Qualidade de Software

Cassiane de Fátima dos Santos Bueno - cfsb@di.ufpe.br
Gustavo Bueno Campelo - gbc@di.ufpe.br

Departamento de Informática
Universidade Federal de Pernambuco - Recife, PE

1. Resumo

O objetivo deste artigo é apresentar um conjunto de características de qualidade no desenvolvimento de um sistema baseado em computador, tendo como idéia central a melhoria e a "medida" da qualidade do software a ser desenvolvido, através da aplicação de conceitos sobre atributos de qualidade, métricas de qualidade de software, sistemas de gerenciamento de qualidade e revisões de software. Além disso, será feita uma breve descrição e comparação entre dois sistemas de gerenciamento de qualidade de software, o ISO 9001 e o CMM.

2. Introdução

O principal objetivo da engenharia de software é ajudar a produzir software de qualidade.

Conceitos de qualidade são imprecisos e difíceis de serem aceitos por todas as pessoas, no entanto, métricas de qualidade de software surgem desde a década de 70 e vêm se desenvolvendo de forma a ajudar no processo de desenvolvimento de software.

A garantia de controle de qualidade de software está intimamente relacionada a atividades de verificação e validação e estão presentes em todo o ciclo de vida do software. Em algumas organizações não existe distinção entre essas atividades. Entretanto, a garantia de qualidade e os processos de verificação e validação de software devem ser atividades distintas. A garantia de qualidade é uma função gerencial, enquanto que a validação e a verificação são processos técnicos no desenvolvimento de software.

Dentre os modelos de gerenciamento de controle de qualidade de software mais conhecidos estão o Capability Maturity Model (CMM) e o ISO 9000-3, que foram motivados pelas falhas nos processos de gerência e manutenção durante o desenvolvimento de software [CESAR97].

3. Conceitos de Qualidade

Definir qualidade de software é uma tarefa difícil. Muitas definições têm sido propostas e uma definição decisiva poderia ser debatida interminavelmente.

3.1. Qualidade

O Dicionário Aurélio define qualidade como: "propriedade, atributo ou condição das coisas ou das pessoas capaz de distingui-las das outras e de lhes determinar a natureza" [Aurélio86]. Como um atributo de um item, a qualidade se refere a coisas que

podem ser medidas, ou seja, comparadas com padrões conhecidos, tais como, tamanho, cor, propriedades elétricas, maleabilidade, etc. Entretanto, é mais difícil categorizarmos qualidade em software, que é uma entidade intelectual, do que em objetos físicos.

Ao se examinar um item baseado em suas características mensuráveis, dois tipos de qualidade podem ser encontrados: qualidade de projeto e qualidade de conformidade [Pressman97].

Qualidade de projeto se refere a características que projetistas especificam para um item (performance, tolerância, etc.). O enfoque maior é nos requerimentos, na especificação e no projeto do sistema.

Qualidade de conformidade é o grau no qual as especificações do projeto são seguidas durante o processo de desenvolvimento. O enfoque maior é na implementação.

Uma definição de qualidade de software que se encaixa no escopo deste artigo é: "conformidade a requisitos funcionais e de desempenho explicitamente declarados, a padrões de desenvolvimento claramente documentados e a características implícitas que são esperadas de todo software profissionalmente desenvolvido" [Pressman95].

3.2. Controle de Qualidade

Pela definição da ISO, controle de qualidade é "a atividade e técnica operacional que é utilizada para satisfazer os requisitos de qualidade" [McDermid94].

O controle de qualidade é feito através de uma série de inspeções, revisões e testes, usados através do ciclo de desenvolvimento para garantir que cada trabalho produzido está de acordo com sua especificação/requerimento. Portanto, o controle de qualidade é parte do processo de desenvolvimento e, como é um processo de *feedback*, ele é essencial para minimizar os defeitos produzidos.

3.3. Garantia de Qualidade

A garantia de qualidade de software não é algo com o qual se começa a pensar depois que o código é gerado. A Garantia de Qualidade de Software ou *SQA (Software Quality Assurance*) é uma atividade que é aplicada ao longo de todo o processo de engenharia de software. Ela abrange:

- métodos e ferramentas de análise, projeto, codificação e teste;
- revisões técnicas formais que são aplicadas durante cada fase da engenharia de software;
- uma estratégia de teste de múltiplas fases;
- controle da documentação do software e das mudanças feitas nela;
- um procedimento para garantir a adequação aos padrões de desenvolvimento de software, se eles forem aplicados;
- mecanismos de medição e divulgação.

Geralmente, a garantia de qualidade consiste daqueles procedimentos, técnicas e ferramentas aplicadas por profissionais para assegurar que um produto atinge ou excede padrões pré-especificados durante o ciclo de desenvolvimento do produto; se tais padrões não são aplicados, a garantia de qualidade assegura que um produto atinge ou excede um nível de excelência (industrial ou comercial) mínimo aceitável.

3.4. Custo de Qualidade

Como seria utópico alcançar a perfeição em um sistema de computação, então o mais importante passa a ser definir qual nível de checagem (de qualidade) seria suficiente para o sistema em questão e os custos associados.

O custo de qualidade inclui todos os gastos financeiros relacionados às atividades de qualidade, os quais podem ser divididos em: custos de prevenção, custos de avaliação e custos de falhas.

Os custos de prevenção incluem:

- planejamento da qualidade;
- revisões técnicas formais;
- teste de equipamentos;
- treinamento.

Os custos de avaliação incluem:

- inspeções dos processos e relações entre eles;
- manutenção dos equipamentos;
- testes.

Os custos de falhas poderiam desaparecer se nenhum defeito ocorresse antes da entrega do produto para o cliente. Os custos de falhas podem ser divididos em: custos de falhas internas e custos de falhas externas.

Os custos de falhas internas incluem:

- retrabalho;
- conserto de *bugs*;
- análise de falhas.

Os custos de falhas externas incluem:

- resolução de queixas;
- troca/devolução do produto;
- suporte em *help on-line*;
- trabalhos de segurança.

Os custos relacionados a encontrar e consertar um defeito aumentam drasticamente quando vamos da prevenção de falhas internas para a prevenção de falhas externas. Dados coletados por Boehm [Boe81], ilustram esse fenômeno (figura 1).

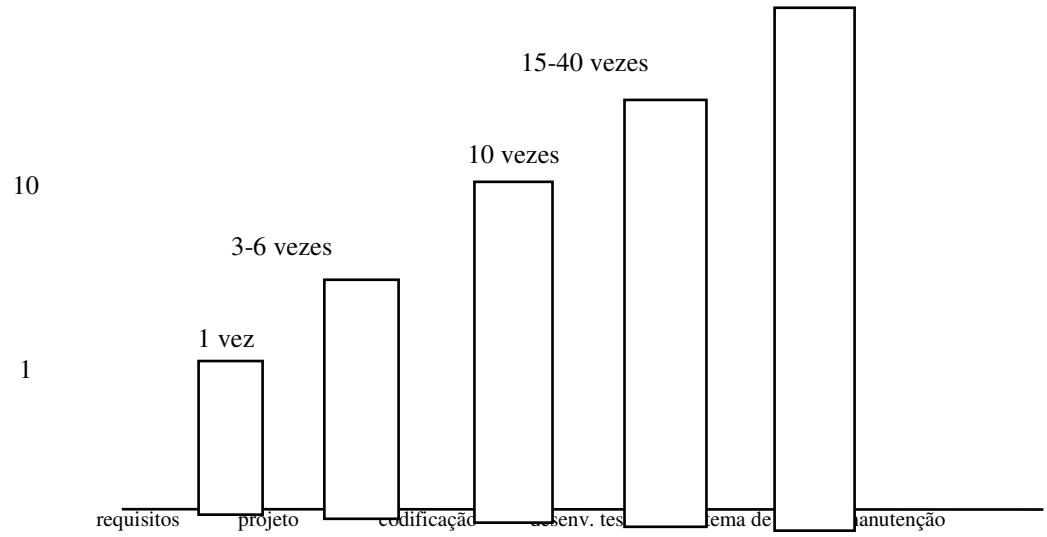


Figura 1

4. Atributos de Qualidade de Software

A qualidade de software não é uma idéia tão simples. É mais fácil descrevê-la através de um conjunto de atributos ou fatores requeridos que variam de acordo com as diferentes aplicações e os clientes que as solicitam.

Existem várias formas de se classificar os fatores de qualidade. Uma delas é classificá-los como fatores externos e fatores internos. Fatores externos são aqueles cuja presença ou falta num produto de software pode ser detectada pelos usuários do produto (velocidade, facilidade de uso). Fatores internos são aqueles que são perceptíveis apenas por profissionais de computação (modularidade). Apesar de apenas os fatores externos terem importância no final, a chave para assegurar que eles são satisfeitos são os fatores internos, ou seja, as técnicas internas são um meio para atingir qualidade de software externa. Alguns atributos externos são: corretude, robustez, extensibilidade, reusabilidade, compatibilidade, eficiência, portabilidade, verificabilidade, integridade e facilidade de uso [Meyer88].

Outra maneira de se classificar os atributos de qualidade é dividí-los em atributos funcionais e atributos não funcionais. Os atributos funcionais tipicamente se aplicam a pedaços do software, módulos do sistema como um todo e estão mais relacionados com o quê deve ser feito. Já os atributos não funcionais podem se aplicar a qualquer produto do processo de desenvolvimento: especificações, código, manuais, etc., e estão mais relacionados com o quão bem deve ser feito [McDermid94].

4.1. Métricas de Qualidade de Software

Um elemento chave de qualquer processo de engenharia é a medição. Nós usamos medidas para melhor entendermos os atributos dos modelos que criamos e, o mais importante é que nós usamos medidas para avaliarmos a qualidade dos produtos de engenharia ou sistemas que nós construímos.

Ao contrário de outras engenharias, a engenharia de software não é baseada em leis quantitativas básicas, medidas absolutas não são comuns no mundo do

software. Ao invés disso, nós tentamos derivar um conjunto de medidas indiretas que levam a métricas que fornecem uma indicação de qualidade de alguma representação do software.

Embora as métricas para software não sejam absolutas, elas fornecem uma maneira de avaliar qualidade através de um conjunto de regras definidas.

4.1.1. Boehm, Brown e Lipow

Para medir qualidade de software deve-se determinar quais características medir e como medir. Boehm, Brown e Lipow (1977) definem uma árvore de atributos de qualidade de software bem definidos e bem diferenciados (figura 2) [Shn80], onde as direções das setas indicam implicações lógicas. Por exemplo, um programa que é fácil de ser mantido deve também ser facilmente testado, entendido e modificado.

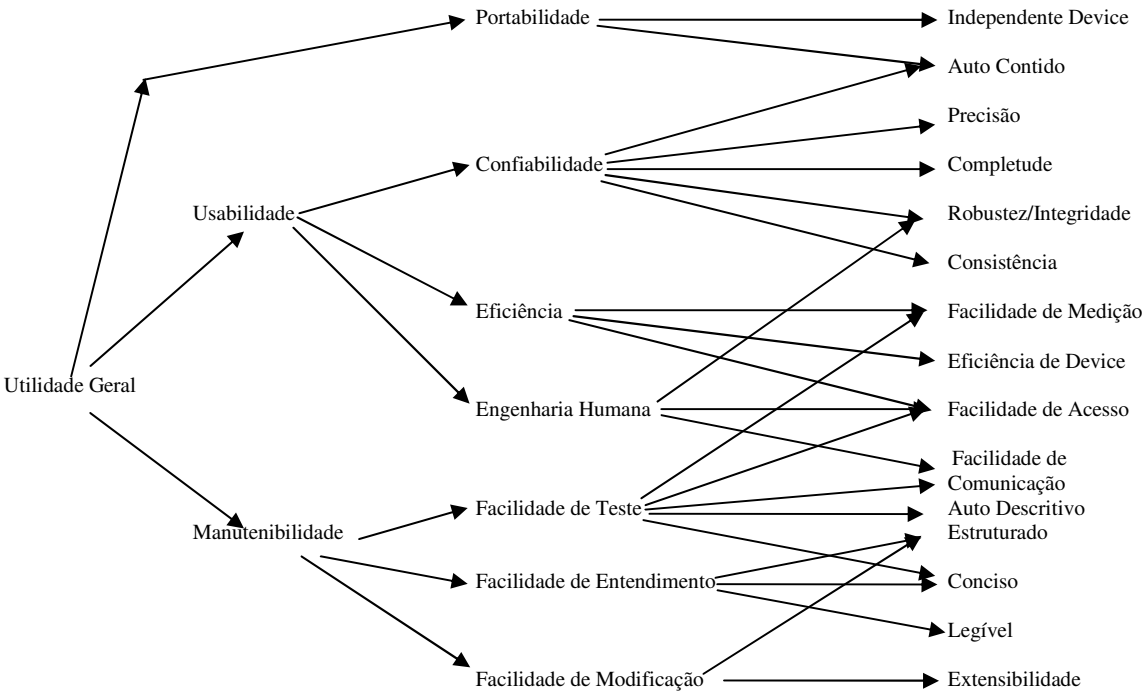


Figura 2

A estrutura de mais alto nível reflete o uso de avaliação da qualidade de software. Boehm, Brown e Lipow enfatizam a aquisição do pacote de software, o qual deve ter as seguintes características de nível médio na estrutura hierárquica: portabilidade, confiabilidade, eficiência, engenharia humana e facilidades de teste, uso e modificação.

As características de mais baixo nível são primitivas que podem ser combinadas para formar características de nível médio. As características primitivas são recomendadas como métricas quantitativas delas próprias e das características de mais alto nível [Shn80].

Uma característica primitiva pode ser definida ou medida através de uma *checklist*. Valores numéricos podem ser dados aos atributos de qualidade, em

alguns casos isso pode ser apropriado, como por exemplo, nos atributos de performance, e em outros pode existir um certo grau de subjetividade.

Tais *checklists* podem ser úteis, mas tendem a crescer e se tornarem incômodas, surgindo a necessidade de uma organização específica e tornando-se específica para uma linguagem/sistema.

4.1.2. Métricas para o Código Fonte (Halstead)

A ciência de software de Halstead (1977) propõe as primeiras leis analíticas para o software de computador. Ela usa um conjunto de medidas primitivas que pode ser derivado depois que o código é gerado, ou estimado assim que o projeto está "completo" [Shn80].

O método de Halstead é baseado na habilidade de se obter as seguintes medidas primitivas num programa:

- *n1* = o número de operadores distintos que aparecem num programa;
- *n2* = o número de operandos distintos que aparecem num programa;
- *N1* = o número total de ocorrências de operadores;
- *N2* = o número total de ocorrências de operandos.

Essas medidas podem ser melhor ilustradas através da seguinte subrotina em FORTRAN:

```
      SUBROUTINE SORT (X, N)
      DIMENSION X(N)
      IF (N .LT. 2) RETURN
      DO 20 I = 2,N
           DO 10 J = 1,I
              IF (X(I) .GE. X(J)) GO TO 10
              SAVE = X(I)
              X(I) = X(J)
              X(J) = SAVE
   10 CONTINUE
   20 CONTINUE
      RETURN
      END
```

Operadores		Contagem	Operandos		Contagem
1	fim de instrução	7	1	X	6
2	indexador de vetor	6	2	I	5
3	=	5	3	J	4
4	IF ()	2	4	N	2
5	DO	2	5	2	2
6	,	2	6	SAVE	2
7	fim de programa	1	7	1	1
8	.LT.	1			

9 .GE. 10 GO TO 10	1 1		
n1 = 10	*N1* = 28	*n2* = 7	*N2* = 22

Halstead usa essas medidas primitivas para desenvolver várias expressões, dentre elas:

- o comprimento global do programa ($N = n1\ log2\ n1 + n2\ log2\ n2$);
- o volume real ($V = N\ log2\ n$): representa o número de bits exigido para especificar um programa, o que o faz independente do conjunto de caracteres da linguagem usada para expressar o algoritmo, mas muda quando um algoritmo é traduzido de uma linguagem para outra;
- o volume mínimo potencial ($V' = (2 + n2')\ log2\ (2 + n2')$, onde $n2'$ é o valor mínimo do número de operandos únicos): não muda quando um algoritmo é traduzido de uma linguagem para outra;
- nível de programa ($L = V'/V$): foi desenvolvido para comparar implementações de um algoritmo em linguagens diferentes. Halstead valida esta métrica comparando valores observados e valores medidos, obtendo um coeficiente de correlação de 0.90;
- esforço de programação ($E = V/L$);
- nível de linguagem ($X = LV'$): implica em um nível de abstração na especificação do procedimento; uma linguagem de alto nível permite a especificação de código num nível de abstração mais elevado do que a linguagem Assembler.

As aplicações da teoria de Halstead incluem várias estimativas, as quais podem ser de tempo de programação, de tamanho de programa, de número de erros a ser esperado de um dado programa, entre outras.

A IBM demonstrou uma outra possível aplicação da teoria de Halstead que foi aplicar os relacionamentos de software a circuitos de hardware, os quais eram expressos como um programa de computador.

Uma grande extensão de pesquisa foi realizada para investigar a ciência de software e pode-se dizer que uma boa concordância foi encontrada entre os resultados analiticamente previstos e os experimentais.

4.1.3. Métricas para Qualidade de Especificação (Davis)

Davis [Dav93] e alguns colegas propõem uma lista de características que podem ser usadas para avaliar a qualidade do modelo de análise e da correspondente especificação de requisitos: falta de ambigüidade, completude, corretude, facilidade de entendimento, verificabilidade, consistência interna e externa, concisão, facilidade de rastreamento, facilidade de modificação, precisão e reusabilidade.

Embora muitas das características acima pareçam qualitativas, cada uma pode ser representada usando uma ou mais métricas, por exemplo, assumindo que

existem *nr* requisitos numa especificação, tal que *nr = nf + nnf*, onde *nf* é o número de requisitos funcionais e *nnf* é o número de requisitos não funcionais.

Para determinar a falta de ambigüidade dos requisitos, Davis e seus colegas sugerem uma métrica que é baseada na consistência da interpretação dos revisores de cada requisito: *Q1 = nui/nr*, onde *nui* é o número de requisitos para os quais todos os revisores tiveram a mesma interpretação. Quanto mais perto de 1 esteja o valor de *Q1*, menos ambígua é a especificação.

A completude dos requisitos funcionais pode ser computada por *Q2 = nu/(ni x ns)*, onde *nu* é o número de requisitos de função únicos, *ni* é o número de entradas definidas ou implicadas pela especificação e *ns* é o número de estados especificados. Q2 mede a percentagem de funções necessárias que tenham sido especificadas para um sistema. Para inserirmos os requisitos não funcionais nesta métrica, devemos considerar o grau de validação dos requisitos: *Q3 = nc/(nc + nnv),* onde *nc* é o número de requisitos que foram validados e *nnv* é o número de requisitos que ainda não foram validados.

4.1.4. Métricas para Sistemas Orientados a Objetos

Software orientado a objetos (OO) é fundamentalmente diferente do software desenvolvido usando métodos convencionais. Por esta razão, métricas usadas para sistemas OO devem focalizar as características que distinguem software OO de software convencional [Pressman97].

Berard [Ber95] define cinco características que tratam métricas especializadas: localização, encapsulamento, *information hiding*, herança e técnicas de abstração de objetos.

Localização é uma característica de software que indica a maneira na qual as informações são concentradas dentro de um programa.

Encapsulamento é o empacotamento de uma coleção de itens.

Information hiding esconde os detalhes operacionais de um componente de programa.

Herança é um mecanismo que capacita a responsabilidade de um objeto ser propagada para outros objetos.

Abstração é um mecanismo que capacita o projetista a focar nos detalhes essenciais de um componente de programa sem se preocupar com detalhes de baixo nível.

♦ **Métricas Orientadas a Classes**

A classe é a unidade fundamental de um sistema OO. Classes freqüentemente são superclasses de outras classes, as quais herdam seus atributos e suas operações. As classes também colaboram com outras classes. Cada uma dessas características pode ser usada como base para uma métrica.

⇒ **Métricas CK**

Chidamber e Kemerer [Chi94] propõem seis métricas de projeto baseadas em classes para sistemas OO:

- Pesos dos Métodos por Classe (WMC): assume que *n* métodos de complexidade C_1, C_2, ..., C_n são definidos para uma classe **C**. A específica métrica de complexidade que é escolhida deve ser normalizada, tal que a complexidade nominal para um método resulte no valor 1.0.

$WMC = \Sigma C_i$, para i = 1 a n.

O número de métodos (e suas complexidades) é um indicador razoável para implementar e testar uma classe. Se o número de métodos para uma dada classe e a árvore de herança crescem, ela se torna mais e mais específica, limitando seu potencial de reuso. Por todas essas razões, WMC deve ser conservado tão baixo quanto possível.

- Profundidade da Árvore de Herança (DIT): Esta métrica é definida como o tamanho máximo do nó à raiz da árvore. Na figura 3, o valor de DIT para a hierarquia de classe é 4.

Com o crescimento do DIT, fica claro que classes de mais baixo nível irão herdar muitos métodos. Um problema com um valor alto para DIT é que existe uma potencial dificuldade de determinar o comportamento das classes de níveis mais baixos, por outro lado, isso implica que muito métodos podem ser reusados.

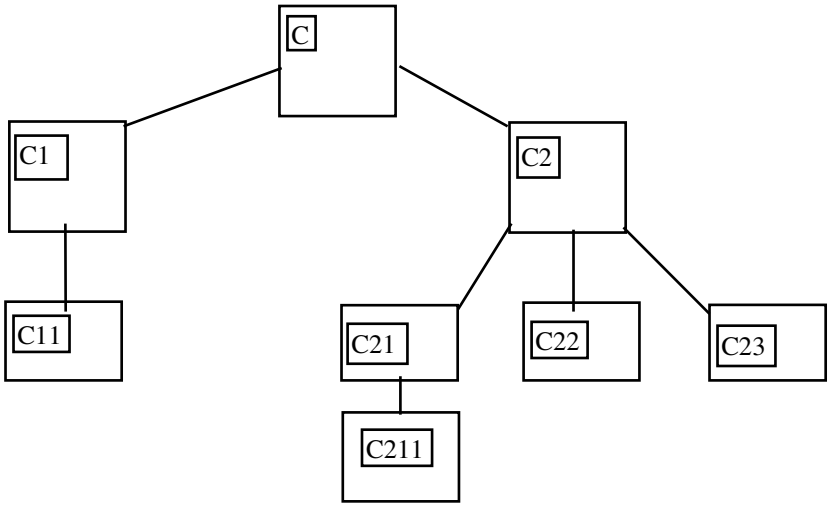


Figura 3

- Número de Filhos (NOC): as classes que são imediatamente subordinadas a uma classe numa hierarquia de classes são ditas suas filhas. Na figura 3, a classe C2 tem três filhas C_{21}, C_{22} e C_{23}.

Com o crescimento dos filhos, o reuso aumenta, mas as abstrações representadas pelas superclasses podem ser diluídas. Isto é, existe a possibilidade de que alguns dos filhos não sejam membros apropriados de seus pais. Com o aumento dos NOC's a quantidade de testes também será aumentada.

- Acoplamento entre Objetos (CBO): O modelo *class-responsability-collaborator* (CRC) [Wir90] pode ser usado para determinar o valor para CBO. Em resumo, CBO é o número de colaborações listadas para uma classe sobre seu cartão de índice (CRC) (figura 4).

O CRC fornece uma maneira simples de identificar e organizar classes que são relevantes para os requisitos do sistema ou produto.

Um modelo CRC é uma coleção de cartões de índice padrão que representam classes. Os cartões são divididos em três grupos. No topo do cartão

escreve-se o nome da classe. No corpo do cartão lista-se as responsabilidades da classe no lado esquerdo, e os colaboradores no lado direito (figura 4).

Colaborações identificam relacionamentos entre classes. Dizemos que um objeto colabora com outro objeto se, para preencher uma responsabilidade, ele necessita enviar para outro objeto qualquer mensagem.

Nome da classe	
Responsabilidades	Colaboradores

Figura 4

Com o aumento do CBO a reusabilidade da classe irá diminuir. Valores altos para CBO também complicam modificações e testes. Em geral, os valores de CBO para cada classe deve ser o mais baixo possível. Esta idéia é consistente com a linha geral usada para reduzir o acoplamento em softwares convencionais.

- Resposta para uma Classe (RFC): o conjunto de respostas para uma classe é o conjunto de métodos que potencialmente podem ser executados para responder uma mensagem recebida por um objeto daquela classe. RFC é definido como o número de métodos em um conjunto de respostas.

Com o aumento do RFC, o esforço dispendido para testes também aumenta, porque a seqüência de testes aumenta, aumentando também a complexidade em projetar a classe.

- Falta de Coesão em Métodos (LCOM): cada método em uma classe, **C**, acessa um ou mais atributos (também chamados variáveis de instância). LCOM é o número de métodos que acessam um ou mais desses atributos. Se nenhum método acessa o mesmo atributo, então LCOM = 0. Se, por exemplo, tivermos uma classe com 7 métodos e três deles têm um ou mais atributos em comum, isso implica que LCOM = 3.

Um LCOM alto implica no aumento da complexidade do projeto da classe. Em geral, valores altos para LCOM implicam que a classe poderia ser dividida em duas ou mais classes. É desejável ter uma alta coesão, para tanto, deve-se manter o LCOM baixo.

$\Rightarrow$ Métricas Propostas por Lorenz e Kidd

Lorenz e Kidd [Lor94] dividem métricas baseadas em classes em quatro categorias: orientadas ao tamanho, baseadas na herança, internas e externas. Métricas orientadas a tamanho em classes OO enfocam a contagem dos atributos e operações para uma classe individual e os valores médios para sistemas OO como um todo. Métricas baseadas em herança enfocam a maneira na qual operações são reusadas através da hierarquia de classes. Métricas internas verificam a coesão e as características do código, e métricas externas examinam o acoplamento e o reuso.

- Tamanho da Classe (CS): o tamanho total de uma classe pode ser determinado usando as medidas do número total de operações (tanto operações de instância privadas como operações de instância herdadas) que são encapsuladas dentro da classe e do número de atributos (tanto atributos de instância privados como atributos de instância herdados) que são encapsulados por uma classe.

Como pode-se notar, grandes valores para CS indicam que a classe pode ter muita responsabilidade, o que pode reduzir a reusabilidade da classe e tornar difícil a implementação e os testes. Em geral, operações e atributos públicos ou herdados deveriam ter um peso maior na determinação do tamanho da classe. Operações e atributos privados tornam possível a especialização e são mais localizados dentro do projeto.

- Médias para o número das operações e dos atributos de uma classe também podem ser computadas. Quanto menor esse número médio, mais as classes podem ser reusadas.

- Número de Operações Sobrescritas (*Overridden*) por uma Subclasse (NOO): quando uma subclasse redefine uma operação herdada isso é caracterizado um o*verriding*. Valores grandes para NOO geralmente indicam um problema de projeto.

Se NOO é grande, o projetista violou a abstração da superclasse, se isso acontecer em alguns pontos do projeto, problemas de teste e manutenção podem surgir.

- Número de Operações Adicionadas a uma Classe (NOA): Subclasses são especializadas pela adição de atributos e operações privadas. Quanto maior o número NOA, mais específica se torna a classe, ficando longe do nível de abstração proposto pelas suas superclasses. Em geral, quanto maior o DIT menor os valores para NOA nas classes de nível mais baixo da hierarquia.

- Índice de Especialização (SI): o índice de especialização fornece uma indicação do grau de especialização para cada subclasse em um sistema OO. A especialização pode ser alcançada adicionando/retirando operações ou por o*verriding*.

$SI = [NOO \times Nível] / M_{total;}$

onde Nível é o nível na hierarquia de classes na qual a classe reside e M_{total} é o número total de métodos da classe. Valores altos para SI indicam que na hierarquia de classes existe uma (ou algumas) classe que não está de acordo com a abstração da superclasse.

♦ Métricas para Testes em OO

As técnicas descritas até agora fornecem uma indicação da qualidade do projeto. Elas também fornecem uma indicação geral da quantidade de esforço de testes requerido para sistemas OO. Métricas para teste são organizadas em categorias que refletem importantes características de projeto:

- Encapsulamento:

Falta de Coesão nos Métodos (LCOM): valores altos para LCOM implicam que mais estados deveriam ser testados para garantir que os métodos não geram efeitos colaterais.

Percentagem Pública e Protegida (PAP): atributos públicos são herdados de outras classes e, por isso são visíveis para essas classes. Atributos

protegidos são uma especialização e são privados para uma específica subclasse. Esta métrica indica a percentagem de atributos de classe que são públicos. Altos valores para PAP aumentam a possibilidade de efeitos colaterais entre classes. Testes devem ser projetados para garantir que tais problemas não venham a ocorrer.

Acesso Público para Dados (PAD): esta métrica indica o número de classes (ou métodos) que podem acessar outro atributo de uma outra classe (uma violação de encapsulamento). Valores altos para PAD aumentam a possibilidade de efeitos colaterais entre classes. Testes devem ser projetados para garantir que tais problemas não venham a ocorrer.

- Herança:

Número de Classes Raiz (NOR): esta métrica é um contador do número de hierarquias distintas de classes que são descritas no projeto do modelo. Quando NOR é aumentado, o esforço para o teste também é.

Fan in (FIN): Quando usado no contexto OO, *fan-in* é uma indicação de herança múltipla. FIN > 1 indica que uma classe herda seus atributos e operações de mais de uma classe raiz. FIN > 1 deveria ser evitado quando possível.

Número de Filhos (NOC) e Profundidade na Árvore de Herança (DIT): métodos da superclasse devem ser retestados para subclasses.

♦ **Métricas para Projeto OO**

Como sabemos, o trabalho de um gerente de projeto é planejar, coordenar, verificar a evolução e controlar o projeto de software. Um dos pontos chave para um gerente de projeto durante a fase de planejamento é uma estimativa do tamanho do software. Tamanho é diretamente proporcional ao esforço e duração. As seguintes métricas OO [Lor94] podem fornecer uma idéia do tamanho do software:

- Número de Scripts de Cenário (NSS): o número de scripts de cenário, ou casos de uso, é diretamente proporcional ao número de classes requeridas para encontrar os requisitos, o número de estados para cada classe e o número de métodos, atributos e colaborações. O NSS é um forte indicador do tamanho do programa.

- Número de Classes Chaves (NKC): Uma classe chave enfoca diretamente o domínio do negócio para um determinado problema e terá uma probabilidade baixa de ser implementada via reuso. Por esta razão, valores altos para NKC indicam um substancial trabalho de desenvolvimento. Lorenz e Kidd [Lor94] sugerem que entre 20% e 40% de todas as classes em um típico sistema OO são classes chaves.

- Número de Subsistemas (NSUB): o número de subsistemas fornecem um entendimento da alocação de recursos, do cronograma e do esforço total de integração.

4.2. Motivação para o uso de Métricas de Qualidade

Independentemente da métrica usada, elas buscam sempre os mesmos objetivos:

- melhorar o entendimento da qualidade do produto;
- atestar a efetividade do processo;
- melhorar a qualidade do trabalho realizado a nível de projeto.

5. Garantia de Qualidade de Software (SQA)

Um dos desafios críticos para qualquer programa de qualidade é tornar possível que qualquer pessoa possa fazer revisões no trabalho de pessoas experientes. Gerentes querem os melhores projetistas para projetar o produto, então, em geral, SQA não pode tê-los. A necessidade é concentrar esforços em métodos de SQA que permitam que o desenvolvimento possa ser revisado por pessoas que não são desenvolvedores. O papel de SQA é monitorar os métodos e os padrões que os engenheiros de software usam e verificar se eles estão usando apropriadamente seus conhecimentos. Pessoas podem ser experientes em SQA sem, no entanto, serem experientes em projeto de software.

5.1. Atividades de Garantia de Qualidade de Software

Em SQA temos uma variedade de tarefas, as quais podemos dividir em dois grandes grupos: os engenheiros de software que fazem o trabalho técnico e o grupo de SQA que tem responsabilidades no plano de qualidade, na inspeção, na conservação de registros históricos, na análise e no *reporting* das atividades de SQA para o gerente do projeto.

Os engenheiros de software buscam qualidade de software aplicando sólidos métodos e medidas, conduzindo revisões técnicas formais e realizando testes de software bem planejados.

O papel do grupo de SQA é ajudar o time de engenharia de software a alcançar um produto de alta qualidade.

O Instituto de Engenharia de Software (SEI) recomenda as seguintes atividades a serem realizadas por um grupo de SQA:

- Preparar um plano de SQA (ver seção 6.3) para o projeto. O plano é desenvolvido durante o planejamento do projeto e é revisado por todas as partes interessadas.
- Participar do desenvolvimento da descrição do processo do projeto do software. O time de engenharia de software seleciona um processo para o trabalho a ser realizado. O grupo de SQA revisa a descrição do processo, verificando se o mesmo está de acordo com a política organizacional, aos padrões internos para o software, aos padrões impostos externamente e a outras partes do plano de projeto de software.
- Revisar as atividades dos engenheiros de software para verificar se o processo de software definido está sendo seguido. O grupo de SQA identifica, documenta e trilha os desvios do processo e verifica quais correções devem ser realizadas.
- Garantir que os desvios no trabalho do software e nos produtos do trabalho são documentados e mantidos de acordo com o procedimento documentado.
- Registrar qualquer discordância e reportar para o gerente.

Além dessas atividades, o grupo de SQA coordena o controle e o gerenciamento de mudanças e ajuda a coletar e analisar métricas de software.

5.2. Medidas de Produtividade de Programação

Zak (1977) lista os seguintes cinco atributos de produtividade: corretude, habilidade para cumprir cronogramas, adaptabilidade, eficiência e, por fim ausência de erros. Ele admite que existem controvérsias em relação a quais variáveis deveriam ser medidas e que efeito elas podem ter na produtividade. Zak dá a seguinte métrica para taxa de programação:

R = L/(ST);

onde R é a taxa de programação em linhas de código fonte por pessoas-mês; L é o número de linhas de código fonte do produto final; S é o nível do *staffing* e T é o cronograma (em meses). Existe um certo consenso em relação aos critérios mais importantes de produtividade: qualidade da documentação externa, linguagem de programação, disponibilidade de ferramentas, experiência do programador no processamento dos dados, experiência do programador na área funcional, comunicação no projeto, módulos independentes e práticas de programação bem definidas.

5.3. Revisões de Software

Revisões de software são um filtro no processo de engenharia de software. Isto é, revisões são aplicadas a vários pontos durante o desenvolvimento de software e serve para descobrir erros que podem ser removidos. Revisões não são limitadas a especificação, projeto e código. Documentos, tais como, plano de teste, procedimentos de gerenciamento de configuração, padrões e normas de usuário deveriam também ser revisados [Sommerville92].

Existem vários tipos de revisão:

- Inspeções no projeto e no código: têm a finalidade de detectar erros no projeto ou código e checar quando padrões têm sido seguidos.
- Revisões gerenciais: esse tipo de revisão é feita para fornecer informações aos gerentes sobre todo o processo no desenvolvimento do projeto de software.
- Revisões de qualidade: o trabalho de um indivíduo ou de um time é revisado por um grupo composto por membros do projeto e gerentes técnicos.

O dicionário de padrões do IEEE define defeito como uma “anomalia do produto”. Um defeito implica em um problema na qualidade que é descoberto depois que o software foi entregue ao usuário final.

O objetivo primário de revisões técnicas formais é encontrar erros durante o processo antes que eles se tornem defeitos após o *release* do software. Um benefício óbvio de revisões técnicas formais é a descoberta de erros antes que eles se propaguem para as próximas fases do processo de software.

Estudos de algumas indústrias indicam que as atividades de projeto introduzem entre 50% e 60% de todos os erros durante o processo de software. Entretanto, técnicas de revisões formais têm atingido um percentual de 75% na descoberta desses erros, reduzindo o custo dos passos subsequentes.

5.4. Confiabilidade

Não há dúvidas de que a confiabilidade de um programa de computador é um elemento importante para sua qualidade. Se um programa falha freqüentemente e repetidamente, não importa muito se outros fatores de qualidade são aceitáveis.

Confiabilidade de software pode ser medida, direcionada e estimada usando dados históricos e de desenvolvimento. Ela é definida em termos estatísticos como a "probabilidade de uma operação de programa de computador ser livre de falha". Para ilustrar isto, digamos que um programa X é estimado ter uma confiabilidade de 0.96 em oito horas de processamento, ou seja, se o programa X fosse executado 100 vezes em oito horas de processamento (tempo de execução), é provável que ele opere corretamente (sem falhas) em 96 das 100 vezes [Pressman97].

Uma questão que surge ao se discutir sobre qualidade é o que significa o termo falha. No contexto de qualquer discussão sobre qualidade e confiabilidade de software, falha é a não conformidade com os requisitos de software. Ainda existem gradações nesta definição. Falhas podem ser apenas incômodas ou catastróficas. Uma falha pode ser corrigida dentro de segundos, enquanto uma outra pode requerer semanas ou até mesmo meses para ser corrigida. Além disso, a correção de uma falha pode resultar na introdução de outros erros que no final resultam em outras falhas.

5.4.1. Medidas de Confiabilidade e Disponibilidade

Os primeiros trabalhos em confiabilidade de software tentaram extrapolar a matemática da teoria de confiabilidade de hardware para a previsão de confiabilidade de software. A maioria dos modelos relacionados à confiabilidade de hardware são estabelecidos em falhas devido ao uso (desgaste), ao invés de falhas devido a defeitos de projeto, porque as falhas de desgaste físico são mais prováveis de acontecer. Infelizmente, o oposto disto é verdade para software.

Ainda existe debate sobre os relacionamentos entre os conceitos chave de confiabilidade de hardware e sua aplicabilidade a software. Embora exista uma ligação, é importante considerar poucos conceitos simples que se aplicam a ambos.

Considerando um sistema baseado em computador, uma simples medida de confiabilidade é o tempo médio entre falhas (MTBF), onde:

MTBF = MTTF + MTTR,

MTTF = tempo médio para a falha e MTTR = tempo médio para o reparo. Muitos pesquisadores argumentam que MTBF é uma medida mais útil do que defeitos/KLOC, pois um usuário final preocupa-se com falhas, e não com o número total de defeitos. O número total de defeitos fornece pouca indicação da confiabilidade de um sistema porque cada defeito dentro de um programa não tem a mesma taxa de falha. Muitos defeitos num programa podem permanecer sem ser detectados por décadas (MTBF longo) e, se eles são removidos, o impacto na qualidade de software é imperceptível [Pressman97].

Além da medida de confiabilidade, deve-se desenvolver uma medidade de disponibilidade, a qual é a probabilidade de que um programa esteja operando de acordo com os requisitos num dado ponto no tempo, e é definida como:

Disponibilidade = MTTF/(MTTF + MTTR) x 100%.

A medida de disponibilidade é mais sensível ao MTTR, uma medida indireta da manutenibilidade de software.

5.4.2. Segurança X Confiabilidade

Na seção anterior, foi discutido o papel da análise da confiabilidade de software. Não obstante, a confiabilidade e a segurança de software estejam estreitamente relacionadas, é importante compreender a sutil diferença entre elas. A confiabilidade de software usa a análise estatística para determinar a probabilidade de que uma falha de software venha a ocorrer. Entretanto, a ocorrência de uma falha não necessariamente resulta num risco ou deformação. A segurança de software examina as maneiras segundo as quais as falhas resultam em condições que podem levar a uma deformação. Ou seja, as falhas não são consideradas no vazio, mas são avaliadas no contexto de um sistema inteiro computadorizado.

Segurança de software é um atividade de garantia de qualidade de software que se concentra na identificação e avaliação de casualidades em potencial que possam exercer um impacto negativo sobre o software e fazer com que todo o sistema falhe.

Logo que as casualidades a nível de sistema são identificadas, técnicas de análise são usadas para definir a gravidade e a probabilidade de ocorrência, tais como, análise em árvore, lógica de tempo real ou modelos de rede de Petri. Após as casualidades serem identificadas e analisadas, os requisitos relacionados à segurança podem ser especificados para o software.

6. Controle de Qualidade

6.1. Como fazer Controle de Qualidade?

Como vimos na seção 5.1, é possível observar que o controle de qualidade é algo que consome tempo no desenvolvimento de sistemas de software, e, é claro, vai além da entrega do sistema e entra na fase de manutenção. Poderíamos generalizar dizendo que deveríamos identificar controle de qualidade sobre todos os produtos em cada estágio. Toda atividade deve culminar em uma atividade de controle de qualidade desejada (ver seção 7).

Desde que desejamos ser capazes de definir atividades de controle de qualidade para toda atividade durante o desenvolvimento, faz sentido verificar como as técnicas usadas para cada atividade podem contribuir para o controle de qualidade delas. Algumas técnicas são boas, têm controle de qualidade embutido, outras não têm, e deve-se, então, aplicar ações de controle de qualidade gerais para fazer o controle eficientemente.

6.2. Plano de SQA

Cada projeto de desenvolvimento e manutenção deveria ter um Plano de Controle de Qualidade (SQAP) que especifica seus objetivos, as tarefas de SQA a serem realizadas, os padrões contra os quais o trabalho de desenvolvimento é para ser medido e os procedimentos e a estrutura organizacional. O Plano de SQA constitui um mapa rodoviário para a instituição da garantia de qualidade de software.

O padrão IEEE (Padrão ANSI/IEEE 730-1984 e 983-1986) para a preparação do SQAP contém os seguintes tópicos [Humphrey89]:

I. Propósito do plano
II. Documentos de referência

III. Administração
    A. Organização
    B. Tarefas
    C. Responsabilidades
IV. Documentação
    A. Propósito
    B. Documentos de engenharia de software exigidos
    C. Outros documentos
V. Padrões, práticas e convenções
    A. Propósito
    B. Convenções
VI. Revisões auditoriais
    A. Propósito
    B. Requisitos de revisão
        1. Revisão dos requisitos de software
        2. Revisões de projeto
        3. Verificação de software e revisões de validação
        4. Auditoria funcional
        5. Auditoria física
        6. Auditorias *in-process*
        7. Revisões administrativas
VII. Gerenciamento de configuração de software
VIII. Reportagem de problemas e ações corretivas
IX. Ferramentas, técnicas e metodologias
X. Controle de código
XI. Controle de mídia
XII. Controle de fornecedores
XIII. Coleta, manutenção e retenção de registros

A seção de documentação deveria descrever a documentação a ser produzida e como é para ela ser revisada. Os documentos de engenharia de software exigidos podem ser: Especificação de Requisitos, Descrição de Projeto, Plano de Verificação e Validação, Relatório de Verificação e Validação e Documentação do Usuário. O Plano de Verificação e Validação é uma descrição dos métodos usados para verificar se os requisitos são implementados no projeto, se o projeto é implementado no código e se o código atinge os requisitos. Outros documentos podem ser: Plano de Desenvolvimento de Software, Plano de Gerência de Configuração de Software, Manual de Padrões e Procedimentos.

A seção de padrões, práticas e convenções especifica um conteúdo mínimo de:

- padrões de documentação;
- padrões de estrutura lógica;
- padrões de codificação;
- padrões de comentários.

A auditoria funcional é uma auditoria feita antes da entrega do sistema para verificar se os requisitos foram encontrados. A auditoria física também é feita antes da entrega do sistema para verificar se o software e a documentação estão consistentes

com o projeto e prontos para serem entregues. A auditoria *in-process* é uma amostra estatística do processo de desenvolvimento que é feita para verificar a consistência do o código versus as especificações de projeto e interface, do projeto versus os requisitos e os planos de testes versus os requisitos. As revisões administrativas são revisões independentes, conduzidas para a verificação da execução do plano de qualidade.

Muitos dos padrões são requeridos para definir apropriadamente a operação de uma organização de software.

6.3. Pessoas de SQA

Alocar pessoas para trabalhar com SQA é uma tarefa difícil dos gerentes de software. A prática de iniciar novas contratações em SQA é uma solução parcial que pode ser efetiva apenas se existem pessoas experientes no mercado. Recrutar pessoas para trabalhar em SQA é difícil também porque os profissionais de software geralmente preferem atribuições de desenvolvimento e a gerência certamente quer atribuir aos melhores projetistas o trabalho de projeto. O esquema de rotatividade também pode ser efetivo, mas infelizmente, o desenvolvimento de software geralmente é adepto a transferir pessoas para trabalhar em SQA e não "pegá-las" de volta.

Uma solução efetiva é requerer que todos os novos gerentes de desenvolvimento sejam promovidos para trabalharem em SQA. Isso poderia significar que potenciais gerentes poderiam passar entre seis meses e um ano em SQA, antes de serem promovidos à gerência. Essa é uma medida extrema, mas pode ser efetiva [Humphrey89].

Para o trabalho de SQA ser efetivo, deve haver bons profissionais na equipe e um completo apoio da gerência, no sentido de investimento mesmo.

7. Sistemas de Gerenciamento de Qualidade

7.1. Personal Software Process (PSP)

O estímulo original para desenvolver o PSP surgiu de questões sobre o Capability Maturity Model (CMM) do Software Engineering Institute (SEI). Muitos viam o CMM como projetado para grandes organizações e não entendiam como ele poderia ser aplicado a trabalhos individuais e em times pequenos de projeto. Apesar do CMM poder ser aplicado para ambas, pequenas e grandes organizações, uma orientação mais específica se tornava claramente necessária. Após alguns anos de pesquisa, 12 das 18 áreas-chave de processo do CMM (ver seção 7.2) foram adaptadas para o trabalho de engenheiros de software individuais.

O principal objetivo do PSP é fazer com que os engenheiros de software fiquem atentos no processo que eles usam para fazer seus trabalhos e estejam sempre verificando suas performances no processo. Engenheiros de software são treinados individualmente para um conjunto de objetivos pessoais, definindo os métodos a serem usados, medindo seus trabalhos, analisando os resultados, e ajustando os métodos para atingir seus objetivos. O PSP é uma estratégia para o desenvolvimento pessoal com o objetivo de aumento de produtividade.

Trabalhos experimentais foram iniciados em algumas corporações para verificar como engenheiros experientes poderiam reagir ao PSP e explorar a introdução de seus métodos. Foi verificado que os engenheiros de software geralmente são atraídos

pela estratégia do PSP e encontram métodos que ajudam em seus trabalhos. Nas palavras de um engenheiro, "Isto não é para a empresa, é para mim". O leitor interessado pode obter informações adicionais em [SEI] e [Humphrey].

O PSP aplica princípios de processo para o trabalho do engenheiro de software por:

- Fornecer um padrão de processo pessoal definido.
- Introduzir uma família de medidas de processo.
- Usar essas medidas para trilhar e avaliar a performance.
- Se esforçar para obter critérios de qualidade e melhorar os objetivos.

Usando o PSP, engenheiros:

- Desenvolvem um plano para todo projeto.
- Registram seu tempo de desenvolvimento.
- Trilham seus defeitos.
- Mantêm dados de um projeto em relatórios resumidos.
- Usam esses dados para planos de projetos futuros.
- Analisam dados que envolvem seus processos a fim de aumentar suas performances.

7.2. Capability Maturity Model (CMM)

O Modelo de Maturidade de Capacidade para Software (CMM) foi desenvolvido pelo Instituto de Engenharia de Software (SEI). Ele descreve os princípios e práticas relacionados à maturidade do processo de software, e seu objetivo é ajudar as organizações a melhorarem seus processos de software em termos de um caminho evolutivo que vai de *ad hoc* e processos caóticos a processos de software maduros e disciplinados.

O CMM é organizado em cinco níveis de maturidade. Um nível de maturidade é uma base evolucionária bem definida direcionada a obter um processo de software maduro. Cada nível de maturidade fornece uma camada como base para um processo de melhora contínuo.

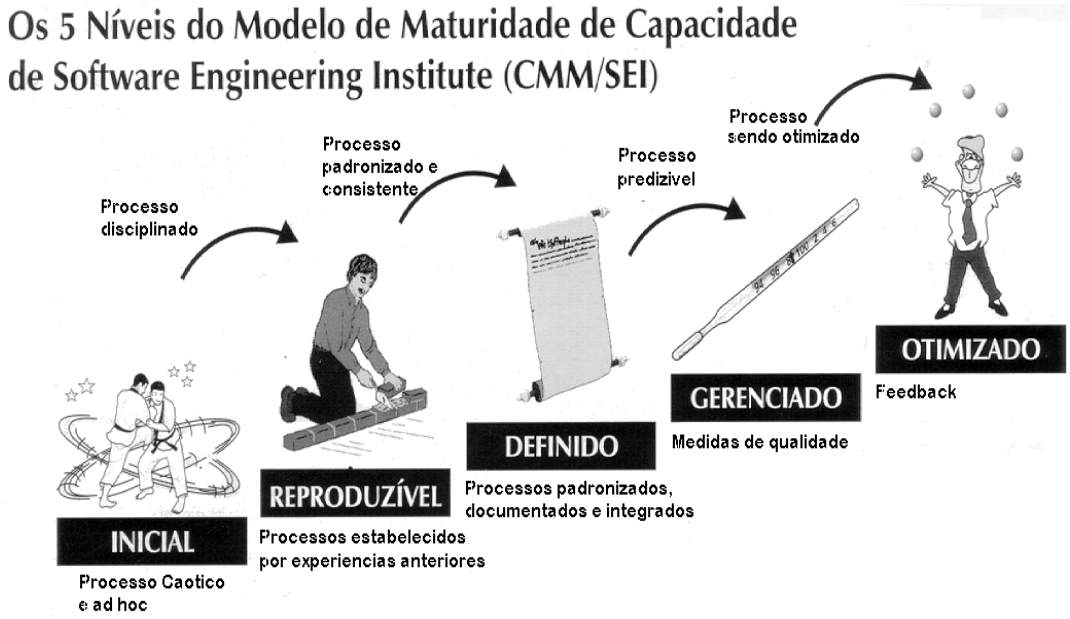
7.2.1. Os Cinco Níveis de Maturidade

As seguintes caracterizações dos cinco níveis de maturidade destacam as mudanças do processo primário feitas em cada nível [Paulk94]:

1. Inicial: O processo de software é caracterizado como *ad hoc* e, ocasionalmente caótico mesmo. Poucos processos são definidos e o sucesso depende de esforços individuais e heróicos.
2. Reproduzível: Os processos de administração do projeto básico são estabelecidos para trilhar custo, cronograma e funcionalidade. A disciplina de processo necessária é estabelecida para se repetir sucessos anteriores em projetos com aplicações similares.
3. Definido: O processo de software para as atividades de administração e engenharia é documentado, padronizado e integrado num processo de software padrão para a organização. Todos os projetos usam uma versão aprovada e feita sob

medida do processo de software padrão da organização para desenvolvimento e manutenção de software.

4. Gerenciado: Medidas detalhadas do processo de software e da qualidade do produto são coletadas. Ambos, o processo de software e o produto, são entendidos e controlados quantitativamente.
5. Otimizado: Um processo de melhora contínuo é capacitado por retorno quantitativo do processo e das idéias e tecnologias inovadoras.



7.2.2. Áreas-Chave de Processo

Exceto para o nível 1, cada nível de maturidade é decomposto em várias áreas-chave de processo que indicam as áreas que uma organização deveria enfocar para melhorar seu processo de software. Cada área-chave de processo (KPA) identifica um grupo de atividades relacionadas que, quando realizadas coletivamente, atingem um conjunto de objetivos considerados importantes para a melhoria da capacidade do processo.

Por definição, não existem áreas-chaves de processo para o nível 1.

As áreas-chave de processo para o nível 2 enfocam os interesses relacionados ao estabelecimento de controle básico de administração de projeto.

As áreas-chave de processo para o nível 3 enfocam os problemas organizacionais e de projeto, como a organização estabelece uma infra-estrutura que institucionaliza uma engenharia de software efetiva e uma administração de processos em todos os projetos.

As áreas-chave de processo para o nível 4 enfocam em estabelecer um entendimento quantitativo de ambos, o processo de software e o produto sendo construído.

As áreas-chave de processo para o nível 5 cobrem os problemas que ambos, organização e projetos, devem endereçar para implementar uma melhora contínua e mensurável do processo de software.

A tabela abaixo mostra todas as áreas-chave de processo para cada nível de maturidade:

Nível de Maturidade	Áreas-Chave de Processo
1. Inicial	Não possui áreas-chave de processo.
2. Reproduzível	Administração dos Requisitos, Planejamento de Projeto de Software, Acompanhamento de Projeto de Software, Gerenciamento de Subcontrato de Software, Garantia de Qualidade de Software e Gerenciamento de Configuração de Software.
3. Definido	Foco no Processo da Organização, Definição do Processo da Organização, Programa de Treinamento, Administração de Software Integrado, Engenharia de Produto de Software e Revisões.
4. Gerenciado	Gerenciamento da Qualidade de Software e Gerenciamento Quantitativo do Processo.
5. Otimizado	Prevenção de Defeito, Gerenciamento de Mudança de Tecnologia e Gerenciamento de Mudança no Processo.

Por conveniência, cada uma dessas áreas-chave de processo é organizada por características comuns, que são atributos que indicam quando a implementação e institucionalização de uma área-chave de processo é efetiva, reproduzível e durável. São elas: Compromisso a Realizar, Habilidade para Realizar, Atividades Realizadas, Medição e Análise e Verificação da Implementação [Paulk94].

Cada área-chave de processo é descrita em termos de práticas chave que contribuem para satisfazer seus objetivos. As práticas chave descrevem a infra-estrutura e as atividades que contribuem para a implementação e institucionalização efetiva da área-chave de processo.

7.3. ISO 9001

Um sistema de controle de qualidade pode ser definido em termos de estrutura organizacional, responsabilidades, procedimentos, processos e recursos para implementar gerenciamento de qualidade. A série de padrões ISO 9000 é um conjunto de documentos que trabalham com sistemas de qualidade que podem ser usados para propostas de garantia de qualidade externa. O ISO 9000 (“Padrões de Gerenciamento e de Garantia de Qualidade - Diretrizes para Seleção e Uso”) descreve elementos de garantia de qualidade em termos genéricos que podem ser aplicados para qualquer empresa de produtos ou serviços oferecidos.

Para uma empresa ser registrada com o certificado ISO 9000, o sistema de qualidade da empresa e as operações são investigadas por três auditores para verificar se o padrão e as operações estão de acordo com as normas estabelecidas. Verificações periódicas são efetuadas para verificar se os padrões estão sendo mantidos.

O modelo de garantia de qualidade ISO 9000 trata uma empresa como uma rede de processos interconectados. Para um sistema de qualidade estar de acordo com a ISO, esses processos devem endereçar as áreas identificadas no padrão e devem ser documentados e praticados como desejado. Documentar um processo ajuda uma organização a entender, controlar e melhorar o mesmo.

O ISO 9000 descreve os elementos de sistemas de garantia de qualidade em termos gerais. Esses elementos incluem a estrutura organizacional, procedimentos, processos e recursos necessários para implementar o plano de qualidade, o controle de qualidade, a garantia de qualidade e o melhoramento da qualidade. Entretanto, o ISO 9000 não descreve como uma organização poderia implementar esses elementos de sistemas de qualidade.

O ISO 9001 ("Sistemas de Qualidade - Modelo para Garantia de Qualidade em Projeto/Desenvolvimento, Produção, Instalação e Serviço") é o padrão de garantia de qualidade que é aplicado para engenharia de software. O padrão contém 20 requerimentos que devem estar presentes para um sistema de garantia de qualidade efetivo. Como o padrão ISO 9001 é aplicado a todas as disciplinas de engenharia, um conjunto especial de guia ISO (ISO 9000-3) tem sido desenvolvido para ajudar a interpretar o padrão para uso no processo de software.

O ISO 9001 define os 20 maiores requerimentos sobre um sistema gerenciador de qualidade [McDermid94]:

1. Gerência de responsabilidades. A organização deve definir e documentar políticas de gerenciamento e objetivos que se comprometem com a qualidade e devem garantir que essas políticas são entendidas, implementadas e mantidas em todos os níveis da organização.
2. Um sistema de qualidade documentado. Esse sistema deve cobrir o controle de todas as atividades em desenvolvimento e ter a documentação de todos os procedimentos.
3. Revisões de contrato. Isso é incluído para garantir que um contrato inicie com os requerimentos do sistema conhecidos por ambas as partes e que o desenvolvedor seja capaz de entregar para o comprador o que foi estabelecido.
4. Controle de projeto. O padrão requer que o desenvolvimento tenha, e use, procedimentos para controlar e verificar a qualidade do projeto do sistema para garantir que ele encontre seus requerimentos.
5. Documentação e controle de mudanças. Esta é uma área especialmente importante para o desenvolvimento de software, onde muito do que é desenvolvido toma a forma de documentos e dados: especificação, projeto, código, dados de teste, etc.
6. Compra/aquisição. Verificar a qualidade, de alguma forma, se desejar incorporar alguma coisa no sistema.

7. Produtos fornecidos pelo comprador. Esta seção de padrão requer procedimentos para verificação, armazenamento e manutenção dos itens comprados/fornecidos.
8. Identificação do produto e *traceability*. Este tem sido um importante procedimento para desenvolvedores de software os quais, como outros engenheiros, constróem suas aplicações a partir de muitos componentes pequenos.
9. Controle do processo. Isto é um requerimento geral em que o processo de produção ele mesmo deve ser planejado e monitorado.
10. Inspeções e testes. O padrão requer que inspeções e testes devem ser feitos durante o processo de desenvolvimento. Registros dos testes devem ser conservados.
11. Inspeções, medidas e testes de equipamentos. Poderíamos considerar equipamentos como ferramentas de software, que devem ser controladas com relação à qualidade, versão, etc.
12. Inspeções e *status* do teste. A qualidade de todos os itens em todos os estágios de seu desenvolvimento deve ser claramente conhecida e o *status* de teste deve ser conhecido de alguma forma todo o tempo.
13. Controle dos produtos que não estão em conformidade com o padrão. Produtos que não estão enquadrados aos padrões não devem ser usados.
14. Ação corretiva. Se um erro é encontrado em um item quando uma checagem de controle é feita sobre ele existem duas coisas que devem ser feitas. Primeiro, o erro deve ser removido do item; segundo, o processo envolvido na sua produção necessita ser checado para verificar se ele deve ser trocado para evitar que tal erro volte a acontecer.
15. Manuseio, armazenamento, empacotamento e entrega do produto. Uma organização que faz e vende um produto de software necessita considerar que seus produtos devem ser replicados de forma confiável, para garantir que versões corretas do software sejam entregues aos clientes corretos.
16. Registros de qualidade. O padrão requer que o desenvolvedor garanta que registros suficientes são mantidos para demonstrar que a qualidade requerida tem sido alcançada e que o sistema de gerenciamento de qualidade está operando eficientemente.
17. Auditoria de qualidade interna.
18. Treinamento. Se a equipe não é adequadamente treinada para fazer seu trabalho é claro que o trabalho das pessoas da equipe não irá atingir o grau de qualidade desejado.
19. *Servicing*.
20. Técnicas estatísticas. São usadas para verificar a aceitação das características do produto.

7.4. Comparação entre ISO 9001 x CMM

O CMM e o ISO 9001 compartilham interesses comuns com qualidade e gerenciamento de processo. Além de possuírem interesses similares, eles são intuitivamente correlacionados.

Questões que podem surgir quando comparando os dois são:

- Em que nível do CMM poderia se encaixar uma organização em conformidade com o ISO 9001?
- Uma organização de nível 2 (ou 3) no CMM poderia ser considerada em conformidade com o ISO 9001?
- Meus esforços na melhoria do processo e no gerenciamento de qualidade deveriam ser baseados no ISO 9001 ou no CMM?

Uma análise feita das diferenças e similaridades entre o CMM e o ISO 9001 indica que, embora uma organização em conformidade com o ISO 9001 não necessariamente satisfaria todas as áreas-chave de processo do nível 2 do CMM, ela satisfaria a maioria dos objetivos do nível 2 e muitos dos objetivos do nível 3. Devido à existência de práticas no CMM que não são endereçadas no ISO 9001, é possível para uma organização no nível 1 receber um certificado ISO 9001; similarmente, existem áreas endereçadas pelo ISO 9001 que não são endereçadas no CMM. Uma organização no nível 3 poderia ter pouca dificuldade em obter um certificado ISO 9001, e uma organização no nível 2 poderia ter vantagens significativas em obter o mesmo certificado [Paulk94].

A maior diferença, contudo, entre esses dois documentos é a ênfase do CMM no contínuo processo de melhora. O ISO 9001 enfoca o critério mínimo para um sistema de qualidade aceitável. Além disso, o CMM enfoca estritamente o software, enquanto que o ISO 9001 tem um escopo mais abrangente: software, hardware, materiais processados e serviços.

A maior similaridade é que para ambos, o CMM e o ISO 9001, a linha base é “dizer o que fazer e fazer o que diz”. A premissa fundamental do ISO 9001 é que todo processo importante deveria ser documentado e tudo que é entregue deveria ter sua qualidade testada através de uma atividade de controle de qualidade. O ISO 9001 requer uma documentação que contém instruções ou um guia do que deveria ser feito ou como deveria ser feito. O CMM compartilha essa ênfase em processos que são documentados. Frases conduzidas como “de acordo com um procedimento documentado” e seguindo “uma política organizacional escrita” caracterizam as áreas-chave de processo do CMM.

Na hora de decidir que norma e métrica utilizar, o empresário, que já está sendo sensibilizado, precisa optar por uma das alternativas. O presidente da Fundação Softex 2000, Kival Weber, acredita que, por ser mais conhecido e embutir um padrão internacional mínimo de qualidade, o ISO talvez traga melhores resultados para a empresa. Mas ele não despreza o CMM: “Ele é mais específico e por isso mesmo complementar.”.

O presidente da Associação Brasileira de Software (Assespro), Fábio Marinho, aconselha os empresários a fazer certos questionamentos antes de decidir sobre a métrica a ser utilizada. Segundo ele, fatores como o tamanho e o volume de desenvolvimento devem ser levados em consideração: “Muitas das pequenas empresas brasileiras de software não estão preparadas para o CMM, extremamente específico. Já com o ISO o caso é diferente, pois ele vê a empresa como um todo.”, explica.

Para o gerente do Centro de Desenvolvimento de Software da IBM Brasil , a certificação CMM ainda não se tornou uma realidade para as empresas brasileiras. Segundo ele, “O software está deixando de ser um produto de qualidade artesanal para ter um padrão industrial.”, por isso a ISO se aplica bem a ele. Já o gerente do Centro de

Desenvolvimento de Sistemas de Vitória - Espírito Santo da Xerox Corporation acha que o CMM está muito mais voltado para um modelo de desenvolvimento de software do que a ISO, pois estabelece uma maior liberdade para o próprio processo [CESAR97].

8. Visão Crítica sobre Controle de Qualidade de Software

O grande problema no controle de qualidade de software é ainda a falta de consciência de muitas empresas e profissionais que lidam com sistemas complexos em adotarem uma política de qualidade nos trabalhos a serem desenvolvidos.

Como prova disso, recorremos a uma pesquisa realizada em 1995 pelo Subprograma Setorial da Qualidade e Produtividade em Software (SSQP/SW), onde foi indicado que [CESAR97]:

- apenas 38,9% das empresas incluem sistematicamente metas ou diretrizes para qualidade em seus planos;
- 25,1% delas coletam sistematicamente indicadores de qualidade de seus produtos e serviços;
- apenas 11,5% têm um programa de qualidade total implantado;
- a grande maioria (85,9%) não conhece o modelo CMM;
- 66,6% não adota nenhum procedimento específico de garantia da qualidade do produto de software;
- a avaliação de desempenho dos funcionários é feita periodicamente em apenas 16,4% das empresas. 54,1% delas disseram fazê-la informalmente.

O importante não é o modelo a ser seguido. Quando o objetivo é otimizar a qualidade do software, deve-se ter cuidado, porém, em definir certos limites para os resultados a serem alcançados. A criação de software com qualidade requer um esforço tanto a nível financeiro quanto a nível de conscientização das pessoas envolvidas no processo, sem, no entanto, saírmos da órbita em que o cliente é o termômetro da qualidade de um determinado produto. Mesmo se o produto (software) alcançar grande parte dos requisitos de qualidade, mas ultrapassar uma data que seja de vital importância para o cliente, o produto não terá qualquer serventia e, por isso, deve-se ter um compromisso, com prazos e outras coisas mais e, em certos casos, é desejável restringir o escopo da qualidade (partindo do máximo da qualidade para uma qualidade muito boa) a ser atingida, a fim de satisfazer as necessidades do cliente. Esta é uma visão realística de encarar qualidade de software nos sistemas a serem desenvolvidos.

9. Conclusão

Qualidade é um conceito complexo, porque ela significa diferentes coisas para diferentes pessoas, ela é altamente dependente do contexto. Então, não há uma simples medida para qualidade de software que seja aceitável para todos os projetos de todas as empresas.

Para estabelecer ou melhorar a qualidade de software, o que se deve fazer é definir os aspectos de qualidade nos quais se está interessado e, então, decidir como fazer para medí-los. Antes de projetar a qualidade num software, os desenvolvedores devem concordar nas características que denotam a qualidade e nos termos que vão descrever essas características.

Apesar dos custos elevados para a introdução de certos sistemas de gerenciamento de qualidade de software, como o CMM ou o ISO 9001, é importante tais métodos serem

introduzidos, a fim de que uma empresa sobreviva por um longo tempo e possa estar sempre atualizada, com um nível de produção de software de alta qualidade, satisfazendo sempre as necessidades de seus clientes.

10. Referências

[Aurelio86] Ferreira, Aurélio B. H., Novo Dicionário da Língua Portuguesa, editora Nova Fronteira, 1986.

[Ber95] Berard, E., Metrics for Object-Oriented Software Engineering, an Internet posting on comp.software-eng, janeiro de 1995.

[Boe81] Boehm, B., Software Engineering Economics, Prentice-Hall, 1981.

[CESAR97] Centro de Estudos e Sistemas Avançados do Recife - CESAR, Informática Brasileira em Análise - ano 1, número 2, junho de 1997.

[Chi94] Chidamber, S.R., e C.F. Kemerer, A Metrics for Object-Oriented Disign, IEEE Trans. Software Engineering, vol. 20, número 6, junho 1994, pg. 476-493.

[Dav93] Davis, A. et al., Identifying and Measuring Quality in a Software Requirements Specification, Proc. 1st Intl. Software Metric Symposium, IEEE, Baltimore, MD, maio de 1993, pg. 141-152.

[Humphrey89] Humphrey, Watts S., Managing the Software Process, Addison-Wesley, 1989.

[Humphrey] Humphrey, Watts S., The Personal Software: Process Overview, Practice and Results, Carnegie Mellon University Pittsburgh, PA 15213.

[Lor94] Lorenz, M., e J. Kidd, Object-Oriented Software Metrics, Prentice-Hall, 1994.

[McDermid94] John A. McDermid, Software Engineer's Reference Book, Butterworth-Heinemann, 1994.

[Meyer88] Meyer, B., Object-oriented Software Construction, Prentice-Hall, 1988.

[Paulk94] Mark C. Paulk, A comparison of ISO9001 and the Capability Maturity Model for Software, Technical Report, julho de 1994.

[Pressman95] R.S. Pressman, , Software Engineering: A Practitioner's Approach, terceira edição, McGrawHill, 1995.

[Pressman97] R.S. Pressman, Software Engineering: A Practitioner's Approach, quarta edição, McGrawHill, 1997.

[SEI] Software Engineering Institute, http://www.sei.com.

[Shn80] Shneiderman, B., Software Psychology: Human Factors in Computer and Information Systems, Winthrop Publishers, 1980.

[Sommerville92] Ian Sommerville, Software Engineering, Addison-Wesley, 1992.

[Wir90] Wirfs-Brock, R., B. Wilkerson, e L. Weiner, Designing Object-Oriented Software, Prentice-Hall, 1990.