidas diretas do *software* e da forma usada para fazer o seu desenvolvimento. As métricas orientadas ao tamanho são muito discutidas e não são totalmente aceitas como maneira eficiente de se calcular o processo de desenvolvimento do programa, mas ainda assim é muito melhor do que contar com o “chute” no cálculo de prazo pelos analistas envolvidos.

A mais comum medida de *software* orientada ao tamanho é a contagem das linhas de código (LOC), que embora seja simples, também é muito discutida. Para alguns, a contagem das linhas de código não deve incluir linhas com comentário ou em branco, já que elas só têm o intuito de documentar e organizar, não afetando a funcionalidade do *software*.

Outra questão levantada é a forte vinculação dessa métrica com a linguagem que foi usada no desenvolvimento, o que impossibilita a utilização de dados históricos em projetos nos quais não se utilizam a mesma linguagem. Outros pesquisadores ainda alertam que as métricas orientadas ao tamanho tendem a penalizar *softwares* que possuem estrutura bem organizada. Se um programador gastar poucas linhas para fazer uma função e outro gastar muitas por falta de conhecimento, este segundo terá mais valor.

A partir desses conceitos podemos definir o seguinte conjunto de métricas como exemplo:

Qualidade	Produtividade	Documentação	Custo

Defeitos / Linhas de código	Linhas de código / Pessoa	Páginas de documentação / Linhas de código	R$ / Linhas de código

Exemplos de métricas orientadas ao tamanho

Portanto podem ser usadas essas métricas para a medição direta de um *software*; é feita a contagem de quantas linhas de código possui e, usando a tabela modelo acima, preenche-se os indicadores que servirão como referência no gerenciamento do projeto.

9 QUALIDADE DE *SOFTWARE*

Durante o desenvolvimento de um *software,* diversas situações acontecem, a saída de um funcionário chave para o projeto, a mudança de algum requisito, problemas não previstos antes, riscos que não foram identificados e uma série de outras situações. Tudo isso pode fazer com que aos poucos a qualidade vá caindo até chegar a um nível menor do que o mínimo esperado.
Pensando nisso, diversos órgãos criaram certificações e padrões para medir a capacidade de uma equipe ou empresa para contornar todos esses imprevistos, cumprir as etapas do ciclo de vida escolhido com sucesso, enfim, de alguma forma, garantir que o produto final tenha determinado grau de qualidade.

Um *software* é considerado de qualidade se for correto, consistente, compreensível, determinado, testável e possível de ser desenvolvido. Para garantir a Qualidade do *Software*, é necessário fazer diversas avaliações incluindo: verificação, validação e testes no *software* considerado.

Um aspecto interessante da qualidade é que não basta sua existência. Ela deve ser reconhecida pelo cliente. Por causa disso, é necessário que exista algum tipo de certificação oficial, emitida com base em um padrão. Você provavelmente já conhece alguns certificados mais comuns:

- O selo do SIF de inspeção da carne.
- O selo da ABIC nos pacotes de café.
- O certificado da Secretaria de Saúde para restaurantes.
- A classificação em estrelas dos hotéis.
- Os certificados de Qualidade da série ISO-9000.

Vamos ver então alguns dos diversos padrões existentes que podem indicar que o *software* produzido é de qualidade.

9.1 *CMM - Capability Maturity Model*

O Modelo de Maturidade da Capacidade (CMM) é uma iniciativa do SEI (*Software Engineering Institute*) para avaliar e melhorar a capacitação de empresas que produzem *software*. O projeto CMM foi apoiado pelo Departamento de Defesa do Governo dos Estados Unidos, que, por ser um grande consumidor de *software,* precisava de um modelo formal que permitisse selecionar os seus fornecedores de forma adequada, sem tomar sustos no final. Embora não seja uma norma emitida por uma instituição internacional (como a ISO ou o IEEE), esta norma tem tido uma grande aceitação mundial, até mesmo fora do mercado americano. O modelo pode ser obtido com facilidade na internet no site do SEI/CMU. O CMM também é conhecido como SW-CMM (*CMM for Software*).

@	SEI - Software Engineering Institute http://www.sei.cmu.edu/cmm/

9.1.1 Maturidade

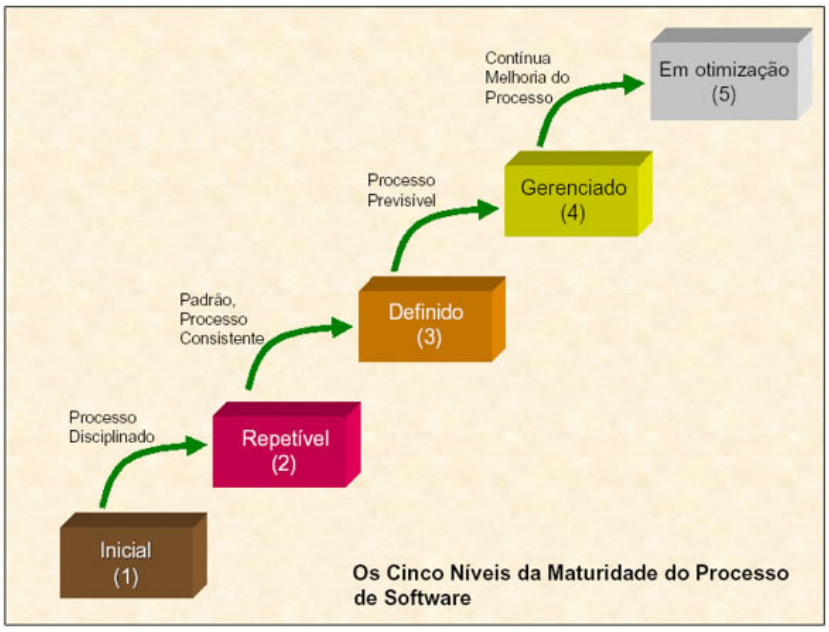
O CMM é um modelo para medir a maturidade que uma organização tem para desenvolver um *software*. A “Maturidade”, neste caso, é a capacidade que a empresa tem para documentar o projeto, se manter organizada, dentro dos prazos e custos, com uma boa qualidade, enfim uma série de fatores. Veja o quadro abaixo que vai lhe dar uma visão mais clara da diferença de uma empresa madura e uma imatura:

Organizações maduras	Organizações imaturas

Papéis e responsabilidades bem definidos.	Processo improvisado.
Existe base histórica.	Não existe base histórica.
É possível julgar a qualidade do produto.	Não há maneira objetiva de julgar a qualidade do produto.
A qualidade dos produtos e processos é monitorada.	Qualidade e funcionalidade do produto sacrificada.
O processo pode ser atualizado.	Não há rigor no processo a ser seguido.
Existe comunicação entre o gerente e seu grupo.	Resolução de crises imediatas.

9.1.2 Níveis

O CMM classifica as organizações em cinco níveis distintos, cada um com suas características próprias. No nível 1, o das organizações mais imaturas, não há nenhuma metodologia implementada e tudo ocorre de forma desorganizada (*ad-hoc*). No nível 5, o das organizações mais maduras, cada detalhe do processo de desenvolvimento está definido, quantificado e acompanhado e a organização consegue até absorver mudanças no processo sem prejudicar o desenvolvimento.



9.1.3 Descrição de cada nível do CMM

1) Inicial. O processo de desenvolvimento é desorganizado e até caótico. Poucos processos são definidos e o sucesso depende de esforços individuais e heróicos.

2) Repetível. Os processos básicos de gerenciamento de projeto estão estabelecidos e permitem acompanhar custo, cronograma e funcionalidade. É possível repetir o sucesso de um processo utilizado anteriormente em outros projetos similares.

3) Definido. Tanto as atividades de gerenciamento quanto de engenharia do processo de desenvolvimento de *software* estão documentadas, padronizadas e integradas em um padrão de desenvolvimento da organização. Todos os projetos utilizam uma versão aprovada e adaptada do processo padrão de desenvolvimento de *software* da organização.

4) Gerenciado. São coletadas medidas detalhadas da qualidade do produto e do processo de desenvolvimento de *software*. Tanto o produto quanto o processo de desenvolvimento de *software* são entendidos e controlados quantitativamente.

5) Otimizado. O melhoramento contínuo do processo é conseguido por meio de um "feedback" quantitativo dos processos e pelo uso pioneiro de idéias e tecnologias inovadoras.

Uma empresa no nível 1 não dá garantia de prazo, custo ou funcionalidade. No nível 2, a empresa já consegue produzir bons *softwares*, no prazo e a um custo previsível. O nível 3 garante um excelente nível de qualidade, tanto no produto quanto no processo de desenvolvimento como um todo. Não há, no mundo, muitas empresas que tenham chegado aos níveis 4 e 5.

9.1.4 Áreas-chave de processo (Key Process Areas ou KPAs)

Uma pergunta que surge é: Como vou classificar cada empresa? Como vou saber se está no nível 1 ou nível 2? Existe o que é chamado de KPA, que são as características de cada nível, se uma empresa cumpre todas as KPA´s de um nível, ela pode se submeter a uma certificação, que é feita por uma organização homologada que visita a empresa e verifica se ela está realmente cumprindo ou não todos os processos corretamente. Vale lembrar que mesmo se não for tentar uma certificação, seguir os procedimentos determinados ajuda muito a alcançar a qualidade.

Exceto no nível 1 que não tem nenhuma definição, todos os níveis são detalhados em áreas-chave de processo. Essas áreas são exatamente aquilo no que a organização deve focar para melhorar o seu processo de desenvolvimento de *software*. Para que uma empresa possa se qualificar em um determinado nível de maturidade CMM, deve fazer todos os processos relacionados às áreas-chave daquele determinado nível. Todas as áreas-chave estão citadas na tabela abaixo:

Nível CMM	Áreas-chave de processo
1) Inicial	
2) Repetível	Processos de gerenciamento de projetos Gerenciamento de requisitos Planejamento do projeto Visão geral e acompanhamento do projeto Gerenciamento de subcontratados Garantia da qualidade do *software* Gerenciamento de configuração
3) Definido	Processos de engenharia e apoio Foco do processo organizacional Definição do processo organizacional Programa de treinamento Gerenciamento de *software* integrado Engenharia de produto de *software* Coordenação intergrupos Revisão conjunta
4) Gerenciado	Qualidade do produto e do processo Gerenciamento quantitativo dos processos Gerenciamento da qualidade de *software*
5) Em Otimizado	Melhoramento contínuo do processo Prevenção de defeitos Gerenciamento de mudanças tecnológicas Gerenciamento de mudanças no processo

Objetivos das áreas-chave de processo

O modelo CMM define um conjunto de dois a quatro objetivos para cada área-chave. Esses objetivos definem aquilo que deve ser alcançado no caso dos processos desta área-chave serem realmente realizados.

Nível	Objetivo principal
1	Não possui KPAs.
2	Estabelecer controles básicos de gerência.
3	Fundir as ações técnicas e gerenciais em um único processo.
4	Entender quantitativamente o processo de *software.*
5	Manter, de maneira contínua, a melhoria do processo.

Características comuns e práticas-chave

As características comuns são itens que devem ser feitos para que se possa verificar a implementação e institucionalização de cada área-chave de processo. Elas podem indicar se a área-chave é eficiente, repetível e duradoura. São cinco as características comuns no modelo CMM e cada uma possui suas práticas-base a serem realizadas.

Característica comum	Descrição	Particularidades
Compromisso em executar	Ações que a organização deve tomar para assegurar que o processo está estabelecido e que irá durar.	Políticas Organizacionais. Envolvimento da Gerência. Responsabilidades.
Habilitações	Pré-condições que devem existir	Recursos.

para executar	para implementar o processo.	Estrutura da organização. Delegação de responsabilidades. Treinamento. Coordenação.
Atividades a realizar	Procedimentos necessários para implementar uma ACP.	Estabelecimento de planos e procedimentos. Execução do trabalho. Acompanhamento. Tomada de ações corretivas, sempre que necessário.
Medições e Análise	Medições a serem realizadas com relação à ACP e aos resultados das atividades.	Medições em geral.
Verificações da implementação	Passos para assegurar que as atividades estão sendo executadas conforme o processo.	Revisões e auditorias realizadas nos diversos níveis de controle.

As práticas-chave descrevem as atividades que contribuem para atingir os objetivos de cada área-chave do processo. Em geral, são descritas com frases simples, seguidas de descrições detalhadas (chamadas de subpráticas) que podem até incluir exemplos. As práticas-base devem descrever “o que” deve ser feito e não “como” os objetivos devem ser atingidos. O modelo CMM inclui um extenso documento em separado, chamado “Práticas-chave para o CMM”, que lista todas as práticas-chave e subpráticas para cada uma das áreas-chave de processo.

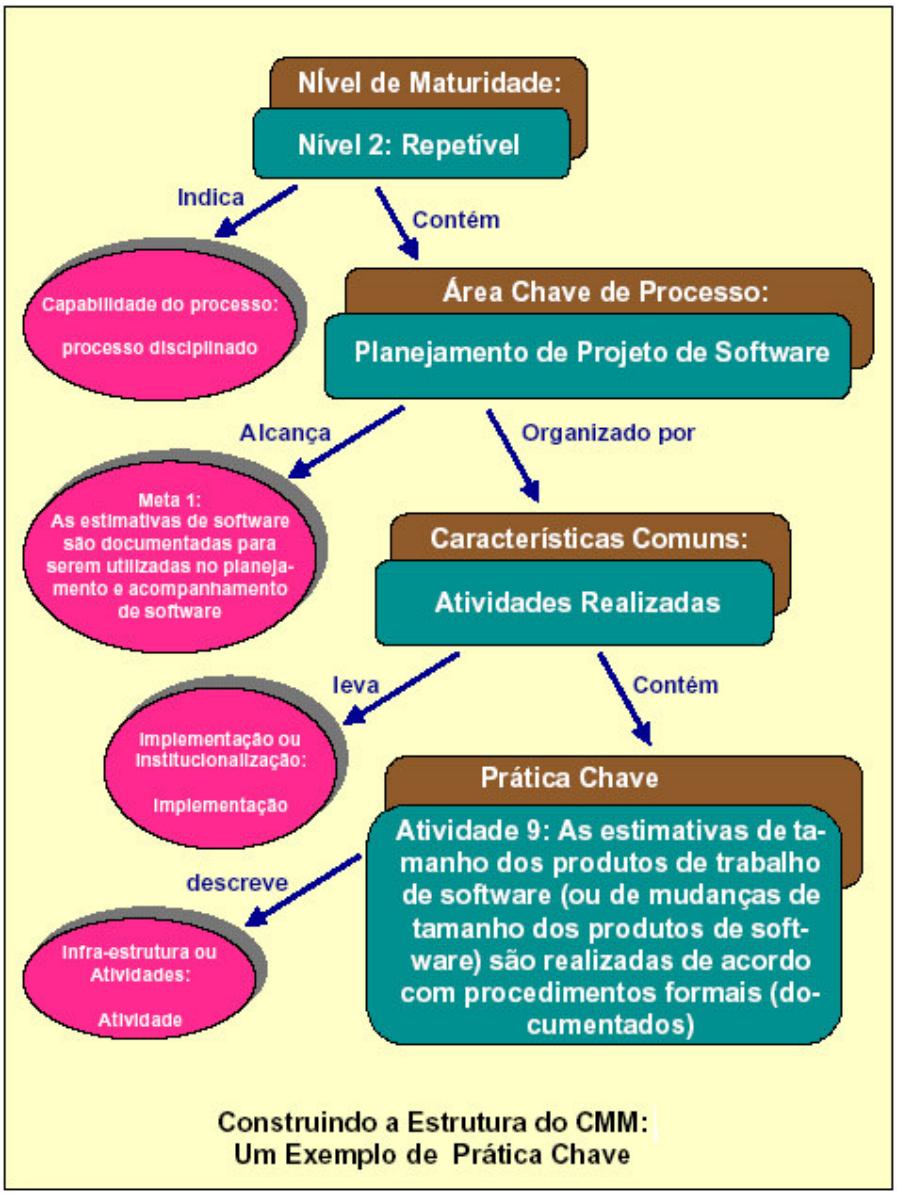
Estrutura

Em resumo, o CMM é definido em função de um conjunto de:

- Níveis de maturidade;
- Áreas-chave de processo;

- Características comuns;
- Práticas-base.

Veja na figura abaixo como esses elementos se interligam na estrutura do CMM usando como exemplo a KPA Planejamento de Projeto de *Software* do Nível 2.



***Key process area*:** Planejamento de projeto de *software*

Meta 1: Estimativas estão documentadas para uso no planejamento e acompanhamento do projeto de *software.*

Meta 2: Atividades e compromissos do projeto de *software* estão planejados e documentados.

Meta 3: Grupos e indivíduos envolvidos concordam com seus compromissos relativos ao projeto de *software.*

Não vou comentar as outras KPA´s porque são muitas. Se você quiser implementar o CMM em algum lugar, certamente, terá que usar o documento completo, com todas as áreas-chave comentadas e, provavelmente, ainda precisará da ajuda de um consultor.

Você já deve ter percebido que conseguir um certificado de qualidade não é tão simples, mas, se a sua empresa vai tirar proveito disto, com certeza vale a pena. Além de produzir *software* de melhor qualidade com este esforço para seguir os padrões, isso vai ser divulgado na mídia e é uma ótima referência. Todos que estão envolvidos com o desenvolvimento de produtos profissionais sabem que somente quem tem um bom nível de maturidade consegue se certificar.

9.2 SPICE - Software Process Improvement and Capability dEtermination - ISO 15504

O SPICE é uma norma em elaboração conjunta pela ISO e pelo IEC. Ela é um padrão para a avaliação do processo de *software*, visando determinar a capacitação de uma organização. A norma tenta também orientar a organização para uma melhoria contínua do processo. Ela cobre todos os aspectos da Qualidade do Processo de *Software*.

O SPICE inclui um modelo de referência, que serve de base para o processo de avaliação. Esse modelo é um conjunto padronizado de processos fundamentais, que orientam para uma boa engenharia de *software*. Ele é dividido em cinco grandes categorias de processo: Cliente-Fornecedor, Engenharia, Suporte, Gerência e

Organização. Cada uma dessas categorias é detalhada em processos mais específicos. Tudo isso é descrito em detalhes pela norma.
Além dos processos, o SPICE define também os seis níveis de capacitação de cada processo, que pode ser incompleto, executado, gerenciado, estabelecido, previsível e otimizado. O resultado de uma avaliação é, portanto, um perfil da instituição em forma de matriz, no qual temos os processos nas linhas e os níveis nas colunas.

Categorias e Processos

Uma das contribuições do modelo SPICE é definir em seu modelo de referência todos os processos envolvidos no desenvolvimento de *software*, agrupados em categorias. Observe no quadro abaixo a estrutura completa das categorias, dos processos que fazem parte delas:

Processo	**Descrição**
CUS - Cliente-Fornecedor Processos que impactam diretamente os produtos e serviços de *software* do fornecedor para o cliente.	
CUS.1	Adquirir *software.*
CUS.2	Gerenciar necessidades do cliente.
CUS.3	Fornecer *software.*
CUS.4	Operar *software.*
CUS.5	Prover Serviço ao cliente.
ENG - Engenharia Processos que especificam, implementam ou mantêm um sistema ou produto de *software* e sua documentação.	
ENG.1	Desenvolver requisitos e o projeto do sistema.
ENG.2	Desenvolver requisitos de *software.*
ENG.3	Desenvolver o projeto do *software.*
ENG.4	Implementar o projeto do *software.*

ENG.5	Integrar e testar o *software.*
ENG.6	Integrar e testar o sistema.
ENG.7	Manter o sistema e o *software.*
SUP - Suporte Processos que podem ser empregados por qualquer um dos outros processos.	
SUP.1	Desenvolver a documentação.
SUP.2	Desempenhar a gerência de configuração.
SUP.3	Executar a garantia da qualidade.
SUP.4	Executar a verificação dos produtos de trabalho.
SUP.5	Executar a validação dos produtos de trabalho.
SUP.6	Executar revisões conjuntas.
SUP.7	Executar auditorias.
SUP.8	Executar resolução de problemas.
MAN - Gerência Processos que contêm práticas de natureza genérica que podem ser usadas por quem gerencia projetos ou processos dentro de um ciclo de vida de *software.*	
MAN.1	Gerenciar o projeto.
MAN.2	Gerenciar a qualidade.
MAN.3	Gerenciar riscos.
MAN.4	Gerenciar subcontratantes.
ORG - Organização Processos que estabelecem os objetivos de negócios da organização.	
ORG.1	Construir o negócio.
ORG.2	Definir o processo.
ORG.3	Melhorar o processo.
ORG.4	Prover recursos de treinamento.
ORG.5	Prover infra-estrutura organizacional.

A norma define detalhes de cada um dos processos mencionados acima. Para cada um deles existe uma definição mais detalhada, uma lista dos resultados da sua

implementação bem sucedida e uma descrição detalhada de cada uma das práticas básicas.

Níveis de Capacitação

O SPICE, entretanto, não se limita a listar categorias e processos. Seu principal objetivo, na realidade, é avaliar a capacitação da organização em cada processo e permitir a sua melhoria. O modelo de referência do SPICE inclui seis níveis de capacitação. Cada um dos processos mencionados acima deve ser classificado nesses níveis. Os níveis são descritos a seguir:

Nível	**Nome**	**Descrição**
0	Incompleto	Há uma falha geral em realizar o objetivo do processo. Não existem produtos de trabalho nem saídas do processo facilmente identificáveis.
1	Realizado	O objetivo do processo em geral é atingido, embora não necessariamente de forma planejada e controlada. Há um consenso na organização de que as ações devem ser realizadas e quando são necessárias. Existem produtos de trabalho para o processo e eles são utilizados para atestar o atendimento aos objetivos.
2	Gerenciado	O processo produz os produtos de trabalho com qualidade aceitável e dentro do prazo. Isso é feito de forma planejada e controlada. Os produtos de trabalho estão de acordo com padrões e requisitos.
3	Estabelecido	O processo é realizado e gerenciado usando um processo definido, baseado em princípios de Engenharia de Software. As pessoas que implementam o processo usam processos aprovados, que são versões adaptadas do processo padrão documentado.
4	Predizível	O processo é realizado de forma consistente, dentro dos limites de controle, para atingir os objetivos. Medidas da realização do

		processo são coletadas e analisadas. Isso leva a um entendimento quantitativo da capacitação do processo a uma habilidade de predizer a realização.
5	Otimizado	A realização do processo é otimizada para atender às necessidade atuais e futuras do negócio. O processo atinge seus objetivos de negócio e consegue ser repetido. São estabelecidos objetivos quantitativos de eficácia e eficiência para o processo, segundo os objetivos da organização. A monitoração constante do processo, segundo esses objetivos, é conseguida obtendo *feedback* quantitativo, e o melhoramento é conseguido pela análise dos resultados. A otimização do processo envolve o uso piloto de idéias e tecnologias inovadoras, além da mudança de processos ineficientes para atingir os objetivos definidos.

Conclusão	A ISSO 15507 é uma norma técnica completa que permite avaliar não só o processo de desenvolvimento, mas também outros aspectos como suporte, gerência, dentre outros.

9.3 MPS.BR

O MPS.BR é um programa para Melhoria de Processo do Software Brasileiro coordenado pela Associação para Promoção da Excelência do Software Brasileiro (SOFTEX). Ele baseia-se nos conceitos de maturidade e capacidade de processo para a avaliação e melhoria da qualidade e produtividade de produtos de *software* e serviços relacionados.

@	MPS.BR http://www.softex.br/mpsBr/_home/default.asp SOFTEX - Associação para Promoção da Excelência do Software Brasileiro http://www.softex.br

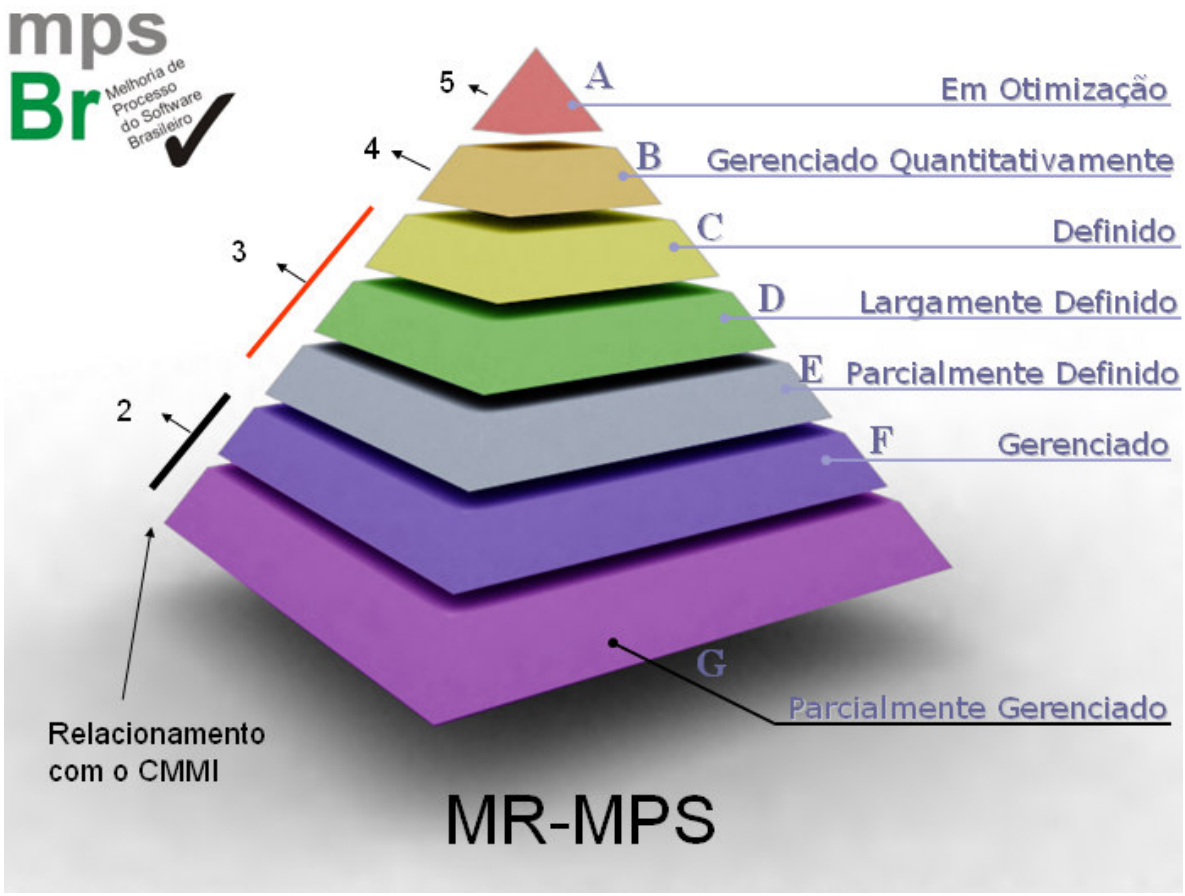
Uma de suas metas é definir e aprimorar um modelo de melhoria e avaliação de processo de *software*, tentando trabalhar principalmente com as micro, pequenas e médias empresas. O MPS.BR também estabelece um processo e um método de avaliação, para dar sustentação e garantir que está sendo empregado de forma coerente com as suas definições, ou seja, também existe um processo de avaliação para afiançar se determinada empresa possui certo nível de qualidade.

O Modelo de Referência MR-MPS define níveis de maturidade que são uma combinação entre processos e capacidade. A capacidade do processo é a caracterização da habilidade deste para alcançar os objetivos de negócio, atuais e futuros; estando relacionada com o atendimento aos atributos de processo associados aos processos de cada nível de maturidade.

Níveis de maturidade

Os níveis de maturidade são os patamares de evolução de processos, caracterizando estágios de melhoria da implementação de processos na organização. O nível de maturidade de uma organização permite prever o seu desempenho futuro ao executar um ou mais processos. O MR-MPS define sete níveis de maturidade: A (Em Otimização), B (Gerenciado Quantitativamente), C (Definido), D (Largamente Definido), E (Parcialmente Definido), F (Gerenciado) e G (Parcialmente Gerenciado). A escala de maturidade se inicia no nível G e progride até o nível A. Para cada um destes sete níveis de maturidade, é atribuído um perfil de processos que indicam onde a organização deve colocar o esforço de melhoria. O progresso e o alcance de um determinado nível de maturidade MPS são conseguidos quando são atendidos os propósitos e todos os resultados esperados dos respectivos processos e dos atributos de processo estabelecidos para aquele nível.

Veja abaixo uma comparação dos níveis do MPS com o CMMI:



Processo

Os processos no MR-MPS são descritos em propósito, resultados e informações adicionais.

O propósito descreve o objetivo geral que tem que ser atingido durante a execução do processo. Os resultados esperados são as metas que a empresa deve alcançar. Esses resultados podem ser constatados por um *software* produzido ou uma mudança significativa de estado ao se executar o processo.

As informações adicionais são referências que podem ajudar na definição do processo pela organização. Essas referências fornecem descrições de atividades, tarefas e melhores práticas que podem apoiar a definição e implementação do processo nas organizações.

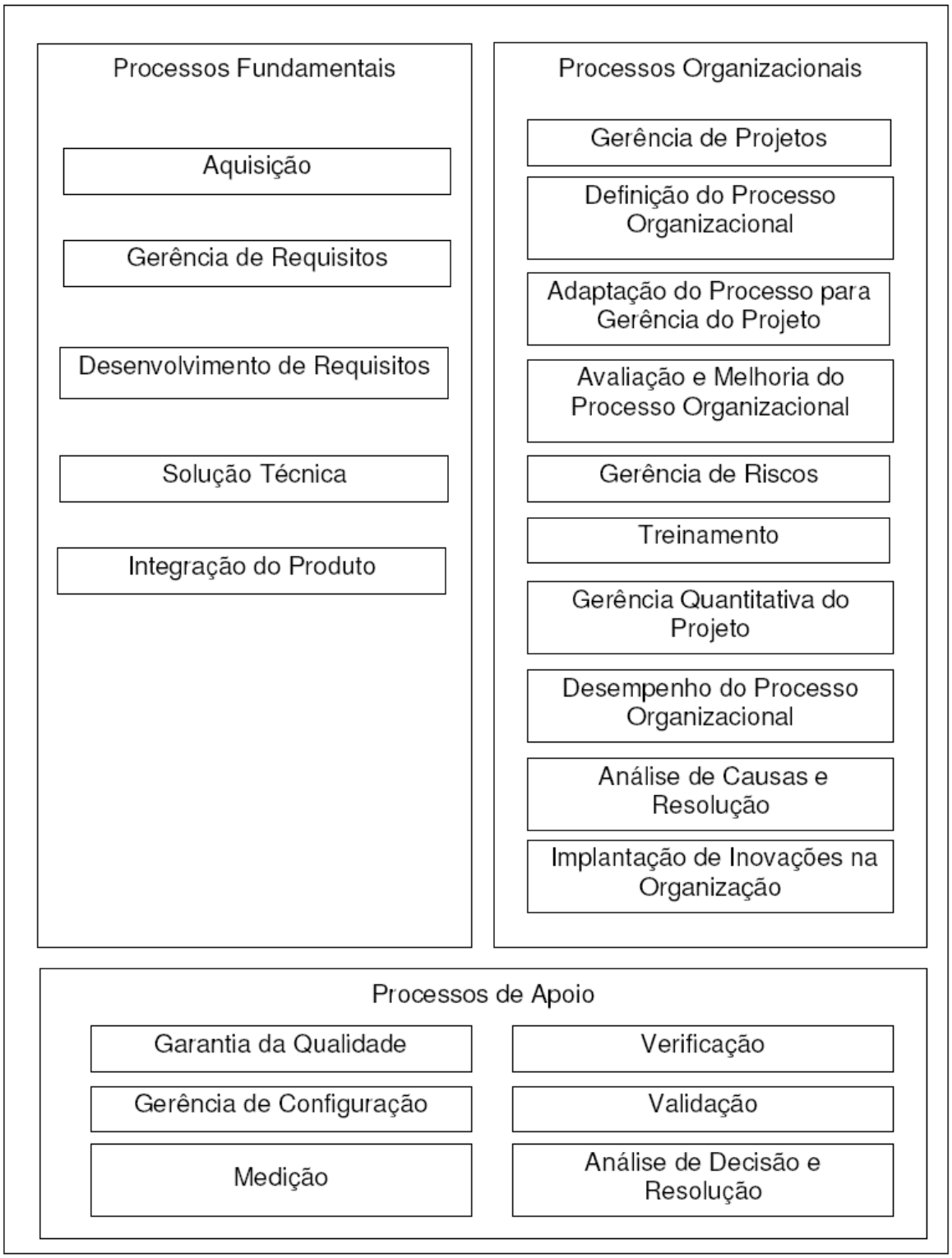
Os processos são agrupados, por uma questão de organização, de acordo com a sua natureza, ou seja, o seu objetivo principal no ciclo de vida de *software*. Esse agrupamento resultou em três (3) classes de processos, que são:

· Processos fundamentais - atendem o início e a execução do desenvolvimento, operação ou manutenção dos produtos de *software* e serviços correlatos durante ciclo de vida de *software*.

· Processos de apoio - auxiliam um outro processo e contribuem para o sucesso e qualidade do projeto de *software*.

· Processos organizacionais - uma organização pode empregar estes processos no âmbito corporativo para estabelecer, implementar e melhorar um processo do ciclo de vida.

Veja a representação dos processos:



Processos do MR-MPS

Para cada nível que a organização pretende alcançar, ela deve focar a melhoria de alguns processos. Veja qual o processo envolvido em cada nível:

Nível	Processos
A (mais alto)	- Implantação de Inovações na Organização - Análise de Causas e Resolução
B	- Desempenho do Processo Organizacional - Gerência Quantitativa do Projeto
C	- Análise de Decisão e Resolução - Gerência de Riscos
D	- Desenvolvimento de Requisitos - Solução Técnica - Integração do Produto - Verificação - Validação
E	-Treinamento - Definição do Processo Organizacional - Avaliação e Melhoria do Processo Organizacional - Adaptação do Processo para Gerência do Projeto
F	- Medição - Gerência de Configuração - Aquisição - Garantia da Qualidade
G (mais baixo)	- Gerência de Requisitos - Gerência do Projeto

Então a empresa que quer ter sua certificação deve começar pelo nível G, cuidando da gerência de requisitos e da gerência de projetos e depois ir subindo gradualmente seu nível de maturidade.

Uma implementação do MR-MPS pode ser conduzida por uma Instituição Implementadora (II) credenciada pela SOFTEX, o que será bem mais fácil já que essa instituição tem experiência e pode conduzir a implementação pelo caminho mais adequado.

10 TESTES DE *SOFTWARE*

Todo *software* desenvolvido de forma profissional deve levar em consideração técnicas que podem ser usadas para garantir a Qualidade de *Software* (QS), dentre essas técnicas usadas durante o processo de *software*, estão a **Validade**, a **Verificação** e os **Testes**.

A **Validade** é o processo usado para assegurarmos que o produto final corresponda aos requisitos levantados com o usuário.

A **Verificação** é o processo utilizado para assegurar consistência, completitude e corretitude do produto em cada fase e entre fases consecutivas do ciclo de vida do *software* e visa verificar se o produto está sendo construído corretamente.

Os **Testes** são as técnicas usadas para testar a execução de um programa, tentar encontrar erros, além de contribuir para aumentar a confiança de que o *software* faz as funções especificadas.

Várias técnicas, estratégias e ferramentas têm surgido no mercado tentando aprimorar e aumentar sua produtividade, os gerentes de projeto normalmente não entendem a importância desta fase e não compreendem a eficácia e a necessidade desta etapa no Processo de *Software*.

Vou apresentar algumas dessas técnicas, citar algumas ferramentas servindo de ponto de partida para os profissionais que precisam usar algum método formal para testes.

Terminologia

Nos testes de *software*, alguns termos são utilizados para diferenciar tipos diferentes de problemas, são eles:

- **Defeito:** deficiência mecânica ou algorítmica que, se ativada, pode levar a uma falha, ou seja, falha na programação.

- **Erro:** item de informação ou estado de execução inconsistente, em que o resultado esperado não é o que o sistema lhe retornou.

- **Falha:** evento em que o sistema viola suas especificações e não faz o que deveria ou faz somente uma parte do que estava documentado nos requisitos.

Técnicas de Teste

O teste de *software* e a depuração são atividades ligadas à validação de *software*. A "validação" é muito confundida com a "verificação de *software*". A verificação garante que o *software* implemente corretamente uma função específica, já a validação garante que o *software* está implementando todos os requisitos do cliente.

Nas seções a seguir, mostro as principais categorias de técnicas de teste consideradas tanto em pesquisas quanto em ferramentas comerciais. Primeiramente vou apresentar alguns critérios usados no teste estrutural de um programa, enfatizando a análise de cobertura dele. Após esta seção, serão apresentados conceitos básicos relacionados ao teste funcional, no qual o programa é visto como uma caixa-preta.

Teste Estrutural

O teste estrutural, também conhecido como teste caixa-branca (White-box) ou teste caixa-aberta, visa testar os requisitos funcionais do *software*, portanto, refere-se aos testes que são realizados nas interfaces do *software*, os casos de teste são derivados a partir da análise da estrutura interna do programa. O objetivo dos casos de teste é causar a execução de caminhos identificados no programa por critérios baseados no fluxo de controle e/ou no fluxo de dados.

Esse tipo de teste é usado principalmente para demonstrar se as funções dos *softwares* são operacionais, se a entrada é adequadamente aceita e a saída é corretamente produzida, além de verificar se a integridade das informações externas é mantida.

Os principais problemas dessa abordagem são:

- programas com laços (repetições) possuem um número infinito de caminhos, já que a análise da estrutura interna do programa é feita sobre o grafo de fluxo de controle do programa estaticamente. Dessa forma, seria necessário aplicar o teste exaustivo (teste com a submissão de todas as entradas possíveis e suas combinações), o que é considerado impraticável;

- existência de caminhos não executáveis (*infeasible paths*) no programa, os quais ocasionam o desperdício de tempo e recursos financeiros na tentativa de gerar casos de teste que possam executar esses caminhos;

- a execução bem sucedida de um caminho do programa selecionado não garante que este esteja correto, já que com outro caso de teste um erro pode ocorrer.

Tentando minimizar os problemas acima apresentados, são utilizados critérios de seleção de caminhos, que podem ser divididos em dois grupos, de acordo com as características nas quais mais se baseiam: fluxo de controle e fluxo de dados. Esses critérios, conhecidos como "critérios de cobertura" ou "critérios de seleção", são condições que devem ser preenchidas pelo teste, as quais selecionam determinados caminhos que visam cobrir o código ou a representação gráfica deste.

A seguir são apresentadas as definições dos principais critérios de cada categoria. Utilizando como base uma pequena aplicação, cujo código é apresentado abaixo, são apresentados os caminhos necessários para satisfazer os critérios baseados no fluxo de controle.

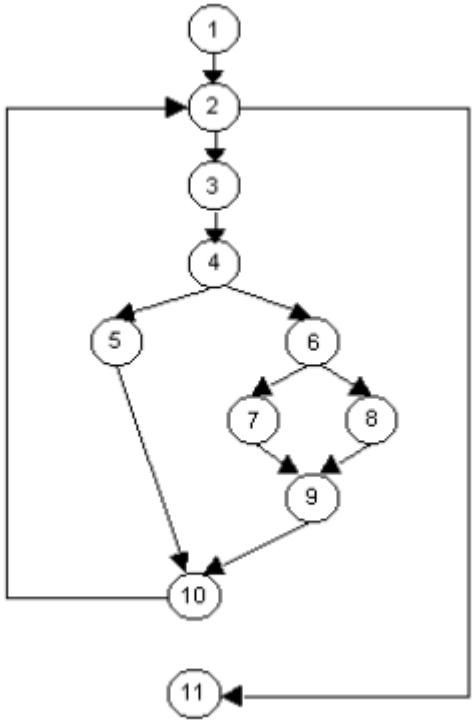
```
/*********************************************************
* PESQUISA BINARIA - Exemplo de teste de metodo *
************************************/
#include <stdlib.h>
#include <stdio.h>

int bsearch(int array[10], int value) {
   int found = 0;
   int high = 10;
   int low = 0;
   int cnt = 0;
   int mid = (high+low)/2;
   printf("\nProcurando por: %d ", value);
   while (!found && (high >= low)) {

      if (value == array[mid]) {
        found = true;
      }
      else if (value < array[mid])
        high = mid-1;
      else
        low = mid + 1;
      mid = (high+low)/2;
      cnt++;
   }
   printf("Steps %d ", cnt);
   return((found) ? mid: -1);
}
main() {
```

```
    int array[10]={10,33,40,67,100,150,200,400,1001};

    printf("Resultado %d ", bsearch(array,67));
    printf("Resultado %d ", bsearch(array,33));
    printf("Resultado %d ", bsearch(array,1));
    printf("Resultado %d ",bsearch(array,1001));
    system("pause");
}
```



Grafo de fluxo de controle do método "*bsearch*"

Os critérios de cobertura de código de unidade dividem-se em duas categorias baseados em fluxo de controle e baseados em fluxo de dados. Estas duas categorias são a seguir apresentadas.

CRITÉRIOS BASEADOS NO FLUXO DE CONTROLE

Os critérios de cobertura baseados no fluxo de controle são baseados na seleção de um conjunto *C* de caminhos no Grafo de Fluxo de Controle (GFC) do programa em teste. Um caminho em um GFC é uma seqüência de nodos *(n1, n2,...,nk)*, *k*>= 2, tal que exista um arco de *ni* para *ni+1* para *i = 1,2,...,k-1*.

Exemplos de critérios baseados no fluxo de controle da unidade são listados a seguir, sendo identificado quando estes são satisfeitos:

- COBERTURA POR NODOS (OU COMANDOS): todos os nodos/comandos devem ser executados pelo menos uma vez.
- COBERTURA POR ARCOS (OU DECISÕES): todos os arcos/decisões devem ser executados pelo menos uma vez.
- COBERTURA DE TODOS OS CAMINHOS: todos os caminhos possíveis devem ser executados pelo menos uma vez. Em programas com laços (repetições), o número de caminhos necessário para satisfazer este critério seria infinito, já que o GFC não fornece informações sobre o critério de término da iteração. Para evitar tal problema, são selecionados apenas dois caminhos relacionados a cada laço do programa: um que não executa o corpo do laço, e outro que o executa apenas uma vez. Tal restrição torna a aplicação do critério possível e pode ser visualizada no exemplo a seguir.

Com base no grafo método *bsearch*, os seguintes caminhos satisfariam os critérios anteriormente definidos:

TODOS-NODOS:
(1, 2, 3, 4, 5, 10, 2, 11)
(1, 2, 3, 4, 6, 7, 9, 10, 2, 11)
(1, 2, 3, 4, 6, 8, 9, 10, 2, 11)

TODOS-CAMINHOS (com restrição de seleção em relação ao laço):
(1, 2, 3, 4, 5, 10, 2, 11)
(1, 2, 3, 4, 6, 7, 9, 10, 2, 11)
(1, 2, 3, 4, 6, 8, 9, 10, 2, 11)
(1, 2, 11)

TODOS-ARCOS:
(1, 2, 3, 4, 5, 10, 2, 11)
(1, 2, 3, 4, 6, 7, 9, 10, 2, 11)
(1, 2, 3, 4, 6, 8, 9, 10, 2, 11)

Teste Funcional

As técnicas de teste funcional derivam os casos de teste a partir da análise da funcionalidade (dados de entrada/saída e especificação) do programa, sem levar em consideração a estrutura interna dele. O teste funcional, também conhecido como teste de caixa preta (*black box*), tem o objetivo de complementar o efeito das técnicas do teste estrutural. As categorias de erros mais evidenciadas pelo teste funcional são: erros de interface, funções incorretas ou ausentes, erros nas estruturas de dados ou no acesso a bancos de dados externos, erros de desempenho e erros de inicialização e término.

O teste funcional é geralmente aplicado quando todo ou quase todo o sistema já foi desenvolvido. Entre as técnicas mais clássicas de teste funcional podem ser citadas as seguintes:

- grafo causa-efeito, no qual são representadas as principais entradas do programa, suas combinações e saídas correspondentes geradas; particionamento de equivalência, em que o domínio dos valores de entrada é dividido em classes de equivalência, reduzindo, assim, os dados de entrada a serem submetidos ao programa, já que todos os dados de uma mesma classe supostamente teriam um comportamento equivalente;

- a análise do valor-limite, em que além de dividir o domínio dos valores de entrada em classes de equivalência, procura escolher dados localizados nos limites dessas classes, aumentando, assim, a probabilidade de identificação de erros.

11 ESTRATÉGIAS DE TESTE DE *SOFTWARE*

As técnicas de teste devem ser utilizadas em conjunto, organizadas em estratégias de teste, por meio das quais se pode estabelecer como, em que ordem e quem realizará cada tarefa. De acordo com Pressman [21], “uma estratégia de teste de *software* integra técnicas de projeto de casos de teste numa série bem-definida de passos que resultam na construção bem-sucedida de *software*”.

Em uma situação ideal, o sistema de *software* deve ser testado por pessoas diferentes daquelas que o programaram. Entretanto, considerando a divisão das tarefas de teste em quatro níveis relacionados ao escopo do *software* sendo testado, tem-se a seguinte distribuição de responsabilidades:

- TESTE DE UNIDADE: geralmente feito pelo próprio programador, pois exige um ambiente de produção adequado e especializado para acompanhar os testes realizados;
- TESTE DE INTEGRAÇÃO: pode ser feito por vários programadores, de forma que os integrantes do grupo, que não participaram do desenvolvimento dos módulos, participem da atividade de teste como testadores ativos, e os programadores dos módulos como testadores de suporte, que auxiliarão os testadores ativos na prestação de informações que eventualmente sejam necessárias;
- TESTE DE SISTEMA: o teste de sistema deve ser feito, preferencialmente, por pessoas que não estiveram ligadas diretamente ao desenvolvimento do sistema. Pode-se utilizar, neste momento, usuários selecionados, assim como analistas de suporte de outros projetos, atendentes de telefone que registram pedidos de alterações do sistema, entre outros;
- TESTE DE ACEITAÇÃO: tipo de teste que deve ser feito, obrigatoriamente, por usuários. Exemplos de teste desta categoria são: teste *alfa* e teste *beta*.

Teste de Integração

O teste de integração dos módulos individualmente testados no teste de unidade seria o primeiro momento no qual pode existir cooperação na atividade de teste. É interessante haver, neste momento, a presença de “testadores ativos” e “testadores de suporte”. O programador da classe assume o papel de testador de suporte, trabalhando como um profissional de suporte ao testador ativo.

Os testadores de suporte devem ajudar na determinação da ordem de integração dos módulos. A integração dos módulos deve ser realizada de forma incremental. A abordagem de integração não-incremental não é recomendada, já que o escopo de código a ser considerado na ocorrência de um erro é bastante grande, tornando-se difícil localizá-lo. Por isso, neste tutorial, somente a abordagem incremental será abordada.

Na abordagem incremental, devem ser construídos módulos de apoio, chamados *drivers* e *stubs*. Um *driver* é um módulo que chama o(s) módulo(s) sendo testado(s), tendo em seu corpo apenas inicializações de variáveis globais, chamadas de rotinas, e inicializações dos parâmetros necessários. Um *stub* é um módulo que é chamado pelo(s) módulo(s) sendo testado(s), contendo em seu corpo apenas a atribuição de valores que serão retornados, quando for necessário. No teste incremental, a necessidade de utilização de *drivers* e/ou *stubs* é determinada por meio da estratégia utilizada. As duas estratégias mais utilizadas são a *topdown* e a *bottom-up*.

Na estratégia *top-down*, a integração inicia com o módulo mais do topo (o módulo inicial) do sistema. Depois disso, devem ser selecionados módulos considerando-se que seu módulo superior (o módulo chamador) tenha sido bem integrado. Para o teste de um módulo superior, é necessária a utilização de *stubs*. Os módulos *stubs* não devem conter apenas mensagens indicando que o fluxo de controle passou por eles, mas sim retornos de valores ou a realização de funções específicas, sempre

que necessário. Caso haja um erro na construção dos *stubs*, poderá haver uma falha no sistema, e a detecção dela implicará tempo e custos perdidos.

Um dos problemas com a estratégia *top-down* é que o fornecimento de valores geralmente não é feito de forma direta no momento inicial, já que módulos que contêm funções de entrada e/ou saída costumeiramente estão localizados na base do diagrama de chamadas.

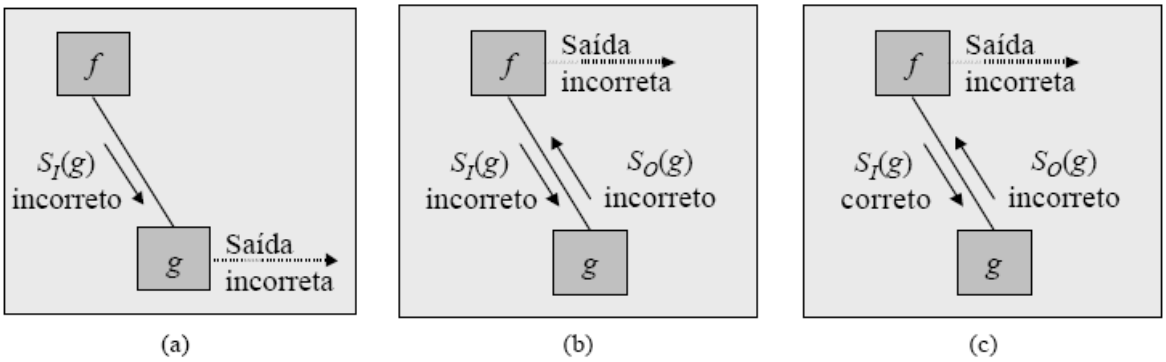
Dessa forma, os valores de entrada necessários aos módulos sendo testados devem ser fornecidos pelos *stubs*. Para um conjunto variado de valores de entrada, deve-se editar um *stub* atribuindo novos valores em suas inicializações, ou então construí-lo de tal forma que sua estrutura permita diretamente a seleção de valores para cada execução. Assim que um conjunto de módulos com seus *stubs* é testado, cada um dos *stubs* é substituído pelo módulo real correspondente, de forma que o teste de integração termine quando o sistema todo estiver integrado e tiver as interfaces entre seus módulos bem testadas.

A seqüência por meio da qual são integrados os módulos pode variar, dependendo do critério adotado para tal. Podem ser escolhidos primeiro os módulos mais críticos, ou os que possuem funções de entrada e/ou saída ou ainda aleatoriamente. Essa decisão vai influenciar no esforço necessário para a codificação dos *stubs*. Um problema que deve ser evitado é a passagem para o teste de um outro módulo sem completar a integração do anterior.

A estratégia *bottom-up*, por sua vez, inicia selecionando-se os módulos da base do diagrama de chamadas. Para que o teste seja realizado, não é necessária a construção de módulos *stubs*, mas sim de *drivers*, para que os módulos reais do sistema sejam chamados. Os *drivers* devem conter inicializações de variáveis globais e de variáveis passadas como parâmetros nas chamadas aos módulos. Assim, podem ser construídas várias versões de um módulo *driver*, para permitir a submissão de vários conjuntos de valores, bem como construí-lo de maneira que

essa variação seja feita de forma automática (como citado anteriormente com os módulos *stubs*).

Um dos grandes problemas da estratégia *bottom-up* é que não é construído, inicialmente, um esqueleto do programa. O programa funcional passa a existir somente quando o último módulo é integrado. A vantagem sobre a estratégia *top-down* é que as funções de entrada e/ou saída são geralmente integradas no início do teste, fazendo com que não sejam necessárias tantas versões de *drivers* quanto de *stubs* na estratégia *top-down*.



Tipos principais de erros de Teste de Integração

11.1 Teste de Aceitação

O teste de aceitação corresponde ao teste de sistema, que deve ser realizado após os testes de unidade e de integração. De acordo com Pressman, a "validação (fase de aceitação) é bem-sucedida quando o *software* funciona de uma maneira razoavelmente esperada pelo cliente". As expectativas do cliente devem estar registradas em um documento de especificação, o qual deve ter sido escrito no início do desenvolvimento do sistema.

Nesta etapa, é fundamental a especificação de um plano de teste. Esse plano pode ter o formato simples de um *checklist*, ou então ser formado por uma lista de procedimentos a serem realizados. Em ambos os casos, o plano deve abranger todos os requisitos funcionais e de desempenho do sistema.
Além disso, a documentação do usuário também deve ser utilizada neste momento. Os manuais e sistemas de ajuda (*on-line* ou impressos) devem estar de acordo com a funcionalidade do *software*. Também a inclusão de tutoriais e exemplos de complexidade média de uso do sistema são bastante úteis para que o usuário entenda como o sistema funciona. Para haver consistência entre os documentos e o próprio sistema de *software*, é interessante que seja realizada a revisão da configuração do *software*, em paralelo com o teste de aceitação.

Após cada caso de teste de validação ter sido realizado, existirá uma das duas condições a seguir: (1) os requisitos funcionais e de desempenho conformam-se à especificação, ou (2) um desvio das especificações é descoberto e uma lista de deficiências é criada.

11.2 Teste de Sistema

O teste de sistema tem o objetivo principal de pôr à prova o sistema completo de *software*. Ou seja, após a realização dos testes de unidade, de integração e de validação, o sistema é testado em conjunto com outros sistemas de *software* e elementos de *hardware*. Dessa forma, os seguintes tipos de teste de sistema são discutidos nas próximas subseções: teste de segurança, teste de estresse e teste de desempenho.

11.3 Teste de Segurança

O teste de segurança tem o objetivo de garantir que o sistema se comporte adequadamente mediante tentativas ilegais de acesso, tais como as que são

comumente feitas por *hackers*. Os mecanismos de segurança implementados no sistema devem protegê-lo realmente desses acessos indevidos.

Esta estratégia de teste deve ser aplicada por programadores que não desenvolveram o *software*. Para isto, os testadores devem tentar penetrar no sistema de várias formas, tais como por meio da obtenção ilegal de senhas, desarme do sistema, tentativas de acesso durante a recuperação do sistema após inserção de falha intencional, entre outros. É importante que o teste não seja feito pelos próprios desenvolvedores, a fim de evitar que estes testem apenas os mecanismos de segurança implementados, os quais são de seu próprio conhecimento.

11.4 Teste de Estresse

O teste de estresse deve confrontar o sistema com situações anormais de uso. A pergunta que deve ser feita pelo testador é: “até que ponto posso aumentar tal recurso antes que falhe?”

Para responder a essa pergunta, o testador deve utilizar o sistema com recursos em quantidade, freqüência e volume anormais, tais como procura excessiva de dados em disco, aumento excessivo de índices de dados, abertura de inúmeras janelas, até que haja um problema de falta de memória, utilização de arquivos com formato não compatível aos esperados pelo programa, entre outros.
O teste de estresse também deve ser feito por programadores que não desenvolveram o sistema. Pode ser utilizado um *checklist* com situações padrão de provocação de “estresse no sistema”.

11.5 Teste de Desempenho

O teste de desempenho é fundamental para sistemas de tempo real. Nesta etapa, deve-se verificar se o desempenho do *software* está de acordo com seus requisitos

especificados, no qual o desempenho de execução do *software* é testado, dentro do contexto de um sistema integrado.

O teste de desempenho deve ser feito combinado com o teste de estresse. Nessas situações, *software* e *hardware* são controlados, a fim de observar seu comportamento perante as situações nas quais foram colocados. O teste de desempenho deve ser feito por outros programadores que não sejam os que desenvolveram o sistema. Entretanto, é interessante que o próprio programador atue como um “testador de suporte”, a fim de fornecer informações que possam ser necessárias.

- **REFERÊNCIAS**

Pressman

Wilson de Pádua

O´Brien SI

ES Pedrycz

http://pt.wikipedia.org/wiki/Programa_de_computador

http://pt.wikipedia.org/wiki/engenharia_de_software

http://www.barreto.com.br/qualidade