**CURSO DE PÓS-GRADUAÇÃO**
**"LATO SENSU" (ESPECIALIZAÇÃO) À DISTÂNCIA**
**MELHORIA DE PROCESSO DE SOFTWARE**

# INTRODUÇÃO À ENGENHARIA DE SOFTWARE E À QUALIDADE DE SOFTWARE

**Alexandre Marcos Lins de Vasconcelos**
**Ana Cristina Rouiller**
**Cristina Ângela Filipak Machado**
**Teresa Maria Maciel de Medeiros**

**Universidade Federal de Lavras - UFLA**
**Fundação de Apoio ao Ensino, Pesquisa e Extensão - FAEPE**
**Lavras - MG**

**Parceria**

Universidade Federal de Lavras - **UFLA**

Fundação de Apoio ao Ensino, Pesquisa e Extensão - **FAEPE**

**Reitor**

Antônio Nazareno Guimarães Mendes

**Vice-Reitor**

Ricardo Pereira Reis

**Diretor da Editora**

Marco Antônio Rezende Alvarenga

**Pró-Reitor de Pós-Graduação**

Joel Augusto Muniz

**Pró-Reitor Adjunto de Pós-Graduação "Lato Sensu"**

Marcelo Silva de Oliveira

**Coordenação do curso**

Ana Cristina Rouiller

**Presidente do Conselho Deliberativo da FAEPE**

Edson Ampélio Pozza

**Editoração**

Quality Group

**Impressão**

Gráfica Universitária/UFLA

**Ficha Catalográfica Preparada pela Divisão de Processos Técnicos da Biblioteca Central da UFLA**

Introdução à Engenharia de Software e à Qualidade de Software / Alexandre Marcos Lins de Vasconcelos, Ana Cristina Rouiller , Cristina Ângela Filipak Machado, Teresa Maria Maciel de Medeiros. – Lavras: UFLA/FAEPE, 2006.

157 p. :il. – (Curso de Pós-graduação "Lato Sensu" (Especialização) à Distância – Melhoria de Processo de Software.

Bibliografia.

1. Engenharia de software. 2. Qualidade. 3. Projeto. 4. Gerenciamento. 5. Desenvolvimento. I. Vasconcelos, A. M. L. de. II. Universidade Federal de Lavras. III. Fundação de Apoio ao Ensino, Pesquisa e Extensão. IV Título.

CDD-005.1

# SUMÁRIO

# LISTA DE FIGURAS

# LISTA DE TABELAS

# 1

# INTRODUÇÃO

Atualmente, há cada vez mais sistemas controlados por software, fazendo com que a economia de praticamente todos os países seja muito dependente da qualidade dos softwares por eles usados, justificando um investimento significativo nesse setor.

Há alguns anos atrás, desenvolvia-se software de uma maneira completamente artesanal. A partir de uma simples definição dos requisitos do software, partia-se imediatamente para a implementação do mesmo. Hoje em dia, ainda há muitas empresas que desenvolvem software dessa maneira, mas várias outras estão mudando suas formas de trabalho.

A forma artesanal de trabalho, geralmente, não traz grandes problemas para o desenvolvimento de software de pequeno porte, o qual não exige um esforço muito grande de implementação. Porém, para softwares de grande porte, sérios problemas na implementação podem comprometer todo o projeto.

Com o desenvolvimento cada vez maior da tecnologia de hardware e a conseqüente disponibilidade de máquinas cada vez mais potentes e baratas, o uso de computadores tem-se tornado cada vez mais difundido em diversas áreas. Isso tem feito com que aumente a demanda por software cada vez maior e mais complexo. No entanto, a demanda por software tem-se tornado maior que a capacidade do mercado para atendê-la.

Muitos projetos são entregues com um grande atraso, custando muito mais que o inicialmente previsto, sendo não confiáveis, difíceis de manter e/ou não tendo um desempenho satisfatório. Além do mais, na tentativa de se consertar os erros, muitas vezes introduzem-se mais erros. Geralmente, a quantidade de problemas é diretamente proporcional ao aumento da complexidade do software produzido nos dias de hoje. Esses problemas no desenvolvimento de software são conhecidos mundialmente como a “crise de software”. Ou seja, a “crise de software” corresponde à incapacidade da indústria de software de atender prontamente à demanda do mercado de software, dentro dos custos e dos níveis de qualidade esperados.

Desde os anos 1960, quando o termo "*crise de software*" foi pronunciado pela primeira vez, muitos problemas desta área foram identificados e muitos deles ainda persistem até os dias de hoje, tais como [Gibbs1994]:

- Previsão pobre – desenvolvedores não prevêem adequadamente quanto tempo e esforço serão necessários para produzir um sistema de software que satisfaça às necessidades (requisitos) dos clientes/usuários. Sistemas de software são geralmente entregues muito tempo depois do que fora planejado;
- Programas de baixa qualidade – programas de software não executam o que o cliente deseja, conseqüência talvez da pressa para se entregar o produto. Os requisitos originais podem não ter sido completamente especificados, ou podem apresentar contradições, e isto pode ser descoberto muito tarde durante o processo de desenvolvimento;
- Alto custo para manutenção – quando o sistema é construído sem uma arquitetura clara e visível, a sua manutenção pode ser muito custosa;
- Duplicação de esforços – é difícil compartilhar soluções ou reusar código, em função das características de algumas linguagens adotadas, por falta de confiança no código feito por outra pessoa e até mesmo pela ausência/deficiência de documentação das rotinas e dos procedimentos já construídos.

O reconhecimento da existência da crise de software tem provocado uma forte mudança na forma de como as pessoas desenvolvem software de grande porte, visto que o processo de desenvolvimento atual é mais disciplinado do que no passado. Foi proposto que o desenvolvimento de software deixasse de ser puramente artesanal e passasse a ser baseado em princípios de Engenharia, ou seja, seguindo um enfoque estruturado e metódico. Assim, surgiu o termo Engenharia de Software, que se refere ao desenvolvimento de software por grupos de pessoas, usando princípios de engenharia e englobando aspectos técnicos e não-técnicos, de modo a produzir software de qualidade, de forma eficaz e dentro de custos aceitáveis.

Portanto, a Engenharia de Software engloba não apenas o desenvolvimento de programas, mas também toda a documentação necessária para o desenvolvimento, instalação, uso e manutenção dos programas. O temo "ciclo de vida de software" compreende todas as etapas, desde a concepção inicial do software, até a sua implementação, implantação, uso e manutenção, de modo que, ao final de cada uma destas etapas, um ou mais documentos são produzidos.

Engenheiros de software devem adotar uma abordagem sistemática e organizada para seu trabalho e usar ferramentas e técnicas apropriadas, dependendo do problema a ser solucionado, das restrições de desenvolvimento e dos recursos disponíveis. Além das técnicas de especificação e implementação de software, os engenheiros de software devem ter conhecimento também de outras técnicas como,

por exemplo, de gerenciamento de software. Dessa forma, aumenta-se a probabilidade de produzir software de grande porte com qualidade, ou seja, software que satisfaça os requisitos do usuário, bem como as expectativas de tempo e de orçamento.

As ODSs (Organizações de Desenvolvimento de Software)[1], com o intuito de minimizar os problemas do desenvolvimento do software, têm geralmente adotado metodologias de desenvolvimento de software. Todavia, os paradigmas metodológicos para desenvolvimento de software têm sido considerados somente em termos de "um método" de análise/projeto/implementação, isto é, um conjunto de conceitos, técnicas e notações. Essa visão elimina os aspectos tecnológicos, contextuais e organizacionais que potencialmente existem dentro de um processo de software.

Os ambientes tradicionais das ODSs geralmente suportam apenas a engenharia do produto, assumindo um processo implícito e tendo como foco principal o produto. Essa visão tem limitado as ODSs no que diz respeito à tomada de decisões, ao estabelecimento e arquivamento de metas organizacionais, à determinação de pontos para melhoria, à estipulação de prazos para entrega de produtos e à obtenção de uma certificação. O capítulo 2 apresenta os aspectos evolutivos das ODSs e seus produtos de software.

De forma geral, pode-se dividir as funções de uma ODS em três grupos principais [Garg1996][2]:

1. Definir, analisar, simular, medir e melhorar os processos da organização;
2. Donstruir o produto de software;
3. Medir, controlar, modificar e gerenciar os projetos de software.

Estes três grupos são abordados, respectivamente, pela Engenharia de Processo, pela Engenharia de Produto e pelo Gerenciamento de Projeto. O relacionamento entre estes grupos é mostrado na Figura 1.1.

[1] Uma ODS representa uma organização independente, ou um departamento ou uma unidade dentro de uma organização, que é responsável por desenvolver, manter, oferecer ou operar um produto ou serviço de software ou um sistema de software intensivo [Wang1999].

[2] A Engenharia de Software deve considerar estas funções objetivando a produção de software de maior qualidade e a melhoria do desempenho da ODS, ou seja, torná-la mais produtiva.

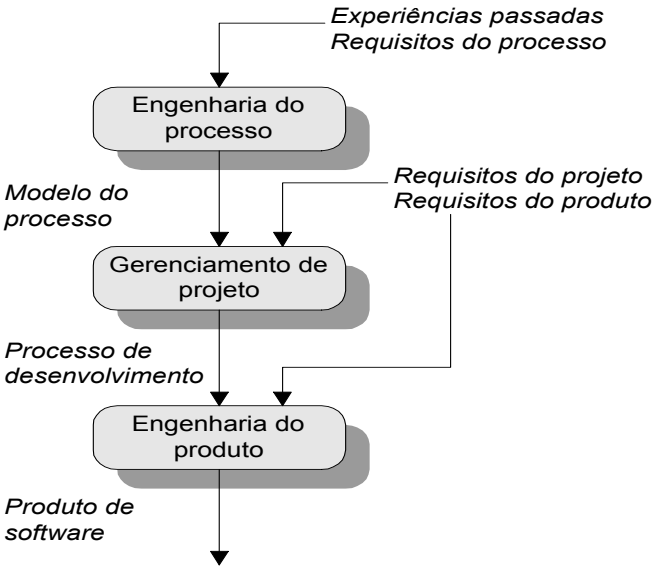


**Figura 1.1: Relacionamento entre engenharia de processo, gerenciamento de projeto e engenharia do produto**

A *engenharia de processo* tem como meta a definição e a manutenção dos processos da ODS. Ela deve ser capaz de facilitar a definição, a análise e a simulação de um processo, assim como estar apta a implantá-lo, avaliá-lo, medi-lo e melhorá-lo. A engenharia de processo trata os processos de software de uma forma sistemática com um ciclo de vida bem definido. O capítulo 6 aborda este tema discorrendo sobre qualidade de software, um tema cada vez mais relevante e que engloba avaliação e melhoria contínua dos processos da ODS.

O *gerenciamento de projeto* tem o objetivo de assegurar que processos particulares sejam seguidos, coordenando e monitorando as atividades da engenharia do produto. Um processo de gerenciamento de projeto deve identificar, estabelecer, coordenar e monitorar as atividades, as tarefas e os recursos necessários para um projeto produzir um produto e/ou serviço de acordo com seus requisitos. Todavia, gerenciar projetos de software é uma atividade complexa devido a uma série de fatores, tais como: dinamicidade do processo, grande número de variáveis envolvidas, exigência de adaptabilidade ao ambiente de desenvolvimento e constantes alterações no que foi planejado. Esses fatores dificultam o trabalho das equipes de desenvolvimento na medição do desempenho dos projetos, na verificação de pontos falhos, no registro de problemas, na realização de estimativas e planejamentos adequados. O capítulo 4 aborda esse tema.

A *engenharia do produto* é encarregada do desenvolvimento e manutenção dos produtos e serviços de software. A principal figura da engenharia do produto é a metodologia de desenvolvimento, que engloba uma linguagem de representação, um

modelo de ciclo de vida e um conjunto de técnicas. Os ambientes tradicionais de desenvolvimento de software têm se preocupado essencialmente com a engenharia do produto, assumindo um processo implícito e tendo como foco o produto. Todavia, a engenharia do produto por si só é insuficiente para suprir as necessidades da ODS e torná-la mais produtiva e adequada às exigências do mercado. O capítulo 3 aborda os modelos de ciclo de vida mais utilizados na engenharia do produto. O capítulo 5 descreve as principais atividades da engenharia de produto.

# 2

# O PRODUTO DE SOFTWARE E A ORGANIZAÇÃO DE DESENVOLVIMENTO

Impulsionados pelas mudanças tecnológicas e pelo amadurecimento das atividades de desenvolvimento de software os produtos, as organizações de desenvolvimento (ODSs) e seus processos associados mudaram no decorrer das últimas décadas. Este capítulo faz uma retrospectiva desses elementos.

A seção 2.1 e 2.2 abordam, respectivamente, a evolução dos produtos de software e a evolução das organizações de desenvolvimento. A seção 2.3 apresenta os processos existentes em uma ODS, demonstrando modelos e elucidando conceitos.

## 2.1 O PRODUTO DE SOFTWARE

Tem-se observado muitas mudanças nos produtos de software que, nos últimos anos, surgem em prateleiras de supermercados ou mesmo disponíveis gratuitamente na Web. Conseqüência do aperfeiçoamento tecnológico e da maturidade no desenvolvimento de software adquiridos no decorrer dos anos.

Anterior à revolução gerada pelo surgimento dos PCs (*Personal Computers*), o software[3], geralmente, era confeccionado dentro da empresa que iria utilizá-lo. Os detalhes do negócio então eram do conhecimento dos desenvolvedores. O software também era monolítico e específico para "rodar" na plataforma da empresa, considerando o seu ambiente e os seus processos. De responsabilidade organizacional e não contratual, se o software executasse as funções a que se propunha, geralmente, já satisfazia os seus usuários/clientes.

O produto de software hoje confeccionado pelas ODSs possui características diferenciadas, desde a sua especificação até a entrega. Em primeiro lugar, ele deve

[3] Utilizaremos os termos "software" e "sistema" como sinônimos, apesar do último ser mais abrangente.

ser o mais geral, flexível e parametrizável possível. Deve estar apto a “rodar” em empresas diferentes, que possuam inclusive processos de negócio também diferentes. O usuário, que determinava os requisitos do software, passou muitas vezes até mesmo a não existir, exigindo que a equipe de desenvolvimento se adaptasse e passasse a buscar o conhecimento de outras formas. Muitas sub-áreas da Engenharia de Software surgiram (ou foram reforçadas) para satisfazer esses novos quesitos como, por exemplo, Engenharia do Conhecimento, Engenharia de Requisitos, Arquitetura de Software, Sistemas Distribuídos.

Alguns fatores de qualidade passaram a ser exigidos para o produto de software. Entre esses fatores, podem-se citar os externos como usabilidade, portabilidade, manutenibilidade etc. e fatores de qualidade internos como reusabilidade, modularidade, flexibilidade etc. Uma constante melhoria no processo que desenvolve o produto também passou a ter relevância, pois um processo de software de alta qualidade deve, por conseqüência, gerar um produto de alta qualidade. Pode-se dizer que qualidade é uma das palavras chaves na Engenharia de Software e é medida atualmente de duas formas: (1) qualidade do produto e (2) qualidade do processo. No capítulo 6 serão detalhados os padrões que avaliam e propõem melhoria da qualidade.

Outro aspecto interessante deste novo software que estamos produzindo é o fato de gerar uma demanda muitas vezes ainda não existente. Podem-se citar vários softwares que surgiram desta forma, como: o Windows, o Netscape, o Word, o Yahoo e os RPGs.

A Figura 2.1 sintetiza algumas diferenças entre o produto de software que produzíamos nos primórdios da computação e os que hoje são desenvolvidos.

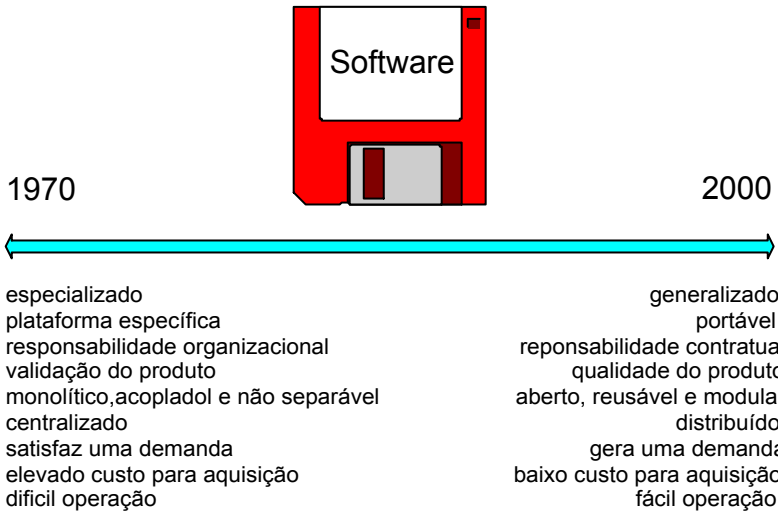


**Figura 2.1: Diferenças do produto de software**

## 2.2 A ORGANIZAÇÃO DE DESENVOLVIMENTO DE SOFTWARE

É natural que um produto confeccionado com características tão diferentes das iniciais também fosse gerado por uma ODS diferente. Nesta seção é visto o quanto as organizações mudaram para satisfazer as necessidades da confecção dos produtos de software e se adequarem às novas tecnologias e tendências mercadológicas.

### 2.2.1 Núcleo de Processamento de Dados

Os primeiros tipos de computadores produzidos foram os *mainframes*, equipamentos caros e de difícil manutenção e operação. Uma empresa que resolvesse adquirir um equipamento deste porte sofria a imediata conseqüência de necessitar contratar, também, uma grande equipe para produzir e operar os softwares (tanto os desenvolvidos por ela, quanto os de suporte ao desenvolvimento).

Os contratos de venda, aluguel e manutenção desses equipamentos ultrapassavam as cifras de milhares de dólares. Somente em grandes empresas, com um bom faturamento e em grandes universidades, observávamos o computador. As equipes dentro dos centros de desenvolvimento de software eram grandes e era requerida muita especialização por parte de quem desenvolvia software.

As ODSs (nomeadas de núcleos ou centros de processamento de dados - CPDs ou NPDs) eram compostas, salvo algumas exceções, por setores agrupados baseados nas tarefas (funções) que cada um desempenhava em relação ao desenvolvimento e operação do software. A Figura 2.2 mostra um exemplo de um organograma de um NPD.

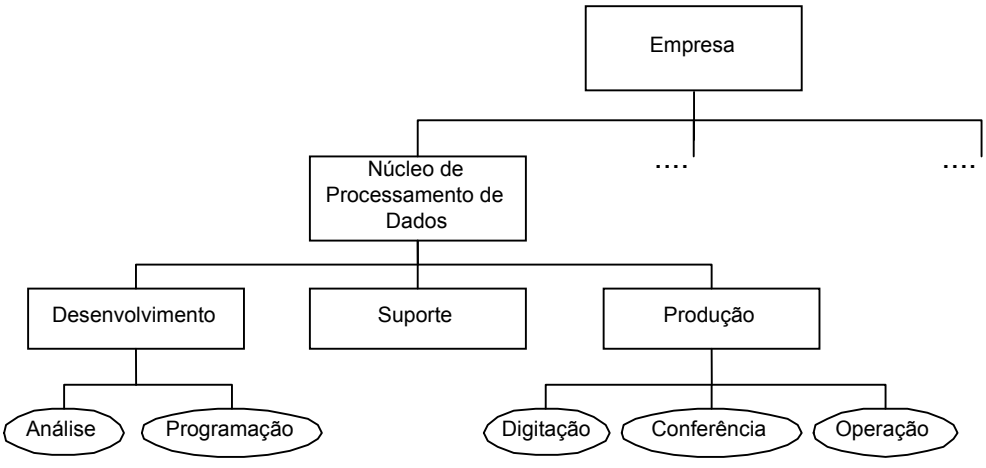


**Figura 2.2: Exemplo da estrutura de um NPD**

Na divisão de desenvolvimento se concentrava a confecção do produto de software. O setor de análise levantava informações junto aos usuários e *especificava*[4] logicamente e fisicamente o software, enquanto o setor de programação codificava as especificações definidas pelos analistas de sistemas. Dentro do setor de programação poderíamos observar duas equipes distintas: equipe de desenvolvimento (encarregada dos novos programas) e a equipe de manutenção (encarregada de manter os programas já desenvolvidos e em produção).

A *divisão de suporte* possuía duas características principais. A primeira era manter em funcionamento os equipamentos e os softwares, instalando e configurando, além de ter contínua preocupação com o desempenho. A segunda característica era promover a capacitação do pessoal do NPD. Isso implicava em estudo de novas tecnologias e treinamento apropriado. Devido à complexidade do gerenciamento dos *mainframes*, a divisão de suporte requeria pessoal altamente especializado. Dar suporte à operação do software e a seu desenvolvimento não era tarefa fácil.

A *divisão de produção e*ra responsável por executar, obter os resultados e implantar as informações. O *setor de digitação* realizava a digitação dos documentos ou planilhas que vinham do usuário, além dos programas do setor de programação. Devido ao pouco acesso que as pessoas tinham aos computadores, os usuários preenchiam planilhas com as informações a serem armazenadas nos computadores para posterior processamento. *O setor de conferência* conferia se havia erros nos dados digitados, se os resultados produzidos pelo processamento estavam corretos etc. Muitos artifícios eram utilizados para garantir a digitação correta dos dados, entre eles o “total de lote” que representava uma totalização dos valores digitados para posterior conferência. As execuções eram solicitadas pelos usuários ou tinham datas pré-determinadas. Essas execuções eram realizadas no *setor de operação* que também administrava o hardware.

A documentação de um sistema era um trabalho muito árduo e cansativo. A utilização de máquinas de datilografia, pastas mantidas manualmente etc., traziam um custo muito elevado para se manter o sistema atualizado. Por isso, alguns NPDs possuíam um *setor de documentação* que era encarregado de realizar todas as alterações feitas manualmente por analistas, programadores e operadores.

A qualidade do produto e do processo que o confeccionava era uma preocupação constante nos NPDs. Todavia, a Engenharia de Software ainda engatinhava em seus conceitos e não havia maturidade em relação a padrões de qualidade do produto e do processo de software. O alto custo do “homem especializado em computação” e de “hora máquina” obrigava também estas organizações a medirem seu processo. Em

[4] Especificar um software neste contexto significa criar um modelo computacional, independente da plataforma computacional que será utilizada.

suma, os NPDs primavam por usar uma metodologia de desenvolvimento, documentar os sistemas, medir as atividades pessoais e dar um custo para cada tarefa desenvolvida.

Os pontos apontados como falhos para esta estrutura estão mais ligados à tecnologia empregada do que à estrutura propriamente dita. Com a saída do desenvolvimento dos NPDs para pequenas equipes de desenvolvimento em empresas específicas, houve uma perda da qualidade do processo e do produto. Inclusive tornando a computação desacreditada perante aqueles que necessitavam e/ou pretendiam desenvolver um software.

### 2.2.2 Pequeno Centro de Desenvolvimento

A evolução do hardware e software mudou significativamente o processo de desenvolvimento e a estrutura das organizações. Com a chegada e barateamento dos PCs, muitas empresas de pequeno e médio porte puderam adquirir computadores e contratar pequenas equipes para automatizar seus processos. Assim, as funções realizadas de formas distintas dentro de um NPD começaram a ser fundidas no ambiente de desenvolvimento e produção. Figuras pejorativas (e que posteriormente passaram até mesmo a incorporar carreiras organizacionais) surgiram como o programalista (programador + analista) e o anador (analista + programador).

Sistemas como folha de pagamento, contas a pagar, contabilidade, controle de estoque, entre outros, invadiram as pequenas e médias empresas. A função “super-valorizada” de quem produzia software tornou-se algo tão comum como um escriturário ou um contador. A Figura 2.3 ilustra um exemplo de um pequeno centro de desenvolvimento dentro de uma empresa de pequeno ou médio porte.

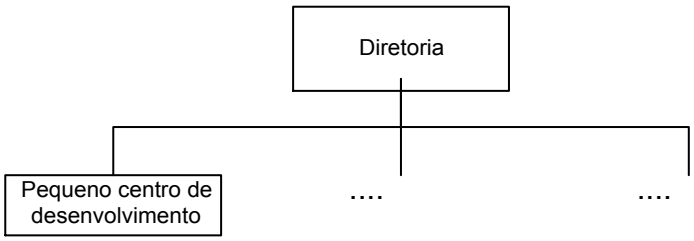


**Figura 2.3: Pequeno centro de desenvolvimento**

Todavia, qual foi o problema destes pequenos centros de desenvolvimento? Em primeiro lugar tinha-se um cliente (usuário) alheio às dificuldades do desenvolvimento de software, que acreditava que qualquer programador resolveria a automação de sua empresa. E em segundo lugar, um grupo de desenvolvimento imaturo metodologicamente e, em sua maioria, descompromissado com o futuro do produto

que confeccionavam. Esses dois pontos trouxeram diversos problemas para a informática. Muitas empresas, em pouco tempo, se viram à mercê de um software inoperável ou de difícil/impossível manutenção. Poucos empresários passaram a confiar em quem produzia software para solucionar os problemas ou melhorar a produtividade de seu negócio. Criou-se uma lacuna entre quem precisava do software e quem o produzia.

### 2.2.3 Fábrica de Software

Com o intuito de preencher esta lacuna, alguns centros de desenvolvimento de software foram montados como empresas apartadas do cliente/usuário. É bom deixar claro que as fábricas de software não surgiram em decorrência de uma idéia, mas sim de bons desenvolvedores que passaram a oferecer sua solução de software para diversas empresas, assegurando uma continuidade do produto que desenvolviam. Isso possibilitava que o comprador do software tivesse uma garantia contratual de manutenção e evolução do produto. Apesar dos sistemas perderem em especificidade, começava a despontar uma solução de software barato (pois era vendido para diversas empresas) e de melhor qualidade.

Atualmente, as fábricas de software são realidades e se tornaram muito mais complexas do que poderíamos imaginar. Além de possuírem as funções de desenvolvimento que existiam nos NPDs, somaram a si funções de negócio e administrativas, algumas até possuindo departamentos especializados em pesquisa (ver Figura 2.4). Trabalham, muitas vezes, dispersas em áreas geográficas diferentes e sub-contratam serviços. São impulsionadas e avaliadas pelos mesmos quesitos de qualquer outra indústria: qualidade do produto e produtividade.

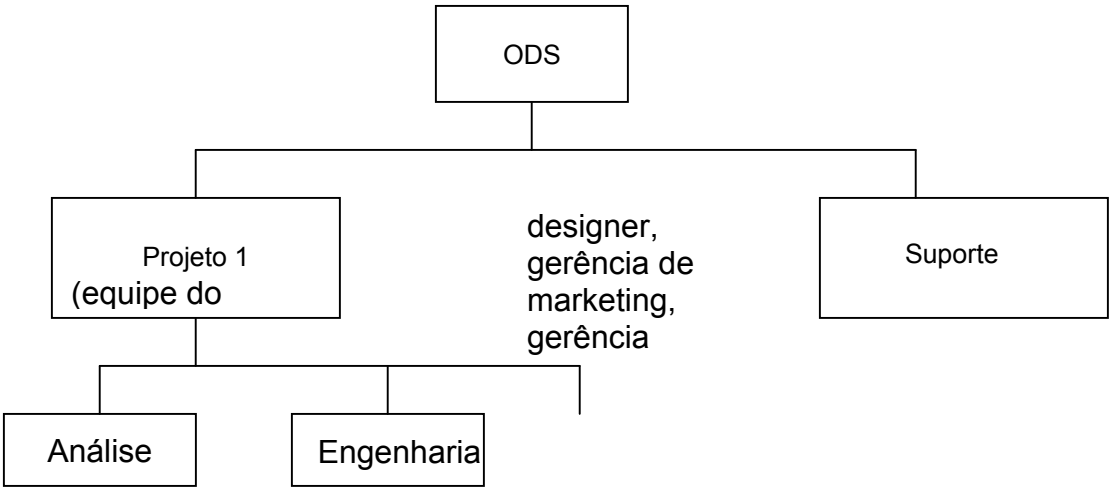


**Figura 2.4: Organograma de uma fábrica de software**

A Figura 2.5 caracteriza algumas diferenças entre a organização que produzia software na década de 1970 e a organização que atualmente desenvolve software. É lógico que nem todas as organizações se encaixam em uma ou outra ponta. Quando se fala, por exemplo, que no ano 2000 as organizações estão fora da empresa que necessita do software, não se está ditando nenhuma regra absurda, apenas trata-se do mais comumente encontrado e de uma tendência que é vislumbrada.


dentro da empresa que necessida do software
pequenas equipes locais
modela funcionalidades e necessidades da empresa
processo de software ad-hoc
lida com responsabilidade organizacional
utilização de metodologias + ciclo de vida
planejamento precário
baixo rodízio de pessoal
produz software caro

fora da empresa
grandes equipes dispersas
exigencia de qualidade
lida com responsabilidade contratua
processos software gerenciado
processos de desenvolvimento complexos
planejamento baseado em estimativa
alto rodízio de pessa
produz software barato

**Figura 2.5: NPDs versus fábricas de software**

A próxima seção apresenta aspectos referentes aos processos utilizados pelas ODSs para produzir software.

## 2.3 OS PROCESSOS DA ODS

Um processo pode ser definido como um conjunto de atividades inter-relacionadas que transformam um conjunto de entradas em resultados [ISO12207:95]. Segundo a ISO15504 [ISO15504:1-9:98] processo de software é um conjunto de processos utilizados por uma ODS ou um projeto de software para planejar, gerenciar, executar, monitorar, controlar e melhorar as atividades que estão relacionadas com software.

O trabalho de Wang [Wang99] apresenta um *framework* unificado de um sistema de processo para uma ODS. A Figura 2.6 mostra este modelo, que está dividido em três agrupamentos principais: o *modelo do processo*, o *modelo de avaliação do processo* e o *modelo de melhoria do processo*.

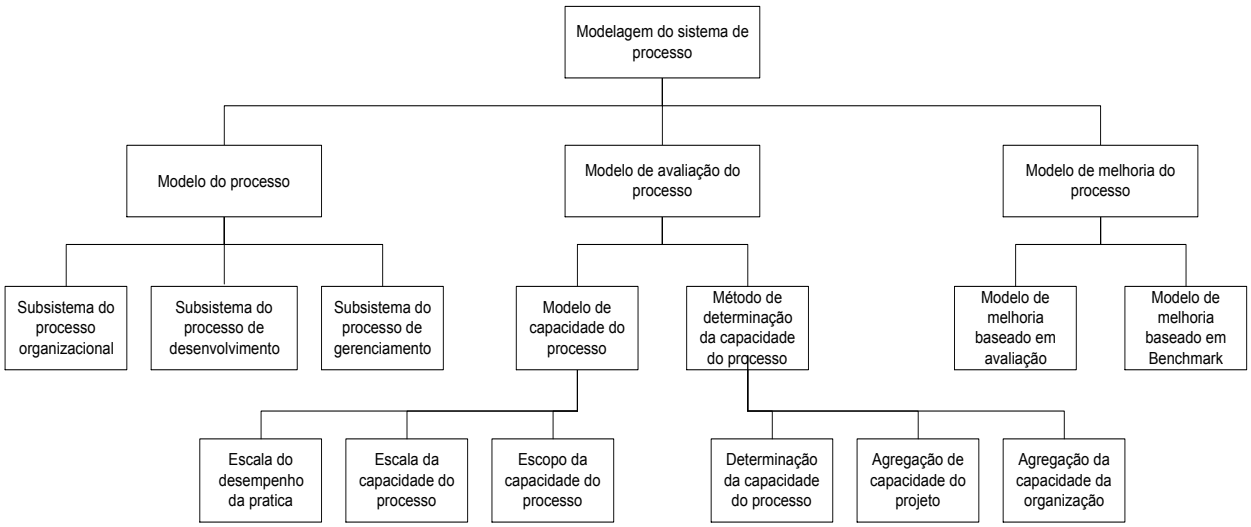


**Figura 2.6: Estrutura para modelagem do sistema de processo da ODS**

O ***modelo do processo*** é utilizado para descrever a organização, sua categoria, sua hierarquia, o inter-relacionamento e as instâncias dos seus processos. O modelo de processo descrito por [Wang1999] identifica três conjuntos de subsistemas: processo organizacional, processo de desenvolvimento de software e processo de gerenciamento. Os *processos organizacionais* regulam as atividades que são geralmente praticadas em uma ODS acima do nível de projeto. Os *processos de desenvolvimento* e de *gerenciamento* são interativos e, em paralelo, atuam sobre o projeto.

O ***modelo de avaliação do processo*** serve para definir procedimentos para avaliar os pontos fortes e fracos dos processos da ODS, além de identificar os pontos para melhoria. Através do ***modelo de melhoria do processo*** podem-se definir procedimentos sistemáticos para uma efetiva melhoria do desempenho dos processos da ODS, mudando os processos correntes ou adicionando a eles novos processos para correção ou melhoria de problemas identificados. O processo de melhoria vem a seguir do processo de avaliação e o relacionamento entre eles forma um ciclo repetitivo até o processo de a ODS estar otimizado. Exemplo disso é o *plan-do-check-act* descrito por Campos [Campos1992]. O capítulo 6 detalhará os aspectos referentes à avaliação e melhoria dos processos de software.

Diversos modelos, normas e padrões definem certificação, avaliação e melhoria para o processo de software, entre eles, podem-se citar: a ISO9000 [ISO9000-3:1997], CMM [Paulk1993] [Paulk1997], CMMI [CMMI:00], ISO15504 [ISO15504:1-9:98] e

BootStrap [Kuvaja1993] [Kuvaja1994]. O capítulo 6 detalhará os aspectos referentes a essas normas e padrões.

Apesar das ODSs durante muito tempo negligenciarem a especificação e o gerenciamento de seus processos, estes sempre existiram. Porém, não há um consenso de que tipo de processo de software deva ser utilizado em uma ODS, pois alguns processos se adequam melhor a certos tipos de aplicações do que outros. Além disso, uma ODS pode, inclusive, possuir diversos padrões de processos de software sendo utilizados em projetos distintos.

O reconhecimento das necessidades dos modelos de processo de software tem deixado um amplo campo de trabalho em muitas direções. As ODSs têm verificado que definindo seus processos pode-se melhorar sua eficácia e a qualidade dos produtos e serviços que realiza.

# 3

# MODELOS DE CICLO DE VIDA DO PROCESSO DE SOFTWARE

A fim de facilitar o entendimento do processo de desenvolvimento de software[5], vários modelos de ciclo de vida têm sido propostos. Estes modelos são descrições abstratas do processo de desenvolvimento, tipicamente mostrando as principais atividades e dados usados na produção e manutenção de software, bem como a ordem em que as atividades devem ser executadas. As atividades presentes nos diversos modelos de ciclo de vida de software não são um padrão; elas dependem da metodologia[6] utilizada no desenvolvimento de um projeto de software. A seguir será descrito, em detalhes, alguns dos principais modelos de ciclo de vida do processo de software.

## 3.1 O MODELO CASCATA

O modelo de ciclo de vida mais antigo e também um dos mais usados é o chamado modelo cascata (ou clássico) (vide Figura 3.1). Foi derivado de modelos existentes em outras engenharias e considera que o processo de desenvolvimento de software é composto por várias etapas que são "executadas" de forma sistemática e seqüencial.

Durante a etapa de *Definição de Requisitos* os serviços, as metas e as restrições impostas ao sistema são identificados junto aos usuários do software. Nessa etapa, os requisitos identificados também são analisados de modo a remover inconsistências e ambigüidades. As atividades dessa etapa serão detalhadas no capítulo 5, seção 5.1.

[5] Ressaltamos que processo de desenvolvimento de software está intimamente ligado à engenharia do produto de software.

[6] Uma metodologia é um processo concreto que descreve em detalhes as atividades, passos, artefatos e respectivos responsáveis envolvidos no desenvolvimento de um projeto de software. Várias metodologias diferentes podem estar relacionadas a um modelo de ciclo de vida de software específico.

Na etapa de *Projeto do Sistema e do Software*, os requisitos identificados são mapeados em componentes de hardware e software, de modo que o sistema possa ser posteriormente implementado. Nessa etapa, a arquitetura geral do sistema também é estabelecida. As atividades dessa etapa serão detalhadas no capítulo 5, seção 5.2.

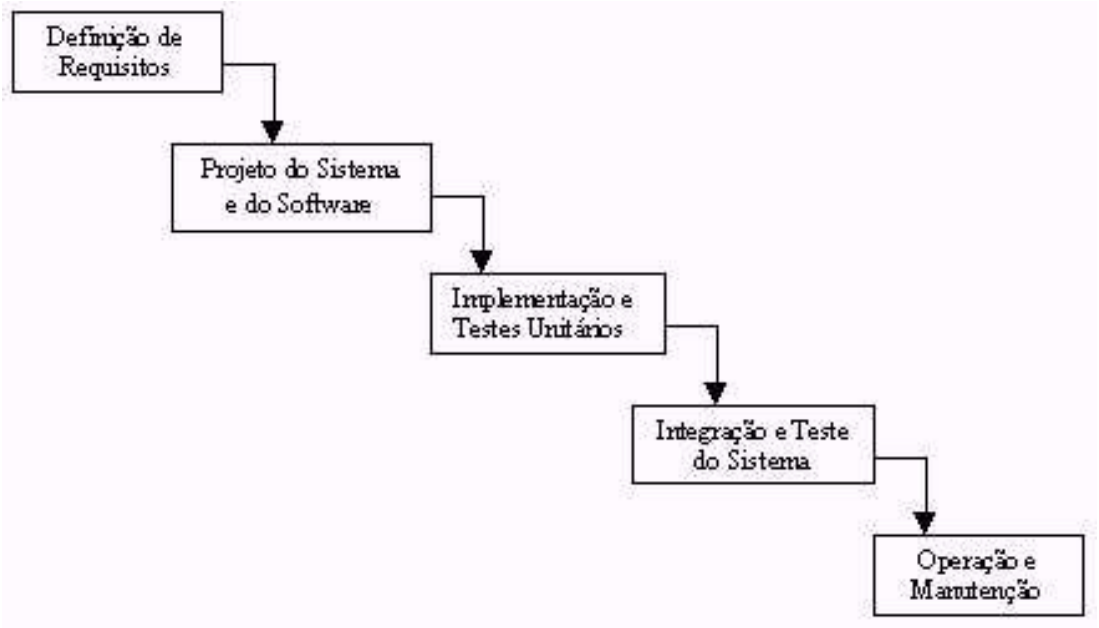


**Figura 3.1: O Modelo Cascata**

Na etapa de *Implementação e Testes Unitários*, o projeto de software é implementado em unidades de programas, utilizando-se uma linguagem de programação. Nessa etapa, as unidades implementadas também são testadas para assegurar conformidade em relação às suas especificações. As atividades dessa etapa serão detalhadas no capítulo 5, seção 5.3.

Na etapa de *Integração e Teste do Sistema*, as unidades de programas são integradas e testadas como um sistema completo, para assegurar que todos os requisitos do software sejam atendidos. Recomenda-se que os testes sejam feitos à medida que as unidades individuais vão sendo integradas (*testes de integração*) e que, ao final da integração, o sistema completo seja testado novamente (*teste de sistema*). Por esse motivo, muitas vezes a *Integração* é considerada como parte integrante da *Implementação*. No capítulo 5, seção 5.4, as atividades de *Testes* serão detalhadas.

Na etapa de *Operação e Manutenção*, o sistema é instalado e colocado em *Operação*. Posteriormente, quando erros forem encontrados no sistema ou quando forem solicitadas mudanças nos requisitos, o sistema entra numa etapa de *Manutenção*. As atividades dessa etapa serão detalhadas no capítulo 5, seção 5.6.

Na sua forma mais simples, o modelo cascata não apresenta iterações, como visto na Figura 3.1. Na prática, porém, o processo de desenvolvimento de software pode ter etapas que são desenvolvidas em paralelo e de forma iterativa (vide Figura 3.2), pois durante uma determinada etapa, problemas existentes na etapa anterior podem ser descobertos (ex: novos requisitos podem ser descobertos durante a realização da etapa de projeto, o que implica uma iteração para a etapa especificação e análise de requisitos).

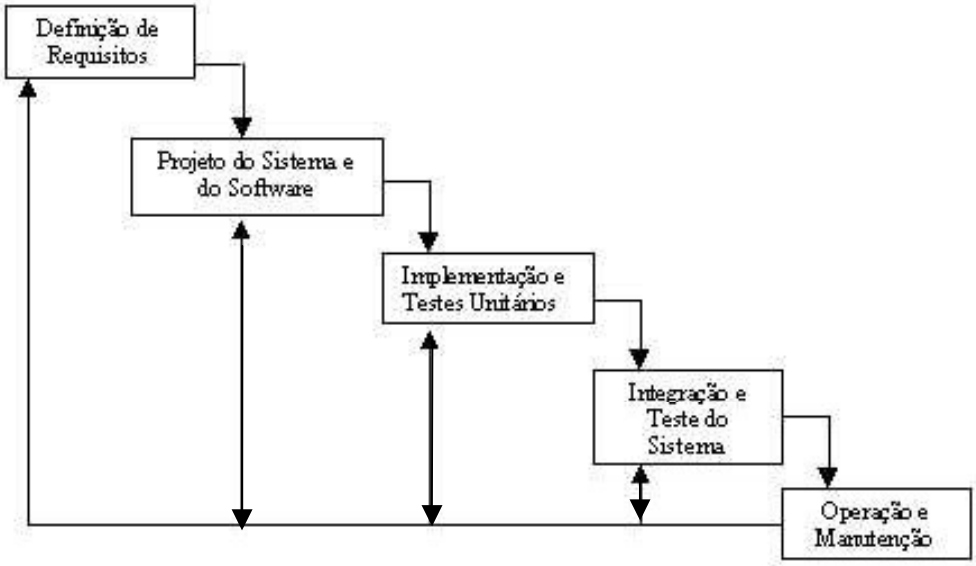


**Figura 3.2: O Modelo Cascata com Iterações**

Outra limitação do modelo cascata é que o mesmo não contempla atividades que são executadas antes do início do ciclo de vida (ex: *Estudo de viabilidade*), bem como atividades que são executadas durante todo o ciclo de vida do software (ex: *Planejamento* e *Gerenciamento*, *Verificação* e *Validação*, *Gerência de Configuração* e *Documentação*).

O *Estudo de viabilidade* deve ser executado antes do início do projeto de software e tem por objetivo: delinear o escopo do problema a ser resolvido; identificar alternativas de solução; identificar seus custos e tempo para execução; identificar potenciais benefícios para o usuário. O ideal é fazer uma análise profunda do problema para se produzir um estudo de viabilidade bem fundamentado. Na prática, porém, tal análise tem sido superficial devido a limitações de tempo e de custo. Como não se tem certeza de que a proposta para a execução do serviço será aceita pelo usuário em potencial, torna-se bastante arriscado investir muitos recursos nessa etapa.

O *Planejamento* de um projeto de software se caracteriza pela definição das tarefas a serem realizadas durante o seu desenvolvimento, bem como pela atribuição de responsabilidades, custos, prazos, marcos de referência e recursos para a realização das mesmas. As atividades de planejamento são executadas desde o início do projeto até a sua finalização.

O *Gerenciamento* está intimamente relacionado ao planejamento e tem o foco em garantir que o desenvolvimento do projeto está ocorrendo de acordo com o planejado. Assim, o gerenciamento de um projeto de software gera *feedbacks* para o planejamento e re-planejamento do projeto. Associadas ao gerenciamento do projeto, estão atividades de gerência de requisitos (gerencia as mudanças de requisitos e seus impactos durante o desenvolvimento do projeto), gerência de configuração (gerencia versões do software e as interdependências entre as partes componentes do software – ex: documentos e programas) e gerência da qualidade (dos produtos desenvolvidos e do processo de desenvolvimento).

No capítulo 4, as atividades de gerenciamento e planejamento são detalhadas; a seção 5.1 apresenta um detalhamento das atividades de gerência de requisitos; a seção 5.5 apresenta um detalhamento das atividades de gerência de configuração; e o capítulo 6 detalha as atividades de gerência da qualidade.

As atividades de *Verificação e Validação* estão relacionadas à gerência da qualidade e são responsáveis pelos *feedbacks* durante o desenvolvimento do software para as demais atividades do ciclo de vida. Na Figura 3.2, essas atividades estão relacionadas às iterações.

Basicamente, *Verificação* investiga, em cada etapa de desenvolvimento, se o software que está sendo construído atende aos requisitos especificados. Como exemplos de atividades de *Verificação*, temos os testes unitários, de integração e de sistemas, os quais serão detalhados posteriormente na seção 5.4; e as revisões e inspeções que serão tratadas no capítulo 6.

A *Validação* investiga se as funções oferecidas pelo software são as que o usuário realmente quer, pois é possível que os requisitos especificados não atendam às reais necessidades dos usuários. As atividades de *Validação* caracterizam-se por contarem com a colaboração direta do usuário em suas execuções. Como exemplos de atividades de Validação estão os testes de aceitação, que serão tratados na seção 5.4 e a validação de requisitos que será tratada na seção 5.1.

As atividades de *Verificação* e *Validação* são executadas durante todo o ciclo de vida do software, a fim de que erros e omissões sejam descobertos antes de serem propagados de uma etapa para outra. Isso é necessário porque quanto mais tarde erros e omissões forem descobertos, mais dispendioso fica para consertá-los. A

diferença entre os dois conceitos pode ser mais bem entendida através da definição dada por Boehm [Boehm1981]:

*"Verificação: Estamos construindo o produto da maneira certa?"*

*"Validação: Estamos construindo o produto certo?"*

A principal desvantagem do modelo cascata é que boa parte do sistema não estará disponível até um ponto adiantado no cronograma do projeto e, geralmente, é difícil convencer o usuário de que é preciso paciência. Além disso, existe a dificuldade de acomodação das mudanças depois que o processo está em andamento. Portanto, esse modelo é mais apropriado quando os requisitos são bem entendidos. No entanto, há também algumas vantagens associadas ao modelo, pois ele oferece uma maneira de tornar o processo mais visível, fixando pontos específicos para a escrita de relatórios e, didaticamente, é uma maneira mais fácil de introduzir os principais conceitos de Engenharia de Software. Por esse motivo, as atividades do ciclo de vida do software serão detalhadas nos capítulos 4 e 5 com base neste modelo de ciclo de vida.

## 3.2 O MODELO DE DESENVOLVIMENTO EVOLUCIONÁRIO

O Modelo de Desenvolvimento Evolucionário (vide Figura 3.3) subdivide-se em dois: Programação Exploratória e Prototipagem Descartável.

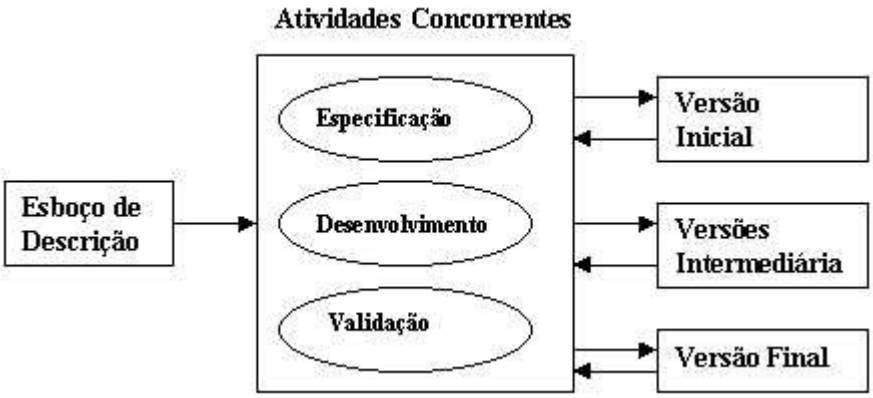


**Figura 3.3: O Modelo de Desenvolvimento Evolucionário**

### 3.2.1 O Modelo de Programação Exploratória

O objetivo desse modelo é o desenvolvimento da primeira versão do sistema o mais rápido possível. Os sistemas desenvolvidos com esse modelo caracterizam-se por não terem o escopo claramente definido, ou seja, a especificação do escopo é feita de forma intercalada ao desenvolvimento. Após o desenvolvimento de cada uma das

versões do sistema, ele é mostrado aos usuários para comentários. Modificações sucessivas são feitas no sistema até que o mesmo seja considerado adequado. A principal diferença dos outros modelos é a ausência da noção de programa correto. Esse modelo tem sido usado com sucesso para o desenvolvimento de Sistemas Especialistas, no contexto da Inteligência Artificial (ex: sistemas de reconhecimento de voz, sistemas de diagnóstico médico etc.).

### 3.2.2 O Modelo de Prototipagem Descartável

O objetivo principal desse modelo é entender os requisitos do sistema. Tem sido usado com sucesso para validar partes do sistema (Interface Gráfica e aspectos do sistema relacionados à arquitetura – ex: performance, portabilidade etc.). Como na programação exploratória, a primeira etapa prevê o desenvolvimento de um programa (protótipo) para o usuário experimentar. No entanto, ao contrário da programação exploratória, o protótipo é então descartado e o software deve ser re-implementado na etapa seguinte, usando qualquer modelo de ciclo de vida (ex: cascata).

## 3.3 O MODELO DE TRANSFORMAÇÃO FORMAL

Uma especificação formal (definição matemática, não ambígua) do software é desenvolvida e, posteriormente, transformada em um programa executável (vide Figura 3.4), através de regras que preservam a corretude da especificação (passos de refinamento).

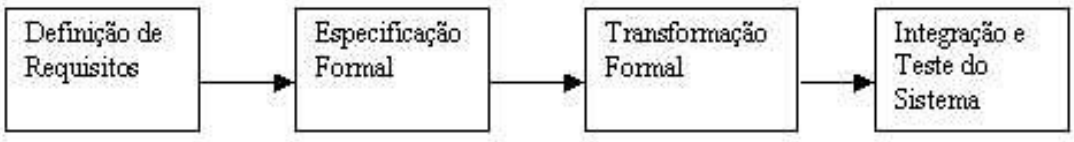


**Figura 3.4: O Modelo de Transformação Formal**

Esse modelo tem sido aplicado ao desenvolvimento de sistemas críticos, especialmente naqueles onde a segurança é um fator crítico (ex: sistema de controle de tráfego aéreo). No entanto, a necessidade de habilitações especializadas para aplicar as técnicas de transformação e a dificuldade de especificar formalmente alguns aspectos do sistema, tais como a interface com o usuário, são fatores que limitam seu uso.

## 3.4 O MODELO DE DESENVOLVIMENTO BASEADO EM REUSO

Esse modelo baseia-se no reuso sistemático de componentes existentes ou sistemas COTS (*Commercial-Off-The-Shelf*). Durante o projeto do sistema, os componentes que podem ser reusados são identificados e a arquitetura do sistema é

modificada para incorporar esses componentes. O sistema é, então, construído seguindo essa arquitetura revisada. O processo de desenvolvimento divide-se nas seguintes etapas (vide Figura 3.5):

- *Especificação de requisitos*: os requisitos do sistema são especificados;
- *Análise de componentes*: identificam-se componentes que são candidatos a serem reusados no projeto do sistema;
- *Modificação dos requisitos*: os requisitos identificados são modificados para se adequarem aos componentes a serem reusados;
- *Projeto do sistema com reuso*: o sistema é projetado, utilizando-se os componentes a serem reusados;
- *Desenvolvimento e integração*: componentes não-existentes são desenvolvidos e todos os componentes são integrados;
- *Validação*: o sistema é validado pelo usuário final.

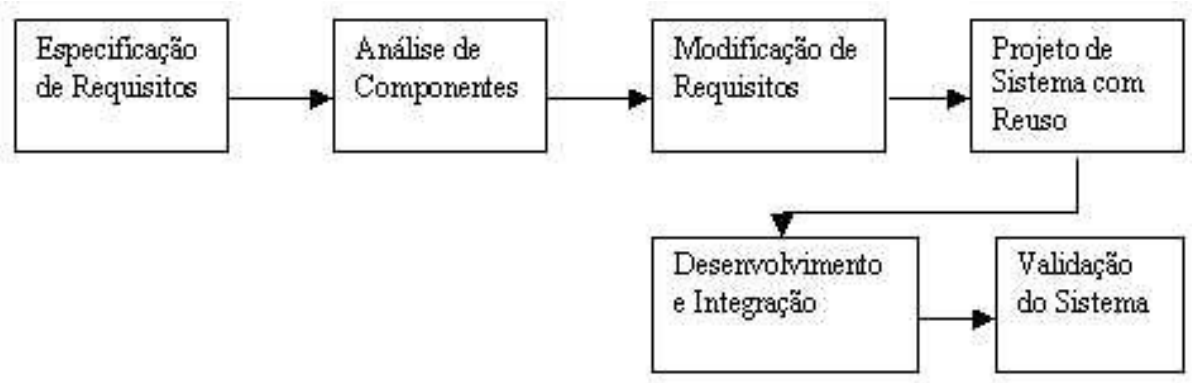


**Figura 3.5: Desenvolvimento baseado em reuso**

Esse modelo de ciclo de vida assume que o sistema é, em sua maior parte, formado por componentes pré-existentes. Assim, o processo de desenvolvimento passa a ser similar ao de uma linha de montagem. Essa abordagem de desenvolvimento tem se tornado muito importante, contudo ainda há pouca experiência com ela em larga escala.

## 3.5 MODELOS ITERATIVOS

Os requisitos de um sistema sempre evoluem durante o curso de um projeto. Assim, a iteração do processo sempre faz parte do desenvolvimento de grandes sistemas de software. Iterações podem ser aplicadas a qualquer modelo do ciclo de vida de software, mesmo no modelo cascata, como vimos anteriormente. Nesse contexto, há duas abordagens relacionadas que são mais adequadas para o tratamento de iterações:

- Desenvolvimento Espiral;
- Desenvolvimento Incremental.

### 3.5.1 O Modelo de Desenvolvimento Espiral

Acrescenta aspectos gerenciais (planejamento, controle e tomada de decisão) ao processo de desenvolvimento de software, ou seja, análise de riscos em intervalos regulares.

O processo de desenvolvimento é representado como uma espiral, ao invés de uma seqüência de atividades (vide Figura 3.6). O modelo define quatro quadrantes, nos quais as atividades (gerenciais ou técnicas) de um projeto são executadas durante um ciclo na espiral:

- *Determinação dos objetivos, alternativas e restrições*: os objetivos específicos para a etapa são identificados e alternativas para realizar os objetivos e restrições são encontradas;
- *Análise das alternativas e identificação e/ou resolução de riscos*: os riscos principais são identificados, analisados e buscam-se meios para reduzir esses riscos;
- *Desenvolvimento e validação da versão corrente do produto*: um modelo apropriado para o desenvolvimento é escolhido, o qual pode ser qualquer um dos modelos de ciclo de vida;
- *Planejamento*: o projeto é revisto e o próximo ciclo da espiral é planejado.

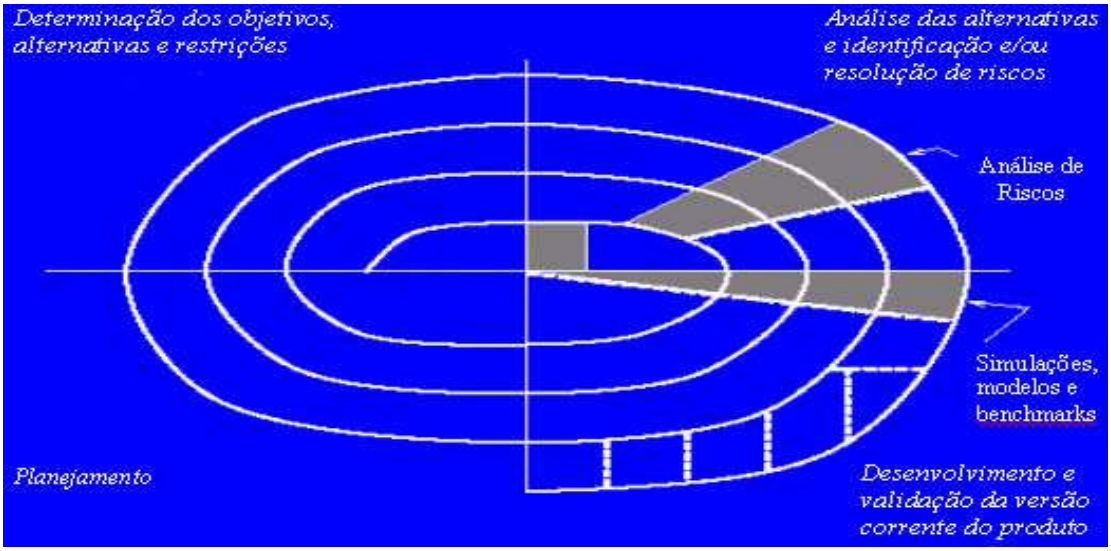


**Figura 3.6: O modelo espiral**

A espiral representa o curso do projeto, onde, a cada volta, um novo produto é construído (ex: documento de requisitos, modelo de projeto, implementação etc.). Cada volta na espiral representa uma etapa no processo de desenvolvimento. Não há etapas fixas como especificação ou projeto (o conteúdo de uma volta na espiral é escolhido dependendo do produto requerido pela etapa). Os riscos são avaliados explicitamente e resolvidos ao longo do processo.

### 3.5.2 O Modelo de Desenvolvimento Incremental

Em vez de entregar o sistema como um todo, o desenvolvimento e a entrega são divididos em incrementos, com cada incremento representando parte da funcionalidade requerida (vide Figura 3.7).

Os requisitos dos usuários são priorizados e os requisitos de mais alta prioridade são incluídos nas iterações iniciais. Uma vez que o desenvolvimento de um incremento é iniciado, os requisitos são "congelados", embora os requisitos possam continuar evoluindo para incrementos posteriores.

Em certos aspectos é similar à programação exploratória. No entanto, o escopo do sistema deve ser claramente entendido antes de se iniciar o desenvolvimento.

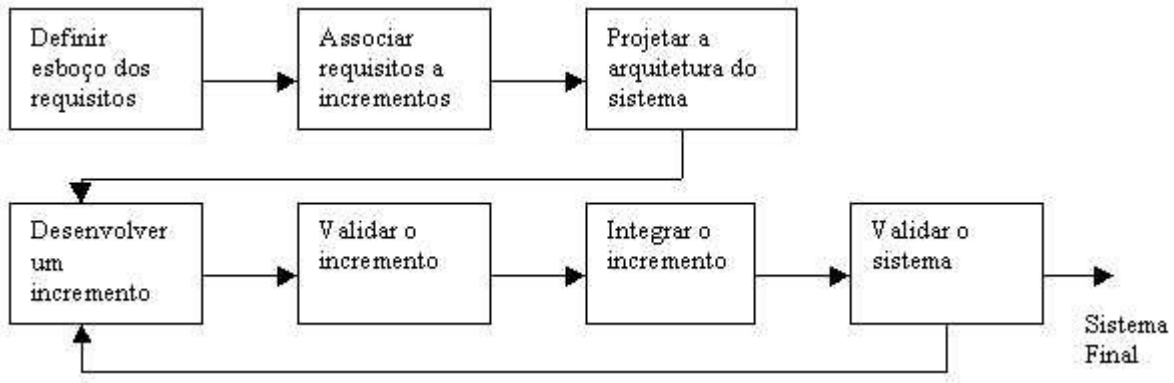


**Figura 3.7: O modelo incremental**

As principais vantagens do modelo incremental são:

- A funcionalidade do sistema estará disponível mais cedo, pois ela é entregue a partir dos incrementos;
- Incrementos iniciais agem como um protótipo para ajudar a elicitar requisitos para incrementos finais;
- Diminuem-se os riscos de falhas no projeto como um todo;
- Os serviços de prioridade mais alta do sistema tendem a receber mais testes.

## 3.6 CONSIDERAÇÕES FINAIS

Modelos de ciclo de vida de software são descrições abstratas do processo de desenvolvimento, tipicamente mostrando a seqüência de execução das principais atividades envolvidas na produção e evolução de um sistema de software.

Existem vários modelos de ciclo de vida e todos eles apresentam pontos positivos e negativos. Parte do trabalho do Engenheiro de Software é escolher o modelo de ciclo de vida mais adequado às necessidades do cliente/usuário e da empresa que irá desenvolver o software.

Nos capítulos 4 e 5 as principais atividades envolvidas no processo de desenvolvimento de software, independentemente do modelo de ciclo de vida adotado, serão descritas em detalhes. O foco será principalmente nas atividades de *Gerenciamento, Requisitos* e *Testes*, as quais estão mais intimamente relacionadas com a qualidade do produto de software. A etapa de planejamento e gerenciamento, apresentada no capítulo 4, é responsável por garantir que o produto seja entregue no prazo e no custo desejados, bem como por definir critérios de acompanhamento do progresso do projeto. A qualidade do software começa a ser fomentada na etapa de requisitos, pois é nessa etapa que são determinados os principais atributos de qualidade relevantes para o usuário do sistema. Finalmente, a etapa de testes é um dos principais mecanismos para garantir que o produto final seja entregue com qualidade. No capítulo 6 outras atividades ligadas à garantia da qualidade serão abordadas.

# 4

# PLANEJAMENTO E GERENCIAMENTO DE PROJETOS DE SOFTWARE

Alguns estudos e pesquisas que foram realizados nos anos 1990 demonstraram que o gerenciamento de projeto é a causa mais evidente das falhas na execução e entrega de produtos e serviços de software. O *SEI-Software Engineering Institute* constatou, já em 1993, que o principal problema que aflige as organizações de software é gerencial e preconizou que “as organizações precisam vencer o seu buraco negro, que é o seu estilo de gerenciar de maneira informal, sem métodos e sem técnicas” [Paulk1993].

Um estudo conduzido pelo *DoD-Department of Defense* [Dod1994] indicou que 75% de todos os sistemas de software falham e que a causa principal é o pobre gerenciamento por parte do desenvolvedor e adquirente, deixando claro que o problema não é de desempenho técnico.

De acordo com um estudo realizado pelo *Standish Group* [Standish2001], publicado no artigo “Extreme CHAOS 2001”, no período 2000/2001 apenas 28% dos projetos de desenvolvimento pesquisados nos E.U.A. obtiveram sucesso, ou seja, foram completados no tempo e no custo previstos, e possuíam todas as funcionalidades inicialmente planejadas. Apesar do aumento de sucesso em relação ao ano de 1994, quando apenas 16% dos projetos conseguiram êxito, este ainda é um percentual bastante reduzido. Este estudo aponta o gerenciamento de software como sendo a principal razão para o sucesso ou a falha de um projeto de software.

Através de uma análise e acompanhamento de cem projetos de software, Jones [Jones1996] relata: “os seis primeiros dos dezesseis fatores associados aos desastres do software são falhas específicas no domínio do gerenciamento de projeto, e os três dos outros dez fatores restantes estão indiretamente associados às praticas de gerenciamento pobre”.

Walker [Walker1997] conclui que: “o desenvolvimento de software ainda é imprevisível; somente 10% dos projetos de software são entregues com sucesso dentro das estimativas de orçamento e custo; a disciplina de gerência é, portanto, mais um discriminador de sucesso ou falha dos projetos”.

Muitas pesquisas enfatizam que o gerenciamento é a principal causa do sucesso ou fracasso dos projetos de software. Apesar disso, o gerenciamento de projetos de software ainda é pouco abordado e praticado nas ODSs [Machado2001]. Jones [Jones1999] destaca que a ausência de um processo de gerenciamento apropriado, aliado a estimativas deficientes de custo e de tempo, é uma das principais causas das falhas dos projetos de software.

Os principais padrões e normas para SPA/SPI (*Software Process Assessment/ Software Process Improvement*) têm colocado o gerenciamento de projetos como um dos requisitos básicos para que uma empresa inicie a melhoria de seu processo. Contudo, a introdução de padrões e normas dentro das ODSs tem se mostrado complexa demais, além de causar uma sobrecarga de trabalho significativa. Segundo Belloquin [Belloquin1999], esses padrões e normas definem práticas que devem ser realizadas, porém não determinam como executá-las. O capítulo 6 irá discorrer sobre esses padrões.

## 4.1 AS DIFICULDADES DO GERENCIAMENTO DE PROJETOS DE SOFTWARE

Já em 1989, Humphrey [Humphrey1989] constatou que: “a ausência de práticas administrativas no desenvolvimento de software é a principal causa de sérios problemas enfrentados pelas organizações, tais como: atrasos em cronogramas, custo maior do que o esperado e presença de defeitos, ocasionando uma série de inconveniências para os usuários e enorme perda de tempo e de recursos”. Ainda hoje esta afirmação tem sido confirmada por diversos autores.

Na atual cultura das ODSs, o planejamento, quando ocorre, é feito de forma superficial [Weber1999] [Sanches2001]. A maioria dos projetos de software é realizada sem um planejamento de como a idéia modelada, durante o levantamento de requisitos e necessidades dos clientes, pode ser transformada em produto.

Os gerentes de projeto estão desacostumados a estimar [Vigder1994]. Quando estimam, costumam basear-se em estimativas passadas, mesmo sabendo que elas podem estar incorretas (não sabem também precisar o quanto elas estão incorretas). Há gerente que se recusa a estimar somente por julgar perda de tempo, uma vez que se corre o risco de obter resultados incorretos e, portanto, estar desperdiçando tempo.

Boas estimativas de custo, esforço e prazo de software requerem um processo ordenado que defina a utilização de métricas de software, método e ferramenta de

estimativa. As ODSs, de forma geral, não detêm conhecimentos e recursos para isso [Vigder1994].

Estimar, medir e controlar um projeto de software é tarefa difícil, pois o desenvolvimento de software é uma atividade criativa e intelectual, com muitas variáveis envolvidas (como metodologias, modelos de ciclo de vida, técnicas, ferramentas, tecnologia, recursos e atividades diversas). Os gerentes inexperientes tentam cumprir prazos dando a máxima velocidade na fase inicial e estão despreparados para os momentos de impasse, quando os ajustes são inevitáveis.

A dinamicidade do processo de software dificulta também o gerenciamento efetivo de projetos de software, devido às alterações constantes nos planos de projetos, redistribuição de atividades, inclusão/exclusão de atividades, adaptação de cronogramas, realocação de recursos, novos acordos com os clientes, entregas intermediárias não previstas etc. Um projeto de software também deve adaptar-se ao ambiente, dependendo da disponibilidade de recursos, ferramentas e habilidades do pessoal ou equipe.

## 4.2 PRINCIPAIS ATIVIDADES DO GERENCIAMENTO DE PROJETOS DE SOFTWARE NAS ODSS

A gerência de projetos trata do planejamento e acompanhamento das atividades voltadas a assegurar que o software seja entregue dentro do prazo previsto e de acordo com os requisitos especificados pelas organizações que estão desenvolvendo e adquirindo o software. O gerenciamento de projeto é necessário porque o desenvolvimento de software está sempre sujeito a restrições de orçamento e cronograma.

Gerentes lutam para cumprir os objetivos dos projetos, os quais têm prazos finais difíceis de serem cumpridos. Freqüentemente, os sistemas que são entregues não satisfazem aos usuários, os gastos com manutenção são muito grandes e os prazos não são cumpridos. Muitas vezes, esses problemas ocorrem não por incompetência das pessoas, mas por falha nas técnicas de gerência empregadas nos projetos. Nas próximas subseções, serão abordados os aspectos envolvidos com a gerência de projetos de software.

O gerenciamento de um projeto de software difere de outros projetos de engenharia porque, no caso do software, o produto não é concreto (a análise do progresso do projeto depende de sua documentação). Não existe um processo padrão de gerência e grandes sistemas de software são normalmente desenvolvidos uma única vez, o que restringe o reuso de informações de projetos anteriores. Dependendo da empresa e do projeto, as atividades mais comuns da gerência são:

- Escrever a proposta do projeto;
- Fazer estimativas do projeto (custo, tempo etc.);
- Definir marcos de referência;
- Analisar os riscos;
- Fazer o planejamento e o cronograma do projeto;
- Selecionar e avaliar pessoal;
- Fazer acompanhamento e revisões;
- Escrever os relatórios de acompanhamento;
- Fazer apresentações sobre o projeto.

### 4.2.1 O Planejamento do Projeto

É uma atividade contínua, desde a concepção inicial do sistema, até a sua entrega. Os planos devem ser revisados regularmente, à medida que novas informações se tornam disponíveis. Caso seja necessário, os planos devem ser atualizados. O plano de projeto é um documento que descreve as atividades, os recursos e o cronograma usados para o desenvolvimento do sistema. Uma possível estrutura para esse documento é descrita na Figura 4.1.

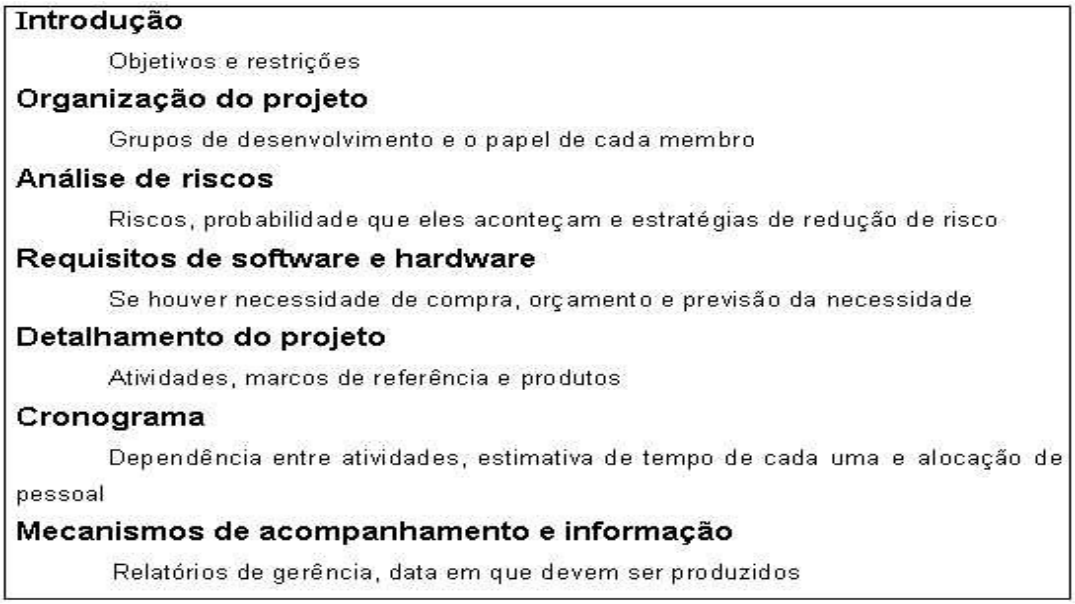


**Figura 4.1: Estrutura de um plano de projeto**

### 4.2.2 A Seleção de Pessoal

Nem sempre é possível conseguir as pessoas ideais para trabalharem num projeto devido a limitações, tais como:

- Orçamento de projeto pode não permitir o uso de pessoal altamente qualificado e, conseqüentemente, bem pago;
- Pessoal com experiência apropriada pode não estar disponível;
- Pode fazer parte da política da organização desenvolver as habilidades dos empregados durante um projeto de software. Ou seja, projetos podem ser usados como forma de treinar o pessoal.

Assim, gerentes têm de alocar o pessoal disponível, dentro das restrições impostas. Muitas vezes, também é papel do gerente participar da seleção para a contratação de pessoal para um projeto.

### 4.2.3 O Gerenciamento de Riscos

Um risco é uma probabilidade de que alguma circunstância adversa aconteça. O gerenciamento de riscos trata da identificação dos riscos e da preparação de planos para minimizar os seus efeitos no projeto. Riscos podem ser classificados de acordo com vários critérios. Uma possível classificação seria:

- *Riscos de projeto*: afetam cronogramas ou recursos;
- *Riscos de produto:* afetam a qualidade ou desempenho do software que é desenvolvido;
- *Riscos de negócio*: afetam a organização que está desenvolvendo ou adquirindo o software.

O processo de gerenciamento de riscos pode ser dividido nas seguintes atividades:

- *Identificação dos riscos:* identificar os riscos de projeto, produto e negócio. Esses riscos podem estar associados à escolha da tecnologia, das pessoas, mudanças de requisitos, estimativas do projeto etc.;
- *Análise dos riscos:* avaliar a probabilidade e as conseqüências dos riscos. Uma possível classificação para a probabilidade e para as conseqüências pode ser:

  Probabilidade: muito baixa, baixa, moderada, alta ou muito alta.

  Conseqüência: catastrófica, séria, tolerável ou insignificante;
- *Planejamento dos riscos:* preparar planos, definindo estratégias para gerenciar os riscos. As estratégias podem ser:

- Estratégias para evitar o risco: a probabilidade de que o risco surja é reduzida.
- Estratégias para minimizar o risco: O impacto do risco no projeto ou no produto será reduzido.
- Planos de contingência: se o risco surgir, os planos de contingência tratarão aquele risco;

- *Monitoramento dos riscos: m*onitorar os riscos ao longo do projeto. Avalia cada risco identificado regularmente para decidir se ele está se tornando menos ou mais provável. Também avalia se as conseqüências do risco têm se modificado. Cada risco-chave deve ser discutido nas reuniões de progresso do gerenciamento.

### 4.2.4 A Definição das Atividades, Marcos de Referência e Produtos Entregues ao Usuário

Associados às atividades, podem existir marcos de referência ("*milestones*") ou produtos. Um marco de referência é um ponto final, bem definido, de uma etapa ou atividade. A escolha dos marcos de referência e das suas freqüências de produção está relacionada ao modelo de ciclo de vida utilizado no projeto. Por exemplo, num projeto desenvolvido utilizando o ciclo de vida em cascata, ao final de cada etapa de desenvolvimento, pode haver um marco de referência. Nesse caso, um possível marco de referência seria o modelo de análise e projeto, o qual seria produzido ao final da etapa de mesmo nome.

Os marcos de referência podem ser também associados à conclusão de uma atividade. Nesse caso, um marco de referência associado a uma atividade da etapa de análise e projeto poderia ser a produção de um diagrama específico do modelo, como, por exemplo, um diagrama de classes, produzido por uma atividade associada a essa etapa de desenvolvimento.

Por outro lado, um produto a ser entregue ao cliente (*"deliverable"*) diferencia-se do marco de referência justamente pelo fato de que nem todos os marcos de referência são entregues ao cliente. Ou seja, produtos são marcos de referência, mas marcos de referência não são necessariamente produtos.

### 4.2.5 A Definição do Cronograma

O cronograma divide o projeto em tarefas e estima o tempo e os recursos requeridos para completar cada tarefa. Sempre que possível, devem ser definidas tarefas concorrentes de modo a fazer o melhor uso do pessoal. Outra premissa que deve ser levada em conta é tentar definir as tarefas que são independentes umas das outras. Isso evita atrasos causados por uma tarefa que está esperando por uma outra

ser completada. No entanto, a definição de bons cronogramas depende da intuição e da experiência dos gerentes de projeto. Ou seja, não existe uma ciência exata que determine a melhor forma de se construir um cronograma. Dentre os principais problemas relacionados à confecção de um cronograma, pode-se citar:

- Estimar o esforço associado à resolução dos problemas e, conseqüentemente, o custo do desenvolvimento de uma solução é difícil; A produtividade não é proporcional ao número de pessoas que estão trabalhando numa tarefa;
- Adicionar pessoas a um projeto atrasado pode fazer com que ele se atrase ainda mais. Isso ocorre devido ao *overhead* da comunicação;
- O inesperado sempre acontece. Sempre permita a contingência no planejamento e uma "folga" no cronograma;
- As tarefas não devem ser muito pequenas, de modo que não haja uma interrupção constante dos desenvolvedores pelo gerente do projeto. Assim, é recomendado que as tarefas tenham a duração entre uma e duas semanas.

## 4.3 A GERÊNCIA DE PROJETOS SOB A ÓTICA DO PMBOK

Ao se abordar o planejamento e a gerência de projeto não se pode deixar de citar o *PMBOK – Project Management Body of Knowledge*, criado pelo *PMI – Project Management Institute.*

O PMI (www.pmi.org) é uma associação de profissionais de gerenciamento de projetos que existe desde 1969. Essa associação criou em 1986 a primeira versão do PMBOK – *Project Management Body of Knowledge*. O PMBOK é um guia que descreve a somatória de conhecimento e as melhores práticas dentro da profissão de gerenciamento de projetos. É um material genérico que serve para todas as áreas de conhecimento, ou seja, tanto para construção de um edifício como para a produção de software.

A meta do gerenciamento de projetos, segundo o PMBOK é conseguir exceder as necessidades e expectativas dos *stakeholders*[7]. Todavia, satisfazer ou exceder essas necessidades envolve um balanceamento entre as várias demandas concorrentes em relação a:

- Escopo, tempo, custo e qualidade (objetivos do projeto);
- *Stakeholders* com necessidades e expectativas diferenciadas;
- Requisitos identificados (necessidades) e requisitos não identificados (expectativas).

[7] O PMBOK [PMBOK2000] define *stakeholders* como sendo os indivíduos ou as organizações que estão ativamente envolvidos em um projeto, cujos interesses podem afetar positivamente ou negativamente o resultado da execução do projeto.

O PMBOK [PMBOK2000] organiza os processos de gerenciamento de projetos em cinco grupos (Figura 4.2): processos de inicialização, processos de planejamento, processos de execução, processos de controle e processos de finalização.

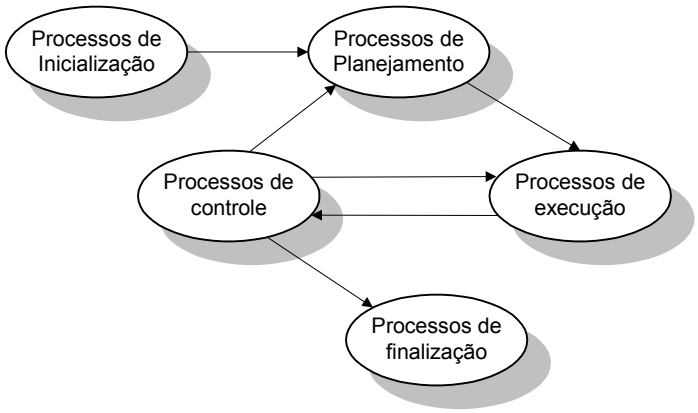


**Figura 4.2: Processos do gerenciamento de projetos do PMBOK**

Os processos de inicialização autorizam o início do projeto ou de uma fase do projeto. Os processos de planejamento definem e refinam os objetivos do projeto selecionando as melhores alternativas para realizá-lo. Os processos de execução coordenam pessoas e outros recursos para a realização do projeto, baseando-se no planejamento. Os processos de controle monitoram e medem o progresso do que está sendo executado, com o intuito de tomar ações corretivas, quando necessárias. Os processos de finalização formalizam o aceite e a finalização do projeto ou de uma fase do projeto.

Os processos do PMBOK estão organizados por áreas de conhecimento, que se referem a um aspecto a ser considerado dentro do gerenciamento de projetos. Dentro dessas áreas de conhecimento os cinco grupos de processos acima descritos podem ocorrer. A Figura 4.3 mostra as 9 áreas de conhecimento do PMBOK.

A seguir serão descritos os processos de cada área de conhecimento do PMBOK. Todos os processos das áreas abaixo descritos interagem uns com os outros e também com os processos das demais áreas de conhecimento. Cada processo pode envolver esforço de um ou mais indivíduos ou grupos de indivíduos dependendo das necessidades do projeto. Cada processo, geralmente, ocorre pelo menos uma vez em cada fase do projeto.

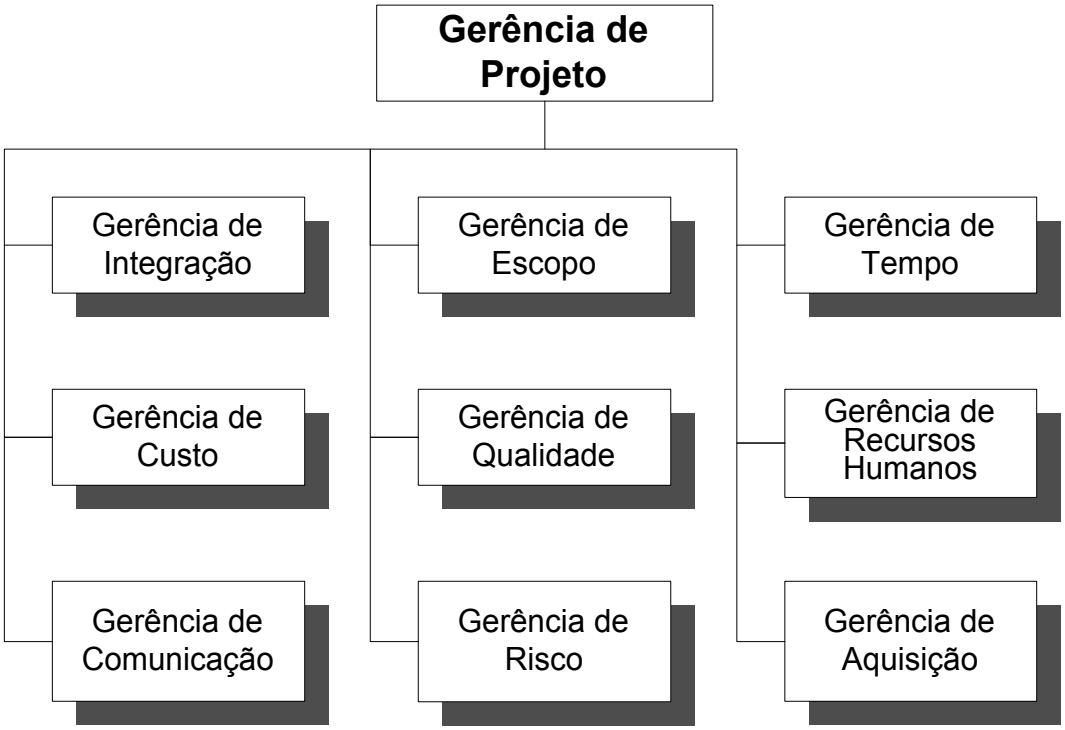


**Figura 4.3: Áreas do gerenciamento de projetos do PMBOK**

### 4.3.1 Gerência de Integração de Projetos

A gerência de integração engloba os processos necessários para garantir que os vários elementos de um projeto sejam propriamente coordenados. Objetiva realizar as negociações dos conflitos entre objetivos e alternativas do projeto, com a finalidade de atingir ou exceder as necessidades e expectativas de todas as partes interessadas. Envolve o desenvolvimento e a execução do plano de projeto e o controle geral das mudanças.

Essa área de processo incluiu os seguintes processos principais:

- Desenvolvimento do plano do projeto - agregar os resultados dos outros processos de planejamento construindo um documento coerente e consistente;
- Execução do plano do projeto - levar a cabo o projeto através da realização das atividades nele incluídas;
- Controle geral de mudanças - coordenar as mudanças através do projeto inteiro.

### 4.3.2 Gerência de Escopo de Projetos

A gerência do escopo do projeto inclui os processos requeridos para assegurar que o projeto inclua todo o trabalho necessário, e tão somente o trabalho necessário,

para complementar de forma bem sucedida o projeto. A preocupação fundamental compreende definir e controlar o que está ou não incluído no projeto.

Essa área de processo incluiu os seguintes processos principais:

- Iniciação - comprometer a organização a iniciar a próxima fase do projeto;
- Planejamento do escopo - desenvolver uma declaração escrita do escopo como base para decisões futuras do projeto;
- Detalhamento do escopo - subdividir os principais subprodutos do projeto em componentes menores e mais manejáveis;
- Verificação do escopo - formalizar a aprovação do escopo do projeto;
- Controle de mudanças de escopo - controlar as mudanças do escopo do projeto.

### 4.3.3 Gerência de Tempo de Projetos

A gerência de tempo do projeto objetiva garantir o término do projeto no tempo certo. Consiste da definição, ordenação e estimativa de duração das atividades, e de elaboração e controle de cronogramas.

Essa área incluiu os seguintes processos principais:

- Definição das atividades - identificar as atividades específicas que devem ser realizadas para produzir os diversos subprodutos do projeto;
- Seqüenciamento das atividades - identificar e documentar as relações de dependência entre as atividades;
- Estimativa da duração das atividades - estimar a quantidade de períodos de trabalho que serão necessários para a implementação de cada atividade;
- Desenvolvimento do cronograma - analisar a seqüência e as durações das atividades e os requisitos de recursos para criar o cronograma do projeto;
- Controle do cronograma - controlar as mudanças no cronograma do projeto.

### 4.3.4 Gerência de Custo de Projetos

A gerência de custo tem por objetivo garantir que o projeto seja executado dentro do orçamento aprovado. Consiste no planejamento dos recursos, estimativa, orçamento e controle de custos.

Essa área de processo incluiu os seguintes processos principais:

- Planejamento dos recursos - determinar quais recursos (pessoas, equipamentos, materiais) e que quantidade de cada deve ser usada para executar as atividades do projeto;

- Estimativa dos custos - desenvolver uma estimativa dos custos dos recursos necessários à implementação das atividades do projeto;
- Orçamento dos custos - alocar as estimativas de custos globais aos itens individuais de trabalho;
- Controle dos custos - controlar as mudanças no orçamento do projeto.

### 4.3.5 Gerência da Qualidade de Projetos

A gerência da qualidade objetiva garantir que o projeto satisfará as exigências para as quais foi contratado. Consiste de planejamento, garantia e controle de qualidade.

Essa área de processo incluiu os seguintes processos principais:

- Planejamento da qualidade - identificar quais padrões de qualidade são relevantes para o projeto e determinar a forma de satisfazê-los;
- Garantia da qualidade - avaliar periodicamente o desempenho geral do projeto buscando assegurar a satisfação dos padrões relevantes de qualidade;
- Controle da qualidade - monitorar os resultados específicos do projeto para determinar se eles estão de acordo com os padrões de qualidade relevantes e identificar as formas para eliminar as causas de desempenhos insatisfatórios.

### 4.3.6 Gerência de Recursos Humanos de Projetos

A gerência de recursos humanos objetiva garantir o melhor aproveitamento das pessoas envolvidas no projeto. Consiste de planejamento organizacional, alocação de pessoal e definição de equipe.

Essa área de processo incluiu os seguintes processos principais:

- Planejamento organizacional - identificar, documentar e designar as funções, responsabilidades e relacionamentos de reporte dentro do projeto;
- Montagem da equipe - conseguir que os recursos humanos necessários sejam designados e alocados ao projeto;
- Desenvolvimento da equipe - desenvolver habilidades individuais e do grupo para aumentar o desempenho do projeto.

### 4.3.7 Gerência de Comunicação de Projetos

A gerência de comunicação tem por objetivo principal garantir a geração adequada e apropriada, coleta, disseminação, armazenamento e disponibilização da informação.

Essa área de processo incluiu os seguintes processos principais:

- Planejamento das comunicações - determinar as informações e comunicações necessárias para os interessados: quem necessita de qual informação, quando necessitará dela e como isso será fornecido;
- Distribuição das informações - disponibilizar as informações necessárias para os interessados do projeto, da maneira conveniente;
- Relato de desempenho - coletar e disseminar as informações de desempenho. Inclui relatórios de situação, medição de progresso e previsões;
- Encerramento administrativo - gerar, reunir e disseminar informações para formalizar a conclusão de uma fase ou de todo o projeto.

### 4.3.8 Gerência de Risco de Projetos

*A* gerência de risco objetiva maximizar os resultados de ocorrências positivas e minimizar as conseqüências de ocorrências negativas. Consiste de identificação, quantificação, tratamento e controle dos riscos do projeto.

Essa área de processo incluiu os seguintes processos principais:

- Identificação dos riscos - determinar quais os riscos são mais prováveis afetar o projeto e documentar as características de cada um;
- Quantificação dos riscos - avaliar os riscos, suas interações e possíveis conseqüências;
- Desenvolvimento das respostas aos riscos - definir as melhorias necessárias para o aproveitamento de oportunidades e respostas às ameaças;
- Controle das respostas aos riscos - responder às mudanças nos riscos no decorrer do projeto.

### 4.3.9 Gerência de Aquisição de Projetos

A gerência de aquisição tem por objetivo principal obter bens e serviços externos à organização executora. Consiste na seleção de fornecedores, planejamento de aquisição, planejamento de solicitação, solicitação de propostas, e administração e encerramento de contratos.

Essa área de processo incluiu os seguintes processos principais:

- Planejamento das aquisições - determinar o que contratar e quando;
- Preparação das aquisições - documentar os requisitos do produto e identificar os fornecedores potenciais;
- Obtenção de proposta - obter propostas de fornecimento, conforme apropriado a cada caso (cotações de preço, cartas-convite, licitação);

- Seleção de fornecedores - escolher entre os possíveis fornecedores;
- Administração dos contratos - gerenciar os relacionamentos com os fornecedores;
- Encerramento do contrato - completar e liquidar o contrato incluindo a resolução de qualquer item pendente.

## 4.4 CONSIDERAÇÕES FINAIS

O bom planejamento é essencial para o sucesso de um projeto. A natureza intangível do software causa problemas para o gerenciamento. Gerentes têm várias atividades, sendo as mais importantes o planejamento e a elaboração de estimativas e de cronogramas. Essas atividades são processos interativos que continuam ao longo do curso do projeto.

Marcos de referência são resultados de atividades e produtos são marcos de referência que são entregues aos clientes.

O gerenciamento de risco é uma área que toma cada vez mais importância no desenvolvimento de software e trata da identificação de riscos que podem afetar o projeto. Trata também do planejamento, para assegurar que esses riscos não se transformarão em ameaças. Os riscos podem ser de projeto, de produto ou de negócio.

O PMBOK fornece conceitos importantes para quem deseja gerenciar adequadamente os projetos de software. Outros modelos também retratam as necessidades da gerência de projetos e serão vistos no capítulo 6.

Estudos realizados na década de 1990 sobre gerenciamento de projetos de software deixaram evidente que as práticas de gerenciamento de projetos devem ser melhoradas para que os projetos de software tenham sucesso. Dada esta preocupação, muitos modelos e normas para SPA/SPI evoluíram principalmente com a inclusão de práticas gerenciais para os projetos de software. Exemplos são: CMM [Paulk1993] para CMMI [CMMI:2000]; o adendo à norma ISO12207 [ISO12207:1995] em 2001 [ISO12207:2000] e os novos processos de gerenciamento inseridos na ISO15504 no decorrer de sua confecção.

Contudo, Machado [Machado2001] demonstrou, recentemente, que apesar das pesquisas evidenciarem que o problema da indústria de software é mais gerencial do que técnico, a gerência de projetos não está sendo considerada como deveria. Através da comparação das práticas de gerenciamento propostas nos padrões e normas para SPA/SPI com as contidas no PMBOK – *Project Management Body Knowledge* [PMBOK2000], concluiu-se que os requisitos de gerenciamento de projeto não estão representados devidamente nos modelos [Machado2001]. O gerenciamento de projeto

de software tem sido priorizado há pouco tempo pelas organizações que definem padrões e normas para software.

Além disso, esses modelos foram originalmente desenvolvidos para o âmbito de empresas bem estruturadas e departamentalizadas, ou seja, tipicamente para o âmbito de grandes empresas, dificultando ainda mais a orientação para empresas de pequeno e médio porte [Bellouquim1999].

Um exemplo de modelo de processo para gerência de projetos de software é o ProGer [Rouiller2001]. O ProGer auxilia as ODSs na organização inicial de seu negócio através do uso de artefatos de alto nível e de baixa complexidade, em conjunto com um modelo de ciclo de vida para os projetos e o estabelecimento de fluxos de trabalho. Esse modelo é apresentado através de um modelo de ciclo de vida para os projetos, da definição dos *stakeholders*, da definição dos fluxos de trabalho, dos artefatos gerados no processo e de sugestões de estimativas e métricas para avaliação do desempenho da execução dos projetos.

O ProGer foi concebido considerando os padrões da engenharia de processo e do gerenciamento de projetos. Foi especificado baseado nos modelos e padrões de SPA/SPI (principalmente, a ISO15504 [ISO15504:1-9:1998] e o CMM–*Capability Maturity Model* [Paulk1993], nas metodologias para desenvolvimento de software (como o RUP-*Rational Unified Process* [Philippe1998] e o *OPEN Process* [Graham1999]), nos procedimentos e normas para gerenciamento de projetos (como o PMBOK-*Project Management Body of Knowledge* [PMBOK2000], TQC-*Total Quality Control* [Campos1992] e [Royce1998]) e nos estudos empíricos realizados em ODSs.

Para a validação desse modelo foram realizados estudos de casos em diversas empresas de software, totalizando o acompanhamento de mais de 100 projetos de software.

# 5

# O PROCESSO DE DESENVOLVIMENTO DE SOFTWARE

Este capítulo aborda as principais atividades envolvidas no processo de desenvolvimento de software, independente do modelo de ciclo de vida adotado. Ao final do capítulo discute-se a automação deste processo através do uso de ferramentas CASE[8].

## 5.1 ENGENHARIA DE REQUISITOS

O objetivo do desenvolvimento de software é a criação de sistemas de software que correspondam às necessidades de clientes e usuários. Uma correta especificação dos requisitos do software é essencial para o sucesso do esforço de desenvolvimento. Mesmo que tenhamos um sistema bem projetado e codificado, se ele foi mal especificado, certamente irá desapontar o usuário e causar desconforto à equipe de desenvolvimento, que terá de modificá-lo para se adequar às necessidades do cliente.

De acordo com o *Insitute of Electrical and Electronics Engineers* - IEEE, o processo de aquisição, refinamento e verificação das necessidades do cliente é chamado de engenharia de requisitos [IEEE1984].

A **Engenharia de Requisitos (E.R.)** aborda uma etapa crucial no ciclo de vida do desenvolvimento de Software por tratar de conhecimentos não apenas técnicos, mas também gerenciais, organizacionais, econômicos e sociais. Várias abordagens têm sido propostas para apoiar a engenharia de requisitos, sendo a maioria baseada em linguagens capazes de expressar os desejos dos clientes/usuários. Entre os aspectos importantes das linguagens de especificação de requisitos, são destacados:

- Poder de expressão: o conjunto de construtores disponíveis na linguagem deve ser rico o suficiente para permitir uma tradução (mapeamento) natural dos fenômenos do mundo real em descrições escritas na linguagem;

[8] Computer Aided Software Engineering.

- Formalidade: definição de regras precisas de interpretação, que permitam a prova de propriedades da especificação dos requisitos, bem como a possibilidade de descobrir inconsistências e/ou incompletudes.

Boehm [Boehm1989] define E.R. como uma disciplina para desenvolver uma especificação completa, consistente e não ambígua que sirva como base para um acordo entre todas as partes envolvidas, descrevendo o que o produto de software irá fazer (mas não como ele será feito). Observe que Boehm define a E.R. como um processo que envolve uma grande colaboração entre o cliente e o desenvolvedor. A especificação de requisitos funciona como um meio de comunicação para atingir um acordo acerca do software pretendido. É enfatizada também a importância de se tentar evitar decisões de projeto no momento da definição de requisitos (desejável, porém difícil de obter na prática).

Meyer [Meyer1988] afirma que especificar o documento de requisitos de um software é definir de uma forma completa e não ambígua:

- As características externas do software oferecidas aos usuários;
- A forma pela qual o software é integrado no sistema.

Já Davis [Davis1993] informa que, durante a etapa de requisitos, é necessário analisar e, portanto, entender o problema a ser resolvido. A análise do problema é a atividade que inclui o entendimento das necessidades do usuário, bem como as limitações impostas na solução.

Estudos mostram que existe um grande número de sistemas de informação que não são apropriados para as necessidades de seus usuários. De fato, o nível de aceitação para sistemas de informação comercial é da ordem de 40%, enquanto para sistemas de tempo real, o índice de aceitação sobe para 75%. Muitas vezes, o motivo de tal insucesso é devido ao fato de que os requisitos não foram bem definidos e/ou entendidos.

Do ponto de vista jurídico, desejamos que o documento de requisitos funcione como um acordo *contratual* entre os clientes e fornecedores de software. Portanto, é necessário que existam técnicas apropriadas para especificação de sistemas. A equipe responsável pelo desenvolvimento de software deve ter a *obrigação* de inquirir sobre os requisitos dos seus clientes, visando a uma melhor compreensão do sistema a ser desenvolvido. Assim, é necessário um melhor conhecimento das técnicas de aquisição de requisitos. Os desenvolvedores de software também devem ser *obrigados* a informar seus usuários sobre a solução proposta, não bastando apenas um manual de usuários. Uma forma razoável para apresentar o sistema seria através da definição apropriada dos requisitos do sistema.

Estudos realizados mostram que, cerca de 40% dos erros cometidos, ocorrem na etapa de especificação de requisitos do sistema (vide Figura 5.1). Como esses erros são os mais caros de consertar, a correção de problemas oriundos da etapa de especificação atinge um patamar de 66% do custo total de correção de erros do projeto.

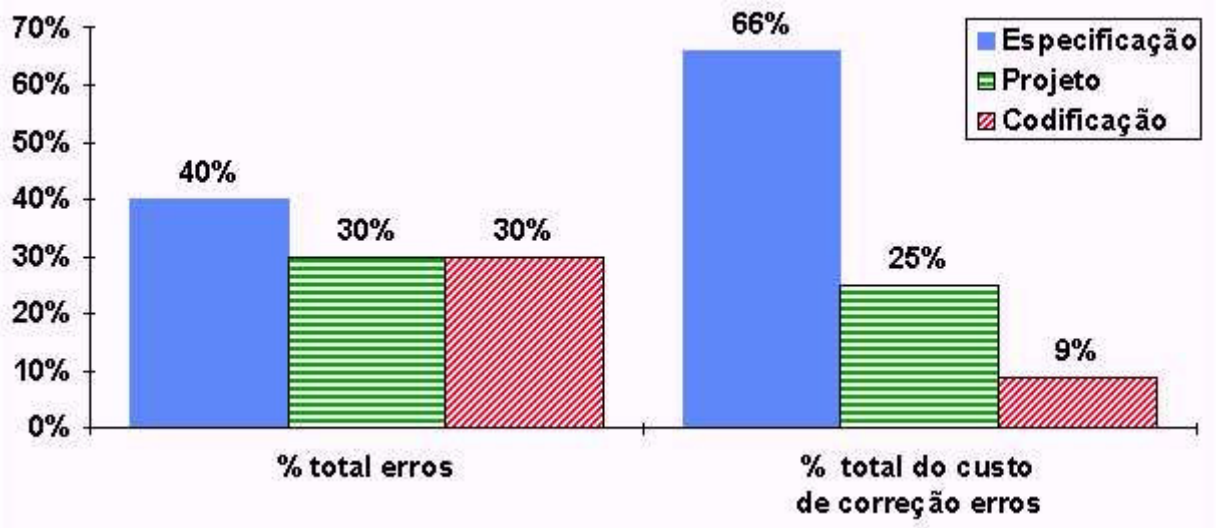


**Figura 5.1: Erros e custo de correção**

Muitos estudos têm mostrado que, quanto mais tarde for detectada uma decisão errada, mais caro será corrigi-la [Boehm1989]. Por exemplo, a correção de um erro de especificação é cerca de cinco vezes mais cara do que uma correção de um erro de codificação.

A engenharia de requisitos força os clientes a considerarem os seus requisitos cuidadosamente e revisá-los no contexto do problema. O objetivo desse processo é alcançar uma especificação completa do sistema de software. São feitas anotações e refinamentos dos requisitos, aumentando a transparência do sistema de software e melhorando a comunicação entre clientes e desenvolvedores. Um projeto só será bem sucedido se houver um consenso entre as partes envolvidas (clientes e desenvolvedores), que é representado pela especificação de requisitos.

A especificação de requisitos funciona como um padrão contra o qual o projeto e a implementação pode ter sua completude e corretude testadas. As atividades da engenharia de requisitos servem também como importante ponto inicial para as subseqüentes atividades de controle e gerência, tais como: estimativa de custo, tempo e recursos necessários.

### 5.1.1 Características Específicas da Engenharia de Requisitos

Os perfis do engenheiro de software e do engenheiro de requisitos são bastante distintos. O **engenheiro de software** é movido por tecnologia, usa uma abordagem transformacional (isto é, procura transformar uma especificação em código) e se preocupa em provar suas transformações (pelo menos num futuro próximo).

Já o **engenheiro de requisitos** se interessa tanto pelo sistema a ser construído, quanto pelo ambiente no qual ele irá funcionar. Ele precisa modelar os conhecimentos, ou seja, representar os conhecimentos adquiridos, através de uma notação adequada. Não existe a possibilidade de se provar formalmente que os requisitos modelados são, de fato, aqueles que o cliente apresentou. O processo é caracterizado por interações humanas que envolvem comunicação, aprendizagem e negociação (vide Figura 5.2).


**Figura 5.2: Engenheiro de Requisitos X Engenheiro de Software**

As tarefas do engenheiro de requisitos podem ser divididas em quatro grandes etapas [Pressman2001]:

- Reconhecimento do problema;
- Avaliação e síntese;
- Especificação;
- Validação.

A etapa de reconhecimento do problema visa entender os elementos básicos, de acordo com a percepção do cliente/usuário. O engenheiro de requisitos precisa estabelecer contato com a equipe técnica e gerencial da organização do cliente/usuário e da organização responsável pelo desenvolvimento do software. O gerente de projeto poderá atuar como coordenador, para facilitar o estabelecimento de caminhos de comunicação.

Durante a etapa de avaliação e síntese de solução, o engenheiro de requisitos deve avaliar o fluxo e estrutura de informações, refinar as funções do software em detalhe, estabelecer características da interface com o usuário e descobrir limitações

que afetam o projeto. O resultado dessa etapa é a síntese de uma solução para o problema.

As tarefas associadas com a especificação de requisitos existem para prover uma representação do software que possa ser revisada e aprovada pelo usuário. A descrição inclui informações básicas de funcionalidade, performance e interface.

Os critérios de validação são descritos visando facilitar a tarefa de determinar se uma implementação é bem sucedida, ou seja, se atende aos requisitos especificados. Esses critérios servirão como base para os testes durante o desenvolvimento do software. No caso onde não é possível elaborar um protótipo, poderá ser produzido um Manual Preliminar do Usuário.

Freqüentemente, questionam-se quais as características necessárias para um engenheiro de requisitos. São várias as tarefas realizadas, exigindo diferentes capacidades:

- Habilidade de lidar com conceitos abstratos, reorganizando-os em divisões lógicas e sintetizando as soluções de acordo com cada divisão;
- Habilidade de absorver fatos pertinentes a partir de fontes conflitantes e confusas;
- Habilidade de entender o ambiente do usuário/cliente;
- Habilidade de lidar com problemas complexos;
- Habilidade de aplicar elementos de software/hardware ao ambiente do usuário/cliente;
- Habilidade de comunicar-se bem, tanto de forma escrita, como oral;
- Habilidade de evitar detalhes desnecessários, concentrando-se nos objetivos gerais do software.

A atividade de engenharia de requisitos requer uma intensa atividade de comunicação. Durante a comunicação, problemas de omissão e má interpretação podem causar dificuldades entre o engenheiro e o cliente/usuário. Freqüentemente, informações obtidas de usuários entram em conflito com requisitos descritos anteriormente por outras pessoas. Nesses casos, é preciso negociar uma solução para o impasse. A qualidade da negociação depende de um bom entendimento e de uma análise rigorosa. Porém, essa atividade não é trivial, pois os indivíduos sabem bem mais do assunto do que são capazes de informar (o chamado de conhecimento *tácito*) e nem sempre desejam, necessariamente, um sistema baseado em computadores.

Grandes sistemas de software possuem, usualmente, uma clientela bastante heterogênea. Diferentes clientes desfrutam de variadas prioridades. Diferentes requisitos possuem diversos graus de importância. Dificilmente, quem encomenda o sistema (o cliente, responsável pelo pagamento) será o usuário (principal) do software.

É comum a imposição de requisitos que são devidos à questão organizacional ou limitações financeiras da empresa. Essas solicitações podem entrar em conflito com os requisitos dos usuários. Portanto, as visões parciais dos clientes muitas vezes não são consistentes entre si.

A evolução dos sistemas é um outro aspecto de suma importância. Sabemos que, independentemente da etapa de desenvolvimento de um sistema, ocorrerão mudanças nos requisitos. Portanto, devemos abordar a questão da coordenação das mudanças dos requisitos, seu impacto em outras partes do software e como corrigir erros de especificação, de forma que efeitos colaterais sejam evitados (ou minimizados). Muitas vezes, as mudanças são realizadas apenas no código, causando uma divergência entre o sistema de software implementado e sua especificação de requisitos. Assim, entre as metas da engenharia de requisitos estão:

- Propor técnicas de comunicação que visem facilitar a aquisição de informações;
- Desenvolver técnicas e ferramentas que resultem em especificações de requisitos adequadas e precisas;
- Considerar alternativas na especificação de requisitos.

A produção de uma boa especificação de requisitos não é uma tarefa fácil, tendo em vista os problemas já expostos. A seguir, são enumeradas as propriedades que uma especificação apropriada deve satisfazer [Stokes1994]:

- *Não ambigüidade*: todas as especificações devem, idealmente, ter uma única interpretação. Essa é uma propriedade difícil de ser alcançada, até mesmo através da aplicação de métodos formais;
- *Completude*: uma especificação de requisitos deve descrever cada aspecto significante e relevante do sistema e deve incluir detalhes a respeito de todas as informações. A natureza subjetiva da definição de completude faz com que essa propriedade seja impossível de ser garantida;
- *Consistência*: não devem existir requisitos contraditórios na especificação;
- *Verificabilidade*: quando o sistema for projetado e implementado, deverá ser possível verificar se seu projeto e implementação satisfazem os requisitos originais;
- *Validação:* o usuário/cliente deve ser capaz de ler e entender a especificação de requisitos e, então, indicar se os requisitos refletem as suas idéias;
- *Modificação*: como os requisitos estão freqüentemente sujeitos a mudanças, todas as especificações de requisitos devem permitir que alterações sejam feitas facilmente, sem a necessidade de que tais modificações sejam realizadas em toda a especificação. Isso exige, geralmente, que alguma estruturação seja imposta na especificação;

- *Compreensão*: clientes, usuários, analistas, projetistas e engenheiros devem ser capazes de entender os requisitos. Como eles possuem formações diferentes, podem preferir utilizar distintas representações para os requisitos;
- *Rastreamento*: devem ser feitas referências entre os requisitos, aspectos de projeto e implementação. Dessa forma, efeitos das modificações nos requisitos, projeto e implementação serão controlados.

### 5.1.2 Requisitos Funcionais e Não-Funcionais

Uma forma de melhorar a compreensão dos requisitos é dividi-los em requisitos *funcionais* e *não-funcionais*. Embora as suas fronteiras nem sempre sejam precisas de se determinar, essa divisão tem sido bastante usada na literatura.

Os *Requisitos Funcionais* definem as funções que o sistema ou componentes do sistema devem executar. Eles descrevem as transformações do sistema ou de seus componentes que transformam entradas em saídas.

Os *Requisitos Não-Funcionais,* também referidos com requisitos de qualidades, incluem tanto limitações no produto (performance, interface de usuários, confiabilidade, segurança, interoperabilidade), como limitações no processo de desenvolvimento (custos e tempo, metodologias a serem adotadas no desenvolvimento, componentes a serem reutilizados, padrões a serem aderidos etc.).

### 5.1.3 O Processo de Engenharia de Requisitos

Esta seção apresenta brevemente as atividades do processo da engenharia de requisitos, mostrando as relações existentes entre elas. Para nossos propósitos, o processo será dividido nas seguintes atividades: elicitação de requisitos, modelagem de requisitos, análise de requisitos e validação de requisitos.

As atividades de *elicitação* e *validação* de requisitos correspondem à etapa de *aquisição* de requisitos, enquanto as atividades de *modelagem* e *análise* de requisitos correspondem à etapa de *especificação de* requisitos (vide Figura 5.3).

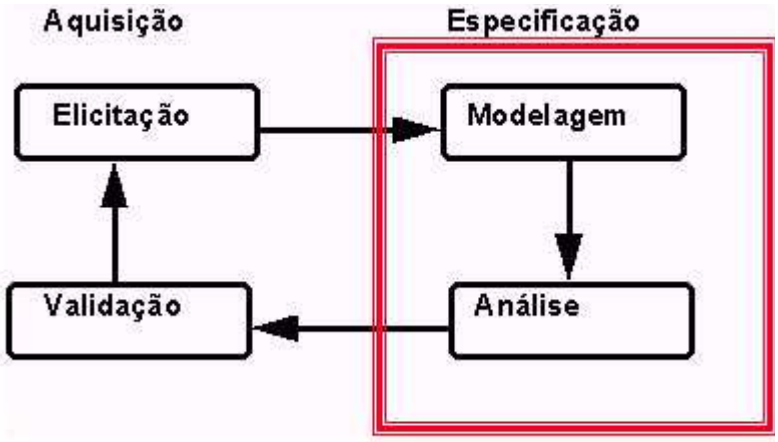


**Figura 5.3: Processo de engenharia de requisitos**

***Elicitação***

Na atividade de *elicitação*, o engenheiro de requisitos procura capturar os requisitos do sistema, buscando também obter um conhecimento do domínio do problema. Usualmente, são feitas entrevistas com o cliente e/ou é consultado material existente descrevendo objetivos e desejos da organização. Também se observam sistemas similares. É importante observar que o uso apenas de entrevista não é suficiente para obter todas as informações necessárias. O engenheiro de requisitos precisa se envolver com o trabalho do cliente, se misturar com os funcionários, observar, aprender e questionar.

Um dos objetivos da etapa de elicitação deve ser o entendimento da razão pela qual os procedimentos atuais do cliente são feitos da forma que são. Trata-se, portanto, de uma atividade de aprendizagem:

- Do comportamento de sistemas existentes, incluindo:
  - Procedimentos manuais;
  - Engenharia reversa de softwares existentes;
  - Interfaces.
- Do conhecimento do domínio de aplicação, ou seja, da parte específica do domínio que está relacionada com o sistema de software a ser implementado;
- Dos objetivos e limitações dos usuários, incluindo:
  - Limitações funcionais;
  - Limitações organizacionais.

### *Modelagem*

Os resultados da etapa de elicitação são documentados em forma de modelos conceituais, que descrevem, estática e dinamicamente, aspectos do problema e do domínio de aplicação. O modelo conceitual seleciona propriedades do domínio de aplicação que são de interesse do cliente, mas ignora detalhes e relações que podem ser importantes para outros propósitos. Nesse trabalho, o interesse maior é na parte de modelagem e de análise.

### *Análise*

Durante a atividade de análise, o objetivo é a obtenção de uma especificação que seja consistente e completa. Pelas razões expostas acima, é muito provável que a descrição obtida até então possua várias inconsistências. O engenheiro de requisitos, durante a análise, deve ser capaz de detectar e resolver inconsistências. A análise é intercalada com a elicitação, pois problemas podem ser descobertos quando os requisitos são elicitados.

A especificação pode estar incompleta, com muitas implicações que precisam ser explicitadas. Um grande esforço deve ser empreendido para completar a especificação de requisitos, embora saibamos que tal tarefa é praticamente impossível.

É bastante comum a existência de omissões no que diz respeito a condições de erro ou exceções. Porém, principalmente em aplicações que são do tipo crítico/segurança (controle de tráfego aéreo, reatores nucleares etc.), um esforço deve ser feito no sentido de definir o comportamento do sistema em situações não desejadas.

Trabalhos recentes [Gotel1994] mostram a importância de acompanhar (rastrear) os requisitos, desde a sua concepção, passando por sua especificação, até a sua implementação. Isso se torna bastante importante, por exemplo, quando da necessidade de gerenciar/modificar o sistema.

### *Validação*

Quando validamos uma especificação estamos garantindo que ela, de fato, retrata as necessidades do cliente. Se a validação não for adequada, mal entendidos e erros serão propagados para as etapas de projeto e implementação, ocasionando, como foi visto anteriormente, um alto custo da correção.

Geralmente, um nível aceitável de validação é obtido através do teste de aceitação, que é uma técnica de confirmação que garante que a especificação de software está de acordo com um critério pré-estabelecido pelo cliente.

Quando uma especificação de requisitos é executável (um protótipo), o trabalho de validação é bastante facilitado, pois os usuários podem experimentar com partes da especificação, dando imediatamente suas impressões sobre o sistema. Dessa forma, os próprios usuários contribuem para que o software seja adequado às suas necessidades, reduzindo custos e colaborando para que os prazos sejam cumpridos.

### 5.1.4 Modelos Conceituais e Linguagens para a Engenharia de Requisitos

Idealmente, dever-se-ia usar linguagens que possuíssem uma única interpretação com o mundo real, ou seja, que cada parte da linguagem (ex. palavra) tivesse um único significado no mundo real. Se isso fosse sempre possível, o engenheiro de requisitos seria capaz de fazer descrições que fossem “não ambíguas” e que pudessem ser corretamente entendidas por usuários/clientes e engenheiros/projetistas (vide Figura 5.4). Nessa seção, serão discutidas as principais abordagens de linguagens para representação de requisitos, isto é, linguagens naturais, linguagens rigorosas e linguagens formais.

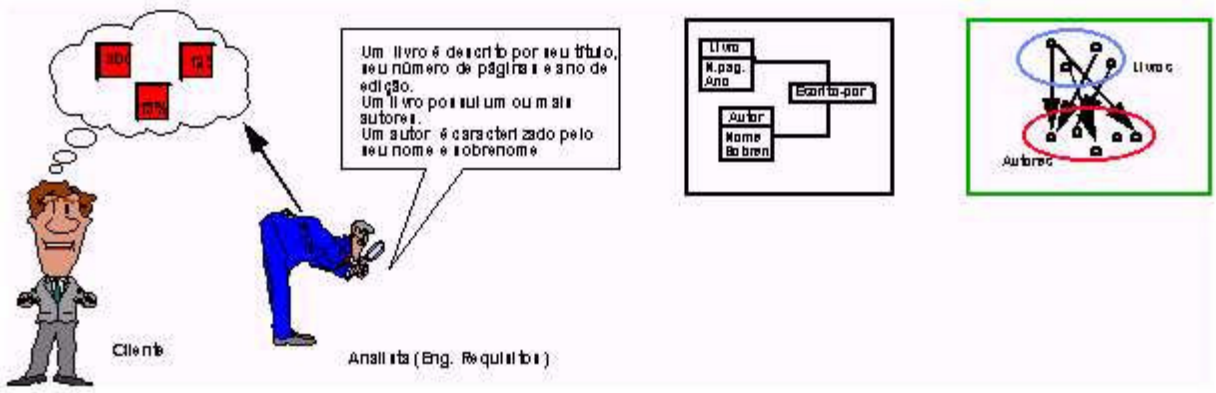


**Figura 5.4: Linguagens para especificação de requisitos**

***Linguagens Naturais***

Obviamente, a *linguagem natural* é a que oferece maior grau de compreensão e acesso para o início de uma descrição [Ford1994]. No nosso caso, o Português poderia ser adotado. O uso de linguagem natural é bastante atrativo pois:

- Usa o conhecimento já existente da linguagem pelo engenheiro de requisitos;
- Permite que as idéias sejam comunicadas de um modo que seja entendida pelo público em geral;
- Possui a vantagem de não fazer nenhuma tentativa de prescrever um método para implementação.

Infelizmente, há muitos problemas com as linguagens naturais [Ryam1993]. Inicialmente, os textos tendem a ser desestruturados. As pessoas tendem a escrever de uma forma desorganizada. Somente após muito treino é que um escritor se habitua a estruturar suas descrições e observações. Também, na prática, a linguagem natural torna-se ambígua, dando margem a interpretações diferentes, dependendo dos diferentes tipos de pessoas e suas experiências. Torna-se inviável determinar se a especificação está completa.

De fato, é muito difícil usar linguagens naturais na especificação de requisitos sem deixar nenhuma ambigüidade ou má interpretação para leitores com diferentes experiências. A origem do problema está no uso de palavras e estruturas que sejam interpretadas de formas diferentes pelo leitor. Uma determinada característica do sistema em desenvolvimento pode ser tão óbvia que ninguém terá a preocupação de escrevê-la, pois a necessidade dela é perfeitamente entendida por todos os membros da equipe.

Como conseqüência, essa característica pode nunca ser incluída na especificação e, portanto, não ser implementada. Muitas vezes, é feita uma super-especificação, abordando detalhes que na realidade são de projeto ou implementação. Especificações em linguagem natural podem apresentar redundâncias e contradições difíceis de detectar.

Na prática, existem sugestões que a linguagem natural seja usada de uma forma estruturada, muitas vezes refraseando especificações para uma linguagem baseada numa linguagem de programação de alto-nível. Pelo menos na teoria, o uso dessas linguagens pode ser eficiente para remover ambigüidades e más interpretações dos requisitos do sistema. A maior desvantagem dessa técnica é a dificuldade do engenheiro de requisitos de se expressar de forma independente do paradigma de implementação. Logo, há um risco maior da especificação ser escrita mais em termos de "*como* ela será implementada", do que em termos do "*que* deve ser entregue".

### *Linguagens Rigorosas*

Desde a década de 1970, tem sido desenvolvido um grande número de técnicas, métodos e ferramentas que ajudam na produção de requisitos. As descrições dos requisitos passaram a ter uma sintaxe mais formal, muitas vezes usando gráficos, acompanhados de informações textuais. Apesar de parte da especificação ainda ser informal, as linguagens passaram a possuir uma limitada interpretação formal. Isso tem permitido encontrar contradições e incompletudes básicas de uma forma mais sistemática.

A linguagem UML [Booch1998] é um exemplo de uma notação que se enquadra nessa categoria. Nessa linguagem, os requisitos funcionais são expressos através de

diagramas de casos de uso, os quais possuem uma sintaxe precisa. No entanto, a especificação mais detalhada dos casos de uso e a especificação dos requisitos não-funcionais ainda continuam a serem feitas usando-se uma linguagem natural.

***Linguagens Formais***

As técnicas de especificação formal são baseadas em notações matemáticas, que possuem regras de interpretação não-ambíguas. Por intermédio de um sistema de inferência, é possível deduzir (provar) propriedades do sistema. Muitas vezes, essas técnicas são acompanhadas por uma metodologia que facilita tanto a especificação, como a verificação formal.

As vantagens do uso de técnicas formais são: clareza, não-ambigüidade, independência de implementação e completude da especificação. Os métodos formais existentes não são muitos atrativos, exatamente pela necessidade de se ter uma certa base matemática para se usufruir totalmente da técnica, pois essa habilidade não é encontrada na maioria dos usuários. Contudo, experiências recentes sugerem que, motivados a escrever um software de alta-qualidade (talvez por ser um software crítico), o desafio de usar um método formal é algo a ser considerado, mesmo para aqueles que não têm maiores inclinações matemáticas. Uma vantagem adicional é a possibilidade de se obter especificações que sejam executáveis, permitindo que um protótipo do sistema seja produzido rapidamente.

Existem limitações nas linguagens formais, decorrentes do sistema matemático usado. Muitas vezes, o formalismo escolhido pode não ser apropriado para modelar todas as propriedades do sistema. Por exemplo, as questões do tempo e de concorrência exigem uma base matemática apropriada. Dentre as linguagens de especificação formal, Z [Spivey1989] tem sido uma das mais usadas.

### 5.1.5 Considerações

Durante a última década, as pesquisas têm se concentrado em como descrever os requisitos de grandes sistemas feitos sob encomenda. Atualmente, verifica-se a existência de um mercado de distribuição para as ***massas***. Estes softwares orientados para massa raramente usam especificação de requisitos, pelo menos do tipo tradicional. Eles se baseiam em pesquisas de mercado, grupos de interesse e *feedback* de usuários que fizeram testes em versões preliminares do software. Portanto, a engenharia de requisitos precisa se adequar a essa nova realidade.

Entre os vários problemas que precisam ser mais bem estudados, estão os desafios organizacionais e gerenciais. As equipes de gerenciamento, geralmente, não têm consciência do papel estratégico da engenharia de requisitos [Lubars1993]. O envolvimento e o comprometimento da gerência na E.R. são, freqüentemente, muito

baixos. Isso implica que os trabalhos de E.R. normalmente não estão relacionados com as visões de negócios e objetivos, processos de reengenharia de negócios e outras mudanças organizacionais. As organizações tendem a não “aprender” com as experiências anteriores de E.R. As especificações de requisitos não são usadas como fonte de conhecimento para futuros projetos de engenharia de requisitos.

Existe uma tendência das empresas, que antes desenvolviam software, de terceirizar sua produção. Como conseqüência, será necessário que os usuários de software melhorem suas práticas de engenharia de requisitos. Eles terão de ser mais exigentes para conseguir o que querem. No desenvolvimento tradicional de requisitos, o desenvolvedor de software é o provável executor de muitas das atividades de engenharia de requisitos descritas anteriormente. Contudo, com a terceirização, os próprios clientes/usuários deverão ser especialistas em requisitos.

Finalmente, é necessário elaborar novas formas de participação de usuários na engenharia de requisitos, pois ainda existem problemas de comunicação entre desenvolvedores e clientes, ocasionando um baixo nível de validação dos requisitos. Os usuários acreditam que os desenvolvedores conhecem seus “requisitos tácitos” e que “validam” as especificações de requisitos sem entenderem completamente suas implicações.

## 5.2 PROJETO DE SOFTWARE

Nesta etapa, os requisitos do sistema são particionados em nível de sub-sistemas de hardware ou software, a fim de que a estrutura interna (arquitetura) seja descrita. Na prática, existe um forte relacionamento entre a etapa de projeto e a especificação do sistema, pois o projetista interage entre o projeto e a especificação, de modo que, ao final desse processo, cada uma das partes componentes do sistema (sub-sistemas) tenham sido especificadas. Por esse motivo, a especificação do software é, muitas vezes, vista como a primeira etapa do projeto do sistema.

Na etapa de projeto, mais e mais detalhes são adicionados à especificação do sistema, de modo a definir os algoritmos e as estruturas de dados concretos a serem usados. Assim, pode-se dizer que a etapa de definição e análise de requisitos determina o que o sistema deve fazer e a etapa de projeto determina como a funcionalidade do sistema deve ser implementada. Os dois principais enfoques para o projeto de sistemas são: o “*top-down*” e o “*bottom-up*”.

No enfoque “*top-down*”, um sistema é recursivamente decomposto em sub-sistemas (geralmente chamados de módulos) até que sub-sistemas “tratáveis” (facilmente implementáveis) sejam identificados. A estrutura geral que resulta desse processo de decomposição é uma hierarquia em forma de árvore, na qual o sistema é a raiz da árvore e os sub-sistemas são seus nós. Essa hierarquia pode se degenerar

em um grafo, à medida que as possibilidades de reuso dos sub-sistemas são identificadas. Esse enfoque é mais utilizado para o desenvolvimento de sistemas seguindo o paradigma imperativo, no qual o estado de um sistema é centralizado e compartilhado entre as funções que operam sobre este estado. Nesse caso, as folhas da árvore correspondem às funções a serem implementadas.

No enfoque "*bottom-up*", o sistema é visto como uma coleção de "*building-blocks*". Este enfoque é mais utilizado para o desenvolvimento de sistemas seguindo o paradigma orientado a objetos, no qual o estado de um sistema é descentralizado entre os objetos que compõem o sistema, ou seja, cada objeto manipula seu próprio estado. Objetos são membros de classes e têm um conjunto de atributos, os quais definem seus estados e operações que agem sobre os atributos. Ou seja, classes definem atributos e operações relacionados a objetos membros. Cada classe pode herdar o comportamento (atributos e operações) de superclasses e, dessa forma, é possível definir subclasses.

Assim como na etapa de especificação e análise de requisitos, métodos estruturados também são usados na etapa de projeto. Nesse caso, esses métodos possibilitam a descrição da arquitetura dos sistemas através de diagramas.

Um método estruturado é composto por uma notação diagramática e um conjunto de diretrizes de como utilizar a notação. Atualmente, a notação UML [Booch1998] tem sido bastante utilizada na etapa de projeto. No entanto, esta notação não corresponde realmente a um método, pois pode ser usada segundo várias diretrizes existentes (ou metodologias), tais como RUP [Kruchten1998] e Catalysis [D'Souza2001]. A seguir descreveremos parte desta metodologia.

## 5.3 IMPLEMENTAÇÃO

Durante essa etapa, o projeto do software é implementado como um conjunto de unidades de uma linguagem de programação (ex: programas ou classes). Essa etapa baseia-se totalmente na disponibilidade de ferramentas e/ou ambientes de apoio à programação (ex: compiladores, depuradores de código e editores sintáticos).

O paradigma da linguagem de programação utilizada nessa etapa está diretamente relacionado ao paradigma utilizado no projeto do sistema. Assim, ao se fazer um projeto orientado a objetos, por exemplo, é recomendado implementá-lo em uma linguagem que siga o mesmo paradigma.

Independentemente do paradigma utilizado na implementação do software, a qualidade do código produzido pode ser julgada com base em alguns atributos:

- Qualidade de sua documentação;
- Uso de um padrão de codificação;

- Legibilidade;
- Nível de complexidade;
- Tamanho;
- Tempo de execução.

A utilização de padrões de documentação ajuda na obtenção da qualidade do código. Esses padrões vão desde a definição do ambiente de programação até de comentários no próprio código. Veja o exemplo de padrão de codificação JAVA no Anexo A.

Há vários outros atributos para tal julgamento. Alguns desses atributos podem ser avaliados através de atividades de revisão e inspeção, descritas no segundo módulo deste curso, outros podem ser avaliados através da etapa de testes, tratada na próxima seção.

## 5.4 TESTES DE SOFTWARE

Existe grande possibilidade de injeção de falhas no processo de desenvolvimento de software. Assim, os custos associados às falhas de software justificam uma atividade de teste cuidadosa e bem planejada, como parte dos esforços, no sentido de garantir a qualidade do software.

### 5.4.1 Introdução

Os testes representam a última oportunidade de detectar erros antes de o software ser entregue aos usuários. A atividade de testes pode ser feita de forma manual e/ou automática e tem por objetivos:

- Produzir casos de teste que tenham elevadas probabilidades de revelar um erro ainda não descoberto, com uma quantidade mínima de tempo e esforço;
- Comparar o resultado dos testes com os resultados esperados, a fim de produzir uma indicação da qualidade e da confiabilidade do software. Quando há diferenças, inicia-se um processo de depuração para descobrir a causa.

A realização de testes **não** consegue mostrar ou provar que um software ou programa está correto. O máximo que os testes de um software conseguem provar é que ele contém defeitos ou erros. Quando os testes realizados com um determinado software não encontram erros, haverá sempre duas possibilidades:

- A qualidade do software é aceitável;
- Os testes foram inadequados.

### 5.4.2 Estágios de Teste

Existem diferentes estágios de teste associados ao desenvolvimento de um produto de software:

- **Teste de unidade:** visa testar *individualmente* cada um dos *componentes* (programas ou módulos), procurando garantir que funcionem adequadamente;
- **Teste de integração**: visa testar o relacionamento entre as diversas unidades integradas. Em outras palavras, garantir que a interface entre os módulos funcione adequadamente, pois não há garantias de que unidades testadas em separado funcionarão em conjunto;
- **Teste de sistema:** conjunto de testes cujo objetivo primordial é colocar completamente à prova todo o sistema, baseado em um computador. Em outras palavras, testa a integração do software com o ambiente operacional - hardware, pessoas e dados reais;
- **Teste de aceitação (homologação):** são testes realizados pelo cliente/usuário com o objetivo de validar o sistema a ser implantado. A motivação maior para esses testes é o fato do desenvolvedor nunca conseguir prever como o usuário realmente usará um software numa situação real. Os testes de aceitação podem ser de dois tipos:
  - ***Testes alfa*:** são feitos por um determinado cliente, geralmente nas instalações do desenvolvedor, que observa e registra os erros e/ou problemas;
  - ***Testes beta*:** são realizados por possíveis clientes, em suas próprias instalações, sem a supervisão do desenvolvedor. Cada cliente relata os problemas encontrados ao desenvolvedor, posteriormente.

### 5.4.3 Abordagens de Teste

Existem basicamente duas abordagens que podem ser aplicadas aos diferentes tipos de teste:

- **Abordagem funcional (caixa-preta):** concentra-se nas interfaces do software e visa mostrar que: as entradas são aceitas; as saídas são as esperadas; a integridade dos dados é mantida. Aplica-se, principalmente, aos testes de validação, sistemas e aceitação, mas pode ser também usada com os testes unitários;
- **Abordagem estrutural (caixa-branca):** visa mostrar que os componentes internos do software (programas) realmente funcionam. Em princípio, os testes de caixa branca pretendem garantir que **todas** as estruturas dos programas e todos os possíveis casos sejam testados. Aplica-se, principalmente, aos testes

unitários e de integração. Na prática, mesmo para pequenos programas, é geralmente impossível testar todas as possibilidades.

### 5.4.4 Tipos de Teste

Existem vários tipos de teste que podem ser executados nos diversos estágios de teste e utilizando as diferentes abordagens existentes:

- **Teste de funcionalidade:** testa a funcionalidade geral do sistema, em termos de regras de negócio (fluxo de trabalho), considerando-se tanto as condições válidas como as inválidas; **Teste de recuperação de falhas:** seu objetivo é forçar o software a falhar de diversas maneiras e verificar se a recuperação é adequada;
- **Teste de segurança de acesso:** tenta certificar-se de que todos os mecanismos de proteção embutidos no software, de fato, o protegerão dos acessos indevidos;
- **Teste de carga:** tenta confrontar o software ou os programas com situações anormais. Ele executa o software de uma forma que exige recursos em quantidade, freqüência e volume bem maiores do que o uso normal;
- **Teste de desempenho:** são testes que visam verificar o desempenho ou *performance* do software. São, muitas vezes, combinados ou feitos juntamente com os testes de estresse. São comuns em software de tempo real;
- **Teste de portabilidade:** são testes que verificam o grau de portabilidade do produto de software em diferentes ambientes de hardware/software.

A execução de todos esses tipos de testes pode ser feita de forma combinada. Ou seja, nenhum desses testes exclui o outro.

### 5.4.5 Responsabilidade pelos testes

Os testes, normalmente, são feitos pela equipe de desenvolvimento, i.e., por engenheiros de software. No entanto, é ideal que pelo menos os testes de sistemas sejam feitos por uma equipe de testes independente. É importante que os grandes sistemas e programas também sejam testados por outras pessoas que não os seus desenvolvedores, e que tais pessoas sejam bastantes críticas. Assim, seria possível a seguinte distribuição de responsabilidades:

**Equipe de desenvolvimento:**

- Testes de unidades;
- Testes de integração.

**Equipe independente de testes:**

- Testes de sistema.

**Usuário:**

- Teste de aceitação.

### 5.4.6 Ferramentas de Teste

O processo de testes pode ser automatizado, através do uso de ferramentas CASE específicas para essa atividade. Alguns tipos de ferramentas de apoio aos testes são descritos a seguir:

- **Ferramenta de geração de massa de dados:** gera dados para serem usados em testes. A geração dos dados é freqüentemente baseada em regras de formação definidas pelos casos de teste;
- **Ferramenta de teste de API:** testa método a método de uma classe, a partir de uma série de combinações de parâmetros. Utiliza a abordagem caixa-preta para cada método;
- **Ferramenta de teste de GUI:** grava (em um *script*) a execução da interface gráfica (cliques do mouse e entradas de teclado) e repete a entrada quantas vezes forem necessárias. O *script* gerado pode ser modificado através do uso de uma linguagem própria de *script;*
- **Ferramenta de teste de cobertura:** após a execução da aplicação, indica quais os trechos do código foram ou não executados, bem como o número de vezes que determinado método/trecho foi executado. Utiliza a abordagem caixa-branca;
- **Ferramenta de teste de carga e stress:** simula acessos simultâneos a uma aplicação multi-usuário, bem como o envio e recuperação de altos volumes de dados;
- **Ferramenta de teste de desempenho/gargalos:** analisa o desempenho do software, quando em execução, e detecta potenciais pontos de gargalo no código fonte.

## 5.5 GERÊNCIA DE CONFIGURAÇÃO

O desenvolvimento de software é uma atividade que precisa ser executada em equipe, principalmente para os softwares de grande porte. Freqüentemente, artefatos (ex: documentos, programas etc.) precisam ser acessados e alterados por várias pessoas da equipe e, nesse caso, a modularização do projeto normalmente não é suficiente para resolver esse problema.

A Gerência de Configuração controla as mudanças e mantém a integridade dos artefatos de um projeto. As principais atividades envolvidas com essa etapa do ciclo de vida são:

- Identificação de itens de configuração (componentes individuais do software – ex: arquivos com programas fonte);
- Restrição às modificações nesses itens (controle de acesso);
- Criação de versões dos itens modificados;
- Auditoria nos itens de configuração (para determinar por que, quando e por quem um artefato foi modificado);
- Retorno a uma versão anterior de um artefato.

Estas atividades, geralmente, são apoiadas por ferramentas de controle de versões e mudanças. Essas ferramentas permitem tornar o processo de desenvolvimento mais produtivo, uma vez que a gerência de configuração é aplicada a todos os artefatos produzidos em todas as etapas do ciclo de vida. Para garantir que todas as atividades desta etapa sejam executadas de maneira satisfatória, recomenda-se que seja produzido um Plano de Gerência de Configuração, o qual irá determinar quando, por quem e com que ferramentas as atividades dessa etapa serão executadas.

## 5.6 OPERAÇÃO E MANUTENÇÃO DE SOFTWARE

Nessa etapa, o sistema é posto em uso (operação). Erros e omissões nos requisitos originais, nos programas e no projeto são descobertos. A necessidade de novas funcionalidades e melhorias é identificada. Conseqüentemente, modificações serão necessárias para que o software continue sendo usado. Essas modificações são chamadas: manutenção do software.

A Manutenção pode envolver mudanças nos requisitos, no projeto e/ou na implementação. Conseqüentemente, podem ser necessários novos testes. Assim sendo, todo o processo de desenvolvimento (ou parte dele) pode ser repetido quando são feitas modificações.

As manutenções podem ser classificadas em:

- Corretivas;
- Adaptativas;
- Perfectivas;
- Preventivas.

A manutenção é dita corretiva quando é feita para corrigir os erros não identificados durante o desenvolvimento e testes do sistema. Esse tipo de manutenção existe porque os testes de software dificilmente conseguem detectar todos os erros.

A manutenção é dita adaptativa quando alterações se tornam necessárias por conta de mudanças no ambiente computacional. Essas manutenções são necessárias porque a vida útil dos aplicativos é longa e não acompanha a rápida evolução da computação.

A manutenção é dita perfectiva quando alterações visam melhorar o software de alguma forma. Geralmente, é o resultado de recomendações de novas capacidades, bem como modificações em funções existentes solicitadas pelos usuários.

A manutenção é dita preventiva quando modificações são feitas com o objetivo de melhorar o software no que se refere à sua confiabilidade ou manutenibilidade, ou para oferecer uma base melhor para futuras ampliações.

Independentemente do tipo de manutenção, existe a certeza de que essa atividade irá ocorrer no desenvolvimento de um software. Sendo assim, o processo de software utilizado por uma organização, bem como as pessoas envolvidas nessa atividade devem estar preparadas para desempenhá-la da forma mais produtiva possível, de maneira a atender rapidamente às necessidades dos clientes.

## 5.7 FERRAMENTAS CASE

Como qualquer outra engenharia, a Engenharia de Software envolve trabalho criativo e também uma grande quantidade de trabalho tedioso de administração e de controle. A execução das atividades não criativas pode, na maioria das vezes, ser automatizada através do uso de ferramentas CASE. O suporte automatizado ao processo de desenvolvimento de software é uma “arma” fundamental para aumentar a produtividade na execução das atividades e garantir a qualidade de um produto de software.

Uma ferramenta CASE oferece um conjunto de serviços fortemente relacionados para apoiar uma ou mais atividades do processo de software. Esses serviços vão, desde a simples edição de textos, até o gerenciamento de configurações, o teste de software e a verificação formal de programas.

As ferramentas CASE são, geralmente, projetadas para funcionar individualmente, com objetivo específico. Por exemplo, apoiar um método de análise orientada a objetos ou a programação em uma linguagem específica. Essas ferramentas, usualmente, possuem repositórios de dados mais sofisticados que simples arquivos, podendo incluir dicionários de dados ou até mesmo utilizar um gerenciador de banco de dados comercial para armazenar seus dados. No entanto,

existem conjuntos de ferramentas que trabalham integradas e que são, geralmente, conhecidos como *Tool suites* ou Ambientes CASE. Esses ambientes podem ser de propósito específico (ex: conjunto de ferramentas que dão apoio a uma única etapa do ciclo de vida de software) ou de propósito genérico (ex: conjunto de ferramentas que dão apoio a diversas atividades do ciclo de vida).

Vale salientar que, para se extrair todos os benefícios do uso de ferramentas CASE, é necessário que as ferramentas selecionadas se adeqüem ao processo de desenvolvimento de software da organização. Assim, é primordial que tal processo esteja bem definido e institucionalizado, para que as ferramentas adequadas sejam selecionadas.

# 6

# QUALIDADE DE SOFTWARE

Em tempos de competitividade acirrada, a consciência da necessidade de processos de produção mais eficientes, que garantam o equilíbrio perfeito entre qualidade e produtividade, vem crescendo substancialmente. Nesse contexto, o fator qualidade tem sido considerado fundamental para o sucesso de qualquer organização.

O termo “qualidade” no contexto organizacional é, em geral, relacionado a uma série de aspectos, tais como normalização e melhoria de processo, medições, padrões e verificações, entre outros. Esses aspectos vêm sendo abordados ao longo dos últimos 50 anos por vários estudiosos no assunto, considerados verdadeiros mestres, cujas propostas marcaram um caminho evolutivo que fundamenta, até hoje, diversos modelos de qualidade internacionais.

Inserido nesse contexto, esse capítulo compreende conceitos básicos e uma visão da evolução da qualidade e produtividade, essenciais para a obtenção de conhecimento abrangente no assunto.

## 6.1 CONCEITUAÇÃO

Apesar da atenção substancial que se tem dado ao assunto, muitas organizações têm dificuldade de definir o termo “qualidade”, tão cobiçado no mercado competitivo. Dessa forma, antes de tudo, é preciso entender o que significa qualidade no contexto do setor produtivo.

Vários significados têm sido dados no sentido de prover um entendimento adequado do termo “qualidade” no contexto da produtividade. *Atendimento às expectativas do cliente*, *conformidade com a especificação*, *conhecimento do processo para melhorá-lo*, *efetividade e usabilidade*. Esses aspectos poderiam ser consolidados na definição dada pela *International Organization for Standardization* (ISO), segundo a qual qualidade é “a totalidade das características de uma entidade que lhe confere a capacidade de satisfazer às necessidades explícitas e implícitas” (NBR ISO 8402). Necessidades explícitas são as condições e objetivos propostos por aqueles que

produzem o software. As necessidades implícitas são necessidades subjetivas dos usuários, também chamadas de fatores externos, e podem ser percebidas tanto pelos desenvolvedores quanto pelos usuários.

Em outras palavras, a definição da ISO confirma a relação entre o grau de qualidade e a satisfação do cliente, em termos de expectativas atendidas.

## 6.2 EVOLUÇÃO DOS CONCEITOS DE QUALIDADE

Os avanços tecnológicos e a crescente preocupação na eliminação de defeitos, aumento na produtividade e redução de custos motivaram o surgimento de modelos de qualidade para o processo de manufatura. A partir da década de 1960, começaram a surgir critérios, modelos e técnicas para a garantia da qualidade no processo de produção.

A indústria japonesa foi a precursora do Controle da Qualidade Total (*Total Quality Control - TQC*), seguida pelos americanos, que definiram o modelo de Gerência da Qualidade Total (*Total Quality Management*), ambos bastante utilizados em todo o mundo. Em 1947, a criação da Organização Internacional de Padronização (ISO) formalizou a necessidade da definição de padrões internacionais no setor da indústria e muito contribuiu para a evolução do setor, definindo normas para a garantia da qualidade direcionadas para produção, serviços e gerenciamento, entre outros contextos. Como exemplo de padrão internacional de qualidade temos a norma ISO9001:2000 [ISO9001:2000] que define requisitos para gerência de sistemas da qualidade, abrangendo todo o ciclo de desenvolvimento de um produto, desde seu pedido, passando pela análise e gerenciamento de requisitos, projeto e fabricação, até sua entrega ao cliente, incluindo infra-estrutura adequada, competências e comprometimento da alta administração.

Observando contribuições ao longo dos últimos anos, algumas iniciativas podem ser consideradas verdadeiros marcos na história evolutiva da qualidade, além da criação da ISO. O início dessa evolução está situado no início do século XX, quando foi criada a Comissão Internacional Eletrotécnica (IEC) em 1906. Abordagens como a introdução da noção de gestão da qualidade por Feigenbaum, em 1957, e do aperfeiçoamento e novas abordagens por Deming, Juran e Crosby, seguidos da proposta do *Total Quality Management,* por Tom Peters, foram fundamentais e são até hoje consideradas nas propostas atuais sobre o assunto.

A seguir são apresentadas as diferentes abordagens defendidas ao longo da evolução da área da qualidade e produtividade.

### 6.2.1 Abordagem de W. Edwards Deming

W. Edwards Deming é reconhecido como o grande mestre do controle da qualidade. Sua abordagem defende que a qualidade inicia com o alto nível gerencial e é uma atividade estratégica. A proposta de Deming enfatizou a necessidade dos métodos estatísticos, participação, educação e proposta de melhoria. Segundo Deming, é fundamental que as especificações sejam sempre revistas, uma vez que os "desejos" do cliente têm um alto grau de instabilidade.

Deming defendeu 14 pontos fundamentais, que podem ser considerados recomendações para o sucesso de um programa de qualidade em uma organização, os quais podem ser resumidos nas seguintes recomendações:

1. Estabeleça um propósito constante direcionado à melhoria de produtos e serviços;
2. Adote uma nova filosofia. O gerenciamento moderno deve estar atento aos desafios, aprender suas responsabilidades e liderar as mudanças;
3. Elimine a necessidade de inspeção em massa, produzindo produtos de qualidade;
4. Elimine a prática de fazer negócios com base apenas nos preços. Ao invés, minimize os custos. Estabeleça um relacionamento duradouro com os seus fornecedores;
5. Melhore constantemente o sistema de produção e de serviços para melhorar a relação qualidade x produtividade e, assim, diminuir custos e aumentar lucros;
6. Institucionalize novos métodos de treinamento no trabalho;
7. Institucionalize e fortaleça os papéis de liderança;
8. Elimine os medos, de modo que cada funcionário possa trabalhar efetivamente pela companhia;
9. Quebre barreiras entre departamentos. Todos devem trabalhar como uma equipe coesa;
10. Elimine slogans, exortações e metas para a força de trabalho produzir com defeito-zero e novos níveis de produtividade, pois isto pode levar a um desgaste no relacionamento entre as pessoas causando baixa qualidade e produtividade;
11. Elimine cotas-padrão arbitrárias e gerenciamento por objetivos;
12. Produza e gerencie com foco na qualidade e não apenas em atingir números;
13. Institucionalize um vigoroso programa de educação e auto-melhoria;
14. Coloque todos na organização para trabalhar pela transformação.

### 6.2.2 Abordagem de Armand Feigenbaum

Armand V. Feigenbaum era estudante de doutorado no *Massachusetts Institute of Technology* nos anos 1950, quando introduziu o conceito de controle da qualidade total e concluiu a primeira edição do seu livro *Total Quality Control (TQC*). Segundo Feingenbaum, o Controle da Qualidade Total representa um sistema efetivo que integra a qualidade do desenvolvimento, qualidade de manutenção e esforços de melhoria da qualidade de vários grupos em uma organização.

Pode-se consolidar o enfoque da abordagem de Feigenbaum nos seguintes aspectos:

- *Liderança para a qualidade* → significa que a qualidade deve ser gerenciada e submetida à manutenção constante, a fim de se obter a excelência;
- *Tecnologia moderna para qualidade* → nessa visão, todos na organização são responsáveis pela qualidade de seus processos, produtos e serviços, o que impõe a integração de colaboradores em diferentes níveis organizacionais;
- *Compromisso organizacional* → a qualidade deve ser vista como um fator estratégico pela organização, que deve ser responsável por treinar sua equipe nas atividades específicas de cada um.

### 6.2.3 Abordagem de Joseph M. Juran

Joseph M. Juran foi um educador chave em gerência da qualidade para os japoneses, que difundiram sua teoria. Ele focou sua abordagem nas atividades gerenciais e na responsabilidade para a qualidade, se preocupando no impacto de trabalhadores individuais e no envolvimento e motivação da força de trabalho nas atividades de melhoria. Ele fundamentou sua abordagem sobre gerência de qualidade em três processos básicos:

- *Planejamento da Qualidade* → o planejamento deve ser iniciado com a determinação das necessidades do cliente, seguido da definição de requisitos básicos para que o produto ou serviço atenda às expectativas do cliente. Processos para desenvolvimento desses requisitos devem ser definidos;
- *Controle da Qualidade* → uma vez estabelecidos direcionamentos e processos para desenvolvimento dos produtos e serviços, o controle da qualidade deve garantir o grau de qualidade e produtividade previamente estabelecido;
- *Melhoria da Qualidade* → as atividades de melhoria da qualidade abrangem a identificação de oportunidades de melhoria, assim como o estabelecimento de responsabilidades para execução dessas melhorias e divulgação de resultados.

### 6.2.4 Abordagem de Philip Crosby

Philip B.Crosby iniciou seus estudos na década de 1960 e defendeu que a qualidade pode ser vista como o grau de conformidade com a especificação, fundamentada em uma abordagem de busca contínua do defeito zero [PMBOK2000]. Sua abordagem inclui o que ele denominou de 4 “certezas” do Gerenciamento da Qualidade:

- Qualidade significa atendimento aos requisitos;
- Qualidade vem através de prevenção;
- Padrão para desempenho da qualidade e “defeitos zero”;
- A medida de qualidade é o preço da não-conformidade.

Em seu livro “*Quality is Free”* [Crosby1979], Crosby define um modelo de maturidade organizacional com 5 estágios baseados no gerenciamento da qualidade. Essa abordagem inspirou o *Capability Maturity Model for Software* (CMM) [Paulk1995], modelo de maturidade criado pelo *Software Engineering Institute*. Os 5 estágios são apresentados a seguir.

1. **Desconhecimento:** “Nós não sabemos que temos problemas com qualidade.” Nesse nível, não há compreensão sobre qualidade como uma ferramenta de gestão. Problemas são tratados à medida que surgem. Inspeções não são realizadas;

2. **Despertar:** “É absolutamente necessário ter problemas com qualidade?” Reconhecimento de que gerenciamento da qualidade pode agregar valor, mas não gerando um crescimento do lucro da organização;

3. **Alinhamento:** “Através de compromisso gerencial e melhoria da qualidade nós identificamos e resolvemos nossos problemas.” Gerenciamento da qualidade se torna uma ferramenta de suporte e ajuda aos processos. Comunicações de ações corretivas são estabelecidas e problemas são priorizados e solucionados seguindo uma ordem;

4. **Sabedoria:** “Prevenção de defeito é uma parte da rotina de nossa operação.” Entendimento absoluto do gerenciamento da qualidade. Reportagem efetiva de status e ações preventivas. Problemas são identificados antes do seu desenvolvimento. Todas as funções estão abertas para sugestões e melhoria nos processos;

5. **Certeza:** “Nós sabemos por que não temos problemas com qualidade.” A função qualidade é considerada uma parte essencial do sistema organizacional. Exceto em casos não comuns, problemas são prevenidos.

Segundo o modelo de maturidade de Crosby, uma organização no nível 1 não tem estimativas sobre o custo da qualidade, que representa 20% em termos do serviço vendido. À medida que o nível de maturidade se eleva, o custo da qualidade tende a diminuir. Em outras palavras, uma organização nível 5 apresenta um custo da qualidade estimado praticamente igual ao real, que gira em torno de 2,5% [Crosby1979].

### 6.2.5 Total Quality Control (TQC)

*Total Quality Control,* ou TQC, é um sistema desenvolvido no Japão, montado pelo Grupo de Pesquisa do Controle da Qualidade da *Union of Japanese Scientists and Engineers* (JUSE), para implementar a melhoria contínua da qualidade. Há mais de 40 anos, o TQC vem sendo aperfeiçoado por estudiosos na área de qualidade. Conforme praticado no Japão, o TQC se baseia na participação de todos os setores da empresa e de todos os empregados no estudo e condução do controle da qualidade.

O TQC é baseado em elementos provenientes de várias fontes (vide Figura 6.1): emprega o método cartesiano, utiliza o controle estatístico de processos, adota conceitos sobre o comportamento humano. Adicionalmente, aproveita o todo o conhecimento ocidental sobre qualidade, principalmente o trabalho de Juran.

O TQC é regido pelos seguintes princípios básicos [Feingenbaum1994]:

- Produzir e fornecer produtos e/ou serviços que atendam concretamente às necessidades do cliente;
- Garantir a sobrevivência da empresa através do lucro contínuo, adquirido pelo domínio da qualidade;
- Identificar o problema mais crítico e solucioná-lo pela mais alta prioridade;
- Falar, raciocinar e decidir com dados e com base em fatos;
- Gerenciar a organização ao longo do processo e não por resultados;
- Reduzir metodicamente as distorções através do isolamento de suas causas fundamentais;
- Não permitir a venda de produtos defeituosos;
- Não repetir erros;
- Definir e garantir a execução da visão e estratégia da alta direção da empresa.

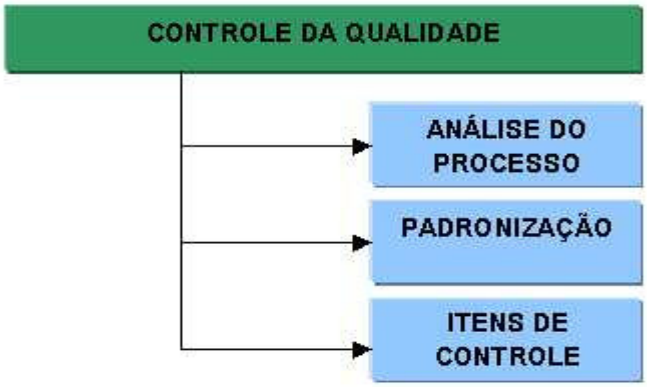


**Figura 6.1: Bases do TQC**

Uma das principais formas de implementação do controle de processos e adotada pelo TQC é a utilização do Ciclo PDCA *(Plan-Do-Check-Action),* que consiste em 4 fases: planejar, executar, verificar e atuar corretamente (vide Figura 6.2).

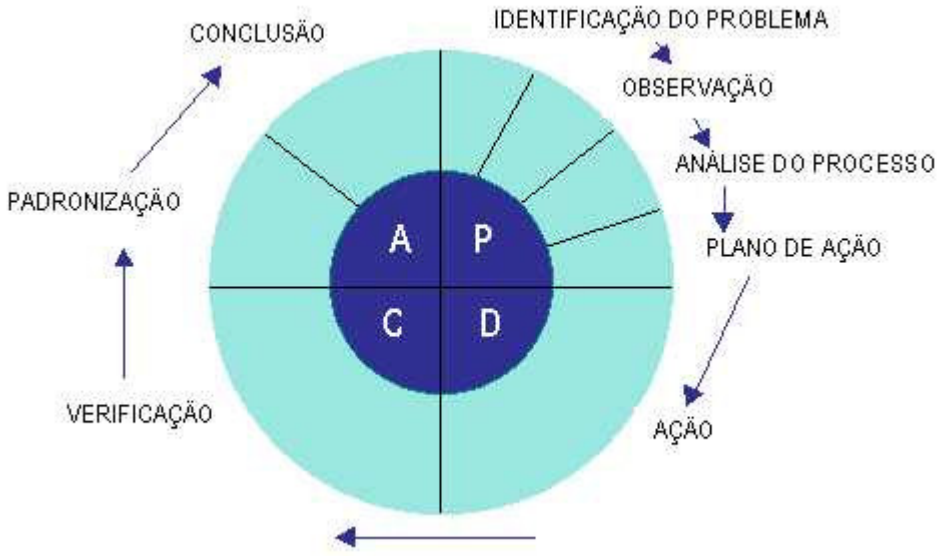


**Figura 6.2: Ciclo PDCA para melhorias**

Na fase *Plan*, o foco está na identificação do problema, análise do processo atual e definição do plano de ação para melhoria do processo em questão. Na fase *Do*, o plano de ação definido deve ser executado e controlado. Em *Check*, verificações devem ser realizadas, a fim de subsidiar ajustes e se tirar lições de aprendizagem. Finalmente, em *Action,* deve-se atuar corretivamente para fundamentar um novo ciclo, garantindo a melhoria contínua.

O PDCA é considerado um ciclo clássico, adotado, até hoje, por organizações no mundo inteiro. Ele deve ser utilizado para todos os processos da organização, em todos os níveis hierárquicos. Ele pode ser utilizado como base para qualquer organização, na definição de uma metodologia de controle ou melhoria de qualquer tipo de processo.

### 6.2.6 Total Quality Management (TQM)

*Total Quality Management (TQM)* é uma abordagem para sucesso em longo prazo, que é medida através da satisfação do cliente e baseada na participação de todos os membros da organização. TQM foca a melhoria de processos, produtos, serviços e cultura organizacional e considera qualidade de um processo como responsabilidade do "dono" do processo.

Nesse contexto, Kan [Kan1995] descreve os seguintes elementos-chave do TQM, conforme demonstrado graficamente na Figura 6.3:

- Foco no cliente → consiste em analisar os desejos e necessidades, bem como definir os requisitos do cliente, além de medir e gerenciar a satisfação do cliente;
- Melhoria de processo → o objetivo é reduzir a variação do processo e atingir a melhoria contínua do processo. Processos incluem tanto processos de negócio quanto processo de desenvolvimento de produtos;
- Aspecto humano → nesse contexto, as áreas-chave incluem liderança, gerência, compromisso, participação total e outros fatores sociais, psicológicos e humanos;
- Medição e análise → o objetivo é gerenciar a melhoria contínua em todos os parâmetros de qualidade por um sistema de medição orientado a metas.

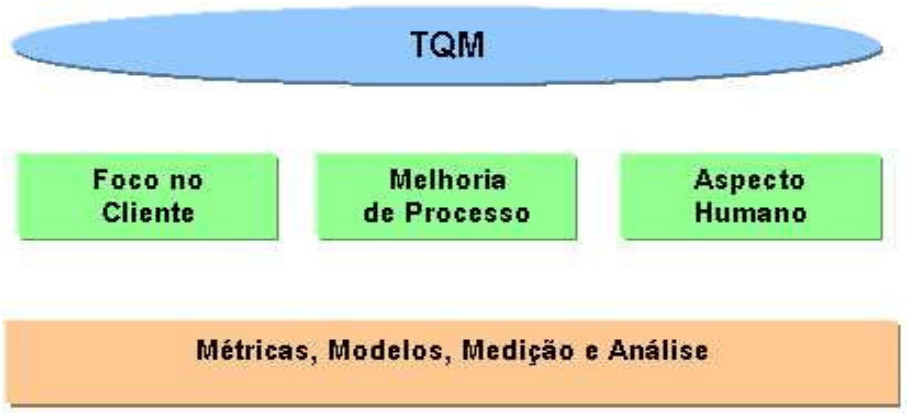


**Figura 6.3: Elementos-chave do TQM**

## 6.3 INTRODUÇÃO À QUALIDADE DE SOFTWARE

No sentido de se tornarem mais competitivas, as organizações de software vêm investindo cada vez mais na qualidade de seus produtos e serviços de software. A qualidade de software está diretamente relacionada a um gerenciamento rigoroso de requisitos, uma gerência efetiva de projetos e em um processo de desenvolvimento bem definido, gerenciado e em melhoria contínua. Atividades de verificação e uso de métricas para controle de projetos e processo também estão inseridas nesse contexto, contribuindo para tomadas de decisão e para antecipação de problemas.

Inserido nesse contexto, esta seção provê uma visão geral da área de qualidade de software, abrangendo atividades e processos fundamentais para garantia da qualidade e produtividade em organizações de software.

### 6.3.1 Prevenção X Detecção

Toda organização possui processos formais e/ou informais que são implementados para produzir seus produtos de desenvolvimento de software (vide Figura 6.4). Esses produtos podem incluir tanto produtos finais, que são usados pelos clientes (como código executável, manuais de usuário), como artefatos intermediários (como documento de requisitos, diagramas de projeto, casos de teste etc.).

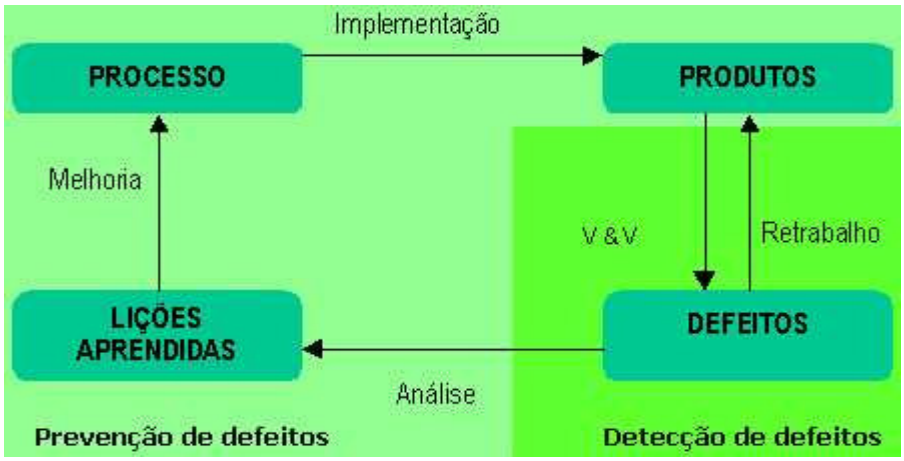


**Figura 6.4: Desenvolvimento de Produtos de Software**

Várias técnicas são utilizadas para identificar defeitos nos produtos de trabalho. Esses defeitos são eliminados através de re-trabalho, que têm efeito imediato na produtividade do projeto. Defeitos também são encontrados em atividades de teste e podem ser analisados, a fim de se identificar suas causas. A partir dessa análise, lições aprendidas podem ser usadas para criar futuros produtos e prevenir futuros defeitos e, dessa forma, ter impacto positivo na qualidade do produto e na produtividade do projeto.

Algumas técnicas de prevenção e de detecção são listadas a seguir:

***Técnicas de Prevenção***

- Treinamento;
- Planejamento;
- Modelagem;
- Atuação do grupo de garantia da qualidade (SQA);
- Uso de lições aprendidas;
- Melhoria de processo.

***Técnicas de Detecção***

- Compilação/Análise de código;
- *Peer Reviews* (revisão por pares)*;*
- Teste e simulação;
- Auditorias;
- Verificações e validações.

### 6.3.2 Planejamento e Gerência da Qualidade de Software

O planejamento e o gerenciamento da qualidade têm representado um papel cada dia mais forte no contexto do desenvolvimento de software. Desde o início de um projeto, a qualidade deve ser vista como um fator crítico para o sucesso do software e deve ser considerada no planejamento e gerenciamento do mesmo. Essa seção apresenta uma visão geral dos processos de Planejamento da Qualidade de Software e Gerenciamento da Qualidade de Software.

***Planejamento da Qualidade de Software***

O processo de planejamento da qualidade de um projeto (vide Figura 6.5) compreende a identificação de quais padrões são relevantes para o projeto e a determinação de como os satisfazer [PMBOK2000]. Esse processo deve ser executado em paralelo ao planejamento do projeto como um todo, uma vez que as atividades de qualidade planejadas influenciam na estimativa de esforço e no custo do projeto como todo.


**Figura 6.5: Visão geral do planejamento da qualidade**

No contexto do desenvolvimento de software, o planejamento da qualidade deve incluir a documentação das atividades de prevenção e detecção de defeitos, tais como auditorias, inspeções, teste, coleta de métricas, além do uso de padrões e descrição do processo de software adotado.

As atividades de planejamento da qualidade são subsidiadas por informações fundamentais, tais como:

- *Política da Qualidade* → compreende intenções e diretrizes de uma organização, no contexto da qualidade, formalmente expressadas pela alta gerência;
- *Estabelecimento do Escopo* → o conhecimento do escopo do projeto é a entrada-chave para o planejamento da qualidade, desde os produtos a serem entregues, assim como os objetivos do projeto;
- *Procedimentos, Padrões e Regulamentos* → padrões e regulamentos relacionados com o contexto do projeto que são aplicáveis devem ser considerados no planejamento do projeto. Isso inclui processos, modelos e técnicas definidas no âmbito organizacional, além de padrões regulamentares fora do escopo da organização;
- *Descrição do Produto* → o entendimento das características do software a ser gerado é fundamental para que se tenha uma visão clara dos processos adequados ao projeto. Com base nessas características, métodos, modelos e processos devem ser selecionados e adaptados para o projeto específico.

O plano da qualidade é o produto principal do processo de planejamento da qualidade de software. Compreende informações sobre como a equipe de qualidade irá garantir o cumprimento da política de qualidade, no contexto do programa ou projeto a ser desenvolvido, podendo ser detalhado ou geral, dependendo das características do projeto. O plano da qualidade pode ser um documento específico ou parte do plano do projeto e deve ser utilizado como base do gerenciamento ao longo de todo o ciclo de vida do projeto.

O IEEE [IEEE1999a] define um modelo básico para plano da qualidade de um projeto de software, que inclui entre outras, as seguintes informações:

- Propósito;
- Documentos de referência;
- Organograma;
- Atividades;
- Responsabilidades;
- Requisitos de documentação;
- Padrões e convenções;
- Métricas;
- Revisões e auditorias;
- Teste;
- Reporte de problemas e ações corretivas;
- Ferramentas, métodos e metodologias;
- Controle de fornecimento;
- Coleta, manutenção e retenção de registros;
- Treinamento;
- Gerenciamento de riscos.

***Garantia da Qualidade de Software***

Garantia da qualidade (vide Figura 6.6) é um conjunto de atividades planejadas e sistemáticas, implementadas com base no sistema da qualidade da organização, a fim de prover a confiança de que o projeto irá satisfazer padrões relevantes de qualidade [PMBOK2000]. As atividades de garantia da qualidade de software são focadas na prevenção de defeitos e problemas, que podem surgir nos produtos de trabalho. Definição de padrões, metodologias, técnicas e ferramentas de apoio ao desenvolvimento fazem parte desse contexto.


**Figura 6.6: Visão geral da garantia da qualidade**

As atividades de garantia da qualidade são apoiadas pelas seguintes informações que representam as entradas do processo:

- *Plano da Qualidade de Software* → o plano da qualidade é a ferramenta básica que direciona todas as atividades de garantia da qualidade e deve ser utilizado efetivamente ao longo do desenvolvimento do projeto;
- *Resultados de medições de qualidade* → métricas coletadas e consolidadas, a partir das atividades de controle da qualidade, devem ser utilizadas para análise e subsídio para melhoria do processo no âmbito do projeto, assim como no âmbito organizacional.

Como principal resultado do processo tem-se a Melhoria da Qualidade, que inclui ações de melhoria no projeto, assim como nos processos, métodos e padrões adotados pelo projeto. Essas ações são identificadas em atividades como auditorias, revisões e coleta de métricas.

***Controle da Qualidade de Software***

Compreende atividades de monitoração de resultados específicos do projeto, a fim de determinar sua aderência a padrões de qualidade e identificar causas de resultados insatisfatórios. Essas atividades são executadas através do uso de técnicas que incluem revisões por pares, diagramas de Pareto, teste e análise de tendências, entre outras.

O controle da qualidade deve ser executado ao longo do desenvolvimento do software, em geral, por membros da equipe de desenvolvimento, assim como por pessoas que tenham independência com relação à equipe de desenvolvimento de software.

### 6.3.3 Custos da Qualidade

Os custos da qualidade abrangem o custo total dos esforços para alcançar a qualidade do produto/serviço, podendo ser categorizados em custos de prevenção, custos de avaliação, custos de falhas internas e custos de falhas externas. A fim de reduzir custos de falhas internas e externas, nós, tipicamente, devemos despender mais esforço em prevenção e detecção. Conforme ilustrado na Figura 6.7, existe um ponto ótimo de equilíbrio entre os diferentes custos de qualidade, que aponta um custo ideal total da qualidade. Vale salientar que esse ponto ótimo, no entanto, pode ser difícil de ser mensurado, considerando custos subjetivos de falhas externas, como a insatisfação do cliente, por exemplo.

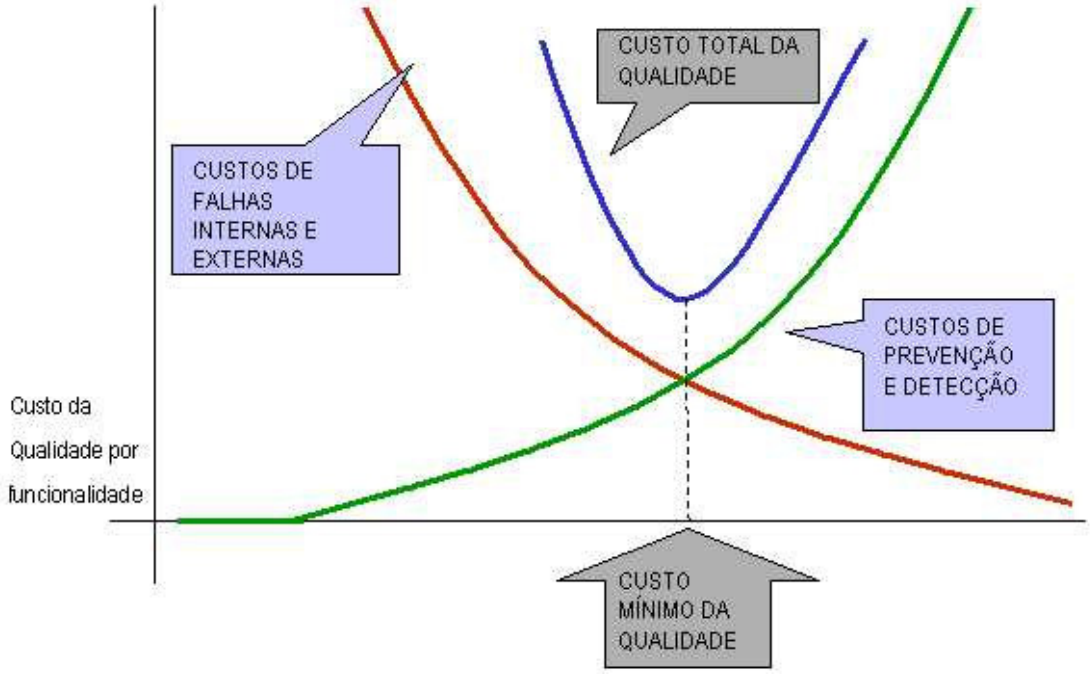


**Figura 6.7: Balanceamento do custo da qualidade [Kezner1998]**

Os custos de prevenção incluem esforço para prevenir defeitos que venham a ser inseridos ao longo do desenvolvimento do software. Os custos de avaliação compreendem o custo do esforço gasto em atividades de detecção de defeitos nos produtos, processos ou serviços. Os custos de falhas internas envolvem manutenção e correção de falhas encontradas antes da liberação do produto ou serviço ao cliente. Inclui esforços gastos nas seguintes atividades:

- Depuração;
- Correção;
- *Re-peer reviews;*
- Teste da correção e teste de regressão.

Finalmente, os custos de falhas externas envolvem manutenção e correção de falhas encontradas após da liberação do produto ou serviço ao cliente. Inclui esforços gastos nas seguintes atividades:

- Depuração;
- Correção;
- Re-peer reviews;
- Teste da correção e teste de regressão;
- Versões corretivas;
- Serviço de atendimento ao cliente;
- Insatisfação do cliente e vendas perdidas.

Alguns métodos de coleta podem ser adotados para subsidiar a análise de custos da qualidade, conforme relacionado a seguir:

- Time sheets;
- Logs e relatórios de teste;
- Help desk;
- Sistemas de reporte de defeitos;
- Custo do projeto;
- Informações de vendas;
- Índice de satisfação do cliente.

## 6.4 MODELOS E PADRÕES DE QUALIDADE DE SOFTWARE

Com o crescimento do setor de software, vários modelos e padrões têm sido propostos ao longo dos últimos anos. Nesse contexto, alguns conceitos fundamentais são importantes de serem entendidos. O primeiro deles é a "política", que representa um princípio governamental, tipicamente usado como base para regras, procedimentos ou padrões e geralmente estabelecido pela mais alta autoridade da organização [Humphrey1989]. Um "padrão" pode ser entendido como uma base para comparação e é usado para suportar tamanho, conteúdo, valor ou qualidade de um objeto ou atividade. Um "guia", por sua vez, é uma prática, método ou procedimento sugerido. Finalmente, um "modelo" pode ser definido como uma representação abstrata de um item ou processo, a partir de um ponto de vista específico. O objetivo do modelo é expressar a essência de algum aspecto de um item ou processo, sem prover detalhes desnecessários.

Diversos modelos de qualidade de software vêm sendo propostos ao longo dos últimos anos. Esses modelos têm sido fortemente adotados por organizações em todo o mundo. A Tabela 6.1 apresenta uma visão geral da evolução dos modelos de qualidade de software, através de algumas iniciativas relevantes, desde 1980. Em seguida, os padrões ISO9000 e os modelos propostos pelo *Software Engineering Institute* são abordados ainda nessa seção.

**Tabela 6.1: Iniciativas para melhoria da qualidade do processo de software**

| ANO | INICIATIVA |
|---|---|
| 1983 | - NQI/CAE: 1 Prêmio Canadense de Excelência |
| 1984 | - Avaliação conduzida pela IBM |
| 1987 | - ISO 9001 versão inicial<br>- NIST/MBNQA: 1º Prêmio de Qualidade Nacional Malcolm Baldrige (USA)<br>- SEI-87-TR-24 (questionário SW-CMM) |
| 1988 | - AS 3563 (Sist. de Gerenciamento de Qualidade de Software) – versão inicial |

Continua...

...continuação

| | |
|---|---|
| 1991 | - IEEE 1074 versão inicial<br>- ISO 9000-3 versão inicial<br>- SEI SW-CMM V 1.0 versão inicial<br>- Trillium V 1.0 versão inicial |
| 1992 | - EFQM/BEA: 1º Prêmio de Excelência do Negócio (Europa)<br>- IEEE adota AS 3563 como “IEEE 1298”<br>- TickIT V2.0 |
| 1993 | - SEI SW-CMM V1.1<br>- BOOTSTRAP<br>- SPICE |
| 1994 | - ISO 9001<br>- Trillium V3.0 |
| 1995 | - ISO 12207 versão inicial<br>- ISO 15504 (SPICE) versão inicial |
| 1996 | - IEEE/EIA 12207 |
| 1997 | - ISO 9000-3<br>- SW-CMM com suporte ao CMM Integration (CMMI) |
| 1998 | - ISO 15504 (SPICE) para o público como relatório técnico<br>- TickIT V4.0 |
| 1999 | - SEI CMMI para projetos piloto |
| 2000 | - Nova versão da ISO9001<br>- CMMI |
| 2001 | - Adendo a ISO12207<br>- Nova versão da ISO9000-3 |
| 2003 | - ISO 15504 |

### 6.4.1 As Normas ISO

A *International Organization for Standardization* (ISO) é uma organização não-governamental, fundada em 23/02/1947, com sede em Genebra – Suíça. A razão de existência da ISO foi motivada pela necessidade de referências internacionais para regulamentar obrigações contratuais entre fornecedores e compradores, que focalizassem a garantia de manutenção e uniformidade da qualidade de produtos.

As normas da ISO há muito tempo são relacionadas à qualidade. Atualmente, a norma ISO9001:2000 é modelo base para auditorias de certificação da família ISO9000. A norma é um padrão internacional que “especifica requisitos para um sistema gerencial de qualidade de uma organização”. Com o crescimento substancial das indústrias de software e, levando-se em conta que a produção de software

apresenta características peculiares, a ISO tem trabalhado na definição de várias normas que podem ser utilizadas como guias e padrões para diversas áreas de atuação dentro do contexto da ciência da computação. Dentre esses, vale ressaltar a importância da norma ISO9000-3, que estabelece um guia para facilitar a aplicação da ISO9001 para desenvolvimento, suporte e manutenção de software. Seguindo o mesmo conteúdo da ISO9001, a ISO9000-3 descreve requisitos para sistemas de qualidade, focados em 4 aspectos:

- *Responsabilidade gerencial* – envolve o compromisso da alta administração;
- *Gerência de recursos* - abrange tanto a parte de infra-estrutura quanto o capital humano da organização;
- *Realização do produto* - compreende requisitos para planejamento, projeto e implementação de produtos;
- *Medição e análise* - contempla requisitos para coleta e análise do sistema da qualidade.

A Figura 6.8 apresenta uma visualização gráfica dos requisitos da Norma ISO9000-3:2000.

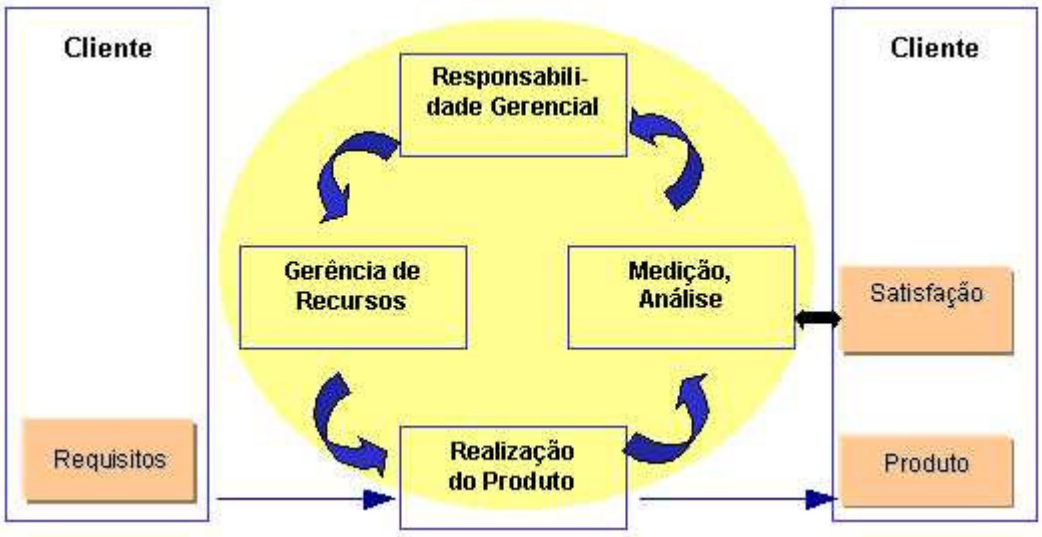


**Figura 6.8: Requisitos da ISO9001/ISO9000-3**

Uma certificação ISO9000, no Brasil, é conduzida por uma empresa acreditada pelo INMETRO. Isso significa que a empresa foi avaliada e julgada por um organismo certificador, pertencente ao Sistema Brasileiro de Certificação, segundo aquela norma.

A Tabela 6.2 apresenta algumas normas ISO aplicadas à qualidade de software, focadas em produto ou processo de software.

**Tabela 6.2: Normas ISO9000 para suporte ao desenvolvimento de software**

| Nome | Descrição |
|---|---|
| Norma ISO/IEC 9126 (NBR 13596) | Define as características de qualidade de software que devem estar presentes em todos os produtos (Funcionalidade, Confiabilidade, Eficiência, Usabilidade, Manutenibilidade e Portabilidade); |
| Norma ISO/IEC 12119 | Estabelece os requisitos de qualidade para pacotes de software; |
| Norma ISO/IEC 14598-5 | Define um processo de avaliação da qualidade de produto de software; |
| Norma ISO/IEC 12207 | Define um processo de ciclo de vida de software; |
| Norma ISO/IEC 9000-3 | Apresenta diretrizes para a aplicação da ISO 9001 por organizações que desenvolvem software ao desenvolvimento, fornecimento e manutenção de software; |
| Norma ISO15504 | Aprovada como norma em 2003 é focada na avaliação de processos organizacionais. |

Dentre as normas citadas, vale a pena destacar a norma ISO12207 e a recém aprovada ISO15504. A ISO12207 descreve um modelo de ciclo de vida de software, que pode servir de base referencial para organizações que desejam formalizar um processo de desenvolvimento de software. A ISO15504 atualmente é uma norma que representa um padrão internacional emergente, que estabelece um *framework* para construção de processos de avaliação e melhoria.

***ISO12207***

O principal objetivo da norma ISO12207 é o estabelecimento de uma estrutura comum para os processos de ciclo de vida de software, para ser utilizada como referência. Além disso, a norma considera que o desenvolvimento e a manutenção de software devem ser conduzidos da mesma forma que a disciplina de engenharia.

A estrutura descrita na ISO12207 (vide Figura 6.9) utiliza-se de uma terminologia bem definida e é composta de processos, atividades e tarefas a serem aplicados em operações que envolvam, de alguma forma, o software, seja através de aquisição, fornecimento, desenvolvimento, operação ou manutenção. Essa estrutura permite estabelecer ligações claras com o ambiente de engenharia de sistemas, ou seja, aquele que inclui práticas de software, hardware, pessoal e negócios.


**Figura 6.9: Processos da ISO12207**

***ISO15504***

Em 1993, um grupo de trabalho conjunto da ISO/IEC-*International Organization for Standardization/International Electrotechnical Commission* estabeleceu um projeto denominado *SPICE-Software Process Improvement and Capability Evaluation* [Dorling1993], que objetivava chegar a um consenso entre os diversos métodos de avaliação do processo de software. Este projeto posteriormente gerou o relatório técnico ISO15504 [ISO15504:1-9:1998].

A ISO15504 atualmente é uma norma que representa um padrão internacional emergente que estabelece um *framework* para construção de processos de avaliação e melhoria do processo de software. Este *framework* pode ser utilizado pelas ODSs envolvidas em planejar, gerenciar, monitorar, controlar e melhorar a aquisição, fornecimento, desenvolvimento, operação, evolução e suporte do software. A ISO15504 não é um método isolado para avaliação e sua característica genérica permite que possa ser utilizada em conjunto com uma variedade de métodos, de técnicas e de ferramentas.

Nove documentos compõem a ISO15504. Alguns têm caráter normativo (como a ISO15504-2, ISO15504-3 e ISO 15504-9) e outros possuem caráter informativo (como a ISO15504-1, ISO15504-4, ISO15504-5, ISO15504-6, ISO15504-7 e ISO15504-8). A seguir faremos um resumo do conteúdo destes documentos.

A ISO15504-1 fornece as informações gerais sobre o conteúdo de todos os documentos da ISO15504 e como eles estão relacionados. Determina também o campo de aplicação da ISO15504, seus componentes e os relacionamentos da ISO15504 com outros padrões internacionais (tais como a série ISO9000 e a ISO12207).

A ISO15504-2 define um modelo de referência de processo e um modelo de capacitação do processo (Figura 6.10) que pode compor a base para a definição e avaliação de um processo de software. Este modelo de referência possui uma abordagem bidimensional.

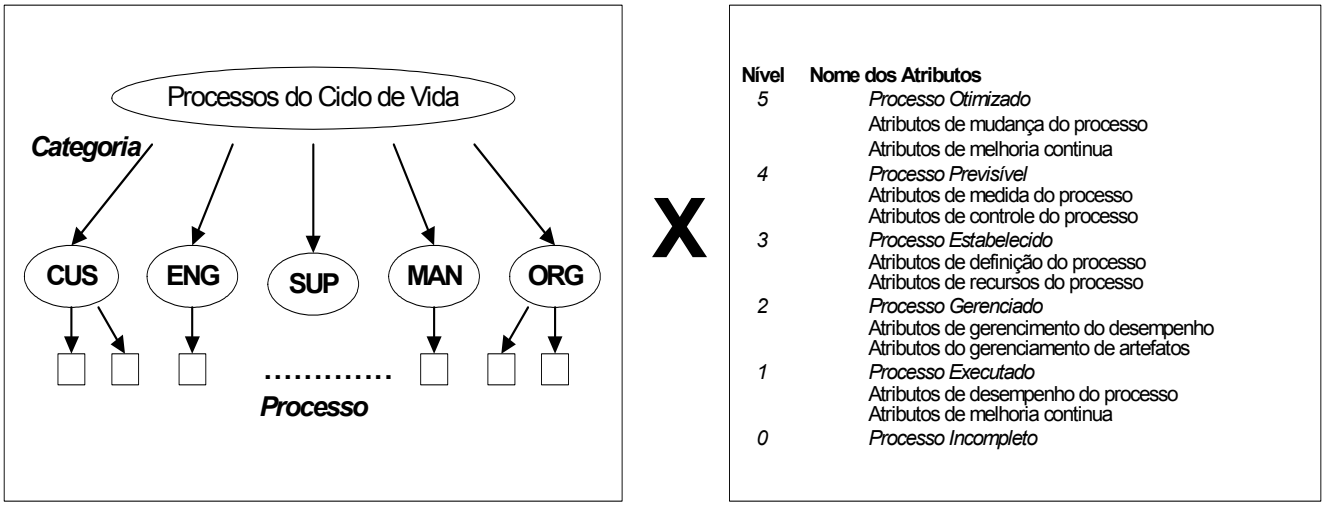


**Figura 6.10: As dimensões do modelo de referência da ISO 15504**

A primeira dimensão auxilia os engenheiros de processo na definição dos processos necessários em uma ODS, assim como sua adequação à ISO15504. A segunda dimensão, similar ao CMM, tem por objetivo determinar a capacidade do processo da ODS, avaliando-o através de um conjunto de atributos pré-estabelecidos. Os atributos de processo estão agrupados nos níveis de capacidade e podem ser aplicados em todos os processos descritos na ODS para determinar a maturidade do processo.

Cinco grupos de processos são definidos pela ISO 15504-2:

1. Fornecedor-Cliente (*CUS-Custormer-Supplier*): são os processos que de alguma forma impactam diretamente o cliente: dentre eles o suporte para desenvolvimento e transações de software para o cliente, fornecimento de operações corretas e uso do produto de software ou serviço;

2. Engenharia (*ENG-Engineering*): esta categoria agrupa os processos que especificam, implementam e mantêm o produto de software, sua relação com o sistema e a documentação do cliente;
3. Suporte (SUP-Support): São processos que podem ser empregados em algum dos outros processos e em vários pontos no ciclo de vida do software objetivando dar suporte a eles;
4. Gerenciamento (MAN-Management): são processos que contêm práticas gerenciais que podem ser utilizadas por alguém que gerencia algum tipo de projeto ou processo dentro do ciclo de vida do software;
5. Organização (ORG-Organization): são processos que estabelecem as finalidades dos processos de desenvolvimento e da organização, do produto, e dos recursos, que, quando utilizados por projetos na organização, realizarão as metas do negócio.

O modelo de referência definido na ISO15504-2 fornece uma base comum para executar avaliações da capacidade do processo de software. Porém, o modelo não pode ser utilizado sozinho, tendo em vista que o detalhamento é insuficiente.

As ISO15504-3 e ISO15504-4 servem como guias para a realização de uma avaliação do processo. Determinam procedimentos para: a definição das entradas da avaliação, a determinação das responsabilidades, a especificação do processo propriamente dito da avaliação e os resultados que devem ser guardados.

O modelo de avaliação definido pela ISO15504-5 está baseado e é compatível com o modelo de referência descrito pela ISO15504-2, podendo ser usado como base para conduzir uma avaliação da capacidade do processo de software. A Figura 6.11 apresenta a estrutura básica do modelo de avaliação da ISO15504-5 e seu relacionamento com a ISO15504-2. Este modelo de avaliação expande o modelo de referência adicionando a definição e uso de indicadores de avaliação. Os indicadores de avaliação são definidos para que os assessores possam julgar o desempenho e capacidade de um processo implementado.

A ISO15504-5 detalha os processos (primeira dimensão do modelo de referência da ISO15504-2) associando a eles algumas práticas básicas. Artefatos e atributos são sugeridos e associados às práticas básicas com o intuito de estabelecer sua realização.

A ISO15504-6 objetiva a preparação de assessores para a execução de avaliações de processos de software. Esta norma também descreve mecanismos que podem ser utilizados para determinar a competência dos assessores e avaliar seu conhecimento, seu treinamento e a sua experiência como assessor.

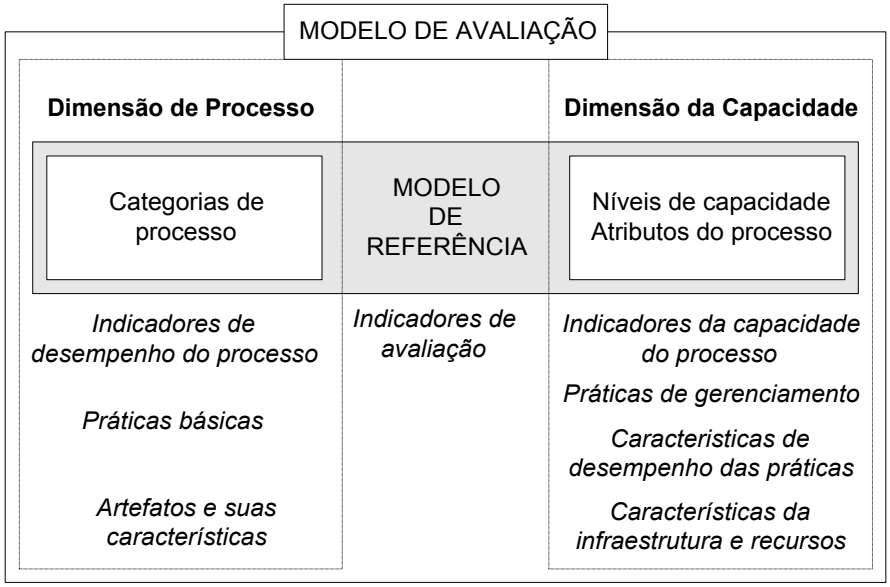


**Figura 6.11: Relacionamento entre o modelo de referência e o modelo de avaliação.**

A ISO15504-7 auxilia a criação de *frameworks* e métodos para executar a melhoria do processo de software de um modo contínuo. Esta parte da ISO15504 estabelece como:

- Invocar uma avaliação do processo de software;
- Usar os resultados de uma avaliação;
- Medir o processo de software e melhorar sua efetividade;
- Identificar ações de melhorias alinhadas com as metas do negócio;
- Usar um modelo de processo compatível com o modelo de referencia definido na ISO15504-2;e
- Garantir uma cultura organizacional no contexto de melhoria do processo de software.

A ISO15504-8 é um guia para a determinação da capacidade do processo do fornecedor. Esta determinação da capacidade do processo é uma avaliação e análise sistemática dos processos de software selecionados dentro de uma organização, cujo objetivo é identificar os pontos fortes, as fraquezas e os riscos associados no desdobramento do processo, encontrando um requisito especificado. Este requisito pode ser um projeto, um produto ou um serviço, uma nova ou existente tarefa, um

contrato ou uma atividade interna, ou algum outro requisito dentro dos processos de software da ODS.

A ISO15504-9 define os termos utilizados em todos os documentos da ISO15504.

A Figura 6.12 ilustra o relacionamento destes principais elementos da 15504.

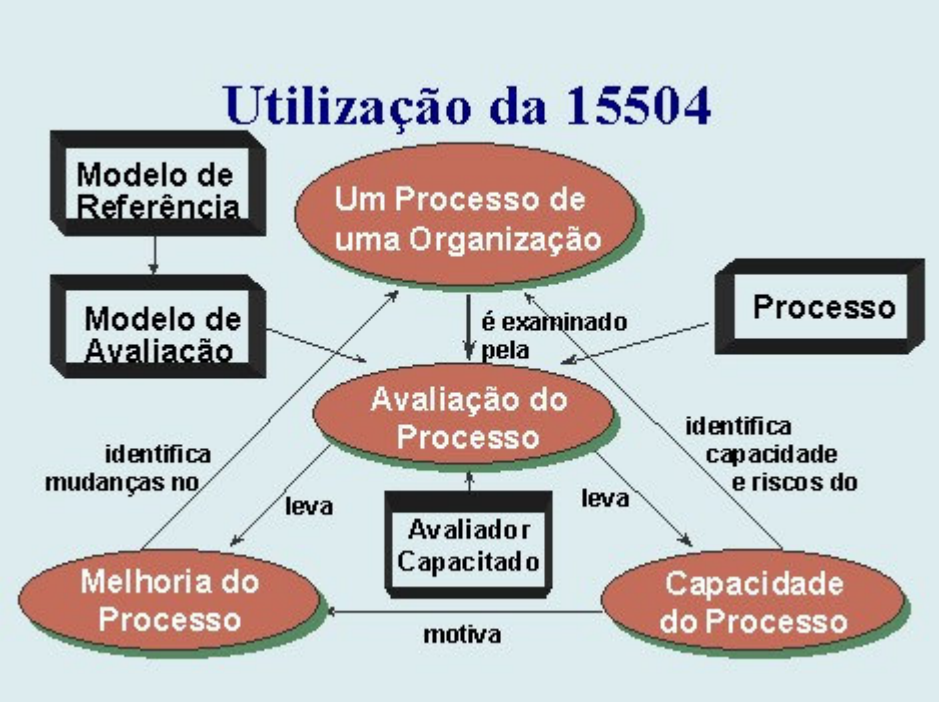


**Figura 6.12: Utilização da ISO15504**

### 6.4.2 Os Modelos do Software Engineering Institute (SEI)

O *Software Engineering Institute* (SEI), dos Estados Unidos, tem contribuído bastante com o fortalecimento da área de qualidade de software, através da definição de modelos internacionais focados no processo de software, assim como na publicação de relatórios técnicos e artigos relacionados ao assunto.

Dentre os modelos propostos pelo SEI, o *Capability Maturity Model for Software* (CMM) destaca-se porque tem sido largamente adotado pela comunidade de software internacional. O CMM é um modelo focado na capacidade organizacional e seu desenvolvimento foi motivado por uma solicitação do Departamento de Defesa dos Estados Unidos, que precisava de um instrumento de avaliação dos fornecedores contratados pelo próprio Departamento.

O CMM categoriza as organizações em 5 níveis de maturidade, desde um processo *ad hoc* e desorganizado (nível 1), até um estágio altamente gerenciado de melhoria contínua (nível 5). Esses níveis de maturidade são definidos em áreas-chave de processo (KPA's), que por sua vez, possuem metas que devem ser atingidas através da implementação de práticas-chave, categorizadas no que o modelo chama de características comuns, conforme observado na Figura 6.13.

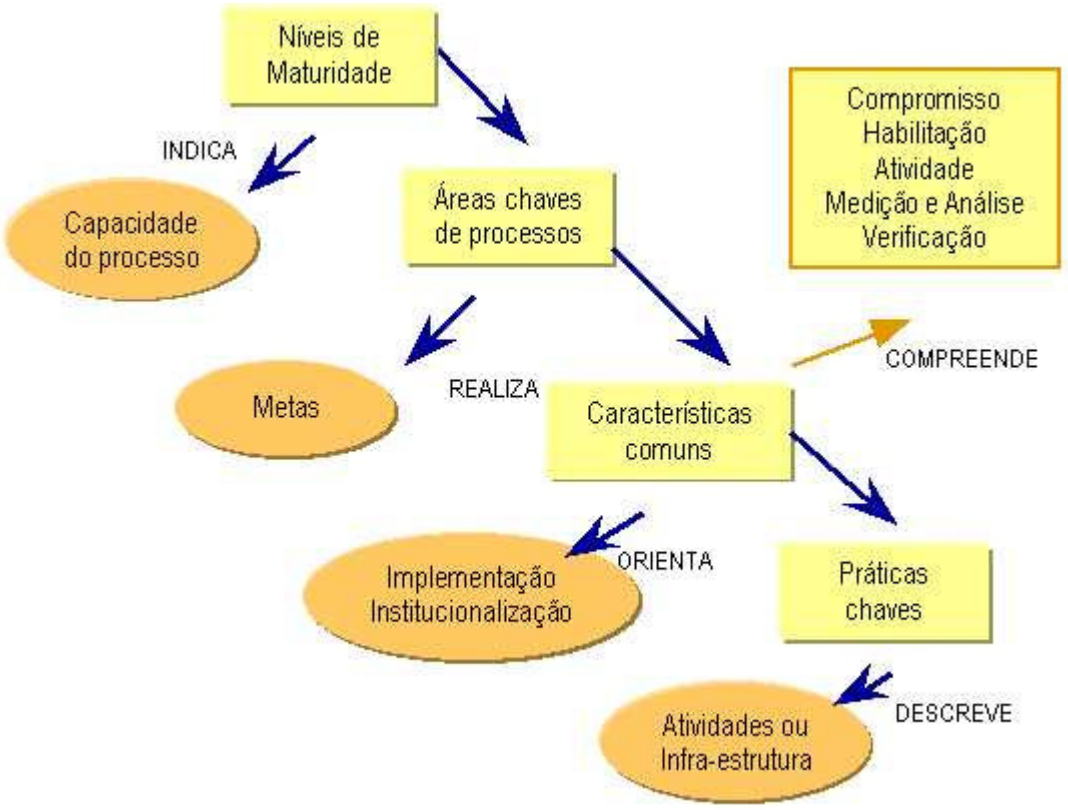


**Figura 6.13: Estrutura do Capability Maturity Model for Software**

O nível de maturidade 2 se concentra em disciplinas de gerência de projetos; o nível 3 é focado na definição, melhoria e uso adequado do processo de software da organização; o nível 4 define práticas referentes à gerência de qualidade. Finalmente, no nível 5, a organização deve ter processos para garantir efetividade na prevenção de defeitos e em mudanças tecnológicas e no processo organizacional.

O CMM foi descontinuado pelo SEI e substituído pelo CMMI, detalhado a seguir. Vale salientar que o CMM foi um dos modelos de qualidade mais adotados pelas indústrias de software brasileiro, juntamente com a norma ISO9001.

### *CMMI*

A sigla CMMI representa as iniciais de *Capability Maturity Model Integration* e nomeia tanto um projeto, quanto os modelos resultantes deste projeto. O Projeto CMMI, que pode ser traduzido como "Projeto de Integração dos Modelos de Maturidade da Capacidade", está sendo executado pelo *Software Engineering Institute* (SEI), em cooperação com a indústria e governo, para consolidar um *framework* para modelos, evoluir e integrar modelos desenvolvidos pelo SEI (inicialmente os modelos SW-CMM, SE-CMM e IPD-CMM) e gerar seus produtos associados, incluindo material de treinamento e método de avaliação. Estes três modelos, que foram evoluídos e integrados inicialmente, foram a versão 2.0 do SW-CMM (*Capability Maturity Model for Software*), o SE-CMM: EIA 731 (*System Engineering Capability Maturity Model*) e o IPD-CMM (*Integrated Product Development Capability Maturity Model*).

Essa integração e evolução tiveram como objetivo principal a redução do custo da implementação de melhoria de processo multidisciplinar baseada em modelos. Multidisciplinar porque além da engenharia de software, o CMMI cobre também a engenharia de sistemas, aquisição e a cadeia de desenvolvimento de produto. Esta redução de custo é obtida por meio da eliminação de inconsistências, redução de duplicações, melhoria da clareza e entendimento, utilização de terminologia comum, utilização de estilo consistente, estabelecimento de regras de construção uniforme, manutenção de componentes comuns, garantia da consistência com a norma ISO15504 e sensibilidade às implicações dos esforços legados.

O nome do modelo CMMI pode ser traduzido para "Modelo Integrado de Maturidade da Capacidade". Em agosto de 2000 foi lançada a versão 1.0 do CMMI, também chamada de CMMI-SE/SW v1 [SEI2000]. Após outras versões intermediárias, em 10 de março de 2002 foi lançada a versão atual denominada de CMMI-SE/SW/IPPD/SS Version 1.1 (CMMI$^{SM}$ *for Systems Engineering / Software Engineering / Integrated Product and Process Development / Supplier Sourcing, Version 1.1*, em português CMMI$^{SM}$ para Engenharia de Sistemas / Engenharia de Software / Desenvolvimento Integrado de Produtos e Processos / Fornecimento) [SEI2002a].

A arquitetura do CMMI é composta basicamente pela definição de um conjunto de áreas de processo, organizadas em duas representações diferentes: uma como um modelo por estágio, semelhante ao SW-CMM, e outra como um modelo contínuo (semelhante à ISO/IEC 15504). A versão atual é composta por 25 áreas de processo, conforme ilustrado na Figura 6.14.

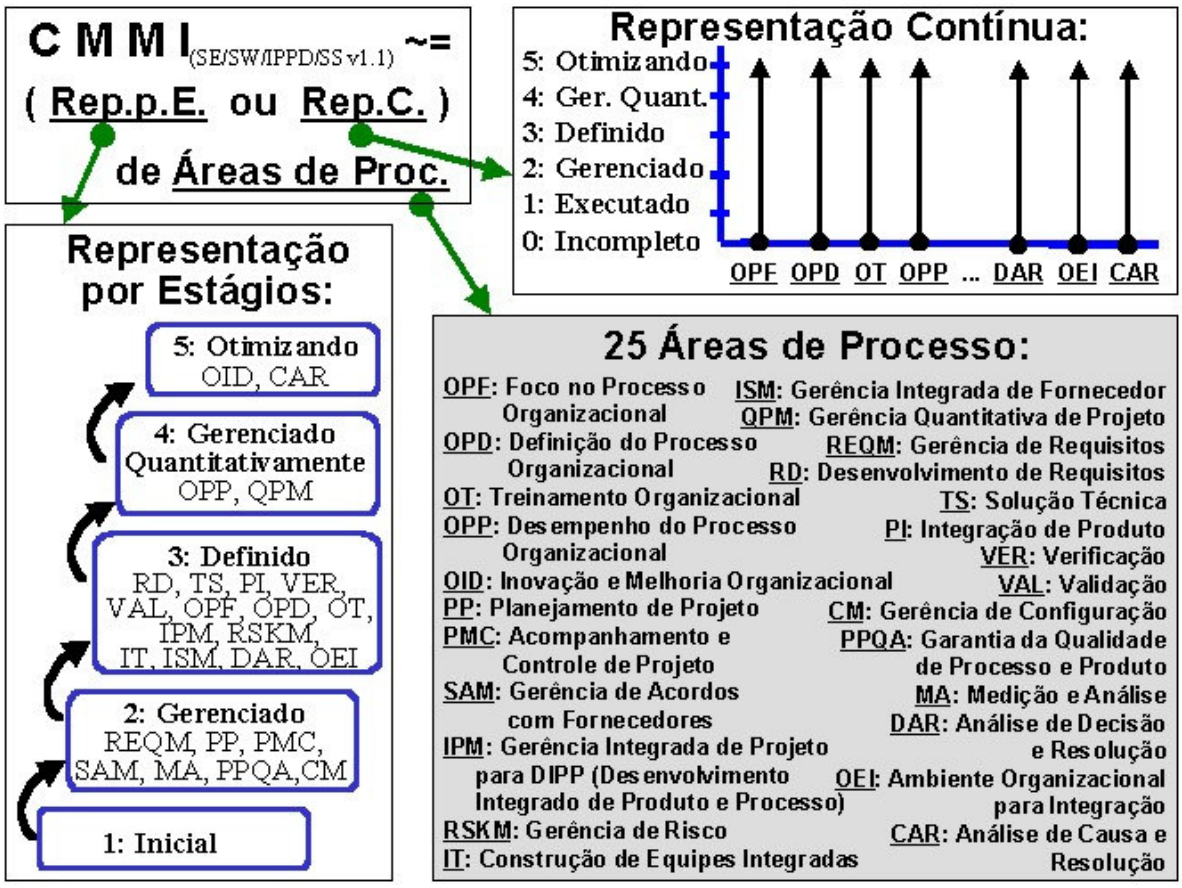


**Figura 6.14: CMMI - Áreas de processo em duas representações: por estágio e contínua**

Cada área de processo é definida no modelo por meio da descrição de seu propósito, notas introdutórias, relacionamentos com outras áreas, metas específicas, metas genéricas, práticas e sub-práticas específicas, práticas e sub-práticas genéricas, produtos de trabalho típicos e referências para outros elementos do modelo relacionados. A descrição de cada área ocupa cerca de 20 páginas.

Na representação por estágio, as 25 áreas de processo estão agrupadas em 4 níveis de maturidade: níveis 2, 3, 4 e 5 (o nível 1 não contém nenhuma área de processo) [SEI2002b]. Em relação a esta representação, o processo de desenvolvimento e manutenção de software ou sistema de uma unidade

organizacional pode estar classificado em um dos seguintes cinco níveis de maturidade:

- Nível 1: Inicial (*Initial*);
- Nível 2: Gerenciado (*Managed*);
- Nível 3: Definido (*Defined*);
- Nível 4: Gerenciado Quantitativamente (*Quantitatively Managed*);
- Nível 5: Otimizando (*Optimizing*).

Cada nível de maturidade é definido basicamente pelo conjunto de áreas de processo do nível.

Na representação contínua, as mesmas 25 áreas de processo estão agrupadas em quatro grupos (gerência de processos, gerência de projetos, engenharia e suporte) e são definidos seis níveis de capacidade de processo [SEI2002c]. Nesta representação, o conjunto de atividades correspondente a cada uma destas áreas de processo, pode ter sua capacidade de execução classificada em um dos seguintes seis níveis de capacidade de processo:

- Nível 0: Incompleto (*Incomplete*);
- Nível 1: Executado (*Performed*);
- Nível 2: Gerenciado (*Managed*);
- Nível 3: Definido (*Defined*);
- Nível 4: Gerenciado Quantitativamente (*Quantitatively Managed*);
- Nível 5: Otimizando (*Optimizing*).

Cada nível de capacidade é definido por um conjunto de características que o processo deve satisfazer para estar naquele nível.

Em relação ao SW-CMM, o CMMI por estágio é uma revisão, com ajustes. No nível 2 de maturidade, por exemplo, foi incluída a área de processo de medição e análise como uma ampliação deste assunto, que já era coberto em parte em cada área do SW-CMM. No nível 3, por exemplo, a área de engenharia de produto do SW-CMM foi mais bem descrita por meio de cinco áreas de processo: Desenvolvimento de Requisitos, Solução Técnica, Integração de Produto, Verificação e Validação. Outras mudanças ocorrem para refletir melhor a orientação para atendimento dos níveis de maturidade.

Em relação à utilização para melhoria de processo, o CMMI por estágio sugere abordagens semelhantes àquelas utilizadas com sucesso com o SW-CMM. Em relação ao CMMI contínuo, a tendência é toda a experiência de utilização da ISO15504 ser aproveitada para ajustar as abordagens já utilizadas com sucesso pelo SW-CMM.

Além do CMM e CMMI, o SEI também definiu outros modelos que podem contribuir com o processo de melhoria de organizações de software. Dentre esses modelos, podem ser citados: o *People-CMM*, voltado para a gerência de pessoas em uma organização; o *Personal Software Process* (PSP), focado no processo pessoal do desenvolvedor de software; e o IDEAL, modelo que propõe um processo para melhoria de processos baseado no ciclo PDCA.

## 6.5 MELHORIA DO PROCESSO DE SOFTWARE

A qualidade do produto de software está diretamente relacionada à qualidade do processo que o produz. Dessa forma, a definição de um processo de software apropriado ao tipo da aplicação a ser desenvolvida, alinhada à cultura dos profissionais envolvidos no projeto de software, é um fator crítico de sucesso para qualquer organização de software.

Nesse contexto, a ISO9001:2000 e o CMM/CMMI recomendam que, além da definição de um processo de software adequado às características do projeto, seja garantida pela organização uma sistemática para melhoria contínua desse processo.

Em [Humphrey1989], Watts Humphrey relaciona passos para melhoria do processo de software:

1. Compreender a situação corrente;
2. Desenvolver uma visão do processo;
3. Estabelecer uma lista de ações de melhoria de processo requeridas em ordem de prioridade;
4. Produzir um plano para execução dessas ações;
5. Iniciar o passo 1 novamente.

### 6.5.1 O Modelo IDEAL

Um dos modelos mais conhecidos, focado na melhoria de processo, é o modelo IDEAL [MacFeeley1999], do *Software Engineering Institute* (SEI). O nome IDEAL é um acrônimo para cada um dos 5 estágios definidos pelo modelo:

- **I**niciação, que abrange o estímulo para melhoria e estabelece a infra-estrutura para melhoria;
- **D**iagnóstico, que compreende a avaliação e caracterização da prática corrente, além do desenvolvimento de recomendações;
- **E**stabelecimento, quando estratégias e prioridades são definidas (plano de ação);

- **A**ção, que foca a definição de processos e métricas, execução de projetos pilotos e acompanhamento;
- Nivelamento (**L***everaging*), que abrange a atualização da abordagem organizacional e análise de lições aprendidas.

A Figura 6.15 apresenta uma visualização gráfica do modelo IDEAL.

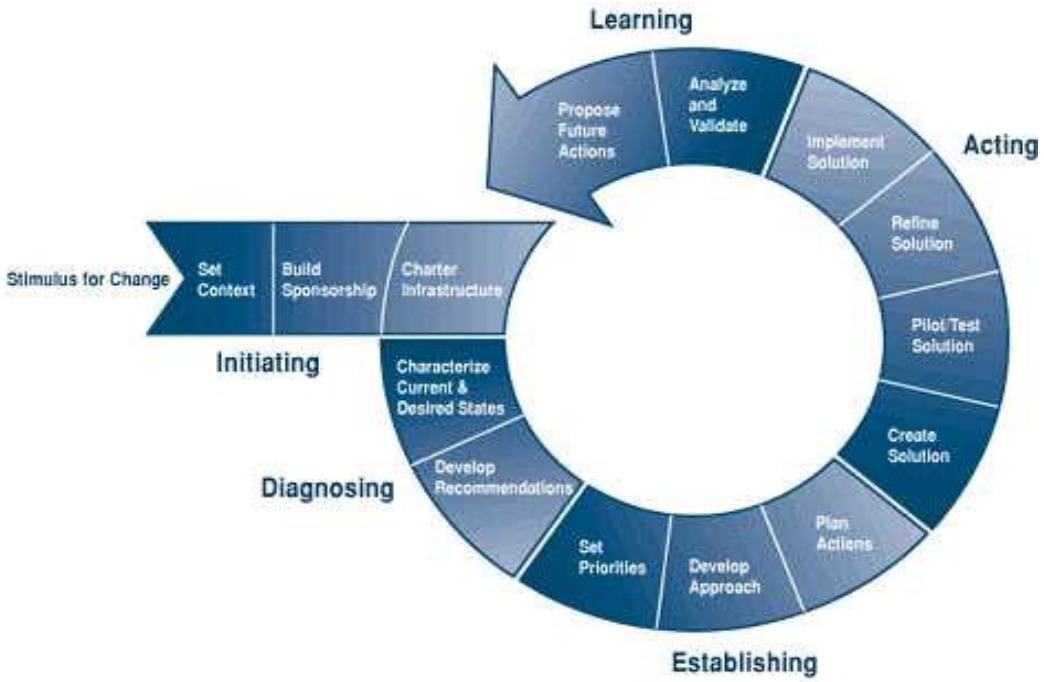


**Figura 6.15: Visualização gráfica do IDEAL [MacFeeley1999]**

O IDEAL é adotado como base de vários métodos de avaliação definidos pelo SEI. Vale salientar seu alinhamento à abordagem do ciclo PDCA, apresentado na seção 6.2.5.

## 6.6 AUDITORIAS E AVALIAÇÕES (*ASSESSMENTS*)

Um dos avanços mais relevantes do controle da qualidade é o crescimento da função de “auditoria da qualidade”. A implementação e execução das auditorias representam uma das áreas mais significativas da engenharia de controle do processo. Por outro lado, as avaliações, também conhecidas como *assessments,* são também largamente adotadas como fundamento para planos de melhoria.

Esta seção tem como objetivo assegurar um entendimento sobre o conceito de auditoria e avaliação interna (*assessment*), seus objetivos e benefícios.

### 6.6.1 Auditorias

Segundo Feingenbaum, “auditoria da qualidade é a avaliação da verificação da eficácia do controle”. Em outras palavras, a auditoria da qualidade pode ser considerada, em alguns casos, a inspeção da inspeção dos itens, ensaio dos ensaios de produto, procedimento para avaliação da eficácia dos procedimentos.

A ISO9000:2000 define uma auditoria como “um processo sistemático, independente e documentado, para se obter evidência e avaliá-la objetivamente, visando determinar a extensão na qual os critérios de auditorias são atendidos”.

É importante entender que uma auditoria é um exercício de coleta de informações, que possibilitará a identificação de necessidade de melhoria ou de ações corretivas. Para garantir uma auditoria efetiva, a veracidade das informações coletadas é essencial, uma vez que decisões importantes e críticas são tomadas e podem impactar positiva ou negativamente para o cenário financeiro de uma organização.

As auditorias são normalmente realizadas com um ou mais dos seguintes objetivos:

- Determinação de conformidade ou não-conformidade do sistema da qualidade com requisitos especificados;
- Identificação da eficácia do sistema da qualidade e do potencial de melhoria;
- Atendimento aos requisitos regulamentares;
- Para fins de certificação.

***Tipos de Auditoria***

Com relação aos elementos sendo auditados, as auditorias podem ser classificadas nos seguintes tipos:

- *Auditorias de Produto* - uma técnica fundamental da engenharia do controle de processo é a da implementação das auditorias de produto;
- *Auditorias de Processo* – o principal papel dessas auditorias é a garantia da execução efetiva de todos os aspectos do procedimento. O planejamento dessas auditorias pode ser direcionado a procedimentos individuais ou a grupos de procedimentos, por exemplo, aplicáveis em áreas como documentação e registros de processo, medições, conformidade com requisitos do processo e conformidade de produtos com padrões de qualidade estabelecidos.

Com relação à avaliação da eficácia da implementação de um sistema da qualidade e a determinação do grau com o qual os objetivos do sistema estão sendo atingidos, as auditorias podem ser classificadas como:

- *Primeira parte* → realizada por uma organização sobre si mesma;
- *Segunda parte* → conduzida por uma organização sobre uma outra, para fins da organização condutora da auditoria;
- *Terceira parte* → realizadas por uma terceira parte, independente, sem interesse nos resultados da auditoria. Nessa classe, incluem-se as auditorias de certificação, como as auditorias ISO9001, as quais podem ser:

  *Inicial:* completa, abrangendo todo o escopo de certificação;

  *De Manutenção:* periódica, conduzida para determinar a manutenção da auditoria inicial;

  *De Re-certificação:* realizada no final do período de certificação no sentido de re-emitir o certificado para um novo período.

Como resultado das auditorias, um relatório é fornecido a gerentes e supervisores da área auditada (vide Figura 6.16). Isso permite que todas as atividades de auditorias fiquem as mais visíveis possíveis, possibilitando a identificação, execução e o acompanhamento de ações corretivas (identificando os respectivos responsáveis). A implementação dessas ações corretivas indicadas constituirá uma área-chave para a atenção em auditorias subseqüentes. A idéia é utilizar os resultados da auditoria para induzir à melhoria do sistema.

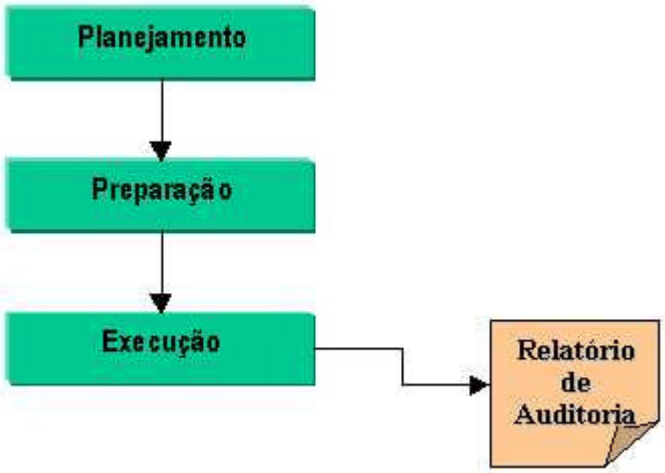


**Figura 6.16: Visão geral do processo de auditoria**

### 6.6.2 Avaliações Internas (*Assessments*)

As avaliações internas, também conhecidas como *assessments*, auxiliam a organização a melhorar, através da identificação de problemas críticos e estabelecimento de ações de melhoria, com foco em revisão e não em auditoria. Elas têm como objetivos a aquisição de conhecimento sobre como a organização trabalha, a fim de identificar principais problemas e recomendar ações (vide Figura 6.17).

A condução e sucesso de uma avaliação interna estão relacionados à garantia de algumas premissas, conforme relacionadas a seguir:

- *Base em um modelo de processo* → toda avaliação interna segue um modelo que serve como base para a avaliação do processo em foco. O CMM é um bom exemplo de um modelo que serve de base para essas avaliações;
- *Confidencialidade* → naturalmente, muitas informações são analisadas criticamente e, dessa forma, expostas à equipe condutora. Assim, a confidencialidade é premissa básica para que a organização submetida à avaliação esteja totalmente à vontade para expor os dados que forem necessários para uma conclusão segura;
- *Envolvimento da alta gerência* → o compromisso da alta direção da organização que está sendo avaliada é fundamental para respaldar os resultados e garantir a execução de ações corretivas;
- *Respeito a diferentes pontos de vista* → cada organização tem cultura própria. É essencial que se leve em consideração a cultura organizacional e a liberdade de escolha com relação à forma de execução de algumas atividades.

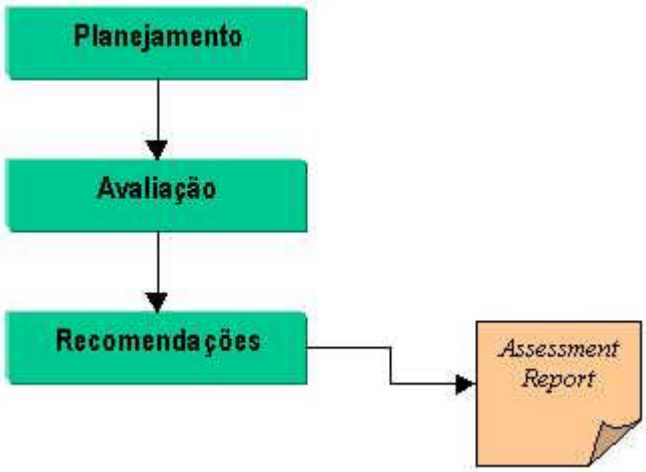


**Figura 6.17: Visão geral do processo de avaliação**

Assim como as auditorias, as avaliações precisam ser bem planejadas e geram como produto principal um documento. Esse documento compreende os resultados das avaliações realizadas durante o período de avaliação, apontando pontos fracos e fortes, identificados durante os trabalhos.

Uma característica interessante das avaliações internas é que elas são conduzidas por um grupo de pessoas, diferente das auditorias, que em geral são conduzidas por 1 ou 2 auditores.

## 6.7 MEDIÇÃO DE SOFTWARE

O uso de métricas tem representado uma ferramenta essencial à gerência de projetos de software. Muitos engenheiros de software medem as características do mesmo, a fim de obter uma visão clara de uma série de aspectos, tais como, o atendimento dos produtos de software aos requisitos especificados, desempenho do processo de desenvolvimento, esforço/custo despendido, entre outros.

As medições podem ser úteis sob diferentes óticas. Por exemplo, o gerente de projeto pode utilizar medições para se certificar se estimativas realizadas no início do projeto foram efetivas, acompanhando esforço, prazo, custo e tamanho, comparando dados planejados com dados reais. Por outro lado o cliente pode usar medições para determinar se o mesmo atende aos requisitos e se possui um grau de qualidade satisfatório.

### 6.7.1 Seleção de Métricas

Atualmente, os programas de métricas de software têm sido definidos para prover informações específicas necessárias para gerenciar projetos de software e melhorar processos e serviços de engenharia de software. Objetivos no nível organizacional, de projeto ou de tarefa, são determinados antecipadamente e, então, métricas são selecionadas baseadas nesses objetivos. Essas métricas são, então, utilizadas para determinar a eficácia do alcance desses objetivos.

Vale salientar que as métricas coletadas não solucionam problemas. O que as métricas podem fazer é prover informação sobre as quais se pode garantir tomadas de decisões embasadas em dados mais concretos.

Uma métrica útil sempre tem um cliente. O cliente dessa métrica é a pessoa (ou pessoas) que irá tomar decisões com base na informação gerada pela métrica. Se uma métrica não tem um cliente, ou seja, se não existe uma pessoa que vai utilizá-la para tomada de decisão, então a métrica não tem sentido para a organização. Em outras palavras, é preciso responder à seguinte pergunta: *Quem precisa da informação? Quem irá usar esta métrica?*

#### *Técnica Goal/Question/Metric (GQM)*

A técnica, ou paradigma, GQM foi definida por Basili e Rombach e provê um excelente mecanismo para definição de um programa de métricas baseado em objetivos. Segundo a técnica, primeiramente deve-se selecionar um ou mais objetivos mensuráveis. Esses objetivos podem ser tanto metas estratégicas, como minimização de custos ou maximização de satisfação do cliente, quanto objetivos mais específicos, como avaliação de um novo método de análise e projeto, por exemplo.

Após a definição dos objetivos, devem-se determinar questões que precisam ser respondidas, a fim de determinar se cada objetivo é alcançado. Finalmente, as métricas são selecionadas no sentido de proverem informação necessária para a resposta a cada questão.

**=> Definindo Objetivos**

Os objetivos definidos podem variar, dependendo do nível de abstração que se deseja considerar no programa de métricas a ser definido. O nível de abstração pode ser:

- *Organizacional* → onde são tipicamente considerados objetivos de negócio relacionados a custo, índice de satisfação do cliente ou mesmo o atendimento a uma margem de lucro almejada;
- *De Projeto* → no nível de projeto, praticamente são considerados objetivos para enfatizar o gerenciamento do projeto e controlar requisitos e metas do projeto, considerando-se fatores de sucesso, tais como custo, prazo, qualidade, atendimento aos requisitos, entre outros;
- *De Atividade* → nesse nível, consideram-se métricas que podem prover informação sobre a execução efetiva de uma determinada atividade.

**=> Identificando Questões**

A pergunta-chave é: *"quais questões precisam ser respondidas para determinar se os objetivos foram alcançados?"*. Com base nesse foco, cada objetivo deve ter suas questões relacionadas. A seguir, é apresentado um exemplo de questões relacionadas a um objetivo de negócio.

Objetivo: manter um alto nível de satisfação do cliente.

- Qual o nível de satisfação atual do meu cliente?
- Como nós estamos em comparação com nossos concorrentes?
- Quais atributos de produtos e serviços são mais importantes para os nossos clientes?

**=> Selecionando Métricas**

Finalmente, métricas devem ser selecionadas sempre com o foco único de resposta às questões definidas. Em outras palavras, são as métricas que provêem informação necessária para responder cada questão definida.

Para que um programa de métricas seja efetivo e para que a métrica selecionada seja útil para a organização é importante que ela seja:

- Facilmente calculada, entendida e testada;
- Passível de estudos estatísticos;
- Expressa em alguma unidade;
- Obtida o mais cedo possível no ciclo de vida do software;
- Passível de automação;
- Repetível e independente do observador.

Além disso, uma métrica deve ser:

- *Válida:* quantifica o que queremos medir;
- *Confiável:* produz os mesmos resultados dadas as mesmas condições;
- *Prática:* barata, fácil de computar e fácil de interpretar.

*Exemplo:*

Levando-se em consideração o objetivo: "garantir que todos os defeitos conhecidos sejam solucionados antes da liberação do produto", questões a serem definidas podem ser:

- Quantos defeitos nós temos por produto?
- Qual o status de cada defeito?

Nesse caso, métricas podem ser definidas como Número de defeitos e Número de defeitos por status.

### 6.7.2 Uso de Métricas para Suporte a Estimativas

Um dos aspectos mais focados em medições é o suporte a estimativas de projetos. Nesse contexto, a medição de tamanho é bastante recomendada pelos modelos de qualidade. Uma dessas medidas mais comum é a contagem de linhas de código (KLOC). Embora essa métrica possa parecer simples, existe discordância sobre o que constitui uma linha de código. Existem impasses, por exemplo, sobre a inclusão ou não de linhas de comentário e linhas em branco, uma vez que estas servem para a documentação interna do programa e não afeta a sua funcionalidade. Além disso, a linguagem de programação adotada pelo projeto é um fator que impacta diretamente na métrica. Medidas baseadas em programas desenvolvidos em um tipo de linguagem de programação não devem ser comparadas a código escrito em outra linguagem. Essa restrição impede, por exemplo, a utilização de dados históricos para projetos que não utilizam a mesma linguagem.

Em outro contexto, métricas orientadas à função concentram-se na complexidade da funcionalidade do software. Foram propostas no início da década de 1970, por pesquisadores da IBM, cujo trabalho era identificar as variáveis críticas que determinam a produtividade da programação.

Em 1979, Allan Albrecht, prosseguindo essas pesquisas, introduziu uma técnica de avaliação conhecida como “pontos por função”. Essa métrica tem sido bastante adotada e está baseada na visão externa do usuário, sendo independente da linguagem utilizada.

Ela permite calcular o esforço de programação e auxilia o usuário final a melhorar o exame e avaliação de projetos.

Os pontos por função de um sistema de informação são definidos em três etapas de avaliação:

- A primeira etapa resulta na contagem de pontos por função não ajustados, que refletem as funções específicas e mensuráveis do negócio, provida ao usuário pela aplicação;
- A segunda etapa da avaliação gera o fator de ajuste, que representa a funcionalidade geral provida ao usuário pela aplicação;
- A terceira e última etapa resulta na contagem de pontos por função ajustados, que por sua vez reflete o fator de ajuste aplicado ao resultado apurado na primeira etapa.

O cálculo do fator de ajuste é baseado em 14 características gerais dos sistemas, que permitem uma avaliação geral da funcionalidade da aplicação. Essas características são relacionadas a seguir:

- Comunicação de dados;
- Processamento distribuído;
- Performance;
- Utilização de equipamento;
- Volume de transações;
- Entrada de dados on-line;
- Eficiência do usuário final;
- Atualização on-line;
- Processamento complexo;
- Reutilização de código;
- Facilidade de implantação;
- Facilidade operacional;
- Múltiplos locais;
- Facilidade de mudanças.

A métrica de pontos por função (FP), assim como as linhas de código (LOC), é controversa.

Esse método é independente da linguagem de programação. Baseia-se em dados que são conhecidos logo no começo da evolução de um projeto, tornando-se mais atraente como abordagem de estimativa.

No entanto, a contagem dos pontos se baseia parcialmente em dados subjetivos, implicando à organização estabelecer um plano com critérios e premissas para subsidiar a medição, antes do início efetivo da utilização. Esse planejamento é fundamental para que os resultados das medições possam ser comparados entre os projetos.

## 6.8 VERIFICAÇÕES E VALIDAÇÕES

Cada vez mais, as verificações e validações de software têm sido consideradas ferramentas úteis no contexto da garantia da qualidade de software. Através delas, são obtidas visões mais concretas com relação a aspectos de qualidade de alguns produtos de software. Nesse contexto, as revisões formais - que incluem inspeções - e atividades de teste são fortemente recomendadas por modelos atuais de qualidade de software, tais como as normas ISO e o CMM.

Segundo o IEEE, verificação e validação de software (V&V) abrangem técnicas de revisão, análise e teste para determinar o quanto um sistema de software e seus produtos intermediários estão de acordo com os requisitos. Esses requisitos incluem tanto capacidade funcional quanto atributos de qualidade.

O esforço de V&V é tipicamente aplicado como parte ou em paralelo ao desenvolvimento do software e atividades de suporte. Ele provê informações sobre engenharia, qualidade e status dos produtos de software ao longo do ciclo de vida, promovendo previamente identificação de defeitos.

Essa seção apresenta informações sobre verificação e validação, especialmente apresentando a técnica de revisões formais. Vale salientar que atividades de teste estão no contexto das verificações e já foram abordadas no capítulo 5.

### 6.8.1 Revisões Formais

As revisões formais são verificações normalmente adotadas para a garantia de produtos de software. Segundo o IEEE [IEEE1999b], os principais tipos de revisões formais de software são:

***Revisões Técnicas* (technical reviews)**

Essas revisões têm o objetivo de avaliar artefatos específicos para verificar se eles estão de acordo com os respectivos padrões e especificações e se eventuais modificações nos artefatos foram efetuadas de maneira correta. Em geral, as revisões

técnicas são aplicadas a documentos - como o Documento de Requisitos, artefatos técnicos de análise e projeto, Projeto de Teste - com objetivo principal de verificar a aderência dos artefatos revisados aos padrões adotados pelo projeto, assim como aspectos que interferem na qualidade, como completude, precisão, consistência, facilidade de manutenção e verificação, entre outros aspectos.

***Inspeções (inspection)***

Representam um tipo de revisão por pares (*peer reviews*). Têm o objetivo principal de identificação e remoção de defeitos. É obrigatória a geração de uma lista de defeitos, com classificação padronizada, requerendo-se a ação dos autores para remoção desses defeitos. Em geral, são aplicadas aos artefatos de desenho, implementação e testes, focalizando a correção destes em relação aos respectivos padrões e especificações, enquanto as revisões técnicas têm maior enfoque na qualidade da documentação. Inspeções serão mais bem detalhadas na Seção 6.8.2.

***Wakthroughs***

São revisões nas quais o autor apresenta, em ordem lógica, sem limite de tempo, o material a um grupo, que verifica o material à medida que ele vai sendo apresentado. Esse tipo de revisão não exige muita preparação prévia e pode ser feito com maior número de participantes. As revisões de apresentação são consideradas como de eficácia média para a detecção de defeitos. Elas podem ser usadas nos marcos de projeto, em que são necessárias apresentações ao cliente.

***Revisões Gerenciais***

São conduzidas pelo gerente de um projeto, com os objetivos principais de avaliar os problemas técnicos e gerenciais do projeto, assim como o seu progresso em relação aos planos. Em geral, pelo menos uma revisão gerencial deve ser realizada ao final de cada iteração. Conforme a política adotada de controle de projetos, elas podem ser também realizadas por período (por exemplo: semana, quinzena ou mês). Finalmente, são aplicáveis a alguns documentos que não requerem, normalmente, uma revisão técnica, por serem de natureza gerencial, ou como revisão preliminar de documentos que serão submetidos a revisões técnicas ou revisões de apresentação ao cliente.

Além dos tipos supracitados, o padrão IEEE-1028 [IEEE1999b] cobre também as auditorias, que têm o objetivo de verificar a conformidade de produtos e projetos com padrões e processos.

Revisões informais também podem ser realizadas, especialmente antes das revisões formais, a fim de que problemas mais relevantes possam ser resolvidos

antecipadamente. Exemplos de revisões informais incluem a programação em pares, adotada pelas metodologias ágeis, como a *Extreme Programming (XP*), e as revisões individuais, que são realizadas pelos autores, seguindo formalmente os roteiros pertinentes, eventualmente com a ajuda de pares.

### 6.8.2 Inspeções de Software

Inspeções representam um tipo de revisões formais por pares, ou *peer reviews*, as quais são técnicas de análise para avaliação de forma, estrutura e conteúdo de um documento, código fonte ou outro produto de trabalho [Wieger2002]. Essa técnica é realizada por um grupo de pessoas que têm o mesmo perfil (daí “pares”) do autor do produto a ser revisado, a fim de identificar discrepâncias do produto com base em padrões e especificações.

Em outras palavras, inspeção é um método formal de revisão por pares, onde um grupo de “pares”, incluindo o autor, se reúne para examinar um determinado produto de trabalho. O produto de trabalho é, tipicamente, submetido à inspeção quando o autor acredita que o mesmo foi concluído ou está pronto para ser “promovido” a uma próxima fase ou atividade do ciclo de vida. O foco da inspeção é a identificação de defeitos, com base em preparação prévia dos participantes. Vale salientar que métricas devem ser coletadas e utilizadas para determinar critérios de entrada para a reunião de inspeção, assim como para serem consideradas no processo de melhoria. A Figura 6.18 apresenta uma visão geral do processo de inspeção.

O resultado final de uma inspeção deve ser um dos seguintes:

- Material aceito sem modificações;
- Material aceito com pequenas modificações (não será necessária nova revisão técnica para o resultado do projeto em pauta, colocando-se um dos membros da revisão técnica à disposição do gerente do projeto para uma revisão informal das modificações);
- Material rejeitado para profundas modificações (haverá necessidade de nova revisão após serem feitas as modificações sugeridas);
- Material rejeitado para reconstrução (será necessária nova confecção do material);
- A revisão não foi completada (foi necessário cancelar ou interromper a reunião de revisão e uma nova revisão será marcada).

Na ausência de consenso para a classificação dos defeitos, prevalece o pior status proposto pelos revisores. Recomendam-se as seguintes diretrizes:

- Nas inspeções de projeto, violações dos requisitos levam, obrigatoriamente, à rejeição;

- Nas inspeções de implementação, violações do desenho levam, obrigatoriamente, à rejeição;
- Violações não justificadas de padrões pertinentes ao material levam, normalmente, à rejeição;
- Defeitos de forma (português, estética, falta de uniformidade de apresentação) levam, normalmente, à aceitação, com pequenas modificações, exceto no caso de documentos de usuário, caso em que levam à rejeição.

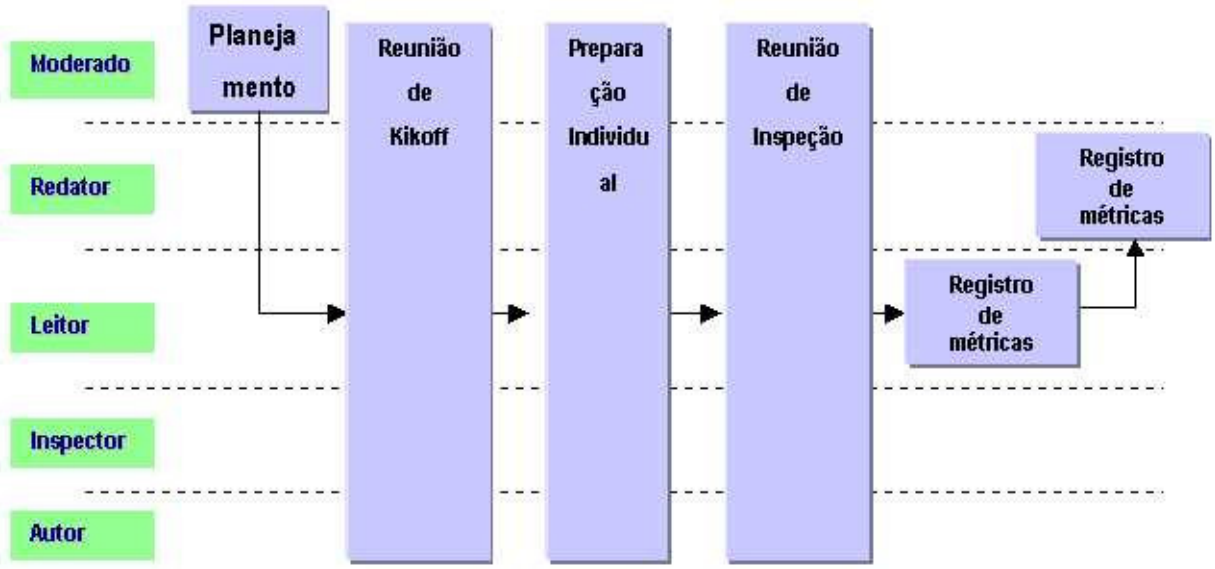


**Figura 6.18: Visão geral do processo de inspeção**

## 6.9 CONSIDERAÇÕES FINAIS

Nos últimos anos, com a crescente demanda por produtos mais eficazes e de baixo custo agregado, somado ao surgimento de um mercado sem limites de competitividade, a qualidade tornou-se um aspecto fundamental a qualquer organização.

Qualidade e produtividade são fatores que foram sendo tratados por diversos autores ao longo de vários anos, desde a década de 1950. Entre tais autores estão Crosby, Deming, Juran e outros que muito contribuíram com o setor produtivo, através de abordagens bastante relevantes, que servem de fundamentos para modelos propostos até hoje.

No contexto da qualidade de software, vários modelos vêm sendo publicados e são, hoje, largamente adotados por várias organizações no mundo. As normas ISO, os modelos propostos pelo *Software Engineering Institute-SEI*, como o *Capability Maturity*

*Model for Software (CMM),* são hoje considerados requisitos para organizações de software que desejam um lugar de destaque no mercado competitivo.

Nesse contexto, conceitos como prevenção e detecção, avaliações e auditorias, coleta e análise de métricas, entre outros, devem ser bem entendidos para se garantir uma visão clara do cenário da qualidade de software.

# 7

# QUALIDADE DE PRODUTO DE SOFTWARE

Qualidade é definida como **"**Totalidade de características de uma entidade que lhe confere a capacidade de satisfazer as necessidades explícitas e implícitas". Atualmente existem normas da ISO/IEC JTC1 que tentam definir as características de qualidade do produto de software. Essas características têm como principal objetivo organizar e tratar o produto de *software* sobre os seus diversos aspectos, contribuindo para que os desenvolvedores entendam e planejem o desenvolvimento do produto para atender aos aspectos de qualidade que mais são relevantes para o produto, visto que o atendimento de todas as características poderia ser economicamente inviável.

A principal norma de qualidade de produto é a ISO/IEC 9126 que especifica um modelo de qualidade o qual categoriza atributos de qualidade de *software* em seis características, as quais são, por sua vez, subdivididas em subcaracterísticas. Estas subcaracterísticas são manifestadas externamente quando o *software* é utilizado como parte de um sistema computacional e são um resultado de atributos internos de *software*. O efeito combinado das características de qualidade de *software* para o usuário é definido como qualidade em uso.

## 7.1 MODELO DE QUALIDADE

A qualidade de *software* pode ser definida e avaliada usando um modelo de qualidade definido. Um modelo de qualidade agrupa os vários aspectos do produto e no caso da norma temos 6 características de qualidade, que agrupadas formam a totalidade do produto de software.

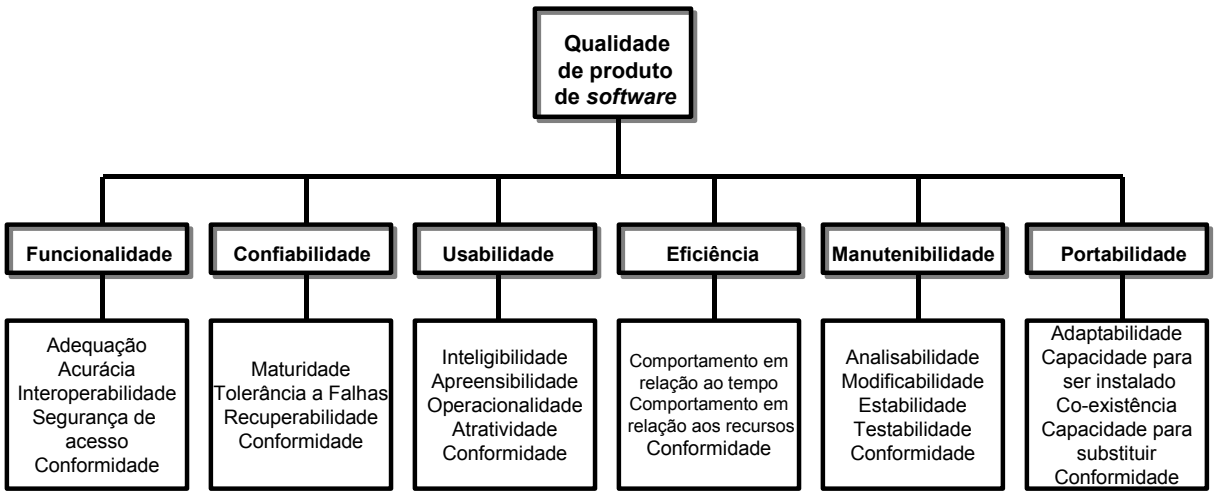


**Figura 7.1: Qualidade de produto de *software***

Os atributos de qualidade de *software* são categorizados em seis características (funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade), que são subdivididas dentro em subcaracterísticas. Subcaracterísiticas podem ser medidas através de métricas internas ou por métricas externas. Métricas internas básicas (tais como tamanho de programa) são medidas de *software* que normalmente não são usadas isoladas como métricas de qualidade de *software*, mais são usadas em combinação com outras medidas para criar métricas de qualidade.

São fornecidas definições para cada característica do *software* e respectivas subcaracterísticas que influenciam a característica de qualidade. A capacidade do *software* referente a cada característica e subcaracterística é determinada por um conjunto de atributos internos que possam ser medidos. Exemplos de métricas internas são fornecidos no documento ISO/IEC 9126-3. As características e subcaracterísticas podem ser medidas externamente pelo quanto a capacidade representada por elas é fornecida pelo sistema contendo o *software*. Exemplos de métricas externas são fornecidos no documento ISO/IEC 9126-2.

## 7.2 FUNCIONALIDADE

Capacidade do produto de *software* de prover funções que atendam necessidades explícitas e implícitas quando o *software* estiver sendo utilizado sob condições especificadas.

NOTAS

1 Esta característica está relacionada com o que *software* faz para atender necessidades, enquanto que outras características estão principalmente relacionadas com quando e como ele atende às necessidades.

2 Para as necessidades explícitas e implícitas nesta característica, a nota da definição de qualidade é aplicável, (ver A.21).

3 Para um sistema que seja operado por um usuário, a combinação de funcionalidade, confiabilidade, usabilidade e eficiência pode ser medido externamente pela qualidade em uso (ver seção 8).

### 7.2.1 Adequação

A capacidade do produto de *software* de prover um conjunto apropriado de funções para tarefas e objetivos do usuário especificados.

### 7.2.2 Acurácia

A capacidade do produto de *software* de prover resultados ou efeitos corretos ou acordados.

NOTA – Isto inclui os dados esperados de resultados de cálculo, com o grau de precisão necessário.

### 7.2.3 Interoperabilidade

A capacidade do produto de *software* de interagir com um ou mais sistemas especificados.

NOTA – Interoperabilidade é usada no lugar de compatibilidade para se evitar possível ambigüidade com capacidade de substituir (ver 7.6.4).

### 7.2.4 Segurança de acesso

A capacidade do produto de *software* para proteger informações e dados de forma que pessoas ou sistemas não autorizados não possam lê-los nem modificá-los e pessoas ou sistemas autorizados não façam acessos danosos a eles.

NOTAS

1 Isto também se aplica a dados na sua transmissão.

2 Segurança é definida como uma subcaracterística de qualidade em uso, já que ela não está relacionada somente com o *software*, mas com o sistema como um todo.

### 7.2.5 Conformidade

A capacidade do produto de *software* em estar de acordo com normas, convenções ou regulamentações em leis e prescrições similares.

## 7.3 CONFIABILIDADE

A capacidade do produto de *software* de manter um nível de desempenho especificado quando usado em condições especificadas.

NOTAS

1 Em *software* não ocorre desgaste ou envelhecimento. As limitações em confiabilidade são decorrentes de defeitos na especificação de requisitos, projeto e implementação. As falhas decorrentes desses defeitos dependem de como o produto de *software* é usado e das opções de programa selecionadas e não do tempo decorrido.

2 A definição de confiabilidade na ISO/IEC DIS 2382-14:1994 é “A habilidade de uma unidade funcional de executar uma função requisitada...”. Neste documento, funcionalidade é somente uma das características de qualidade de *software*. Portanto, a definição de confiabilidade tem sido expandida para “manter seu nível de desempenho...” no lugar de “... executar uma função requisitada”.

### 7.3.1 Maturidade

A capacidade do produto de *software* de evitar falhas decorrentes de defeitos no *software*.

### 7.3.2 Tolerância a falhas

A capacidade do produto de *software* de manter um nível de desempenho especificado em casos de defeitos no *software* ou de violação de sua interface especificada.

NOTA - O nível de desempenho especificado pode incluir a capacidade de suportar falhas.

### 7.3.3 Recuperabilidade

A capacidade do produto de *software* de restabelecer seu nível de desempenho e recuperar os dados diretamente afetados no caso de uma falha.

### 7.3.4 Conformidade

A capacidade do produto de *software* de estar de acordo com normas, convenções ou regulamentações relativos a confiabilidade .

## 7.4 USABILIDADE

A capacidade do produto de *software* de ser compreendido, aprendido, usado e apreciado pelo usuário, quando usado sob condições especificadas.

NOTAS

1 Alguns aspectos como funcionalidade, confiabilidade e eficiência também afetarão a usabilidade, mas para os propósitos da ISO/IEC 9126 não são classificados como usabilidade.

2 Usuários podem incluir operadores, usuários final e intermediário que estão sob influência de ou dependentes do uso do *software*. A usabilidade deve endereçar todos os diferentes ambientes que o *software* pode afetar, os quais podem incluir preparação para uso e avaliação de resultados.

### 7.4.1 Inteligibilidade

A capacidade do produto de *software* de permitir ao usuário reconhecer se o *software* se aplica a suas necessidades e como ele pode ser usado para determinadas tarefas e condições de uso.

NOTA - Isto dependerá da documentação e impressões iniciais oferecidas pelo *software*.

### 7.4.2 Apreensibilidade

A capacidade do produto de *software* de permitir ao usuário aprender sua aplicação.

NOTA - Os atributos internos correspondem a aplicabilidade para aprendizagem, como definido na ISO 9241-10.

### 7.4.3 Operacionalidade

A capacidade do produto de *software* de permitir o usuário operá-lo e controlá-lo.

NOTAS

1 Aspectos de aplicabilidade, modificabilidade, adaptabilidade e capacidade de instalação podem afetar a operacionalidade.

2 Operacionalidade corresponde à capacidade de controle, tolerância a erros e conformidade com as expectativas do usuário como definido na ISO 9241-10.

3 Para um sistema que é operado por um usuário, a combinação de funcionalidade, confiabilidade, usabilidade e eficiência pode ser medida externamente pela qualidade em uso.

### 7.4.4 Atratividade

A capacidade do produto de *software* de ser apreciado pelo usuário.

NOTA - Isto se refere a atributos de *software* que possuem a intenção de fazer o *software* mais atrativo para o usuário, como o uso de cores e da natureza do projeto gráfico.

### 7.4.5 Conformidade

A capacidade do produto de *software* de estar de acordo com normas, convenções, guias de estilo ou regulamentações relativas a usabilidade.

## 7.5 EFICIÊNCIA

A capacidade do produto de *software* de fornecer desempenho apropriado, relativo à quantidade de recursos usados, sob condições especificadas.

NOTAS

1 Recursos podem incluir outros produtos de *software*, facilidades de *hardware* e materiais ( por exemplo: impressora, disquetes) .

2 Para um sistema que é operado por um usuário, a combinação de funcionalidade, confiabilidade, usabilidade e eficiência pode ser medida externamente pela qualidade em uso.

### 7.5.1 Comportamento em relação ao tempo

A capacidade do produto de *software* de fornecer tempo de resposta e tempo de processamento apropriados e taxas de I-O (entrada e saídas) quando executando suas funções, sob condições estabelecidas.

### 7.5.2 Utilização de recursos

A capacidade do produto de *software* de usar quantidade e tipos de recursos apropriados quando o *software* executa suas funções sob condições estabelecidas.

### 7.5.3 Conformidade

A capacidade do produto de *software* de estar de acordo com normas e convenções relativas à eficiência**.**

## 7.6 MANUTENIBILIDADE

A capacidade do produto de *software* de ser modificado. As modificações podem incluir correções, melhorias ou adaptações do *software* devido a mudanças no ambiente ou nos seus requisitos.

### 7.6.1 Analisabilidade

A capacidade do produto de *software* de permitir o diagnóstico de deficiências ou causas de falhas no *software* ou a identificação de partes a serem modificadas.

### 7.6.2 Modificabilidade

A capacidade do produto de *software* de permitir que a modificação especificada seja implementada.

NOTAS

1 Implementação inclui modificação na codificação, projeto e documentação.

2 Se o *software* for modificável pelo usuário final, a modificabilidade pode afetar a operacionalidade.

## 7.7 ESTABILIDADE

A capacidade do produto de *software* de minimizar efeitos inesperados de modificações de *software*.

### 7.7.1 Testabilidade

A capacidade do produto de *software* de permitir o *software* modificado ser validado.

### 7.7.2 Conformidade

A capacidade do produto de *software* de estar de acordo com normas ou convenções relativas à manutenabilidade.

## 7.8 PORTABILIDADE

A capacidade do produto de *software* de ser transferido de um ambiente para outro.

NOTA - O ambiente pode incluir ambiente organizacional, de *hardware* ou de *software*.

### 7.8.1 Adaptabilidade

A capacidade do produto de *software* de ser adaptado para diferentes ambientes especificados sem necessidade de aplicação de outras ações ou meios além daqueles fornecidos para essa finalidade pelo *software* considerado.

NOTAS

1 Adaptabilidade inclui a possibilidade de alterar, por exemplo: campos de tela, tabelas, volume de transações, formato de relatórios etc.

2 Se o *software* for adaptável pelo usuário final, adaptabilidade corresponde à aplicabilidade para personalização como definido na ISO 9241-10 e pode afetar a operacionalidade.

### 7.8.2 Capacidade para ser instalado

A capacidade do produto de *software* para ser instalado em um ambiente especificado.

NOTA - Se o *software* é para ser instalado pelo usuário final, a capacidade para ser instalado afeta a adequação e a operacionalidade.

### 7.8.3 Coexistência

A capacidade do produto de *software* para coexistir com outros *softwares* independentes em um ambiente comum compartilhando recursos comuns.

### 7.8.4 Capacidade para substituir

A capacidade do produto de software para ser usado em substituição de outro produto de software especificado para o mesmo propósito no mesmo ambiente.

NOTAS

1 Por exemplo, a capacidade para substituir de uma nova versão de um produto de software é importante para o usuário quando estiver atualizando a versão.

2 A capacidade para substituir é utilizada no lugar de compatibilidade para evitar possível ambigüidade com interoperabilidade (veja 7.1.3).

3 A capacidade para substituir pode incluir atributos de capacidade para ser instalado e adaptabilidade. O conceito tem sido introduzido como uma subcaracterística própria devido a sua importância.

### 7.8.5 Aderência

A capacidade do produto de *software* para aderir a normas ou convenções relativas à portabilidade.

## CARACTERÍSTICAS DE QUALIDADE EM USO

O quanto que um produto usado por usuários especificados atende suas necessidades para atingir metas especificadas com eficácia, produtividade, segurança e satisfação em contextos de uso especificados.

NOTAS

1 Qualidade em uso é a visão do usuário da qualidade de um sistema contendo *software* e é medida em termos do resultado do uso do *software* ao invés das propriedades do próprio *software*.

2 Exemplos de métricas para qualidade em uso são dados na ISO/IEC 9126-2.

3 A definição de qualidade em uso em NBR 14598-1 (que é reproduzida no Anexo A) não inclui atualmente a nova característica de “segurança”.

4 Usabilidade é definida em ISO 9241-11 de uma maneira similar à definição de qualidade em uso nesta parte da NBR 9126. Qualidade em uso pode ser influenciada por qualquer das características de qualidade, e assim é definido na NBR 9126-1 em termos de inteligibilidade, apreensibilidade, operabilidade e atratividade.

## 7.9 EFICÁCIA

O quanto que o produto de *software* permite aos usuários atingir metas especificadas com acurácia e completitude em um contexto de uso especificado.

## 7.10 PRODUTIVIDADE

Os recursos dispendidos pelo sistema e usuários em relação à eficácia atingida quando o produto de *software* é utilizado em um contexto de uso especificado.

NOTA - Recursos relevantes podem incluir tempo, esforço, materiais ou custos financeiros.

## 7.11 SEGURANÇA

O quanto que o produto de *software* limita o risco de danos (para pessoas) ou avarias em um nível aceitável em um contexto de uso especificado.

NOTAS

1 A definição é baseada na ISO 8402:1994.

2 Segurança inclui o risco de danos para a saúde dos usuários de um sistema e riscos de avarias ou lesões resultante do uso do produto de *software*.

3 Riscos para a segurança são usualmente um resultado de deficiências na funcionalidade, confiabilidade ou usabilidade.

## 7.12 SATISFAÇÃO

O quanto que o produto de *software* satisfaz os usuários em um contexto de uso especificado.

NOTA - Questionários psicometricamente válidos podem ser utilizados para obter medidas de satisfação.

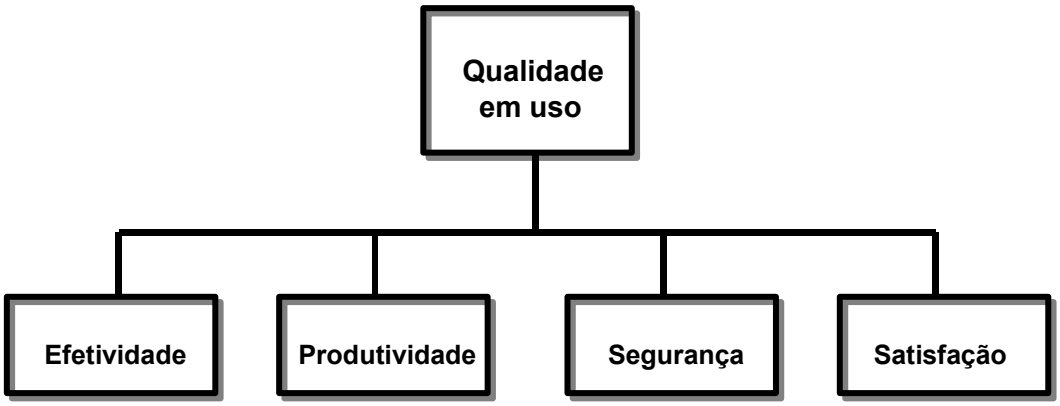


**Figura 7.2: Qualidade em uso**

# 8

# CONCLUSÃO

A Engenharia de Software é uma disciplina que está envolvida com todos os aspectos da produção de software, desde a sua concepção até a sua entrega, operação e manutenção. Neste livro, os aspectos da produção de software foram abordados com foco em três grandes áreas da Engenharia de Software: a Engenharia de Produto, a Gerência de Projetos e a Engenharia de Processo.

Em relação à engenharia de produto apresentaram-se os modelos de ciclo de vida e as principais atividades envolvidas no desenvolvimento de software. No que se refere à gerência de projetos de software abordou-se as dificuldades para sua execução, as principais atividades envolvidas e o PMBOK. Quanto à engenharia de processo foram apresentados aspectos relacionados à qualidade de software (a engenharia de processo é comumente vista como uma extensão da garantia da qualidade).

Vale ressaltar que o objetivo maior da Engenharia de Software é produzir software de qualidade, dentro do prazo e no custo desejado pelo cliente. Nesse contexto, os padrões e normas para SPA/SPI têm realizado uma excelente contribuição para a área auxiliando na definição, avaliação e melhoria dos processos de uma ODS. Todavia, apesar da sua importância, eles ainda estão sendo pouco considerados pelas ODSs. Diversos motivos dificultam o uso desses padrões, dentre eles o fato de que a definição e uso de um processo de software envolve uma complexa inter-relação de fatores organizacionais, culturais, tecnológicos e econômicos.

No que se refere ao gerenciamento de projetos de software especificamente, pode-se estar de acordo com as conclusões realizadas nos trabalhos de [Fernandes1989] e [Maidantchik1999]: “apesar do esforço da comunidade de engenharia de software em definir modelos e padrões para a construção de um efetivo processo de gerenciamento de projetos, a maioria das ODSs ainda sentem dificuldade em definir os seus processos e não gerenciam os seus projetos de forma satisfatória”.

Por fim, conclui-se que impulsionado pelas mudanças tecnológicas os produtos, as ODSs e seus processos associados mudaram no decorrer das últimas décadas. Atualmente, as fábricas de software são medidas por dois fatores que estão relacionados a qualquer outro tipo de indústria: qualidade de seus produtos e capacidade de ser cada vez mais produtiva. Essa é a essência atual para a sobrevivência e sucesso de uma empresa de software. Nesse contexto, a Engenharia de Software tem sido cada vez mais considerada pela comunidade de software por oferecer uma excelente contribuição.

# 9

# EXERCÍCIOS DE FIXAÇÃO

1. Defina com suas palavras o que você entende por Engenharia de Software.
2. Explique com suas palavras o que vem a ser a crise de software. Você acha que este é um problema que um dia terá uma solução definitiva? (defenda o seu argumento).
3. Tente relembrar, sem consultar o texto deste, quais são os principais modelos de ciclo de vida de software, descrevendo suas características.
4. No fórum virtual, coloque algumas considerações sobre a dificuldade de se implantar os conceitos da Engenharia de Software em uma organização.
5. Descreva com suas palavras (no máximo 4 parágrafos) as principais características, entradas e saídas dos processos de Planejamento e Gerenciamento, Requisitos, Projeto de Sistemas, Implementação, Testes e Gerência de Configuração.
6. Defina os conceitos de produto (*deliverable*), *release* e marco de referência (*milestone*).
7. Explique cada um dos estágios de teste.
8. Defina com suas palavras o que você entende por Qualidade de Software.
9. Explique com suas palavras o que vem a ser o TQC e o TQM.
10. No fórum virtual, coloque algumas considerações sobre a dificuldade de se implantar os conceitos da Qualidade de Software em uma organização.
11. Descreva com suas palavras o ciclo PDCA e o modelo IDEAL.
12. Descreva o que são auditorias e *assesments.*
13. Descreva sumariamente a metodologia proposta pela técnica Goal-Question-Metric para seleção de métricas.
14. Fale sobre as técnicas de verificação e validação.

# 10

# REFERÊNCIAS BIBLIOGRÁFICAS


[ABNT/SC10, 1999] - ABNT/SC10, ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS/SUBCOMITÊ DE SOFTWARE. Guia para utilização nas normas sobre avaliação de qualidade de produto de software – ISO/IEC 9126 e ISO/IEC 14598. Curitiba-PR:ABNT/SC10, 1999.

[Bellouquim1999] Bellouquim, A. *CMM em Pequenas Organizações: seria mesmo possível?* Developers Magfazine, Janeiro, 1999.

[Boehm1981] B. W. Boehm, "*Software Engineering Economics",* Prentice-Hall, Englewood Cliffs, NJ (1981).

[Boehm1989] B. W. Boehm, "*Software Risk Management",* IEEE Computer Society Press: Washington (1989).

[Booch1998] Grady Booch et al, "*The Unified Modeling Language user Guide",* Addisson-Wesley Object Technology Series, (1998).

[Campos1992] Campos, V. C. *TQC: Controle da Qualidade Total*, Fundação Christiano Otooni, 7ª edição, 1992.

[CMMI:2000] *CMMI Model Componentes Derived from CMMI – SE/SW*, Version 1.0 Technical report CMU/SEI-00-TR24. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000.

[Crosby1979] Crosby, P. *Quality is Free* - McGraw-Hill, 1979.

[D'Souza2001] D'Souza and Arlan Cameron Wills, *"Objects, Components, and Frameworks with UML:The Catalysis Approach",* (Addison-Wesley Object Technology Series), Desmond, Francis, ISBN: 0201310120, Addison-Wesley Pub Co.

[Davis1993] Davis, A. M., "*Software Requirements: Object, Functions & States"*, Englewood-Cliffs, NJ (1993).

[Dod1994] Defense Science Board*, Report of the Defense Science oard Task force on Acquiring Defense Software Commercially*, Washington D.C., Junho, 1994.

[Feingenbaum1994] Feingenbaum, A. V. *Controle da Qualidade Total II*, McGrawHill, 1994.

[Fernandes1989] Fernandes, Aguinaldo Aragon & Kugler, J. L. C*. Gerência de Projetos de Sistemas*. Rio de Janeiro, LTC, 1989.

[Ford1994] N. J. Ford and M. Woodroffe, "*Intoducing Software Engineering",* Prentice Hall Englewood Cliffs (1994).

[Garg1996] Garg, P. K. *Process-centered Software Engineering Environments* Published by IEEE Computer Society Press, Los Alamitos, CA 90720-1264, 1996.

[Gibbs1994] *Software's Chronic Crisis*. Trends In Computing,W. Wayt Gibbs. (1994)

[Gotel1994] O.C.Z. Gotel and A.C.W. Finkelstein, *"An Analysis of the Requirements Traceability Problem"*, pp. 94-101 in *Proceedings of the First International Conference on Requirements Engineering*, Colorado Springs, CO (1994).

[Graham1999] Graham, I. S & Henderson-Sellers, B. & Younessi, H. *The OPEN Process Specification*, Addison Wesley Longman Inc. outubro, 1999.

[Humphrey1989] Humphrey, W. *Managing the Software Process*. Addison Wesley, 1989.

[IEEE1984] IEEE Std. 830 "IEEE Guide to Software Requirements Specification". *The Insitute of Electrical and Electronics Engineers*: New York (1984).

[IEEE1990] IEEE Std. 610.12 "IEEE Standard Glossary of Software Engineering Terminology". *The Institute of Electrical and Electronics Engineers*: New York (1990).

[IEEE1999a] IEEE Standards Software Engineering, Vol.2 – *IEEE Standards for Software Quality Assurance Plans* – IEEE Std.730-1997, IEEE, 1999.

[IEEE1999b] IEEE Standards Software Engineering, Vol.2 – *IEEE Standards for Software Reviews* – IEEE Std.730-1997, IEEE, 1999.

[ISO/IEC 9126-2] - the International Organization for Standardization and the International Electrotechnical Commission. ISO/IEC TR 9126-2:2003, Software engineering - Product quality - Part 2: External metrics. Geneve: ISO, 2002.

[ISO/IEC 9126-3] - the International Organization for Standardization and the International Electrotechnical Commission. ISO/IEC TR 9126-3:2003, Software engineering - Product quality - Part 3: Internal metrics. Geneve: ISO, 2002.

[ISO/IEC 9126-4] - the International Organization for Standardization and the International Electrotechnical Commission. ISO/IEC TR 9126-4:2004, Software engineering - Product quality - Part 4: Quality in Use. Geneve: ISO, 2002

[ISO12207:1995] ISO/IEC 12207*, Information Technology – Software Life-Cycle Processes*, International Standard Organization, Geneve,1995.

[ISO12207:2000] International Standard Organization. *ISO/IEC 12207 Amendement: Information Technology – Amendement to ISO/IEC 12207*, versão PDAM 3, novembro 2000.

[ISO15504:1-9:1998] ISO/IEC TR 15504, Parts 1-9: *Information Technology – Software Process Assessment*, 1998.

[ISO9000-3:1997] ISO9000-3, *Quality management and Quality Assurance Standards.*

[ISO9001:2000] International Standard Organization Certification for IMS Company.

[Jones1996] Jones, C. *Patterns of Software Systems Failure and Success*. International Thomson Computer Press, Boston, Massachusetts, 1996.

[Jones1999] Jones, C. *Software Project Management in the 21st Century*. American Programmer, Volume XI, N. 2, fevereiro 1999.

[Kan1995] Kan, Stephen – *Metrics and Models en Software Quality Engineering* – Addison-Wesley, Reading, 1995.

[Kruchten1998] Philippe Kruchten, "*The Rational Unified Process: an Introduction"*, Addisson-Wesley Object Technology Series, (1998).

[Kuvaja1993] Kuvaja, P. et all *BOOTSTRAP: Europe´s assessment method*, IEEE Software, vol 10, N. 3, 1993.

[Kuvaja1994] Kuvaja, P. et al. *Software Process Assessment & Improvement - The BOOTSTRAP Approach*, Blackwell, 1994.

[Lubars1993] M. Lubars, C. Potts, and C. Richter, "A Review of the State of Practice in Requirements Modeling," pp. 2-14 in *Proceedings of the IEEE International Symposium on Requirements Engineering*, San Diego, CA (January, 1993)

[MacFeeley1999] McFeeley, Bob – *IDEALSM: A User's Guide for Software Process Improvement* – Software Technology Conference, 1999. (www.sei.cmu.edu/ideal)

[Machado2001] Machado, C. A. & Burnett, R. C. *Gerência de Projetos na Engenharia de Software em Relação as Práticas do PMBOK*. XII CITS - Conferência Internacional de Tecnologia de Software. Junho, 2001.

[Maidantchik1999] Maidantchik, C. & Rocha, A R. C. & Xexeo, G. B *Software Process Standardization for Distributed Working Groups*. In Proceedings of the 4 th IEEE International Software Engineering Standards Symposioum, Curitiba, Paraná, Brasil, maio de 1999.

[Meyer1988] Meyer, B., *Object-Oriented Software Construction*, Prentice Hall, (1988).

[NBR ISO/IEC 9126-1] - ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. NBR ISO/IEC 9126-1:2003 - Engenharia de software - Qualidade de produto - Parte 1: Modelo de qualidade. Rio de Janeiro: ABNT, 2003.

[Paulk1993] Paulk, M. & Curtis, B. & Crissis, M & Weber, C*. Capability Maturity Model for Software*, Version 1.1, Technical report CMU/SEI-93-TR-24, Software Engineering Institute, Pittsburgh, fevereiro, 1993.

[Paulk1995] Paulk, M.C. et al. – *The Capability Maturity Model: Guidelines for Improving the Software Process* - Addison-Wesley, 1995.

[Paulk1997] Paulk, M. C & Weber, C. V & Curtis, B. & Chrissis, M. B. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Carnegie Mellon University, Software Engineering Institute, Addison-Wesley Longman Inc, 1997.

[Philippe1998] Philippe, K. *The Rational Unified Process: an Introduction*. Addison-Wesley Object Technology Series, 1998.

[PMBOK2000] *A Guide to the Proiject Management Body of Knowledge*, PMI-Project Management Institute, Newtown Square, Pennsylvania, USA, 2000.

[Pressman2001] R. S. Pressman., "*Software Engineering: A Practioner's Approach",* 5th edition, McGraw-Hill International Editions, (2001).

[Rouiller2001] Rouiller A. C Gerenciamento de Projetos de Software para Pequenas Empresas. Tese de doutorado. Universidade Federal de Pernambuco, 2001. www.uflatec.com.br/ana.

[Royce1998] Royce, W. *Software Project Management: a unified framework*. Addison Wesley Longman, 1998, USA.

[Ryan1993] K. Ryan, "The Role of Natural Language in Requirements Engineering," pp. 240-242 in *Proceedings of the IEEE International Symposium on Requirements Engineering*, San Diego, CA (January, 1993).

[Sanches2001] Sanches, R. & Júnior, W. T. *Proposta de um Modelo de Processo de Planejamento de Projeto de Software para Melhoria de Gerenciamento de Projetos*. XII CITS - Conferência Internacional de Tecnologia de Software, junho, 2001.

[SEI2000] Sei, An Overview of Capability Maturity Model Integration (CMMI) – Version 1.0, Tutorial presented at SIMPROS 2000 [23], 2000.

[SEI2002a] Sei, Web Site do software Engineering Institute – SEI, http://www.sei.cmu.edu/ (CMMI nodels available at www.sei.cmu.edu/cmmm).

[SEI2002b] Sei, Web Site do software Engineering Institute – SEI, http://www.sei.cmu.edu/ (CMMI nodels available at www.sei.cmu.edu/cmmm).

[SEI2002c] Sei, Web Site do software Engineering Institute – SEI, http://www.sei.cmu.edu/ (CMMI nodels available at www.sei.cmu.edu/cmmm).

[Spivey1989] J. M. Spivey, "*The Z Notation"*, Prentice Hall, (1989).

[Standish2001] The Standish Group, "Extreme Chaos", www.standishgroup.com/sample_research/PDFpages/extreme_chaos.pdf, acessado em outubro de 2004.

[Stokes1994] D. A. Stokes, "Requirements Analysis", chapter 16 of *Software Engineering's Handbook,* Ed. John Mcdermid, Butterworth Heinemann (1994).

[Vigder1994] Vigder, M. R. & Kark, A. W. *Software Cost Estimation and Control*. National Research Council Canada. Institute for Information Technology, 1994. http://wwwsel.iit.nrc.ca/abstracts/NRC37116.abs, acessado em dezembro de 2001.

[Walker1997] Walker, E. *Managing Successful Iterative Development Projects: A Seminar on Software Best Practices*, version 2.3, Rational Software Corporation, Menlo Park, California, 1997.

[Wang1999] Randolph Y. Wang1999, David A. Patterson, and Thomas E. Anderson. Virtual log based file systems for a programmable disk.

[Weber1999] Weber, K. C *Qualidade e Produtividade em Software*. 3 ed. São Paulo: Makron Books do Brasil Ltda, 1999.

[Wieger2002] Wieger, Karl – *Peer Reviews in Software* – Addison-Wesley, Boston, 2002.

# ANEXO A – PADRÃO DE CODIFICAÇÃO JAVA

## 1.1 Arquivos fonte Java

Cada arquivo fonte Java contém uma classe pública ou uma interface. Quando classes privadas e interfaces são associadas a uma classe pública, você poderá colocá-las em um mesmo arquivo fonte. A classe pública ou a interface deverá ser a primeira no arquivo.

Arquivos fontes Java possuem a seguinte ordem:

- Comentários iniciais (opcional);
- Instruções (*statements)* de pacotes *(package)* e importações *(import)*;
- Declaração de classes e interfaces.

Exemplo:

```
//comentários iniciais (opcional)
/*
 * PesquisaCEP.ajva
 *
 * Versão 1.0
 *
 * Data: 01/05/2004
 *
 * Copyright(c) 2006 - UFLA
 */

//pacote ao qual a classe pertence
```

```
package pesquisa.cep;


//importações de classes
import java.sql.Connection;
import java.util.Collection;
import gov.br.UFLA.reuso.cep.Localidade;
import gov.br.UFLA.reuso.cep.UF;


//declaração de classe
public class PesquisaCEP {
        ...
}
```

## 1.2 Comentários Iniciais (opcional)

Todos os arquivos fontes deverão começar com um comentário *c-style* (padrão C) que lista o nome da classe, versão e notas de direitos autorais.

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
/*
```

## 1.3 Package e Import

A primeira linha 'sem-comentários' da maioria dos arquivos fontes Java é um *statement* de *package*. Depois disso, *statements* de *import* podem seguir. Por exemplo:

```
package java.awt;

import java.awt.peer. CanvasPeer;
```

## 1.4 Declaração de Classes e Interfaces

A tabela seguinte descreve as etapas (partes) da declaração de uma classe ou interface, na ordem em que deverão aparecer.

| | *Partes da declaração de classes e interface* | *Observações* |
|---|---|---|
| 1 | Comentários de Documentação (/**..*/) | Veja " Comentários de documentação" para informações sobre como usar este comentário |
| 2 | Declaração de Classe ou Interface. | |
| 3 | Implementação da classe / interface, comentário (/* ... */), se necessário. | Este comentário deverá conter qualquer informação sobre a classe ou interface, que não seja apropriado para documentação. |
| 4 | Variáveis Estáticas (static). | Primeiramente as variáveis |

| | *Partes da declaração de classes e interface* | *Observações* |
|---|---|---|
| | | públicas (public) da classe, então as protegidas (protected) e finalmente as privadas (private). |
| 5 | Instância de Variáveis | Primeiro a *public*, então a *protected* e finalmente a *private*; |
| 6 | Construtores | |
| 7 | Métodos | Os métodos devem ser agrupados pela funcionalidade de preferência pelo escopo ou acessibilidade. Por exemplo, um método privado de uma classe pode estar entre 2 duas instâncias de métodos públicos. O objetivo é tornar a leitura e o entendimento do código algo simples. |

**Nota:** ver exemplo no item 9.

## 2. Endentação

Quatro espaços devem ser utilizados como unidade de endentação.

**Nota:** Recomenda-se utilizar a opção Format do Eclipse ou Reformat Code do NetBeans para facilitar a estruturação e organização apresentadas em todo o item 2.

## 2.1 Tamanho de Linha

Evite linhas maiores que 80 caracteres, geralmente não mais que 70 caracteres.

## 2.2 Wrapping Lines

Quando uma expressão não se ajusta somente em uma linha, quebre-a de acordo com estes princípios:

- Quebra depois de uma vírgula;
- Quebra antes de um operador;
- Alinhar a nova linha com o início de uma expressão no mesmo nível da linha anterior.

Aqui estão alguns exemplos de quebra para chamada de métodos:

```
someMethod(longExpression1, longExpression2, longExpression3, longExpression4,
longExpression5);


var = someMethod1(longExpression1,
someMethod2(longExpression, longExpression));
```

Os dois exemplos, a seguir, são de quebra de expressões aritméticas. O primeiro é o preferido, desde que a quebra ocorra fora dos parênteses.

```
longName1 = longName2 * (longName3 + longName4 – longName5)
                + 4 * longName6; // recomendado


longName1 = longName2 * (longName3 + longName4
                            - longName5) + 4 * longName6; // evitar
```

Os dois exemplos, a seguir, são de endentação para declarações de métodos. O primeiro é o caso convencional. No segundo exemplo poderia ser deslocada a segunda e terceira linha para a direita, ao invés de endentar somente com 8 espaços.

```
// ENDENTAÇÃO CONVENCIONAL
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
    . . .
}
```

```
// ENDENTAÇÃO COM 8 ESPAÇÕS PARA EVITAR ENDETAÇÕES PROFUNDAS
private static synchronized horkingLongMethodName(int anArg,
        Object anotherArg, String, yetAnotherArg,
        Object andStillAnother) {
   . . .
}
```

Line wrapping para declarações **if** deveriam usar a regra de the 8 espaços, desde que a convencional (4 espaços) endentação dificulte a visão do corpo. Por exemplo:

```
// NÃO USE ESTA ENDENTAÇÃO
if ((condition1 && condition2)
    || (condition3 && condition4)
```

```
    || !(condition5 && condition6)) {        // bad wraps
   doSomethingAboutIt();                     // torna esta linha fácil de perder-se
}

// USE ESTA ENDENTAÇÃO
if ((condition1 && condition2)
        || (condition3 && condition4)
        || !(condition5 && condition6)) {
   doSomethingAboutIt();
}

// OU USE ESTA
if ((condition1 && condition2) || (condition3 && condition4)
        || !(condition5 && condition6)) {
   doSomethingAboutIt();
}
```

Aqui são apresentadas três maneiras para formatar operadores ternários:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;
```

```
alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

## 3. Comentários

Os programas Java podem ter dois tipos de comentários: comentários de implementação e comentários de documentação. Comentários de implementação são aqueles encontrados em C++, este é delimitado por /* . . . */, e / /. Comentários de documentação (conhecido como “doc comments”) são somente Java e são delimitados por /** . . . */. Comentários doc podem ser extraídos para arquivos HTML usando a ferramenta javadoc.

Comentários de implementação são significativos para comentar sobre uma implementação em particular. Comentários Doc são significativos para descrever a especificação de um código, de uma perspectiva livre de implementação para ser lida por desenvolvedores que provavelmente, não necessariamente possuem o código fonte em mãos.

Comentários devem ser usados para dar uma visão geral do código e prover informação adicional para determinada parte do código, na qual apenas lendo-se, o código não é possível o entendimento. Comentários devem conter somente informações relevantes para a leitura e entendimento do programa. Por exemplo, informações sobre como o determinado pacote é construído ou em qual diretório ele está armazenado, não devem ser incluídos como um comentário.

**Nota:** A freqüência de comentários algumas vezes reflete baixa qualidade do código. Quando você se sente forçado a adicionar um comentário, considere reescrever o código para deixá-lo limpo.

Comentários não devem ser colocados em grandes caixas de desenho com asteriscos ou outros caracteres. Comentários nunca devem incluir caracteres especiais.

### 3.1 Formatos de implementação de comentários

Programas podem ter 4 estilos de comentários de implementação: bloco, linha-simples, trailing e final de linha.

### 3.1.1 Comentários de bloco

Comentários de bloco são usados para prover descrições de arquivos, métodos, estrutura de dados e algoritmos. Comentários de bloco podem ser usados no início de cada arquivo e antes de cada método. Eles podem, também, ser usados em outros lugares, como dentro dos métodos. Comentários de bloco dentro de um função ou método devem ser endentados no mesmo nível do código.

Um comentário de bloco deve ser precedido por uma linha em branco para definilo como "fora" do código.

```
/*

      * Aqui é um comentário de bloco.
 */
```

### 3.1.2 Comentário de linha simples

Comentários curtos podem aparecer em uma linha simples endentada no nível do código que ela segue. Se um comentário não puder ser escrito como uma linha simples, este deverá seguir o formato de um comentário de bloco. (ver item 3.1.1). Um comentário de linha simples deverá ser precedido de uma linha em branco. Aqui um exemplo de um comentário de linha simples dentro de um código Java.

```
If (condition) {


   /*  aqui o comentário para condição */

  . . .
}
```

### 3.1.3 Comentários Trailing

Comentários muito curtos podem aparecer na mesma linha de código, mas deverá estar longe o suficiente para separá-los de outras instruções. Se mais de um comentário curto aparecer em um bloco de código, todos eles deverão estar usando a mesma endentação ou tabulação.

Aqui um exemplo de comentário trailing em um código Java:

```
If (a == 2)
   return TRUE;                    /* caso especial */
} else {
   return isPrime(a);              /* trabalha somente para impar */
}
```

### 3.1.4 Comentários de final de linha

O delimitador de comentário ‘ // ‘ pode ser utilizado para comentar uma linha de código toda ou parte dela, bem como excluir uma parte do código de sua seção. A seguir, exemplos dos três estilos:

```
If (foo > 1) {
   // coloque o comentário aqui.
   . . .
}
else {
      return false;                // Explique a funcionalidade.
}

//if (bar > 1) {
//
//         //coloque o comentário aqui.
//          ...
//}
//else {
```

```
        return false;
    //}
```

**Nota:** não comentar um código se ele não é usado.

## 3.2 Comentários de documentação

Comentários doc (comentários de documentação) descrevem classes Java, interfaces, construtores, métodos e atributos. Cada comentário doc é definido dentre os delimitadores ‘ /** . . . */ ‘, com um comentário por classe, interface. Este comentário deverá aparecer antes da declaração:

```
/**
 * A classe Exemplo e funcionalidades . . .
 * /
public class Exemplo {  . . .
}
```

Note que classes *top-level* e interfaces não são endentadas, enquanto seus membros são. A primeira linha de um comentário doc (/**) para classes ou interfaces não é endentada, as linhas subseqüentes de comentários doc tem cada uma 1 espaço de endentação (verticalmente, alinhando os asteriscos).

Comentários doc não deverão ser inseridos dentro de um método ou de um bloco de definição de construtores, devido ao Java associar comentários de documentação com a primeira linha depois do comentário.

**Nota:** ver exemplo no item 9.

# 4. Declarações

## 4.1 Números por Linha

Uma declaração por linha é recomendada desde que seja acompanhada de comentários. Em outras palavras,

```
int level;          // qual o nível
```

```
int size;          //  tamanho da tabela
```

é melhor que

```
int level, size;
```

Não coloque diferentes tipos de dados na mesma linha. Exemplo:

```
int foo, fooarray[];
```

**Nota:** Os exemplos acima usam um espaço entre o tipo e o identificador. Outra alternativa aceitável é o uso de *tabs* (tabulações), exemplo:

```
int         level;          // qual o nível
int         size;           //  tamanho da tabela
Object      currentEntry;   // tabela de entrada de dados
```

## 4.2 Inicialização

Tente inicializar variáveis locais onde elas são declaradas. A única razão, para não inicializar uma variável onde ela é declarada é se o valor inicial depende de algum outro processamento primeiro.

## 4.3 Placement (localização)

Coloque declarações somente nos início dos blocos. (Um bloco é definido pelo código que está entre as chaves '{' e '}'). Não espere a variável ser usada a primeira vez para declará-la ou inicializá-la; isto pode confundir um programador descuidado.

```
void myMethod() {
    int int1 = 0;                    // No inicio do bloco do método.

    If (condition) {
        int int2 = 0;                // inicio do bloco " if "
        . . .
```

```
        }
    }
```

A única exceção para esta regra são os índices, como de *loops* **for**, que em Java podem ser declarados na própria instrução.

```
For (int i = 0; i < maxLoops; i++) { . . . }
```

Evite declarações locais que escondem declarações de níveis maiores. Por exemplo, não declare o mesmo nome de variável em um bloco interno:

```
int count;
. . .
myMethod() {
    if (condition) {
        int count;                  // evitar!
        . . .
    }
    . . .
}
```

## 4.4 Declarações de classes e interfaces

Quando estiver codificando classes e interfaces Java, as seguintes regras de formatação deverão ser seguidas:

- Sem espaços entre um método e o parênteses " ( "que inicia a lista de parâmetros;
- A chave "{" aparece no final da mesma linha da declaração da instrução;
- A chave "}" aparece no final do bloco da instrução endentada de acordo a instrução, exceto no caso em que instrução é nula, assim a chave fecha logo após a que foi aberta "{".

```
Class Sample extends Object {
```

```
        int ivar1;
        int ivar2;

        Sample (int i, int j) {
            ivar1 = i;
            ivar2 = j;
         }
         int emptyMethod() {}
         . . .
}
```

- Métodos são separados por uma linha em branco.

## 5. Statements (instruções)

### 5.1 Instruções simples

Cada linha deverá conter no máximo uma instrução. Exemplo:

```
argv++;                    // Correto
argc++;                    // Correto
argv++; argc --;           // Evitar
```

### 5.2 Instruções compostas

Instruções compostas são instruções que possuem uma lista de instruções "internas" entre chaves "{ instruções }". Veja os exemplos nas próximas sessões.

- Instruções "internas" deverão ser endentadas um ou mais níveis que as instruções compostas;

- A chave aberta “{“ deverá estar no final da linha, onde inicia-se a instrução composta; a chave fechada “}” deverá começar em uma linha endentada com o inicio da instrução composta;

- As chaves são usadas “ao redor” de todas as instruções, exceto para instruções simples, quando elas são partes de uma estrutura de controle, tais como uma instrução IF-ELSE or FOR.

## 5.3 Instrução Return

A instrução return com um valor não deverá usar parênteses, ao menos que este valor torne-se mais claro com o uso de parênteses.

```
return;

return myDisk.Size();

return (size ? size : defaultSize);
```

## 5.4 Instruções if, if-else, if else-if else

A instrução if-else deverá utilizar uma das seguintes formas:

```
if  (condition) {
    statements;
}

if (condition) {
```

```
    statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
  statements;
} else
    statements;
}
```

**Nota:** Instruções if sempre utilizam chaves {}. Evite usar as seguintes formas, pois são consideradas como erros.

```
If (condition)          // evite, omitir as chaves nestes casos.
    statements;
```

## 5.5 Instrução for

Uma instrução for deverá ter a seguinte forma:

```
for (inicialização; condição; atualização) {
    statements;
}
```

Quando utilizar o operador vírgula na cláusula de inicialização ou atualização de uma instrução for, evite usar mais de 3 variáveis. Se necessário, use instruções em separado antes do laço for (para inicialização da cláusula) ou no final do loop (para a cláusula de atualização).

## 5.6 Instrução while

Uma Instrução while deverá ter a seguinte forma:

```
while (condition) {
    statements;
}
```

Uma instrução while vazia deverá ter a seguinte forma:

```
while (condition)
```

## 5.7 Instrução do-while

Uma instrução do-while deverá ter a seguinte forma:

```
do {
     statements;
} while (condition);
```

## 5.8 Instrução switch

Uma instrução switch deverá ter a seguinte forma:

```
switch (condition) {
case ABC:
        statements;
        /* "falsa passagem" */
case DEF:
        statements;
        break;

case XYZ:
        statements;
        break;

default:
       statements;
       break;
}
```

Cada instrução *switch* deverá incluir um caso *default*. O uso do *break* em uma instrução default é redundante, mas não previne os erros de "passagens falsas" se outra instrução *case* for adicionada.

## 5.9 Instruções try-catch

Uma instrução *try-catch* deverá ter o seguinte formato:

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
}
```

Uma instrução *try-catch* pode também ser seguida da instrução *finally,* esta não será levada em consideração quando bloco *try* não for completado com sucesso.

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
} finally {
    statements;
}
```

# 6. Espaços em branco

## 6.1 Linhas em branco

Linhas em branco facilitam a leitura do código, uma vez que se define seções de código relacionadas.

Duas linhas em branco deverão ser sempre usadas para as seguintes circunstâncias:

- Entre seções de um arquivo fonte;
- Entre a as declarações de classes e interfaces.

Uma linha em branco deverá sempre ser usada nas seguintes circunstâncias:

- Entre métodos;

- Entre variáveis locais em um método e sua primeira declaração;
- Antes de um bloco ou de um comentário simples;
- Entre seções lógicas dentro de um método para facilitar a leitura.

## 6.2 Espaço em branco

Espaços em branco deverão ser usados nas seguintes circunstâncias:

- Uma palavra chave seguida de parênteses deverá ser separada por um espaço. Exemplo:

```
while (true) {
    . . .
}
```

Note que um espaço em branco não deve ser usado entre o nome de um método e o parênteses que será aberto. Isto ajuda a distinguir “as chaves” da chamada de um método.

- Um espaço em branco deverá aparecer depois de vírgulas em listas de argumentos;
- Todos os operadores binários exceto “ . “ deverão ser separados de seus operandos por espaços. Espaços em branco nunca deverão aparecer entre operadores unários, como o operador “-“ ou o operador de incremento “++”, e decremento “—“. Exemplos:

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++) {
   n++;
}
```

prints(“size is ” + foo + “\n”)

- As expressões dentro de uma instrução *for* deverão ser separadas por espaços em branco. Exemplo:

For (expr1; expr2; expr3)

- Operadores de *cast* deverão ser seguidos por um espaço em branco. Exemplo:

myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (I + 3)) + 1);