UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE PRODUÇÃO

SANDRA FERRARI

PROPOSTA DE METODOLOGIA PARA CONTROLE DE QUALIDADE EM UMA FÁBRICA DE SOFTWARE

Tese de Doutorado

FLORIANÓPOLIS

2007

UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE PRODUÇÃO

SANDRA FERRARI

PROPOSTA DE METODOLOGIA PARA CONTROLE DE QUALIDADE EM UMA FÁBRICA DE SOFTWARE

Tese apresentada ao Programa de Pós-Graduação em Engenharia de Produção da Universidade Federal de Santa Catarina como requisito parcial para a obtenção do título de doutor em Engenharia de Produção.

Orientador: Prof. Dr. Roberto Carlos dos Santos Pacheco

Florianópolis

2007

Dados Internacionais de Catalogação-na-Publicação (CIP)
(Biblioteca Central - UEM, Maringá – PR., Brasil)

F375p

Ferrari, Sandra
Proposta de metodologia para controle de qualidade em uma fábrica de software / Sandra Ferrari. -- Florianópolis : [s.n.], 2007.
313 f. : tabs.

Orientador : Prof. Dr. Roberto Carlos dos Santos Pacheco.
Tese (doutorado) - Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia de Produção, 2007.

1. Engenharia de software. 2. Software - Controle de qualidade. 3. Reuso de software. 4. Qualidade de software. 5. Fábrica de software. 6. Teste e inspeção de software. 7. Software - Modelo de qualidade. I. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia de Produção. II. Título.

CDD 21.ed. - 005.1

Sandra Ferrari

Proposta de Metodologia para Controle de Qualidade em uma Fábrica de Software

Esta Tese foi julgada e aprovada para obtenção do título de Doutor em Engenharia de Produção no **Programa de Pós-Graduação em Engenharia de Produção** da Universidade Federal de Santa Catarina.

Prof. Antônio Sérgio Coelho, Dr.
Coordenador do Programa

Banca Examinadora

Prof. Roberto Carlos dos Santos Pacheco, Dr.
Universidade Federal de Santa Catarina
Orientador

Prof. Marcello Thiry Comicholi da Costa, Dr.
Universidade Federal de Santa Catarina

Prof. João Bosco da Mota Alves, Dr.
Universidade Federal de Santa Catarina

Profa. Itana Maria de Souza Gimenes, Dra.
Universidade Estadual de Maringá

Prof. Donizete Carlos Bruzarosco, Dr.
Universidade Estadual de Maringá

Aos meus filhos, Tarso e Nina, mais mestres que aprendizes.

AGRADECIMENTOS

A todos que direta e indiretamente contribuíram para o desenvolvimento deste trabalho, externo o meu mais profundo reconhecimento e agradecimento.

Ao Prof. Dr. Roberto Pacheco, pela confiança em mim depositada, pela amizade e pela orientação durante todo o desenvolvimento do trabalho.

À CAPES-PICD pelo apoio financeiro recebido;

Ao Prof. Dr. Marcello Thiry pela prontidão, disposição e dedicação recebidas na forma de orientação, ao longo da realização deste trabalho.

À professora Itana M. de S. Gimenes, por ter tornado possível o término deste trabalho.

Ao Marco Aurélio pelo valioso suporte para a realização do estudo de caso.

Aos professores Wesley Romão e Donizete Bruzarosco pelo apoio recebido.

À Universidade Estadual de Maringá pelo apoio e liberação para cursar o doutorado.

À Cássia Lima grande e querida amiga, por todas as valiosas e aconchegantes acolhidas.

À Rosimeri Maria de Souza, pela paciência, orientação e ajuda junto à PPGEP.

Aos meus filhos pelo incentivo e apoio em todos os momentos.

RESUMO

A indústria de software vem, ao longo dos anos, evidenciando esforços, no sentido de acompanhar as mudanças no mercado devido à acirrada competitividade, decorrentes de uma demanda constante e crescente por produtos de alta qualidade. A permanência num mercado altamente dinâmico e exigente impulsiona as organizações na busca de tecnologias, técnicas e metodologias que possam apoiar suas atividades com o intuito de construir produtos de alta qualidade, dentro de orçamentos e cronogramas reais, com custos razoáveis sem, contudo, extrapolar os prazos de entrega nem comprometer a produtividade e a qualidade de seus produtos e processos. A preocupação com a qualidade tem levado as organizações a adotarem distintas abordagens que, se levadas a efeito de maneira sistemática podem contribuir grandemente para a solução dos problemas enfrentados pelas organizações de desenvolvimento de software. O objetivo principal deste trabalho é a proposição de uma abordagem que compreende a utilização combinada (integrada) de reuso sistemático, teste e inspeção de software, no contexto de fábrica de software, visando à melhoria da qualidade. A metodologia proposta é validada, aplicando o modelo UML *Components* utilizando-se como suporte computacional a ferramenta X-Packager, a qual descreve e empacota os *assets* gerados ao longo do processo de desenvolvimento. Pode-se destacar como contribuições deste trabalho, a sistematização da aplicação de atividades combinadas que levam à qualidade; um melhor entendimento dos conceitos e abordagens relacionados a tais atividades; um melhor entendimento dos modelos de qualidade de produto e de processo e das diversas abordagens de fábrica de software, que embora não sejam recentes, são extremamente atuais e têm merecido a atenção de profissionais, pesquisadores e empresas que atuam nesta área.

Palavras chave: Qualidade de software, reuso, teste, inspeção, fábrica de software.

ABSTRACT

Software industry has over the years put effort into reacting to changes boosted by an ultra-competitive market, with a growing demand for high-quality products. The permanence in a highly dynamic and demanding market pushes organizations towards the search for technology, techniques and methodologies that can support their activities. The goal is to produce high-quality products within realistic budget and schedule, within reasonable costs without, however, affecting the productivity and the quality of products and processes. The concern with software quality has taken the organizations to adopt distinct approaches that, if taken seriously and systematically, can contribute greatly for the solution of the problems faced by software development organizations. This work proposes a methodology for quality control that combines systematic reuse, testing and software inspection, in the context of software factories. The results of a case study are also presented in order to validate the proposed methodology, by means of the application of the UML Components model and by using X-Packager, a tool that describes and packages the assets that are generated during software development. Some of the contributions of this work are: the systematization of the application of combined activities that promote software quality; a better understanding of the concepts and approaches related to such activities; a better understanding of process and product quality models as well as software factories, a hot research topic that has deserved the attention of professionals, researchers and companies who work within this area.

Key Words: Software Quality, reuse, test, inspection, Software Factory.

SUMÁRIO

LISTA DE ACROSSEMIAS 11

LISTA DE FIGURAS 13

LISTA DE TABELAS 15

1. INTRODUÇÃO 17

1.1 JUSTIFICATIVA 19

1.2 OBJETIVOS 21

1.2.1 ***Objetivos específicos*** *23*

1.3 CONTRIBUIÇÕES 23

1.4 ORGANIZAÇÃO DO TRABALHO 24

2. MODELOS DE QUALIDADE 27

2.1 MODELOS DE QUALIDADE DE PROCESSO DE SOFTWARE 29

2.1.1 ***O Modelo CMMI*** *32*

2.1.1.1 Componentes do modelo CMMI 35

2.1.1.2 Áreas de processo 38

2.1.1.3 Representação contínua 48

2.1.1.4 Representação por estágio 58

2.1.2 ***O Padrão ISO 12207*** *69*

2.1.2.1 Processos fundamentais 71

2.1.2.2 Processos de suporte ao ciclo de vida 73

2.1.2.3 Processos organizacionais do ciclo de vida 76

2.1.3 ***O Padrão ISO 15504*** *78*

2.2 MODELOS DE QUALIDADE DE PRODUTO DE SOFTWARE 93

2.2.1 ***O Padrão ISO 9126*** *103*

2.2.1.1 Qualidade de Produto e o Ciclo de Vida 107

2.2.1.2 Usando um Modelo de Qualidade 108

2.2.1.3 Modelo de qualidade para qualidade interna e externa 108

2.2.1.4 Modelo de qualidade para qualidade em uso 112

2.3 QUALIDADE NO BRASIL 113

3. FÁBRICA DE SOFTWARE 126

3.1 A ORGANIZAÇÃO DE SOFTWARE INDUSTRIALIZADA – JAPÃO 129

3.2 A FÁBRICA DE SOFTWARE GENÉRICA – EUROPA 133

3.3 A FÁBRICA DE COMPONENTES BASEADA EM EXPERIÊNCIA - ESTADOS UNIDOS 137

3.4 A ORGANIZAÇÃO MADURA DE SOFTWARE – ESTADOS UNIDOS 141

3.5 A FÁBRICA DE SOFTWARE DA MICROSOFT 145

4. INSPEÇÕES E TESTE DE SOFTWARE 160

4.1 MÉTRICAS E MEDIÇÕES DE SOFTWARE 161
4.2 INSPEÇÕES DE SOFTWARE 166
4.3 TESTE DE SOFTWARE 180
4.3.1 ***Planejamento de teste*** *186*
4.3.2 ***Estratégias de teste*** *189*
4.3.3 ***Casos de teste*** *190*

5. REUSO DE SOFTWARE 195

5.1 DESENVOLVIMENTO DE SOFTWARE BASEADO EM COMPONENTES 209
5.1.1 ***Desenvolvimento de software com reuso*** *219*
5.1.2 ***Desenvolvimento de software para reuso*** *224*
5.2 MODELAGEM E REUSO 229
5.3 REUSO BASEADO EM GERADOR 237
5.4 FRAMEWORKS, PADRÕES DE PROJETO E REUSO 243

6. ABORDAGEM PROPOSTA 253

6.1 DESCRIÇÃO DA ABORDAGEM PROPOSTA 254
6.2 COMO UTILIZAR A METODOLOGIA PROPOSTA 257
6.3 AVALIAÇÃO DA ABORDAGEM PROPOSTA 261
6.3.1 ***Uml components*** *262*
6.3.2 ***O modelo x-arm*** *269*
6.3.3 ***A ferramenta x-packager*** *276*
6.4 ESTUDO DE CASO 279
6.5 CONSIDERAÇÕES FINAIS 288

7. TRABALHOS FUTUROS 290

8. GRÁFICOS DAS REFERÊNCIAS BIBLIOGRÁFICAS UTILIZADAS 291

9. REFERÊNCIAS BIBLIOGRÁFICAS 292

LISTA DE ACROSSEMIAS

COBOL: *Common Business Oriented Language*.

CCM: *Capability Maturity Model.*

CMMI: *Capability Maturity Model Integration.*

COTS: *Commercial- Of-The-Shelf*.

DBC: Desenvolvimento Baseado em Componentes.

EAP: Equipes de Ação de Processo.

EIA/IS: *Electronic Industries Alliance*/*Interim Standard*.

EUA: Estados Unidos da América.

GAT: *Guidance Automation Toolkit*.

GEP: Grupo de Engenharia de Processo.

GQM: *Goal Question Metrics*.

IPD-CMM: *Integrated Product Development Team – Capability Maturity Model*.

IPDD: *Integrated Product and Process Development.*

ISO: *International Organization for Standardization*.

ISO/IEC: *International Organization for Standardization*/ *International Electrotechnical Commision*.

KLOC: *Thousand (Kilo) of Lines of Code*.

MCT: Ministério da Ciência e Tecnologia.

PBQP: Programa Brasileiro de Qualidade e Produtividade de Software.

PSP: *Personal Software Process.*

Q&P: Qualidade e Produtividade.

QPS: Qualidade e Produtividade de Software.

RUP: *Rational Unified Process*.

SA-CMM: *Software Acquisition - Capability Maturity Model*.

SE-CMM: *Systems Engineering - Capability Maturity Model*.

SECAM: *Systems Engineering Capability Assessment Model*.

SECM: *Systems Engineering Capability Model*.

SEI: *Software Engineering Institute*.

SEPIN: Secretaria de Política de Informática do MCT.

SPICE: *Software Process Improvement and Capability dEtermination*.

SOFTEX: Associação para Promoção da Excelência do Software Brasileiro.

SQA: *Software Quality Assurance*.

SQL: *Structured Query Language*.

SSQP-SW: Sub-comitê Setorial da Qualidade e Produtividade em Software.

SW-CMM: *Capability Maturity Model for Software*.

TQM: *Total Quality Management*.

USA: *United States of America*.

VLSI: *Very Large Scale of Integration*.

X-ARM: *XML-based Asset Representation Model*.

X-Packager: *X-ARM Packager*.

XML: *Extensible Markup Language*.

WQS: *Workshop* de Qualidade de Software.

LISTA DE FIGURAS

Figura 1. Modelos que influenciaram diretamente o CMMI (KULPA, 2003) 34
Figura 2. Componentes do modelo CMMI 41
Figura 3. Diferenças entre as organizações maduras e não maduras de software. 45
Figura 4. Estrutura da representação contínua (SEI, 2002a) 51
Figura 5. Estrutura da representação por estágio (SEI, 2002a) 58
Figura 6. Estrutura do padrão ISO/IEC 12207 (2003) 70
Figura 7. Relacionamento de avaliação de processo (ISO/IEC 15504, 2003) 81
Figura 8(a). Uso da norma 15504 para melhoria de processo (ROCHA, 2001) 87
Figura 8(b). Uso da norma 15504 para a determinação da capacidade de processo (ROCHA, 2001) 88
Figura 9. Uma visão da arquitetura bi-dimensional da ISO/IEC 15504 (EMAM, 2000) 89
Figura 10. Qualidade no ciclo de vida 106
Figura 11. Modelo de qualidade para qualidade interna e externa 109
Figura 12. Modelo de qualidade para qualidade em uso 112
Figura 13. Q&P em software: um projeto coletivo 115
Figura 14. Qualidade dos processos 117
Figura 15. Melhoria do processo de software 118
Figura 16. Conhecimento do CMM 119
Figura 17. Processos de ciclo de vida de software 120
Figura 18. Avaliação de processo de software 121
Figura 19. Qualidade dos produtos 122
Figura 21. Produzindo uma instância de uma fábrica de software (AAEN, 1997) 135
Figura 22. A organização de uma fábrica de componentes baseada em experiência, (AAEN, 1997) 139
Figura 23. As áreas chave de processo no CMM (PAULK, 1993) 142
Figura 24. Os quatro pilares de uma fábrica de software (EADIE, 2006) 151
Figura 25. O ciclo de vida da fábrica (EADIE, 2006) 152
Figura 26. Uma fábrica de software (EADIE, 2006) 156
Figura 27. O processo de inspeção (SOMMERVILLE,1996) 174
Figura 28. Trabalho em equipe na reparação de erros (WEINBERG, 1996) 177

Figura 29. Porque as equipes podem melhor identificar as falhas (WEINBERG, 1996).....178
Figura 30. Porque as equipes são potencialmente melhores que os indivíduos para reparar erros (WEINBERG, 1996) 179
Figura 31. O processo de teste.....183
Figura 32. Conteúdo do plano de teste (SOMMERVILLE, 1996).....187
Figura 33. Fases de teste no processo de desenvolvimento de software Sommerville (1996) 188
Figura 34. Recursos para o desenvolvimento de software 196
Figura 35. Reuso em um processo de desenvolvimento padrão (SOMMERVILLE, 1996) ..221
Figura 36. Processo de desenvolvimento dirigido a reuso (SOMMERVILLE, 1996).....221
Figura 37. O processo de aumento de reusabilidade (SOMMERVILLE, 1996).....226
Figura 38. Categorização de métricas e modelos de reuso (FRAKES, 1996).....232
Figura 39. Reuso de conhecimento de domínio da geração de aplicação (SOMMERVILLE, 1996).....237
Figura 40. Fatores que influenciam o desenvolvimento de software em termos de custo/benefício (HERZUM, 2000) 244
Figura 41. Modelo da proposta de abordagem para controle de qualidade.....254
Figura 42. *Workflows* do processo de desenvolvimento (CHEESMAN e DANILES, 2001) 263
Figura 43. Modelos resultantes da modelagem de domínio (GIMENES *et al*, 2005).....265
Figura 44. Estágios da Modelagem de especificação (GIMENES *et al*, 2005).....266
Figura 45. *Workflow* da modelagem de especificação 267
Figura 46. Hierarquia de nomes no X-ARM (SANTOS, 2006, p. 23).....274
Figura 47. Artefatos gerados no *workflow* de requisitos 287
Figura 48. Artefatos gerados no *workflow* de especificação 287

LISTA DE TABELAS

Tabela 1. Áreas de processo, categorias e níveis de maturidade associados (CHRISSIS, 2003) 38
Tabela 2. Comparação entre os níveis de capacidade e de maturidade 47
Tabela 3. Vantagens comparativas da representação contínua e por estágio (CHRISSIS, 2003) 65
Tabela 4. Comparando as representações (CHRISSIS, 2003) 66
Tabela 5(a). Uma visão geral dos níveis de capacidade e atributos (EMAM, 2000) 90
Tabela 5(b). Uma visão geral dos níveis de capacidade e atributos (EMAM, 2000) 91
Tabela 6. Escala de avaliação de atributos de quatro pontos, (EMAM, 2000) 92
Tabela 7. Modelo de fatores de qualidade de software segundo McCall 96
Tabela 8. Modelo de fatores de qualidade de Software, segundo Deutsch e Willis 97
Tabela 9. Lista de fatores de qualidade, segundo (ALSPAUGH, 2005) 98
Tabela 10. Modelos de fatores de qualidade (KHADDAJ, 2004) 99
Tabela 11. Características e sub-características do modelo ISO 9126 (JUNG, 2004) 101
Tabela 12. Papéis (funções) no processo de inspeção (SOMMERVILLE, 1996) 172
Tabela 13. Controles de inspeções (SOMMERVILLE, 1996) 175
Tabela 14. Aspectos reutilizáveis de projetos de software (JONES, 1991) 201
Tabela 15 – A visão acadêmica versus a visão industrial de componentes de software 212
Tabela 16. Tipos de reuso de software (FRAKES, 1996) 233
Tabela 17(a). Definições de tipos de reuso (FRAKES, 1996) 235
Tabela 17(b). Definições de tipos de reuso (FRAKES, 1996) 235
Tabela 17(c). Definições de tipos de reuso (FRAKES, 1996) 236
Tabela 19. Comparação dos modelos de representação de artefatos de software (SANTOS, 2006) 270
Tabela 20. Categorias e tipos de *assets* do X-ARM 272
Tabela 21. Atividades e artefatos do *workflow* de requisitos 279
Tabela 22. Atividades e artefatos do *workflow* de especificação 281
Tabela 23. Atividades e artefatos do *workflow* de especificação 282
Tabela 24. Atividades e artefatos do *workflow* de especificação 283
Tabela 25. Atividades e artefatos do *workflow provisioning* 284

Tabela 26. Tipos e quantidade de *assets* gerados (SANTOS, 2006)..... 285

1. INTRODUÇÃO

À medida que aumenta o uso de software, inclusive em situações críticas, recrudesce a constatação de que esses produtos demandam uma qualidade garantida. A qualidade dos produtos de software pode ser avaliada tanto pelas características do próprio produto, quanto pelo processo por meio do qual o software é construído. Vários modelos e normas de avaliação têm sido propostos visando à melhoria e a regulamentação da qualidade dos produtos e processos de desenvolvimento de software. Dentre os quais, destacam-se, as normas ISO[1] 9000-3, cujo foco é a melhoria do processo de software e a ISO 9126, a qual enfatiza a melhoria da qualidade do produto de software e, os modelos de avaliação e melhoria do processo de software, como o CMMI[2] (SEI, 2002a), a ISO 15504 (EMAN, 2000) e ISO 12207 (ISO/IEC, 2003).

Qualidade é um tema complexo, dado que existem muitos pontos de vista sobre qualidade e muitas visões sobre que ações executar, a fim de melhorar a qualidade de software. Uma distinção particularmente importante é entre o que representa qualidade para o usuário e o que representa qualidade para o desenvolvedor de um produto de software.

Existem vários modelos (CMMI, ISO 15504, ISO 12207) que, embora não determinem passos a serem seguidos para a consecução da qualidade, servem como uma diretriz para as organizações que querem galgar níveis de qualidade, tornando-se para tanto, organizações maduras. Esses modelos são referências valiosas, uma vez que possibilitam às organizações a escolha de um caminho a ser seguido em busca da qualidade desejada e também, porque possibilitam que as organizações avaliem seus processos de melhoria.

Conforme abordado pelo CMMI (SEI, 2002a), existem alguns estágios evolutivos pelos quais uma organização deve passar para conseguir a produção de software por meio de um processo otimizado e continuamente melhorado. Cada estágio abrange um conjunto de atividades

[1] Do inglês: *International Organization for Standardization*.

[2] Do inglês: *Capability Maturity Model Integration*.

técnicas que devem ser desenvolvidas para que uma organização – após processo de avaliação - possa ser considerada como tendo atingido um determinado estágio de capacitação do processo de software.

Dentre as atividades que contribuem para a melhoria da qualidade do produto estão inspeção e teste de software. Inspeção de software tem sido vista como um dos meios mais efetivos para promover qualidade e produtividade no desenvolvimento de software (TYRAN, 2006; BOEHM, 2006). O principal objetivo da inspeção de software é detectar falhas em um artefato de software. Isso ajuda a reduzir o número de falhas e a aumentar a qualidade de um produto de software (THELIN, 2004; BIFFL, 2003).

Enquanto o teste de software é considerado uma atividade muito eficiente para detectar defeitos por meio da execução dinâmica do software (MARTIN, 2006), a inspeção de software é considerada a melhor prática na verificação estática do software, sendo considerado o processo mais estruturado para detecção e remoção de defeitos nos artefatos de software (BOEHM, 2006),

Inspeção de software contribui para a melhoria da qualidade do produto e do processo, uma vez que os dados coletados durante o processo de inspeção são usados não somente para corrigir defeitos, mas também para avaliar e melhorar o processo de desenvolvimento. As vantagens da inspeção de software, considerada pela indústria, como uma técnica forte para a remoção de defeitos, são bem documentadas, dado que *frameworks* de melhoria de processo como o CMMI, defendem fortemente o uso de inspeções (KELLY, 2004).

À medida que o processo de produção de software se torna cada vez mais competitivo, a melhoria da qualidade dos produtos de software, além de ser um diferencial, passa a ser também um fator crítico para empresas que desejam sobreviver em um mercado, em que a concorrência se torna mais acirrada quanto maior é a demanda por produtos de qualidade. Assim sendo, adotar uma abordagem visando assegurar a melhoria da qualidade do processo de produção e do próprio software passa a ser também uma questão de sobrevivência. Para

Lavazza (2005), a qualidade está ganhando mais atenção atualmente, tanto quanto mais ênfase está sendo colocada na produção de produtos de software de alta qualidade.

1.1 JUSTIFICATIVA

O desenvolvimento de software de grande porte, sofisticado e complexo, os constantes avanços na indústria de hardware e a exigência cada vez maior do mercado por produtos com alto nível de qualidade, têm suscitado uma necessidade também crescente de maior empenho dos profissionais da área de engenharia de software, no que tange a estudos em busca de métodos e metodologias para apoiar o processo de produção de software, visando atender a um mercado, cuja demanda por produtos de software de qualidade, está cada vez mais exigente.

As dificuldades enfrentadas pelos pesquisadores estão relacionadas ao fato de que, não existe um conjunto de características de qualidade bem definido e que seja comum às organizações de desenvolvimento de software. Tampouco, há um consenso sobre quais aspectos de qualidade devem ser considerados e sobre como implementar uma sistemática eficiente para a obtenção da qualidade almejada por ambos os segmentos: produtores e consumidores de produtos de software de qualidade (BASILI, 1995; WONG, 2004; HILBURN, 2000).

O desenvolvimento de software tal como praticado atualmente, é lento, dispendioso e propenso a erros, produzindo freqüentemente, produtos com grande número de defeitos, causando sérios problemas de usabilidade, confiabilidade, desempenho, segurança e outras qualidades de serviço (GREENFIELD, 2004). É nesse panorama que surgem as abordagens de fábrica de software, como uma etapa lógica na contínua evolução dos métodos e práticas para o desenvolvimento de software de uma maneira rápida, econômica e com qualidade, enfatizando, em sua maioria, o reuso sistemático como uma forma de aumentar a produtividade e a qualidade dos seus produtos e serviços.

A grande preocupação das empresas, entidades de classe e do governo com aspectos de qualidade e o constante empenho da comunidade acadêmica da área em pesquisar tal tema, por si só, são fatores suficientes, além de importantes, para impulsionar e legitimar o trabalho em questão.

É evidente a necessidade existente atualmente nas organizações de aumento e melhoria de sua produtividade, de melhoria da qualidade de seus produtos de software e de redução do custo de produção. A engenharia de software vem buscando desenvolver métodos, técnicas, ferramentas e notações que sirvam de suporte para alcançar estes objetivos. Uma técnica importante para melhorar a qualidade e a produtividade do processo de software é a reuso de soluções (GIMENES, 2002).

Segundo Basili (1995), uma organização que deseja melhorar a qualidade em um ambiente de desenvolvimento, ao invés de em um ambiente de processo, deve construir ou empacotar modelos, de modo que sejam reutilizáveis por outros projetos na organização, isto é, reutilizar conhecimento e experiência. No centro de uma estratégia de melhoria reside a necessidade de experiência reutilizável e uma infra-estrutura organizacional apontada para a capitalização e reuso de experiência e produtos de software (BASILI, 1995).

São muitas as razões que dificultam ou mesmo inviabilizam a implantação de abordagens que garantam a consecução dos objetivos pretendidos de qualidade. As razões vão desde o desconhecimento ou pouco conhecimento dos objetivos de negócios da organização até a falta ou falha na adoção de uma determinada abordagem concernente à qualidade. Por alguns dos argumentos já apresentados anteriormente, fica evidente que, mesmo em face aos diversos métodos, metodologias e técnicas visando à obtenção da qualidade de processo e de produto, nem sempre é possível aplicar e avaliar qualidade. Também é difícil aprender por meio da experiência de outras organizações uma vez que a realidade de cada uma é, na maioria das vezes muito distinta, devido aos aspectos culturais e às políticas de negócio vigentes. Por outro lado, a competitividade do mercado e a falta de resultados de estudos concernentes a essa questão, inviabilizam o intercâmbio de informações entre empresas com interesses comuns acerca de qualidade de software.

1.2 OBJETIVOS

Nem sempre é fácil definir qual o melhor tipo de atividade que deve ser levada a efeito quando se trata de verificar ou assegurar qualidade. Normalmente, tais atividades cobrem um determinado espectro de ciclo de desenvolvimento, não conseguindo, portanto, detectar todos os problemas que podem ocorrer durante o processo de desenvolvimento. Assim, diante da necessidade premente de assegurar que processo e produto tenham o nível de qualidade que possa atender as exigências ditadas pelo mercado e, diante da impossibilidade de que apenas uma atividade possa assegurar a qualidade requerida, este trabalho propõe uma abordagem que compreende o uso combinado - em conjunto - das atividades de inspeção e teste, mais a prática de reuso, as quais devem ser realizadas sistematicamente, para assegurar a melhoria da qualidade - de modo que ao se complementarem, aumentam grandemente a possibilidade de obtenção de produtos e processos com mais qualidade e, portanto, mais confiáveis. A **hipótese** que fundamenta o desenvolvimento de uma metodologia para apoiar o processo de produção de software com qualidade está embasada nas evidências de que a adoção de uma abordagem que privilegia a qualidade contribui efetivamente para as melhorias desejadas, além de facilitar o processo de produção de diferentes sistemas e diminuir os custos com a sua produção e manutenção.

Com o intuito de contribuir para com a área de engenharia de software, no que concerne à melhoria da qualidade de software e do processo por meio do qual o mesmo é produzido, este trabalho tem como objetivo a proposição de uma abordagem que contempla o uso sistemático e combinado – em conjunto - das atividades de teste e inspeção, aliadas à prática de reuso como um meio efetivo para assegurar qualidade. A abordagem proposta está inserida no contexto de fábrica de software (GREENFIEL e SHORT, 2003; VRIES, 2006; GREENFIELD, 2004) e tem como propósito, servir como uma diretriz para as empresas que adotam o conceito de fábrica de software para o desenvolvimento de seus produtos, no que concerne a qualidade.

A proposição de uma abordagem que possibilite aos profissionais de engenharia de software, a construção de produtos de software de maneira sistemática, dentro de um nível de qualidade

que possa atender as demandas de mercado, não é uma preocupação recente. Entretanto, a consideração em conjunto, das atividades de inspeção e teste, aliada à prática de reuso, faz desta, uma proposta atual e inovadora em termos de concepção, objetivos e aplicação.

Busca-se, portanto, com a abordagem proposta, uma maneira de controlar a qualidade de processo e de produtos de software durante todo o ciclo de desenvolvimento, de modo a assegurar que os artefatos sejam liberados com um maior grau de confiança possível aos usuários.

A idéia é que ao longo do processo de desenvolvimento, os artefatos possam ser inspecionados, revisados e testados visando à eliminação de defeitos, numa tentativa de: **1**) impedir que os possíveis defeitos se propaguem aos estágios subseqüentes do ciclo de desenvolvimento; **2**) que os produtos de software sejam liberados com a menor quantidade de defeitos possíveis; **3**) que o reuso de artefatos produzidos durante o ciclo de desenvolvimento seja sistemático; **4**) que seja possível construir produtos de qualidade dentro de cronogramas, orçamentos e prazos previstos e; **5**) que haja um processo iterativo que possibilite o acompanhamento de todos os envolvidos em todas as atividades inerentes ao processo de desenvolvimento de um produto de software de modo que a aplicação da abordagem ora proposta seja eficiente e eficaz.

Objetiva-se, portanto, o desenvolvimento de uma abordagem que possa servir como diretriz para as empresas de desenvolvimento de software que visam oferecer produtos com maior confiabilidade, de alta qualidade e com custos não tão elevados, respeitando cronogramas, orçamentos e os prazos de entrega estipulados, de maneira a permanecerem competitivas num mercado cada vez mais exigente quanto aos padrões de qualidade, custo e tempo.

1.2.1 Objetivos específicos

São objetivos específicos deste trabalho de tese:

- Realizar um estudo de caso para avaliar a metodologia proposta.
- Estudar, entender e aplicar o UML *Components* no estudo de caso.
- Estudar e entender o modelo X-ARM, no qual está baseada a ferramenta X-Packager.
- Compreender o funcionamento da ferramenta X-Packager.
- Avaliar a metodologia proposta utilizando a ferramenta X-Packager e aplicando o UML *Components*.

A proposta é avaliada por meio de um estudo de caso. No estudo de caso, o processo de desenvolvimento que faz parte da metodologia proposta, é instanciado pelo modelo UML *Components* (CHESSMAN, 2001), utilizando-se a ferramenta X-Packager[3] a qual foi "criada [...] com o objetivo de facilitar a descrição e o empacotamento de *assests* de acordo com o modelo X-ARM" (SANTOS, 2006, p. 5).

1.3 CONTRIBUIÇÕES

As principais contribuições deste trabalho são: a proposição de uma abordagem visando assegurar a qualidade de processo e de produto no contexto de fábrica de software, de modo que em cada etapa do desenvolvimento a qualidade do processo de produção possa ser avaliada com o intuito de verificar sua qualidade e, ao término do processo de produção, seja também possível verificar a qualidade do produto com o objetivo de melhorá-lo. Dessa maneira, a idéia é fazer com que: o processo produtivo seja menos custoso; haja um aumento

[3] X-ARM *Packager*.

significativo na produtividade e na qualidade de processo e produto e, finalmente, que possa haver mais confiança nos produtos liberados aos clientes e usuários.

Outra contribuição é evidenciar a possibilidade de implementação da metodologia proposta por meio da ferramenta X-Packager, a qual possibilita o empacotamento dos artefatos com todas as funcionalidades presentes no modelo X-ARM, tais como: “identificação, classificação, informações sobre uso dos *assets* e o próprio empacotamento dos artefatos que compõem os *assets*” (SANTOS, 2006, p. 141).

A metodologia utilizada para o desenvolvimento do trabalho em questão contempla uma ampla e profunda revisão bibliográfica sobre: os modelos de qualidade – produto e processo; reuso de software; inspeção e teste e fábrica de software. O estudo criterioso dos temas abordados nos capítulos subseqüentes oferece o referencial teórico necessário para subsidiar a proposição da abordagem pretendida. Além disso, é realizado um estudo de caso visando avaliar a abordagem proposta, utilizando o modelo UML *Components* (CHESSMAN, 2001) para instanciar o processo de desenvolvimento e a ferramenta X-Packager, para empacotar os artefatos gerados em cada etapa da aplicação do modelo em questão.

1.4 ORGANIZAÇÃO DO TRABALHO

O restante deste trabalho está organizado da seguinte forma: no Capítulo 2, são descritos os modelos de qualidade de software que são referência na área de engenharia de software. Embora esses modelos não determinem caminhos a serem seguidos, os mesmos constituem uma valiosa diretriz uma vez que apontam uma direção que pode levar à obtenção dos níveis de qualidade almejados por uma determinada empresa. Por serem padrões internacionais de qualidade e referências para certificações quanto a padrões de qualidade, é mister o entendimento dessas diretrizes, quando e como aplicá-las, assim como, das melhoria alcançadas com a sua utilização. Assim sendo, uma descrição bem detalhada, que possibilite uma compreensão adequada dos conceitos, políticas, estratégias e práticas que norteiam tais

modelos, constitui-se uma ferramenta valiosa para os profissionais da área. Contudo, uma das maiores contribuições reside no fato de que o estudo e compreensão de tais modelos são fundamentais, não apenas para que as empresas possam ter um suporte para a implantação de programas de melhoria, mas também, para embasar a abordagem de melhoria proposta neste trabalho.

As diversas abordagens de fábrica de software são apresentadas no Capítulo 3. Neste Capítulo são apresentados os conceitos de fábrica de software relativos à: organização de software industrializada – Japão; fábrica de software genérica – Europa; fábrica de componentes baseada em experiência – Estados Unidos; a organização madura de software - Estados unidos e a abordagem de Fábrica de Software proposta pela Microsoft.

O Capítulo 4 contém a explicitação das atividades de inspeção e teste, que, juntamente com a prática de reuso, compõem a essência da abordagem para melhoria da qualidade ora proposta. A primeira delas é a atividade de inspeção e todo o processo que envolve a sua realização, desde o planejamento, execução, documentação e disseminação dos resultados de sua aplicação. A segunda atividade descrita, para assegurar qualidade é a atividade de teste. É apresentada uma descrição dos estágios que fazem parte do processo de teste, do planejamento de teste, das estratégias de teste e dos casos de teste. Ambas as atividades (inspeção e teste) fazem parte de uma atividade mais abrangente, conhecida como verificação e validação, também explicitada, embora não em profundidade.

No Capítulo 5, são descritos os conceitos de reuso de software, os diversos tipos de reuso existentes, as vantagens de seu uso e sua aplicação sistemática ou casual, ao longo do tempo, pelos profissionais da área. É feita uma explanação da aplicação de reuso desde o seu surgimento até a abordagem mais atual que é o reuso de experiência e conhecimento. É enfatizada a importância de se adotar a abordagem de reuso como uma medida efetiva para melhorar a produtividade e a qualidade do produto e do processo de desenvolvimento de software.

A abordagem para melhoria proposta neste trabalho é explicitada no Capítulo 6. Trata-se de uma abordagem de melhoria da qualidade, baseada na aplicação combinada de reuso, teste e inspeção, considerando o contexto de uma fábrica de software. O uso combinado de inspeção e teste tem o propósito de funcionar como uma atividade guarda chuva, possibilitando que a qualidade seja assegurada em todos os estágios do processo de desenvolvimento. A abordagem envolve sistematicamente as atividades de inspeção e teste, aliadas à prática de reuso de software, desde a etapa de concepção do software até a sua manutenção.

Ainda no Capítulo 6, é apresentada uma avaliação da abordagem proposta e são apresentados os resultados obtidos por meio de um estudo de caso, resultante da aplicação do modelo UML *Components* (CHESSMAN, 2001) na metodologia ora apresentada. São apresentadas as considerações finais sobre o trabalho realizado, bem como apontados os trabalhos futuros, possíveis de serem desenvolvidos a partir deste trabalho.

2. MODELOS DE QUALIDADE

A satisfação do cliente tem sido o lema de muitas organizações na tentativa de sobreviverem e serem bem sucedidas no mundo atual, onde a competitividade aumenta constantemente. Ao mesmo tempo em que as organizações enfatizam a satisfação do cliente, há uma crescente percepção de que a qualidade do software é a ligação delicada no desenvolvimento de produtos de qualidade. O software se difunde no dia-a-dia devido ao seu poder e a tecnologia de software está avançando de modo surpreendente, a despeito desse avanço, parece que a complexidade dos problemas relacionados ao software está crescendo mais rapidamente que a capacidade dos profissionais da área em desenvolver e manter software.

As organizações em geral, poderiam implementar boas práticas apropriadas para os objetivos de desenvolvimento de suas aplicações. Boas práticas referem-se às práticas de engenharia de software que melhor apóiam o desenvolvimento eficiente de software (HERZUM, 2000). Ou seja, as práticas de engenharia de software incluem metodologia, processo de desenvolvimento e boas práticas exigidas para se ter uma organização que utiliza ferramentas integradas e ambientes de desenvolvimento para produzir software eficientemente.

De acordo com Rocha (2001), a preocupação com o processo de software está relacionada à necessidade de entender, avaliar, controlar, aprender, comunicar, melhorar, predizer e certificar o trabalho dos engenheiros de software. Assim sendo, são necessários mecanismos que possibilitem a documentação, medição, definição, análise, avaliação, comparação e a alteração de processos, conforme explicitado por Lindvall (2000), bem como, o gerenciamento de processos e de produtos.

Boas práticas são exigidas em todo o processo de desenvolvimento e cobrem todos os aspectos de desenvolvimento em larga escala, desde coleta de requisitos, verificação, validação e garantia da qualidade e, desde gerenciamento de projeto até engenharia de processos incluindo medições e abordagens de otimização de processo. De um modo amplo, uma boa prática pode ser definida como sendo todos os aspectos não específicos em termos

de tecnologia que são requeridos para preencher a lacuna entre a metodologia instanciada e o desenvolvimento real (HERZUM, 2000).

Tais boas práticas, segundo as colocações de Emam (2000), são comumente codificadas em um modelo de avaliação, tal como o CMM para software ou o padrão internacional ISO/IEC[4] 15504. Tais modelos também ordenam as práticas em uma seqüência recomendada de implementação, fornecendo, portanto, um caminho de melhoria predefinido. Alguns desses modelos incluem o processo de engenharia de requisitos dentro de seu escopo, definindo assim, quais são consideradas as melhores práticas da engenharia de requisitos.

De acordo com Butler (1997), a modelagem por si só não fornece uma vantagem econômica direta, mas os modelos são ferramentas essenciais e a modelagem é uma atividade essencial na melhoria da qualidade de produtos, de processos e reuso efetivo. Modelos são abstrações que fornecem aos desenvolvedores, controle intelectual sobre a complexidade de um sistema. Para Kulpa (2003), um modelo é considerado uma diretriz das melhores práticas encontradas no estudo de outras organizações que funcionam bem e são muito bem sucedidas. Um modelo não contém os passos necessários ou a seqüência de passos necessários para implementar um programa de melhoria de processo.

A escolha de um modelo depende dos problemas existentes em uma organização os quais se deseja solucionar. Sem o uso de um modelo como referência, não se tem uma base a partir da qual é possível planejar a melhoria desejada e também não se tem nada contra o que seja possível medir e comparar os resultados. O desenvolvimento de um modelo não se constitui uma tarefa banal, requer tempo, demanda um alto custo e, como esses modelos sumarizam as melhores práticas de organizações bem sucedidas, não há necessidade de as organizações criarem seus próprios modelos, já que podem seguir a maioria das melhores práticas documentadas nesses modelos. Muitos modelos possibilitam que uma organização substitua as práticas alternativas por aquelas práticas no modelo escolhido que não estão em conformidade com a organização. Entretanto, é preciso cautela, quanto mais alternativas

[4] Do inglês: *International Electrotechnical Commision*.

forem selecionadas, quanto mais se desvia das boas práticas em um modelo, é menos provável que se esteja melhorando os problemas na organização (KULPA, 2003).

De acordo com o explicitado em (SEI, 2002a; SEI, 2002b; SEI, 2002d; SEI, 2002f), um modelo é uma representação simplificada do mundo e os modelos de maturidade e capacidade, contêm os elementos essenciais para processos efetivos para uma ou mais classes de conhecimento.

Nas seções a seguir, são apresentados os modelos CMMI e as normas ISO 12207 e ISO 15504, padrões esses que são referência para a comunidade de engenharia de software no que concerne à melhoria da qualidade de processos.

2.1 MODELOS DE QUALIDADE DE PROCESSO DE SOFTWARE

O objetivo do SEI (*Software Engineering Institute*) é prover uma liderança na promoção do estado da prática de engenharia de software visando à melhoria da qualidade de sistemas que dependem de software e, o propósito do Programa de Processo de Software do SEI é fornecer uma direção quanto à assistência às organizações de software para o desenvolvimento e melhoria contínua de suas capacidades no que tange a identificação, adoção e utilização de gerenciamento correto e práticas técnicas. Essas práticas compreendem o processo de engenharia de software disciplinado, bem definido e medido efetivamente com o objetivo de fornecer software de qualidade que reúna custo e cronograma confiáveis.

Para Curtis (2000), a década de melhoria de processo de software acabou. Embora muitas organizações de software tenham aprendido a implementar programas bem sucedidos de melhoria de processo durante a década de 90, outras se esforçaram sem fazer melhorias duradouras. Essas empresas atrasadas gastarão a próxima década tentando sair do atraso

enquanto os líderes se responsabilizam por novos desafios. Os desafios de processo na próxima década incluem integração, harmonização e aceleração de processo.

A integração de modelos de maturidade é uma das preocupações do SEI, o qual está envolvido no desenvolvimento de um número de outros CMMs que foram inspirados no sucesso do CMM para software. Esses modelos, de acordo com Paulk *et al* (1996) cobrem tópicos como engenharia de sistema, pessoas, aquisição de software e sistemas fidedignos. Ao mesmo tempo, em que há uma preocupação com a proliferação em potencial de modelos de maturidade, há também, o reconhecimento de que a comunidade de usuários está crescendo. Diante disso, um grupo tarefa do SEI foi criado para monitorar e integrar os esforços dos diferentes modelos de maturidade do SEI de modo apropriado e também, para desenvolver um critério de integração de modelo de maturidade para ser utilizado no desenvolvimento de modelos de maturidade.

A década de melhoria de processo afetou não somente desenvolvedores como também clientes. Os anos 90 começaram com o Departamento de Defesa dos Estados Unidos da América (EUA) avaliando seus licitantes quanto à maturidade de seus processos e terminou aumentando o número de clientes avaliando o processo de desenvolvimento de seus fornecedores e *out-sources*. À medida que os desenvolvedores amadureceram os clientes freqüentemente realizaram seus próprios processos não maduros de aquisição, atrapalhando a eficiência de seus fornecedores. Conseqüentemente, os clientes começaram a melhorar seus processos de aquisição de modo que pudessem trabalhar mais eficientemente com desenvolvedores maduros, à medida que os clientes continuaram a amadurecer, passaram a querer integrar seus processos com os processos de seus fornecedores de software. Assim, o novo desafio para os desenvolvedores será criar processos flexíveis que possam ser adaptados para integração com processos de aquisição de diferentes clientes (CURTIS, 2000). Um importante desafio técnico envolverá a detecção de inconsistências entre processos, incorporados a partir de diferentes organizações.

Para Curtis (2000), as organizações de software encontram-se por si mesmas complicadas em um emaranhado de padrões e modelos de melhoria, um problema que aflige muitos segmentos da indústria. Muitos desses modelos compartilham ampla sobreposição de conteúdo. As

organizações de software irão querer mostrar como seus vários processos de desenvolvimento se relacionam com processos similares contidos em vários padrões. Então, os desenvolvedores necessitarão harmonizar seus processos de desenvolvimento com elementos similares, compartilhados entre vários padrões de processo, a fim de investigar como os seus processos satisfazem esses padrões.

O desafio para organizações de sistemas na próxima década será integrar atividades de melhoria entre as disciplinas de engenharia, de modo que possam integrar ao invés de somente coordenar seus processos de desenvolvimento de sistemas. O desafio relacionado à integração de processo vem de várias formas. À medida que as organizações de software começaram a melhorar seus processos, a engenharia de sistemas e outras organizações de desenvolvimento começaram a envidar esforços similares para melhoria.

Outras disciplinas de engenharia criaram seus próprios modelos de melhoria, geralmente orientados pelo CMM. Contudo, organizações freqüentemente se esforçaram tentando integrar melhorias diretamente de múltiplos modelos desprovidos de arquiteturas comuns. O CMMI representa uma tentativa para gerenciar essas dificuldades de integração (SEI, 2002a; SEI, 2002b; SEI, 2002c; SEI, 2002d; SEI, 2002f).

Atualmente as organizações desejam distribuir produtos melhores, mais baratos e com mais rapidez. Ao mesmo tempo, atuando em ambientes com alta tecnologia, quase todas as organizações resolveram por si mesmas construir produtos cada vez mais complexos. Geralmente, uma única empresa não desenvolve todos os componentes que compõem um produto. Mais comumente, alguns componentes são construídos pela própria organização e outros são adquiridos, para então serem integrados em um produto final. Dessa forma, as organizações devem ser capazes de gerenciar e controlar esse desenvolvimento complexo de produto bem como sua manutenção. Muitas organizações, entretanto, provêm a si mesmas no comércio de software, de acordo com Chrissis (2003).

Organizações que não são tipicamente empresas de software descobrem que muitos de seus negócios necessitam de software. O software é o que, freqüentemente, as diferencia de suas

concorrentes no mercado competitivo. Os problemas enfrentados por essas organizações atualmente envolvem tanto a engenharia de software como a engenharia de sistemas e, estão se tornando cada vez mais, um aspecto crítico de seus negócios. Essencialmente, essas organizações são desenvolvedores de produto que necessitam de um meio para gerenciar uma abordagem integrada para seus softwares e engenharia de sistemas como parte da obtenção de seus objetivos de negócio (CHRISSIS, 2003).

No mercado atual, existem modelos de maturidade, metodologias e diretrizes que podem auxiliar uma organização a melhorar o modo como realiza seus negócios. Contudo, muitas abordagens de melhoria disponíveis concentram-se em uma parte específica de negócio e não adotam uma abordagem sistemática para os problemas enfrentados pela maioria das organizações. Por exemplo, existem vários modelos de maturidade tais como o CMM para software do SEI, cujo foco é a melhoria do software e o *Eletronic Industries Alliance's* (EIA's) ou *Systems Engineering Capability Model* (SECM), o qual focaliza a engenharia de sistemas. Devido ao fato de focarem a melhoria de uma determinada área de negócio, esses modelos têm perpetuado os problemas e obstáculos existentes nas organizações.

2.1.1 O Modelo CMMI

O CMMI fornece a oportunidade para evitar ou eliminar esses problemas e obstáculos entre de modelos integrados que transcendem os aspectos cobertos em cada um desses modelos (disciplinas). O CMMI consiste de boas práticas destinadas ao desenvolvimento e manutenção de produtos e serviços, abrangendo o ciclo de vida do produto desde a concepção até sua distribuição e manutenção. Nesse modelo, há uma ênfase nas engenharias de sistema e de software e a integração necessária para construir e manter um produto completo. O objetivo fundamental do CMMI é a integração de processos para a melhoria de produtos (CHRISSIS, 2003).

Desde 1991, os CMM's têm sido desenvolvidos para uma ampla gama de disciplinas. Alguns dos mais conhecidos incluem modelos para engenharia de sistemas, engenharia de software, aquisição de software, desenvolvimento e gerenciamento de força de trabalho e desenvolvimento de processo e produto integrado.

Embora esses modelos tenham se mostrado úteis para muitas organizações, o uso de múltiplos modelos tem sido problemático. Muitas organizações gostariam de focar seus esforços de melhoria entre disciplinas. Contudo, as diferenças entre esses modelos específicos à disciplina, incluindo sua arquitetura, conteúdo e abordagem, têm limitado a capacidade dessas organizações para focar suas melhorias de forma bem sucedida. Além do mais, a adoção de múltiplos modelos que não estão integrados em uma organização, é custosa em termos de atividades de treinamento, avaliação e melhoria (CHRISSIS, 2003).

O projeto do CMMI foi formado para solucionar o problema de usar múltiplos CMM's e o objetivo da equipe que desenvolveu o CMMI era combinar três modelos fonte (Chrissis, 2003; SEI, 2002a), a saber:

- *Capability Maturity Model for Software* (SW-CMM) v2.0, *Draft* C.
- *Systems Engineering Capability Model* (SECM), também conhecido como *Electronic Industries Alliance* 731 (EIA731).
- *Integrated Product Development Capability Maturity Model* (IPD-CMM) v. 0.98.

A combinação desses modelos em um único *framework* de melhoria foi concebida para uso por organizações em busca da melhoria de processo de toda a organização.

Esses três modelos fonte foram selecionados devido à sua freqüente e extensa adoção nas comunidades de engenharia de sistema e de software e devido às suas diferentes abordagens para melhoria de processos em uma organização. A partir desses modelos, a equipe de produto do CMMI desenvolveu um conjunto coeso de modelos integrados que pode ser adotado tanto pelas organizações que atualmente usam os modelos fonte, como por organizações que adotaram recentemente o conceito CMM.

A Figura 1 retrata os modelos - cuja definição está em (SEI, 2002a) - que influenciaram diretamente o desenvolvimento do CMMI (KULPA, 2003), a saber:

- *Systems Engineering Capability Maturity Model* (SE-CMM). Avalia a maturidade da organização em seus processos de engenharia de sistemas, concebidos como algo maior que o software. Um sistema inclui o hardware, o software e quaisquer outros elementos que participam do produto completo.
- *Software Acquisition Capability Maturity Model* (SA-CMM). Usado para avaliar a maturidade de uma organização em seus processos de seleção, compra e instalação de software desenvolvido por terceiros.
- *Integrated Product Development Team Model* (IPD-CMM). Mais abrangente que o SE-CMM, inclui também outros processos necessários à produção e suporte ao produto tais como suporte ao usuário, processos de fabricação etc.
- *Systems Engineering Capability Assessment Model* (SECAM).
- *Systems Engineering Capability Model* (SECM).

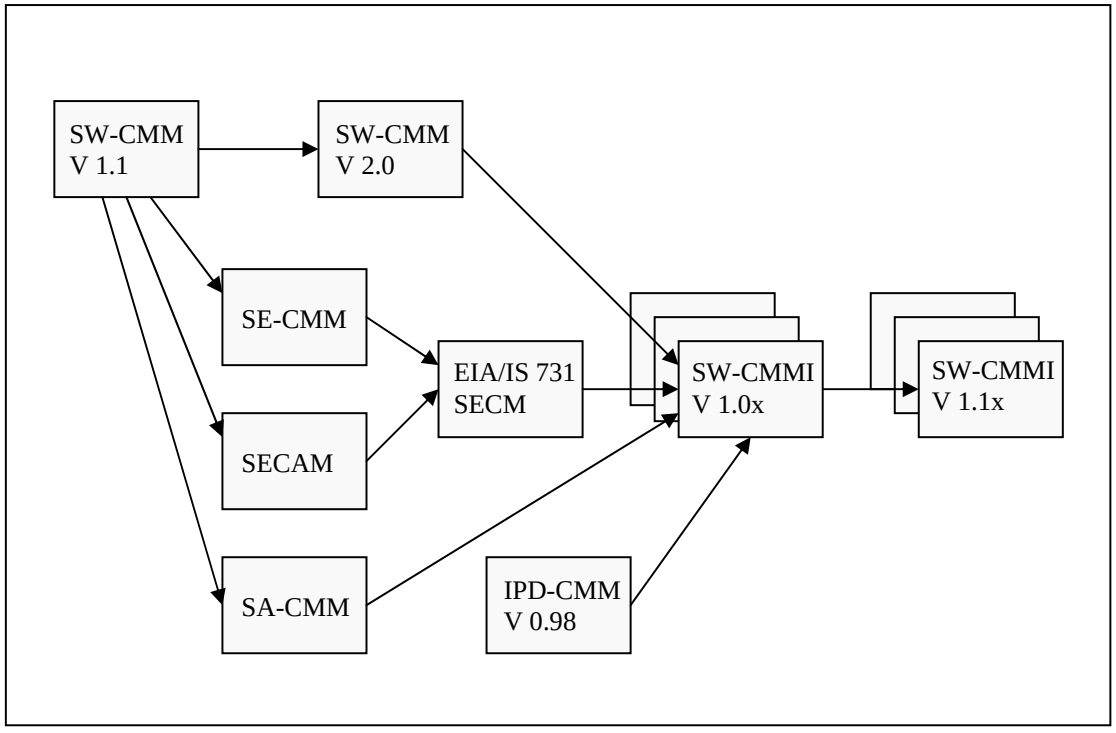


Figura 1. Modelos que influenciaram diretamente o CMMI (KULPA, 2003)

2.1.1.1 Componentes do modelo CMMI

Os componentes das representações, contínua e por estágio, são: áreas de processo, objetivos específicos, práticas específicas, objetivos genéricos, práticas genéricas, produtos típicos de trabalho, sub-práticas, notas, amplificações de disciplinas, elaborações de práticas genéricas e referências (SEI, 2002b).

De acordo com Kulpa (2003), o CMMI é estruturado com segue:

- Níveis de maturidade (representação por estágio) ou níveis de capacidade (representação contínua).
- Áreas de processo.
- Objetivos: genéricos e específicos.
- *Features* comuns.
- Práticas: genéricas e específicas.

Áreas de processo. Uma área de processo é um *cluster* de boas práticas relacionadas em uma área que, quando implementadas coletivamente, satisfazem um conjunto de objetivos considerados importantes para tornar a melhoria significativa naquela área. Todas as áreas de processo são comuns a ambas as representações, sendo que na representação contínua, as áreas de processo são organizadas por categorias de áreas de processo (SEI, 2002a).

Objetivos específicos. Os objetivos específicos se aplicam a uma área de processo e definem a única característica que descreve o que deve ser implementado para satisfazer a área de processo. Os objetivos específicos são componentes requeridos do modelo e são usados em avaliações para ajudar a determinar se uma área de processo é satisfeita. Podem existir práticas específicas em diferentes níveis de capacidade mapeadas para o mesmo objetivo. Contudo, todo objetivo tem, pelo menos uma prática no nível 1 de capacidade mapeada para ele.

Práticas específicas. Uma prática específica é uma atividade considerada importante para a obtenção do objetivo específico associado. As práticas específicas descrevem as atividades esperadas para resultar na consecução dos objetivos específicos de uma área de processo. Cada prática específica está associada com um nível de capacidade. As práticas específicas são componentes esperados do modelo.

A representação contínua tem mais práticas específicas que a representação por estágio porque tem dois tipos de práticas específicas: básicas e avançadas, enquanto que a representação por estágio tem somente um tipo de prática específica (KULPA, 2003).

Práticas base. Na representação contínua, todas as práticas específicas, com um nível de capacidade 1 são denominadas ‘práticas base’ e todas as práticas específicas com um nível de capacidade 2 ou maior são denominadas ‘práticas avançadas’.

Produtos típicos de trabalho. São componentes informativos do modelo que fornecem exemplos de saída a partir de uma prática específica ou genérica. Esses exemplos são denominados ‘produtos típicos de trabalho’ porque freqüentemente são outros artefatos efetivos, porém, não listados.

Sub-práticas. São descrições detalhadas que fornecem diretrizes para interpretar práticas específicas e genéricas. Podem ser expressas como prescritivas, mas são atualmente, componentes informativos nos modelos CMMI, usadas somente para fornecer idéias que podem ser úteis para a melhoria de processo.

Amplificações de disciplinas. São componentes informativos do modelo que contêm informações relevantes para uma disciplina em particular e estão associadas com as práticas específicas.

Objetivos genéricos. Cada nível da capacidade (1-5) tem somente um objetivo genérico que descreve a institucionalização que a organização deve atingir em determinado nível de capacidade. Assim, existem 5 objetivos genéricos, cada um aparece em cada área de processo.

A obtenção de um objetivo genérico em uma área de processo, significa controle melhorado no planejamento e implementação do processo, associado com aquela área de processo indicando se esses processos são efetivos, repetíveis e permanentes. Objetivos genéricos são componentes requeridos do modelo e são usados na avaliação para determinar se uma área de processo é satisfeita. Somente o título e o enunciado do objetivo genérico aparecem nas áreas de processo.

Práticas genéricas. Práticas genéricas fornecem institucionalização para assegurar que os processos associados com a área de processo serão efetivos, repetíveis e permanentes. Práticas genéricas são categorizadas por nível de capacidade e são componentes esperados nos modelos CMMI.

Na representação contínua, as práticas genéricas existem para todos os níveis de capacidade, enquanto que na representação por estágio aparecem práticas genéricas apenas nos níveis 2 e 3, não existindo, portanto, práticas genéricas nos níveis 1, 4 e 5. Na representação contínua, cada prática genérica mapeia um objetivo genérico. Somente o título da prática genérica, o enunciado e as elaborações aparecem nas áreas de processo (KULPA, 2003).

Elaboração de práticas genéricas. As elaborações de práticas genéricas são componentes informativos do modelo, que aparecem em cada área de processo, para fornecer diretrizes sobre como as práticas deveriam ser aplicadas unicamente para áreas de processo.

Referências. Referências são componentes informativos do modelo que direcionam o usuário para informações adicionais ou mais detalhadas nas áreas de processo relacionadas.

2.1.1.2 Áreas de processo

As áreas de processo freqüentemente interagem e afetam umas às outras independentemente de seus grupos definidos. Por exemplo, a área de processo Análise de Decisão e Resolução fornece práticas específicas, tratando as avaliações formais que são usadas na área de processo Solução Técnica, para selecionar uma solução técnica a partir de soluções alternativas (KULPA, 2003). A Tabela 1 contempla as áreas de processo, suas categorias e níveis de maturidade associados.

Tabela 1. Áreas de processo, categorias e níveis de maturidade associados (CHRISSIS, 2003)

Área de Processo	**Categoria**	**Nível**
Análise Causal e Resolução	Suporte	5
Gerenciamento de Configuração	Suporte	2
Análise de Decisão e Resolução	Suporte	3
Gerenciamento Integrado de Projeto	Gerenciamento de Projeto	3
Gerenciamento Integrado de Fornecedor	Gerenciamento de Projeto	3
Equipe Integrada	Gerenciamento de Projeto	3
Medição e Análise	Suporte	2
Ambiente Organizacional para Integração	Suporte	3
Inovação Organizacional e Distribuição	Gerenciamento de Processo	5
Definição do Processo Organizacional	Gerenciamento de Processo	3
Foco no Processo Organizacional	Gerenciamento de Processo	3
Desempenho do Processo Organizacional	Gerenciamento de Processo	4
Treinamento Organizacional	Gerenciamento de Processo	3
Integração de Produto	Engenharia	3
Monitoração e Controle de Projeto	Gerenciamento de Projeto	2
Planejamento de Projeto	Gerenciamento de Projeto	2
Garantia de Qualidade de Produto e de Processo	Suporte	2
Gerenciamento Quantitativo de Projeto	Gerenciamento de Projeto	4
Desenvolvimento de Requisitos	Engenharia	3
Gerenciamento de Requisitos	Engenharia	2
Gerenciamento de Riscos	Gerenciamento de Projeto	3
Gerenciamento de acordos com Fornecedores	Gerenciamento de Projeto	2
Solução Técnica	Engenharia	3
Validação	Engenharia	3
Verificação	Engenharia	3

Conforme pode ser observado, as áreas de processo são agrupadas em quatro categorias:

Gerenciamento de processo. As áreas de processo da categoria Gerenciamento de Processo contêm atividades cruzando projetos relacionados para definição, planejamento, recursos, desenvolvimento, implementação, monitoração, controle, avaliação e melhoria de processo.

As áreas básicas de processo da categoria Gerenciamento de Processo fornecem à organização uma capacidade básica para documentar e compartilhar as melhores práticas, avaliar processo organizacional e aprender entre toda a organização.

As áreas básicas de processo da categoria Gerenciamento de Processo, integram engenharia de software e processos de engenharia de sistemas em um cenário de melhoria de processo orientado a produto. A melhoria dos processos de desenvolvimento de produto tem como alvo os objetivos essenciais de negócio ao invés de disciplinas específicas.

Gerenciamento de projeto. As áreas de processo da categoria Gerenciamento de Projeto cobrem as atividades de gerenciamento relacionadas com planejamento, monitoração e controle de projeto.

As áreas básicas de processo da categoria Gerenciamento de Projeto tratam as atividades relacionadas com o estabelecimento e manutenção do plano de projeto, estabelecimento e manutenção de compromissos, monitoramento de progresso em contraposição ao plano, tomada de ações corretivas e gerenciamento de acordos com fornecedores.

Engenharia. As áreas de processo da categoria Engenharia cobrem as atividades de desenvolvimento e manutenção que são partilhadas pelas disciplinas de engenharia (ex. engenharia de sistemas e engenharia de software). As seis áreas de processo na área de processo da categoria Engenharia têm inter-relacionamentos inerentes. Esses inter-relacionamentos são provenientes da aplicação de processo de desenvolvimento de produto ao invés de processos de disciplinas específicos, tais como engenharia de sistema ou engenharia de software.

As áreas de processo da categoria Engenharia se aplicam ao desenvolvimento de qualquer produto ou serviço no domínio de engenharia de desenvolvimento (produtos de software, produtos de hardware, serviços ou processos).

Suporte. As áreas de processo da categoria Suporte cobrem as atividades que apóiam o desenvolvimento e a manutenção de produto. As áreas de processo da categoria Suporte tratam processos que são usados no contexto de execução de outros processos. Em geral, as áreas de processo da categoria Suporte, tratam processos que são destinados ao projeto e podem tratar processos que geralmente se aplicam mais à organização.

A área de processo Garantia da Qualidade de Produto e de Processo pode ser usada com todas as áreas de processo para fornecer uma avaliação do objetivo para os processos e artefatos descritos em todas as áreas de processo.

Os componentes do modelo CMMI descritos até então, são agrupados em três categorias, as quais são descritas a seguir conforme mostra a Figura 2, e refletem o modo como os componentes são interpretados (KULPA, 2003):

Requerido. Os componentes requeridos descrevem o que uma organização deve realizar para satisfazer uma área de processo. Essa realização deve ser visivelmente implantada nos processo de uma organização. Os componentes requeridos no CMMI são os objetivos específicos e os objetivos genéricos. Esses componentes devem ser alcançados pelos processos planejados e implementados de uma organização. Os componentes requeridos são essenciais para avaliar uma área de processo. A obtenção de um objetivo é usada nas avaliações como base para se decidir se a área de processo e a maturidade organizacional determinadas foram alcançadas e satisfeitas.

Somente a declaração (o enunciado) do objetivo específico ou genérico é um componente do modelo. O título de um objetivo específico ou genérico e qualquer nota associada com o objetivo é considerado componente informativo do modelo.

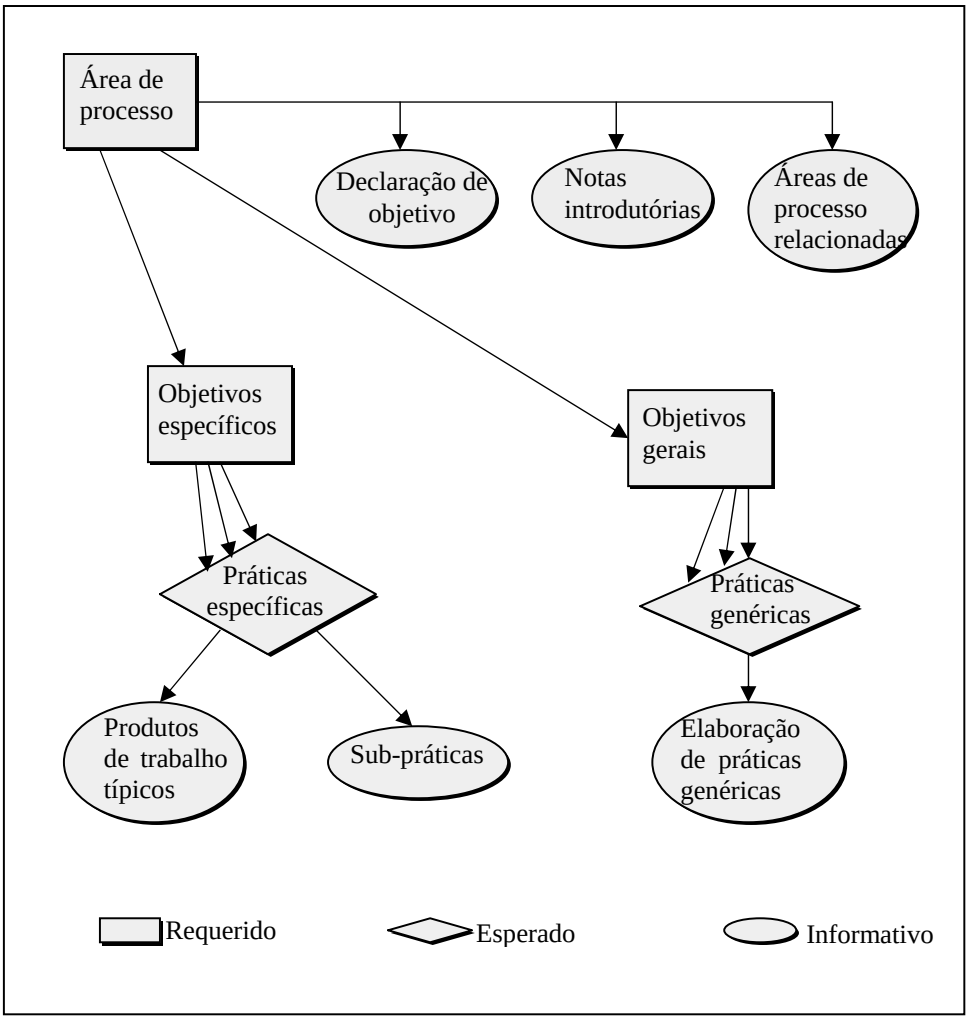


Figura 2. Componentes do modelo CMMI

Esperado. Os componentes esperados descrevem o que uma organização tipicamente irá implementar para alcançar um componente requerido. Os componentes esperados orientam aqueles componentes que implementam melhorias ou avaliam desempenho. Práticas específicas e práticas genéricas são componentes esperados do modelo. Os componentes esperados orientam as características comuns, 'implementando melhorias' e 'executando avaliações'. Espera-se também que as práticas tal como descritas ou as alternativas aceitáveis para elas, estejam presentes nos processos planejados e implementados da organização, antes que os objetivos possam ser considerados como alcançados.

Somente a declaração (o enunciado) da prática, é um componente esperado do modelo. São considerados componentes informativos do modelo, o título de uma prática e quaisquer notas relacionadas a tal prática.

Informativo. Os componentes informativos fornecem detalhes que auxiliam as organizações a conseguirem começar a pensar sobre como abordar componentes requeridos e esperados. Sub-práticas, produtos típicos de trabalho, amplificações de disciplinas, elaborações de práticas genéricas, títulos de objetivos e práticas, notas de objetivos e práticas e referências são componentes informativos do modelo que auxiliam os usuários do modelo, a entenderem os objetivos e as práticas e como os mesmos podem ser alcançados. Os componentes informativos fornecem detalhes que podem ajudar os usuários do modelo a começarem a pensar em como abordar objetivos e práticas.

Disciplinas são referenciadas pelas áreas de processo associadas com as mesmas e pelos componentes do modelo, denominados amplificação de disciplinas. Uma amplificação de disciplina é um componente do modelo que contém informações para uma determinada disciplina e essas informações se aplicam somente à melhoria do processo de engenharia de software escolhido (CHRISSIS, 2003).

Classes de conhecimento

O Objetivo do CMMI é fornecer um CMM que compreenda o desenvolvimento e manutenção de produtos e serviços, mas também forneça um *framework* extensível, de modo que novas classes de conhecimento possam ser adicionadas. Atualmente, quatro classes de conhecimento estão disponíveis para o planejamento da melhoria do processo usando CMMI, de acordo com Chrissis (2003):

Engenharia de sistemas. Cobre o desenvolvimento de sistemas inteiros, os quais podem ou não incluir software. Os engenheiros de sistemas concentram-se em transformar as necessidades, expectativas e as restrições do usuário em produtos, dando suporte a esses produtos durante todo o seu ciclo de vida. Nessa disciplina, amplificação de disciplina para engenharia de sistemas recebe ênfase especial (SEI, 2002b).

A melhoria dos processos de engenharia de sistemas requer a seleção das seguintes áreas de processo: **1**) Análise causal e resolução; **2**) Gerenciamento de configuração; **3**) Análise de decisão e resolução; **4**) Gerenciamento integrado de projeto (os dois primeiros objetivos específicos); **5**) Medição e análise; **6**) Inovação organizacional e desenvolvimento; **7**) Definição do processo organizacional; **8**) Foco no processo organizacional; **9**) Desempenho do processo organizacional; **10**) Treinamento organizacional; **11**) Integração de produto; **12**) Monitoramento e controle de projeto; **13**) Planejamento de projeto; **14**) Garantia da qualidade de produto e de processo; **15**) Gerenciamento quantitativo de projeto; **16**) Desenvolvimento de requisitos; **17**) Gerenciamento de requisitos; **18**) Gerenciamento de riscos; **19**) Gerenciamento de contrato com o fornecedor; **20**) Solução técnica; **21**) Validação; e **22**) Verificação.

Engenharia de software. Cobre o desenvolvimento de sistemas de software. Os engenheiros de software concentram-se em abordagens sistemáticas, disciplinadas e confiáveis, para o desenvolvimento, operação e manutenção de software. Para melhorar os processos de engenharia de software, é necessário escolher as mesmas áreas de processo listadas para engenharia de sistemas. A única diferença é que a amplificação de disciplinas para a engenharia de software recebe ênfase especial (SEI, 2002b).

Produto integrado e desenvolvimento de processo. É uma abordagem sistemática que obtém uma colaboração oportuna de *Stakeholders* relevantes à vida do produto para satisfazer as necessidades, expectativas e requisitos do usuário. Os processos para apoiar uma abordagem IPPD[5] são integrados com outros processos na organização. Se uma organização ou projeto escolhe IPPD, ela executa as boas práticas do IPPD concorrentemente com outras

[5] Do inglês: *Integrated Product and Process Development*.

boas práticas usadas para produzir produtos (aqueles relacionados à engenharia de sistemas). Isto é, se uma organização ou projeto deseja usar IPPD, a mesma deve selecionar uma ou mais disciplinas em adição ao IPPD (SEI, 2002b).

Se a melhoria ocorrer no produto integrado e desenvolvimento de processo, a escolha deverá dar-se a partir das mesmas áreas de processo listadas para engenharia de sistemas, com duas áreas de processo adicionais e boas práticas adicionais na área de processo Gerenciamento Integrado de Processo. As disciplinas para IPPD recebem ênfase especial. As áreas adicionais de processo são: Equipe Integrada e Ambiente Organizacional para Integração.

Gerência de fornecedores (*Supplier sourcing*). À medida que os esforços tornam-se mais complexos, gerenciadores de projeto podem usar fornecedores para executar funções ou adicionar modificações a produtos que são especificamente necessários para o projeto. Quando essas atividades são críticas, o projeto se beneficia desde análises fonte obtidas e, desde atividades de monitoramento de fornecedor antes que o produto seja liberado. Sob essas circunstâncias, a disciplina, Fornecedor Fonte cobre a aquisição de produtos a partir dos fornecedores. Similar às boas práticas do IPPD, as boas práticas do fornecedor fonte devem ser selecionadas em conjunção com as boas práticas usadas para produzir produtos.

Para Chrissis (2003), a única distinção entre modelos CMMI para engenharia de sistemas e engenharia de software é o tipo de amplificação da disciplina incluída. Essa similaridade foi uma decisão intencional tomada durante o desenvolvimento do CMMI, uma vez que o mesmo concentra-se no desenvolvimento do produto, melhorando a ambos: engenharia de sistemas e funções de engenharia de software com uma abordagem integrada.

Todos os modelos fonte para o CMMI são considerados modelos de capacidade e maturidade; contudo, cada um tem uma abordagem diferente. A revisão e análise de cada um dos modelos fonte conduzem a dois tipos de abordagem para apresentar modelos de capacidade e maturidade. Esses tipos de abordagem têm sido rotulados como ‘representações’ na comunidade de melhoria de processo. “Uma representação reflete a organização, o uso e a apresentação dos componentes em um modelo” (CHRISSIS, 2003).

Entende-se por maturidade da capacitação, o quanto o processo é capaz de assegurar o gerenciamento do projeto, a qualidade dos produtos gerados, a adaptação do processo às características específicas das empresas e dos projetos e, também, o quanto o processo é capaz de ser continuamente aprimorado. Para se entender quais melhorias são mais críticas para uma organização é necessário entender, a diferença entre organizações maduras e não maduras de software. Na Figura 3, estão descritas as principais diferenças entre tais organizações (CHRISSIS, 2003).

Organizações maduras	**Organizações não maduras**
Habilidades para gerenciar os processos de desenvolvimento e manutenção de software.	Processos de software são, geralmente, improvisados.
Execução das atividades de acordo com o processo planejado.	Processo de software não é rigorosamente seguido ou mantido.
Resultados esperados quanto a custo, orçamento, funcionalidade e qualidade do produto são, normalmente, obtidos.	Freqüentemente excedem cronogramas e orçamentos, porque não são baseados em estimativas realistas.
Definições de processos são atualizadas quando necessário. Há infra-estrutura para apoiar os processos.	Não há objetivos que formam a base para julgar a qualidade do produto ou solucionar problemas de processos ou produtos.
Melhorias são desenvolvidas com testes piloto controlados e análise de custo e benefícios.	Pouco entendimento sobre como os passos do processo afetam a qualidade.
Cronograma e orçamento são baseados em desempenho histórico e são realistas.	Difícil de predizer a qualidade do produto.
Funções e responsabilidades dentro do processo são claramente definidas no projeto e na organização.	Atividades para assegurar a qualidade, como teste e revisões, são freqüentemente minimizadas ou descartadas.
Gerentes monitoram a qualidade dos produtos e dos processos que os produzem.	O cliente tem pouco discernimento sobre o produto até a liberação do mesmo.
Há um objetivo quantitativo base para avaliar a qualidade do produto e analisar os problemas com o produto e com o processo.	Por serem organizações reacionárias, seus gerentes normalmente estão focados em solucionar crises imediatas.
Seguem um processo disciplinado consistente, porque todos os participantes entendem o valor do que fazem e existe uma infra-estrutura necessária para apoiar o processo.	Quando prazos rigorosos são impostos, podem comprometer a funcionalidade e qualidade do produto, para atender ao cronograma.

Figura 3. Diferenças entre as organizações maduras e não maduras de software.

O CMMI é uma estrutura (*framework*) que descreve os principais elementos de um processo de software efetivo. Descreve os estágios a partir dos quais as organizações de software evoluem quando definem, implementam, medem, controlam e melhoram seus processos de software e, fornece uma diretriz para a seleção de estratégias de melhoria de processos,

permitindo a determinação da capacitação dos processos em andamento e a conseqüente identificação das questões mais críticas para a melhoria de processo e qualidade de software.

Todos os modelos de capacidade e maturidade têm áreas de processo que são definidas por níveis. Dois dos modelos fonte usam outros termos para conceituar uma área de processo. O CMM para software usa o termo áreas chave de processo e o SECM usa o termo áreas foco.

Os níveis são usados no CMMI para descrever um caminho evolucionário recomendado para uma organização que deseja melhorar os processos usados para desenvolver e manter seus produtos e serviços.

O CMMI suporta dois tipos de caminhos de melhoria. O primeiro caminho possibilita que uma organização consiga a melhoria incremental de processos correspondentes a uma área de processo em particular (ou áreas de processo) selecionada pela organização. O segundo caminho possibilita que uma organização melhore de maneira incremental, um conjunto de processos relacionados tratando sucessivos conjuntos de áreas de processo.

Esses dois caminhos de melhoria estão relacionados a dois tipos de níveis que correspondem a dois tipos de representação. A representação contínua (para a qual se utiliza o termo, nível de capacidade) e a representação por estágio (para a qual se utiliza o termo, nível da maturidade).

Não importa a representação escolhida, o conceito de nível é o mesmo. Os níveis caracterizam a melhoria de um estado indefinido para um estado que usa informação quantitativa para determinar e gerenciar melhorias, as melhorias necessárias para se alcançar os objetivos de negócio da organização (CHRISSIS, 2003).

Para atingir um determinado nível, uma organização deve satisfazer todos os objetivos apropriados de uma área de processo ou um conjunto de áreas de processo, marcado para melhoria, indiferente se é um nível de capacidade ou de maturidade.

A **representação por estágio** é a abordagem utilizada no CMM para software. É uma abordagem que usa conjuntos pré-definidos de áreas de processo para definir um caminho de melhoria para uma organização. Esse caminho de melhoria é descrito por um componente do modelo, denominado nível de maturidade. Um nível de maturidade é um platô evolucionário bem definido em direção à obtenção de processos organizacionais melhorados. Os níveis de capacidade se aplicam para a obtenção da melhoria de processo de uma organização em uma área particular de processo. Esses níveis são um meio para a melhoria incremental dos processos correspondentes a uma determinada área de processo (CHRISSIS, 2003).

A **representação contínua** é a abordagem usada no SECM e no IPPD-CMM, a qual possibilita à organização selecionar uma área chave de processo específica e proceder a uma melhoria relativa na área selecionada. A representação contínua usa níveis de capacidade para caracterizar melhoria relativa para uma área de processo individual. Os níveis de maturidade se aplicam para a obtenção de melhoria de processo em uma organização em múltiplas áreas de processo. Esses níveis são um meio de prever as saídas gerais do próximo projeto empreendido.

A Tabela 2 compara os níveis de capacidade com os níveis de maturidade. Cabe ressaltar que os nomes de quatro dos níveis são os mesmos, em ambas as representações. A diferença é que não há nível 0 (zero) de maturidade para a representação por estágio e, no nível um, o nível de capacidade é “executado”, enquanto o nível de maturidade é “inicial”, portanto, o ponto de início é diferente para as duas representações (CHRISSIS, 2003).

Tabela 2. Comparação entre os níveis de capacidade e de maturidade

Níveis	**Representação contínua**	**Representação por estágio**
	Níveis de capacidade	**Níveis de maturidade**
0	Incompleto	
1	Executado	Inicial
2	Gerenciado	Gerenciado
3	Definido	Definido
4	Quantitativamente gerenciado	Quantitativamente gerenciado
5	Em otimização	Em otimização

Vale lembrar, que o fato de os níveis de maturidade de 2 a 5 usarem os mesmos termos que os níveis de capacidade de 2 a 5, é intencional, segundo Chrissis (2003), porque os conceitos de níveis de maturidade e de capacidade são complementares, uma vez que os níveis de maturidade são usados para caracterizar melhoria organizacional relativa a um conjunto de áreas de processo e os níveis de capacidade caracterizam melhoria organizacional relativa a uma área de processo em particular.

Dado que cada representação tem vantagens sobre a outra, algumas organizações usam ambas para tratar necessidades particulares em vários momentos no seu programa de melhoria.

2.1.1.3 Representação contínua

A representação contínua oferece uma abordagem flexível para melhoria de processo. Uma organização pode escolher melhor o desempenho de um único ponto de dúvida relacionado ao processo ou pode trabalhar em várias áreas que estão estritamente alinhadas com os objetivos de negócio da organização. A representação contínua também possibilita a uma organização a melhoria de diferentes processos em velocidades distintas. Existem algumas limitações nas escolhas de uma organização devido às dependências entre algumas áreas de processo (SEI, 2002c).

A representação contínua (SEI, 2002a; SEI, 2002b) usa seis níveis de capacidade, perfil de capacidade, *target staging* e *equivalent staging* como princípios para os componentes de modelo. A representação contínua agrupa áreas de processo por categorias de afinidade e designa níveis de capacidade para melhoria de processo dentro de cada área de processo.

Perfil de capacidade. Na representação contínua, um perfil de nível de capacidade é uma lista de áreas de processo e seus níveis de capacidade correspondentes. Este perfil é um meio para a organização rastrear seus níveis de capacidade por área de processo.

O perfil é um perfil de realização, quando representa o progresso de cada área de processo da organização, enquanto ascende aos níveis de capacidade. Alternativamente, o perfil é um perfil marcado, quando representa os objetivos de melhoria de processo da organização.

Um perfil de realização, quando comparado com um perfil marcado, permite que seja possível não apenas o acompanhamento do progresso de melhoria de processos da organização, mas também demonstrar o progresso da organização para gerenciamento. O perfil de capacidade representa os caminhos de melhoria de processo ilustrando a evolução da melhoria para cada área de processo.

Equivalent staging. É usado para relacionar os níveis de maturidade das áreas de processo aos níveis de maturidade da representação por estágio. É uma maneira de comparar resultados obtidos com o uso de representação contínua, com aqueles usados na representação por estágio. Ou seja, possibilita que os resultados de avaliações realizadas com o uso de representação contínua sejam traduzidos em níveis de maturidade. Algumas vezes, pode ser desejável converter um perfil de realização para uma organização em um nível de maturidade. Essa conversão se torna possível pelo *equivalent staging* (KULPA, 2003).

Target staging. É uma seqüência de perfis marcados que descrevem o caminho de melhoria de processo a ser seguido por uma organização. Quando cria perfis marcados, a organização deve atentar para as dependências entre práticas genéricas e áreas de processo. Quando uma prática genérica é dependente de determinada área de processo, seja para executar uma prática genérica ou para fornecer um produto pré-requisito, a prática genérica será ineficiente (ineficaz) quando a área de processo não está implementada. Quando um perfil marcado é escolhido com essas dependências já consideradas o perfil marcado é admissível (KULPA, 2003).

Todos os modelos CMMI com representação contínua refletem níveis de capacidade em seus projetos e conteúdos. Os níveis de capacidade consistem de práticas genéricas e específicas relacionadas a uma área de processo que pode melhorar os processos da organização associados com tal área de processo. Assim que os objetivos específicos e genéricos de uma

área de processo em um nível particular de capacidade são satisfeitos, os benefícios de melhoria de processo são alcançados (KULPA, 2003).

Os níveis de capacidade se concentram no crescimento da capacidade da organização para executar, controlar e melhorar seu desempenho em uma área de processo. Possibilitam o acompanhamento e a avaliação e demonstram o progresso da organização à medida que melhoram os processos associados com uma área de processo. Fornecem uma ordem recomendada para a abordagem de melhoria de processo.

Dentro de cada área de processo, primeiro são listados os objetivos específicos e as práticas específicas e, em seguida, os objetivos genéricos e as práticas genéricas. A representação contínua usa os objetivos genéricos para organizar as práticas genéricas (SEI, 2002a; SEI, 2002b).

Os níveis de capacidade são usados para medir o caminho de melhoria em cada área de processo, a partir de um processo não executado até um processo em otimização. Por exemplo, uma organização pode aspirar à obtenção do nível 2 de capacidade em uma área de processo e o nível 4 em outra. À medida que os processos da organização atingem um nível de capacidade ela direciona sua visão para o próximo nível de capacidade, dentro da mesma área de processo ou decide expandir seu escopo e criar o mesmo nível de capacidade entre um amplo número de áreas de processo.

Se os processos da organização que necessitam melhoria são conhecidos e, se há um entendimento das dependências entre as áreas de processo descritas no CMMI, a representação contínua poderia ser uma boa escolha para a organização, segundo a visão de Chrissis (2003).

A representação contínua está relacionada tanto com a seleção de uma área de processo em particular como com a granularidade de melhoria da área de processo em questão (isto é, melhorar não apenas o ‘que se faz’ mas também o ‘como se faz’). Nesse contexto, se os

objetivos específicos são plenamente executados ou estão incompletos é importante. Por isso, o nome ‘incompleto é atribuído ao ponto inicial na representação contínua.

A Figura 4 ilustra os componentes de um modelo CMMI na representação contínua.

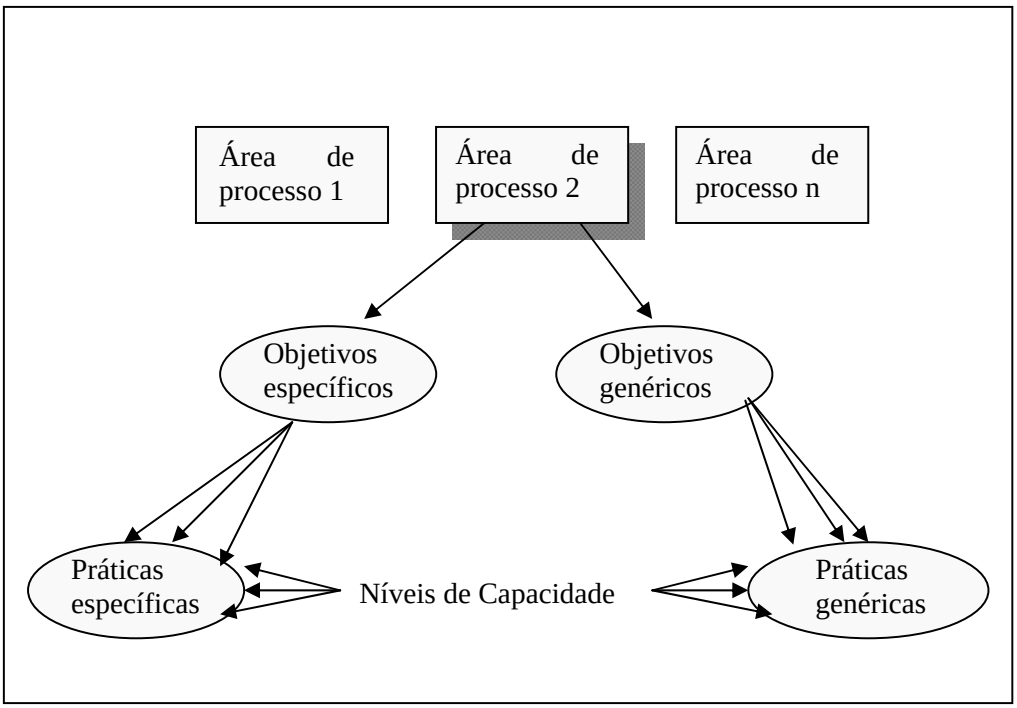


Figura 4. Estrutura da representação contínua (SEI, 2002a)

Conforme mostra a Figura 4, os objetivos específicos organizam as práticas específicas e os objetivos genéricos organizam as práticas genéricas. Cada prática específica e genérica corresponde a um nível de capacidade. Objetivos específicos e práticas específicas se aplicam à múltiplas áreas de processo. Objetivos genéricos e práticas genéricas definem a seqüência de níveis de capacidade que representam melhorias na implementação e efetividade de todos os processos escolhidos para melhoria. Os modelos CMMI são projetados para descrever níveis de processo. Na representação contínua, os níveis de capacidade fornecem uma ordem recomendada para a abordagem de melhoria de processo em cada área de processo. Ao mesmo tempo, a representação contínua permite alguma flexibilidade para a ordem, na qual as áreas de processo são tratadas. Esta representação se concentra nas melhores práticas que a

organização pode usar para melhorar processos nas áreas de processo escolhidas (SEI, 2002b).

Antes de começar a usar o modelo CMMI para melhoria de processo, é necessário mapear os processos da organização para as áreas de processo do CMMI. Esse mapeamento permite o controle da melhoria de processo na organização, auxiliando a organização a rastrear (acompanhar) o nível de conformação da organização ao modelo CMMI que se está usando. Contudo, não é intenção que haja um mapeamento uma a uma, de todas as áreas de processo do CMMI com os processos da organização (KULPA, 2003).

O seis níveis de capacidade, de acordo com o estabelecido em Kulpa (2003), designados pelos números de 0 (zero) a 5 (cinco) são:

Nível de capacidade 0: Incompleto. Um ‘processo incompleto’ é um processo que não foi executado ou foi parcialmente executado. Um ou mais dos objetivos específicos da área de processo não foram alcançados e não existe nenhum objetivo genérico para este nível, uma vez que não há razão para institucionalizar um processo parcialmente executado.

É importante, nesse ponto, explicitar o significado de implementação e institucionalização uma vez que ambos encerram conceitos distintos. Segundo Kulpa (2003) a implementação (grifo nosso) é simplesmente a execução de uma tarefa dentro de uma área de processo. Uma tarefa é realizada de acordo com um processo, mas a ação realizada ou o processo em questão pode não estar totalmente inserido na cultura da organização. Uma área de processo é uma área definida pelo CMMI como uma área que a organização deve focar. É um conjunto de tarefas relacionadas que lidam com uma área de interesse. A institucionalização (grifo nosso) segundo a mesma autora, é o resultado da implementação do processo repetidas vezes. O processo se torna totalmente integrado na organização, ou seja, o processo continuará, ainda que seus criadores deixem a organização. Há uma infra-estrutura para apoiar esse modo de realizar negócios, a qual fornece suporte para o processo e para os indivíduos seguirem-no.

Nível de capacidade 1: Executado. Um ‘processo executado’ é um processo que satisfaz os objetivos específicos da área de processo. Suporta e possibilita o trabalho necessário para produzir artefatos de saída identificados, usando artefatos de entrada identificados.

Uma distinção crítica entre um processo incompleto e um processo executado é que um processo executado satisfaz todos os objetivos específicos de uma área de processo.

Nível de capacidade 2: Gerenciado. Um ‘processo gerenciado’ é um processo executado que tem a infra-estrutura básica apropriada para apoiar o processo. É planejado e executado de acordo com uma política; emprega profissionais qualificados tendo recursos adequados para produzir resultados controlados; envolve *stakeholders* relevantes; é monitorado, controlado e revisado e é avaliado quanto à aderência à sua descrição de processo.

O processo pode ser instanciado por um projeto individual, grupo ou função organizacional. O gerenciamento do processo está relacionado com a institucionalização da área de processo e com a obtenção de outros objetivos específicos definidos para o processo, tais como custo, cronograma e objetivos de qualidade.

Os objetivos para processo podem ser objetivos específicos para os processos individuais ou podem ser definidos para um escopo mais amplo (para um conjunto de processos) com processos individuais contribuindo para a obtenção desses objetivos, os quais podem ser revisados como parte das ações corretivas adotadas para o processo.

O controle provido por um processo gerenciado ajuda a assegurar que processo estabelecido é mantido durante os períodos problemáticos. Os requisitos e objetivos para o processo são estabelecidos. O estado dos artefatos e a distribuição de serviços são visíveis para gerenciamento em pontos definidos (marcos principais e término de tarefas principais). Compromissos são estabelecidos entre aqueles que realizam o trabalho e *stakeholders* relevantes e são revisados quando necessário. Artefatos são revisados entre os *stakeholdres* relevantes e são controlados. Artefatos e serviços satisfazem seus requisitos especificados.

Nível de capacidade 3: Definido. Um ‘processo definido’ é um processo gerenciado que é adaptado a partir de um conjunto de processos padrão da organização - de acordo com as diretrizes adaptadas da organização - e que coopera com produtos de serviço, medições e outras informações sobre melhoria de processo para a avaliação do processo organizacional.

O conjunto de processos padrão da organização - base do processo definido – é estabelecido e melhorado ao longo do tempo. Processos padrão descrevem os elementos fundamentais do processo que são esperados no processo definido. Os processos padrão também descrevem as relações (a disposição e as interfaces) entre esses elementos de processo. A infra-estrutura ao nível organizacional, para apoiar o uso atual e futuro do conjunto de processos padrão da organização, é estabelecida e melhorada ao longo do tempo.

Para Kulpa (2003), um processo definido expressa claramente: **1**) finalidade; **2**) entradas; **3**) critério de entrada; **4**) atividades; **5**) funções (papéis); **6**) medições; **7**) passos de verificação; **8**) saídas e; **9**) critérios de saída.

Nível de capacidade 4: Quantitativamente gerenciado. Um ‘processo quantitativamente gerenciado’ é um processo definido que é controlado usando estatística e outras técnicas quantitativas. Objetivos quantitativos para qualidade e desempenho de processo são estabelecidos e usados como critério no gerenciamento de processo. Qualidade e desempenho de processo são compreendidos em termos estatísticos e são gerenciados durante a vida do processo.

Os objetivos quantitativos são baseados na capacidade do conjunto de processos padrão da organização, nos objetivos de negócio da organização, nas necessidades do cliente, usuários finais, organização e implementadores de processo, sujeitos aos recursos disponíveis. As pessoas que executam o processo estão diretamente envolvidas no gerenciamento quantitativo do processo.

O gerenciamento quantitativo é realizado em todo o conjunto de processos que produz um produto ou fornece um serviço. Os sub-processos que contribuem significativamente com o

desempenho do processo como um todo são estatisticamente gerenciados. Para os sub-processos selecionados, medições detalhadas de desempenho de processo são coletadas e estatisticamente analisadas. Causas especiais de variação são identificadas e, quando apropriado, a origem da causa especial é tratada para prevenir ocorrências futuras. A qualidade e a medida de desempenho de processo são incorporadas no repositório de medição da organização para apoiar a tomada de decisão baseada em fatos (KULPA, 2003).

Uma distinção crítica entre um processo definido e um processo quantitativamente gerenciado é a previsibilidade do desempenho de processo. O termo ‘quantitativamente gerenciado’ implica o uso de estatística apropriada e outras técnicas quantitativas para gerenciar o desempenho de um ou mais sub-processos críticos de modo que o desempenho futuro do processo possa ser previsto. Um processo definido provê somente previsibilidade qualitativa (KULPA, 2003).

Nível de capacidade 5: Em otimização. Um ‘processo em otimização’ é um processo quantitativamente gerenciado que é alterado e adaptado para satisfazer os objetivos relevantes de negócio atuais e os projetados. Concentra-se na melhoria contínua do desempenho de processo tanto por meio de melhoria incremental como de melhoria tecnológica inovadora. Melhorias de processo que poderiam tratar a origem das causas de variação de processo e melhoria mensurável dos processos da organização são identificadas, avaliadas e apropriadamente desenvolvidas.

Essas melhorias são selecionadas com base em um entendimento quantitativo de sua contribuição esperada, para alcançar os objetivos de melhoria de processo da organização versus o custo e impacto para a organização. O desempenho de processo dos processos da organização é continuamente melhorado. As melhorias de processo tecnológicas inovadoras ou incrementais são sistematicamente gerenciadas e desenvolvidas na organização. Os efeitos das melhorias de processo desenvolvidas são medidos e avaliados, em contraposição aos objetivos da melhoria quantitativa de processo.

Uma distinção crítica entre um processo quantitativamente gerenciado e um processo em otimização é que o processo em otimização é melhorado continuamente com o tratamento de causas de variação comum do processo. Um processo quantitativamente gerenciado está relacionado com tratar causas especiais de variação de processo e fornecer estatística previsível para os resultados. Embora o processo possa produzir resultados previsíveis, os resultados podem ser insuficientes para alcançar os objetivos definidos. Em um processo que é otimizado, causas comuns de variação de processo são tratadas alterando aquele processo de uma maneira que irá conduzir a uma troca no significado ou um decréscimo na variação quando ela retorna à estabilidade. Essas mudanças têm como fim, a melhoria do desempenho do processo e a obtenção dos objetivos de melhoria dos processos definidos da organização. Uma causa comum de variação de processo é uma causa que é parte inerente ao processo e afeta o desempenho total do processo (KULPA, 2003).

Os níveis de capacidade das áreas de processo são obtidos por meio da aplicação de práticas genéricas ou alternativas adequadas. Atingir o nível de capacidade 1 para uma área de processo é equivalente a dizer que os objetivos específicos de uma área de processo foram alcançados (CHRISSIS, 2003).

Atingir o nível de capacidade 2 para uma área de processo é equivalente a dizer que há uma política que indica que o processo será executado. Há um plano e treinamento para executá-lo, recursos são providos, responsabilidades são estabelecidas e artefatos selecionados para a execução do processo são controlados e assim por diante. Em outras palavras, um processo no nível de capacidade 2 pode ser planejado e monitorado como qualquer projeto ou atividade de suporte.

Obter o nível de capacidade 3 para uma área de processo supõe que existe um processo organizacional padrão ou processos, associados com tal área de processo, o qual pode ser adaptado às necessidades dos projetos. Os processos na organização são mais consistentemente definidos e aplicados, porque são baseados nos processos-padrão da organização.

Alcançar o nível 4 para uma área de processo supõe que esta área de processo é um condutor chave de negócios que a organização deseja para gerenciar, usando técnicas estatísticas e quantitativas. Essa análise dá à organização mais visibilidade sobre o desempenho dos sub-processos selecionados, que a tornam mais competitiva no mercado.

Chegar ao nível de capacidade 5 para uma área de processo supõe que os sub-processos selecionados estão estabilizados e que se deseja reduzir as causas de variações dentro desses processos. A variação, segundo Chrissis (2003) é uma ocorrência natural em qualquer processo e, embora seja conceitualmente possível, não seria econômico melhorar todos os processos no nível 5 (cinco), assim sendo, poder-se-ia enfatizar aqueles processos que poderiam ajudar a organização a atingir os seus objetivos de negócio.

Conforme o descrito pelo (SEI, 2002a; SEI, 2002b), ao escolher a representação contínua para uma determinada organização, espera-se que o modelo:

- Permita selecionar a ordem de melhoria que melhor satisfaz os objetivos e suaviza as áreas de risco da organização.
- Possibilite comparações entre organizações em uma área de processo, tendo como base a área de processo ou pela comparação de resultados decorrentes do uso de *equivalent staging*.
- Proporcione uma fácil integração do *Electronic Industries Alliance Ínterim Standard* (EIA/IS) 731, para o CMMI.
- Propicie uma comparação fácil de melhoria de processo para a Organização Internacional para Padronização e Comissão Eletrotécnica Internacional (ISO/IEC) 15504, dado que a organização de áreas de processo é similar à ISO/IEC 15504.

2.1.1.4 Representação por estágio

A representação por estágio oferece um modo sistemático e estruturado para a abordagem de melhoria de processo, um passo a cada vez. A execução de cada estágio garante que uma melhoria adequada foi colocada como fundamentação para o próximo estágio. As áreas de processo são organizadas pelos níveis de maturidade que tomam muito do suposto trabalho realizado de melhoria de processo (SEI, 2002c).

Os componentes de um modelo CMMI com uma representação por estágio são mostrados na Figura 5.

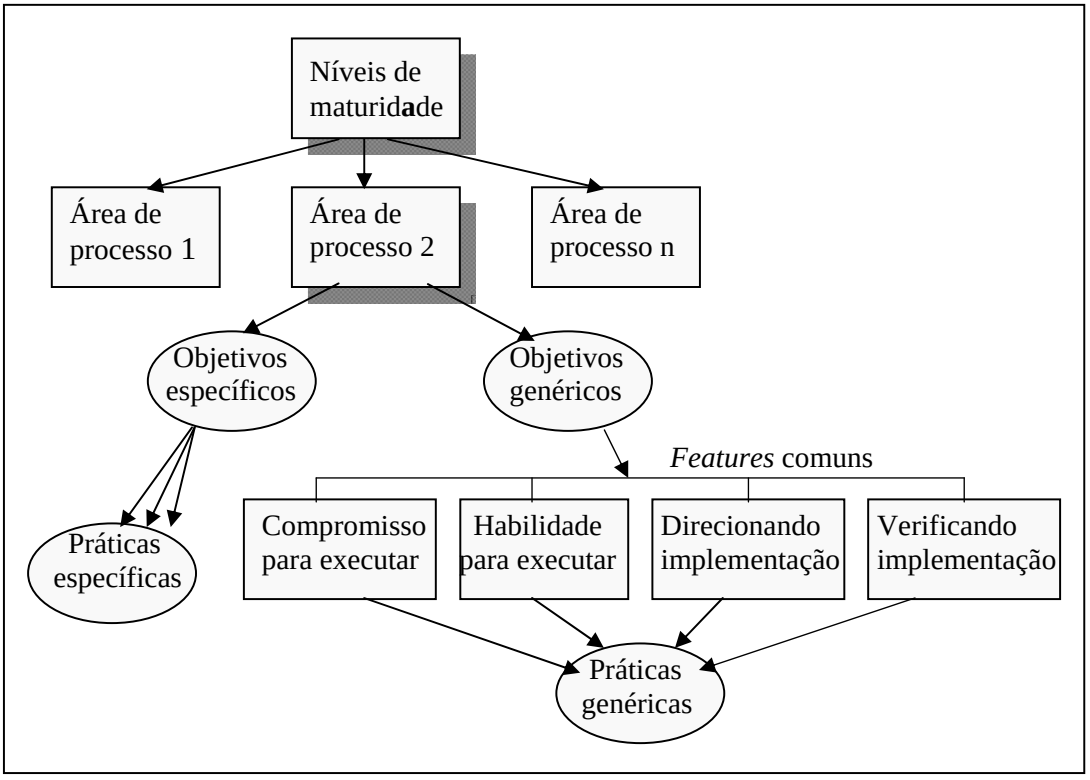


Figura 5. Estrutura da representação por estágio (SEI, 2002a)

Todos os modelos do CMMI com uma representação por estágio refletem níveis de maturidade em seu projeto e conteúdo. Um nível de maturidade consiste de práticas genéricas e específicas relacionadas a um conjunto predefinido de áreas de processo que melhoram o desempenho global da organização. O nível de maturidade de uma organização fornece um meio para prever o desempenho de uma organização em uma dada disciplina. Experiências têm mostrado (SEI, 2002a), que as organizações fazem seu melhor quando concentram seus esforços de melhoria de processo, sobre um número gerenciável de áreas de processo de cada vez ou sobre aquelas áreas de processo que exigem incrementos de sofisticação à medida que a organização melhora.

Um nível de maturidade é um platô evolucionário definido para melhoria organizacional de processo. Cada nível de maturidade estabiliza uma parte importante dos processos da organização, preparando-a para passar para o próximo nível de maturidade. Os níveis de maturidade são medidos por meio da obtenção de objetivos genéricos e específicos associados com cada conjunto predefinido de áreas de processo (CHRISSIS, 2003).

Conforme mostra a Figura 5, os níveis de maturidade organizam as áreas de processo (SEI, 2002a). Os cinco níveis de maturidade compõem cada um, uma camada base para melhoria de processos em andamento e são descritos a seguir.

Nível de maturidade 1: Inicial. Nesse nível, os processos são freqüentemente “ad hoc” e caóticos. A organização normalmente não propicia um ambiente estável para apoiar esses processos. O sucesso numa organização depende da competência e do heroísmo de pessoas na organização e não do uso de processos provados. Apesar desse caos, organizações no nível de maturidade 1 geralmente produzem produtos e serviços que funcionam, contudo, freqüentemente excedem seus orçamentos e não cumprem seus cronogramas. Organizações no nível de maturidade 1 são caracterizadas por uma tendência ao excesso de obrigações, abandono de processo em tempos de crise e inabilidade para repetir seus sucessos.

Nível de maturidade 2: Gerenciado. Os projetos da organização asseguram que os requisitos são gerenciados e que os processos são planejados, executados, medidos e controlados. A

disciplina de processo refletida pelo nível de maturidade ajuda a assegurar que práticas existentes são mantidas durante os períodos de crise. Quando essas práticas são consideradas, projetos são executados e gerenciados de acordo com seus planos documentados. O *status* dos artefatos e a distribuição de serviços são visíveis para gerenciamento em pontos definidos - no marco principal e ao término das tarefas principais. Compromissos são estabelecidos entre *stakeholders* relevantes e são revisados quando necessário. Artefatos e serviços satisfazem as descrições especificadas de processos, padrões e procedimentos.

Nível de maturidade 3: Definido. Os processos são caracterizados como entendidos e são descritos de acordo com padrões, procedimentos, ferramentas e métodos. O conjunto de processos padrão da organização, o qual é a base para o nível de maturidade 3, é estabelecido e melhorado ao longo do tempo. Esses processos padrão são usados para estabelecer consistência na organização. Projetos estabelecem seus processos definidos adaptando o conjunto de processos padrão da organização, de acordo com a diretriz de adaptação.

Uma diferença crítica entre os níveis de maturidade 2 e 3 é o escopo de padrões, descrições de processo e procedimentos. No nível de maturidade 2, os padrões, descrições de processo e procedimentos podem ser totalmente diferentes em cada instância específica do processo. No nível de maturidade 3, os padrões, descrições de processo e procedimentos para um projeto, são adaptados a partir do conjunto de processos padrão da organização, para satisfazer um projeto em particular ou unidade organizacional e, portanto, são mais consistentes exceto para diferenças permitidas pelas diretrizes adaptadas (CHRISSIS, 2003).

Outra distinção crítica é que no nível de maturidade 3, os processos são descritos mais rigorosamente que no nível 2. Um processo claramente definido expressa o propósito, as entradas, o critério de entrada, atividades, funções, medições, passos de verificação, saída e critério de saída. No nível de maturidade 3, os processos são gerenciados mais proativamente usando a compreensão das inter-relações das atividades de processo e das medidas detalhadas do processo, seus artefatos e seus serviços.

Nível de maturidade 4: Quantitativamente gerenciado. A organização e projetos estabelecem objetivos quantitativos para qualidade e desempenho de processo e os utiliza como critério no gerenciamento de processo. Os objetivos quantitativos são baseados nas necessidades dos clientes, dos usuários finais, da organização e dos implementadores de processo. Qualidade e desempenho de processo são entendidos em termos estatísticos e são gerenciados durante a vida dos processos. Para os sub-processos, medições detalhadas de desempenho de processo são coletadas e estatisticamente analisadas. Medidas de qualidade e de desempenho de processo são incorporadas no repositório de medição, para apoiar a tomada de decisão baseada em fatos. Causas especiais de variação de processo são identificadas e, quando apropriado, as origens das causas especiais são corrigidas para prevenir ocorrências futuras.

Para Chrissis (2003), uma distinção crítica entre os níveis de maturidade 3 e 4 é a previsibilidade de desempenho de processo. No nível 4, o desempenho de processo é controlado com o uso de estatística ou outras técnicas quantitativas e é quantitativamente previsível. No nível 3, os processos são somente previsíveis qualitativamente.

Nível de maturidade 5: Em otimização. Nesse nível, uma organização melhora continuamente seus processos, baseada em um entendimento quantitativo das causas comuns de variação inerentes aos processos. Esse nível se concentra em melhorar continuamente o desempenho de processo por meio de processo incremental e inovador e de melhorias tecnológicas. Objetivos quantitativos para melhoria de processo para a organização são estabelecidos, continuamente revisados para refletir mudanças nos objetivos de negócio e usados como critério na melhoria do gerenciamento de processo.

Os efeitos das melhorias de processo realizadas são medidos e avaliados em contraposição aos objetivos qualitativos de melhoria de processo. Tanto os processos definidos quanto o conjunto padrão de processos da organização, são considerados atividades mensuráveis de melhoria.

Uma distinção crítica entre os níveis de maturidade 4 e 5, segundo Chrissis (2003), é o tipo de variação de processo tratada. No nível 4, a organização está preocupada em tratar causas

especiais de variação de processo e fornecer previsibilidade estatística de resultados. Embora os processos possam traduzir resultados previsíveis, os resultados podem ser insuficientes para se alcançar os objetivos definidos. No nível de maturidade 5, a organização está interessada em tratar causas comuns de variação de processo e mudar o processo para deslocar a média de desempenho de processo ou reduzir a variação experienciada inerente ao processo, para melhorar o desempenho de processo e para alcançar os objetivos quantitativos de melhoria de processo estabelecidos.

Conforme o descrito pelo (SEI, 2002a; SEI 2002b), ao escolher a representação por estágio para uma determinada organização, espera-se que o modelo:

- Forneça uma seqüência comprovada de melhoria começando com práticas básicas de gerenciamento e progredindo por meio de um caminho pré-definido e provado, de níveis sucessivos, cada um servindo como uma fundamentação para o próximo.
- Permita comparações entre organizações pelo uso de níveis de maturidade.
- Forneça uma migração fácil do SW-CMM para o CMMI.
- Ofereça uma classificação única que sumariza os resultados das avaliações e possibilite comparações entre organizações.

Dado que a representação por estágio está relacionada com a maturidade global de um conjunto de processos, é de pouca conseqüência se processos individuais são executados ou estão incompletos. Por isso, o nome ‘inicial’ é atribuído ao ponto inicial da representação por estágio.

Ambos os níveis - maturidade e capacidade - fornecem um meio para medir (avaliar) quão bem as organizações podem melhorar e melhoram seus processos.

A representação por estágio prescreve a ordem para implementar cada área de processo de acordo com os níveis de maturidade, os quais definem o caminho de melhoria para uma organização, do nível inicial até o nível em otimização. A obtenção de cada nível de maturidade garante uma fundamentação adequada de melhoria que vai sendo colocada para o

próximo nível e possibilita, por fim, a melhoria incremental. Caso não se saiba por onde começar e qual processo escolher para melhorar, a representação por estágio é uma boa escolha uma vez que oferece um conjunto específico de processos para melhoria, que tem sido determinado por mais de uma década de pesquisa e experiência na comunidade de software (CHRISSIS, 2003).

Algumas diferenças são notórias entre as duas representações. A representação por estágio tem níveis de maturidade e *features* comuns, enquanto que a representação contínua não.

Ambas as representações fornecem meios para implementar melhoria de processo visando alcançar os objetivos de negócio. Ambas as representações provêm o mesmo conteúdo essencial e usam os mesmos componentes de modelos, a saber: áreas de processo, objetivos específicos, práticas específicas, objetivos genéricos, práticas genéricas, produtos típicos de trabalho, sub-práticas, notas, amplificações de disciplinas, elaborações de práticas genéricas e referências.

De acordo com Kulpa (2003), o que não é prontamente aparente da visão de alto nível das Figuras 4 e 5, é que a representação contínua concentra-se na capacidade da área de processo como medida para os níveis de capacidade, enquanto a representação por estágio foca a maturidade organizacional como medida dos níveis de maturidade. Essas dimensões (dimensão de capacidade e de maturidade) são utilizadas para atividades de avaliação e *benchmarking*, tanto como, para orientar esforços de melhoria de uma organização.

Existem três categorias de fatores que podem influenciar a decisão quando da seleção de uma representação (CHRISSIS, 2003):

Negócios. Uma organização com conhecimento maduro de seus próprios objetivos de negócios, provavelmente tem um pronunciado mapeamento de seus processos para seus objetivos de negócio. Tal organização pode achar que a representação contínua é útil para avaliar seus processos e na determinação do quão bem os processos da organização apóiam e vão ao encontro de seus objetivos de negócios. Se uma organização com foco em linhas de produto decide melhorar todos os seus processos, será melhor atendida pela representação por

estágio. A representação por estágio irá ajudar uma organização a selecionar os processos críticos para aos quais será enfatizada a melhoria. A mesma organização pode optar por melhorar processos em linha de produto. Nesse caso, pode selecionar a representação contínua – e um grau de avaliação diferente de capacidade pode ser obtido para cada linha de produto. Ambas as abordagens são válidas. A consideração mais importante é sobre quais objetivos de negócios se deseja com o programa de melhoria de processo para suporte e, como esses objetivos de negócio se alinham com a duas representações.

Fatores culturais. Considerados quando da seleção de uma representação, estão relacionados à habilidade da organização para desenvolver um programa de melhoria de processo. Por exemplo, uma organização pode selecionar a representação contínua se a cultura comum está baseada em processo e experienciada na melhoria de processo ou tem um processo específico que necessita ser melhorado rapidamente. Uma organização que tem pouca experiência em melhoria de processo pode escolher a representação por estágio, a qual fornece diretriz adicional sobre a ordem na qual mudanças devem ocorrer.

Legado. Se uma organização tem experiência com representação por estágio, pode estar experiente para continuar com a representação por estágio do CMMI, especialmente se a organização investiu recursos e desenvolveu processos em toda a organização, que estão associados com a representação por estágio. O mesmo é verdade para representação contínua. Ambas as representações estão incluídas no CMMI de modo que as organizações que as têm usado com sucesso poderiam continuar de uma maneira que é confortável e familiar tanto como bem sucedida.

Conforme explica Chrissis (2003), não importa se usadas para a melhoria ou para a avaliação de processo, as duas representações são designadas essencialmente para oferecer resultados equivalentes. Mais de oitenta por cento do conteúdo do modelo CMMI é comum a ambas as abordagens, por conseguinte, uma organização não necessita selecionar uma representação acima de outra. Ainda segundo Chrissis (2003), uma organização pode encontrar utilidade em ambas as representações, dado que é raro que uma organização implemente qualquer representação exatamente como determinado.

A Tabela 3 mostra as vantagens comparativas entre cada representação e serve como um guia para orientar a determinar qual é a representação correta para uma organização.

Tabela 3. Vantagens comparativas da representação contínua e por estágio (CHRISSIS, 2003)

Representação Contínua	**Representação Por Estágio**
Permite autonomia explícita para selecionar a ordem de melhoria que melhor reúna os objetivos de negócio da organização e suavize as áreas de risco da organização.	Possibilita que as organizações tenham um caminho de melhoria predefinido e provado.
Possibilita visibilidade elevada da capacidade alcançada em cada área de processo individual.	Concentra-se em um conjunto de processos que guarnece uma organização com uma capacidade específica que é caracterizada por cada nível de maturidade.
Fornece uma avaliação do nível de capacidade que é usada primeiramente para melhoria em uma organização e raramente é comunicada externamente.	Fornece uma avaliação do nível de maturidade que é freqüentemente usado na comunicação de gerenciamento interno, declarações externas para a organização e durante aquisições como um recurso para qualificar licitantes.
Permite a melhoria de distintos processos a serem executados em diferentes medidas.	Sumariza os resultados da melhoria de processo em uma forma simples – um único número de nível de maturidade.
Reflete uma nova abordagem que ainda não tem dados para demonstrar sua linha de retorno sobre o investimento.	Constrói numa história relativamente longa de uso que inclui estudos de caso e dados que demonstram retorno comprovado no investimento.
Fornece uma migração fácil do SECM para o CMMI.	Fornece uma migração fácil do SW-CMM para o CMMI.
Propicia uma fácil comparação de melhoria de processo para a ISO/IEC 15504 devido ao fato de que a organização das áreas de processo é derivada da ISO/IEC 15504.	Possibilita a comparação com a ISO/IEC 15504, mas a organização das áreas de processo não corresponde à organização usada na ISO/IEC 15504.

Organizações que são bem sucedidas na melhoria de processo freqüentemente definem um plano de melhoria que focaliza uma única necessidade da organização e, portanto, usam os princípios tanto da representação contínua como da representação por estágio. Por exemplo, organizações que escolhem a representação por estágio e estão no nível dois de maturidade, freqüentemente implementam as áreas de processo do nível dois de maturidade e também a área de processo “Foco no Processo Organizacional”, a qual está inserida no nível três de maturidade. Outro exemplo, uma organização que escolhe a representação contínua, para

orientar seus esforços de melhoria dos processos internos e depois escolhe a representação preventiva, para conduzir uma avaliação.

A Tabela 4 mostra uma análise comparativa entre a representação contínua e a representação por estágio, a qual serve como para orientar a determinar, qual é a representação mais adequada para uma determinada organização.

Tabela 4. Comparando as representações (CHRISSIS, 2003)

Representação contínua	**Representação por estágio**
Áreas de processo são organizadas por categorias de áreas de processo.	Áreas de processo são organizadas por nível de maturidade.
Melhoria é medida usando níveis de capacidade. Níveis de capacidade. • Medem a maturidade de um processo em particular de uma organização. • Alcance de 0 a 5.	Melhoria é medida usando níveis de Níveis de maturidade. • Medem a maturidade de um conjunto de processos de uma organização. • Alcance de 1 a 5.
Existem dois tipos de práticas específicas: básica e avançada. Todas as práticas específicas aparecem na representação contínua.	Existe somente um tipo de prática específica. Os conceitos de práticas básicas e avançadas não são usados. Todas as práticas específicas aparecem nessa representação, exceto quando um par de práticas básicas avançadas aparece na representação contínua, em tal caso, somente a prática avançada aparece na representação por estágio.
Níveis de capacidade são usados para organizar as práticas genéricas. Todas as práticas genéricas estão incluídas em cada área de processo.	*Features* comuns são usadas para organizar as práticas genéricas. Somente o nível 2 e o nível 3 incluem as práticas genéricas.
Equivalent staging possibilita a determinação de um nível de maturidade a partir do perfil de alcance de uma organização.	Não há necessidade de um mecanismo de equivalência voltado para a representação contínua porque cada organização pode escolher o que é melhor e quanto melhor, usando a representação por estágio.

Dado que muitas organizações vêm usando o SW-CMM ou o SECM, é importante observar que há um benefício claro para fazer a transição para os produtos CMMI ou para começar a melhoria de processo usando produtos CMMI em vez de outros, uma vez que o CMMI oferece uma cobertura mais detalhada do ciclo de vida do produto que outro processo de

melhoria de produto. Por exemplo, a ênfase em engenharia do CMMI supera a encontrada no SW-CMM e, a ênfase em gerenciamento de processo do CMMI supera a encontrada no SECM (CHRISSIS, 2003). Os produtos do CMMI incorporaram as lições aprendidas durante o desenvolvimento, manutenção e uso dos modelos fonte a partir dos quais foi desenvolvido, por isso, tratam alguns dos problemas encontrados tanto no SW-CMM como no modelo SECM, por exemplo.

Para Chrissis (2003), o CMMI propicia uma oportunidade para eliminar obstáculos e barreiras que tipicamente existem em diferentes partes de uma organização e que tipicamente não são tratadas em outro modelo de melhoria de processo. A combinação de informações úteis sobre a engenharia de um produto e as práticas provadas para processos de gerenciamento resulta em um conjunto de modelos bem integrado que facilitará o gerenciamento de projeto e a melhoria do processo de desenvolvimento e dos produtos resultantes.

Para Zubrow (2002), a nova característica contida nesse modelo, é a área de processo Análise e Medição cujo propósito é sustentar a capacidade de medição que é usada para apoiar informações de gerenciamento necessárias. Descrita no nível de maturidade 2 com a área de processo suporte, ‘provê’ serviços a outros processos e suporta todas as áreas de processo, fornecendo práticas que orientam projetos e organizações ao alinhar necessidades de medição e objetivos com uma abordagem de medição que fornecerá resultados objetivos que podem ser usados para tomar decisões bem informadas e tomar ações corretivas apropriadas.

O CMMI é valioso para organizações que produzem soluções somente para software. As funções de engenharia de sistemas, tipicamente não tratadas em detalhe em outros modelos somente para software, são valiosas para aquelas soluções na produção só de software. O tratamento de requisitos, por exemplo, é discutido em muito mais detalhe que no CMM para software. Embora não tratadas previamente nos CMM’s para organizações somente de software, essas práticas usam terminologia e arquitetura de modelo familiar que ajuda a gerenciar e a prevenir dificuldades relacionadas aos requisitos de software – um conceito que não é novo para muitas organizações de software (CHRISSIS, 2003).

O CMMI possibilita ao usuário selecionar a representação de modelo (ou ambas as representações) que melhor satisfaz seus objetivos de negócio. A flexibilidade existente em todo o modelo CMMI suporta tanto a abordagem contínua como a abordagem por estágio, para melhoria de processo com terminologia comum, arquitetura e métodos de avaliação. Embora o foco inicial fosse sobre produto e engenharia de serviço, modelo CMMI foi projetado também para outras disciplinas, suportando assim, melhoria de processo de grandes empresas. Assim como qualquer outro CMM, o CMMI requer critério profissional para interpretar informações. Embora as áreas de processo descrevam o comportamento que deveria ser exibido em uma organização, todas as práticas devem ser interpretadas usando conhecimento profundo do CMM, da organização, do ambiente de negócios e das circunstâncias envolvidas (CHRISSIS, 2003).

O propósito do CMMI (SEI, 2002a; SEI, 2002b) é fornecer diretrizes para melhorar os processos de uma organização e capacitá-la para gerenciar o desenvolvimento, a aquisição e a manutenção de produtos ou serviços. O CMMI considera abordagens comprovadas em uma estrutura que ajuda uma organização a avaliar a maturidade organizacional ou a capacidade de uma área de processo, estabelecendo prioridades para melhorias e para implementar essas melhorias.

O *framework* CMMI também foi concebido para apoiar a integração futura de outras disciplinas. Além disso, o CMMI foi desenvolvido para ser consistente e compatível com o relatório técnico da ISO/IEC 15504 para avaliação de processo de software, Chrissis (2003).

Para Mutafelija (2003), o CMMI foi criado para harmonizar vários modelos de maturidade e capacidade: engenharia de sistemas, engenharia de software, aquisição e desenvolvimento integrado de produto. O CMMI consolida a transição de atividades e fornece uma abordagem sistemática para a institucionalização de processo em todos esses domínios. Em adição, o CMMI foi escrito tendo em mente a ISO 15504.

2.1.2 O Padrão ISO 12207

Há algum tempo, tem havido uma proliferação de padrões, procedimentos, métodos, ferramentas e ambientes para desenvolvimento e gerenciamento de software. Essa proliferação tem criado dificuldades na engenharia e no gerenciamento de software, especialmente na integração de produtos e serviços. A disciplina de software necessita migrar dessa proliferação para um *framework* comum, o qual provê uma linguagem comum, para que engenheiros de software possam criar e gerenciar software. A ISO/IEC 12207 fornece esse *framework* (ISO/IEC 12207, 1998).

O *framework* fornecido pela ISO/IEC 12207 cobre o ciclo de vida do software desde a concepção de idéias por meio de refinamento e consiste no processo de aquisição e fornecimento de produtos e serviços de software. Uma organização, dependendo do seu objetivo, pode selecionar um subconjunto apropriado para satisfazer tal objetivo. É, portanto, um padrão especificado para ser adaptado para uma organização em particular, um projeto ou aplicação. É especificado também para ser usado quando o software é uma entidade única ou uma parte integrante ou embutida de um sistema completo.

Esse padrão estabelece um *framework* comum para processos do ciclo de vida de software, com terminologia bem definida, que pode ser referenciada pela indústria de software. Contém processos, atividades e tarefas para serem aplicadas durante a aquisição de um sistema que contém software, podendo ser um produto de software ou um serviço de software, durante o fornecimento, desenvolvimento, operação e manutenção de produtos de software. Esse padrão também fornece um processo que pode ser utilizado para definir, controlar e melhorar processos do ciclo de vida.

A ISO/IEC 12207 descreve a arquitetura dos processos do ciclo de vida de software, mas não especifica os detalhes de como implementar ou executar as atividades e tarefas inclusas nos processos.

Esse padrão agrupa as atividades que podem ser executadas durante o ciclo de vida de software em cinco processos fundamentais, oito processos de suporte e quatro processos organizacionais, conforme ilustrado na Figura 6. Cada processo do ciclo de vida é dividido em um conjunto de atividades. Posteriormente cada atividade é dividida em um conjunto de tarefas. O padrão ISO/IEC 12207 tem vários processos – descritos a seguir - que são aplicados ao longo do ciclo de vida de software por várias organizações dependendo de suas necessidades e objetivos.

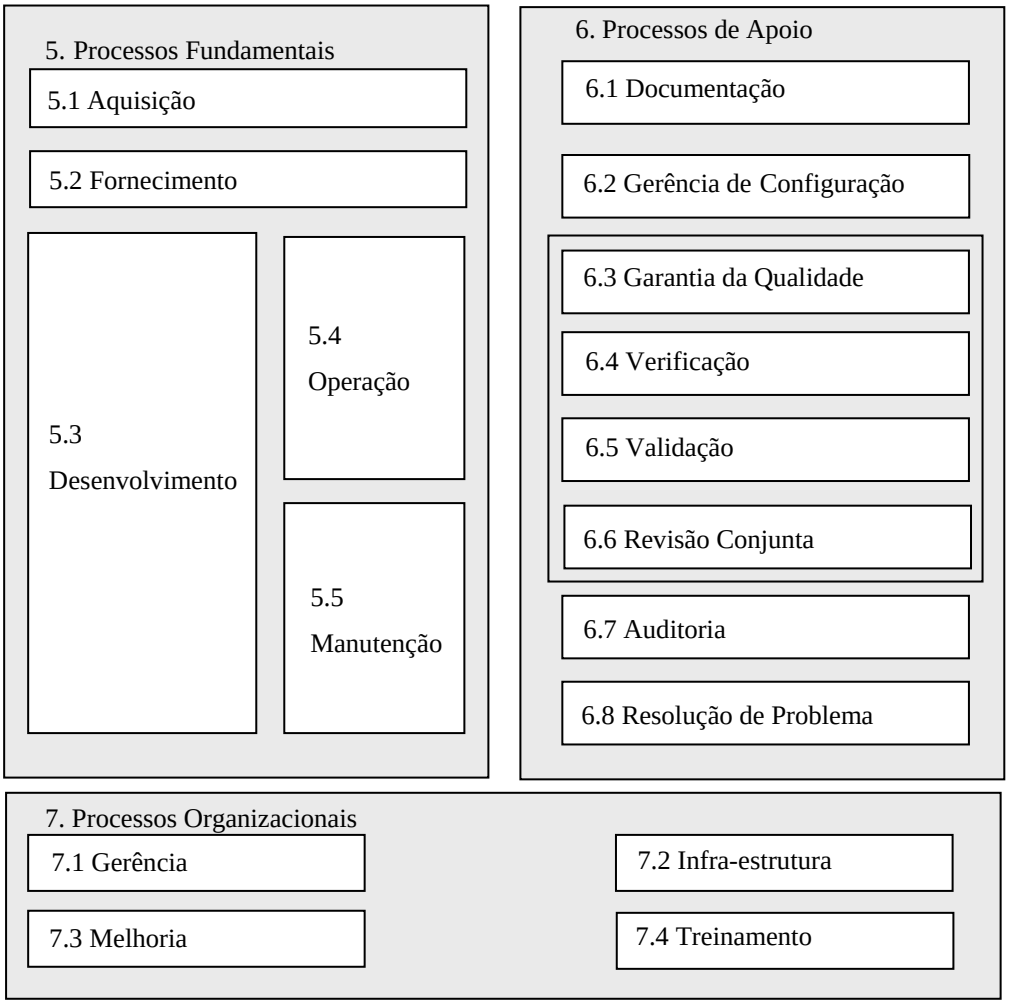


Figura 6. Estrutura do padrão ISO/IEC 12207 (2003)

2.1.2.1 Processos fundamentais

Consistem de cinco processos que atendem as partes fundamentais durante o ciclo de vida de software. Uma parte fundamental tem início com o desenvolvimento, operação ou manutenção de software. Essas partes fundamentais são o adquirente, o fornecedor, o desenvolvedor, o operador e o mantenedor de produtos de software. As atividades e tarefas em um processo fundamental são de responsabilidade da organização que inicia e executa este processo. A organização garante a existência e a funcionalidade dos processos. São os seguintes os processos fundamentais que compõem esse padrão:

Processo de aquisição. Define as atividades e atividades do adquirente, a organização que adquire o sistema, produto de software ou serviço de software. O processo inicia-se com a definição da necessidade de se adquirir um sistema, um produto de software ou um serviço de software. Continua com a preparação e emissão de um pedido de proposta, seleção de fornecedor e gerenciamento do processo de aquisição por meio da aceitação do sistema, produto de software ou serviço de software.

O adquirente gerencia o processo de Aquisição ao nível de projeto, seguindo o Processo de Gerenciamento, o qual é instanciado nesse processo; estabelece uma infra-estrutura sob o processo seguindo o Processo de Infra-estrutura; adapta o processo para o projeto seguindo o processo de adaptação; e gerencia o processo ao nível organizacional, seguindo o Processo de Melhoria e Processo de treinamento.

Processo de Fornecimento. Define as atividades e tarefas do fornecedor, a organização que fornece o sistema, produto ou serviço de software para o adquirente. O processo pode ser iniciado tanto pela decisão de preparar uma proposta para respondera um pedido de proposta do fornecedor como pela assinatura e celebração de um contrato com o adquirente para fornecer o sistema, produto de software ou serviço de software. O processo continua com a determinação de procedimentos e recursos necessários para gerenciar e garantir o projeto,

incluindo o desenvolvimento e a execução dos planos de projeto até a entrega do sistema, produto de software ou serviço de software para o adquirente.

O fornecedor gerencia o processo de Fornecimento ao nível de projeto seguindo o Processo de gerenciamento, o qual é instanciado neste processo; estabelece uma infra-estrutura sob o processo seguindo o Processo de Infra-estrutura; adapta o processo para o projeto seguindo o Processo de adaptação; e gerencia o processo ao nível organizacional seguindo o processo de Melhoria e o Processo de Treinamento.

Processo de desenvolvimento. Define as atividades do desenvolvedor, a organização que define e desenvolve o produto de software. Compreende as atividades para análise de requisitos, projeto, codificação, integração, teste e instalação e aceitação, relacionadas aos produtos de software. Pode conter atividades relacionadas ao sistema se estipulado no contrato. O desenvolvedor executa ou apóia as atividades nesse processo de acordo com o contrato.

O desenvolvedor gerencia o processo de desenvolvimento ao nível de projeto seguindo o Processo de Gerenciamento, o qual é instanciado nesse processo; estabelece as infra-estruturas sob o processo seguindo o Processo de Infra-estrutura; adapta o processo ao projeto seguindo o Processo de Adaptação; e gerencia o processo ao nível organizacional seguindo o Processo de Melhoria e o Processo de Treinamento. Quando o desenvolvedor é o fornecedor do produto de software desenvolvido, o desenvolvedor executa o Processo de Fornecimento.

Processo de operação. Define as tarefa e atividades do operador, a organização que fornece o serviço de operação para um sistema de computador em seu ambiente de funcionamento para seus usuários. Este processo cobre a operação do produto de software e o suporte operacional aos usuários. Devido ao fato de a operação do produto de software estar integrada à operação do sistema, as tarefas e atividades deste processo referem-se ao sistema.

O operador gerencia o Processo de Operação ao nível de projeto seguindo o Processo de Gerenciamento, o qual é instanciado neste processo; estabelece uma infra-estrutura sob o

processo seguindo o Processo de Infra-estrutura; adapta o processo para o projeto seguindo o Processo de Adaptação; e gerencia o processo ao nível organizacional seguindo o Processo de Melhoria e o Processo de Treinamento. Quando o operador é o fornecedor do serviço de operação, o operador executa o Processo de Fornecimento.

Processo de manutenção. Define as tarefas e atividades do mantenedor, a organização que fornece o serviço de manutenção do produto de software, isto é, gerencia modificações para produto de software para mantê-lo atual e em perfeita operação. Esse processo inclui a migração e a retirada de circulação do produto de software. O processo é ativado quando o produto de software é submetido a modificações no código e na documentação associada devido a um problema ou à necessidade de melhoria ou adaptação. O objetivo é modificar produtos de software existentes preservando sua integridade. O processo termina quando o produto de software é retirado de circulação.

O mantenedor gerencia o Processo de Manutenção ao nível de projeto seguindo o Processo de Gerenciamento, o qual é instanciado neste processo; estabelece uma infra-estrutura sob o processo seguindo o processo de Infra-estrutura; adapta o processo para o projeto seguindo o Processo de Adaptação; e gerencia o processo ao nível organizacional seguindo o Processo de Melhoria e o Processo de Treinamento. Quando o mantenedor é o fornecedor do serviço de manutenção, o mantenedor executa o Processo de Fornecimento.

2.1.2.2 Processos de suporte ao ciclo de vida

O processo de suporte ao ciclo de vida consiste de oito processos. Um processo de suporte tem como objetivo dar suporte a outro processo visando primordialmente o sucesso e a qualidade do projeto de software. A organização emprega e executa o Processo de Suporte gerenciando-o ao nível de projeto seguindo o Processo de Gerenciamento; estabelece uma infra-estrutura sob o mesmo, seguindo o Processo de Infra-estrutura; adapta-o para o projeto seguindo o Processo de Adaptação; e gerencia-o ao nível organizacional seguindo o Processo

de Melhoria e o Processo de Treinamento. Revisões conjuntas, auditoria, verificação e validação, podem ser usadas como técnicas para a Garantia da Qualidade. Os processos de suporte são:

Processo de documentação. Define as atividades para registrar informações produzidas por um processo ou atividade do ciclo de vida. Contém um conjunto de atividades que planeja, projeta, desenvolve, produz, edita, distribui e mantém os documentos necessários a todos os interessados tais como: gerentes, engenheiros e usuários do sistema ou produto de software.

Processo de gerenciamento de configuração. Define as atividades para aplicação de procedimentos técnicos e administrativos ao longo de todo o ciclo de vida para: identificar e definir itens de software em um sistema; controlar modificações e liberações dos itens; registrar e apresentar a situação dos itens e dos pedidos de modificação; assegurar a completeza, a consistência e a corretitude dos itens; e controlar o armazenamento, o manuseio e a distribuição dos itens.

Processo de garantia da qualidade. Define as atividades para assegurar objetivamente que os produtos de software e processos do ciclo de vida estão em conformidade com os requisitos especificados e aderem aos planos estabelecidos. A garantia da qualidade pode ser interna ou externa dependendo se as evidências da qualidade do produto ou do processo apontam para o gerenciamento do fornecedor ou do adquirente. Revisões conjuntas, auditoria, verificação e validação podem ser usadas como técnicas para assegurar qualidade.

Processo de verificação. Define as atividades (para o adquirente, o fornecedor ou uma parte independente) para verificar os produtos e serviços de software em profundidade variada, dependendo do projeto de software. É um processo para determinar se os produtos de software de uma atividade atendem plenamente aos requisitos ou condições a eles impostas. No que concerne a custo e efetividade de desempenho, verificação deve ser integrada tão cedo quanto possível com o processo (tal como fornecimento, operação ou manutenção) que o utiliza. Este processo pode incluir análise, revisão e teste. Quando o processo é executado por

uma organização independente do fornecedor, desenvolvedor, operador ou mantenedor, é denominado de Processo de Verificação Independente.

Processo de validação. Define as atividades (para o adquirente, o fornecedor ou uma parte independente) para validar os produtos de software do projeto de software. É um processo para determinar se os requisitos e o produto final (sistema ou produto de software) atendem ao uso específico proposto. Caso o processo seja executado por uma organização independente do fornecedor, desenvolvedor, operador ou mantenedor, é denominado de Processo de Validação Independente.

Processo de revisão conjunta. Define as atividades para avaliar a situação e os produtos de uma atividade. Revisões conjuntas são feitas tanto nos níveis de gerenciamento de projeto, como nos níveis técnicos e são executas enquanto durante a vigência do contrato. Este processo pode ser empregado por quaisquer duas partes, onde uma parte (parte revisora) revisa outra parte (parte revisada) em uma reunião conjunta.

Processo de auditoria. Define as atividades para determinar conformidade com os requisitos, planos e contratos, quando apropriado.

Processo de resolução de problema. Define um processo para analisar e resolver problemas (inclusive não-conformidades) de qualquer natureza ou origem, que são descobertas durante a execução do desenvolvimento, operação, manutenção ou outros processos. O objetivo é fornecer os meios - no tempo adequado e de forma responsável e documentada - para assegurar que todos os problemas encontrados sejam analisados e resolvidos e que tendências sejam identificadas.

2.1.2.3 Processos organizacionais do ciclo de vida

Os processos organizacionais do ciclo de vida consistem de quatro processos e têm como objetivo a melhoria dos processos dentro da organização. Os processos organizacionais são:

Processo de gerenciamento. Define as atividades básicas de gerenciamento, incluindo gerenciamento de projeto relacionado à execução do processo de ciclo de vida. Contêm as tarefas e atividades genéricas que podem ser empregadas por quaisquer duas partes que têm que gerenciar seu(s) respectivo(s) processo(s). O gerente é responsável pelo gerenciamento de produto, de projeto e de tarefas do(s) processo(s) aplicáveis, tais como aquisição, fornecimento, desenvolvimento, operação, manutenção ou processos de suporte.

Processo de infra-estrutura. Define as atividades básicas para estabelecer e manter a estrutura necessária para qualquer outro processo do ciclo de vida. A infra-estrutura pode incluir hardware, software, ferramentas, técnicas, padrões e recursos para desenvolvimento, operação e manutenção.

Processo de melhoria. Define as atividades básicas que uma organização (adquirente, fornecedor, desenvolvedor, operador, mantenedor ou gerente de outro processo) executa para estabelecer, medir, controlar e melhorar um processo de ciclo de vida.

Processo de treinamento. Define as atividades para fornecer e manter pessoal adequadamente treinado. A aquisição, fornecimento, desenvolvimento, operação ou manutenção de produtos de software é amplamente dependente de pessoal com conhecimento e qualificação. É imperativo que o treinamento de pessoal seja planejado e implementado com antecedência de modo que o pessoal treinado esteja disponível assim que o produto de software for adquirido, fornecido, desenvolvido, operado ou mantido.

Processo de adaptação. Define as atividades básicas necessárias para adaptar este padrão a uma organização ou projeto específico.

Variações nas políticas e procedimentos de uma organização, métodos e estratégias de aquisição, tamanho e complexidade de projeto, requisitos do sistema e métodos de desenvolvimento, entre outras coisas, influenciam como o sistema é adquirido, operado e mantido. A ISO/IEC 12207 foi concebida para um projeto geral para acomodar tais variações tanto quanto possível. Logo, no interesse de redução de custo e melhoria da qualidade, este padrão deve ser adaptado para um projeto individual e todas as partes envolvidas no projeto devem ser envolvidas na adaptação.

A garantia da qualidade de software tradicionalmente media a qualidade dos produtos do projeto, incluindo artefatos entrementes. Com a introdução do CMM, o foco mudou para a qualidade do processo, sendo a qualidade de produto ainda uma parte integral do desenvolvimento do projeto, porém fora do escopo da metodologia de projeto. Com o CMMI, a garantia da qualidade de processo foi mantida junto com a qualidade de produto (FANTINA 2005).

Organizações que não exercitam ambos os aspectos da garantia de qualidade aumentam os riscos de sérias dificuldades. Existe uma tendência de se acreditar que a qualidade do produto pode ser testada e a qualidade do processo é desnecessária. Vários, mas não todos os problemas inerentes a esse modo de pensar são relacionados a seguir:

- Ausência de controle de projeto. Gerentes de projeto, financiadores (investidores) e outros, não sabem onde o projeto está posicionado em termos de orçamento, cronograma e qualidade.
- Inabilidade para repetir sucessos de projeto.
- Inabilidade para prevenir falhas repetitivas de projeto. Não monitoração do gerenciamento preventivo de processo, levando em conta onde os erros foram feitos.
- Tempo excessivo de teste. Isto pode resultar em datas de entregas esquecidas (não cumpridas) clientes insatisfeitos e perda de oportunidades competitivas.
- Re-trabalho oneroso.

- Qualidade de produto inferior.

A função da garantia de qualidade é assegurar que os padrões de qualidade estão sendo atendidos e que os passos (fases) para qualidade estão sendo executados para asseverar um produto ou serviço de alto nível.

2.1.3 O Padrão ISO 15504

A Organização Internacional para Padronização (ISO) desenvolveu uma série de padrões concernentes a processo de software, mais conhecido como projeto ou padrão SPICE[6], o qual é resultado de amplas e diversas pesquisas a respeito do assunto em questão. Além de tratar questões relacionadas ao processo de software, o padrão SPICE abrange outras questões relacionadas ao processo de desenvolvimento de software como recursos humanos, tecnologia, suporte ao cliente, práticas de desenvolvimento, gerenciamento e manutenção ademais de qualidade.

A ISO/IEC 12207 é o primeiro padrão internacional a fornecer um conjunto de processos do ciclo de vida, atividade e tarefas para software que é parte de um grande sistema, produto e serviços de software. O padrão fornece uma arquitetura de processos comum para a aquisição, fornecimento, desenvolvimento, operação e manutenção de software. Fornece também o suporte necessário para processos, atividades e outras tarefas e processo organizacional, atividades e tarefas para gerenciamento e melhoria de processo (ISO/IEC 12207, 2002).

Em 2002 foi publicada a Emenda 1 para estabelecer um conjunto co-ordenado de informação de processo de software que pode ser usado para definição, avaliação e melhoria de processo. A Emenda 1 resolve a questão da granularidade relacionada ao uso da ISO/IEC 12207 para

[6] *Software Process Improvement and Capability dEtermination*.

avaliação de processo e fornece o propósito e o resultado do processo para estabelecer um Modelo de Referência de Processo em conformidade com os requisitos da ISO/IEC 15504-2. Um Modelo de Referência de Processo fornece definições de processos em um ciclo de vida descrito em termos de objetivos e resultados dos processos. Um Modelo de Referência de Processo fornece um mecanismo por meio do qual, modelos de avaliação definidos externamente são relacionados com o *framework* de avaliação definido pela ISO/IEC 15504, (ISO/IEC 12207, 2002).

A ISO/IEC 15504 (2002) fornece uma abordagem estruturada para a avaliação de processos para os seguintes propósitos:

- Por uma ou parte de uma organização com o objetivo de entender o estado de seus próprios processos para melhoria de processo.
- Por uma ou parte de uma organização com o objetivo de determinara adaptabilidade de seus próprios processos para um requisito em particular ou classe de requisitos.
- Por uma ou parte de uma organização com o objetivo de determinar a adaptabilidade de outros processos da organização para um determinado contrato ou classe de contratos.

Um método para uma organização melhorar a qualidade de produto é usar um método provado, consistente e confiável para avaliar o estado de seus processos e usar os resultados como parte de um programa coerente de melhoria. O uso de avaliação de processo em uma organização deveria encorajar:

- A cultura da melhoria constante e o estabelecimento de mecanismos próprios para apoiar e manter tal cultura.
- A engenharia de processo para satisfazer os requisitos do negócio.
- A otimização de recursos.

Com isso, espera-se que a organização se torne uma organização capaz de maximizar suas sensibilidades para com os requisitos do cliente e do mercado; minimizar os custos do ciclo de vida completo de seus produtos e, como resultado, maximizar a satisfação do usuário final.

Adquirentes podem se beneficiar do uso da avaliação de processo. Seu uso na determinação da capacidade pode:

- Reduzir incertezas quando da seleção de fornecedores, possibilitando que riscos associados com a capacidade dos contratantes possam ser identificados antes de realizar o contrato.
- Possibilitar controles apropriados a serem considerados para a contenção de riscos.
- Prover uma base quantificada para escolher com imparcialidade as necessidades do negócio, requisitos e custos estimados de projeto em contraposição à capacidade de competir/concorrer com fornecedores.

Os principais objetivos de uma abordagem padronizada para avaliação de processo são, segundo o que estabelece a ISO/IEC 15504 (2003):

- Prover uma abordagem pública e compartilhada para avaliação de processo.
- Conduzir a um entendimento do uso de avaliação de processo para melhoria de processo e avaliação de capacidade.
- Facilitar a avaliação da capacidade em consecução.
- Poder ser controlada e revisada regularmente, à luz das experiências de uso.
- Poder ser alterada somente por consenso internacional.
- Encorajar a harmonização de esquemas existentes.

A ISO/IEC 15504 (2003) fornece um *framework* para avaliação de processos. Esse *framework* pode ser usado por organizações envolvidas no planejamento, gerenciamento, monitoramento, controle e melhoria de aquisição, fornecimento, desenvolvimento, operação, evolução e suporte a produtos/serviços.

O *framework* para avaliação de processo, conforme estabelece a ISO/IEC 15504 (2002):

- Facilita a auto-avaliação.
- Fornece uma base para uso em melhoria e determinação da capacidade do processo.
- Leva em conta o contexto no qual o processo executado está implementado.

- Produz uma avaliação de processo.
- Considera a habilidade do processo para alcançar seus propósitos.
- É apropriado para todos os domínios de aplicação e tamanhos de organização.
- Pode fornecer um *benchmark* objetivo entre organizações.

A avaliação de processo tem dois contextos principais para o seu uso, conforme mostra a Figura 7.

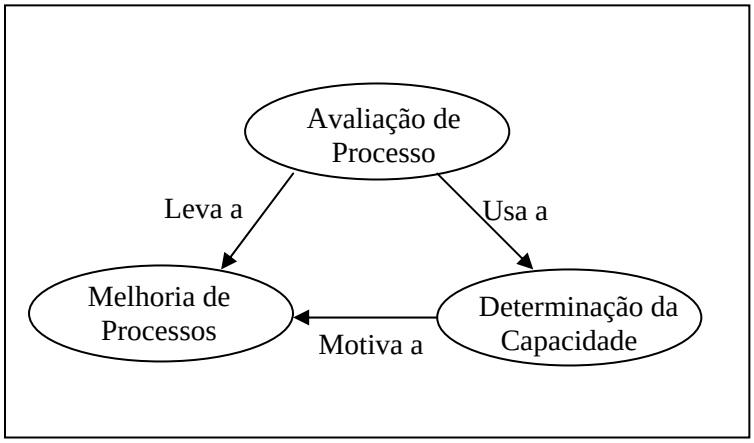


Figura 7. Relacionamento de avaliação de processo (ISO/IEC 15504, 2003)

Dentro de um contexto de melhoria de processo, a avaliação de processo fornece um meio de caracterizar a prática corrente em uma unidade organizacional em termos de capacidade dos processos selecionados. A análise dos resultados identifica pontos fracos e fortes e riscos inerentes ao processo. Isso fornece uma direção para a priorização de melhorias do processo.

Determinação da capacidade de processo está relacionada com analisar a capacidade pretendida dos processos selecionados em contraposição a um perfil de capacidade de processo alvo, a fim de identificar algum dos riscos envolvidos em executar um processo usando os processos selecionados. A capacidade pretendida pode ser baseada nos resultados

de avaliações anteriores de processos relevantes, ou pode ser baseada em uma avaliação realizada com o propósito de estabelecer a capacidade pretendida.

A avaliação de processo examina os processos usados por uma organização para determinar se são efetivos para a obtenção de seus objetivos. A avaliação caracteriza a prática corrente dentro em uma unidade organizacional, em termos da capacidade dos processos selecionados. Os resultados podem ser usados para conduzir as atividades de melhoria de processo ou determinar a capacidade de processo, por meio da análise dos resultados no contexto das necessidades do negócio de uma organização, identificando pontos fracos e fortes e riscos inerentes ao processo.

Avaliação de processo como definido pela ISO/IEC 15504-3 (2003) está baseada em um modelo de duas dimensões contendo uma dimensão de processo e uma dimensão de capacidade. A dimensão de processo é formada por um Modelo de Referência de Processo externo, o qual define um conjunto de processos caracterizados por declarações de objetivos (propósito/utilidade) de processo e resultado de processo. A dimensão de capacidade consiste de um *Framework* de Medição que compreende seis níveis de Capacidade de Processo e seus Atributos de Processos Relacionados.

A avaliação de processo é realizada para entender a capacidade dos processos correntes em uma Unidade Organizacional. A avaliação de processo pode compreender todos ou um subconjunto dos processos (gerenciamento de projeto, desenvolvimento, manutenção, gerenciamento de configuração) usados por uma organização.

A avaliação de processos de uma Unidade Organizacional é feita utilizando um Modelo de Avaliação de Processo baseado em um Modelo de Referência de Processo - ISO/IEC 12207. O Modelo de Referência de processo descreve os processos em termos de objetivo e resultado. Um Modelo de Avaliação de Processo detalha indicadores necessários para avaliar a obtenção dos atributos do processo (ISO/IEC 15504-3, 2003). Existe um conjunto de nove atributos de processo aplicáveis a qualquer processo para caracterizar a capacidade de um processo implementado.

Atributos de processo são agrupados em níveis de capacidade que definem uma escala ordinal de capacidade de processo e fornecem um caminho racional para melhoria de cada processo individualmente. Cada atributo de processo representa características mensuráveis com suporte à obtenção do objetivo do processo e contribui para alcançar os objetivos de negócio da organização.

O *framework* de medição para capacidade de processo é baseado nos conceitos de processo com atributos comuns. Esses atributos de processo foram definidos e alocados para níveis de capacidade.

O *framework* de medição define uma escala ordinal de seis pontos de um caminho para incrementar a capacidade de processo, indo de um processo o qual não é capaz de atingir seus objetivos – nível de capacidade zero - até um processo que otimiza seu desempenho – nível de capacidade 5. Cada processo tem um conjunto de atributos de avaliação de processo que constituem o perfil de um processo. O modelo de nível de capacidade de processo é descrito em termos de nível de avaliação dos atributos do processo que devem ser obtidos a fim de alcançar um determinado nível (ISO/IEC 15504-3, 2003).

Um Modelo de Referência de Processo descreve um conjunto de um ou mais processos em termos de propósito e resultado esperado. O propósito descreve os objetivos de mais alto nível que o processo deve alcançar, enquanto que os resultados associados são as saídas esperadas de desempenho bem sucedido do processo. A declaração dos propósitos em conjunção com os resultados descreve **o que** alcançar, mas não prescreve **como** o processo deve alcançar seus objetivos (ISO/IEC 15504-3, 2003).

Para Mutafelija (2003), a ISO 15504 é suficientemente genérica e pode ser aplicada a qualquer avaliação de processo e esforço de melhoria. Na ISO 15504, a melhoria de processo é contínua, inicia com uma avaliação de processo e continua até que um resultado esperado seja alcançado e confirmado pela próxima avaliação. As avaliações fornecem as linhas bases usadas para desenvolver os passos a serem executados no próximo ciclo de melhoria.

Na ISO 15504, segundo Mutafelija (2003), a abordagem de melhoria de processo é baseada em um conjunto de princípios de melhoria de processo, explicitados como segue:

- O uso de resultados de avaliação de processo, quando desenvolver uma estratégia de melhoria de processo.
- Avaliação de processo descreve a capacidade de um processo atual que pode ser comparado a uma capacidade alvo.
- A melhoria de processo é um processo contínuo.
- O uso de medições para monitorar o progresso de melhoria de processo e para efetuar correções.
- Uso de avaliações de processo para determinar se os resultados de melhoria de processo desejados foram encontrados.
- Gerenciamento de risco avaliando tanto risco de implementação, como risco de falha na iniciativa de melhoria.

Como um padrão internacional, a norma ISO 15504 contém todas as ferramentas necessárias para implementar melhoria de processo, incluindo, um modelo referencial, diretrizes para avaliação de processo e diretrizes para planejamento de melhoria de processo.

A abordagem de melhoria de processo tem oito fases baseadas nos princípios de melhoria, os quais são listados a seguir (MUTAFELIJA, 2003):

1. **Examinar as necessidades da organização e objetivos de negócio**. Cada organização precisa estabelecer seus objetivos de melhoria de processo e vinculá-los às metas e aos objetivos de negócio. Um benefício adicional de se estabelecer objetivos de melhoria de processo em termos de objetivos de negócio é que gerenciamento sênior consegue visibilidade significativa nos resultados de melhoria de processo.

2. **Iniciar melhoria de processo.** A experiência mostra que a maioria dos esforços de melhoria de processo bem sucedidos são executados como projetos e são baseados nos planos escritos. Planos de melhoria de processo especificam o escopo de melhoria de processo (em termos de entidades organizacionais e processos a serem incluídos no

esforço de melhoria), esquematizam as fases de projeto, estabelecem marcos e identificam riscos e a abordagem de gerenciamento.

3. **Preparar para e conduzir uma avaliação de processo.** Para medir o progresso e o sucesso de uma iniciativa de melhoria de processo é exigida uma linha base de processo. Vários métodos de avaliação, associados com processos detalhados são disponibilizados. Comum a todos os métodos de avaliação é a habilidade para identificar oportunidades de melhoria de processo. Embora a repetitividade dos resultados da avaliação possa variar com a formalidade do método, mesmo o método mais informal fornece resultados que podem ser usados para estabelecer um mapa do caminho de melhoria de processo.

4. **Analisar o trabalho de avaliação e gerar um plano de ação.** Sob análise de recomendação de avaliação, a organização avaliada desenvolve planos de melhoria de processo detalhados. Planos estratégicos respondem pelos objetivos de negócio e prioridades, enquanto os planos táticos descrevem os passos para melhoria de processo detalhada, para implementação de melhorias que satisfaçam os objetivos estratégicos.

5. **Implementar melhorias.** Uma infra-estrutura de melhoria de processo possibilita o desenvolvimento de novos processos e transfere as melhorias de processos para projetos. Tipicamente um grupo de engenharia de processo (GEP) é formado para conduzir essa tentativa. O GEP analisa resultados de avaliação e refina o plano de melhoria de processo. Uma vez determinado o curso de ação, o GEP formará as equipes de ação de processo (EAP) responsáveis por pesquisar e implementar atividades de melhoria de processo específicas.

6. **Confirmar melhorias.** Periodicamente o GEP mede e monitora o progresso de melhoria de processo, relata o progresso para o comitê de gerenciamento e institui ações corretivas conforme exigido. Re-avalia riscos e desenvolve ações para levar o projeto de melhoria de processo em direção aos seus objetivos.

7. **Manter ganhos de melhoria.** O GEP ajuda a manter ganhos de melhoria de processo e assegura que a organização não retorna aos seus estados prévios. O GEP suporta a transição de melhorias para outros projetos e monitora sua implementação.

8. **Monitorar desempenho.** Melhoria de processo requer um compromisso de longa duração. Não é raro ocorrer mudanças nos objetivos da organização ao longo do tempo, requerendo mudanças nos objetivos de melhoria de processo, o gerente sênior pode redirecionar as atividades de melhoria para melhor satisfazer a situação atual.

Ainda segundo Mutafelija (2003), o padrão ISO 15504 reconhece também a importância de questões culturais, de liderança de gestão para o sucesso do esforço de melhoria de processo e fornece diretrizes específicas para essas questões. Diretrizes para estabelecer comunicação efetiva e trabalho de equipe na organização, também são providas, incluindo o que segue:

- Necessidade de estabelecer um conjunto de valores.
- Atitudes e comportamento.
- Medição.
- Educação e requisitos de treinamento.

De acordo com Rocha (2001), a ISO 15504 realiza avaliações de processos de software com dois objetivos: “a melhoria dos processos e a determinação da capacidade de processos de uma organização”. Se o objetivo de uma organização está focado na melhoria dos processos, a mesma pode efetuar a avaliação produzindo uma representação dos processos que serão usados para a elaboração de um plano de melhorias, conforme mostra a Figura 8(a). Nesse caso, a organização “deve definir os objetivos e o contexto, bem como escolher o modelo e o método para a avaliação e definir os objetivos de melhoria”.

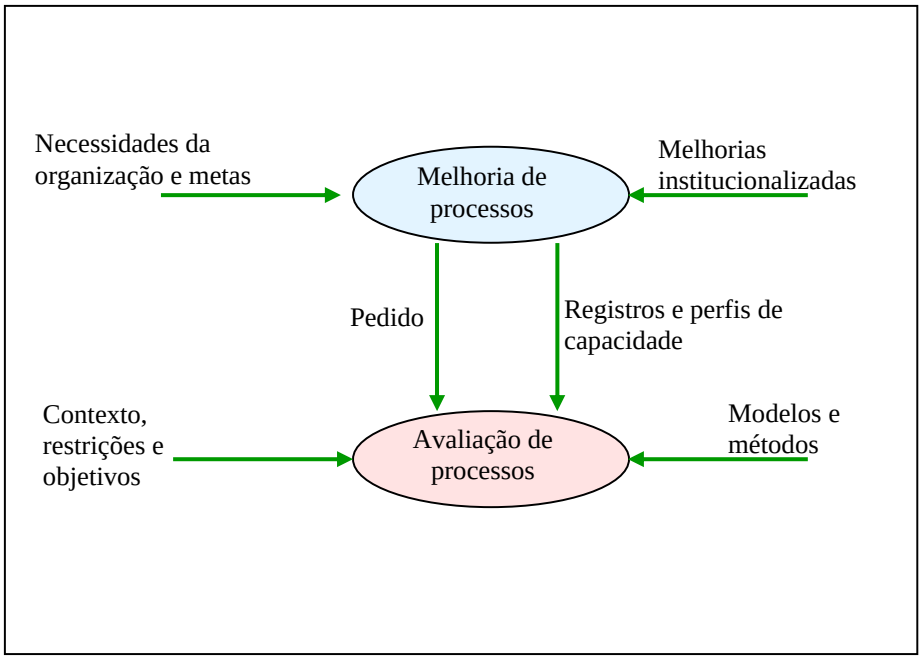


Figura 8(a). Uso da norma 15504 para melhoria de processo (ROCHA, 2001)

A norma ISO/IEC 15504, mais conhecida como SPICE, promoveu uma inovação quanto ao conceito de modelo, dado que é uma das precursoras com respeito ao conceito de modelo contínuo de processo. Os modelos contínuos de processo se caracterizam pelo fato de separarem os níveis de capacidade de processo, dos processos propriamente ditos, o que possibilita a uma organização eleger os processos relevantes para a organização, levando em conta os objetivos e as estratégias de negócios de referida organização, além, de nortearem a avaliação e as melhorias de tais processos, com base nos níveis de capacidade (ROCHA, 2001).

A ISO 15504 é, atualmente, um padrão de avaliação de processo que incentiva o uso de modelos de processos compatíveis e técnicas de avaliação. A arquitetura da ISO 15504 serviu como base para vários modelos, tal como o EIA/IS-731 e a representação contínua do modelo CMMI (MUTAFELIJA, 2003).

Quando se trata da determinação da capacidade de processos, a organização tem o objetivo de avaliar um fornecedor em potencial, obtendo o seu perfil de capacidade, tal como mostra a Figura 8(b). Para tanto, a empresa “deve definir os objetivos e o contexto da avaliação, os modelos e métodos de avaliação e os requisitos esperados”. O perfil possibilita ao contratante estimar o risco associado à contratação do fornecedor em potencial, visando a apoiar a tomada de decisão quanto à sua contratação ou não.

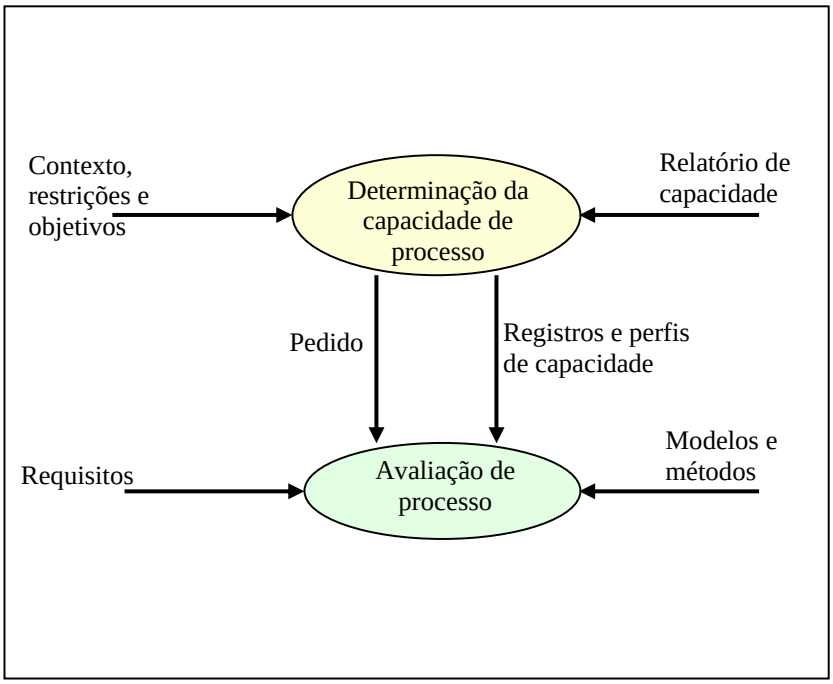


Figura 8(b). Uso da norma 15504 para a determinação da capacidade de processo (ROCHA, 2001)

Emam (2000) aclara que a norma ISO/IEC 15504 é um padrão internacional sobre avaliação do processo de software, o qual define um conjunto de processos de engenharia de software e uma escala que pode ser usada para avaliar a capacidade de cada processo separadamente, o requisito inicial para sua implementação, já que fornece um caminho de melhoria predefinido. Um dos processos definidos é a análise de requisitos de software. Uma das premissas básicas da escala de medição é que alta capacidade de processo está associada com melhor desempenho de processo.

A arquitetura do padrão ISO/IEC 15504 é bi-dimensional, como mostra a Figura 9. Como pode ser visto, uma dimensão consiste dos processos que são atualmente avaliados (a dimensão do processo). A segunda dimensão consiste da escala de capacidade que é usada para avaliar a capacidade do processo (a dimensão da capacidade). O processo de análise de requisitos de software é definido na categoria processo de engenharia na dimensão do processo.

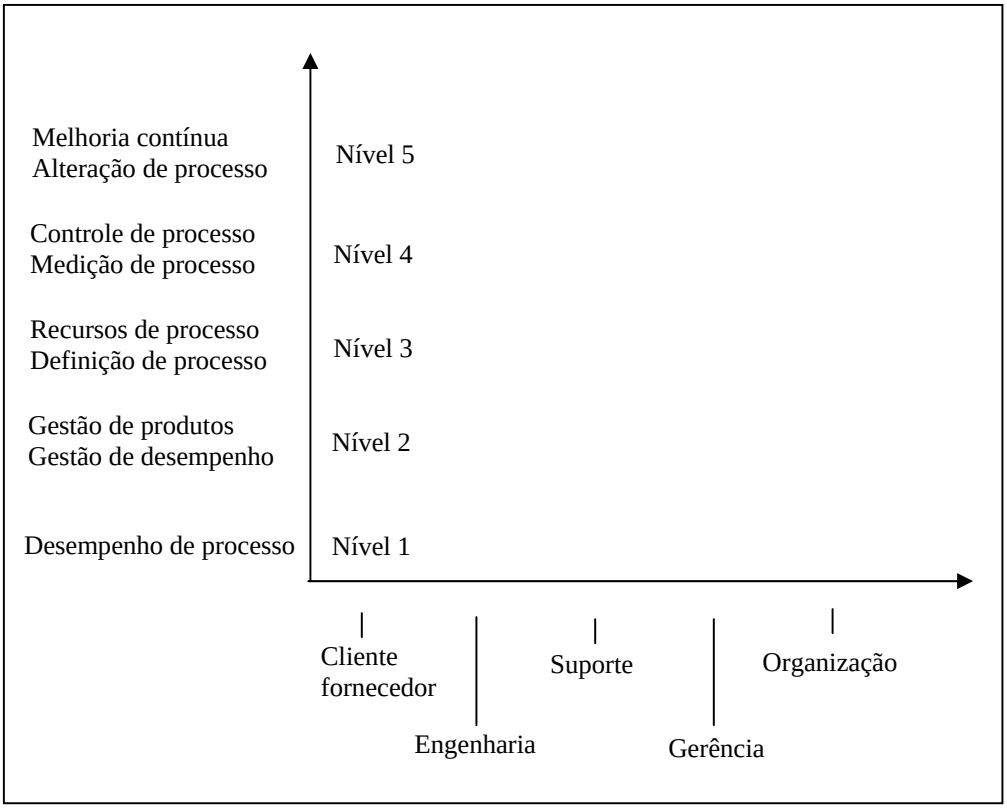


Figura 9. Uma visão da arquitetura bi-dimensional da ISO/IEC 15504 (EMAM, 2000)

No padrão ISO/IEC 15504 existem cinco níveis de capacidade que podem ser classificados do nível 1 (um) ao nível 5 (cinco). Um nível zero também é definido, mas este não é avaliado diretamente. Esses seis níveis são mostrados nas Tabelas 5(a) e 5(b). No nível 1, um atributo é diretamente avaliado e, existem dois atributos para cada um dos quatro níveis restantes.

Tabela 5(a). Uma visão geral dos níveis de capacidade e atributos (EMAM, 2000)

Identificação	Título
Nível 0	**Processo Incompleto** Há falhas gerais para lograr o propósito do processo. Não existem produtos ou saídas de processo facilmente identificáveis.
Nível 1	**Processo Executado** O propósito do processo geralmente é alcançado. A obtenção pode não ser rigorosamente alcançada e seguida. Fases dentro da organização reconhecem que uma ação deveria ser executada e há, em geral, concordância de que essa ação é executada como e quando requerido. Há produtos identificáveis para o processo e esses comprovam a consecução do propósito.
1.1	**Atributo de Desempenho de Processo**
Nível 2	**Processo Gerenciado** O distribuidor de processo constrói produtos de qualidade aceitável dentro da escala de tempo definida. A execução, de acordo com os procedimentos especificados, é planejada e seguida. Execução de produtos conforme os padrões e requisitos especificados. A distinção primária entre o Nível Executado é que a execução do processo é planejada, gerenciada e evolui dentro de um processo definido.
2.1	**Atributo de Gerenciamento de Desempenho**
2.2	**Atributo de gerenciamento do Produto a ser Desenvolvido**
Nível 3	**Processo Estabelecido** O processo é executado e gerenciado, usando um processo definido baseado em bons princípios de engenharia de software. Implementações individuais do processo usam processos aprovados, documentados, e versões de padrão adaptadas. Os recursos necessários para estabelecer a definição do processo são igualmente considerados. A distinção primária do Nível Gerenciado, é que o processo do Nível Estabelecido, é planejado e gerenciado, usando um processo padrão.
3.1	**Atributo de Definição de Processo**
3.2	**Atributo de Controle de Processo**

Conforme explicitado em (ROCHA, 20001), a dimensão de processo “corresponde à definição de um conjunto de processos considerados universais e fundamentais para a boa prática da engenharia de software” e a dimensão de capacidade “corresponde à definição de um modelo de medição com base na identificação de um conjunto de atributos que permite determinar a capacidade de um processo para atingir seus propósitos”.

Tabela 5(b). Uma visão geral dos níveis de capacidade e atributos (EMAM, 2000)

Identificação	Título
Nível 4	**Processo Previsível** O Processo definido é executado de maneira consistente na prática, dentro de limites de controle definidos, para atingir os objetivos. Medidas detalhadas de desempenho são coletadas e analisadas. Isto conduz a uma compreensão quantitativa da capacidade do processo e a uma habilidade melhorada para predizer desempenho. Desempenho é gerenciado objetivamente. A qualidade dos produtos é quantitativamente conhecida. A distinção primária do Nível Estabelecido é que o processo definido é quantitativamente compreendido e controlado.
4.1	**Atributo de Medição de Processo**
4.2	**Atributo de Controle de Processo**
Nível 5	**Processo em Otimização** O desempenho do processo é otimizado para satisfazer as necessidades de negócios futuras e atuais e o processo alcança repetitividade ao encontrar seus objetivos definidos de negócio. Efetividade e objetivos de eficiência são estabelecidos, baseados nos objetivos de negócio da organização. Monitoramento contínuo de processo em preparação para esses objetivos é possibilitado pela obtenção de *feedback* quantitativo e a melhoria é alcançada pela análise dos resultados. Otimizar um processo envolve direcionar idéias inovadoras em tecnologia e mudar processos não efetivos para atingir metas e objetivos. A distinção primária do Nível Predizível é que o processo definido e o processo padrão apóiam refinamento e melhoria contínua baseado em uma compreensão quantitativa do impacto das mudanças nesses processos.
5.1	**Atributo de Alteração de Processo**
5.2	**Atributo de Melhoria Contínua**

Emam (2000) esclarece ainda, que durante a avaliação não é necessário avaliar todos os processos na dimensão do processo. Na realidade, uma organização pode definir uma avaliação para cobrir somente o subconjunto dos processos que são relevantes para seus objetivos de negócios. Portanto, nem todas as organizações que conduzem uma avaliação baseada na ISO/IEC 15504 cobrirão os processos de análise de requisitos.

O esquema de avaliação, conforme explicitado em (EMAM, 2000), consiste de uma escala de consecução, de quatro pontos para cada atributo. Os quatro pontos são designados com T, A, P e N, para Totalmente alcançado; Amplamente alcançado; Parcialmente alcançado e Não

alcançado. Um resumo da definição de cada uma dessas categorias de respostas é dado na Tabela 6.

Um dos benefícios da ISO 15504, conforme enfatizado em (ROCHA, 2001) é a grande "diversidade de formato de apresentação, geralmente com mais riqueza de detalhes que o CMM, que fornece um único número para representar o nível maturidade dos processos da organização". Tal afirmação, por analogia, também vale para o modelo CMMI.

Tabela 6. Escala de avaliação de atributos de quatro pontos, (EMAM, 2000)

Avaliação e Designação	Descrição
Não alcançado - N	Não há evidência de consecução do atributo definido.
Parcialmente alcançado - P	Há alguma obtenção do atributo definido.
Amplamente alcançado - A	Há obtenção suficiente do atributo definido.
Totalmente alcançado - T	Há obtenção total do atributo definido.

Um dos possíveis formatos de apresentação mostra o perfil de capacidade típico para os processos selecionados, ilustrando, para cada processo, o grau de atendimento aos requisitos associados aos níveis de capacidade. Uma outra forma de apresentação pode ser usada quando várias instâncias de processo são avaliadas, normalmente, uma em cada projeto (ROCHA, 2001, p. 33).

Diferentemente do que ocorre no CMMI, o qual associa áreas de processo aos níveis de maturidade, a norma 15504 possui uma abordagem maleável, que compreende duas dimensões, as quais possibilitam o estabelecimento de uma relação entre os níveis de maturidade e qualquer processo do ciclo de vida de software.

Novas técnicas para orientar avaliações de processo têm surgido e a 15504 fornece um *framework*, no qual essas técnicas podem ser validadas e avaliadas em relação aos benefícios para a indústria (ROCHA, 2001).

Na visão de Siy (2001), tem havido muita discussão sobre técnicas orientadas a processo para criar organizações que sejam capazes de desenvolver software de alta qualidade, rapidamente e de baixo custo. O cerne desse propósito é o foco na maturidade do processo de software, aprendizagem organizacional para fomentar a melhoria contínua e acordos contratuais que suportem um foco exclusivo nas atividades de construção de software, em oposição a um foco mais amplo sobre o desenvolvimento do início ao fim de um produto completo.

Concomitante à preocupação com a qualidade dos processos de desenvolvimento, existe a preocupação com a qualidade dos produtos de software. Visando embasar ainda o trabalho em questão, a seguir, são descritos alguns modelos que são referência para a avaliação da qualidade de produto.

2.2 MODELOS DE QUALIDADE DE PRODUTO DE SOFTWARE

Um grande número de tarefas de manutenção de software é influenciado em maior ou menor grau, pela qualidade dos sistemas de software e, muitos dos esforços são destinados a assegurar um alto nível de qualidade do software desenvolvido (REFORMAT, 2003).

Questões relacionadas à qualidade surgem, nos diversos estágios do processo de desenvolvimento. Essa questão corrobora as ações que visam entender as tarefas que efetivamente influenciam a qualidade de software como atividades que devem ser planejadas antes de o software começar a ser desenvolvido. Ou seja, a qualidade de software é um objetivo a ser alcançado e, portanto, deve ser projetada, planejada e levada a efeito dentro de

padrões e critérios estabelecidos que permitam a sua obtenção durante todo o processo de desenvolvimento de software, desde a sua concepção até a fase de manutenção.

A garantia da qualidade de software é uma atividade fundamental para qualquer ocupação ou negócio que gere produtos que sejam destinados ao uso de terceiros e, segundo Pressman (1997), compreende várias tarefas relacionadas a sete atividades, as quais são aplicadas ao longo de todo o processo de engenharia de software, a saber:

- Uma abordagem de gerenciamento da qualidade.
- Tecnologia (métodos e ferramentas) efetiva para engenharia de software.
- Revisões técnicas formais que são realizadas durante todo o processo de software.
- Uma estratégia multicamada de teste.
- Controle de documentação e das alterações nela realizadas.
- Um procedimento para assegurar conformidade com os padrões de desenvolvimento de software (quando aplicado).
- Medição e mecanismos de divulgação.

Existem muitas maneiras de definir qualidade de software. De acordo com o explicitado em (GASS, 2002), qualidade na área de software concerne a uma coleção de sete atributos que um produto de software deve ter:

- Portabilidade. Como construir um produto de software que seja facilmente movido de uma plataforma a outra.
- Confiabilidade. Como construir software de uma maneira que minimize a chance de conter erros.
- Eficiência. Construção de produto de software que economiza tanto no tempo de execução como na usabilidade (engenharia humana), testabilidade, inteligibilidade e modificabilidade.

Para (GAO, 2003), as maiores limitações das abordagens convencionais de SQA (*Software Quality Assurance*) incluem:

- Ausência de diretrizes claras para medição das características de qualidade do produto.

- Ausência de relação clara entre as métricas existentes e as características de qualidade.
- Confiança excessiva no controle de processo.

Existem vários outros diferentes conjuntos de nomes para tais atributos, mas essa é uma boa lista, geralmente aceita e tem pelo menos 30 anos. Esses atributos não são apresentados por ordem de prioridade. Não há uma ordem geral correta com a qual se deve tentar alcançar os atributos de qualidade de software. Isso não significa, contudo, que os atributos não devem ser ordenados. Para qualquer projeto, é de vital importância estabelecer uma lista apropriada desses atributos (GASS, 2002).

Para Khaddaj (2004), qualidade é uma idéia multidimensional refletida em um modelo de qualidade, onde cada parâmetro no modelo define uma dimensão da qualidade.

As categorizações apresentadas são apenas duas possibilidades dentre os vários modelos de qualidade existentes na literatura da área. As Tabelas 7, 8 e 9, a seguir, baseadas em (ALSPAUGH, 2005) mostram três modelos distintos para qualidade, os quais enfatizam os atributos de qualidade do software.

O modelo de fatores de qualidade de software, mostrado na Tabela 7, enfatiza três aspectos importantes de um software: suas características operacionais, sua manutenibilidade no que tange a modificações e sua adaptabilidade a novos ambientes.

Tabela 7. Modelo de fatores de qualidade de software segundo McCall[7]

Categoria	Fatores	Exemplos
Fatores de operação do produto	Corretitude	A resposta correta
	Confiabilidade	Taxas de falhas < que N%, MTBF[8] > P
	Eficiência	MIPS[9], Bytes, bytes por segundo, segundos
	Integridade	Acesso à leitura, acesso a leitura/escrita, permissões, autorizações
	Usabilidade	Quanto o usuário pode fazer, quanto treinamento requer o usuário
Fatores de revisão do produto	Manutenibilidade	Identificar e corrigir falhas, verificar correções
	Flexibilidade	Adaptação a outro uso, extensão de função
	Testabilidade	Qualquer coisa para auxiliar o teste do sistema
Fatores de transição do produto	Portabilidade	Diferentes ambientes
	Reusabilidade	De módulos, em projetos futuros
	Interoperabilidade	Com outros sistemas

Esses fatores de qualidade podem, inclusive, ser categorizados como fatores internos e fatores externos. Os fatores internos são perceptíveis apenas aos programadores/desenvolvedores de software, dentre os quais se destacam: modularidade e legibilidade. Os fatores perceptíveis aos usuários, são denominados fatores externos, os quais incluem, dentre outros: usabilidade, eficiência, portabilidade e integridade.

Conforme pode ser observado, a maioria dos fatores de qualidade existentes no modelo de McCall, faz parte de outros modelos de qualidade, os quais são mostrados nas Tabelas 8 e 9.

A usabilidade, por exemplo, quinto fator de qualidade da categoria dos fatores de operação do produto do modelo apresentado na Tabela 7 e quarto fator da categoria qualidades funcionais, conforme pode ser visto no modelo apresentado na Tabela 8, tem o seu correspondente na categoria facilidade de uso, de acordo com o apresentado na Tabela 9.

---

[7] James McCall.

[8] Do inglês: *Mean Time Between Failure*. Tempo médio entre a ocorrência de falhas.

[9] Do inglês: *Million Instructions Per Second.*

Tabela 8. Modelo de fatores de qualidade de Software, segundo Deutsch e Willis[10]

Categoria	Fatores	Exemplos
Qualidades funcionais	Corretitude	McCall
	Confiabilidade	McCall
	Integridade	McCall
	Usabilidade	McCall
	Capacidade de sobrevivência	Continuidade dos serviços em circunstâncias adversas
Qualidades de desempenho	Corretitude	McCall
	Eficiência	McCall
	Interoperabilidade	McCall
	Segurança	"Coisas ruins nunca acontecem"
Qualidades de Mudanças	Manutenibilidade	McCall
	Flexibilidade	McCall
	Portabilidade	McCall
	Reusabilidade	McCall
	Expansibilidade	Extensão da função (parte do fator de fllexibilidade de McCall)
Qualidades de gerenciamento	Verificabilidade	Verificação eficiente de outros fatores (substitui amplamente Testabilidade)
	Gerenciabilidade	Do processo de desenvolvimento

[10] Michael S. Deutsch e Ronald R. Willis.

Tabela 9. Lista de fatores de qualidade, segundo (ALSPAUGH, 2005)

Categoria	Fatores	Explicação	McCall	Deutsch/Willis
Função (o que faz)	Corretitude	Com respeito aos requisitos especificados	Operação 1	Funcionalidade 1
	Adaptabilidade	Para sua tarefa		
	Precisão	Suficientemente fechada		
	Confiabilidade	Baixa taxa de falha, MTBF	Operação 2	Funcionalidade 2
	Integridade	Segurança contra acesso não autorizado, acidental ou deliberado	Operação 4	Funcionalidade 3
	Capacidade de sobrevivência	Frente a problemas externos		Funcionalidade 5
	Tolerância a	Quanto às suas próprias falhas		
	Recuperabilidade	Após transtornos		
Facilidade de uso	Facilidade de ser entendido	Facilidade de ser entendido		
	Facilidade de ser aprendido	Pelos novos desenvolvedores, usuários		
	Usabilidade	Por todos os usuários	Operação 5	Funcionalidade 4
	Operabilidade	Quão difícil fazê-lo fazer o que é necessário		
Desempenho (quão bom)	Eficiência	Com respeito a recursos: ciclos de processador, memória, espaço em disco, comunicação banda larga	Operação3	Desempenho 2
	Segurança	"Nada de mal acontece"		Desempenho 4
	Interoperabilidade	Com outros sistemas	Transição 3	Desempenho 3
Alteração (por modif.)	Manutenibilidade	Erros encontrados, fixados e testados	Revisão 1	Alteração 1
	Expansibilidade	Adicionar novas capacidades		Alteração 5
	Flexibilidade	Adaptar a novos usos, estender função	Revisão 2	Alteração 2
	Portabilidade	Para diferentes plataformas ou ambientes	Transição 1	Alteração 3
	Reusabilidade	Em outros sistemas	Transição 2	Alteração 4
Gerência	Verificabilidade	Quão fácil decidir se atende as necessidades		Gerenciamento 1
	Testabilidade	Quão fácil para testar	Revisão 3	
	Gerenciabilidade	Quão fácil para gerenciar o ciclo de vida do software		Gerenciamento 2
	Analisabilidade	Localizar erros		
	Estalabilidade	Manter qualidades anteriores, sob modificação		
	Instabilidade	Fácil de instalar		
	Conformidade	Com padrões ou convenções		
	Substitutibilidade	Quão difícil para usar um sistema em lugar de outro		

Para Khaddaj (2004), a popularidade dos modelos de qualidade se reflete no fato de que a ISO 9126, se baseia nesses modelos. Na Tabela 10, são mostrados três modelos de fatores de qualidade, os modelos de McCall, de Boehm e a ISO 9126. É interessante notar que esses modelos se sobrepõem e, com algumas variações, consideram os mesmos fatores de qualidade em seus modelos, embora, alguns sejam mais abrangentes que outros. Entretanto, cabe ressaltar que, em alguns casos, os critérios ou características de qualidade são categorizados em sub-características de qualidade, como é o caso da ISO 9126.

Tabela 10. Modelos de fatores de qualidade (KHADDAJ, 2004)

Critério	McCall	Boehm	ISO 9126
Corretitude	X	X	Manutenibilidade
Confiabilidade	X	X	X
Integridade	X	X	
Usabilidade	X	X	X
Eficiência	X	X	X
Manutenibilidade	X	X	X
Testabilidade	X		Manutenibilidade
Interoperabilidade	X		
Flexibilidade	X	X	
Reusabilidade	X	X	
Portabilidade	X	X	X
Clareza		X	
Modificabilidade		X	Manutenibilidade
Documentação		X	
Resiliência		X	
Capacidade de ser compreendido		X	
Validade		X	Manutenibilidade
Funcionalidade			X
Generalidade		X	
Economia		X	

Para Tian (2004), a qualidade pode determinar o sucesso ou a falha dos produtos de software no mercado competitivo atual. Dentre as muitas características de qualidade, alguns aspectos estão relacionados diretamente à corretitude[11] funcional ou à conformidade com as especificações, enquanto outros tratam da usabilidade, portabilidade e assim por diante. Corretitude – isto é, quão bem o software está em conformidade com os requisitos e

[11] Do inglês: *correctness*.

especificações – é tipicamente o mais importante aspecto da qualidade, particularmente quando operações cruciais dependem do software.

Como pode ser constatado, na Tabela 10, estão contidos os fatores de qualidade ou critérios, utilizados para avaliar a qualidade de um produto de software, relacionados a dois dos modelos de qualidade vistos nas tabelas anteriores e, além desses, o modelo ISO 9126. Fica evidenciado, mais uma vez, que muitos dos critérios coincidem nos três modelos.

No caso do modelo de qualidade ISO/IEC 9126, os critérios de qualidade, ou seja, as características de qualidade, foram decompostas em várias outras sub-características, como pode ser observado na Tabela 11.

A ISO/IEC 9126 define um modelo que compreende seis características e 27 sub-características de qualidade de produto de software e, devido ao fato de ser um modelo de qualidade genérico, pode ser aplicado a qualquer produto de software, tendo somente que ser adaptado a um propósito específico (JUNG, 2004).

A ISO/IEC 9126 também define um ou mais métodos para medir cada uma das suas sub-características. Por exemplo, o nível de qualidade da característica funcionalidade de produto de software pode ser representado pelos valores médios de suas cinco sub-características. Juntas, essas sub-características incluem as propriedades que o padrão postula para constituir a funcionalidade. Contudo, se qualquer das medidas de funcionalidade não for correlata à funcionalidade, o valor agregado pode falhar e não representar completamente a qualidade de funcionalidade tal como define a ISO/IEC 9126.

De acordo com Jung (2004), na ISO/IEC 9126, a ‘satisfação’ implica na ‘capacidade de um produto de software para satisfazer usuários em um contexto específico de uso’. Satisfação nesse sentido, refere-se à resposta dos usuários quando da interação com o produto. Inclui julgamento sobre o uso do produto e não sobre a propriedade do software em si. Em adição à definição da ISO/IEC 9126, satisfação do usuário pode incluir também qualidade do serviço, custo ou a reputação do desenvolvedor (JUNG, 2004).

Tabela 11. Características e sub-características do modelo ISO 9126 (JUNG, 2004)

Características	Sub-características
Funcionalidade	▪ Capacidade de sobrevivência ▪ Precisão ▪ Interoperabilidade ▪ Segurança ▪ Conformidade com a funcionalidade
Confiabilidade	▪ Maturidade ▪ Tolerância a falhas ▪ Recuperabilidade ▪ Conformidade com a confiabilidade
Usabilidade	▪ Facilidade de ser entendido ▪ Facilidade de ser aprendido ▪ Operabilidade ▪ Atratividade ▪ Conformidade com a usabilidade
Eficiência	▪ Comportamento de tempo ▪ Utilização de recursos ▪ Conformidade com a eficiência
Manutenibilidade	▪ Analisabilidade ▪ Modificabilidade ▪ Estabilidade ▪ Testabilidade ▪ Conformidade com a manutenibilidade
Portabilidade	▪ Adaptabilidade ▪ Instalabilidade ▪ Coexistência ▪ Substitutibilidade ▪ Conformidade com a portabilidade

Como pode ser depreendido pelas tabelas apresentadas, várias das sub-características do modelo ISO 9126, são categorizadas nos modelos como características, ou fatores, ou critérios de qualidade nos modelos precedentes.

Mesmo para segmentos do mercado nos quais novas características (*features*) e usabilidade têm prioridade, tal como software para uso pessoal no mercado de massa, corretitude é ainda uma parte fundamental das expectativas do usuário. As falhas, defeitos e erros são com freqüência, coletivamente chamados de defeitos e são chaves para o aspecto de corretitude da qualidade de software (TIAN, 2004).

Embora possa haver concordância entre os profissionais de engenharia de software de que um nível elevado de qualidade é uma meta importante e almejada por fabricantes e clientes, definir o que seja qualidade não se constitui uma tarefa fácil, uma vez que a qualidade é entendida como uma combinação complexa de fatores que variam de acordo com diferentes aplicações e com os clientes que as solicitam, conforme explicita (PRESSMAN,1995). Esses fatores que afetam a qualidade de software podem ser categorizados em dois grandes grupos:

- Aqueles que podem ser medidos diretamente, como, por exemplo, erros/KLOC[12]/unidade de tempo.
- Aqueles que somente podem ser medidos indiretamente como, por exemplo, confiabilidade, usabilidade, manutenibilidade, entre outros.

A atividade de medição deve ser realizada em ambos os casos, dado que é necessário comparar o software com algum dado para que seja possível obter alguma indicação de qualidade.

Desenvolver um bom sistema de software é uma tarefa muito complexa. Com a finalidade de produzir um bom produto de software, várias medições para atributos de qualidade de software devem ser levadas em conta. Medir a complexidade de um sistema tem um papel fundamental no controle e gestão da qualidade de software porque a medição geralmente afeta os atributos da qualidade de software como confiabilidade, estabilidade e manutenibilidade. Assim sendo, a garantia da qualidade de software deve ser tratada tendo em vista as novas estratégias, ferramentas, metodologias e técnicas aplicáveis ao ciclo de vida de desenvolvimento de software (GILL, 2005).

De acordo com o padrão ISO, “um modelo de qualidade é um conjunto de características e relacionamento entre essas características, o qual fornece uma base para especificar os requisitos da qualidade e avaliar a qualidade”.

[12] Do inglês: *Thousand (**K**ilo) of Lines of Code*.

2.2.1 O Padrão ISO 9126

Os computadores têm sido utilizados em amplas e variadas áreas de aplicação e sua operação correta é um aspecto crítico para o sucesso dos negócios e/ou segurança das pessoas. Desenvolver ou selecionar produtos de software de alta qualidade é, portanto, de fundamental importância.

Uma especificação compreensiva e avaliação da qualidade de produto de software é um fator chave para assegurar qualidade adequada. Isto pode ser obtido definindo características de qualidade adequadas, levando em conta o propósito de uso do produto de software (ISO/IEC 9126-1, 2001).

A ISO/IEC 9126 (1991): Avaliação de produto de software – características de qualidade e diretrizes para seu uso, a qual foi desenvolvida para apoiar essas necessidades, define 6 (seis) características de qualidade e descreve um modelo de processo para avaliação de produto de software.

Como as características de qualidade e métricas associadas podem ser úteis não somente para avaliar um produto de software como também para definir requisitos de qualidade e outros usos, a ISO/IEC 9126 (1991) foi substituída por dois padrões relacionados: A ISO/IEC 9126 (Qualidade de Produto de Software) e ISO/IEC 14598 (Avaliação de Produto de Software).

A ISO/IEC 9126 - Qualidade de Produto de Software descreve um modelo de duas partes para qualidade de produto de software:

- Qualidade interna e externa.
- Qualidade em uso.

A primeira parte do modelo específica quatro características de qualidade em uso, mas não elabora o modelo para qualidade em uso sob o nível da característica. Qualidade em uso é o efeito combinado para o usuário, de seis características de qualidade do produto de software.

As características definidas são aplicáveis a todo tipo de software. As características e sub-características fornecem uma terminologia consistente para qualidade de produto de software. Fornecem também um *framework* para especificar requisitos de qualidade para software.

A ISO/IEC 9126 possibilita que a qualidade do produto de software, seja especificada e avaliada a partir de diferentes perspectivas associadas com aquisição, requisitos, desenvolvimento, uso, avaliação, suporte, manutenção, garantia da qualidade e auditoria de software.

Pode ser usada por desenvolvedores, adquirentes, equipe de garantia da qualidade e avaliadores independentes, particularmente aqueles responsáveis por especificar e avaliar produtos de qualidade de software. Exemplos de usos do modelo de qualidade definido pela ISO/IEC 9126, são:

- Validar a inteireza de uma definição de requisitos.
- Identificar requisitos de software.
- Identificar objetivos de projeto (*design*) de software.
- Identificar objetivos de teste de software.
- Identificar critérios para garantia de qualidade.
- Identificar critérios de aceitação para um produto de software completo.

A parte da ISO/IEC 9126 que trata da qualidade de produto de software, pode ser usada em conjunção com a ISO/IEC 15504 (a qual está relacionada com a avaliação do processo de software) para prover:

- Um *framework* para definição de qualidade de produto de software no processo consumidor-fornecedor.
- Suporte para revisão, verificação e validação e um *framework* para avaliação quantitativa da qualidade no processo de suporte.

- Suporte para determinar objetivos de qualidade organizacional no processo de gerenciamento.

Também pode ser usada em conjunto com a ISO/IEC 12207 (relacionada ao ciclo de vida do software) para prover:

- Um *framework* para definição de requisitos de qualidade de produto de software.
- Suporte para a revisão, verificação e validação no suporte aos processos do ciclo de vida.

Pode ser usada em conjunção com a ISO 9001 (a qual está relacionada com processos de garantia da qualidade) para prover:

- Suporte para determinar objetivos de qualidade.
- Suporte para revisão, verificação e validação de projeto (*design*).

Existem várias abordagens para a qualidade. A necessidade de qualidade do usuário inclui requisitos para qualidade em uso, em um contexto de uso específico. As necessidades identificadas podem ser usadas quando da especificação da qualidade externa e interna, usando características e sub-características de qualidade do software.

Avaliar um produto de software para satisfazer as necessidades de qualidade de software é um dos processos no ciclo de desenvolvimento de software onde o produto de software pode ser avaliado medindo-se atributos internos (tipicamente medidas estáticas de produtos intermediários) ou medindo-se atributos externos (tipicamente medindo o comportamento do código quando executado) ou medindo os atributos da qualidade em uso. O objetivo é que o produto tenha o efeito requerido em um contexto particular de uso, conforme mostra a Figura 10.

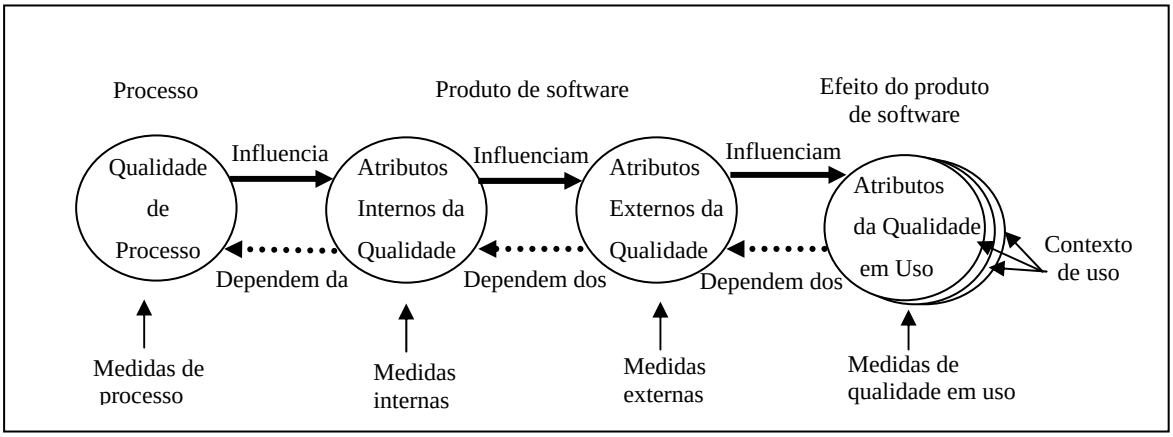


Figura 10. Qualidade no ciclo de vida[13]

A qualidade de processo (a qualidade de qualquer dos processos do ciclo de vida definida na ISO/IEC 12207) contribui para melhorar a qualidade do produto e, a qualidade do produto contribui para melhorar a qualidade em uso. Assim, avaliar e melhorar um processo é um meio para melhorar a qualidade de produto e, avaliar e melhorar a qualidade de produto é um meio de melhorar a qualidade em uso. Similarmente, avaliar a qualidade em uso pode fornecer *feedback* para melhorar um produto e, avaliar um produto pode fornecer *feedback* para melhorar um processo (ISO/IEC 9126, 2001).

Atributos internos apropriados de software são pré-requisitos para se obter o comportamento externo requerido e, o comportamento externo apropriado é um pré-requisito para se obter qualidade em uso, de acordo com o que pode ser visto na Figura 10.

[13] ISO/IEC 9126-1:2001(E) *Software Engineering – Product Quality – Quality Model.*

2.2.1.1 Qualidade de Produto e o Ciclo de Vida

As visões de qualidade interna, qualidade externa e qualidade em uso mudam durante o ciclo de vida do software. Por exemplo, qualidade especificada como requisitos de qualidade no começo do ciclo de vida é, na maioria das vezes, vista a partir da visão do usuário e difere da qualidade do produto entrementes, tal como qualidade de projeto (*design*), a qual é normalmente vista a partir da visão dos desenvolvedores. As tecnologias utilizadas para alcançar o nível de qualidade necessário, tal como especificação e avaliação da qualidade, necessitam apoiar esses diversos pontos de vista. É necessário definir essas perspectivas e tecnologias associadas para qualidade, a fim de gerenciar qualidade corretamente em cada estágio do ciclo de vida.

O objetivo é conseguir a qualidade necessária e suficiente para atender as necessidades dos usuários. Contudo, segundo o contido na (ISO/IEC 9126, 2001), as necessidades expressas por um usuário nem sempre refletem suas reais necessidades dado que:

- Um usuário normalmente não está ciente de suas reais necessidades.
- Necessidades podem mudar após terem sido explicitadas.
- Diferentes usuários podem ter diferentes ambientes de operação.
- Pode ser impossível consultar todos os possíveis tipos de usuário.

Desse modo, os requisitos não podem ser completamente definidos antes do início do projeto (*design*). É necessário ainda, entender as reais necessidades do usuário com o máximo de detalhes possível e representá-las nos requisitos. O objetivo não é necessariamente obter a qualidade perfeita, mas a qualidade necessária e suficiente para cada contexto de uso especificado (ISO/IEC 9126, 2001).

Escalas de medição para as métricas usadas para requisitos de qualidade podem ser divididas em categorias correspondendo a diferentes graus de satisfação de requisitos. Por exemplo, a escala poderia ser dividida em duas categorias: satisfatório e não satisfatório ou, em quatro categorias: requisitos excessivos, alcançados minimamente, aceitáveis e não aceitáveis. As

categorias devem ser especificadas de modo que ambos, usuário e desenvolvedor possam evitar custo desnecessário e cronograma excedente.

2.2.1.2 Usando um Modelo de Qualidade

A qualidade de um produto de software deve ser avaliada usando um modelo de qualidade definido. O modelo de qualidade deve ser usado quando os objetivos de qualidade para produtos de software e produtos intermediários são estabelecidos. Qualidade do produto de software deve ser hierarquicamente decomposta em um modelo de qualidade composto de características e sub-características, as quais podem ser usadas como um *checklist* de questões relacionadas à qualidade. É praticamente impossível medir todas as características e sub-características para todas as partes de um produto de software extenso. Similarmente, não é comumente prático medir qualidade em uso para todos os possíveis cenários de tarefas do usuário. Recursos para avaliação necessitam ser alocados entre os diferentes tipos de medição dependendo dos objetivos de negócios e da natureza do produto e projeto (*design*) de processos (ISO/IEC 9126, 2001).

2.2.1.3 Modelo de qualidade para qualidade interna e externa

Este modelo categoriza os atributos de qualidade em seis características que são subdivididas em sub-características, conforme mostra a Figura 11. As sub-características podem ser medidas por métricas internas e externas.

> Para que a avaliação seja mais efetiva é importante que se utilize de um modelo que permita estabelecer e avaliar requisitos de qualidade e também que o processo de avaliação seja bem definido e estruturado (KOSCIANSKI *et al*, 1999).

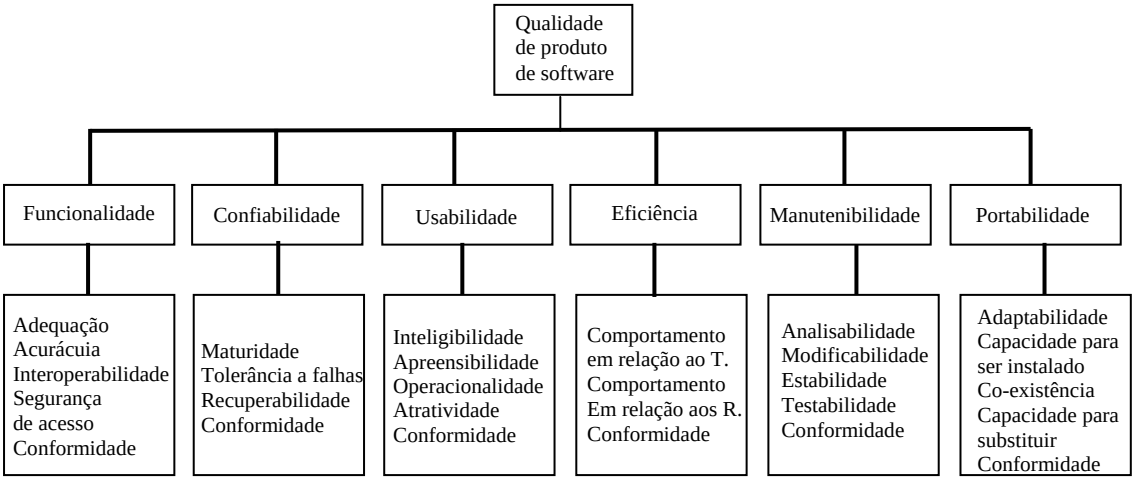


Figura 11. Modelo de qualidade para qualidade interna e externa

Funcionalidade. Capacidade do produto de software prover funções que atendam as necessidades explícitas e implícitas quando o software é usado sob condições específicas.

- **Adequação**. Capacidade do produto de software prover um conjunto apropriado de funções para tarefas e objetivos específicos.
- **Acurácia**. Capacidade do produto de software fornecer resultados e efeitos corretos com a precisão necessária.
- **Interoperabilidade**. Capacidade do produto de software interagir com um ou mais sistemas especificados.
- **Segurança de acesso**. Capacidade do produto de software proteger informações e dados, de modo que pessoas não autorizadas não leiam ou modifiquem essas informações e dados e, que não seja negado a pessoas autorizadas ou sistema, o acesso a essas informações e dados.
- **Conformidade à funcionalidade**. Capacidade do produto de software aderir a padrões, convenções ou regulamentações prescritas similares, relacionadas a funcionalidade.

Confiabilidade. Capacidade do produto de software manter um nível especificado de desempenho quando usado sob condições específicas.

- **Maturidade.** Capacidade do produto de software de evitar falhas, devido a erros no software.

- **Tolerância à falhas.** Capacidade do produto de software manter um nível de desempenho especificado no caso de o software falhar ou violação nas suas interfaces.
- **Recuperabilidade.** Capacidade do produto de software re-estabelecer um nível de desempenho especificado e recuperar o dado diretamente afetado, no caso de uma falha.
- **Conformidade à confiabilidade.** Capacidade do produto de software aderir a padrões, convenções ou regulamentos relacionados à confiabilidade.

Usabilidade. Capacidade do produto de software ser entendido, aprendido, usado e atrativo ao usuário, quando usado sob condições especificadas.

- **Inteligibilidade.** Capacidade do produto de software possibilitar ao usuário entender se o software é adequável e como pode ser usado para determinadas tarefas e condições de uso.
- **Apreensibilidade.** Capacidade do produto de software possibilitar ao usuário aprender a sua aplicação.
- **Operacionalidade.** Capacidade do produto de software possibilitar ao usuário a sua operação e controle.
- **Atratividade.** Capacidade do produto de software ser atrativo ao usuário.
- **Conformidade à usabilidade.** Capacidade do produto de software aderir a padrões, convenções, guias de estilo e regulamentações relacionadas a usabilidade.

Eficiência. Capacidade do produto de software prover desempenho apropriado, relativo à quantidade de recursos usados, sob condições de uso explicitadas.

- **Comportamento em relação ao tempo.** Capacidade do produto de software prover tempos de resposta e de processamento apropriados e *throughput rates* (taxa de produção) quando executar sua função sob condições explicitadas.
- **Comportamento em ralação aos recursos.** Capacidade do produto de software usar quantidades e tipos apropriados de recursos quando o software executa sua função sob condições explicitadas.

- **Conformidade à eficiência.** Capacidade do produto de software aderir a padrões ou convenções relacionadas à eficiência.

Manutenibilidade. Capacidade do produto de software ser modificado. Modificações podem incluir correções, melhorias ou a adaptação do software devido a mudanças no ambiente e nos requisitos e especificações funcionais.

- **Analisabilidade.** Capacidade do produto de software ser diagnosticado quanto a deficiências ou causas de falhas no software ou a partes identificadas para serem modificadas.
- **Modificabilidade.** Capacidade do produto de software possibilitar que uma modificação especificada seja implementada.
- **Estabilidade.** Capacidade do produto de software de evitar efeitos não esperados decorrentes de modificação no software.
- **Testabilidade.** Capacidade do produto de software possibilitar que o software modificado seja validado.
- **Conformidade à manutenibilidade.** Capacidade do produto de software aderir a padrões e convenções relacionadas à manutenibilidade.

Portabilidade. Capacidade do produto de software ser transferido de um ambiente para outro.

- **Adaptabilidade.** Capacidade do produto de software ser adaptado a diferentes ambientes sem requerer ações ou propor outras que não aquelas providas para esse propósito para o software considerado.
- **Instalabilidade - (Capacidade para ser instalado).** Capacidade do produto de software ser instalado em um ambiente especificado.
- **Co-existência.** Capacidade do produto de software co-existir com outro software independente, em um ambiente, compartilhando recursos.
- **Capacidade para substituir.** Capacidade do produto de software ser usado no lugar de outro produto de software, para o mesmo propósito, no mesmo ambiente.
- **Conformidade à portabilidade.** Capacidade do produto de software aderir a padrões e convenções relacionadas à portabilidade.

A cada característica e sub-característica de qualidade de software que influencia a qualidade do software é atribuída uma definição. Para cada característica e sub-característica, a capacidade do software é determinada por um conjunto de atributos internos que pode ser medido. Exemplos de métricas internas são dados na ISO/IEC 9126-3. As características e sub-características podem ser medidas externamente pelo grau de capacidade do sistema que contém o software. Exemplos de métricas externas são dados na ISO/IEC 9126-2 (2001).

2.2.1.4 Modelo de qualidade para qualidade em uso

Qualidade em uso é a qualidade vista pelo usuário de um ambiente contendo software e é medida a partir de resultados do uso do software em um ambiente, e não a partir das propriedades intrínsecas do software. Alcançar a qualidade em uso depende da obtenção da qualidade externa necessária, a qual, por sua vez, é dependente da obtenção da qualidade interna necessária, conforme ilustrado na Figura 10.

Os atributos de qualidade são categorizados em quatro características: efetividade, produtividade, segurança e satisfação, conforme mostra a Figura 12.

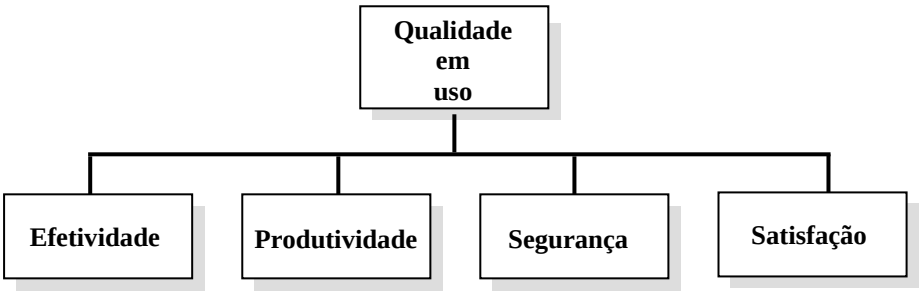


Figura 12. Modelo de qualidade para qualidade em uso

Efetividade. A capacidade do produto de software permitir que o usuário alcance os objetivos especificados com exatidão e inteireza, em um contexto específico de uso.

Produtividade. Capacidade do produto de software possibilitar ao usuário gastar quantidades apropriadas de recursos em relação à efetividade obtida em um contexto de uso específico.

Segurança. Capacidade do produto de software atingir apropriadamente, níveis aceitáveis de riscos de danos a pessoas, negócios, software ou ambiente em um contexto de uso específico.

Satisfação. Capacidade de o software satisfazer os usuários em um contexto específico de uso.

Essencialmente, existem duas abordagens que podem ser seguidas para assegurar qualidade de produto, uma assegura o processo por meio do qual o produto é desenvolvido e a outra avalia a qualidade do produto final. Ambas são importantes e ambas requerem a existência de um sistema para gerenciar a qualidade (ISO/IEC 9126-1, 2001).

2.3 QUALIDADE NO BRASIL

A preocupação com a qualidade transcendeu a academia, os institutos de pesquisas e passou a ser uma questão estratégica para a política econômica e tecnológica do País. Desde a década passada vem ganhando espaço e conquistando seu lugar de destaque no cenário político brasileiro, em função das grandes e rápidas mudanças no cenário da economia mundial, que têm forçado países como o Brasil a adotar uma estratégia de desenvolvimento marcada pela associação de “capital humano, tecnologia e flexibilidade institucional” (SOFTEX, 2001).

Segundo o Ministério de Ciência e Tecnologia (MCT 2002, p. 1)[14], o Brasil vem “desenvolvendo uma política estruturante para o setor de informática, apoiada em três pontos fundamentais”, a saber: o primeiro ponto está relacionado com o desenvolvimento de

[14] Disponível em: www.mct.gov.br/sepin/pni/pni.htm.

hardware, buscando a inovação tecnológica; o segundo, está baseado numa política de desenvolvimento de software e, o terceiro, está relacionado com a reestruturação e desenvolvimento do setor de Micro-eletrônica.

Para o MCT (2002, p. 1), esses pontos em conjunto ou isoladamente, visam "criar condições de modificação do cenário das tecnologias de informação no País, no seu conceito de produção, viabilizando a participação do setor no mercado internacional".

Ainda segundo o MCT (2002, p. 1), o "grande desafio é a inserção do Brasil na nova 'economia digital', onde o setor de software desponta como agente crítico da participação brasileira nesta economia globalizada e transnacional, em cenário altamente competitivo".

> A atual política de software propõe uma nova abordagem para a formação de recursos humanos na área, com implantação de fábricas de software como regime de complementação da formação, uma auto regulamentação desafiadora para o setor, ações relacionadas com a estruturação de marketing internacional e programa de parcerias entre empresas dos diversos países, para incentivar a expansão de mercado desta indústria, criando uma gestão em conjunto com o setor privado e acadêmico, que permite um constante direcionamento das ações num setor onde o dinamismo é a principal característica (MCT, 2002, p. 1).

Partindo de parcerias com órgãos governamentais, empresas privadas e instituições de ensino, a política brasileira para este setor, além de buscar a excelência, no caso da produção de software, visa também fomentar o desenvolvimento de outros setores econômicos, criando ainda, oportunidades para reduzir as diferenças sociais do País. A Figura 13 (SOFTEX, 2001, p.1)[15], ilustra a parceria com os diversos segmentos da sociedade, que segundo a Sociedade SOFTEX garante o sucesso de um projeto coletivo e dá uma idéia da importância estratégica da política adotada pelo governo em conjunto com esses segmentos da sociedade, relativa ao setor de Tecnologia da Informação.

[15] Documento disponível em: http://www.mct.gov.br/upd_blob/2700.pdf.

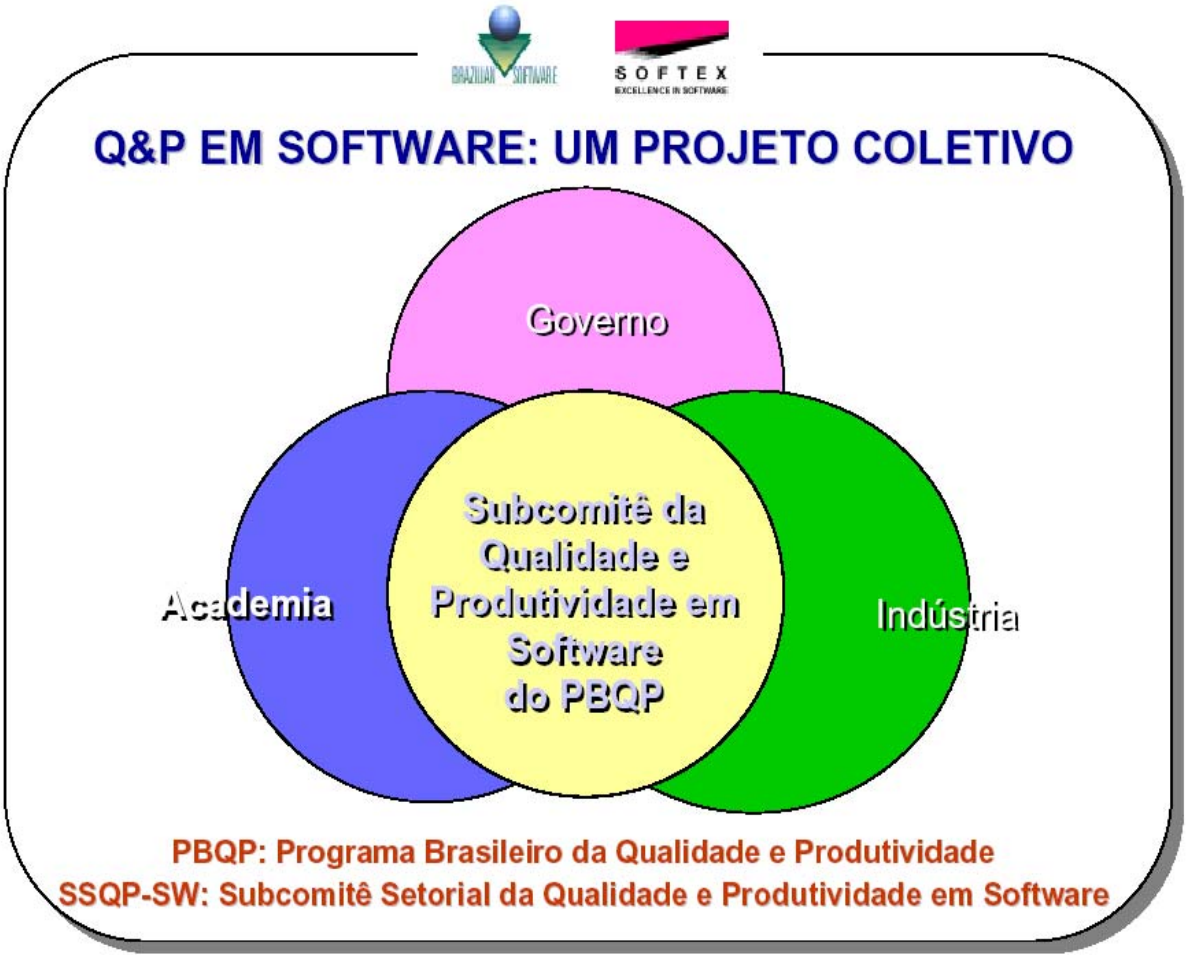


Figura 13. Q&P em software: um projeto coletivo [16]

A Associação para Promoção da Excelência para o Software Brasileiro (SOFTEX)[17] "é responsável pela gestão do Programa SOFTEX, um dos mais importantes instrumentos de apoio à produção e comércio do software brasileiro". Suas ações visam promover a competitividade da Indústria de Software, Internet e Comércio Eletrônico no país e a qualificação de recursos humanos para o setor.

[16] Fonte: Documento "Qualidade e Produtividade em Software no Brasil – O Sucesso de um Projeto Coletivo". Disponível em http://www.mct.gov.br/upd_blob/2700.pdf.

[17] Disponível em: http://www.softex.br/cgi/cgilua.exe/sus/start.htm?sid=2.

A SOFTEX está presente em quase todo o território brasileiro, por meio de uma rede de agentes – os quais prestam apoio operacional às empresas de software – que juntamente com instituições parceiras promovem ações tecnológicas e de mercado para capacitar as empresas de software.

Para a SOFTEX[18], "o caminho para a excelência do software brasileiro passa pela qualidade. É por isso que avaliar os indicadores de qualidade e produtividade das empresas sempre foi uma das preocupações da SOFTEX". Para tanto, conta com o apoio de equipe do Programa Brasileiro de Qualidade e Produtividade em Software.

O PBQP foi criado com o objetivo de incentivar a indústria nacional de software a atingir padrões de Q&P (Qualidade e Produtividade). "O PBQP organiza pesquisas, fóruns e *workshops* para promover a troca de experiências, gerar novos negócios e conscientizar os empresários da importância da qualidade (grifo nosso) para o setor" (SOFTEX, 2005, p. 1). Tem também o propósito de ajudar o esforço de modernização da indústria por meio da promoção da qualidade e produtividade, em busca de padrões internacionais e aumento de competitividade.

A cada dois anos, as empresas de software passam por um diagnóstico da qualidade organizado pela SEPIN/MCT, onde são analisadas a qualidade e a produtividade das organizações por meio de pesquisa direta. "Boa parte desse resultado pode ser creditado ao Sistema SOFTEX que, por meio de seus Agentes, realiza ações locais para qualificar empresas com as certificações mais importantes ao setor" (SOFTEX, 2005, p. 1).

Como parte desse diagnóstico, vem sendo realizada uma pesquisa desde a década de 90, visando avaliar o conhecimento e a aplicação de normas e modelos de qualidade, bem como a certificação das empresas segundo essas normas e modelos.

[18] Disponível em: www.softex.cgi/cgilua.exe/start.htm?sid=33.

Pesquisa sobre “Qualidade e Produtividade no Setor de Software Brasileiro - 2005”, com 701 empresas participantes e com 488 formulários completos, possibilitou o levantamento de informações relevantes sobre empresas produtoras de software, conforme mostram as Figuras 14, 15, 16, 17, 18, 19 e 20.

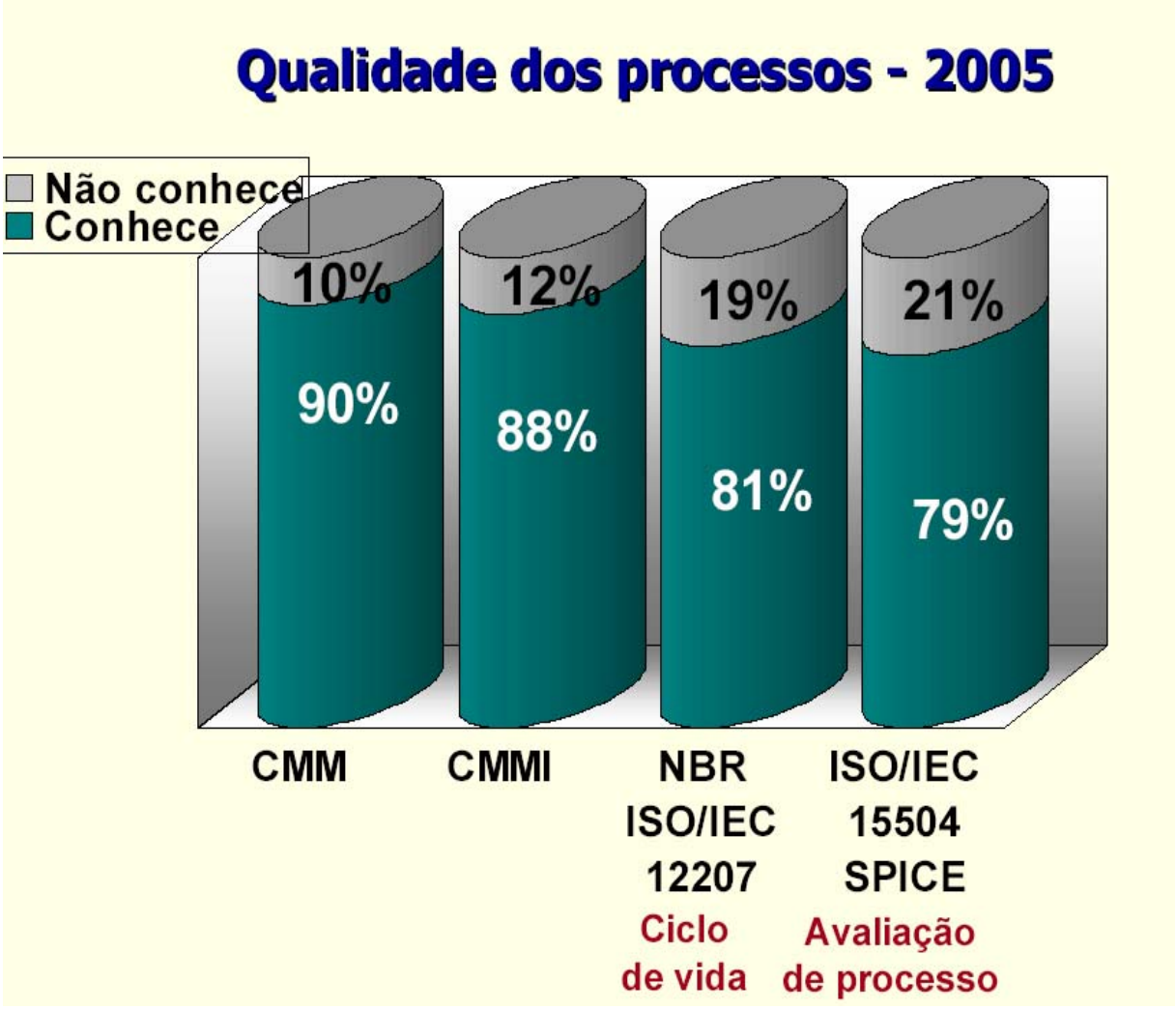


Figura 14. Qualidade dos processos [19]

Por meio da Figura 14, que ilustra a situação das empresas pesquisadas com relação ao conhecimento – no que tange a qualidade dos processos - das mesmas com respeito aos

[19] Disponível em: http://www.mct.gov.br/upd_blob/2674.pdf.

modelos e normas que tratam da melhoria da qualidade de processos de software, pode-se constatar que a grande maioria do universo de empresas pesquisadas tem conhecimento dos modelos CMM, CMMI e das normas ISO 12207 e 15504.

A Figura 15 ilustra – ainda em relação à qualidade dos processos - o percentual de empresas que conhecem, porém não usam; começam a usar; utilizam sistematicamente e; não conhecem processos para melhoria da qualidade.

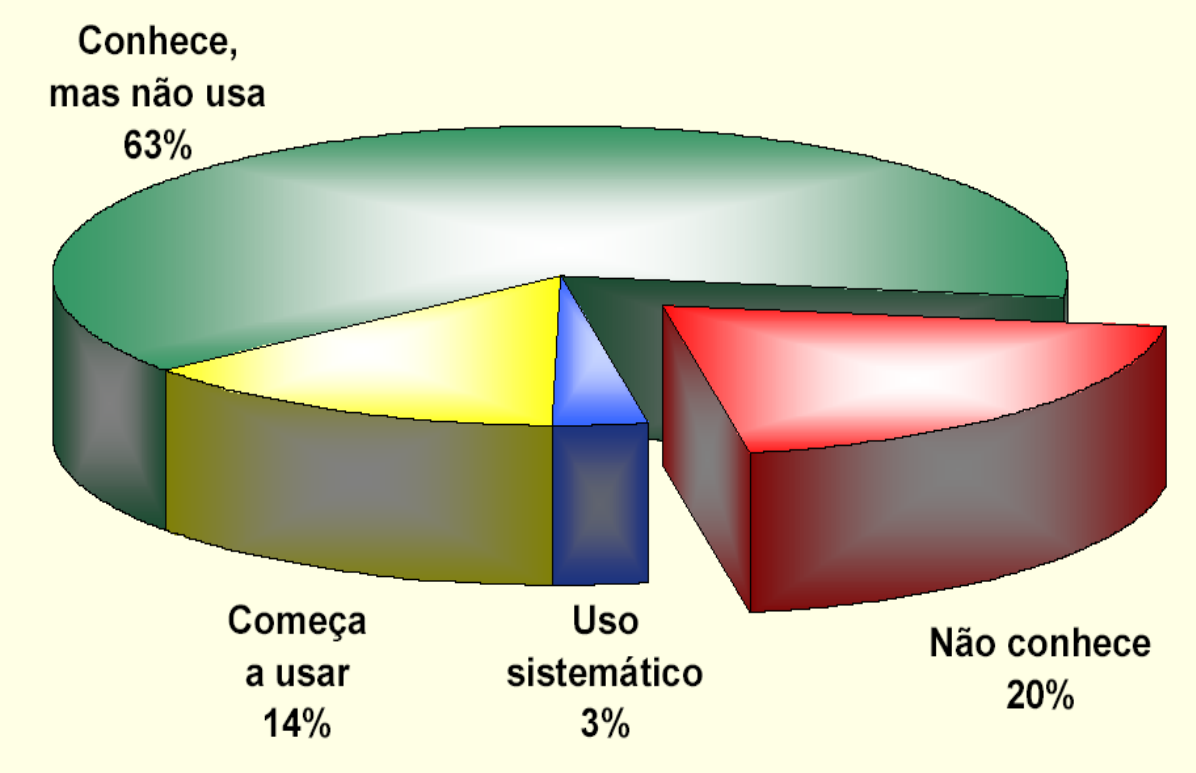


Figura 15. Melhoria do processo de software[20]

Como pode ser constatado, numa rápida comparação entre as Figuras 14 e 15, não obstante o elevado percentual de empresas que conhecem os modelos e normas para melhoria da

[20] Disponível em: http://www.mct.gov.br/upd_blob/2674.pdf.

qualidade de processos, o percentual de empresas que utilizam tais modelos e normas para a melhoria da qualidade de seus processos corresponde ao percentual mais baixo.

Em relação ao percentual referente ao conhecimento das empresas com relação aos modelos e às normas para melhoria da qualidade de processo, pode-se perceber claramente por meio das Figuras 16, 17 e 18, que o percentual de conhecimento das empresas teve um crescimento grande e rápido. Em 1995, 86% (oitenta e seis por cento) das empresas não conheciam o CMM, contudo, em dez anos esse percentual baixou significativamente em 2005, para 10% (dez por cento), conforme ilustrado na Figura 16.

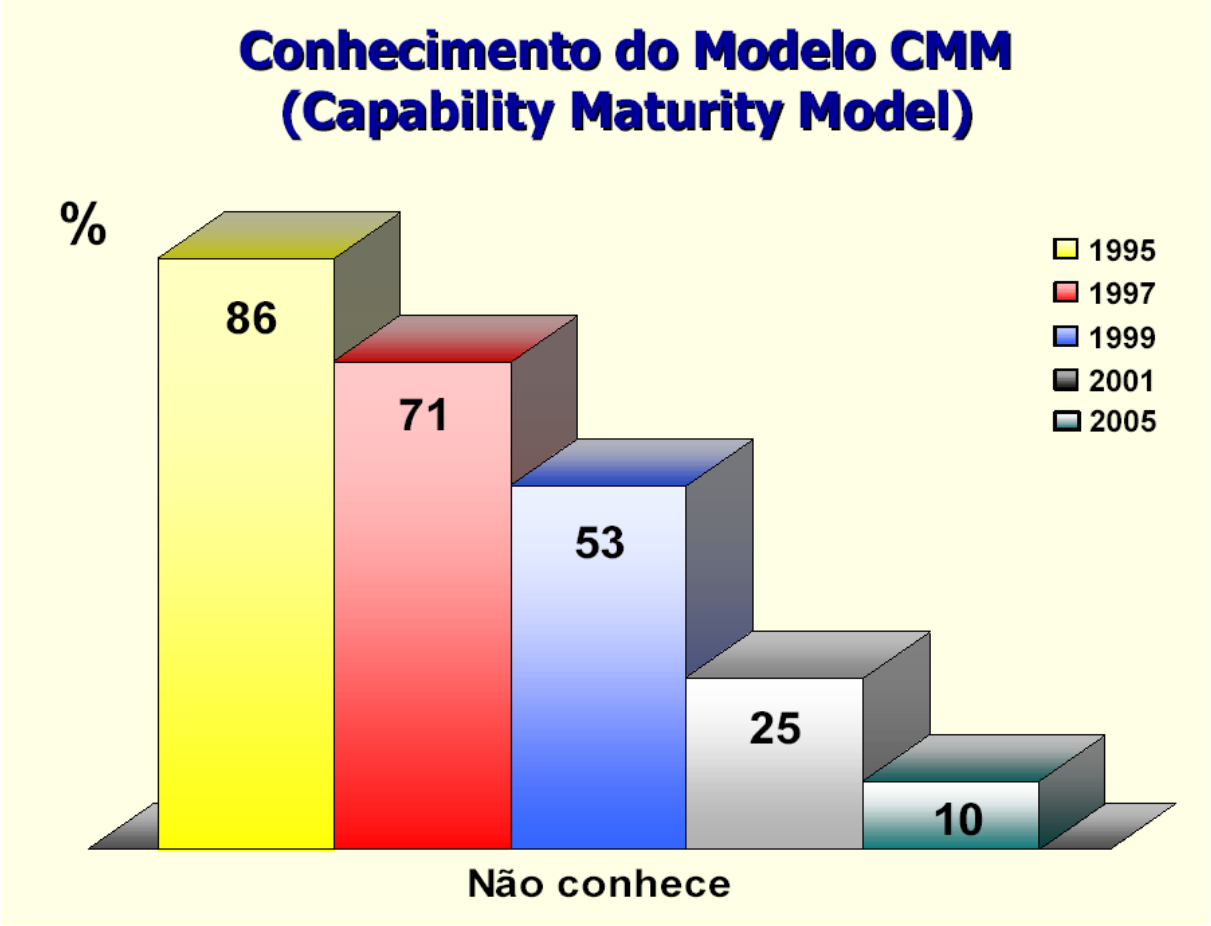


Figura 16. Conhecimento do CMM[21]

[21] Disponível em: http://www.mct.gov.br/upd_blob/2674.pdf.

O mesmo pode ser verificado em relação ao conhecimento das empresas com respeito às normas relacionadas à qualidade de processos, tais como a ISO 12207 (processos de ciclo de vida de software) e a ISO 15504 (avaliação de processo de software).

Conforme pode ser visto por intermédio da Figura 17, num período ainda mais curto, houve uma diminuição significativa - de 75% em 1997 para 19% - do percentual de empresas que não possuíam conhecimento da norma ISO 12207, sendo que os decréscimos mais significativos ocorreram entre o ano de 1999 e o ano de 2001 e entre o ano de 2001 e o ano de 2005.

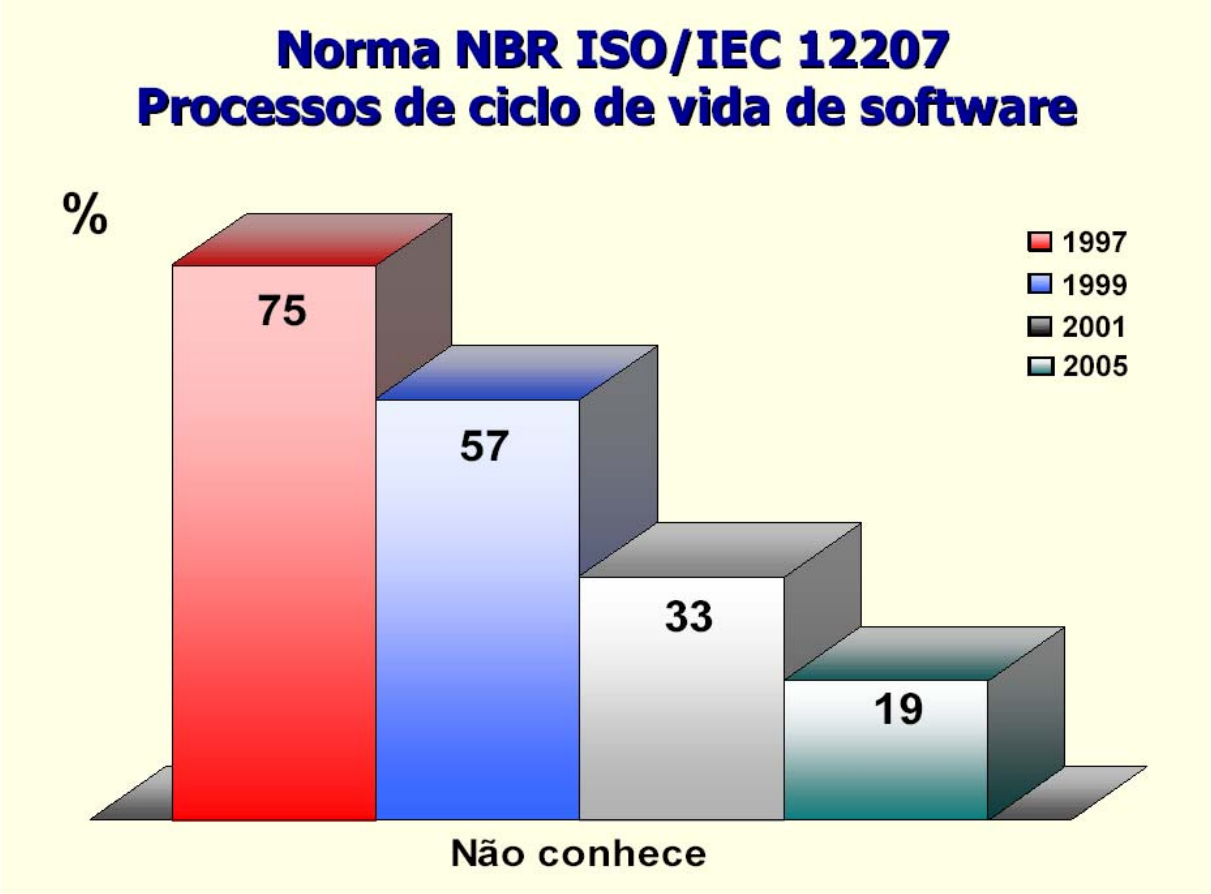


Figura 17. Processos de ciclo de vida de software[22]

[22] Disponível em: http://www.mct.gov.br/upd_blob/2674.pdf.

A Figura 18, por sua vez, ilustra o decréscimo ocorrido em relação ao conhecimento das empresas pesquisadas, quanto à norma ISO 15504, ao longo de um período que abrange os anos de 1997 a 2005. Como se pode observar, em 1997, 82% (oitenta e dois por cento) das empresas não tinha conhecimento da norma ISO 15505, enquanto que em 2005, apenas 21% (vinte e um por cento) das empresas desconheciam tal norma. Pode-se observar também, que a diminuição mais significativa, ocorreu entre os anos de 1999 e 2001, cujo decréscimo de 30% (trinta por cento) do percentual de empresas que não possuíam conhecimento da norma ISO 15504.

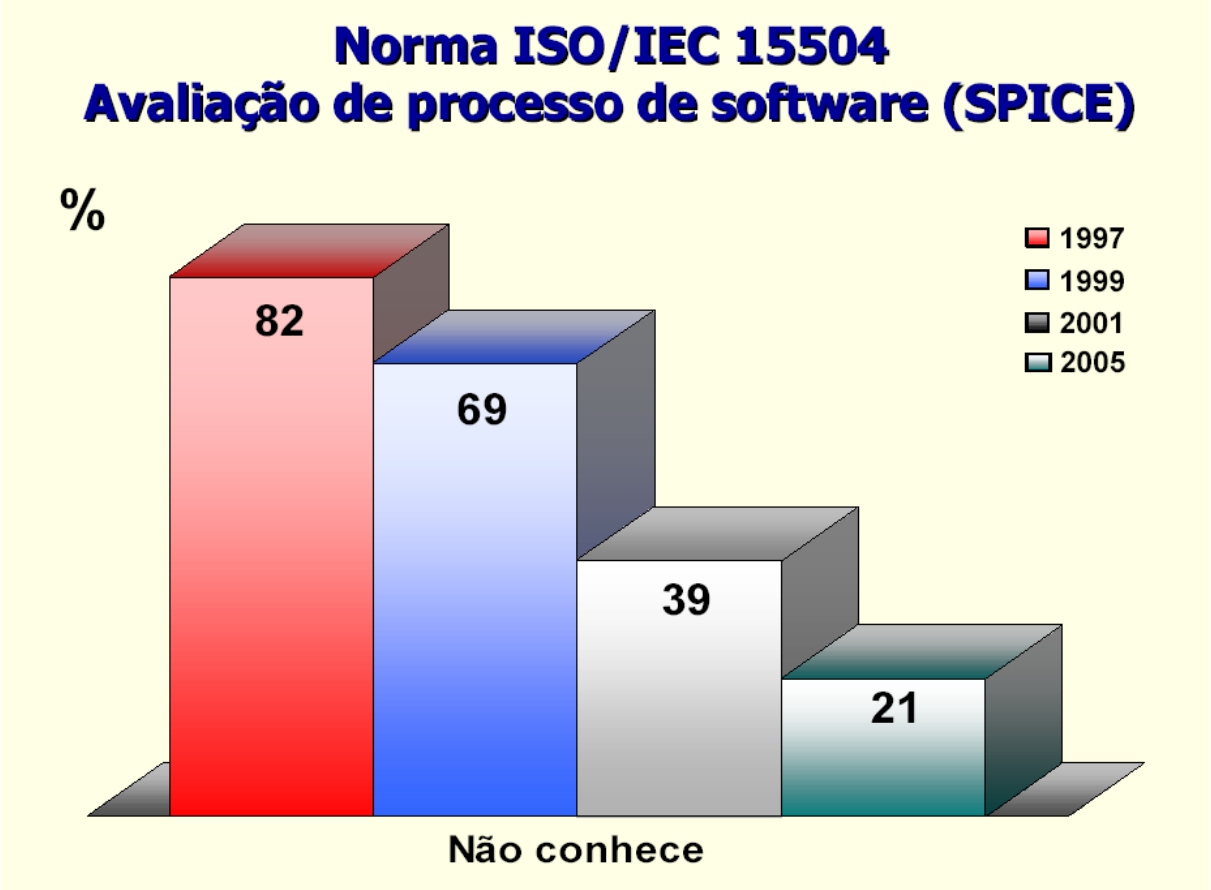


Figura 18. Avaliação de processo de software[23]

Em relação à qualidade de produtos, a Figura 19, ilustra o percentual obtido, no que concerne ao conhecimento das normas para melhoria da qualidade de produtos de software por parte

[23] Disponível em: http://www.mct.gov.br/upd_blob/2674.pdf.

das empresas contempladas por dita pesquisa e, novamente, o percentual referente ao conhecimento das empresas referente a essas normas é alto. Contudo, pode-se observar que o percentual se mantém praticamente o mesmo para todas as normas, com uma pequena variação, com respeito à norma 9126-1. É interessante e importante ressaltar, que em 2002, o percentual das empresas que não conheciam essa norma, era de 34% (trinta e quatro por cento), portanto, àquela época, 2% (dois por cento) a mais de empresas desconheciam as normas em questão.

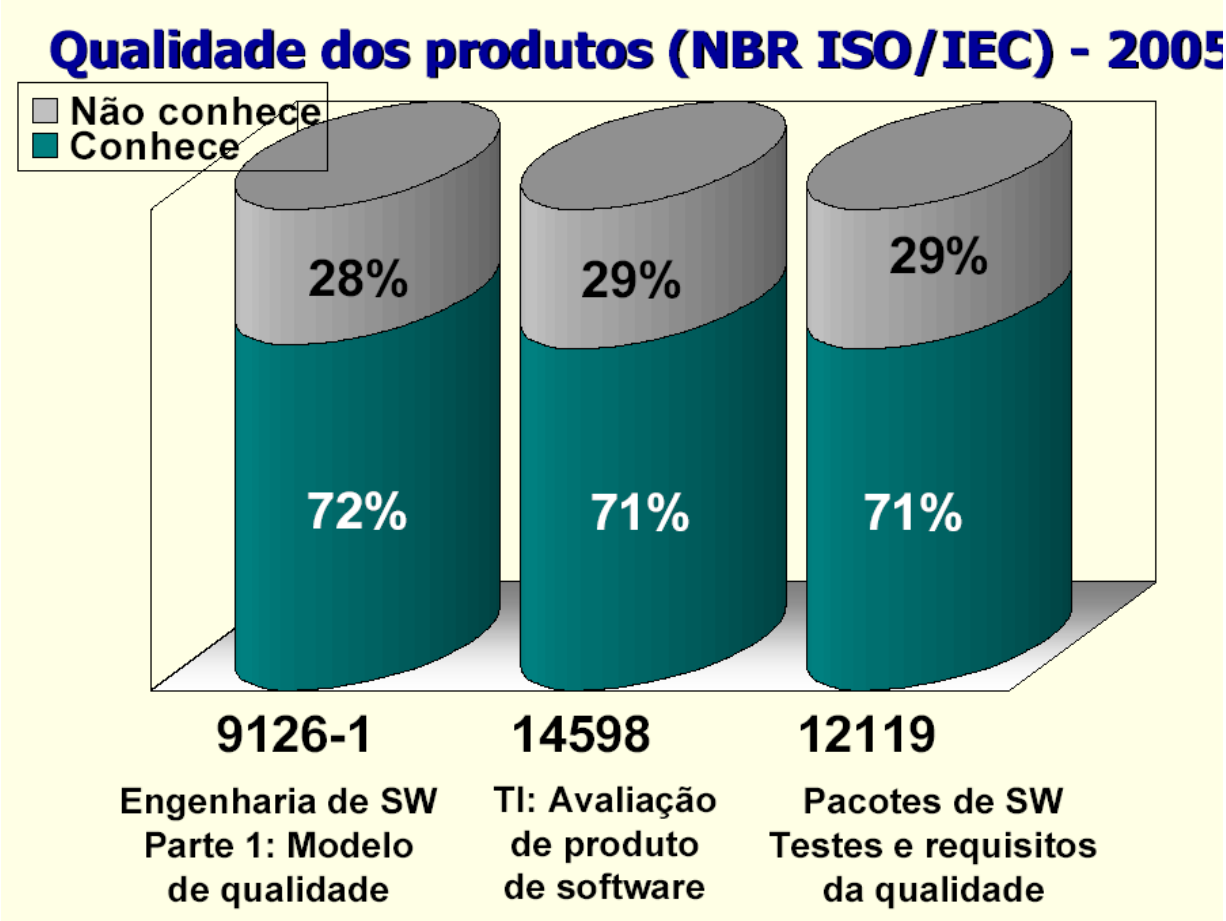


Figura 19. Qualidade dos produtos[24]

[24] Disponível em: http://www.mct.gov.br/upd_blob/2674.pdf.

Com relação à Qualidade e Produtividade no Setor de Software Brasileiro, a pesquisa citada anteriormente, é uma “pesquisa direta, junto a organizações desenvolvedoras de software no Brasil, visando identificar o estágio atual da qualidade e produtividade neste setor e, fundamentalmente, direcionar ações dos diversos agentes responsáveis pela formulação e execução da Política de Software no Brasil” (MCT/SEPIN, 2001, p.2).

No que tange ao Diagnóstico da Qualidade em Software no Brasil – A Evolução, um dos objetivos dessa pesquisa é fazer uma “análise da evolução histórica da qualidade no setor de software brasileiro cobrindo indicadores obtidos a partir de resultados de pesquisas diretas realizadas a cada dois anos, desde 1993, junto a empresas desenvolvedoras de software no Brasil” (MCT/SEPIN, 2001, p.4).

A Figura 20 ilustra a evolução, desde 1997 até julho de 2005, do número de empresas qualificadas segundo o CMM. Nota-se que houve um acréscimo de 4.900% (quatro mil e novecentos por cento) entre os anos de 1997 e 2005 e, com exceção de 98, 99 e 2000 cujo percentual se manteve estável, houve um crescimento vertiginoso nos demais anos. A diferença, em termos de quantidade, de 1997 para 2005 é bastante significativa e mostra que o setor vem envidando esforços em busca da qualidade.

A integração de agentes, parcerias e empresas associadas, aliadas à estreita relação da SOFTEX com o MCT, evidenciam claramente a preocupação e os esforços envidados no sentido de alçar o setor de software à categoria de bem estratégico para a economia e política do País.

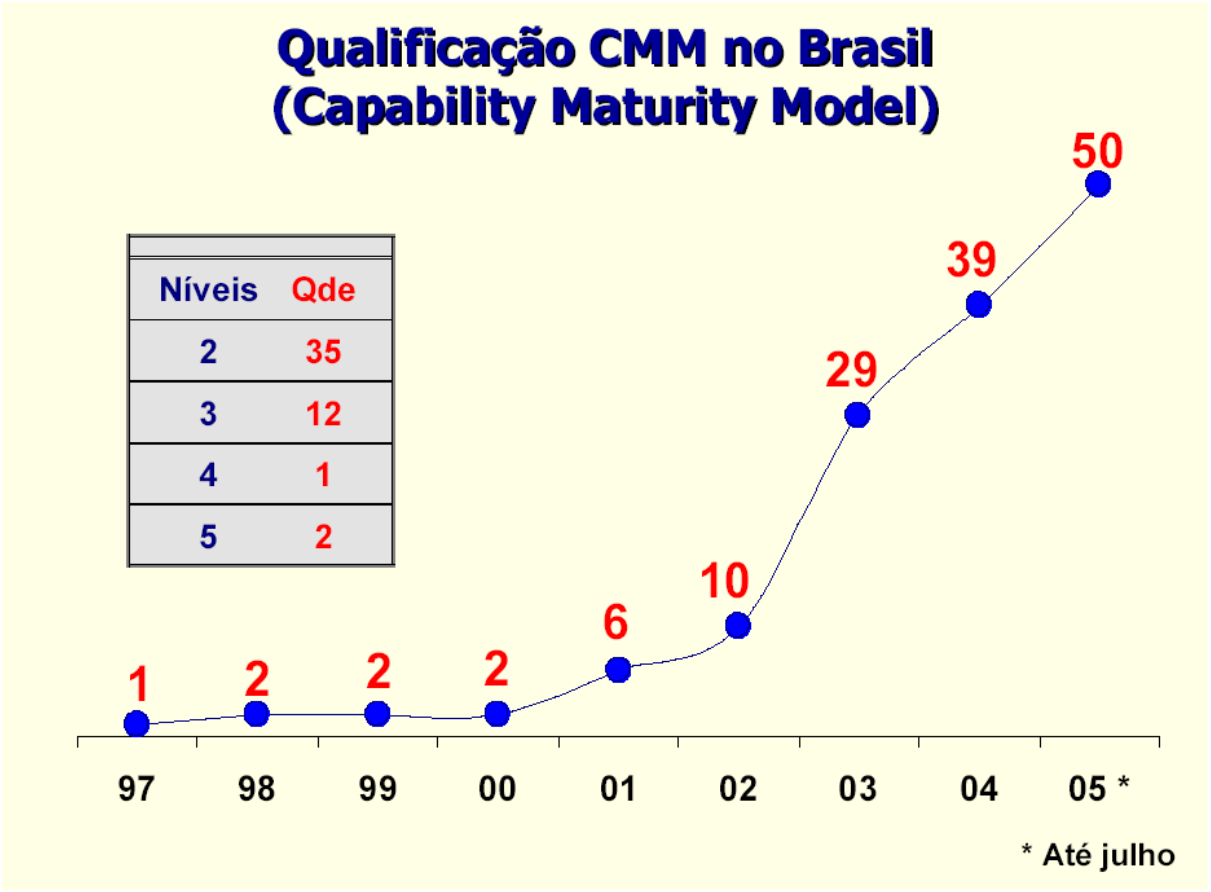


Figura 20. Qualificação CMM no Brasil[25]

O MCT, por intermédio da Secretaria de Política de Informática (SEPIN), está empenhado em organizar as Pesquisas de Qualidade e Produtividade no Setor de Software Brasileiro. Essa pesquisa foi concebida em 1993 no âmbito do Programa Brasileiro da Qualidade e Produtividade de Software – PBQP Software, o qual tem os seguintes objetivos:

> Atingir padrões internacionais de qualidade e produtividade no setor de software e estimular a adoção de normas, métodos, técnicas e ferramentas a qualidade e da engenharia de software, promovendo a melhoria da qualidade dos processos, produtos e serviços de software brasileiro, de modo a tornar as empresas mais capacitadas a competir em um mercado globalizado (MCT, 2002, p. 2)[26]

[25] Disponível em: http://www.mct.gov.br/upd_blob/2674.pdf.

[26] Fonte MCT: Disponível em www.mct.gov.br/sepin/Dsi/PBQP/ObjetivoPBQPSoftware.htm.

O PBQP é composto por voluntários ligados ao governo, academia e setor privado, interessados na melhoria da qualidade e produtividade do software brasileiro. Busca-se, por meio do PBQP, especificamente, a melhoria contínua do grau de satisfação dos seus clientes, da qualidade de vida no trabalho e no país, e da lucratividade e competitividade das empresas brasileiras.

O trabalho realizado ao longo de mais de uma década, em parceria com vários segmentos representantes da classe política, econômica e acadêmica, fruto de uma política implantada para o setor de software, balizam o MCT a realizar prognósticos alvissareiros para esse setor no Brasil.

> Há bases de dados históricos nacionais que permitem afirmar que o Brasil tem projetos e estratégias na direção do alcance de padrões internacionais efetivos em qualidade e produtividade no setor de software, existindo evidências de que a qualidade de software no País tem apresentado tendência de melhoria contínua (MCT, 2002, p. 2)[27].

Os resultados apresentados demonstram que os esforços conjuntos dos diversos segmentos envolvidos na a política têm sido significativos para o setor de software no Brasil.

[27] Fonte: MCT. Disponível em: http://www.mct.gov.br/Temas/info/Dsi/Quali2001/Apresentacao2001.htm.

3. FÁBRICA DE SOFTWARE

Dado que a metodologia proposta está inserida no contexto de fábricas de software se faz necessário apresentar alguns conceitos e abordagens que permeiam este tema.

A adoção do conceito de fábrica de software e sua conseqüente implantação por uma empresa envolvem grandes e sensíveis mudanças estruturais e organizacionais. Os profissionais envolvidos na produção de software, além de incorporar a cultura voltada para a qualidade, têm que ter aptidão para o envolvimento com uma nova forma de atuar uma vez que serão responsáveis pelos produtos por eles produzidos e não apenas responsáveis pelo processo de desenvolvimento de ditos produtos.

As constantes mudanças no mundo dos negócios vêm provocando, ao longo dos tempos, uma nova ordem na indústria de software fomentada por clientes conhecedores de suas necessidades, altamente críticos quanto às características dos produtos que buscam adquirir, pela grande quantidade de empresas que atuam na área e que estão constantemente inovando suas estratégias para se manterem, não apenas competitivas, mas na dianteira no mercado de software e, também, pela diversidade de produtos liberados para o mercado com qualidade cada vez maior.

A ampla e crescente utilização de software nos diversos setores da sociedade, aliada à existência de clientes cada vez mais exigentes, mais cientes de suas necessidades e que, por isso mesmo, demandam produtos de software de alta qualidade, tem levado as organizações a dedicar especial atenção à produtividade e à qualidade do produto e do processo de construção dos produtos de software numa tentativa de eliminar os problemas que envolvem a construção de produtos de software.

São muitos e das mais diversas origens os problemas que conduzem a projetos falhos e não confiáveis. Dentre as principais causas, destacam-se problemas relacionados à falta de compreensão ou à interpretação incorreta das reais necessidades do usuário, falhas e ou

limitações nas atividades de verificação e validação e cronogramas e orçamentos inflexíveis que não condizem com a realidade da organização no que tange aos recursos humanos, tecnológicos, físicos e materiais disponíveis, o que leva ao não cumprimento de prazos e a altos custos, comprometendo sobremaneira a produtividade e a qualidade do processo e do produto.

Diante desses problemas, a preocupação das organizações é constante, o que faz com que as mesmas tenham uma postura visionária e procurem não apenas tentar solucionar tais problemas, mas também e, principalmente, se antecipar às novas tendências técnicas e tecnológicas do mercado, visando a solucionar e melhorar continuamente os seus processos e os seus produtos, resultantes de suas estratégias de negócios.

Uma maneira de tentar diminuir os problemas citados consiste em implementar e fazer uso de uma metodologia de desenvolvimento bem definida. Nesse sentido muitas empresas desenvolvedoras de produtos de software estão adotando uma nova estrutura organizacional orientada para projetos, denominada fábrica de software. O conceito de fábrica de software “é baseado na idéia de estabelecer uma linha de produção de soluções capazes de satisfazer as necessidades dos diferentes clientes, tendo em vista a formalização de produtos e de atividades” (CABRAL, 2004; MARQUES, 2004).

Para Herzum (2000), uma vez que a fábrica tenha sido suficientemente instituída/constituída (instalada) a organização pode se concentrar, primeiro, em produzir e colocar em funcionamento eficientemente a fábrica de software e, segundo, conduzir a organização a ótimos níveis de produtividade. Fazer uma fábrica de software funcionar eficientemente requer considerações adicionais, incluindo organização, gerenciamento de conhecimento, distribuição e produtos e considerações sobre suporte.

A fábrica de software é onde a construção de software é conduzida a níveis eficientes. Preocupações relacionadas incluem aspectos organizacionais tais como gerenciamento de pessoas e o acompanhamento de aprendizagem da organização, gerenciamento de conhecimento e o processo para apoiar melhoramento contínuo.

A indústria de software é, provavelmente, a única maior indústria que em muitos casos produz a fábrica, os processos de desenvolvimento e os produtos construídos ao mesmo tempo. E na realidade, o que se faz na maior parte do tempo é continuamente construir e reconstruir a capacidade de desenvolver as ferramentas, o ambiente de desenvolvimento tanto como os processos – o processo do todo a cada projeto desenvolvido. Esta é a principal razão para o elevado custo de desenvolvimento, o tempo demasiado longo para vendas e os riscos e a complexidade de desenvolvimento (HERZUM, 200).

Os principais objetivos de uma fábrica de software, segundo Cabral (2004) são:

- Aumentar a qualidade do produto.
- Adequá-lo às necessidades do cliente.
- Reduzir custos.
- Fornecer um processo de desenvolvimento de software mais produtivo tornando possível a construção de produtos estáveis, expansíveis e flexíveis.

Existem várias abordagens de fábrica de software, o que evidencia que este não é um conceito fechado e pode ser adaptado às necessidades das diversas organizações que enfrentam um mercado dinâmico e exigente, onde a produção de produtos de qualidade prescinde a melhoria contínua de processo e de produtos, uma vez que produtividade com qualidade tornou-se um diferencial, onde o equilíbrio entre custo e qualidade determina a sobrevivência das organizações num mercado em constante transformação.

O termo fábrica de software assinala um compromisso de longo prazo e esforços integrados – superior ao nível de projetos individuais – para acentuar operações de software. Isso, além de poderoso, é também uma idéia necessária, levando-se em conta os desafios envolvidos na profissionalização das operações de software (AAEN, 1997).

O termo fábrica pode ser usado para denotar uma ou mais facilidades para construir o local de algum tipo de produção. Para muitas pessoas o conceito de fábrica de software também implica uma maneira particular de organizar as tarefas com considerável especialização de trabalho, formalizar o comportamento e padronizar o processo de trabalho.

A fábrica é uma organização composta por pessoas engajadas em um esforço comum, o trabalho é de um modo ou de outro, organizado, a padronização é usada para a coordenação e formalização e a sistematização é importante, mas sempre haverá várias opções para o projeto (*design*) de uma fábrica de software em particular, (AAEN, 1997).

Existem várias e contrastantes visões sobre fábrica de software e todas possuem pontos fortes e fracos. As abordagens apresentadas a seguir, são: a abordagem Japonesa para uma organização de software industrializada; a abordagem Européia para uma fábrica de software genérica, (FERNESTRÖM, 1992); a abordagem Norte Americana, para fábrica de componentes baseada em experiência (SEAMAN, 2003), a abordagem Norte Americana para organização madura de software e, por fim, a Fábrica de Software proposta pela Microsoft.

3.1 A ORGANIZAÇÃO DE SOFTWARE INDUSTRIALIZADA – JAPÃO

Como uma abordagem de fábrica de software japonesa, foi escolhido o conceito de fábrica de software da empresa Toshiba, (AAEN, 1997).

Contexto: a fábrica de software produz primordialmente sistemas de controle, de reatores nucleares, de turbinas etc. O estabelecimento da fábrica de software, em 1981, foi motivado pelo desejo por software de alta qualidade no sentido de minimizar o número de defeitos no software. O foco na qualidade do software é complementado pelo foco na produtividade, para assegurar que os esforços visando à qualidade não diminuam a competitividade devido ao aumento de custos.

Objetivo: aumentar a qualidade de software e melhorar a produtividade. Posteriormente, os objetivos de qualidade e produtividade foram mantidos juntamente com o objetivo adicional de criar um ambiente onde projeto (*design*), programação, teste, instalação e manutenção pudessem ser realizados de uma maneira unificada.

Estratégia: a estratégia em 1981 incluía três elementos. O primeiro, para projetar construções que apóiam o desenvolvimento do processo de software, o segundo, para apoiar a construção do "*Software Work Bench*", o qual é um suporte integrado de software para as atividades realizadas no processo de desenvolvimento e o terceiro, para estabelecer uma organização que controla e monitora o processo de desenvolvimento de software. Posteriormente, em 1987, essas estratégias foram mantidas, porém, outras iniciativas foram adicionadas, tais como:

- Espaços de trabalho apropriadamente planejados.
- Ferramentas de software, interfaces com o usuário e facilidades para manutenção de ferramentas.
- Sistema de gerenciamento de uma linha base padronizada para revisão de projeto, inspeção e gerenciamento de configuração.
- Metodologia e disciplinas técnicas padronizadas.
- Programa de educação.
- Sistema de gerenciamento de progresso de projeto.
- Sistema de gerenciamento de custos.
- Sistema de gerenciamento de produtividade.
- Sistema de garantia de qualidade com métricas de qualidade padronizadas.
- Atividades periódicas de qualidade.
- Suporte à documentação.
- Biblioteca de software para apoiar a manutenção.
- Biblioteca técnica de dados.
- Sistema de desenvolvimento de carreira.

Organização: a organização da fábrica de software Toshiba é determinada por alguns dos elementos, a saber: banco de software, gerenciamento de projeto, reusabilidade, medição da produtividade, medição da qualidade e círculos de qualidade. O termo '*Software Work Bench'* é usado para denotar um sistema integrado de apoio aos trabalhadores na fábrica. Tal sistema consiste de um determinado número de subsistemas que, coletivamente, oferecem apoio a: programação, depuração, arquivos de projetos, geração e reuso de programa, teste de programa, especificação de requisitos, descrição e documentação do projeto de software, manutenção de software em operação nos *sites* dos clientes, gerenciamento de projeto, garantia da qualidade, controle de configuração e de reuso de software.

Seguindo o disposto em Aaen (1997), a fábrica de software possui um modelo em cascata padronizado para desenvolvimento de sistemas. A estratégia de gerenciamento de projeto usada é denominada '*look-forward-management*'. A idéia é calcular os custos a partir de equações contendo dados da história organizacional. Todos os projetos são decompostos em unidades de carga de trabalho. Uma unidade de carga de trabalho é definida como uma atividade a ser realizada por uma pessoa, para completar uma configuração de software. O progresso nos projetos é gerenciado com base nos últimos relatórios diários ou semanais nessas unidades de carga de trabalho. O uso de relatórios freqüentes de progresso possibilita rastrear desde o progresso atual até o progresso esperado, antes que itens estejam completos. Portanto, ações corretivas podem ser executadas durante o processo. Isto, em combinação com o uso de dados organizacionais para estimar o tempo esperado e recursos, forma a experiência para o termo '*look-forward-management*'.

Qualidade e produtividade são medidas na fábrica de software e existem fatores de qualidade definidos para cada linha base (*baseline*) do modelo de ciclo de vida. Qualidade de software é entendida em termos de confiabilidade e as medidas da qualidade expressam o número de defeitos esperados deixados no código após o teste e o tempo esperado entre as ocorrências de falhas. Essas medidas são estimadas a partir do número de defeitos encontrados durante a atividade de teste. A produtividade também é medida em todos os estágios no modelo de ciclo de vida.

A fábrica de software utiliza dois tipos de medidas: medidas baseadas em custos e medidas baseadas em capacidade. As medidas baseadas em custos referem-se ao custo de pessoas por mês, rendimento de pessoa/mês e custo equivalente às linhas de código produzidas por programador. As medidas de capacidade são usadas no gerenciamento do progresso, na determinação do trabalho e no planejamento de educação. Ao nível de projeto e de fábrica, o número de linhas equivalentes a cada montador (programador) por pessoa/mês, páginas de especificação por pessoas/mês e itens de teste por pessoas/mês são criados. Ao nível de pessoas, um espectro pessoal é criado o qual inclui produtividade e taxa de falha. Por exemplo, o número de páginas produzidas por um analista por hora é uma medida de produtividade de um analista. O número de falhas por página é a taxa de falha. Na fase de

programação, a medida de produtividade é o número de linhas de código produzidas, equivalente a um programador/hora.

O reuso de software, é considerado a questão mais crítica na melhoria da qualidade e produtividade, dado que a fonte principal de reuso é o software documentado e confiável. Para fomentar o reuso, uma organização para reuso foi criada. Os elementos chave são: o comitê geral de partes/peças de reuso de software e o centro de reuso de software. O comitê geral reúne, seleciona e autoriza demandas para criar, atualizar e descartar módulos reusáveis e o departamento de manufatura processa essa demanda. O comitê geral é o lugar onde todas as peças/partes reusáveis são guardadas, palavras-chave são determinadas para descrever as funcionalidades das partes/peças reusáveis e são oferecidos mecanismos de pesquisa.

Círculos de qualidade são grupos de voluntários, os quais se encontram para discutir e intercambiar idéias sobre melhoria da qualidade, da produtividade, melhoria da metodologia etc. Cada ano um ciclo de conferência é realizado para toda a organização e para toda a fábrica. Os melhores grupos são recompensados sendo convidados para essas conferências onde vários prêmios são oferecidos para distinguir as atividades de excelência.

As características dominantes do projeto (*design*) organizacional envolvem um determinado esforço para tornar a rotina de trabalho operacional, simples e repetitiva e para padronizar os processos de trabalho. Responsabilidades e rotinas são padronizadas e há uma hierarquia claramente definida de autoridade combinada com uma estrutura administrativa elaborada. Tudo, em todo o projeto (*design*) organizacional corresponde ao que é denominado de Burocracia de Máquina.

Implementação de melhoria: embora esta abordagem coloque uma pesada ênfase nas infra-estruturas (construção e ferramentas), os esforços de implementação englobam todos os nove elementos envolvidos na implementação de fábricas de software, identificados em (CUSUMANO, 1991 apud AAEN, 1997), em seu estudo sobre a iniciativa japonesa, a saber:

- Compromisso com a melhoria de processo.
- Foco no produto-processo e segmentação.

- Análise e controle de qualidade do processo.
- Processo moldado e centralizado.
- Padronização de habilidades.
- Padronização dinâmica.
- Reusabilidade sistemática.
- Integração de ferramentas auxiliadas por computador.
- Melhoria incremental de produto.

Em resumo, esta abordagem de fábrica de software: prima pelo aumento da produtividade e da qualidade de desenvolvimento e manutenção; possui uma estratégia baseada em infra-estruturas; faz uso de métricas; utiliza uma metodologia padronizada para todos os projetos; faz uso de reuso em todas as fases e tem como foco a tecnologia.

3.2 A FÁBRICA DE SOFTWARE GENÉRICA – EUROPA

A segunda idéia a ser considerada é a abordagem européia para fábrica de software genérica, a qual foi consolidada sob o programa Eureka e é denominada Fábrica de Software Eureka de acordo com Ferneström (1992).

Contexto: os participantes do projeto são grandes companhias européias, desenvolvedores de computadores, *software houses*, institutos de pesquisa e universidades. O projeto foi estabelecido com um período de dez anos (1987-1996), com dois mil e quatrocentos homens por ano de trabalho, financiado pela indústria (50%) e pelo governo nacional (50%). A maioria dos trabalhos desenvolvidos foi executada em subprojetos.

Neste projeto, o objetivo é fornecer tecnologia, padrões, suporte organizacional e outras infra-estruturas necessárias a fim de que tais fábricas sejam construídas e moldadas a partir de componentes comercializados por fornecedores independentes. O conceito de fábrica de

software denota uma combinação de ferramentas de software e processos de software a serem estabelecidos em uma organização. Uma fábrica consiste, dessa maneira, de partes computadorizadas e não computadorizadas.

O foco principal no projeto está no ambiente de suporte à fábrica, isto é, na parte de suporte computadorizado de uma fábrica de software. O projeto define uma arquitetura CASE[28] centrada em comunicação para ser combinada com suporte específico para descrever e representar atividades de engenharia de software. A idéia é que juntos possam auxiliar os construtores de fábrica de software em seus esforços para integrar produtos CASE aos modelos de processo de software e, nesse sentido, a Fábrica de Software Eureka, pode ser vista como uma fábrica de software genérica.

Objetivo: o objetivo é produzir uma arquitetura e um *framework* para ambientes de desenvolvimento integrado de software – blocos básicos de construção, componentes gerais e ambientes para diferentes áreas de aplicação (como aplicações comerciais, aplicações de tempo real, sistemas de telecomunicações e sistemas embutidos). Baseado nisso, poder-se-ia 'compor' ambientes moldados de apoio, com tarefas a serem realizadas pelos usuários em projetos específicos. O princípio básico é adaptar a organização - o ambiente de apoio - à fábrica.

Estratégia: a fábrica de software genérica desenvolve componentes e ambientes de produção que são partes de fábricas junto com padrões e diretrizes para componentes de software. Os padrões de componentes asseguram que os componentes diferentes de um ambiente de desenvolvimento integrado de software podem comunicar-se por meio de um condutor de software comum, o qual lida com comunicação, conversão de dados, configuração e outros serviços. O condutor torna possível, a seleção e a combinação de componentes que se adaptam às necessidades específicas de uma organização ou projeto, assim, a composição de um ambiente de usuário consiste de selecionar, combinar e montar um determinado número de componentes.

[28] Do inglês: *Computer-Aided Software Engineering*.

Uma fábrica de software é apresentada como mostra a Figura 21. O nível Genérico define a arquitetura de referências para fábricas de software com padrões para componentes de acordo com a Fábrica de Software Eureka. O nível Base de Componente denota a base de componentes disponíveis para construir ambientes de apoio à fábrica e, o número de componentes, de acordo com o padrão da Fábrica de Software Genérica, é continuamente expandido. O nível Modelo de Fábrica descreve as fábricas de software específicas em termos de modelos de processos a serem apoiados pela fábrica e as características dos componentes integrados no ambiente de suporte. Por fim, o nível Fábrica de Software denota a instância customizada da fábrica de software, colocada em seu devido lugar dentro da organização de desenvolvimento de software.

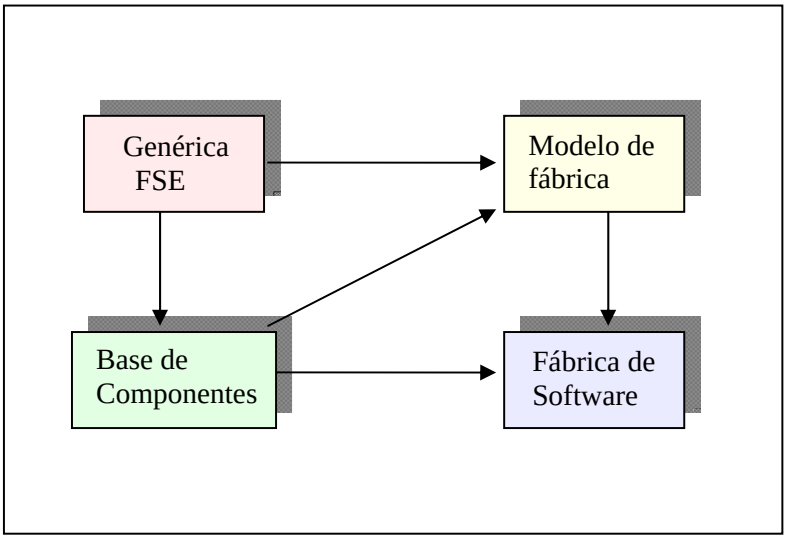


Figura 21. Produzindo uma instância de uma fábrica de software (AAEN, 1997)

Organização: A modelagem de processo representa um papel importante na definição dos requisitos para o ambiente de suporte à fábrica. Os usuários são alocados em contextos de trabalho que formam partes dos modelos explícitos do processo de produção de software e, tais contextos de trabalho existem para apoiar o trabalho dos indivíduos, para aumentar a previsibilidade do processo e para fazer a ligação entre as partes de tarefas computadorizadas e não computadorizadas.

Os processos de software descrevem o suporte aos procedimentos de trabalho na fábrica e, parte dessa descrição trata da adequação do ambiente de suporte à fábrica, para suprir a organização e os projetos. Os modelos de processo cobrem o relacionamento entre funções, tarefas, atividades e ferramentas e, as descrições de tarefas servem como *templates* para tarefas específicas a serem executadas durante o projeto.

As características importantes dominantes do projeto (*design*) organizacional envolvem: um determinado esforço para formalizar a operacionalização do trabalho, para relacionar trabalho automatizado e não automatizado, em processos coerentes e, com isso, padronizar os processos de trabalho. O projeto (*design*) organizacional consiste de um projeto de processo elaborado, focando tarefas e ferramentas e o objetivo é embutir o processo de software nas ferramentas. Assim como a Fábrica Industrializada do Japão, seu projeto organizacional corresponde à Burocracia de Máquina de Mintzberg, (MINTZBERG, 1983 apud AAEN, 1997). Contudo, essas duas abordagens diferem significativamente quanto à estratégia: a fábrica de software industrializada busca impor a padronização, principalmente via procedimentos de trabalho, enquanto a fábrica de software genérica utiliza ferramentas automatizadas.

Implementação de melhoria: o gerenciamento do esforço de melhoria não é explicitamente descrito na Fábrica de Software Eureka. Pelo contrário, a adoção do ambiente de suporte à fábrica a partir da fábrica de software genérica é vista como um processo anatômico. A ênfase principal está em estabelecer infra-estrutura tecnológica e, dos nove elementos de implementação descritos anteriormente, (vide item 3.1) somente uns poucos representam papel de destaque nesta abordagem: foco no processo-produto e segmentação, reusabilidade sistemática, ferramentas auxiliadas por computador e integração.

Resumindo, a abordagem da fábrica de software européia, possui: ambientes integrados de desenvolvimento orientados ao cliente; uma estratégia orientada a ferramentas, enfatizando a padronização de componentes e a adaptação de processo; uma metodologia adaptada por projeto; foco no processo-produto; não utiliza métricas e pratica o reuso sistematicamente.

3.3 A FÁBRICA DE COMPONENTES BASEADA EM EXPERIÊNCIA - ESTADOS UNIDOS

A terceira idéia a ser considerada, é a abordagem norte americana para fábrica de componentes baseada em experiência conforme disposto em Aaen (1997).

Contexto: a fábrica de componentes baseada em experiência foi desenvolvida no Laboratório de Engenharia de Software, o qual existe desde 1976, como um consórcio entre a NASA/*Goddard Space Flight Center*, *University of Maryland* e *Computer Science Corporation* (AAEN, 1997) e seus objetivos são:

- Entender o processo de software em um ambiente de produção.
- Determinar o impacto das tecnologias disponíveis.
- Instilar os métodos identificados/refinados dentro do processo de desenvolvimento.

A abordagem tem sido usada para experimentar novas tecnologias em um ambiente de produção, para extrair e aplicar experiências e dados a partir de experimentos e, para medir o impacto, com respeito a custo, confiabilidade, qualidade etc.

Objetivo: presume-se que mudanças significativas são necessárias no modo como o software é produzido. As organizações de software necessitam incrementar a qualidade e a produtividade e uma solução é resumida em três objetivos: melhorar a efetividade do processo, reduzir a quantidade de re-trabalho e reusar produtos do ciclo de vida (AAEN, 1997).

Estratégia: a estratégia - melhorias contínuas baseadas no reuso de experiências anteriores e automação flexível – consiste de três elementos chave: um paradigma de melhoria, uma organização dedicada à experiência e uma abordagem de contingência.

Um paradigma de melhoria (planejar, executar, analisar, sintetizar) é aplicado para integrar esforços no nível projeto e organizacional (AAEN, 1997): projetos são planejados explicitando objetivos e utilizando pacotes de experiências a partir de projetos anteriores. Projetos são executados e controlados por medição, conseqüentemente, os resultados são analisados e comparados com os objetivos planejados e, por fim, experiências são sintetizadas e "empacotadas" na forma de novos modelos ou modelos atualizados para serem utilizados nos projetos futuros.

Uma organização dedicada à experiência é concebida devido ao fato de que o reuso de experiência requer recursos separados para criar e manter objetivos reusáveis. As atividades são divididas em duas organizações – lógica e física – separadas: a organização do projeto cujo foco é a distribuição de software apoiado por pacotes de experiências reusáveis e uma fábrica de experiência cujo foco é o suporte a projetos fornecendo experiências reusáveis (AAEN, 1997).

Uma abordagem contingente é considerada devido a que 'todos os ambientes têm suas características e buscam seus objetivos por meio de distintos meios, diferentes de qualquer um dos outros' e porque 'a organização deve ser capaz de mudar sua configuração', Aaen (1997). A arquitetura de uma fábrica de componentes baseada em experiência é, portanto, descrita em diferentes níveis de abstração:

- O nível referência (representando agentes com funções definidas).
- O nível conceitual (representando fluxo de dados e de controle entre agentes).
- O nível implementação (definindo a técnica atual e a implementação organizacional).

A arquitetura de referência é usada para caracterizar as iniciativas estabelecidas e para descrever possíveis abordagens alternativas, isto é, uma arquitetura *clustered*, na qual todo desenvolvimento acontece na organização do projeto, versus uma arquitetura separada, na qual não há desenvolvimento, ocorrendo somente projeto (*design*) e integração na organização do projeto.

Organização: a organização da fábrica de componentes baseada em experiência é ilustrada na Figura 22. A organização do projeto é responsável principalmente pelas atividades de planejamento e desenvolvimento; o foco está em solucionar o problema. A fábrica de experiência é responsável principalmente pelas atividades de aprender a transferir tecnologias; o foco está em entender soluções e empacotar experiência para reuso. 'Esta organização reconhece o fato de que a melhoria do processo de software e do produto requer a acumulação contínua de experiências avaliadas (aprendizado), de uma forma que possa ser efetivamente compreendida e modificada (modelos de experiência), armazenada em um repositório de modelos integrados de experiência (base de experiência), que possa ser acessado/modificado para satisfazer as necessidades do projeto corrente (reuso)' segundo Aaen (1997).

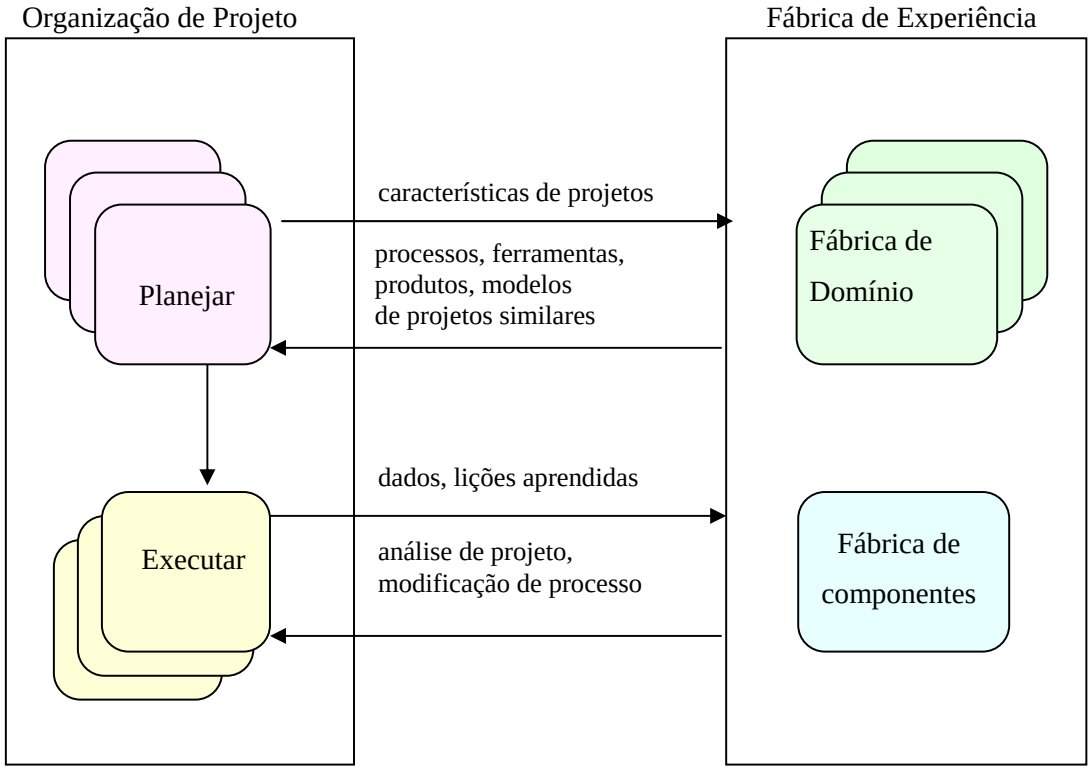


Figura 22. A organização de uma fábrica de componentes baseada em experiência, (AAEN, 1997)

A fábrica de componentes é dividida em várias sub-organizações, cada uma, dedicada a um tipo particular de experiência. A primeira divisão resulta em um número de fábricas de domínio, cujo propósito é identificar, coletar, organizar e fornecer experiências relacionadas a um domínio de aplicação específico (sistemas de controle de produção ou sistemas financeiros). A subdivisão seguinte da fábrica de experiência é a fábrica de componentes, cujo propósito é desenvolver e agrupar componentes de software reusáveis. Um componente de software é qualquer produto do ciclo de vida de software, tais como: componentes de código, projetos (*designs*), coleções de componentes de código e projeto e documentos em geral. Um componente de software reutilizável é uma coleção composta de componentes de software empacotados com tudo o que é necessário para reusá-lo e mantê-lo. Isso inclui o código, sua especificação funcional, seu contexto, um conjunto completo de casos de teste, uma classificação de acordo com uma determinada taxonomia e um manual para o reutilizador, segundo Aaen (1997).

As características dominantes do projeto organizacional são: unidades organizadas para projeto são responsáveis pelo planejamento e desenvolvimento e, recebem suporte de unidades especialistas. O foco está no aprendizado, treinamento e transferência de tecnologia, ou seja, o foco está nas habilidades e experiências coletivas ao invés de nos procedimentos, como o meio preferido para coordenar o trabalho. Tudo, em todo esse projeto organizacional corresponde ao que se denomina de Burocracia Profissional (AAEN, 1997).

Implementação de melhoria: o uso prático do conceito de fábrica de componentes baseada em experiência requer uma abordagem de gerenciamento incremental e, o ponto de partida é a operação presente na organização de software. Uma única fábrica de experiência é projetada e implementada como uma instanciação da arquitetura de referência, refletindo as características específicas da organização em questão. A fábrica é usada para coletar dados sobre pontos fortes e fracos, para determinar linhas base para melhorias, para estabelecer experimentos com novas técnicas e métodos e, para coletar experiências a serem reusadas em novos projetos. Desse modo, a organização começa a entender o relacionamento entre certas características do processo e as qualidades do produto. À medida que os processos de software são alterados, novas linhas base podem ser estabelecidas, identificando novas possíveis melhorias (AAEN, 1997). Esta abordagem coloca uma ênfase vigorosa na melhoria

contínua e indica uma classe ampla de esforços de implementação a fim de facilitá-la. Embora todos os nove itens de implementação (vide item 3.1) possam ser identificados, o foco principal está no processo de software.

Esta abordagem baseia-se em componentes; prima por maior eficiência dos processos, por menos re-trabalho e por mais reuso; possui uma estratégia de melhoria contínua, baseada na experiência adquirida, adota uma metodologia adaptada por projeto; seu foco está na tecnologia; faz uso da atividade de reuso e não utiliza métricas.

3.4 A ORGANIZAÇÃO MADURA DE SOFTWARE – ESTADOS UNIDOS

A quarta abordagem apresentada por Aaen (1997) é um conceito similar ao de fábrica de software, a organização madura de software, definida pelo CMM.

Contexto: o início do CMM foi determinado pelas necessidades do departamento de defesa dos USA (Estados Unidos da América) de avaliar contratadores de software. Para se realizar tal avaliação foi desenvolvido um questionário e, baseado nas experiências, no uso desses questionários, no desenvolvimento e no *feedback* da indústria, o SEI desenvolveu o modelo CMM, o qual pode ser visualizado na Figura 23.

Objetivo: baseado nas experiências iniciais, provenientes dos contratadores, o objetivo é criar um *framework* para melhoria do processo de software para obter um processo de desenvolvimento de software previsível, confiável e auto-melhorável, que produza software de alta qualidade (PAULK, 1993). Previsível significa que estimativas de custo e compromissos com cronograma podem ser cumpridos; confiável significa que a capacidade do processo é conhecida e; auto-melhorável significa que há um foco constante na melhoria do processo e que conhecimento e habilidades para melhoria são estabelecidos. Tudo isso,

deve ser alcançado pela atenção dada aos processos (melhoria) e não aos métodos ou ferramentas.

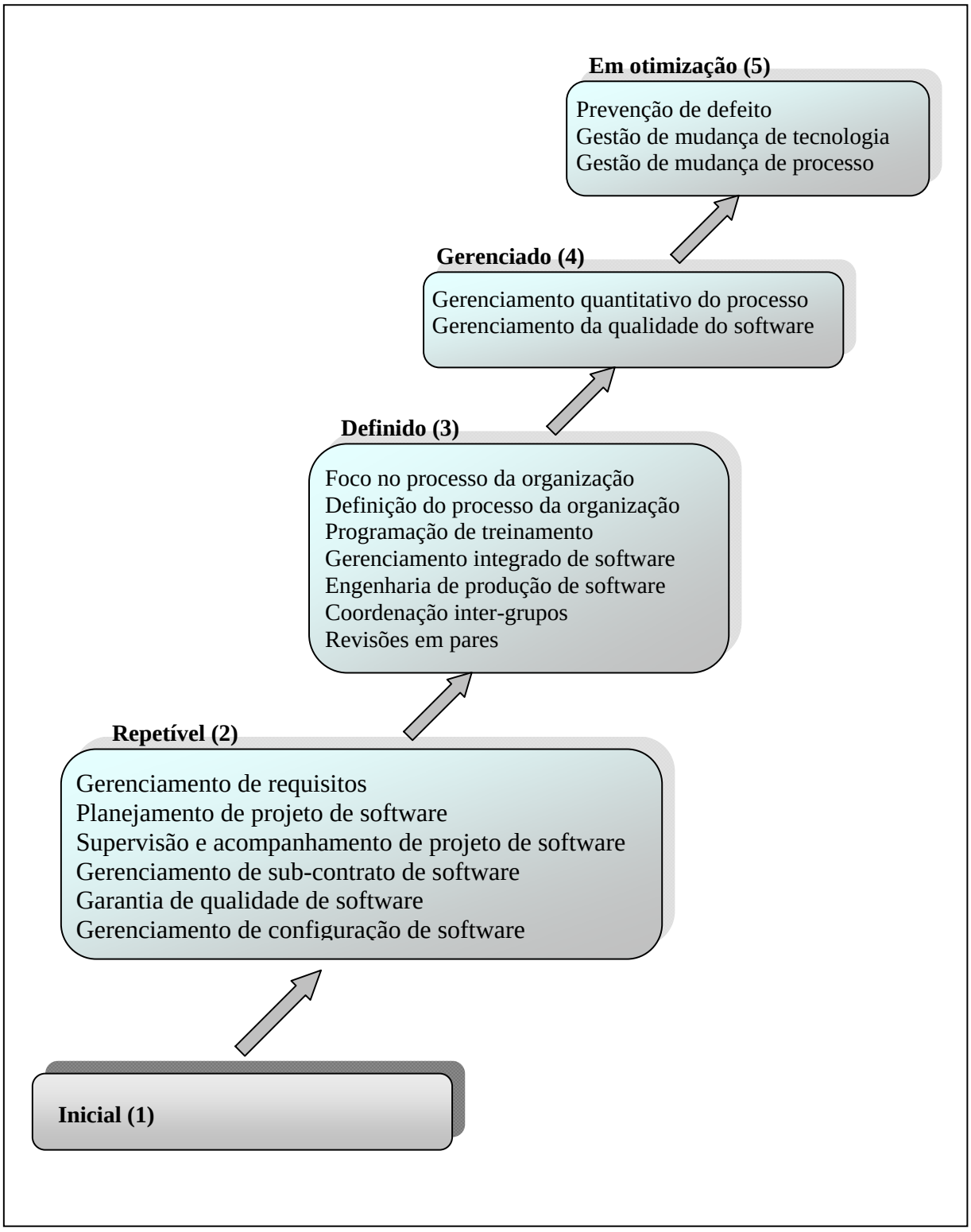


Figura 23. As áreas chave de processo no CMM (PAULK, 1993)

Organização: a organização ideal, tal como descrita pelo CMM, tem um conjunto de características e, nessa organização, o software é desenvolvido de uma maneira disciplinada na qual o planejamento e o acompanhamento dos projetos de software são estáveis e os sucessos anteriores são repetidos. Ao nível de projeto, os processos de projeto estão sob o controle efetivo do sistema de gerenciamento de projeto, seguindo planos realistas baseados no desempenho de projetos anteriores. O processo de software é descrito e consistente e, tanto a engenharia de software como o gerenciamento de atividades são estáveis e repetíveis. Cronograma e funcionalidade estão sob controle e a qualidade do software é acompanhada.

O processo de software e a qualidade do produto de software são previsíveis, dado que são medidos e mantidos dentro de limites mensuráveis. Tendências na qualidade do processo e do produto podem ser previstas dentro das fronteiras quantitativas desses limites e, quando os limites são excedidos, medidas são tomadas (AAEN, 1997).

Ë evidente a similaridade entre o Modelo CMM e esta abordagem de fábrica de software. Essa constatação aliada ao fato de que o modelo CMMI foi desenvolvido exatamente para integrar os diversos modelos CMM, a aproxima sobremaneira do modelo resultante dessa integração.

No nível organizacional, a capacidade de processo está baseada em uma compreensão ampla de todas as atividades da organização, funções e responsabilidades em um processo definido de software. A organização, como um todo, está focada na melhoria contínua do processo e tem meios para identificar pontos fortes e fracos do processo, com o objetivo de prevenir a ocorrência de defeitos. Dados sobre a efetividade do processo são usados para realizar análise de custos e benefícios de novas tecnologias e alterações propostas para o processo de software da organização e, as inovações que exploram as melhores práticas de engenharia de software são identificadas e transferidas entre a organização.

Não importa o nível de maturidade em que está a organização, a melhoria contínua é um conceito chave na fábrica de software CMM. Quando a organização se compromete a usar o CMM e todos os níveis na organização foram informados sobre a estratégia escolhida, deve

ser conduzida uma avaliação da organização a fim de descobrir quais processos são os mais relevantes para melhorar, essa avaliação, envolve a utilização de questionários e entrevistas, conduzidos em colaboração entre consultores e uma equipe interna de avaliação, de acordo com Aaen (1997). O resultado da avaliação é um número de recomendações, sobre o que melhorar e, baseado nessas recomendações, a gerência decidirá quais áreas tratar.

Para encontrar meios de aprimorar e implementar as melhorias, a maioria dos esforços de melhoria do CMM é organizada em grupos de processos de engenharia de software. Esses grupos gerenciam o esforço de melhoria, acompanham as atividades de melhoria, apóiam e facilitam o trabalho dos grupos “ad-hoc” nas áreas chave de processo. Os grupos “ad-hoc” se compõem de engenheiros de software e outros envolvidos nos projetos de desenvolvimento de software e, cada grupo trabalha na implementação de melhorias em uma única área chave de processo. Um grupo de processo de engenharia de software é freqüentemente constituído por membros do projeto de software que trabalham no grupo por um período de dois anos e depois retornam ao trabalho nos projetos de desenvolvimento de software (AAEN, 1997).

Quando a organização acredita que as áreas de processo problemas foram melhoradas, uma nova avaliação é realizada com o intuito de descobrir novas áreas carentes de melhoria (PAULK, 1993).

As características dominantes do projeto organizacional têm muito em comum com as da fábrica de componentes baseada em experiência: unidades organizadas por projeto são responsáveis pelo planejamento e desenvolvimento e essas unidades recebem apoio de unidades especializadas. As unidades de apoio têm a responsabilidade principal pelo treinamento, pelo gerenciamento de alterações, pela garantia da qualidade e pela definição de um processo padrão de software na organização. Posteriormente, as unidades de apio são responsáveis pelo aprendizado de experiências dos projetos. Os projetos têm a responsabilidade principal pelo desenvolvimento, gerenciamento, coordenação, revisões e prevenção de defeitos e essa responsabilidade se amplia para adotar o processo padrão de software para o projeto. Assim como a fábrica de componentes baseada em experiência, o foco aqui é nas habilidades e experiências coletivas ao invés de nos procedimentos como um meio para coordenar o trabalho e, o projeto organizacional corresponde à Burocracia

Profissional. Contudo, essas duas abordagens diferem com respeito à definição de unidades especializadas: a fábrica de componentes baseada em experiência atribui responsabilidades detalhadas para unidades especializadas, enquanto que a organização madura de software identifica um grande número de objetivos e compromissos com funções organizacionais ao invés de especificar unidades especializadas (AAEN, 1997).

Implementação de melhoria: Os elementos chave no gerenciamento do processo de melhoria são melhorias contínuas, compromisso da alta gerência, identificar resistências na organização e permitir que pessoas próximas ao processo em questão melhorem aqueles processos. Nesta abordagem, há uma forte ênfase na melhoria passo a passo. Os esforços de implementação abrangem um amplo espectro e todos os elementos de implementação (vide item 3.1) podem ser identificados. Tal como na fábrica de componentes baseada em experiência, o foco principal desta abordagem está no processo de software.

Para finalizar, pode-se afirmar que esta abordagem de fábrica: possui processo eficiente, previsível, confiável e auto-melhorável; metodologia de melhoria passo a passo; metodologia adaptada por projeto; utiliza-se de métricas; faz pouco uso de reuso e está focada no processo de software.

3.5 A FÁBRICA DE SOFTWARE DA MICROSOFT

As abordagens de software apresentadas anteriormente, retratam distintos períodos e as experiências e vivências mais bem sucedidas da época em que foram concebidas. A fábrica de software japonesa foi concebida no início da década de oitenta; a fábrica de software européia foi implantada no início da década de noventa; a fábrica de componentes baseada em experiência resulta de trabalhos iniciados no final da década de oitenta e início da década de noventa e a organização madura de software, foi desenvolvida no início da década de noventa.

Cada uma dessas abordagens reflete a preocupação dos meios acadêmicos e da indústria em solucionar problemas existentes à época e resultam das tentativas de acompanhar as mudanças da área e de melhorar a maneira de se construir software.

Os diversos problemas que envolvem atualmente o desenvolvimento de software suscitam iniciativas visando oferecer às organizações um meio para que as mesmas possam sanar os problemas existentes e, ao mesmo tempo, aumentar o seu nível de qualidade e de produtividade. A fábrica de software proposta pela Microsoft possui características distintas e inovadoras em relação às abordagens apresentadas anteriormente. Contudo, tem em comum com as demais, o propósito de auxiliar as organizações na solução de seus problemas e na melhoria da qualidade e produtividade.

A fábrica de software proposta pela Microsoft está assentada sobre três pilares básicos:

- Desenvolvimento baseado em componentes.
- Desenvolvimento dirigido à modelo.
- Linha de produto de software.

A confluência desses três pilares forma uma abordagem para desenvolvimento de aplicação baseada no conceito de fábrica de software. Essa abordagem promete maiores ganhos em termos de produtividade e previsibilidade que aqueles produzidos por melhoria incremental pelo paradigma de orientação a objeto e também promete formar um conjunto de aplicações mais efetivo quanto a custo, por meio de reutilização sistemática, possibilitando a formação de redes de fornecimento e abrindo as portas para a customização em massa, (GREENFIELD e SHORT, 2003).

A proposta de fábrica de software da Microsoft enfatiza a necessidade de uma mudança na maneira de construir software. Tal proposta considera que a indústria de software ainda se mantém confiante na arte das habilidades dos desenvolvedores especializados no trabalho intensivo de tarefas manuais, mas que, a crescente pressão para reduzir custos e prazos de entrega e para melhorar a qualidade de software pode catalisar a transição para métodos mais automatizados.

Conforme explicitado em Greenfield (2004), o desenvolvimento de software jamais será reduzido a um processo puramente mecânico. Ao contrário, a chave para atender a demanda global é deixar de desperdiçar o tempo dos desenvolvedores competentes com tarefas rotineiras e maçantes e encontrar meios de fazer melhor uso dos preciosos recursos ao invés de gastá-los na construção manual de produtos finais que irão requerer manutenção e, até mesmo, substituição em poucos meses ou anos, quando surgir uma plataforma mais importante ou quando mudanças nas condições de mercado, exigirem alterações nos requisitos do negócio. Um meio de fazer isso é dar aos desenvolvedores, meios para encapsular seu conhecimento como *assets* reusáveis que outros possam utilizar. Padrões (*patterns*) já demonstraram ser efetivos, porém, limitados para reuso de conhecimento. O próximo passo é da documentação para a automatização, usando linguagens, *frameworks* e ferramentas para automatizar a aplicação de padrões (*patterns*), (GREENFIEL, 2004).

Para, Fuller (2006), as fábricas de software também tentam considerar o fato de que não existem desenvolvedores à disposição, assim, todos ganham mantendo o conhecimento dos especialistas dentro da fábrica. A organização ganha, por ter especialistas trabalhando nas tarefas mais importantes ao invés de ensinando as equipes de desenvolvimento. As equipes de desenvolvimento não precisam mais ter que fazer muitas tarefas rotineiras e desestimulantes, estando, portanto, liberadas para realizarem tarefas mais interessantes e desafiadoras.

De acordo com Vries (2006), é possível construir software dentro dos prazos e custos estimados e com qualidade. Contudo, existe pouca consciência organizacional sobre a ineficiência da abordagem atual para construir software. Sem a percepção dos problemas existentes não haverá uma direção para melhoria. Para começar a construir sistemas de software previsíveis, é necessário realizar uma mudança cultural. É necessário possibilitar aos profissionais, o conhecimento sobre o que fazer, quando, por que e como fazê-lo, além de automatizar mais os aspectos tediosos e rotineiros do seu trabalho. Trata-se da industrialização do desenvolvimento de software, por meio da aplicação de técnicas devidamente comprovadas em outras indústrias, na indústria de software, com o intuito de tornar as coisas melhores para os clientes e para as próprias organizações.

A complexidade, envolvendo projeto e soluções, tem crescido drasticamente ao longo dos últimos 30 anos. Do mesmo modo que o catálogo de aplicações evolui, surgem estratégias de processo que podem ajudar uma organização a superar os problemas que preocupam o desenvolvimento de software atualmente. Promover a sensibilidade por meio da adoção de metodologias de linha de produção assegurará maior sucesso dos sistemas liberados usando esses processos. Reuso raramente ocorre por acidente e, usando processos estratégicos tal como desenvolvimento dirigido à modelo, a estrutura levará à descoberta de componentes reusáveis como um estágio intencional em oposição à fase oportunista (FULLER, 2006).

A base para o entendimento desse aspecto – a industrialização do software - envolve uma comparação entre a produção de bens físicos e o desenvolvimento de software e, a chave para clarificar essa questão é entender as diferenças entre produção e desenvolvimento e entre economia de escala e economia de escopo.

A abordagem de fábrica proposta pela Microsoft se reporta aos desafios similares pelos quais passaram outras indústrias, quando decidiram sair do processo artesanal e migrar para a industrialização de seus produtos. Essa mudança fez com que essas indústrias aprendessem a adaptar e a juntar componentes padrões para produzir produtos similares, porém distintos, padronizando, integrando e automatizando seus processos de produção, desenvolvendo ferramentas extensíveis que podem ser configuradas para automatizar tarefas repetitivas, desenvolvendo linhas de produto para obter economia de escala e de escopo e formando uma rede de fornecimento para distribuir custo e risco entre uma rede de fornecedores altamente especializados e independentes (GREENFIELD, 2004). Essas mudanças possibilitam uma produção efetiva quanto a custo de uma grande variedade de produtos para satisfazer uma ampla faixa da demanda dos clientes.

Economia de escopo pode ser obtida no contexto de uma família de produtos, cujos elementos variam, embora compartilhem características comuns. Uma família de produtos pode conter tanto produtos finais, como aplicações de gerenciamento de portifólio ou componentes, tais como *framework* para gerenciamento de contas usado para gerenciamento de portifólio e aplicações para gerenciar o relacionamento com o cliente (GREENFIELD e SHORT, 2003).

Uma família de produtos fornece um contexto, no qual muitos problemas comuns aos elementos da família podem ser relacionados coletivamente. Ao construir com base nos conceitos de linha de produto de software, as fábricas de software utilizam (exploram) esse contexto para prover amplas soluções para uma família, enquanto gerenciam variações entre os elementos da família.

Uma fábrica de software captura conhecimento sistematicamente sobre como produzir os elementos de uma família específica de produtos, disponibiliza-o na forma de *assets* tais como, padrões (*patterns*), *frameworks*, modelos e ferramentas e, sistematicamente utiliza esses *assets* no desenvolvimento dos elementos da família, reduzindo custos e prazos para entrega no mercado e melhorando a qualidade do produto durante o desenvolvimento (GREENFIELD e SHORT, 2003).

De acordo com o disposto em Greenfield (2004), o reuso sistemático pode gerar economia de escala e de escopo, dois aspectos econômicos, bem conhecidos em outras indústrias. Embora ambos reduzam tempo e custo e melhoram a qualidade do produto, produzindo produtos coletivamente - ao invés de individualmente - diferem no modo como produzem esses benefícios. A economia de escala aparece quando múltiplas instâncias idênticas de um único projeto são produzidas coletivamente ao invés de individualmente e a economia de escopo aparece quando, múltiplos, porém, distintos projetos e protótipos são produzidos coletivamente ao invés de individualmente. São exemplos desses tipos de economia:

- Em mercados com clientes de *desktops*, onde cópias de produtos como sistemas operacionais e aplicações de produtividade são produzidos em massa, o software apresenta economia de escala.
- Em mercados como empresas, onde as aplicações de negócio, desenvolvidas para obter vantagens competitivas, se jamais, são produzidas em massa, o software apresenta economia de escopo.

Escopo continua sendo o elemento básico de interesse para todos os artefatos da fábrica de software. No caso de aplicações de produtividade de negócio, economia de escopo e de escala é obtida (FULLER, 2006).

A fim de obter retorno de investimento, componentes reusáveis devem ser reusados o suficiente para recuperar mais do que o custo de seu desenvolvimento, diretamente, pela redução de custos, ou indiretamente, pela redução de riscos, pela redução de tempo até a chegada do produto ao mercado (*time-to-market*) ou melhoria da qualidade. Componentes reusáveis são *assets* financeiros, dentro de uma perspectiva de investimento. Dado que o custo para construir um componente reusável geralmente é muito elevado, níveis proveitosos de reuso são provavelmente alcançados por acaso. Portanto, uma abordagem de reuso sistemático é exigida (GREENFIELD, 2004) e isso geralmente envolve:

- Identificar o domínio, no qual múltiplos sistemas serão desenvolvidos.
- Identificar problemas recorrentes em tal domínio.
- Desenvolver conjuntos de *assets* de produção integrados que solucionam esses problemas e então aplicá-los aos sistemas desenvolvidos naquele domínio.

A convergência das idéias chave de linha de produto, desenvolvimento baseado em componentes e desenvolvimento dirigido a modelos, faz dessa, uma abordagem de fábrica de software inovadora. Essa inovação se mantém na integração dessas idéias em um *framework* coeso para apoiar novas ferramentas e novas práticas. Combinando técnicas dirigidas a modelos e baseadas em componente com princípios de linha de produto, as fábricas de software introduzem um novo modelo de desenvolvimento de aplicações, onde ferramentas de desenvolvimento altamente extensíveis são rápida e facilmente configuradas para criar fábricas de software para domínios específicos. A obtenção dessa visão é o objetivo das fábricas de software. Isso irá requerer repensar ferramentas e métodos, linguagens e *frameworks* (GREENFIELD e SHORT, 2003).

Em uma outra abordagem (EADIE, 2006; FULLER, 2006), derivada da fábrica de software da Microsoft, a fábrica de software está assentada sobre quatro pilares, a saber:

- Linha de produto.
- *Frameworks* de arquitetura.
- Desenvolvimento dirigido à modelo.
- *Guidance in context* (Diretriz sensível ao contexto).

Esta abordagem também aposta na industrialização do software para solucionar os problemas relacionados a orçamentos, cronogramas, segurança, defeitos, falhas de projeto, desperdício de tempo com re-trabalho, dentre outros.

Para Eadie (2006), esses pilares combinados são mais que a soma de suas partes para impulsionar a industrialização de software. Os quatro pilares que compõem a fábrica de software são ilustrados na Figura 24 e descritos a seguir.

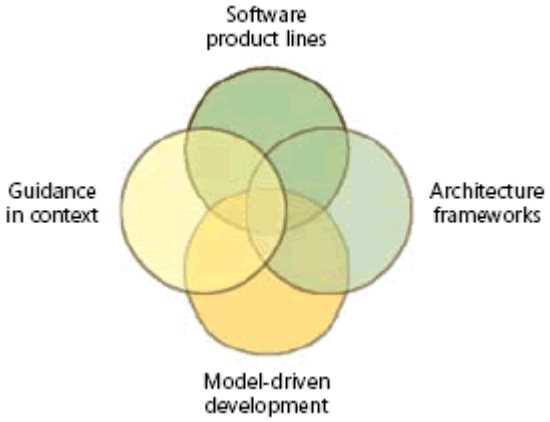


Figura 24. Os quatro pilares de uma fábrica de software (EADIE, 2006)

Linha de produto. Conjunto de sistemas de software compartilhando um conjunto gerenciado comum de características que satisfazem as necessidades específicas de um segmento particular de mercado ou missão que são desenvolvidos a partir de um conjunto de *assets* em um modo determinado.

Para Eadie (2006), linhas de produto amortizam o investimento em *assets* tais como:

- Requisitos e análise de requisitos.
- Modelagem de domínio.
- Arquitetura de software e projeto (*design*).
- Engenharia de desempenho.
- Planos, casos e dados de teste.
- Processos, métodos e ferramentas.

- Orçamento, cronograma e planos de trabalho.
- Componentes.

Existem três atividades associadas à linha de produto: desenvolvimento de *assets* essenciais e desenvolvimento de produto usando tais *assets* essenciais e, essas duas atividades são apoiadas pela terceira atividade que é o gerenciamento técnico e organizacional. Os *assets* essenciais podem ser *assets* extraídos dos *assets* existentes ou construídos desde o começo para a linha de produto. Os *assets* essenciais com processos associados para cada *asset*, junto com a produção de requisitos, um escopo de linha de produto e um plano de produção são criados para construir o produto (EADIE, 2006).

***Frameworks* de arquitetura**. Definem pontos de vista que satisfazem e separam os interesses chave dos *stakeholders*. Tais *frameworks* organizam ferramentas, processos e conteúdo, de acordo com tal ponto de vista. Relacionam e integram fases do ciclo de vida, componentes do sistema e níveis de abstração, como mostra a Figura 25. *Framework* de arquitetura também é descrito, com freqüência, como o esquema ou *framework* de processo para a fábrica de software. Este *framework* pode ser usado para guiar o *workflow* e as atividades para o projeto, até que as atividades estejam completas.

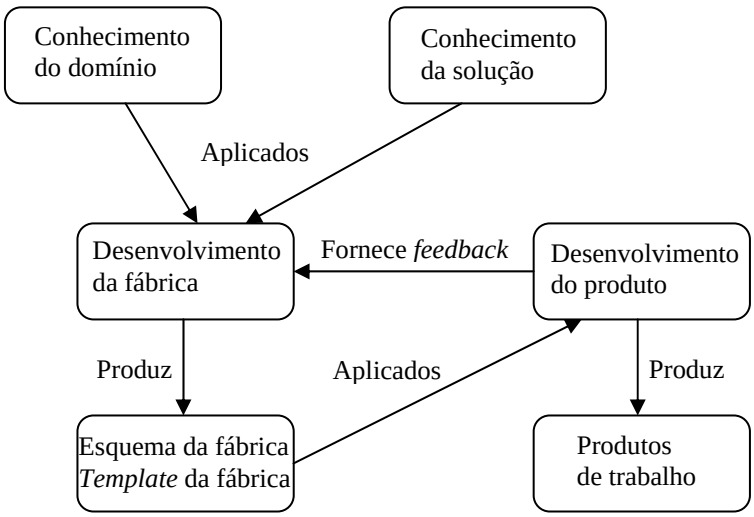


Figura 25. O ciclo de vida da fábrica (EADIE, 2006)

De acordo com Vries (2006), o esquema da fábrica de software fornece um mecanismo útil para métricas da organização. Dado que os pontos de vistas focalizam um aspecto específico do processo de desenvolvimento de software, é possível usar esses pontos de vista para definir medições focadas na produtividade e qualidade. Utilizando tais medições, é possível acumular dados relacionados a aspectos específicos do processo de desenvolvimento de software. Por meio da análise dos dados é possível determinar quais pontos de vista necessitam ser melhorados, como melhorá-los e o que se ganha com a melhoria dos mesmos.

Para implementar essa abordagem é necessário um meio para expressar o tamanho do produto, os gastos com orçamento e cronograma e a qualidade do produto para ser capaz de quantificar a previsibilidade, a produtividade e a qualidade para cada ponto de vista. Medindo cada ponto de vista, tanto quanto, o desempenho global da fábrica, é possível determinar como cada ponto de vista afeta o desempenho da fábrica como um todo e, conseqüentemente, quanto deve ser investido para melhor apoiar um dado ponto de vista (VRIES, 2006).

Produtividade é um dos aspectos do desenvolvimento de software que necessita ser melhorado. Entretanto, para quantificar produtividade, é preciso dispor de uma métrica que possa ser usada para expressar a produtividade em termos de volume de produto de software construído em um dado período de tempo. Quando aptos para prever o tamanho do sistema e para medir o crescimento do tamanho do produto durante o desenvolvimento, é possível prever melhor o tempo requerido para completar o projeto e para medir produtividade em termos de horas gastas por unidade de tamanho e produto (VRIES, 2006).

Medindo o crescimento e o tamanho, é possível identificar diferenças entre os valores atuais e os planejados e é possível começar a analisar e gerenciar as diferenças que se tornaram aparentes. Por outro lado, quantificando o tamanho dos produtos construídos, medindo o tempo gasto para construí-los e registrando o número de defeitos encontrados, é possível descrever o desempenho da fábrica (VRIES, 2006).

Neste ponto, podem surgir dúvidas sobre como é possível predizer tamanho e crescimento do produto com o cuidado suficiente para tornar esse tipo de análise e de medição úteis.

Certamente isso não é possível se o desenvolvimento de aplicações software é feito de modo arbitrário (VRIES, 2006). Contudo, caso se esteja usando uma fábrica de software, tem-se duas vantagens que melhoram significativamente a previsibilidade: **primeiro,** é o fato de se estar desenvolvendo um elemento de uma família específica de produtos com características conhecidas e não apenas uma aplicação arbitrária. Dado que uma fábrica possibilita descrever uma família de produto e suas características principais e, mais importante, refinar tal descrição com a experiência obtida a partir da execução de múltiplos projetos – sabe-se mais sobre uma aplicação em desenvolvimento usando uma fábrica do que sobre uma aplicação arbitrária. **Segundo**, é o fato de se desenvolver uma aplicação aplicando diretrizes comprovadas pela fábrica. Padronizando o modo de fazer algumas coisas, a fábrica tende a remover variações gratuitas do processo de desenvolvimento, fazendo com que o produto e o seu crescimento sigam – provavelmente – padrões similares de uma aplicação para a próxima (VRIES, 2006).

Desenvolvimento dirigido à modelo. É uma técnica que utiliza metadado capturado nos modelos, para automatizar as tarefas de desenvolvimento de software. Modelagem de dados tem sido considerada como uma das melhores maneiras para construir sistemas, sendo usada para capturar os propósitos do sistema. Contudo, modelos são, freqüentemente, descartados e têm sido vistos como elementos de segunda classe em comparação ao código. Nas fábricas de software, modelos são usados para expressar formalmente os artefatos de dado domínio e para capturar o propósito que ferramentas podem interpretar para prover formas mais avançadas de automação (EADIE, 2006).

Frameworks de arquitetura podem ser usados para criar linguagens feitas sob medida altamente focadas em problemas específicos, plataformas ou tarefas. Metadado que é capturado pelos modelos é usado para derivar automação ao invés de ser referenciado apenas uma vez num dado momento. Em fábricas de software o desenvolvimento dirigido à modelo é o instrumento usado pelo conjunto de instrumentos da linguagem específica a domínio. A linguagem específica a domínio permite criar um ambiente de modelagem que descreve cuidadosamente o produto que está sendo construído e também garante uma tradução mais acurada do modelo pelos membros das equipes, os quais traduzem o código gerado a partir do modelo.

De acordo com o explicitado em Eadie (2006), diretriz neste contexto é usada para descrever o que fazer e para fornecer auxílio para fazê-lo. A diretriz é particionada para casos de uso comuns e é anexada às fases do processo e às partes da arquitetura. Essa diretriz é sensível ao contexto e aparece somente quando necessário. Trabalhos feitos com ferramentas tais como a GAT (*Microsoft Guidance Automation Toolkit*) possibilitam pacotes instaláveis contendo conjuntos organizados de *assets* de diretrizes configuráveis para casos de uso comuns. A diretriz também pode ter pré e pós-condições para permitir que o *workflow* possa variar de acordo com o conteúdo das restrições.

Na abordagem proposta por Eadie (2006), fábricas de software podem ser de dois tipos: horizontal e vertical, conforme ilustra a Figura 26. Fábricas horizontais são aquelas que não focam uma indústria específica, mas são amplas e, em geral, podem ser absorvidas por muitas outras fábricas. Uma fábrica vertical, por outro lado, está focada em uma indústria ou domínio, como, por exemplo, finanças, governo ou saúde.

É possível capturar conhecimento da indústria neste ponto de vista também. Pode-se afirmar que cerca de 70-80 por cento das necessidades básicas de qualquer cliente em uma dada indústria são as mesmas e, que a diferença de 20-30 por cento leva à adaptação de soluções para atender as necessidades individuais do cliente. Provavelmente as organizações já usam *frameworks* ao nível de indústria, que capturam o que é comum entre vários negócios dentro de uma indústria específica e utilizam os pontos comuns dessa informação (EADIE, 2006).

Se as indústrias capturam o que é comum a elas usando seus processos de negócio e os componentes comuns da empresa, a variabilidade para um determinado cliente pode ser capturada extraindo suas regras de negócio. Prover partes fixas e ajustáveis requer ter as partes certas para permitir sistemas configuráveis. Exemplos dessas partes incluem: processos e serviços comuns, modelos de dados, componentes e objetos. Para ajustar-se ao cliente, um sistema deve também ser extensível e inter-operável com sistemas existentes.

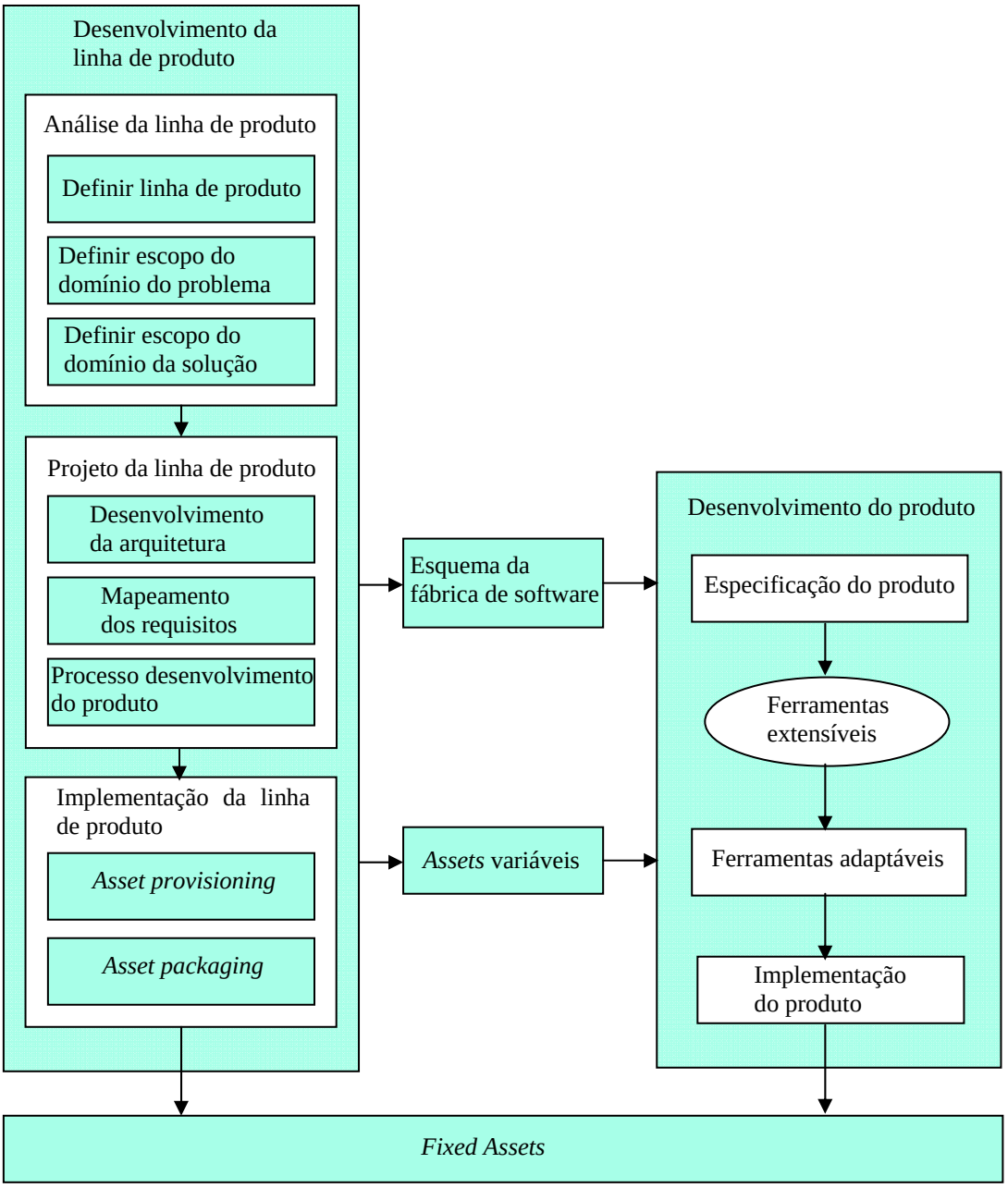


Figura 26. Uma fábrica de software (EADIE, 2006)

Segundo Eadie (2006), uma vez definidos a variabilidade e o que é comum, uma fábrica de software torna-se o mecanismo para dispô-los e executa-los com alto nível de produtividade. Esse mecanismo possibilita a um cliente rapidamente configurar, adaptar e juntar componentes desenvolvidos independentemente, independente de localização, para produzir futuros sistemas distintos, porém, similares. Fábricas de software poderiam possibilitar que arquitetos, desenvolvedores e testadores usem *assets* de um modo mais ordenado.

Para iniciar uma fábrica de software deve-se ter: processos comuns, serviços comuns, modelos de dados, componentes, objetos, padrões (*patterns*), modelos, planos de teste, *scripts* e dados. Uma fábrica de software bem sucedida pode ser dimensionada por uma estratégia de reuso de software por meio de uma abordagem de linha de produto bem gerenciada que alcança os objetivos do negócio listados previamente. Contudo, vale ressaltar como uma fábrica de software poderia apoiar esses objetivos (EADIE, 2006):

- Necessidade de soluções padrões simples. Uma fábrica de software apóia este objetivo criando a fábrica pelo menos uma vez qualificando muitos dos produtos do *template*.
- Aplicações são projetadas e implementadas rapidamente, portanto, podem chegar ao mercado rapidamente. Mais de 80 por cento do trabalho pode ser automatizado por uma fábrica de software.
- Liberar o trabalho correto e a experiência tecnológica com preço correto, no lugar e no tempo certos. O conhecimento do domínio e a experiência são mantidos na fábrica de software e os especialistas estão disponíveis o tempo todo durante o ciclo de vida de desenvolvimento.
- Aplicações/soluções de alta qualidade são feitas corretas na primeira vez. A fábrica de software é provada, testada e confirmada.
- Liberar o que o cliente ordenar. Uma fábrica de software atende as necessidades de negócios do cliente mais cedo, com menos risco e criando um produto sob medida em oposição ao desenvolvimento de um projeto adaptado.

Para Eadie (2006), fábricas de software possibilitam que projetos saltem do início para uma *baseline* bem definida. Pessoas sabem o que fazer e como fazer. Elas possuem *assets* provados e confiáveis para ajudá-las a fazer o que têm que fazer, rápida e corretamente. Esse alicerce torna as manutenções futuras mais fáceis, melhora a qualidade e a consistência e também torna mais fácil prever o impacto das mudanças. Novos contratados necessitam menos treinamento, cronogramas e orçamentos são mais precisos e a fábrica de software captura conhecimento para uso futuro.

Para Vries (2006), é possível auxiliar a organização ou a indústria como um todo, capturando o conhecimento obtido a partir de experiência e transferindo-a para outros projetos, usando fábricas de software.

Os artefatos mais valiosos que qualquer desenvolvedor pode produzir são aqueles que podem ser aplicados entre numerosos domínios de problemas. Essa versatilidade é devido ao fato de que *patterns*, *frameworks*, diretrizes, modelos de referência e ferramentas para automação são resultados essenciais de qualquer iteração de processo. Estratégias de resultado que focam uma arquitetura devem extrair e aplicar técnicas comprovadas para solucionar problemas em aplicações. Desenvolvedores são tipicamente “embutidos” no nível de projeto, mantendo, porém, a inteligibilidade do escopo da empresa. Ao longo do tempo, a coleção das melhores práticas extraídas, forma uma biblioteca para que outras equipes possam compor suas soluções (FULLER, 2006).

Conforme já explicitado (GREENFIELD, 2004), outras indústrias aumentaram sua capacidade de produção, mudando do “artesanato” - onde todos os produtos eram criados a partir do início, por uma pessoa ou pequenas equipes – para fábrica, onde uma ampla gama de produtos é rapidamente agrupada a partir de componentes reusáveis, criados por múltiplos fornecedores e onde as máquinas automatizam as tarefas rotineiras e desestimulantes. Essas indústrias padronizaram processos, projetos e os empacotaram, utilizando linhas de produto para facilitar reuso sistemático e criaram uma cadeia de fornecimento para distribuir custos e riscos. Algumas são capazes, atualmente, de customização em massa, onde variantes de produtos são produzidos econômica e rapidamente, dentro da demanda, para satisfazer os requisitos específicos de clientes em particular.

Como pode ser observado, as abordagens de fábrica de software propostas (GREENFIELD e SHORT, 2003; EADIE, 2006; FULLER, 2006, VRIES, 2006) oferecem uma visão de futuro, pois consideram que as fábricas de software serão amplamente utilizadas para produzir aplicações. Essa visão implica mudanças significativas, não somente nos métodos e ferramentas, mas também na economia do desenvolvimento de software. Além do mais, tais propostas consideram que, tal como ocorreu nas outras indústrias, o processo de construção de software também pode ser industrializado.

Para Greenfield (2004), as fábricas de software são apenas o próximo passo lógico na continuação da evolução dos métodos e práticas para desenvolvimento de software. Contudo, prometem mudar o caráter da indústria de software, introduzindo padrões (*patterns*) de industrialização.

Não obstante as diferenças existentes em todas as abordagens de fábrica de software apresentadas, todas, sem exceção, atendem, de uma maneira ou outra, os modelos de qualidade de processo e de produtos vistos anteriormente. Ou seja, todas as abordagens cobrem os aspectos de qualidade de produto e/ou de processo, contemplados nos diversos modelos apresentados. Duas das atividades para assegurar qualidade (inspeção e teste), contempladas nos modelos de qualidade apresentados e que fazem parte da proposta de metodologia apresentada neste trabalho, são descritas no próximo capítulo.

4. INSPEÇÕES E TESTE DE SOFTWARE

O Comitê Técnico da ISO/IEC 9126-1 (2001) já afirmava no início da década de 2000, que a indústria de software estava entrando em um período de maturidade ao mesmo tempo em que o software estava se tornando um componente crucial para muitos produtos. Afirmava também que esse aspecto pervasivo do software tornava-o o novo principal fator no comércio (negócios) e que futuramente devido às novas demandas globais com respeito à segurança e qualidade, a necessidade de compromissos internacionais sobre procedimentos para avaliação da qualidade de software tornar-se-ia muito importante.

Para Kan (2002), qualidade deve ser definida e medida se melhorias devem ser obtidas. O maior problema na engenharia e gerenciamento de qualidade é que o termo “qualidade” é ambíguo, de modo que é comumente mal entendido. A confusão pode ser atribuída a várias razões. Primeiro, qualidade não é uma idéia simples, mas um conceito multidimensional. A dimensão de qualidade inclui a entidade de interesse, o ponto de vista sobre a entidade e os atributos de qualidade da entidade. Segundo, para qualquer conceito existem níveis de abstração, quando pessoas falam sobre qualidade, uma parte pode referir-se ao seu significado específico. Terceiro, o termo “qualidade” faz parte da linguagem diária das pessoas e o uso popular e profissional do termo pode ser muito diferente.

Para melhorar a satisfação dos clientes, tanto como a satisfação com relação a vários atributos de qualidade, os atributos de qualidade devem ser considerados quando do planejamento e do projeto de software.

Na engenharia de software, mais especificamente na disciplina de engenharia de qualidade, a qualidade pode e deve ser operacionalmente definida, medida, monitorada, gerenciada e melhorada. Qualidade que não pode ser controlada e gerenciada, também não pode ser quantificada (KAN, 2002).

4.1 MÉTRICAS E MEDIÇÕES DE SOFTWARE

A melhoria da qualidade, do desempenho e da produtividade, é objetivo chave, para qualquer organização que desenvolve software e medição quantitativa – e métricas de software em particular – a qual pode ajudar na obtenção deste objetivo, uma vez que fornece um meio formal para estimar qualidade e complexidade de software (MENS, 2001).

Métricas de software referem-se a uma grande variedade de medições que são realizadas nos produtos de software.

Para (KAN, 2002), métricas de software podem ser classificadas em três categorias:

- Métricas de produto. Descrevem as características de produto tal como tamanho, complexidade, características de projeto (*design*), desempenho e nível de qualidade.
- Métricas de processo. Podem ser usadas para melhorar o desenvolvimento e manutenção de software.
- Métricas de projeto. Descrevem as características de projeto e execução.
- Métricas de qualidade de software são um subconjunto das métricas de software que enfatizam os aspectos de qualidade de produto, processo e projeto (*project*). Em geral, métricas de qualidade de software são mais estritamente associadas com métricas de processo e de produto e com métricas de projeto.

Quando se trata de medir software, não se tem um entendimento amplamente aceito do que seja qualidade e tampouco existem medidas de qualidade de software comumente aceitas, por isso, segundo Jorgensen (1999), é necessário um *framework* para reflexões, a respeito de medição de qualidade, sobre:

- Pré-condições para medição da qualidade.
- Como analisar a significância das medidas de qualidade.
- Como interpretar e usar os valores mensurados.

Dependendo de como se define qualidade de software, a mesma pode ser medida diretamente, indiretamente ou de forma prognosticada. As pré-condições diferem para os diversos tipos de medição. Seguindo o raciocínio do mesmo autor, anteriormente referenciado, as pré-condições para medição direta de qualidade de software são as seguintes:

- O sistema relacional empírico de qualidade de software é estabelecido. Isto significa que, pelo menos, poder-se-ia ter um entendimento comum de relações de mesma qualidade e de qualidade melhor que, isto é, deve-se ser capaz de decidir se o software A tem melhor, a mesma ou pior qualidade que o software B.
- Um sistema numérico ou simbólico com relações formais equivalentes às relações empíricas de qualidade é estabelecido, por exemplo, as relações formais "=" e ">" representando, respectivamente, as mesmas relações e melhor qualidade no sistema empírico relacional.
- Uma medida (mapeamento) da qualidade do atributo de software para números ou símbolos é definida.
- Uma ferramenta ou método de medição, implementado à medida que é adotado.

As pré-condições para medição indireta da qualidade de software são:

- As pré-condições para medição dos atributos medidos diretamente, são encontradas (satisfeitas).
- Uma conexão empírica completa entre os atributos medidos diretamente e a qualidade do software medida indiretamente é estabelecida. A conexão se completa quando todos os aspectos do entendimento acordado de qualidade de software podem ser determinados a partir de atributos medidos diretamente.
- A conexão é cuidadosamente traduzida em um sistema relacional formal.

A predição (prognóstico) de qualidade de software é similar à medição indireta de software. A diferença é que os prognósticos não requerem uma conexão empírica completa ou uma tradução acurada para o sistema relacional formal, isto é, não é necessário que todas as relações empíricas sejam preservadas no sistema relacional formal (JORGENSEN, 1999).

A medição possibilita que gerentes e profissionais possam entender melhor o processo de engenharia de software, bem como o produto de software derivado de tal processo. Usando medidas diretas e indiretas, as métricas de produtividade e qualidade do processo e do produto de software podem ser definidas, Pressman (1997).

Tanto as métricas diretas como as indiretas são utilizadas na indústria de software. As métricas diretas utilizam-se das linhas de código enquanto as métricas indiretas derivam de medidas do domínio da informação e da avaliação subjetiva da complexidade do problema. Ambas são medidas do software e do processo entre do qual o mesmo é desenvolvido.

Métricas de qualidade de software podem ser divididas em métricas de qualidade de processo e métricas de qualidade de produto. A essência da engenharia de qualidade é investigar o relacionamento entre métricas aplicadas a processo, características de projeto (*design*) e qualidade de produto final e, baseados nas conclusões, construir qualidade tanto no processo como no produto. Além disso, a qualidade deve ser vista a partir da perspectiva de ciclo de vida de software completo e, relacionado a isso, deve-se incluir métricas que medem o nível de qualidade do processo de manutenção como outra categoria de métricas da qualidade de software (KAN, 2002).

Para Jorgensen (1999), muitos dos padrões de qualidade de software, tais como, a norma ISO 9001/9000-3 e o modelo CMM requerem ou recomendam a medição da qualidade de software. Entretanto, há um descompasso entre a exigência de que a medição da qualidade deveria ser executada e as diretrizes sobre como essas medições deveriam ser realizadas. Esse mesmo autor cita como exemplo, que o padrão de qualidade de software ISO 9000-3 (o qual é a guia de orientação para o uso da ISO 9001 na produção de software) declara na seção 6.4.1 que: “não há atualmente nenhuma medida aceita universalmente de qualidade de software”, o fornecedor de produtos de software deveria coletar e atuar sobre as medidas quantitativas de qualidade desses produtos. Desse modo, ao mesmo tempo, que reforça a necessidade de medição da qualidade, a ISO 9000-3 admite, que não há uma medida de qualidade aceita universalmente.

É freqüente a argumentação de que é muito difícil conseguir a medição de software na prática. Para Rifkin (2001) as medições tradicionais contemplam as decisões que apóiam o aumento da produtividade e da qualidade e a redução de custos, considerados elementos chave para organizações estrategicamente focadas na excelência operacional, as quais oferecem uma lista pequena e limitada de serviços, mas oferecem produtos excelentes a preços competitivos. Entretanto, é importante salientar que as diferentes organizações têm diferentes necessidades de medição e diferentes modos de lidar com essas necessidades, dado que nem todas as organizações alcançam a excelência operacional, ainda que tenham altas prioridades quantos aos níveis de qualidade dos elementos acima citados. Apesar dessas disparidades, segundo o mesmo autor, as medições se aplicam à todas as áreas de melhoria do processo de software.

Para Lavazza (2000) a medição é um fator chave para o gerenciamento e a melhoria do desenvolvimento de software, uma vez que o processo de medição objetiva definir e colocar em funcionamento um programa de medição (que é um conjunto de métricas, específicas ao contexto) e descrever as diretrizes e procedimentos para coleta de dados e análise. Ressalta, também, que medição eficaz e eficiente requer suporte metodológico e técnico.

Conforme explicitado por Mendonça (2000), na comunidade de software existem muitos grupos envolvidos nos processos de desenvolvimento, manutenção e gerenciamento de software, os quais necessitam usar medição para caracterizar, controlar, predizer e melhorar esses processos. Argumenta ainda que, em geral, as organizações de software têm desenvolvido seus *frameworks* de medição baseado em uma diversidade de fontes ou necessidades, sem um conjunto de objetivos bem estruturados. Esse cenário pode conduzir ao uso de dados e medições precariamente estruturados, o que pode levar as organizações de software a perderem sua compreensão global dos dados (e sua utilidade) em *frameworks* de medição grandes e pobremente estruturados.

Ressalta ainda (MENDONÇA, 2000) que é comum encontrar organizações de software que:

- Coletam dados redundantes.
- Coletam dados que ninguém usa.
- Coletam dados que poderiam ser úteis a pessoas que nem mesmo sabem que eles existem dentro da organização.

Por essas razões, a melhoria das medições da organização é um problema relevante para muitas organizações de software e, dito autor, acredita que a solução para esse problema deve tratar duas questões chave:

- Melhor entender e estruturar a medição em andamento.
- Melhor explorar o dado que a organização já coletou.

O Paradigma GQM[29] é uma técnica sistemática para desenvolvimento de programas de medição para processos e produtos de software e está baseado no pensamento de que a medição deveria ser orientada a objetivo, isto é, coleções de dados baseadas em uma lógica (um raciocínio) explicitamente documentadas, de acordo com Lavazza (2000).

Para Basili (1995), a abordagem GQM fornece um método para identificar e controlar processos chave de negócios de um modo mensurável. Uma organização pode usá-lo para definir métricas durante o projeto, processo e produto de software e, as métricas resultantes são adaptadas à organização e aos seus objetivos e refletem a significação de qualidade a partir de diferentes pontos de vista (desenvolvedores, usuários, operadores e assim por diante).

Um modelo GQM é uma hierarquia que inicia com um objetivo, especifica o propósito de medição, o objeto a ser medido, problemas para medir e ponto de vista de quem faz a medição (BASILI, 1995).

Para Rifkin (2001), o GQM é um método em cascata, que vai dos objetivos de negócio até as decisões que necessitam de informações, para determinar o que medir para fornecer tal informação. O Paradigma GQM foi proposto, segundo Mendonça (2000), como um recurso para medir software de uma maneira resoluta e, o primeiro passo deste paradigma, é definir objetivos de medição para as necessidades específicas de uma organização. Tais objetivos são refinados de um modo operacional e fácil de ser trabalhado, em um conjunto de questões quantificáveis. Essas questões por sua vez indicam um conjunto específico de métricas e

[29] Do inglês: *Goal Question Metrics*.

dados a serem coletados. Para Kitchenham (2001), os paradigmas de métricas tais como o GQM, são importantes para assegurar que medidas são úteis, simples e diretas. Contudo, assim como outras abordagens, o GQM estaciona no ponto em que conceitos de medições ou métricas específicas são definidos. Ou seja, essas abordagens não definem como tais medidas podem ser coletadas e armazenadas e, em geral, não definem como podem ser analisadas. Ainda assim, a utilização deste paradigma tem sido bem sucedida em várias organizações.

Existem muitas razões, segundo Pressman (1995), que justificam a necessidade de se medir software, ainda que esta não possa ser considerada uma tarefa fácil. O software é medido normalmente, numa tentativa de indicar a qualidade do produto; avaliar a produtividade das pessoas envolvidas no processo de construção do produto; avaliar os benefícios (em termos de produtividade e qualidade) oriundos do uso de métodos, técnicas, ferramentas e outros dispositivos utilizados para dar suporte ao processo de desenvolvimento de software; criar uma base de dados históricos visando fornecer dados para futuras estimativas de projeto, avaliar o processo por meio do qual o produto é construído etc.

Nas seções seguintes serão abordadas duas atividades que fazem parte do processo de verificação e validação, o qual assegura que o software está em conformidade com sua especificação e atende as necessidades do usuário, dado que para satisfazer os objetivos do processo de verificação e validação, tanto a técnica dinâmica como a análise estática de checagem do sistema, devem ser usadas. Teste e inspeção de software estão inseridos na atividade de verificação e validação de software e, dadas as suas características, fazem parte da abordagem proposta sendo, portanto, objetos de estudo e considerações.

4.2 INSPEÇÕES DE SOFTWARE

Um sistema de qualidade de software tem dois objetivos. O primeiro, é construir qualidade no software desde o início. Isto significa garantir que o problema ou necessidade a ser tratada

está clara e cuidadosamente explicitada, e que os requisitos para a solução estão apropriadamente especificados, expressos e entendidos (HORCH, 2003).

Para que a qualidade seja construída no sistema de software desde o início, os requisitos de software devem ser claramente entendidos e documentados. Enquanto os requisitos e as necessidades atuais do usuário que os mesmos devem satisfazer são conhecidos e entendidos, há uma pequena probabilidade de que o usuário esteja satisfeito com o sistema de software a ele entregue.

Ainda segundo Horch (2003), o segundo objetivo do sistema de garantia da qualidade é manter a qualidade no software ao longo do ciclo de vida do software. O sistema de garantia de software compreende dez elementos, os quais contribuem para ambos os objetivos. Embora cada elemento possa contribuir com ambos os objetivos, existe um forte relacionamento entre alguns elementos e um ou outro dos dois objetivos, os quais são listados como segue:

- Padrões.
- Revisões.
- Teste.
- Análise de defeitos.
- Gerenciamento de configuração.
- *Security*.
- Educação.
- Gerenciamento de vendedor.
- *Safety*.
- Gerenciamento de risco.

A inspeção de software faz parte de uma atividade mais ampla, conhecida como verificação e validação. A verificação refere-se ao conjunto de atividades que garantem que o software implementa uma função específica corretamente. A validação refere-se a um conjunto diferente de atividades que garantem que o software construído atende às exigências do usuário.

Para (HORCH, 2003), revisões são a primeira e principal forma de atividade de controle de qualidade. Controle de qualidade está relacionado com a busca por erros ou defeitos em vários produtos de desenvolvimento de software. Controle de qualidade tem o propósito de detectar e eliminar os defeitos no produto. Revisões são a linha de frente desse propósito. Revisões são mais econômicas que teste porque consomem menos recursos que o teste. São curtas, requerem pequenos grupos de revisores e podem ser programadas e re-programadas com um mínimo de impacto em termos de orçamento e cronograma.

Existem duas abordagens fundamentais para verificação. A primeira consiste em experimentar o comportamento de um produto para ver se o produto funciona conforme o esperado (isto é, testar o produto). A outra consiste em analisar o produto – ou qualquer documentação de projeto relacionada a ele – para inferir sua correta operação, como uma conseqüência lógica das decisões de projeto. As duas categorias técnicas de verificação são também classificadas como dinâmica e estática, dado que a primeira requer – por definição – executar o sistema a ser verificado, enquanto a segunda não. São técnicas complementares e não excludentes.

Conforme dito anteriormente, a verificação estática não requer que o programa seja executado, ao contrário, compreende o exame do código fonte de um programa ou um projeto e a detecção de falhas antes da execução. Cada erro pode ser considerado isoladamente. Erros de interação não são significativos e um componente completo pode ser validado em uma única sessão. Isso significa que menos tempo é requerido para encontrar cada erro. Verificação estática é, portanto, mais efetiva em termos de custo que teste de defeitos, Sommerville (1996). Isto não significa, porém, que a verificação estática pode substituir o teste, ao contrário, deve ser usada como um processo de verificação inicial para encontrar a maioria dos defeitos. A verificação estática pode averiguar a conformidade com uma especificação, mas não pode prever o comportamento dinâmico.

Como qualquer outro trabalho de engenharia, todos os processos de projeto e todos os produtos desses processos devem ser verificados. Em certo sentido, o próprio processo de verificação deve ser verificado, ou seja, uma vez que o sistema tenha sido testado quanto ao seu comportamento, os experimentos deveriam ser averiguados para ver se foram feitos corretamente - verificação da validade dos experimentos, segundo Ghezzi (1991).

A atividade de inspeção diz respeito às técnicas estáticas, as quais se referem à análise e verificação das representações do sistema tais como documentação de requisitos, diagramas de projeto e código fonte do programa. A inspeção pode ser aplicada em todos os estágios do processo de desenvolvimento por meio de revisões estruturadas, o que inclui inspeções de programa e análise e verificação formal. Podem somente checar a correspondência entre um programa e sua especificação (verificação), não podem demonstrar que o software é operacionalmente útil (SOMMERVILLE, 1996).

Para Pressman (1997), revisões de software são um ‘filtro’ para o processo de engenharia de software. Isto é, as revisões são realizadas em vários pontos durante o processo de desenvolvimento de software e servem para descobrir erros – que podem, então, ser removidos. Revisões de software servem para ‘purificar’ os artefatos de software que resultam da análise, projeto e codificação. Uma revisão – qualquer revisão – é um meio de usar a diversidade de um grupo de pessoas para:

- Salientar as melhorias necessárias no produto de uma única pessoa ou de uma equipe.
- Confirmar aquelas partes de um produto, nas quais a melhoria é ou não desejada ou necessária.
- Realizar um trabalho técnico de qualidade mais uniforme ou pelo menos mais previsível, que o que pode ser obtido sem revisões, a fim de tornar o trabalho técnico mais gerenciável.

Existem muitos tipos diferentes de revisões que podem ser conduzidas como parte da engenharia de software. Uma reunião informal em volta de uma máquina de café é uma forma de revisão, onde problemas técnicos são discutidos. Uma apresentação formal do projeto para clientes, gerentes e equipe técnica é uma forma de revisão. Revisão técnica formal - algumas vezes, denominada *walkthrough* – é um filtro mais efetivo do ponto de vista da garantia da qualidade. Conduzido por e para engenheiros de software, a revisão técnica formal é um meio efetivo de melhoria da qualidade de software (PRESSMAN, 1997).

O objetivo principal da revisão técnica formal é encontrar erros durante o processo de modo que não se tornem defeitos após a liberação do software. O benefício óbvio da revisão técnica

formal é a descoberta de erros, de modo que tais erros não se propaguem aos estágios posteriores no processo de desenvolvimento (PRESSMAN, 1995).

De acordo ainda com Pressman (1997) a revisão técnica formal é uma atividade de garantia da qualidade de software, cujos objetivos são:

- Descobrir erros na função, lógica ou implementação.
- Verificar em quê o software sob revisão atende seus requisitos.
- Assegurar que o software foi representado de acordo com padrões pré-definidos.
- Obter software que é desenvolvido de uma maneira uniforme.
- Tornar projetos mais genéricos.

Em adição, a revisão técnica formal serve como um campo de treinamento, possibilitando que os engenheiros possam observar diferentes abordagens para análise, projeto e implementação de software. A revisão técnica formal também serve para promover *backup* e continuidade, porque os envolvidos se familiarizam com partes do software que podem não ser vistas de outro modo.

O benefício especial das inspeções é o fato de descobrir erros antes que o teste possa mostrar o problema. A vantagem final obtida com as inspeções é que o dado obtido por meio de inspeções e usados para identificar as áreas problemas mais comuns, determinam suas causas e mudam o processo de desenvolvimento, no sentido de reduzir e prevenir erros (SMITH, 2003).

Um dos benefícios notórios das inspeções é a detecção de erros antes que se propaguem para as fases seguintes do processo de desenvolvimento de software. A detecção e, a conseqüente eliminação de grande parte dos erros, possibilita que o processo de inspeção reduza substancialmente o custo das fases posteriores de desenvolvimento e manutenção.

A premissa básica das inspeções de software é que as mesmas detectam e removem defeitos, antes que se propaguem às fases subseqüentes de desenvolvimento, onde a descoberta e a

correção implicam em altos custos (SOMMERVILLE, 1996; BIFFL, 2000; LAITENBERGER, 2001; GENUCHTEN, 2001; PRESSMAN, 1995; PRESSMAN, 1997).

A verificação deve ser levada a efeito em cada fase do processo de desenvolvimento, uma vez que a verificação de desempenho, após a execução do código disponível, poderia dificultar o reparo de quaisquer defeitos que tenham sido detectados. Dados experimentais de projetos práticos têm mostrado que o custo de remover um erro após o software ter sido completamente desenvolvido, é muito maior que a remoção de erros nos estágios iniciais (GHEZZI, 1991).

Toda a qualidade de software deve ser verificada. Não se deve verificar apenas se o software implementado se comporta de acordo com os documentos de especificação, é preciso certificar também outras propriedades do software, como, portabilidade, desempenho, funcionalidade e manutenibilidade, dentre outras.

A atividade de verificação como qualquer outra atividade do processo de desenvolvimento de software, deve seguir princípios e técnicas apropriadas, não podendo ser relegada à percepção ou experiência dos envolvidos em sua execução.

A diferença chave entre inspeções de programa e outros tipos de revisões é que o principal objetivo das inspeções é a detecção de defeitos ao invés de se estender em questões mais amplas de projeto. Defeitos podem ser erros lógicos, anomalias no código que podem indicar uma condição de erro ou a não concordância com padrões organizacionais ou de projeto (SOMMERVILLE, 1996).

Ainda de acordo com Sommerville (1996), quando uma organização decide introduzir inspeção no processo de software, deve:

- Preparar uma lista de prováveis erros para orientar o processo de inspeção. Estabelecida no início do processo de inspeção e atualizada à medida que mais experiência vai sendo adquirida no processo de inspeção.

- Aceitar que verificação estática irá confrontar custos de projeto, de modo que, mais dinheiro será gasto nas fases iniciais de um projeto de software, o que deveria, conseqüentemente, reduzir os custos com testes.
- Definir uma política que ratifique que as inspeções são parte do processo de verificação e não avaliações pessoais.
- Estar preparada para investir em treinamento de equipes líderes de inspeção.

O processo de inspeção é formalmente executado por uma equipe pequena, que analisa as representações do sistema e salienta os possíveis defeitos. As organizações estabelecem as equipes e o número de membros que as compõem, baseado em suas experiências com inspeções e, dentro da própria organização, diferentes papéis podem ser adotados pela mesma pessoa, bem como o tamanho da equipe pode variar de uma inspeção para outra. A Tabela 12 mostra uma sugestão de seis papéis (funções), que podem ser adotados em um processo de inspeção.

Tabela 12. Papéis (funções) no processo de inspeção (SOMMERVILLE, 1996)

Função	Descrição
Autor ou proprietário	Programador ou projetista responsável por produzir o programa ou documento. Responsável por anotar os defeitos encontrados durante a inspeção.
Inspetor	Encontra erros, omissões e inconsistências em programas e documentos. Pode, também, identificar questões mais amplas as quais revelam o escopo da equipe de inspeção.
Leitor	Parafraseia o código ou documento em uma reunião de inspeção.
Escrevente	Registra os resultados da reunião de inspeção.
Presidente ou moderador	Gerencia o processo e facilita a inspeção. Relata os resultados para o moderador chefe.
Moderador chefe	Responsável pelas melhorias no processo de inspeção, atualização da lista de prováveis erros, desenvolvimento de padrões etc.

Uma inspeção utiliza padrões e listas para encontrar e determinar como muitas deficiências de produtos e processos são possíveis. As características chave de inspeções são: preparação individual, coleta de dados e uma sintaxe definida para o registro de defeitos (GENUCHTEN, 2001).

Ao longo do desenvolvimento do ciclo de vida, particularmente nos estágios iniciais de desenvolvimento de software, a inspeção de documentos de software é uma medida efetiva para a garantia da qualidade, para detectar defeitos e fornecer *feedback* oportuno sobre qualidade para gerentes e desenvolvedores (BIF, 2000).

O processo de inspeção pressupõe algumas condições que são essenciais ao seu cumprimento, isso implica que o processo de inspeção anteriormente à sua execução deve ser cuidadosamente planejado, assim, antes que seja iniciada uma inspeção de programa, de acordo com Sommerville (1996) é fundamental que:

- Haja uma especificação precisa do código a ser inspecionado, dado que é impossível inspecionar um componente no nível de detalhe requerido para detectar defeitos sem uma especificação completa.
- Os membros da equipe de inspeção tenham familiaridade com os padrões organizacionais.
- Haja uma visão recente e sintaticamente correta do código disponível, uma vez que não há um ponto na inspeção de código que seja ‘quase completa’, mesmo que um atraso provoque atrasos no cronograma.

O moderador é responsável pelo planejamento da inspeção, o que compreende a seleção de uma equipe de inspeção e a organização de local apropriado para as reuniões, além de assegurar que o material a ser inspecionado e a sua especificação, estejam completos. O material a ser inspecionado é apresentado à equipe de inspeção durante o estágio de supervisão, vindo a seguir um período de preparação individual. Cada membro da equipe de inspeção estuda a especificação e o programa e busca defeitos no código.

Um processo genérico de inspeção é mostrado na Figura 27, o qual pode ser adaptado conforme as exigências das organizações.

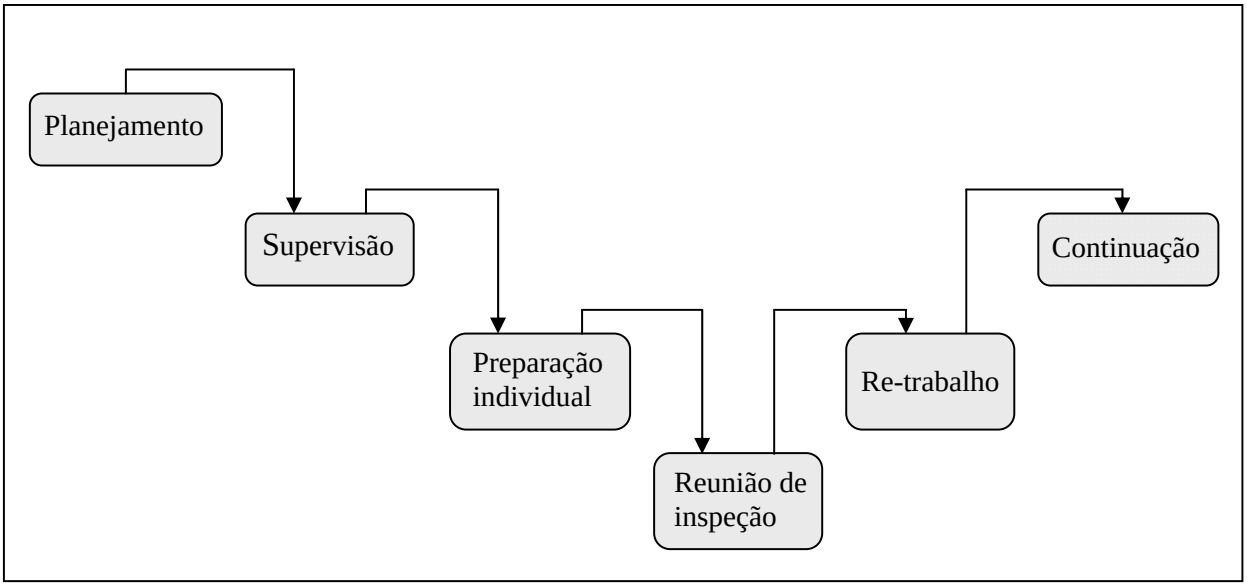


Figura 27. O processo de inspeção (SOMMERVILLE,1996)

O processo de inspeção em si deve ser relativamente curto e envolver exclusivamente a identificação de defeitos, anomalias e o não cumprimento de padrões. À equipe de inspeção cabe a tarefa de detectar defeitos, porém, não deve sugerir como esses defeitos devem ser corrigidos e tampouco, recomendar mudanças em outros componentes. Cabe ao autor do programa sua alteração e a correção dos problemas identificados. No estágio posterior, o moderador decide se há necessidade de uma re-inspeção de código e, em se decidindo que não é necessária uma inspeção completa e que os defeitos foram detectados com sucesso, o documento é aprovado pelo moderador e divulgado a todos os envolvidos no processo de inspeção.

Segundo o entendimento de Sommerville (1996), o processo de inspeção deve ser conduzido, por uma lista de erros comuns de programação e algumas possíveis verificações, que podem ser realizadas durante o processo de inspeção, as quais são mostradas na Tabela 13.

Tabela 13. Controles de inspeções (SOMMERVILLE, 1996)

Classe de falhas	Controle de inspeção
Falhas de dados	Todas as variáveis do programa são inicializadas antes que seus valores sejam usados? Todas as constantes foram nominadas? Deve a fronteira menor do conjunto, ser 0 (zero), 1 (um) ou qualquer outro? Deve a fronteira maior do conjunto, ser igual ao tamanho do conjunto ou ao tamanho – 1? Se cadeias de caracteres são usadas, um delimitador é assinalado explicitamente?
Falhas de controle	Para cada declaração condicional, há condição correta? Há certeza de término para cada laço? As declarações compostas são corretamente suportadas? Nas declarações do tipo "caso", todos os casos possíveis são considerados?
Falhas de entrada e saída	Todas as variáveis de entrada são usadas? Todas as variáveis de saída indicam um valor antes que saídas sejam produzidas?
Falhas de interface	Todas as funções e procedimentos que realizam chamadas têm o número correto de parâmetros?
	Os tipos de parâmetros formal e atual são comparados? Todos os parâmetros estão na ordem correta? Se componentes acessam memória compartilhada, eles têm o mesmo modelo de estrutura de memória compartilhada?
Falhas de gestão de armazenamento	Se uma estrutura ligada é modificada, todos os *links* são corretamente assinalados? Se armazenamento dinâmico é usado, os espaços são alocados corretamente? Espaço é explicitamente desalocado após não ser mais requerido?
Falhas de gestão de exceção	Todas as possíveis condições de erros foram consideradas?

De acordo com Genuchten (2001), os resultados de uma inspeção envolvem tipicamente, três indicadores de desempenho: **eficácia** é o número de defeitos detectados por página; **eficiência**; é o número de defeitos detectados por pessoa/hora envolvida. Eficácia e eficiência podem ser calculadas tanto para a preparação como para a reunião. **Rendimento** (produção), é a fração de defeitos descobertos pela inspeção, em oposição àqueles que escaparam e são detectados nas fases posteriores do desenvolvimento. Inspeções bem executadas deveriam ser

capazes de encontrar de 60 a 80% de defeitos no ciclo de vida, antes que o software seja testado.

Considerando que uma organização obtém experiência quando do planejamento e execução do processo de inspeção, pode usar os resultados de cada processo, como meio de melhorar o processo de inspeção, uma vez que é possível fazer uma análise dos defeitos encontrados durante o processo de inspeção. Assim, a equipe de inspeção, juntamente com o autor da parte inspecionada, pode sugerir motivos para que esses defeitos tenham ocorrido. Desse modo, onde for possível, o processo deve ser modificado para eliminar as razões para a ocorrência de defeitos, evitando que possam ocorrer futuramente em outros sistemas.

Um exemplo de melhoria de processo de software é o desenvolvimento e realização de inspeções de documentos de desenvolvimento de software, como uma técnica madura para detectar e evitar defeitos de software (GENUCHTEN, 2001).

Um argumento valioso a ser considerado quanto ao processo de inspeção é o fato de que sua realização sistemática está relacionada à melhoria da qualidade e da produtividade e, um outro fator, também muito importante, conforme explicitado em Butler (1997), é que a implementação de procedimentos de revisão, os quais são aplicados freqüentemente e de modo consistente, em toda a organização, para detectar defeitos em requisitos, projeto ou código, é um passo importante para se obter o nível três do CMM.

Tanto os testes como as revisões funcionam como aprimoramento do produto, contudo, as revisões funcionam também na melhoria do processo e, segundo Weinberg (1996), geralmente, são as primeiras e se constituem um método fácil de aprimoramento de processo a ser introduzido nas organizações. A possibilidade de múltiplas formas de pensar e de diferentes padrões de pensamento conduz a uma escolha ampla de idéias sobre como corrigir falhas, dado que a maioria dos erros que um membro da equipe deixou passar não coincidirá com os erros que outro membro não consegue perceber, assim sendo, equipes de revisão podem fazer um trabalho melhor de reparação de erros, conforme mostra a Figura 28.

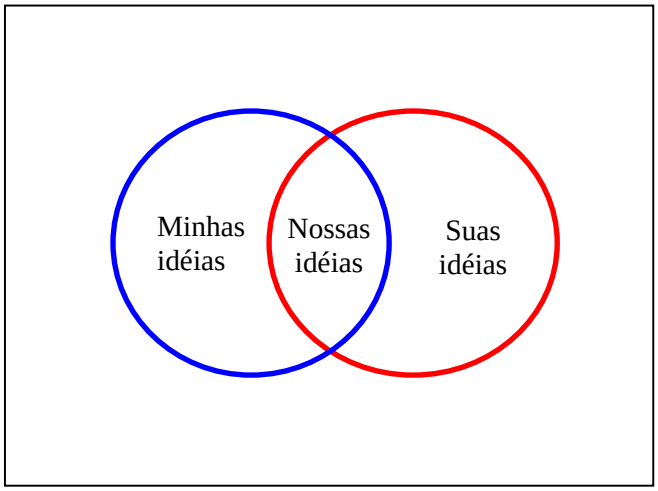


Figura 28. Trabalho em equipe na reparação de erros (WEINBERG, 1996)

Considerando que o trabalho de software consiste de muitas e variadas tarefas, é pouco provável que um determinado tipo de personalidade, um conjunto de habilidades ou um ponto de vista seja a melhor opção para todas as partes de um trabalho de software, o que, por si só, justifica a necessidade de diferentes profissionais para compor as equipes para atuar no processo de revisões técnicas formais. Dada a sua importância para a área de engenharia de software, Weinberg (1996) propõe diversas formas de equipe, sendo que, as que têm relação direta com a tarefa de inspeção, são resumidamente descritas a seguir.

Equipes para apontar as falhas. Treinadas para trabalhar em conjunto na identificação de falhas e impedir que relatórios sobre falhas circulem aleatoriamente e não cheguem até os responsáveis por resolvê-las. Sua composição diversificada possibilita a identificação eficiente de falhas. A Figura 29 explicita porque é fácil compor as equipes de identificação de falhas e conseguir que se tornem rapidamente produtivas, uma vez que, os erros que ambos os membros deixaram escapar, serão em número menor do que aqueles que um dos membros deixaria de detectar. Isso possibilita que as equipes possam realizar um trabalho melhor de identificação de falhas.

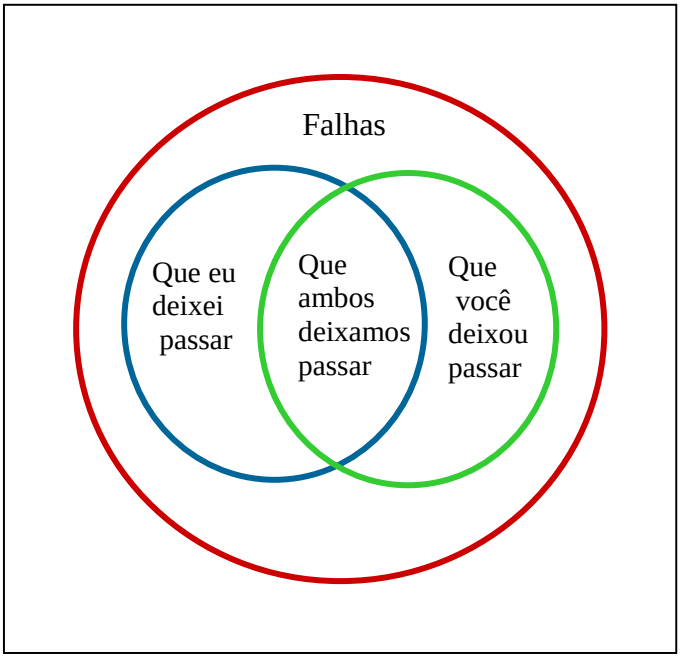


Figura 29. Porque as equipes podem melhor identificar as falhas (WEINBERG, 1996)

Equipes para a resolução de falhas. A idéia de equipes de localização de falhas se estende às equipes de resolução de falhas. A Figura 30 mostra que distintas opiniões das possíveis correções oferecem mais opções de escolha para a correção de erros, ou seja, as equipes podem fazer um trabalho melhor de reparação de erros, devido ao fato de que, diante de múltiplas formas de pensar, existe um leque maior de idéias sobre como corrigir as falhas encontradas.

Equipes de revisão técnica. O processo de resolução de falhas consiste primeiramente na escolha de uma solução e, posteriormente, da condução das revisões técnicas referentes a cada solução proposta para impedir efeitos colaterais não esperados. Durante a atividade de remoção de falhas, as maiores perdas, podem resultar de efeitos colaterais ou de falhas introduzidas na tentativa de solucionar outras falhas, por isso, segundo Weinberg (1996), equipes devem ser treinadas para trabalhar na prevenção de efeitos colaterais quando da correção de falhas. Uma equipe adequadamente estruturada tem melhores condições de prever efeitos colaterais do que um indivíduo sozinho, devido à dinâmica ilustrada nas Figuras 28 e 29. Além dessas, existem também, as: equipes de desenvolvimento de projetos; equipes

independentes para assegurar a qualidade do software; equipes para avaliação da engenharia de software e; os grupos de processo de engenharia de software.

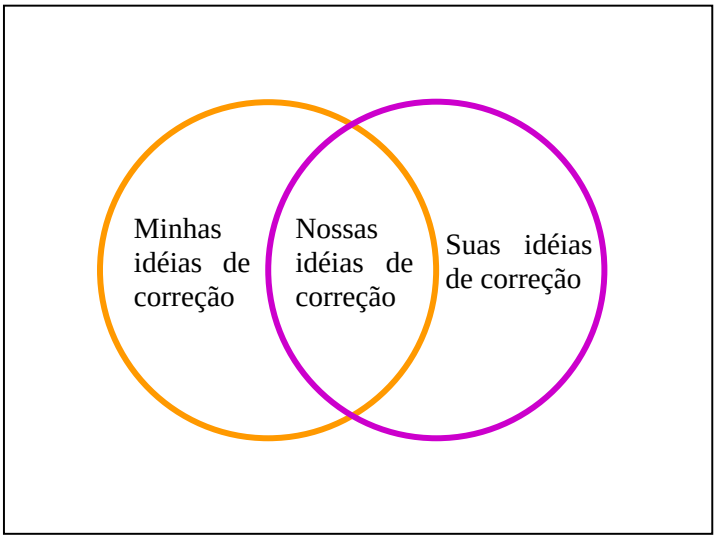


Figura 30. Porque as equipes são potencialmente melhores que os indivíduos para reparar erros (WEINBERG, 1996)

Um aspecto positivo das revisões técnicas é o fato de possibilitarem uma aprendizagem constante durante sua realização. Pode-se aproveitar o poder das revisões para melhorar outros processos, classificando problemas de codificação e divulgando estatísticas e classificando falhas quanto à sua origem, o que pode indicar onde é necessária a melhoria, no projeto ou nos requisitos, por exemplo. Esse é um dos benefícios das revisões, daí a importância do momento em que são realizadas Weinberg (1996).

A realização tardia de revisões possibilita a detecção de mais falhas do que em revisões realizadas mais cedo, dificultam, porém, a identificação da origem das falhas, sendo que as revisões realizadas mais cedo ajudam essa identificação, sendo esse o verdadeiro benefício das revisões. Desse modo, programando as revisões o mais cedo possível, pode-se melhorar tanto a qualidade do produto como a qualidade do processo.

Revisões podem levar à recomendação de que o produto deve ser descartado ou totalmente refeito e, por isso mesmo, tendem a produzir informações valiosas sobre o processo (WEINBERG, 1996).

A garantia da qualidade de software envolve um conjunto de atividades técnicas espalhadas por todo o processo de desenvolvimento, com o objetivo de assegurar que tanto o processo de desenvolvimento quanto o produto final atinjam os níveis de qualidade especificados.

Inspeção de software é uma das atividades sistemáticas que visam a garantir a qualidade do produto e do processo e deve ser exercitada durante todo o processo de desenvolvimento de software.

Pelo processo de verificação, o sistema deveria ser verificado e validado em cada fase do processo de desenvolvimento, fazendo uso de documentos produzidos durante a fase anterior. Verificação e validação começam com revisões de requisitos e continuam entre o projeto e revisões de código até chegar ao teste do produto Sommerville (1996).

4.3 TESTE DE SOFTWARE

Teste é uma importante atividade técnica para assegurar a qualidade de software. O teste sistemático contribui para que defeitos sejam identificados e, conseqüentemente, eliminados durante o processo de depuração, de forma que o produto liberado tenha o menor número de erros possível e, por isso mesmo, seja mais confiável.

A atividade de teste tem dois objetivos principais: fazer um julgamento sobre a qualidade ou aceitabilidade e descobrir problemas ou erros. A função primária do teste é avaliar a “saúde” do projeto e para avaliar a qualidade dos produtos, sua aderência a padrões, funcionalidade e sua conformidade com os requisitos. Um bom programa de teste é essencial para um

desempenho operacional confiável, e reduzirá significativamente os custos de manutenção e suporte (SMITH, 2003).

De acordo com (HORCH, 2003), o programa de teste começa na fase de requisitos e efetivamente nunca se acaba. Dado que o próprio teste é baseado nos requisitos de software, o planejamento da atividade de teste tem início durante a fase de requisitos e continua durante todo o ciclo de vida do desenvolvimento de software. Assim como os requisitos mudam, o teste também deve mudar para acompanhar as mudanças nos requisitos.

A confiabilidade é a mais importante característica dinâmica da maioria dos sistemas de software. Como, em geral, a expectativa de qualidade em todos os sistemas tem aumentado, não é mais aceitável liberar software com falhas. Software no qual não se pode confiar, resulta em alto custo para o usuário final (SOMMERVILLE, 1996).

Segundo Pailo (1997), o teste de software se constitui um dos elementos para aprimorar a produtividade e ajudar a fornecer evidências da confiabilidade e da qualidade do software em complemento a outras atividades ao longo do processo de desenvolvimento de software. Os objetivos do teste é encontrar defeitos e verificar se o software atende seus requisitos, tal como entendidos pelo usuário (HORCH, 2003).

O principal método usado para encontrar algum nível de confiança na fidedignidade do software é o teste de programa (NTAFOS, 2001).

De acordo com Chernak (2001), teste de software auxilia os desenvolvedores de aplicações a gerenciarem os riscos, encontrando e removendo defeitos no software antes que o mesmo seja liberado. Metodologia formal de teste define vários tipos de teste, incluindo o teste de função. Focalizando a funcionalidade do sistema e procurando tantos defeitos, quanto possível, esse teste suporta mais diretamente a responsabilidade para a qualidade do sistema final. Implementar o teste de função como um processo de teste formal permite aos testadores cobrir melhor a complexidade funcional da aplicação de software sob teste. Documentos de

plano de teste e especificação de casos de teste são importantes artefatos do processo de teste formal.

Em se tratando de sistemas complexos, casos de teste são críticos para teste efetivo. Contudo, o mero fato de que testadores usam especificações de casos de teste, não garante que os sistemas sejam suficientemente testados. Numerosos outros fatores também determinam se os testadores têm realizado bem o teste e, se este foi efetivo.

Em última análise, os testadores podem somente avaliar a efetividade completa do teste quando um sistema está em produção. Contudo, se esta avaliação descobre que um sistema foi insuficientemente testado ou que os casos de teste não foram efetivos, é difícil trazer proveito para o projeto atual. Para reduzir tal risco, a equipe de projeto pode avaliar a efetividade do teste, realizando no processo uma avaliação da efetividade do caso de teste. Desse modo, é possível identificar problemas e corrigir o processo de teste antes de liberar o sistema (CHERNAK, 2001).

O processo de teste de software pode ser definido como segue:

- Planejamento de teste. Nesta fase as principais tarefas são: a definição do escopo, dos objetivos e da abordagem para teste. O resultado principal é um documento de plano de teste.
- Projeto de teste. Envolve o projeto de casos de teste, tendo como resultado principal, o início das especificações dos casos de teste.
- Preparação e execução de teste. Nesta fase, a preparação do ambiente de teste, a execução dos casos de teste e a descoberta de defeitos, são tarefas necessárias e os principais resultados são os relatórios de defeitos.
- Avaliação e melhoria de teste. Aqui, a tarefa principal é analisar os resultados do teste e o principal resultado é um relatório sucinto de teste.

Em todas as etapas, exceto na última, há um número de fatores que determinam a efetividade dos casos de teste funcionais em um dado projeto.

Na Figura 31, é mostrado o processo de teste mais amplamente utilizado, segundo Sommerville (1996), o qual consiste de cinco estágios.

O modelo de teste de software, mostrado na Figura 31, é baseado na idéia de integração de sistema incremental onde componentes simples são integrados para formar módulos, os quais são integrados para formar subsistemas que, por sua vez, são integrados em um sistema completo. Em essência, o teste de integração deveria ser finalizado em um nível antes de se passar para o próximo nível.

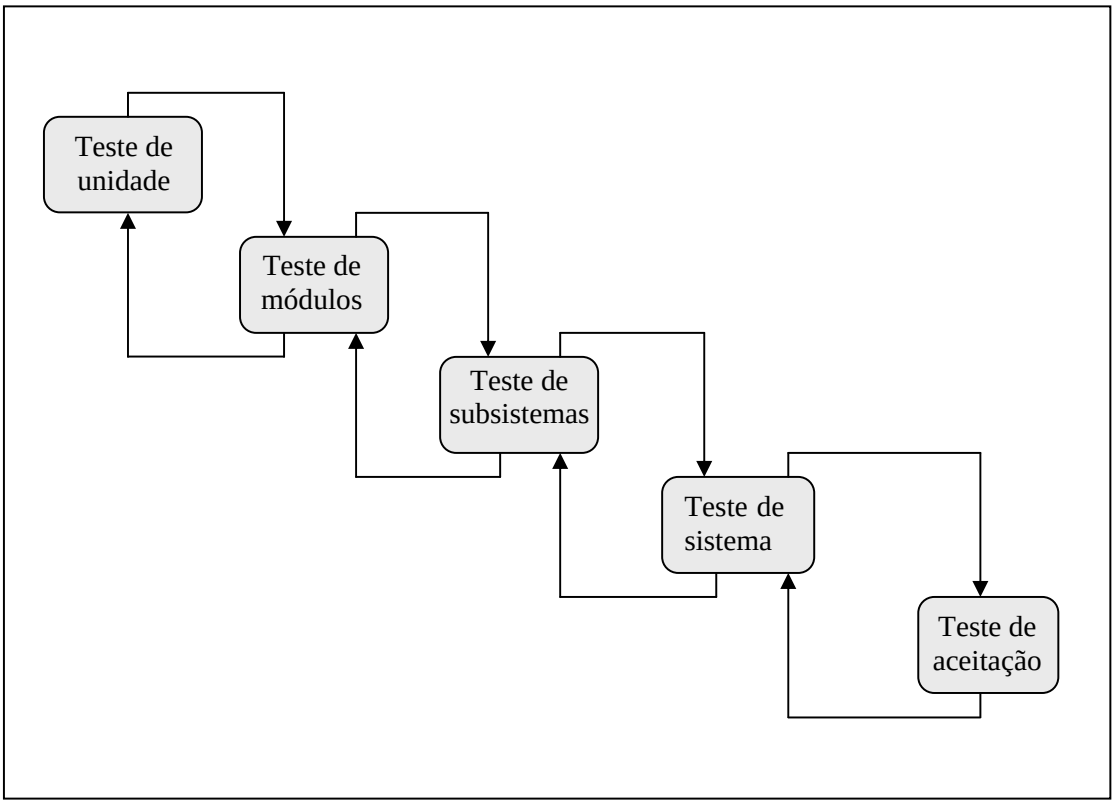


Figura 31. O processo de teste

Para Delamaro (2001), teste de software pode ser considerada uma atividade incremental que permeia a maior parte do – se não todo o – processo de desenvolvimento de software. Durante o ciclo de desenvolvimento de software a atividade de teste tem início com o desenvolvimento de módulos individuais ou unidades, definidas como sendo programas

testáveis que fornecem funcionalidades para um sistema de software. Uma unidade é testada antes que seja integrada a outras unidades para formar um subsistema ou sistema. Após a integração de unidades em subsistemas ou sistemas, os mesmos são testados para averiguar a conformidade com os seus requisitos. Durante o teste de unidade, constrói-se uma ou mais entradas de teste – conjunto de teste – e executa-se a unidade em contraposição a essas entradas, em alguma ordem convenientemente escolhida. Os subsistemas ou sistemas são testados de forma similar. Em qualquer caso, desenvolve-se um conjunto de teste e uma coleção de casos de teste para testar a unidade ou o subsistema em questão.

Para Sommerville (1996), em geral, a seqüência de atividades de teste é: teste de componente, teste de integração e teste do usuário. Contudo, como os defeitos são descobertos em qualquer estágio, os mesmos requerem modificações no programa para corrigi-los e isto pode exigir outros estágios no processo de teste a ser repetido. Conforme ilustra a Figura 31, os estágios do processo de teste são:

- **Teste de unidade**: componentes individuais são testados independentemente, sem outros componentes do sistema, para assegurar que operam corretamente.
- **Teste de módulo**: um módulo é composto de vários componentes dependentes, tal como uma classe de objetos, um tipo abstrato de dados ou uma coleção de procedimentos ou funções. Um módulo encapsula componentes relacionados que podem ser testados em outros módulos do sistema.
- **Teste de subsistema**: esta fase envolve o teste dos conjuntos de módulos, os quais foram integrados nos subsistemas. Os subsistemas podem ser projetados e implementados independentemente. Os problemas mais comuns que aparecem nos grandes sistemas de software referem-se às interfaces inadequadas entre os diversos subsistemas que o compõem. O processo de teste de subsistema deveria, portanto, concentrar-se na detecção de erros de interfaces, exercitando rigorosamente tais interfaces.
- **Teste de sistema**: Os subsistemas são integrados de modo a formarem um sistema completo. Nesta etapa, o interesse da atividade de teste está em encontrar erros que resultam de interações não previstas entre os subsistemas e os componentes do sistema e também em validar o sistema, para que atenda seus requisitos funcionais e não funcionais.

- **Teste de aceitação**: Estágio final do processo de teste antes que o sistema seja liberado para uso operacional. O sistema é testado com dados reais fornecidos pelo mantenedor ao invés de dados hipotéticos. O teste de aceitação pode revelar erros e omissões na definição dos requisitos do sistema, uma vez que o dado real exercita o sistema de diferentes maneiras a partir do dado de teste. O teste de aceitação também pode revelar problemas de requisitos, onde as facilidades do sistema realmente não atendem às necessidades do usuário ou o desempenho do sistema não é aceitável.

Não raro, o teste de aceitação é chamado de teste alfa. O processo de teste alfa continua até que o desenvolvedor do sistema e o cliente acordem que o sistema a ser liberado é uma implementação aceitável dos requisitos do sistema.

Quando um sistema é entendido como um sistema de software, um processo de teste, denominado teste beta é usado. O teste beta envolve a distribuição de um sistema a um número de usuários em potencial, os quais concordam em usar o sistema e, relatam os problemas encontrados durante o seu uso. Isto expõe o produto ao uso real e possibilita a detecção de erros, os quais, podem não ter sido previstos pelos desenvolvedores do sistema. Após esse *feedback*, o sistema é modificado e também distribuído para futuros testes beta ou para o mercado.

O teste de software é caro, assim, é necessário fazer um planejamento acurado, para conseguir tirar o máximo proveito e para controlar os custos dessa atividade. É necessário que as organizações de software façam cronogramas de testes realistas, forneçam recursos adequados para teste e não permitam cronogramas condensados, mesmo quando o cronograma de desenvolvimento falhe. É importante que as organizações reconheçam que testar um software é tão importante quanto desenvolvê-lo. É importante colocar algumas das pessoas mais experientes e capacitadas no grupo de teste e recompensá-las, com prêmios e retribuições, apropriadamente por suas contribuições e realizações (BASILI, 2004).

Teste é como qualquer outro projeto. Deve ser planejado, projetado, documentado, revisado e conduzido (HORCH, 2003).

4.3.1 Planejamento de teste

Para planejar e executar testes, os testadores devem considerar o software, a função que o mesmo executa, as entradas e como essas podem ser combinadas e o ambiente no qual o software eventualmente opera. Os principais componentes de um plano de teste estão ilustrados na Figura 32.

O planejamento de teste de software compreende o estabelecimento de padrões para o processo de teste e não para descrever teste de produto em si. Os planos de teste são idealizados pelos projetistas de software envolvidos no projeto e execução de teste de software. Não são meros documentos gerenciais, pois possibilitam que a equipe técnica tenha uma imagem global do teste do sistema e identifique seu próprio trabalho nesse contexto e fornecem informação à equipe técnica, a qual é responsável por assegurar que os recursos apropriados de hardware e de software estejam à disposição da equipe de teste.

O processo de teste.

Uma descrição das fases mais importantes do processo de teste.

Delinear requisitos.

Usuários são mais interessados no sistema, em encontrar seus requisitos e o teste deve ser planejado, de modo que todos os requisitos sejam individualmente testados.

Itens de teste.

Os produtos do processo de software a serem testados devem ser especificados.

Cronograma de teste.

Um cronograma completo de teste e alocação de recursos para esse cronograma, o qual está ligado ao cronograma mais geral de desenvolvimento de projeto.

Procedimentos para registrar testes.

Não é suficiente simplesmente executar testes. Seus resultados devem ser registrados sistematicamente, de modo que seja possível auditar o processo de teste para averiguar se foi realizado corretamente.

Requisitos de hardware e software.

Deve especificar a utilização de ferramentas de hardware e de software requeridas.

Restrições.

Restrições que afetam o processo de software (tal como, escassez de equipe) devem ser antecipadas.

Figura 32. Conteúdo do plano de teste (SOMMERVILLE, 1996)

O plano de teste deve incluir quantidades significativas de contingências, de modo que, deslizes no projeto e na implementação possam ser acomodados e, a equipe de teste possa ser re-alocada em outras atividades. O plano de teste não é um documento estático, deve ser revisado regularmente até que esteja concluído o processo de teste. Sua preparação deve iniciar quando da formulação dos requisitos e, desenvolvido em detalhes, assim que o software é projetado. A Figura 33, mostra a relação entre planos de teste e as atividades do processo de software.

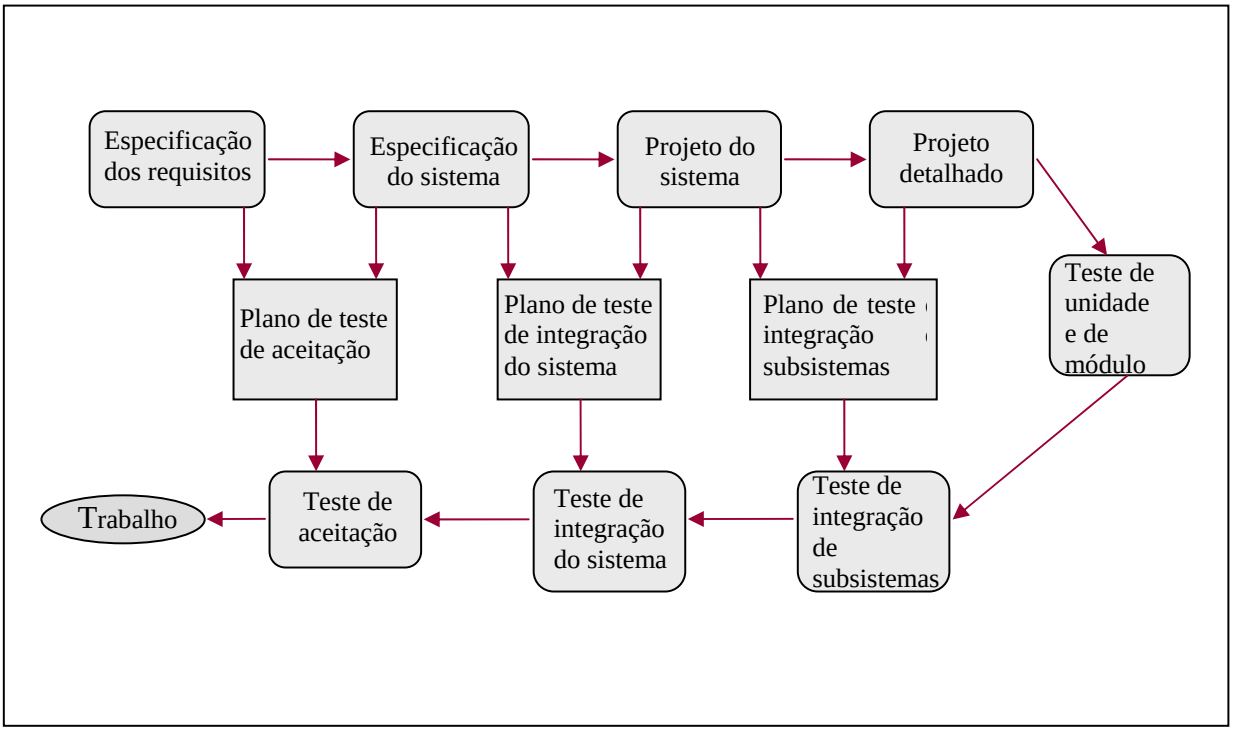


Figura 33. Fases de teste no processo de desenvolvimento de software Sommerville (1996)

A Figura 33, de acordo com Sommerville (1996), é uma versão horizontal do que algumas vezes é chamado modelo V, do processo de software. Esse modelo é uma extensão do modelo de ciclo de vida clássico, onde cada fase do processo relacionado com o desenvolvimento tem uma fase de verificação e validação associada e os planos de teste são as ligações entre essas atividades.

A atividade de teste de software combina uma estratégia de múltiplos passos com uma série de métodos de projeto de casos de testes que ajudam a garantir que defeitos sejam detectados efetivamente (FERRARI, 1998).

4.3.2 Estratégias de teste

Teste de software envolve um conjunto de atividades que pode ser planejado antecipadamente e realizado de forma sistemática. Desse modo, deve-se definir um conjunto no qual seja possível alocar técnicas de projeto de casos de teste e métodos de teste específicos, para o processo de engenharia de software.

Uma estratégia de teste de software integra técnicas de projeto de casos de teste numa série bem definida de passos que resultam na construção bem sucedida de um produto de software Pressman (1995).

Uma estratégia de teste de software proporciona o mapeamento das atividades a serem seguidas durante o processo de teste, que permitam a organização de garantia da qualidade de software e que possibilitem o adequado acompanhamento do cliente. Assim sendo, deve envolver o planejamento de teste, o projeto de casos de teste, a execução de casos de teste e a avaliação dos resultados obtidos.

O estabelecimento de estratégias, de métodos e de critérios de teste visa a fornecer uma maneira sistemática para selecionar um subconjunto relativamente pequeno, do domínio de entrada – um conjunto de casos de teste T – e, ainda assim, ser efetivo quanto à meta principal das atividades de teste: revelar a presença de defeitos no programa (MALDONADO, 1991).

Uma estratégia de teste de software deve ser flexível o bastante para promover, a criatividade e a adequação necessárias para testar todos os grandes sistemas baseados em software. Ao mesmo tempo, a estratégia deve ser rígida o suficiente para promover razoável planejamento e rastreamento administrativo, à medida que o projeto progride (PRESSMAN, 1995).

4.3.3 Casos de teste

O teste de software tem como objetivo principal revelar defeitos. Bons casos de teste têm alta probabilidade de revelar um erro ainda não descoberto. Executar bons casos de teste aumenta a confiança de que o programa está de acordo com a especificação (VERGÍLIO, 1997).

Um aspecto importante da atividade de teste envolve decidir sobre quão bem uma série de entradas de teste testa uma parte de código. Normalmente, o objetivo é descobrir tantas falhas quanto possíveis, com um poderoso conjunto de testes, dado que uma série de testes que tem potencial para descobrir mais defeitos é obviamente melhor do que uma outra que pode descobrir apenas uns poucos defeitos (MICHAEL, 2001).

Aspecto importante a ser considerado na atividade de teste é o projeto de casos de teste, o qual se concentra num conjunto de técnicas para a construção de casos de teste, que atendam aos objetivos globais da atividade de teste.

Atividade das mais relevantes e pertinentes ao teste - o projeto de casos de teste - concentra-se em um conjunto de técnicas, critérios e métodos para elaborar casos de teste, que fornecem ao projetista de software uma abordagem sistemática e teoricamente fundamentada, de acordo com Pailo (1997).

O projeto de casos de teste requer tanto rigor quanto o projeto do software a ser construído. Por várias razões, muitos engenheiros de software não dão a devida importância à atividade de teste; procuram desenvolver casos de teste que parecem certos, mas que apresentam pouca garantia de estarem completos. Essa atitude contraria um dos objetivos da atividade de teste, que é o de se projetar casos de teste que tenham maior probabilidade de descobrir a maioria dos erros, no menor tempo e esforço possíveis.

Casos de teste são selecionados a partir de um espaço de entrada D(E) do programa, de acordo, com alguma estratégia de teste. O tamanho do conjunto de teste normalmente é muito menor em comparação ao tamanho de D(E) e, sempre existe o problema e o desafio quanto ao teste de software relacionado à seleção de casos de teste que sejam confiáveis quanto à fidedignidade do programa a partir de uma amostra pequena. A maioria das estratégias de teste usa, alguma forma de informação sobre o programa ou sua especificação para gerar casos de teste.

Durante a atividade de teste, os critérios de teste escolhidos estabelecem o que será requerido no teste. No caso de critérios estruturais, são requeridos os componentes estruturais do programa. Os critérios selecionam esses componentes, a partir do programa em teste. Requerer um elemento significa “exigir” o teste desse elemento. No critério “todos os nós”, os componentes (elementos requeridos) são os nós (comandos) do grafo; no critério “todos potenciais usos”, os componentes são associações definição-potencial uso existentes no programa. Exercitar ou cobrir todos os elementos requeridos significa satisfazer esses critérios.

A cobertura é uma medida definida, tendo como base o programa a ser testado, os critérios utilizados e os dados de teste. Assim, cobertura de casos de teste pode ser definida como a relação entre o número de elementos executados pelo número de elementos requeridos.

Se a atividade de teste for bem conduzida, maiores serão as chances de os defeitos serem descobertos no software. A detecção de defeitos durante a atividade de teste contribui para reduzir o custo dos passos subseqüentes. Um dos benefícios secundários da atividade de teste é que a mesma aumenta a confiança de que as funções de software trabalham de acordo com as especificações e que os requisitos de desempenho sejam cumpridos. Além disso, dados resultantes da atividade de teste proporcionam boa indicação da confiabilidade do software e alguma indicação da qualidade do software como um todo (FERRARI, 1998).

Durante uma atividade de teste, todos os resultados são avaliados, comparando-se os resultados obtidos com os resultados esperados. À medida que os resultados da atividade de

teste são reunidos e avaliados, passa a existir uma indicação qualitativa da qualidade e da confiabilidade do software.

Se erros graves, que exijam modificação de projeto, forem encontrados com regularidade, a qualidade e a confiabilidade do software são suspeitas e testes adicionais são indicados. Se, por outro lado, as funções do software funcionarem adequadamente e os erros encontrados forem facilmente corrigíveis, pode-se, de acordo com (PRESSMAN, 1995), tirar uma entre duas conclusões:

- A qualidade e a confiabilidade do software são aceitáveis.
- Os testes são inadequados para revelar erros graves. Finalmente, se atividade de teste não revelar nenhum erro, pode haver dúvida de que a atividade de teste não foi suficientemente elaborada e que erros estão escondidos no software.

Os defeitos que não são revelados durante a atividade de teste, serão descobertos pelos usuários e corrigidos pelo desenvolvedor durante a fase de manutenção, onde o custo por correção será muito maior que o custo por correção durante a fase de desenvolvimento. Isso significa que os erros que não são descobertos na fase de teste serão, certamente, descobertos pelo usuário após o produto lhe ter sido entregue.

A grande vantagem dos métodos de projeto de casos de teste de software é o fato de oferecerem uma abordagem sistemática ao teste, além de oferecerem um mecanismo que ajuda a garantir a integridade do teste e proporcionar probabilidade maior de revelar erros de software.

Existem algumas técnicas de teste, sendo que as mais conhecidas são: a abordagem de teste denominada teste de caixa preta ou técnica de teste funcional e, a abordagem denominada teste de caixa branca ou técnica de teste estrutural (critérios baseados no fluxo de controle e critérios baseados no fluxo de dados).

A técnica de teste funcional aborda o software de um ponto de vista macroscópico, baseia-se principalmente na especificação do software para derivar os requisitos de teste.

O teste funcional possibilita que o engenheiro de software derive conjuntos de condições de entrada que exercitem completamente todos os requisitos funcionais para determinada aplicação. O teste funcional não é uma alternativa para a técnica de teste estrutural. É considerada uma abordagem complementar que tem a probabilidade de descobrir uma classe de erros diferente da classe de erros da técnica estrutural e, ao contrário do teste estrutural, o teste funcional tende a ser aplicado durante as últimas etapas da atividade de teste (FERRARI, 1998).

O teste estrutural é uma técnica de projeto de casos de teste, que usa a estrutura de controle do projeto procedimental para derivar casos de teste.

A técnica estrutural de teste utiliza principalmente informações oriundas do texto do programa. Baseia-se principalmente na especificação do software para derivar os requisitos de teste.

As técnicas de teste estrutural e funcional não são excludentes, ao contrário, são complementares, dado que cada uma revela distintas classes de erros.

São muitos os tipos de testes existentes e que podem ser utilizados para assegurar a qualidade do produto de software, entretanto, como não é objetivo desta proposta o detalhamento de tais testes, dado que os mesmos devem ser considerados no contexto da proposta em si e não como objetos específicos desta proposta, procurou-se dar uma visão mais abrangente sobre o assunto sem, contudo, minorar sua importância para a área de engenharia de software, nem para o trabalho em questão.

No que concerne à qualidade de processo, o teste sistemático é uma atividade essencial para se atingir o nível três - (*Software Product Engineering*) - do CMM, segundo Ferrari (1998). Aspecto importante e, por isso mesmo, considerado na proposta deste trabalho.

Para Smith (2003), usando uma combinação de revisões e teste, o progresso ou desvio de um projeto, a partir de uma linha básica planejada, pode ser cuidadosamente seguido e até mesmo previsto.

5. REUSO DE SOFTWARE

O mercado de tecnologia de informação é caracterizado por rápidas mudanças, causadas pela evolução da tecnologia e pelas dificuldades em transferir novas tecnologias para os produtos e para os processos. Por outro lado, a indústria de software vem enfrentando e continuamente demanda aplicações novas e mais sofisticadas, expressão direta de novas expectativas dos clientes relacionados a aplicações de software.

Há muito, a comunidade de engenharia de software vem formando uma consciência de que, para a obtenção de produtos com alta qualidade e que sejam economicamente viáveis torna-se necessário um conjunto sistemático de processos, técnicas e ferramentas (LAZILHA, 2002). Entre as técnicas mais relevantes desse conjunto, está a reutilização. Parte-se do princípio que a reutilização de partes bem especificadas, desenvolvidas e testadas, possibilita a construção de software com maior confiabilidade e em menos tempo, segundo Gimenes (2002). Muitas técnicas que favorecem o reuso, têm sido propostas ao longo dos anos. Entre estas estão: engenharia de domínio, *frameworks*, padrões, arquitetura de software e desenvolvimento baseado em componentes. Entretanto, o que falta nesse contexto é uma maneira sistemática e previsível de se realizar a reutilização. Desse modo, o enfoque de linha de produto de software surge como uma proposta de construção sistemática de software no contexto de fábrica de software.

Essa consciência tem suscitado maior interesse e empenho de pesquisadores no sentido de tentar difundir abordagens que primem pela compreensão de que o processo de construção de software, em muitos aspectos, é similar ao processo de construção de qualquer outro produto cujo uso é destinado a terceiros. É importante entender que o software e o processo que possibilita a sua construção constituem ativos importantes para a organização, ou seja, a construção de software deve ser vista como um item de grande valor econômico, uma vez que muitos custos estão envolvidos na sua produção, manutenção e distribuição. Atualmente, um software que oferece maior nível de qualidade e confiabilidade é um grande diferencial em um mercado cuja demanda por produtos de qualidade está cada vez mais exigente. Por

conseguinte, a qualidade é uma característica que deve ser planejada, projetada e não uma preocupação a ser levada a efeito apenas quando o produto já estiver construído.

Para Pressman (1997), uma das tarefas de planejamento de software diz respeito à estimativa de recursos exigidos para executar o esforço de desenvolvimento de software. A Figura 34 ilustra tais recursos de desenvolvimento.

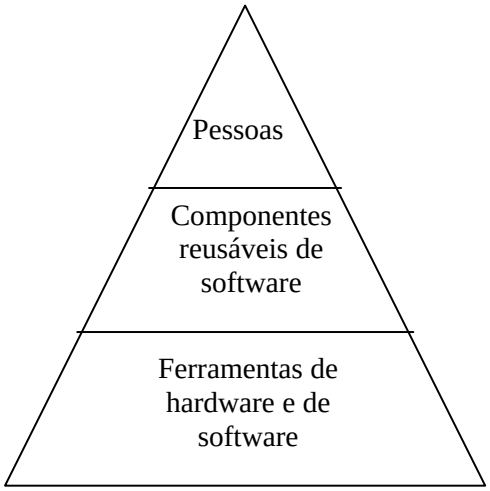


Figura 34. Recursos para o desenvolvimento de software

No contexto de engenharia de software, reuso significa tanto uma velha como uma nova idéia (PRESSMAN, 1997). Programadores têm reusado suas idéias, objetos, argumentos, abstrações e processos desde os primórdios da computação, mas essa abordagem de reuso tem sido “ad hoc”. Atualmente, sistemas computacionais complexos e de alta qualidade devem ser construídos em períodos de tempo cada vez menores. Isso demanda uma abordagem de reuso mais organizada.

O ambiente de desenvolvimento – ferramentas de hardware e de software – compõe a fundação dos recursos e fornece uma infra-estrutura para apoiar o esforço de desenvolvimento. Em um nível mais alto, estão os componentes – blocos de construção de software que podem reduzir drasticamente os custos de desenvolvimento e acelerar o prazo de

entrega. No topo da pirâmide está o recurso principal: as pessoas. Cada recurso é especificado com quatro características: descrição do recurso; uma declaração de disponibilidade; tempo cronológico no qual o recurso será exigido e; duração de tempo no qual o recurso será aplicado.

Para Pressman (1997), qualquer discussão sobre recursos de software seria incompleta sem o reconhecimento de reusabilidade, isto é, a criação e reuso de blocos de construção de software. Tais blocos de construção de software devem ser catalogados para facilitar a referência, padronizados para facilitar a aplicação e validados para facilitar a integração.

Na visão de Morisio (2000), reuso de software ocorre de duas formas: **reuso "ad hoc" ou oportunista,** acontece por sorte (oportunidade) devido somente à boa vontade de indivíduos. **Reuso sistemático**, resulta do planejamento, execução e monitoração das mudanças nos processos ao nível organizacional para assegurar reuso regular.

Reuso promete certas vantagens, principalmente porque um novo software não necessita ser escrito. Ciclos de tempo são reduzidos porque poucos artefatos precisam ser desenvolvidos; melhora a qualidade porque os artefatos já produzidos estão livres de defeito e; a combinação de ambos reduz os custos. Além disso, somente um produto deve ser mantido ao invés de vários.

Embora o conceito seja simples, a implementação não é. Alcançar reuso de um produto de trabalho requer satisfazer pelo menos três condições (MORISIO, 2000):

- **Reusabilidade funcional**. O produto de trabalho deve oferecer a funcionalidade necessária ao projeto.
- **Reusabilidade técnica**. O produto de trabalho deve ser de fácil integração ao ambiente de operação do projeto (tal como, um banco de dados).
- **Qualidade**. O nível de qualidade deve atender (satisfazer) os requisitos de projeto.

Para assegurar as duas primeiras condições, as organizações devem estabelecer processos e funções (papéis) para identificar a provável funcionalidade necessária aos projetos futuros e capturá-la nos artefatos reusáveis isolados, tanto quanto possível, a partir do ambiente operacional. Finalmente, para atender a terceira condição, as organizações necessitam processos que qualifiquem e documentem artefatos.

De acordo com Pressman (1997), quando artefatos reusáveis são utilizados durante o processo de software, menos tempo é gasto na produção de planos, modelos, documentos, códigos e dados que são exigidos para construir um sistema. Portanto, o mesmo nível de funcionalidade é entregue ao usuário com menos esforço. Por isso, a produtividade é melhorada.

O interesse pelo processo de software e o seu desenvolvimento dinâmico têm crescido nas últimas décadas, como resposta à necessidade de aumentar a produtividade e a qualidade, ao mesmo tempo reduzindo os prazos de entrega e, levando em consideração, aspectos de manutenção. Todavia, não se tem alcançado os resultados esperados e projetos de software ainda ultrapassam orçamentos e cronogramas estimados ou, no pior caso, são interrompidos ou cancelados (VALÉRIO, 1997).

Na visão de Rine (1997), não há um conjunto de fatores de sucesso que seja comum entre as organizações e que tenha alguma relação previsível para reuso de software. Entretanto, muitas organizações de desenvolvimento de software acreditam que o reuso de software irá melhorar sua produtividade e qualidade e estão em processo de planejamento para desenvolver ou já estão desenvolvendo uma capacidade de reuso de software.

Para Valério (1997), as empresas de software padecem de um mal crônico que resulta da falta de habilidade para responder adequadamente à demanda do mercado. À adaptação natural para a evolução e às rápidas mudanças que ambos, o ambiente externo e as condições internas estão sendo submetidos.

Colocar uma abordagem de reuso e tecnologia em prática, requer que uma organização projete um processo adequado para realizá-la. Em geral, uma organização deve definir novas

funções de responsabilidades e reuso, adicionar reuso a processos, modificar não-reuso em processos e instalar ferramentas – tipicamente um repositório para artefatos reusáveis. Funções e processos clarificam quando e quem desenvolve artefatos reusáveis para a organização (MORISIO, 2000). Sugere ainda, que as organizações devem:

- Avaliar o seu potencial de reuso. Verificando quantos produtos de software ou projetos similares são produzidos em horas extras. Um contexto de negócio estável e bem definido é, normalmente, um requisito para potencial de reuso.
- Se o potencial de reuso é alto, um programa de reuso pode ser iniciado e a capacidade de reuso determinada é alcançada. Inicialmente, deve-se conseguir um compromisso da gerência para adquirir recursos e obter poder para:
  - Alterar não-reuso nos processos. Definição de requisitos e análise, projeto (*design*) de alto nível e teste requerem mudanças específicas que levam em conta a avaliação de disponibilidade. Gerenciamento de projeto também é impactado, especialmente com relação a cronograma, custo e produtividade.
  - Adicionar reuso aos processos. Análise de domínio pode conduzir à identificação de artefatos reusáveis. Artefatos podem ser pequenos ou grandes, incluindo projeto (*design*) e requisitos. Podem ser desenvolvidos a partir do zero ou reconstituídos a partir de legado. Podem ser produzidos e mantidos por um grupo específico ou por projetos.
  - Contemplar fatores humanos. Uma ou mais técnicas (tal como treinamento, anúncios de eventos, grupos de discussão e informativos) podem ser usadas. Incentivos apenas não são suficientes.
  - Construir um repositório e suportá-lo com uma ferramenta. Uma ferramenta específica adicionada ao sistema de gerenciamento de configuração ou o plano de sistema de gerenciamento de configuração, são possibilidades.

Ainda segundo Morisio (2000), adicionar reuso a processos, normalmente implica definir e determinar funções chave de reuso. Deve-se estar seguro para verificar o domínio (posse) de processos e requisitos. Mudar não-reuso em processos e adicionar reuso a processos será muito mais difícil quando o domínio desses processos está situado em outro lugar – isto é, quando sub-contrato está envolvido. A disponibilidade de recursos na organização, normalmente relacionada ao seu tamanho, deve ser cuidadosamente considerada para tornar escolhas sustentáveis.

Devido ao fato de o reuso oferecer vantagem competitiva, há pouco incentivo para compartilhar experiências entre as organizações. Há poucas pesquisas para determinar se os fatores de sucesso de reuso de software de uma organização em particular são aplicáveis a outras organizações. Tal pesquisa é importante para fornecer uma diretriz para as organizações planejarem ou implementarem um programa de reuso. Isso significa que a grande maioria das informações correntes são provenientes da literatura, a qual contém teorias não provadas ou teorias aplicadas de um modo limitado a poucos projetos piloto ou estudos de caso dentro de domínios de aplicação particulares.

Em razão do exposto acima, Rine (1997), afirma que não há um conjunto de fatores de sucesso que seja comum entre as organizações e que tenham alguma relação previsível para reuso de software.

Morisio (2000) enfatiza que a engenharia de software deve aplicar processos definidos e repetíveis para alcançar resultados previsíveis. Contudo, os processos variam de organização para organização e de projeto para projeto, assim, uma abordagem claramente pode não servir a todas as organizações. A despeito da diversidade no contexto de negócios - tradição técnica e gerencial e tamanho - as organizações podem realmente alcançar reuso, usando processos diversos.

Reuso de software significa que artefatos desenvolvidos em outro lugar – em outro projeto, grupo ou organização – são usados novamente. Tipicamente, o produto de trabalho é código, mas reuso pode ser efetivo para um nível mais alto, considerando os requisitos, projeto (*design*), testes e todos os artefatos originados durante o processo de desenvolvimento (MORISIO, 2000).

Paralelo às atividades acerca de um processo de software formalizado e controlado, diferentes metodologias e técnicas têm sido propostas com o intuito de resolver problemas específicos e para melhorar a atividade de desenvolvimento. Reuso de software é uma das técnicas mais interessantes uma vez que promete alcançar uma redução nos prazos e nos custos de investimento e ao mesmo tempo, incrementar a qualidade e a manutenibilidade do produto.

Entretanto, essas possibilidades ainda não estão plena e amplamente exploradas pelas empresas de software. A análise de domínio - processo no qual a informação no ciclo de vida do software, é identificada e agrupada para tornar-se reusável - é outra técnica para melhorar o processo de software (VALÉRIO, 1997).

Reuso de software, o uso de artefatos existentes de software ou conhecimento para criar novos softwares, é um método chave para melhorar significativamente qualidade e produtividade de software. Para Frakes (1996), as organizações que implementam programas de reuso de software sistemáticos devem ser capazes de medir seu progresso e identificar as estratégias de reuso mais efetivas. Reuso não se aplica somente a fragmentos de código-fonte, mas a todos os produtos intermediários gerados durante o processo de desenvolvimento de software, incluindo requisitos, documentos, especificações de sistemas e qualquer informação de que o desenvolvedor necessite para construir software. Isto significa que os desenvolvedores podem buscar o reuso de documento de requisitos, especificações de sistemas, estruturas de projeto e qualquer outro artefato de desenvolvimento.

Jones (1991) identifica dez aspectos potencialmente reusáveis de projetos de software, conforme apresentado na Tabela 14.

Tabela 14. Aspectos reutilizáveis de projetos de software (JONES, 1991)

01. Arquiteturas	06. Estimativas (*templates*)
02. Código fonte	07. Interfaces humanas
03. Dados	08. Planos
04. Projetos (*designs*)	09. Requisitos
05. Documentação	10. Casos de teste

Reusabilidade é o grau para o qual uma coisa pode ser reusada. Para alcançar resultados finais significantes, um programa de reuso deve ser sistemático, (FRAKES, 1996).

Reuso não deve acontecer por acidente, deve ser uma atividade planejada: planejar o reuso de componentes antigos e buscar novos componentes reusáveis. As chaves para o reuso bem sucedido de software são, atitudes, ferramentas e técnicas. Atitude está relacionada à importância que se dá à criação de abstrações reusáveis. Ferramentas, que possam apoiar o processo de construção de artefatos reusáveis e técnicas que ofereçam um conjunto sistemático por meio do qual seja possível a construção de elementos reusáveis dentro dos padrões estabelecidos e com o nível de qualidade exigido (JONES, 1991).

No entendimento de Pressman (1997), reuso deveria ser uma parte integrante de todo o processo de software, dado que a atividade de reuso se estende além de artefatos liberáveis, inclui também métodos de modelagem de análise específicos, técnicas de inspeção, técnicas de projeto de casos de teste, procedimentos para garantia da qualidade e, muitas outras práticas de engenharia de software que podem ser reusadas.

Na concepção de Jan (2000), reuso de software tem sido um propósito duradouro da comunidade de engenharia de software. O problema mais difícil com relação a reuso ao nível de grandes componentes que podem compor a parte maior de um sistema e, da qual muitos aspectos podem ser adaptados, não foram contemplados pelo paradigma de orientação a objetos. Esse entendimento levou ao desenvolvimento de *frameworks* orientados a objetos, isto é, aplicações amplas e abstratas em um domínio particular que podem ser adaptadas para aplicações concretas. A idéia de programação orientada a componentes foi desenvolvida como uma abordagem para o reuso de software.

Para Jones (1991), o incentivo econômico para se reutilizar software é o mesmo destinado ao uso de peças padronizadas no projeto de outros produtos industriais, porque as peças padrão são mais baratas, mais confiáveis e normalmente mais fáceis de serem consertadas ou substituídas em caso de falhas e/ou defeitos.

Para Butler (1997), o reuso de software amortiza o custo de desenvolver um componente, entre muitos projetos e suporta melhor a qualidade de produtos devido aos esforços em inspeções e teste.

Desde que a atividade de reuso teve início, grande parte dos programadores e projetistas de software tem espontaneamente reutilizado elementos lógicos, assim como módulos e partes de projetos. Entretanto, a reutilização espontânea e com fins específicos, ainda está muito distante de conseguir a influência econômica que poderia ser possível, caso 50% a 80% das aplicações pudessem ser construídas com componentes reutilizáveis (JONES, 1991).

Conforme explicitado em (WEINBERG, 1996), quando o código usado em um lugar pode ser reutilizado em outro, as economias podem ser enormes, assim como, o aumento da confiabilidade e o decréscimo da variabilidade. Uma vez que se constrói uma coleção de módulos úteis, confiáveis e reutilizáveis, é possível baixar o custo, elevar a qualidade e melhorar a previsibilidade do desenvolvimento de software.

De acordo com Jan (2000), uma ampla classe de artefatos de reuso tornou-se disponível e é usada extensivamente pela indústria de software. Exemplos tradicionais incluem sistemas operacionais, sistemas de gerenciamento de banco de dados e compiladores, exemplos mais recentes incluem interfaces gráficas, servidores e navegadores para *Web*. Embora muitos engenheiros de software não percebam o uso desses artefatos de software como reuso, para cada um desses exemplos, houve um tempo quando a funcionalidade capturada pelo componente era parte do código da aplicação, ao invés de construída como um componente separado.

A razão pela qual, os engenheiros de software não consideram o uso desses componentes como reuso de software é que os mesmos se tornaram parte da infra-estrutura presente quando do desenvolvimento de produtos de software. É possível identificar o ciclo de vida de tais componentes. Primeiro, a funcionalidade é somente parte do código da aplicação de um produto de software e, em algum ponto, a funcionalidade é identificada como uma entidade conceitual e começa a ser modelada como um subsistema no código da aplicação. O terceiro estágio é caracterizado pelos pesquisadores (empresas, universidades) que começam a desenvolver protótipos de sistemas os quais generalizam a funcionalidade capturada em um número de subsistemas. Quarto, as empresas comerciais, normalmente começam identificando uma oportunidade de mercado para converter o protótipo em um produto. Na quinta fase, o produto atende a ampla aceitação de mercado e é usado pela maioria dos

engenheiros de software como parte do desenvolvimento de software. A fase final é onde o produto é incorporado em uma infra-estrutura provida por terceiros. Nesse ponto, o produto desaparece como uma parte independente da aplicação de software (JAN, 2000).

Conforme observado em um estudo de Jones (1984), o reuso envolve cinco subsistemas importantes que devem ser avaliados:

- Dados reutilizáveis ou técnicas para padronizar o fluxo de informação de módulo e de aplicação a aplicação.
- Arquitetura reutilizável ou planejar a inclusão de peças reutilizadas como passo fundamental no projeto de um programa e de um sistema a partir das especificações (das especificações em diante).
- Projetos reutilizáveis ou a compactação de ótimas estruturas esquemáticas de programas de aplicação genérica, mais importantes, como os pacotes de programa de processamento de palavras e pacotes de programas contábeis.
- Programas reutilizáveis ou a criação de software comercial que seja útil em uma ampla variedade de computadores e, além disso, a adoção de um intercâmbio de dados padrão de forma que qualquer programa possa transmitir informação a outros programas quando necessário. A invocação e as conseqüências de controles padrão, também necessidades dos programas reutilizáveis.
- Módulos reutilizáveis de elementos funcionais padrão, que podem ser interligados para criar novos programas e sistemas. Aqui também existe a necessidade de um intercâmbio de dados padrão e invocação e seqüência de controle.

Além desses cinco temas primários, para Jones (1984), a utilização de reuso tem uma série de considerações secundárias, dentre as quais, as mais importantes são:

- Biblioteca de apoio de catálogos para um repertório de funções e módulos reutilizáveis.
- Biblioteca e catálogos de apoio para o usuário e para a documentação de manutenção que acompanha os módulos e funções padrão, de modo que tal informação não necessite ser recriada toda vez que uma função padrão é incluída no programa.

- Indicação e certificação de módulos padrão reutilizáveis, para garantir que os mesmos atendem a rigorosos critérios de confiabilidade e qualidade. Elaboração de uma nova classe de métodos de especificação e de projeto que inclua a capacidade para especificar elementos padrão de sistema, bem como funções novas e únicas que requeiram codificação manual, isto é, métodos de projeto que tenham verbos explícitos baseados em funções padrão.
- A linguagem, a ferramenta e o apoio ambiental ao código reutilizável e as funções padrão para avaliar a adoção em larga escala de tais métodos dentro das empresas;
- Migração seletiva de funções padrão que são usadas com elevada freqüência do software ao VLSI[30] ou micro-código, para aumentar o desempenho das mesmas.
- Arquiteturas e computadores capazes de manipular níveis elevados de paralelismo e concorrência, uma vez que as funções padrão são intrinsecamente capazes de execução paralela, se as máquinas sobre as quais elas executam puderem apoiar paralelismo.
- Aumento das técnicas convencionais de programação, com uma nova geração de geradores de sistemas e de aplicações que operem selecionando e combinando módulos que satisfaçam as necessidades da aplicação.
- Mudanças contábeis e financeiras internas às organizações que adotam funções padrão e códigos reutilizáveis, de modo que o criador inicial de um módulo padrão tenha como receber alguma forma de compensação, enquanto os usuários das funções padrão efetuarem algum tipo de pagamento.

Para Kontio (1995), o reuso tem sido considerado uma importante solução para muitos problemas no desenvolvimento de software.

De acordo com Jones (1991) o software reutilizável e os projetos padrão apresentam interessantes desafios, tanto técnicos como comerciais. Alguns dos maiores problemas enfrentados com respeito a código reutilizável e funções padrão nas organizações são financeiros:

[30] Do inglês: *Very Large Scale of Integration*.

- Como pagar os custos gerais da administração de uma biblioteca de código reutilizável compartilhada que pode ser utilizada por muitos projetos.
- Como compensar os responsáveis pelo desenvolvimento das funções padrão, de forma que tenham algum incentivo para criar prioritariamente módulos reutilizáveis.
- Como mudar os métodos internos de rateio de custos, de modo que o código reutilizável possa ser comprado pelos projetos e, ainda assim, ser econômico ao uso.

Um dos argumentos contrários ao software reutilizável, na concepção de Weinberg (1996) é o fato de este vir pronto para ser reutilizado. Pois não se pode reutilizar um software antigo que tenha sido simplesmente agrupado, é necessário fazer um investimento adicional para que o software seja verdadeiramente reutilizável e, quanto maior for o investimento, mais vezes o software deve ser reutilizado para que seja possível recuperar o investimento.

Conforme o contido em Butler (1997), existem três diferentes escopos de reuso:

- Reuso geral. É independente do domínio da aplicação. Reusa quase que inteiramente componentes como estruturas de dados comuns e interfaces gráficas com o usuário, que são encontrados na maioria das aplicações.
- Reuso dentro de um domínio de aplicação. Restringe seu foco para um único domínio. É importante para especificar os aspectos comuns de sistemas de aplicação para determinado domínio e também para especificar a gama de variabilidade encontrada no domínio.
- Reuso dentro de uma família de produto. Enfoca uma linha de produto dentro de um domínio de aplicação. Uma arquitetura genérica é desenvolvida para esta família de produto, de modo que há uma função bem definida para cada componente.

O reuso pode ser realizado dentro de uma organização ou pode ser inter-organizacional por meio de *marketing* de componentes reusáveis. Atualmente, um pequeno número de empresas vende bibliotecas gerais de software, mas é um negócio difícil. Há reuso de padrões em

poucas indústrias, tais como aquelas que utilizam interfaces, protocolos ou formato de intercâmbio de dados. Por outro lado, o reuso dentro de uma organização tem sido proveitoso em cada um dos três escopos. Reuso é mais bem sucedido quando é sistemático e foca um domínio de aplicação restrito ou uma família de produto. Reuso sistemático, ao longo do tempo, impulsiona com sucesso a melhoria na produtividade e qualidade em uma organização (BUTLER, 1997).

Para Rine (1997), uma abordagem bem sucedida de linha de produto para desenvolvimento e comercialização de software envolve fatores técnicos e não técnicos. Tal abordagem envolve o reuso de artefatos de software e práticas e experiências de alta qualidade. Muitos relatórios sobre reuso de software se concentram somente nos fatores técnicos de reuso e muitos relatórios nunca falam da necessidade de reuso de experiências e práticas de alta qualidade. Uma vez identificado o reuso de práticas de alta qualidade, o mesmo pode ser modelado com padrões, diagramas de fluxo de dados, cenários e macros ao nível de modelo, em adição a sistemas especialistas que sejam aptos para formular modelos de processos de software repetível e de alta qualidade em concordância com o nível - 3 (três) ou superiores - do CMM. Todavia, poucas organizações que utilizam reuso bem sucedido têm formalmente buscado concordância com o modelo CMM devido ao alto custo envolvendo o processo de certificação.

Reuso que foca uma linha de produto dentro de um domínio de aplicação pode alcançar alto percentual de reuso em cada fase do desenvolvimento de software. Análise de domínio pode identificar características arquiteturais comuns e pode desenvolver uma linguagem para descrever aspectos variáveis da família de produto. Uma arquitetura genérica é desenvolvida para tal família de produto, de modo que haja uma função bem definida para cada componente. A arquitetura e sua compreensão têm varias denominações: arquitetura de referência, arquitetura de domínio específico de software ou *framework* de aplicação (BUTLER, 1997). A função de um componente dentro do *framework* pode também fazer parte da taxonomia do domínio e ser usada para classificação e verificação.

O reuso de software, segundo Weinberg (1996), pode ser considerado em diferentes níveis:

- Reuso de sistemas de aplicação. A totalidade de um sistema de aplicação pode ser reusada. O problema chave nesse caso é assegurar que o software é portável, que pode ser executado em várias plataformas diferentes.
- Reuso de subsistema. Os subsistemas principais podem ser reusados.
- Reuso de objeto ou módulo. Componentes de um sistema representando uma coleção de funções podem ser reusados.
- Reuso de função. Componentes de software que implementam uma única função (tal como uma função matemática) podem ser reusados.

Para Henninger (1997), duas tecnologias básicas têm sido utilizadas nas metodologias de reuso de software:

- Reuso baseado em composição, também conhecido reuso como baseado em partes ou baseado em componentes.
- Técnicas generativas (reuso baseado em linguagens).

Nas tecnologias de composição, os componentes são entidades auto-contidas tais como, rotinas de bibliotecas, estruturas de dados, programas, objetos e outros mais. Esses componentes de reuso são examinados pelos desenvolvedores para encontrar e adaptar os componentes em uma nova aplicação. *Pipes* Unix e biblioteca de código para linguagens de programação são exemplos de técnica de composição. As abordagens generativas estão estreitamente relacionadas à tecnologia de compilador, onde partes de código parametrizado são invocadas por seleção do usuário para criar aplicações adaptadas (HENNINGER, 1997).

Na seção a seguir será descrita a abordagem de reuso baseada em componentes, enfocando duas alternativas para reuso. O desenvolvimento de software voltado ao uso de componentes existentes e o desenvolvimento de software voltado à construção de componentes visando ao reuso.

5.1 DESENVOLVIMENTO DE SOFTWARE BASEADO EM COMPONENTES

Embora, o conceito de reuso de componente de software tenha sido introduzido em 1968, não recebeu muita atenção da indústria e dos meios acadêmicos até o final da década de 80. Depois da metade da década de 80, o rápido aumento da complexidade nos sistemas de aplicação de software forçou os vendedores de software a atentar para os métodos econômicos para construir complicados sistemas de software em um curto ciclo de vida de desenvolvimento para atender as necessidades do mercado. Reuso de software baseado em componentes de alta qualidade, contudo, tornou-se popular nos meios acadêmicos e na indústria de software devido à vantagem de reduzir custos e tempo de desenvolvimento (GAO, 2003).

Dado que os componentes de software são partes construídas visando à construção de sistemas de software, criar componentes de software altamente reusáveis é de vital importância para o sucesso do projeto de desenvolvimento de software. Para Gao (2003), construir um sistema baseado em componentes reusáveis não é uma idéia nova, uma vez que nas duas décadas passadas, diversos especialistas envidaram esforços para estudar reuso de componentes em projetos, linhas de produtos e organizações. Esses esforços contribuíram para que o conceito de reuso de software seja amplamente aceito pela indústria de software atualmente.

O reuso de componentes de software durante o desenvolvimento de software é considerado um importante fator para melhorar a qualidade do processo, do produto e da produtividade, para diminuir os custos e para reduzir o tempo para o lançamento de um produto no mercado (SCHMIETENDORF, 2000; BUTLER, 1997; HERZUM, 2000; WANG, 2000; JAN, 2000; GAO, 2003).

A ênfase no reuso de componentes – ou reuso de porções significativas deles – pode baixar os custos de desenvolvimento e melhorar a produtividade, acelerando o tempo do ciclo de desenvolvimento de software e eliminando redundâncias (VOTH, 2004).

Para Werner (2000), não faz muito tempo, o desenvolvimento de grande parte dos produtos de software disponíveis no mercado era baseado em uma abordagem de desenvolvimento em blocos monolíticos, formados por um grande número de partes inter-relacionadas e, estes relacionamentos estavam implícitos, na maioria das vezes. O Desenvolvimento Baseado em Componentes (DBC) surgiu como uma nova perspectiva para o desenvolvimento de software, cujos objetivos incluem: quebrar esses blocos monolíticos em componentes inter-operáveis, reduzir a complexidade no desenvolvimento e reduzir os custos decorrentes da utilização de componentes que podem ser adequados a outras aplicações.

A caracterização do que seria um componente de software ainda não é um assunto fechado e acordado na comunidade de pesquisadores da área. Os distintos grupos de pesquisa adotam a característica que mais adequada ao seu contexto, sobre o que seria um componente.

Para Lazilha (2002), a técnica de desenvolvimento baseado em componentes visa fornecer um conjunto de ferramentas, técnicas e notações que possibilitem que, ao longo do processo de software, ocorra tanto a produção de novos componentes quanto a reutilização de componentes existentes.

Segundo Wang (2000), a engenharia de software baseada em componente tem como propósito a construção integrada de componentes de software pré-construídos. Alta produtividade é alcançada pelo uso de componentes padrão. Os princípios da engenharia de software baseada em componentes podem ser descritos seguindo dois princípios: reusar, mas não reinventar; reunir componentes pré-construídos ao invés de codificar linha por linha.

A abordagem de desenvolvimento de software baseado em componentes tem sido usada para desenvolver, grandes e complicados sistemas de software utilizando componentes disponíveis e reusáveis. O maior objetivo é reduzir o custo de desenvolvimento de software, incluído o tempo para reusar componentes disponíveis (GAO, 2003).

No contexto de engenharia de software baseada em componentes, os componentes são concisamente definidos como blocos básicos de construção. A granularidade desses blocos de

construção afeta a reusabilidade, a produtividade e a manutenção do componente (WANG, 2000).

Similar a um módulo reusável, um componente reusável tem cinco elementos: especificação dos requisitos do componente; interfaces; código fonte e programa executável; documentos de validação e relatórios de teste (GAO, 2003).

Segundo a visão de (D'SOUZA, 1999 apud LAZILHA, 2002), um componente pode ser definido como uma unidade de software independente, que encapsula dentro de si seu projeto e implementação e oferece interfaces bem definidas para o meio externo, para que esse componente possa se unir com outros componentes e dar origem aos sistemas baseados em componentes. As interfaces são chamadas de Interfaces Fornecidas ou Interfaces Requeridas. A interface fornecida define as operações que o componente oferece para outros componentes. A interface requerida define as operações que o componente requisitará de outros componentes. As interfaces servem apenas para a comunicação entre componentes, ocultando dos usuários os detalhes de implementação.

Bosch (2000) classifica os componentes em três tipos:

- Componentes desenvolvidos pelo próprio cliente.
- Componentes adquiridos para um determinado domínio.
- Os chamados componentes *Commercial-Of-The-Shelf* (COTS) ou componentes de prateleira, que são componentes genéricos encontrados no mercado.

No entendimento de Werner (2000), para que uma abordagem consistente de DBC seja bem sucedida, é fundamental que os conceitos que estão por trás das funcionalidades disponibilizadas pelos componentes e seus relacionamentos, sejam bem compreendidos. Outras questões também importantes são: a concretização do conhecimento em componentes reutilizáveis e o uso de um modelo que apresente o interfaceamento entre os componentes (estilo arquitetural). Além disso, para se desenvolver produtos de software baseados em componentes, é necessário o auxílio de métodos e ferramentas que facilitem a especificação dos produtos de software e garantam a sua qualidade.

Para que a reutilização de componentes seja possível, é preciso que estes sejam projetados com a finalidade de reuso. O projeto de um componente deve ser levado a efeito de modo que, além de tornar sua execução correta e eficiente, deve ser genérico para que se torne adaptável a vários propósitos (domínios) e não somente ao propósito (domínio) primário.

Ainda com relação a componentes de software, existe uma diferenciação entre a compreensão acadêmica e a industrial sobre esse conceito, conforme apresentado na Tabela 15, conforme entendimento de Bosch (2000). A visão acadêmica trata os componentes (artefatos) como entidades caixa-preta com uma interface restrita. A visão industrial trata os componentes como grandes pedaços de software, como *frameworks* orientados a objetos, com uma estrutura interna complexa e nenhum limite de encapsulamento explícito. Devido à falta de limite de encapsulamento, os engenheiros de software podem ter acesso a qualquer entidade interna do componente, incluindo as entidades privadas.

Tabela 15 – A visão acadêmica versus a visão industrial de componentes de software

Visão Acadêmica	Visão na Indústria
Os artefatos reutilizáveis são componentes caixas-pretas.	Os artefatos são grandes pedaços de *software* (às vezes com mais de 80 KLOC) com uma complexa estrutura interna e nenhum limite de encapsulamento. Ex: *frameworks* orientados a objetos.
Os artefatos possuem uma interface restrita a um ponto de acesso.	A interface do artefato é provida por meio de entidades. Não há diferença explícita entre esta e as entidades sem interface.
Os artefatos possuem poucos pontos de variação explicitamente definidos que são configurados durante a instanciação.	Variação é implementada por meio de configuração, especialização ou substituição de entidades do artefato. Às vezes podem ser requeridas múltiplas implementações (versões) de um artefato para se cobrir os requisitos de variação.
Os artefatos implementam interfaces padronizadas e podem ser comercializados no mercado.	Os artefatos são desenvolvidos principalmente nas organizações. Artefatos adquiridos externamente devem ser adaptados para alcançar os requisitos da arquitetura de linha de produto.
Foco na funcionalidade do artefato e na verificação formal da funcionalidade.	Atributos de funcionalidade e qualidade (por exemplo: desempenho, confiabilidade, tamanho do código, reusabilidade e manutenibilidade) têm igual importância.

Mesmo ao usar apenas entidades de interface, a utilização de artefatos é muito complexa devido ao tamanho do código. Variações, do ponto de vista acadêmico, estão limitadas em

número e são configuradas durante a instanciação por meio de outros componentes do tipo caixa-preta. Na prática, a variabilidade de um componente é implementada por intermédio de configuração, mas também por meio de implementação ou substituição de entidades internas do componente. Além disso, múltiplas implementações de um artefato podem estar disponíveis para tratar a variabilidade requerida. Os componentes do ponto de vista acadêmico são providos de interfaces padronizadas e são usados em componentes de mercado. Na tentativa de se fazer isto, há um enfoque na funcionalidade do componente e sua apresentação formal. Na prática, quase todos os artefatos são desenvolvidos internamente. Em casos excepcionais, um componente pode ser adquirido externamente, requerendo, neste caso, uma adaptação dos componentes internos. Adicionalmente, a qualidade dos atributos dos artefatos tem no mínimo, a mesma prioridade, quando comparada à sua funcionalidade de acordo com (LAZILHA, 2002).

Para Herzum (2000), poucas organizações têm sido capazes de elevar o desenvolvimento de software para produzir software de qualidade e efetivo quanto ao custo e de um modo repetível, usando uma abordagem baseada em componente. Atualmente há uma confluência de condutores de negócio, tecnologia, processo de desenvolvimento, padrões (*standards*) e maturidade arquitetural de modo que o desejo de produzir software industrializado, o qual tem sido o objetivo de muitas indústrias durante muito tempo, pode enfim, ser alcançado.

A manufatura efetiva de software em termos de custo não é somente uma questão de tecnologia, embora, a tecnologia seja um capacitador necessário. Tampouco é somente uma questão de metodologia ou de processo de desenvolvimento; esses são apenas uma parte do problema. Ao contrário, requer que todos os aspectos de desenvolvimento de software sejam tratados com um conjunto de conceitos coordenados e integrados, arquiteturas, metodologias etc.

Pressman (1997) defende a tese de que o reuso de componente de software deve ser suportado por um ambiente que abranja os seguintes elementos:

- Um banco de dados de componentes. Capaz de armazenar componentes de software e informações de classificação necessárias para recuperá-los.

- Um sistema de gerenciamento de biblioteca que forneça acesso ao banco de dados.
- Um sistema de recuperação de componentes de software que permita ao cliente da aplicação, recuperar componentes e serviços, a partir do servidor de bibliotecas.
- Ferramentas – CASE – que apóiam a integração de componentes reusados, a um novo projeto (*design*) ou implementação.

Cada uma dessas funções interage com ou está incorporada dentro dos limites de uma biblioteca de reuso.

Herzum (2000) propõe uma abordagem baseada em componentes completa, denominada abordagem de componentes de negócio. Segundo sua concepção, é a primeira abordagem inteiramente focada em componentes ao longo do ciclo de desenvolvimento e conjunto de pontos de vista arquitetural. Tal abordagem é amplamente baseada em uma nova holística e, talvez um conceito revolucionário em essência, denominado componentes de negócio que tratam realidades de distribuição, redução de dependência e desenvolvimento autônomos em uma única construção de software multifacetada. A abordagem inclui:

- Um *framework* que traz o pensar componente para um mundo real de sistema e que faz sentido para diferentes granularidades de componentes.
- Uma metodologia orientada a objeto para fornecer os conceitos e a reflexão requerida para lidar com os desafios reais de componentes baseados em negócio para a empresa.

Desenvolvimento baseado em componentes é uma abordagem de desenvolvimento de software onde todos os aspectos e fases do ciclo de vida de desenvolvimento, incluindo análise de requisitos, arquitetura, projeto, construção, teste, distribuição, infra-estrutura para suporte técnico e também o gerenciamento de projeto, são baseados em componentes (HERZUM, 2000).

Essa definição, no entender de Herzum (2000), declara explicitamente que o desenvolvimento baseado em componentes consiste em construir software usando reflexão baseada em componentes, para estendê-lo de modo que todo o desenvolvimento de software seja centrado em componentes. Desenvolvimento baseado em componentes é extremamente benéfico não somente quando os componentes já existentes estão disponíveis para agrupamento, mas também se esses componentes são construídos para serem partes de um projeto. Na realidade, pensar um sistema de informação em termos de componentes, mesmo que esses componentes não existam, é o melhor modo de subjugar as dificuldades de desenvolvimento de sistemas, distribuídas em larga escala atualmente.

Diferentemente dos sistemas de software tradicionais, software baseado em componentes tem, com freqüência, demonstrado a seguinte característica única: heterogeneidade, como uma das características chave do software baseado em componentes, a qual possibilita que software baseado em componente seja construído a partir de componentes que são implementados usando diferentes linguagens, executados em diferentes plataformas, mesmo em múltiplas localizações e grandes distâncias. De um lado, essa característica suporta alto grau de expansibilidade e flexibilidade, por outro lado, requer alta interoperabilidade de componentes (GAO, 2003).

Dado que a qualidade do sistema depende da qualidade do componente, qualquer componente defeituoso causa impacto em todos os sistemas construídos com tal componente.

As principais vantagens de uma abordagem baseada em componentes adotada corretamente, segundo (HERZUM, 2000), são descritas a seguir:

- O desenvolvimento baseado em componente (DBC) constrói sobre o passado. É considerada uma abordagem de fácil compreensão para manufatura de software. Abrange todos os aspectos de desenvolvimento de software, desde a concepção e construção de componentes individuais até o agrupamento efetivo quanto a custo de componentes em sistemas e federações de sistemas e a evolução desses sistemas e federações. A abordagem de desenvolvimento baseado em componentes constrói as técnicas mais bem sucedidas, princípios e boas práticas do passado, para dar um fundamento teórico e prático acerca do desenvolvimento

de software que trata da manufatura de software em larga escala e suporta sua industrialização. O desenvolvimento de software baseado em componentes não apenas retém todas as vantagens de todas as abordagens precedentes, incluindo as abordagens orientadas a objetos distribuídos, como também resolve suas limitações.

- O desenvolvimento de software baseado em componentes subjuga a complexidade de desenvolvimento. Atualmente, as organizações mais importantes e de posição mais avançadas nessa área de atividade que adotam desenvolvimento de software baseado em componentes, utilizam essa abordagem principalmente no desenvolvimento de sistema em larga escala como um meio de dominar os riscos e a complexidade de projeto, com sua abordagem centrada em arquitetura e reuso. Por exemplo, mesmo se usado somente para a atividade de modelagem de negócios complexos, desenvolvimento baseado em componente oferece excelente meio para dividir um domínio de negócio.
- O desenvolvimento baseado em componentes altera a natureza do sistema de informação. Talvez mais importante de tudo, devido às suas implicações de longo alcance é que a natureza do software – a natureza do que é uma aplicação – mude radicalmente. Os componentes tornam-se altamente visíveis em tempo de execução, e isso afeta o modo como o software é construído, agrupado, distribuído, testado, expandido, comercializado e vendido. Desenvolvimento de software baseado em componentes não é somente uma abordagem de desenvolvimento, é também uma abordagem de distribuição e isso conduz a um novo modo de vender e comprar soluções de software.
- O desenvolvimento baseado em componentes possibilita o desenvolvimento autônomo de partes. Estruturar o sistema de informação em termos de um conjunto de unidades de distribuição quase autônoma suporta novas e desafiadoras oportunidades para altos níveis de desenvolvimento, teste e distribuição concorrente. Os desafios surgem das várias camadas de arquitetura e dos conceitos requeridos para possibilitar, realmente, o desenvolvimento concorrente de componentes autônomos.
- O desenvolvimento baseado em componente subjuga a complexidade de distribuição. Organizar todo o processo de desenvolvimento em torno de componentes é um mecanismo poderoso para reduzir a complexidade e os custos de todas as fases de desenvolvimento. Tal redução é particularmente conspícua

na evolução de sistema, dado que muitas alterações ou correções no sistema afetam somente um único componente. Ao contrário, não é incomum para um sistema construído usando a abordagem tradicional onde uma pequena modificação passa por todo o ciclo de desenvolvimento. Um pedido de modificação pode afetar conceitualmente somente um módulo, mas a implementação da modificação normalmente acaba impactando vários módulos.

- O desenvolvimento baseado em componente cobre todos os aspectos de desenvolvimento. O DBC pode ser abordado de modo que todos os aspectos de desenvolvimento são tratados usando um pequeno conjunto de conceitos unificados. Um exemplo é que componentes distribuídos às empresas podem ser definidos de modo a incluir seus próprios aspectos de persistência, de tal modo que a modelagem do banco de dados e a modelagem do componente podem ser tratadas junto com um conjunto comum de princípios, na medida em que não há separação real entre ambos.
- O desenvolvimento baseado em componentes cobre todo o ciclo de vida do software. A teoria de DBC trata e informa o ciclo de vida completo de software, não apenas o ciclo de vida de desenvolvimento mais importante. Também contempla manutenção, distribuição e adequação; toda a cadeia de fornecimento de software é afetada e simplificada por uma abordagem centrada em componente.
- O desenvolvimento baseado em componente reduz prazo de entrega. Um desenvolvimento baseado em componente maduro reduzirá o *time-to-market*, para o desenvolvimento de software. Isso pode ser verificado quando se usa *frameworks* de componente e *templates* baseados em componentes. Efetivamente, é possível reunir novos sistemas, facilmente, a partir de componentes existentes, reduzindo o tempo de desenvolvimento para um grau que atualmente poderia parecer impossível.

De acordo com Hennings (1997), as bibliotecas de componentes de software reusáveis, continuarão a crescer e a questão de recuperação de componentes dessas bibliotecas de software tem despertado a atenção da comunidade de engenharia de software. Principalmente nas abordagens de reuso baseado em componentes, onde desenvolvedores combinam partes de software em uma aplicação, bibliotecas de componentes são necessárias para alcançar

reuso de software. Reuso de software baseado em componentes enfrenta um dilema inerente: para que a abordagem seja vantajosa, o repositório deve conter componentes suficientes para apoiar os desenvolvedores. Porém, quando existem muitos exemplos disponíveis, encontrar e escolher os componentes apropriados torna-se penoso.

Embora várias técnicas de recuperação tenham sido usadas para tratar o problema de encontrar componentes relevantes, a questão envolvendo a eficiência e eficácia com que os repositórios são construídos, populados e evoluídos para atender as mudanças das organizações tem recebido pouca atenção. No mínimo, deve haver uma categorização dos componentes, antes de serem colocados no repositório (HENNINGER, 1997).

Métodos são necessários para fornecer efetividade adequada para recuperação, com pouca indexação e esforço de estruturação, permitindo que as organizações tirem vantagens de artefatos valiosos, acumulados a partir de desenvolvimentos anteriores, sem grandes investimentos iniciais. Além do problema quanto a custo, as estruturas requeridas para recuperação normalmente são estáticas e incapazes de se adaptarem ao contexto dinâmico de desenvolvimento. Uma vez criadas, as estruturas permanecem estáticas, envolvendo freqüentemente o re-projeto do repositório completo. Isso é um problema significativo não somente porque é difícil obter uma estrutura correta na primeira vez, mas também porque o domínio evolui constantemente sob as pressões de mudanças tecnológicas, projetos dinâmicos e a característica fluída das necessidades do cliente (HENNINGER, 1997).

Para Henninger (1997), técnicas de recuperação são utilizadas para compensar a pouca estrutura de recuperação e para ajudar os desenvolvedores a encontrarem componentes reusáveis.

Existem duas atividades básicas no desenvolvimento de software baseado em componentes: a primeira refere-se ao desenvolvimento de software para reuso e a segunda atividade está relacionada ao desenvolvimento de software com reuso, ambas as abordagens são descritas nas seções a seguir.

5.1.1 Desenvolvimento de software com reuso

Tradicionalmente, reuso é abordado em duas fases: construção de software **para** reuso e construção de software **com** reuso. A construção de software para reuso consiste no desenvolvimento de qualquer artefato de software explicitamente para reuso. Isso exige que recursos e orçamentos, devem ser especificados de modo que desenvolvedores considerem a etapa adicional necessária para assegurar que os vários artefatos de software são reusáveis do ponto de vista da qualidade. O programa de reuso pode determinar, que essas etapas são executadas por uma organização de reuso específica, ao invés de pelo produtor original do artefato de software. A construção de software com reuso implica que a organização define as etapas extras no processo de desenvolvimento requerido para assegurar que a busca e a utilização apropriada de artefatos de software pré-existentes é realizada, segundo o entendimento de Herzum (2000).

O reuso de software existente tem sido reconhecido como a mais promissora abordagem para alcançar os objetivos de redução de custo de desenvolvimento e manutenção, para melhoria da qualidade dos sistemas de software como um todo e para diminuir o tempo de entrega no mercado - fator de vantagem competitiva para muitas organizações. As organizações aptas a desenvolverem sistemas a partir de componentes existentes, diminuem os custos de desenvolvimento, aumentam a qualidade devido ao fato de os componentes terem sido testados em outros contextos, o tempo de entrega é reduzido e o custo de manutenção é diminuído dado que transformar componentes beneficia múltiplos sistemas (JAN, 2000).

O desenvolvimento de software com reuso é uma abordagem para desenvolvimento, a qual tenta maximizar o reuso de componentes de software existentes. Uma vantagem óbvia dessa abordagem segundo Sommerville (1996), é a redução do custo total de desenvolvimento. Poucos componentes de software necessitam ser especificados, projetados, implementados e validados. Todavia, a redução de custos é somente uma vantagem potencial de reuso. Reuso sistemático no processo de desenvolvimento oferece as seguintes vantagens adicionais:

- Confiabilidade do sistema é intensificada. Componentes reusados, os quais têm sido exercitados na construção de sistemas, tendem a ser mais confiáveis que novos componentes. Esses componentes foram testados em sistemas operacionais e, por conseguinte, foram expostos às condições realistas de operação.
- Risco no processo como um todo é reduzido. Há menos incerteza quanto a custos de se reusar um componente do que de desenvolver um novo componente. Esse é um fator importante para gerenciamento de projeto, dado que reduz as incertezas nas estimativas de custos de projeto. Isso é particularmente verdade, quando componentes relativamente grandes, tal como subsistemas são reusados.
- Uso efetivo pode ser entendido por especialistas. Ao invés de os especialistas realizarem o mesmo trabalho, em diferentes projetos, podem desenvolver componentes reusáveis que encapsulam seus conhecimentos.
- Padrões organizacionais podem ser incorporados em componentes reusáveis. Alguns padrões, tal como padrões de interface com o usuário, podem ser implementados como um conjunto de componentes padrões. Por exemplo, componentes podem ser desenvolvidos para implementar menus em uma interface com o usuário. Todas as aplicações apresentam os mesmos formatos de menus para usuários. O uso de interface padrão aumenta e melhora a confiabilidade, dado que os usuários são menos propensos a cometer erros quando se deparam com uma interface familiar.
- Tempo de desenvolvimento de software pode ser reduzido. Produzir um sistema para o mercado o mais cedo possível é, muitas vezes, mais importante que o custo total de desenvolvimento. Reusar componentes aumenta a produção de sistemas porque, tanto o tempo de desenvolvimento como o de validação é reduzido.

Na abordagem descrita por Sommerville (1996), o reuso pode ser associado ao processo de desenvolvimento, incorporando uma atividade de reuso específica, conforme ilustrado na Figura 35. O projetista do sistema conclui um projeto de alto nível e as especificações de tal projeto. Essas especificações são usadas para encontrar componentes de reuso, os quais podem ser incorporados no nível de arquitetura ou em níveis mais detalhados de projeto.

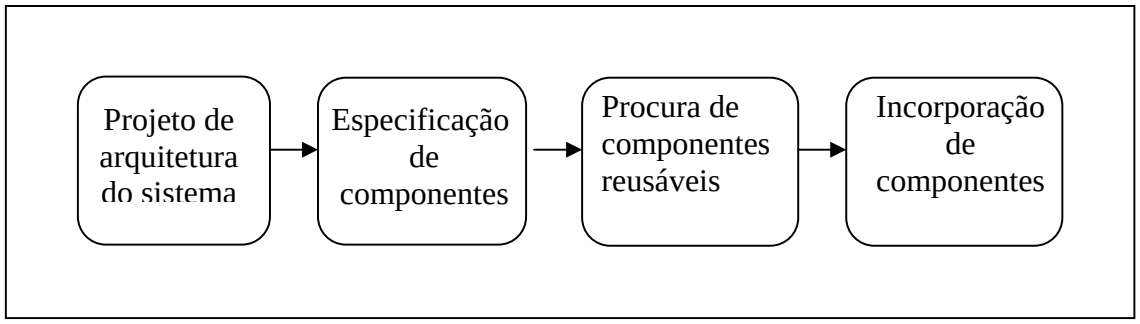


Figura 35. Reuso em um processo de desenvolvimento padrão (SOMMERVILLE, 1996)

Embora esse modelo de desenvolvimento possa resultar em reuso significante, o mesmo contrasta com outras abordagens nas quais a reusabilidade conduz o processo de projeto. Antes de iniciar o projeto, os engenheiros procuram encontrar componentes reusáveis, ou seja, baseiam seus projetos nos componentes encontrados, conforme ilustrado na Figura 36.

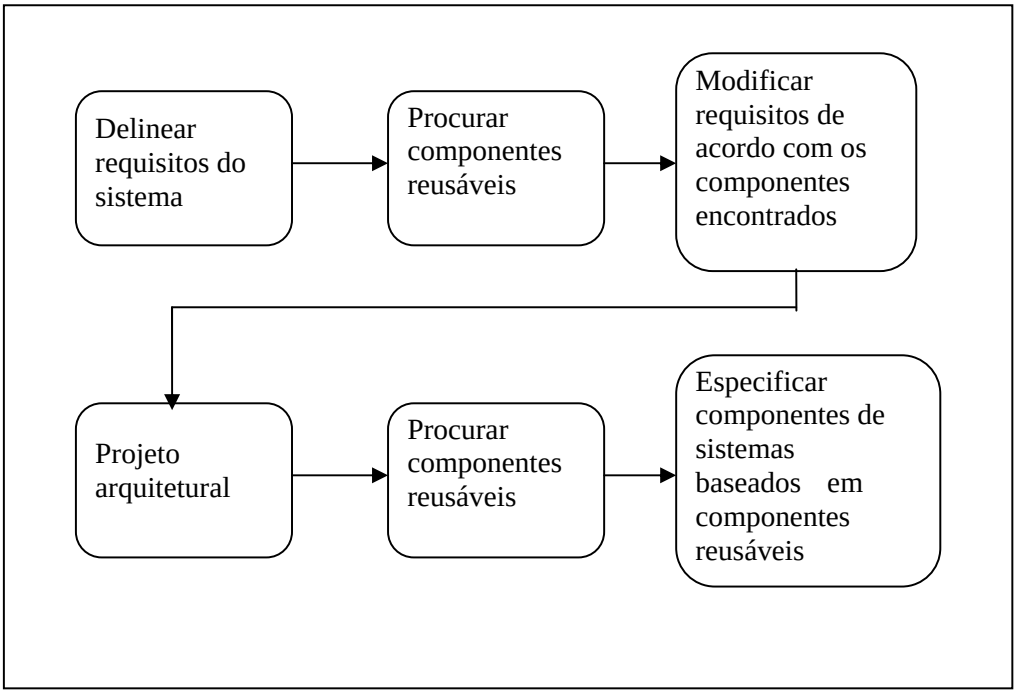


Figura 36. Processo de desenvolvimento dirigido a reuso (SOMMERVILLE, 1996)

No desenvolvimento dirigido a reuso, os requisitos do sistema são modificados de acordo com os componentes reusáveis disponíveis. O projeto também é baseado considerando os componentes existentes. Isso significa que pode haver comprometimento dos requisitos. O projeto pode ser menos eficiente que um projeto com propósito especial, o que, com freqüência, pode ser compensado devido à diminuição do custo de desenvolvimento e ao aumento de confiabilidade do sistema, Sommerville (1996).

Uma questão crítica quando do uso de componentes, diz respeito a como identificá-los e como descrever suas propriedades qualitativas e funcionais para permitir uma fácil identificação, recuperação e o conseqüente reuso dos respectivos componentes (SCHMIETENDORF, 2000).

Para Sommerville (1996), existem três condições para o desenvolvimento de software com reuso, a saber:

- Deve possibilitar a obtenção de componentes reusáveis apropriados. Organizações precisam de uma base de componentes reusáveis adequadamente catalogados e documentados. O custo de encontrar um componente apropriado nesse catálogo é relativamente baixo.
- O reusador de componentes deve ter confiança de que os componentes se comportarão conforme o esperado e serão confiáveis. Idealmente, todos os componentes em um catálogo da organização devem ser certificados, para se confirmar que tem sido possível, obter alguns padrões de qualidade.
- Os componentes devem ter documentação associada para possibilitar a compreensão e adaptação do mesmo a uma nova aplicação.

A introdução de um processo de desenvolvimento com reuso é difícil porque, como muitos outros avanços tecnológicos, tem-se o problema sobre o que deve vir antes do que, ou seja, antes que os gerentes possam estar convencidos dos méritos do reuso, estes devem ser demonstrados em projetos reais. Contudo, antes que tais méritos sejam demonstrados, algum investimento inicial é necessário para se alcançar a condição precedente, isto é, a confiança

dos gerentes. Organizações têm visto o reuso como inerentemente arriscado e têm relutado em realizar esse investimento inicial (SOMMERVILLE, 1996).

À parte desses problemas no que tange a dispor de um programa de reuso, outras dificuldades na introdução de desenvolvimento com reuso são:

- Dificuldade de quantificar reduções de custos associados com reuso. Componentes de software têm que ser encontrados em uma biblioteca, entendidos e, algumas vezes, adaptados para funcionar em um novo ambiente. Esses custos de reuso podem, algumas vezes, suplantar o custo de re-implementar o componente.
- Ferramentas CASE não apóiam o desenvolvimento com reuso. Pode ser difícil ou impossível integrar tais ferramentas com uma biblioteca de componentes de sistema.
- Alguns engenheiros de software, algumas vezes, preferem reescrever componentes, uma vez que acreditam que podem melhorar o componente reusável. Isto é uma conseqüência natural de um processo educacional que se concentra no desenvolvimento original de software ao invés de reuso.
- As técnicas atuais para classificar, catalogar e recuperar componentes de software são incompletas. Engenheiros devem estar razoavelmente confiantes de que vão encontrar um componente na biblioteca, antes, porém, farão rotineiramente uma pesquisa de componente como parte de seu processo de desenvolvimento normal.

Essas dificuldades têm indicado que desenvolvimento com reuso tem sido uma exceção ao invés de uma norma. Além disso, existem problemas organizacionais na introdução de um programa de reuso que podem causar dificuldades (SOMMERVILLE, 1996).

Pressman (1997), faz algumas sugestões para estabelecer uma abordagem de reuso, definindo os passos irão ajudar uma organização a proceder às mudanças necessárias para explorar completamente o conceito de reusabilidade de software:

- Estabelecer um plano interno de reuso de software. Tal plano pode ajudar uma organização a controlar tanto qualidade como custos associados ao software.

- Exigir que reusabilidade de software seja parte integral de qualquer treinamento técnico gerencial.
- De acordo com um plano de reusabilidade de software interno, adquirir ferramentas e bibliotecas que contribuam mais positivamente para o reuso de software.
- Incentivar o uso de métodos e ferramentas que tenham sido demonstrados para intensificar o reuso de software.
- Rastrear e medir o reuso de software e o impacto de reuso de software. Decisões políticas devem ser tomadas com base em fatos concretos, não em conjecturas.
- Reconhecer que muito mais que apenas módulos pode ser reusado. Ferramentas, dados de teste, projeto, planos, ambientes e outros itens de software podem ser reusados.
- Acima de tudo, reconhecer que o reuso de software não é um negócio usual.

Um compromisso para reuso de software irá requerer algumas mudanças. Essas mudanças devem ser introduzidas de uma maneira efetiva. Deve-se levar em conta que o conceito de reuso é estranho para a maioria do pessoal técnico e gerencial.

De acordo com Herzum (2000), o desenvolvimento com componentes, requer que uma organização esteja pelo menos no nível dois e, preferivelmente, próximo ao nível três de maturidade do modelo CMM.

5.1.2 Desenvolvimento de software para reuso

Segundo Pressman (1997), em um cenário ideal, um componente de software desenvolvido para reuso poderia ser verificado quanto à corretitude e poderia não conter defeitos. Na realidade, a verificação formal não é realizada rotineiramente e defeitos podem ocorrer e ocorrem de fato. Contudo, a cada reuso, defeitos são encontrados e eliminados e, como

resultado, melhora a qualidade dos componentes. Ao longo do tempo, o componente torna-se virtualmente livre de defeitos.

Reuso sistemático requer uma base de componentes reusáveis, catalogada e documentada convenientemente. Uma concepção errônea é o entendimento de que esses componentes estão disponíveis nos sistemas existentes. Uma biblioteca de componentes deve ser criada para a extração e documentação desses. É improvável que componentes que são criados como parte do desenvolvimento de um sistema, sejam reusáveis imediatamente. Esses componentes são gerados considerando os requisitos do sistema no qual eles são originalmente incluídos. Para serem reusáveis, esses componentes têm que ser generalizados para satisfazer uma extensa gama de requisitos (SOMMERVILLE, 1996).

A documentação de um componente deve ser diferente da documentação de software tradicional, uma vez que deve descrever não apenas o que o componente faz, mas também, quais são as interações entre os componentes e os usuários. Assim, de acordo com Wang (2000), descrição deve conter duas partes:

- Funcionalidades estáticas.
- Comportamento dinâmico, isto é, os relacionamentos com outros componentes.

Idealmente, uma vez desenvolvido e usado em um sistema, um componente pode ser reusado sem alteração. Mais comumente, porém, será necessário adaptar o componente de algum modo, levando em conta os requisitos do sistema em desenvolvimento. A Figura 37 mostra o processo de melhoria da reusabilidade de um componente.

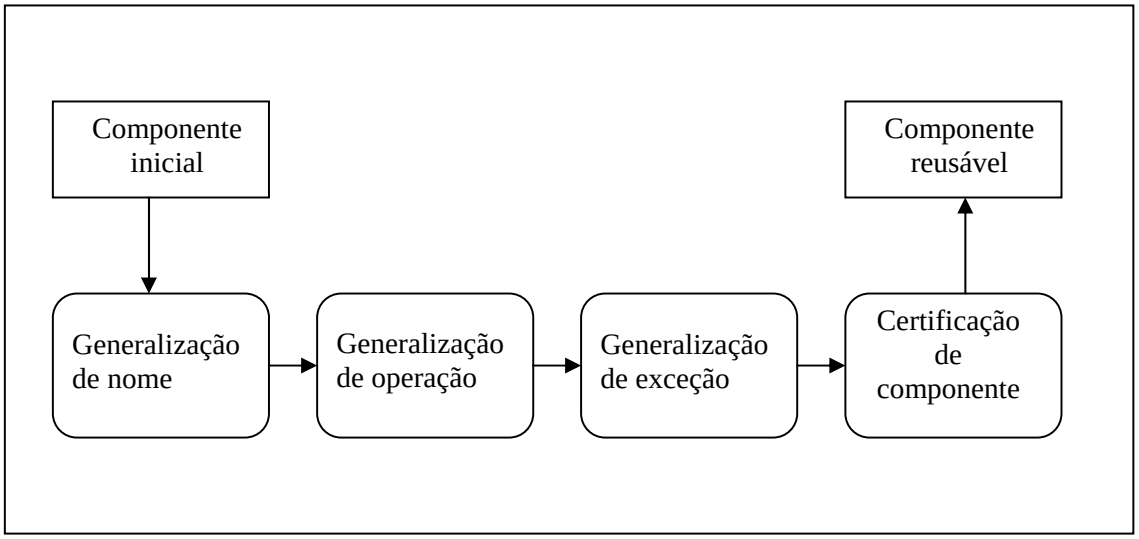


Figura 37. O processo de aumento de reusabilidade (SOMMERVILLE, 1996)

Adaptar um componente para torná-lo reusável pode envolver a realização de diferentes tipos de mudanças (SOMMERVILLE, 1996):

- **Generalização de nome**. Os nomes usados nos componentes podem ser modificados, dado que estes são neutros e não uma reflexão direta de alguma entidade específica da aplicação.
- **Generalização de operação**. Pode envolver operações adicionais para um componente ou a remoção de operações as quais são muito específicas para algum domínio de aplicação.
- **Generalização de exceção**. Pode envolver a checagem de cada componente para averiguar quais exceções o componente deve gerar e, incluir essas exceções na interface do componente.

Após a generalização a qualidade do componente generalizado deve ser verificada. Isto pode exigir inspeções de programa ou teste. Idealmente, os dados de teste para um componente devem estar disponíveis para reusadores, de modo que também possam ser reusados. O componente pode ser certificado como tendo obtido os padrões de qualidade requeridos.

Em geral, é improvável que o reuso de componente seja específico a um domínio. Se um componente é escrito para algum domínio, tal como sistemas de biblioteca, é improvável que o mesmo seja reusado eficazmente em termos de custos em outros domínios. Alguns domínios, tal como o domínio de interface com o usuário, são usados em muitos tipos diferentes de aplicação, de modo que é possível o reuso entre aplicações. A análise de domínio está relacionada à compreensão do domínio, de modo que as abstrações do domínio possam ser modeladas como componentes reusáveis (SOMMERVILLE, 1996).

Ainda, segundo Sommerville (1996), a conversão de componentes existentes em componentes reusáveis genéricos está sujeita, a custos consideráveis. Se esses custos já são encontrados em simples orçamentos de projeto, é improvável que o gerente de projeto possa apoiar o desenvolvimento de componente reusável. Como a principal responsabilidade dos gerentes de projeto é minimizar custos, é compreensivo que relutem em investir qualquer esforço extra em criar componentes, os quais, não lhes trarão qualquer retorno direto.

Para Sommerville (1996), o desenvolvimento de componentes reusáveis requer uma decisão organizacional para incrementar custos de curto prazo para ganhos potenciais de longo prazo. É difícil quantificar esse ganho potencial, de modo que, poucas organizações têm estado propensas a fazer esse investimento em uma biblioteca de investimento, por exemplo. Entretanto, mais e mais organizações estão atualmente, considerando componentes de software como bens ativos a serem explorados e desenvolvendo componentes para reuso.

Aumentar os níveis de reuso de software constitui uma das mais difusas e profundas influências na engenharia de software atualmente. Inovações tecnológicas (linguagens orientadas a objetos, modelos de objetos distribuídos, linguagens específicas de domínios) e inovações de processo (modelagem de domínio, análise arquitetural, métricas de reuso) estão possibilitando às organizações conseguir continuamente custo convincente, qualidade e requisitos de tempo. Além disso, há um mercado para reuso, onde é possível, literalmente, se adquirir biblioteca de classes, *frameworks* e outros componentes, (DEVANBU, 2000).

Para Butler (1997), fatores críticos para a introdução bem sucedida de reuso sistemático em uma organização cobrem uma série de áreas técnicas e não técnicas. Esses fatores incluem:

- Suporte de gestão *top-down* de longo prazo é vital porque podem ser necessários vários anos antes que o investimento com reuso se pague e porque reuso sistemático requer mudanças nas políticas organizacionais e nas estruturas gerenciais.
- Dimensionamento, coleta e análise de dados devem ser executados durante o processo de software a fim de quantificar os custos e benefícios do reuso. Isto é necessário para justificar o suporte gerencial contínuo para reuso.
- Mudanças econômicas no modo como a organização cria projetos são necessárias para justificar e incorporar o uso de componentes reusáveis. *Marketing* deve identificar os domínios de negócio onde há abundante oportunidade para uma variedade de produtos estreitamente relacionados e, a partir daí, empregar reuso. Uma organização necessita realizar uma análise de custo e benefício para esses domínios, a fim de demonstrar suficiente reuso de componentes.
- Fatores técnicos também devem ser considerados, tais como: o nível de maturidade do processo da organização, a identificação de domínios de aplicação restritos e bem compreendidos, engenharia de domínio e a construção de *frameworks* de aplicação e bibliotecas de componentes reusáveis.

Para Fowler (1995), a existência de bibliotecas reusáveis, conduz a um enorme crescimento da produtividade do programador.

A construção de repositórios de artefatos reusáveis é uma das preocupações das organizações que optam pela adoção de uma abordagem que contemple reuso de software.

Quando da construção de repositórios para reuso de software, as organizações de desenvolvimento de software se defrontam com dois problemas inter-relacionados:

- Adquirir o conhecimento inicial para construir o repositório.
- Modificar o repositório para atender a evolução e a dinâmica necessárias a essas organizações.

No entendimento de Henninger (1997), para ser útil, um repositório necessita ter o suporte de:

- Ferramentas para criar estruturas de informação iniciais.
- Mecanismos flexíveis para pesquisar e visualizar o repositório
- Ferramentas para refinar e adaptar informação a sobre a maneira como os usuários trabalham com o repositório.

Para não extrapolar o objetivo e o escopo do presente trabalho, nem todos os fatores relacionados à prática de reuso serão aqui abordados. Entretanto, é necessário que algumas considerações sobre modelos sejam feitas, visando demonstrar sua importância também na abordagem de reuso sistemático.

5.2 MODELAGEM E REUSO

Os pré-requisitos para reuso incluem uma linguagem de modelagem comum, decisões arquiteturais comuns, semânticas comuns e muitos outros elementos, Herzum (2000).

Para Butler (1997), a modelagem por si só não fornece uma vantagem econômica direta, mas modelos são ferramentas fundamentais e modelagem é uma atividade essencial na melhoria da qualidade de produtos e processos e reuso efetivo. O reuso conta com modelos que fornecem análise de domínio de um domínio de aplicação, os quais podem incluir:

- Uma taxonomia de entidades, atividades e componentes.
- Uma arquitetura de domínio específico de software ou *framework*.
- Linguagens visuais e textuais que descrevem cada produto de software em uma família de produto.

Ainda segundo (BUTLER, 1997), os modelos são abstrações que fornecem aos desenvolvedores, controle intelectual sobre a complexidade de um sistema. Por essa razão, é importante compreender os modelos usados no software e os modelos usados pelas pessoas

durante as fases de processo de software tais como análise e projeto. Os modelos de projeto podem ser classificados em três categorias:

- Modelos diagramáticos, baseados em uma representação gráfica visual.
- Modelos textuais, escritos em pseudocódigo ou linguagem natural.
- Modelos matemáticos, baseados em lógica, álgebra, teoria dos conjuntos ou outras notações matemáticas.

Geralmente, existem múltiplas visões de um sistema e cada visão foca um único aspecto do sistema numa tentativa de reduzir a complexidade. Pontos de vistas ou perspectivas comuns usados pelos desenvolvedores incluem (BUTLER, 1997):

- Modelos estruturais dos aspectos estáticos de um sistema, os quais podem incluir a organização de subsistemas e módulos no nível arquitetural.
- Visões funcionais que buscam descrever o que o sistema faz em termos de suas tarefas.
- Visões comportamentais, que descrevem a natureza dinâmica ou casual de eventos e respostas do sistema durante a execução.
- Esquemas de modelagem de dados ou diagramas de classe dos dados usados no sistema e o relacionamento entre eles.

As representações usadas por essas visões também apóiam decomposição e hierarquias e, posteriormente, dão suporte às pessoas quanto ao controle da complexidade, mesmo em se tratando de aspectos simples de um sistema. Por exemplo, é possível usar módulos aninhados para descrever a visão estrutural de um sistema; diagramas de fluxo de dados hierárquicos para descrever funcionalidade; diagramas de estados, os quais fornecem aninhamentos, abstração e decomposição para modelar comportamento e; uma hierarquia de herança de classe para modelar dados (BUTLER, 1997).

O importante é saber o que se quer modelar e conhecer o modelo a ser utilizado. Vários modelos podem ser utilizados em cada fase do processo de desenvolvimento de software e, talvez, o maior benefício seja o uso combinado de vários modelos durante o ciclo de desenvolvimento de software. Embora não seja objetivo deste trabalho esmiuçar os modelos

existentes – exceto aqueles diretamente relacionados à qualidade de software e do processo - é importante alinhavar algumas considerações acerca de alguns modelos diretamente relacionados ao reuso sistemático.

A Figura 38 ilustra os modelos de reuso e métricas os quais são categorizados em tipos, de acordo com Frakes (1996):

- Modelos de custo e benefício de reuso. Incluem análise econômica de custo e benefício tanto como resultados de produtividade e qualidade.
- Modelos de avaliação de maturidade. Categorizam programas de reuso pelo quão avançados estão na implementação de reuso sistemático, usando uma escala ordinal de fases de reuso.
- Quantidade de reuso. Métricas de quantidade de reuso são usadas para avaliar e monitorar uma tentativa de melhoria de reuso, acompanhando percentuais de reuso de objetos do ciclo de vida ao longo do tempo.
- Modos de falha. A análise de modos de falha é usada para identificar e ordenar os empecilhos para reuso em uma dada organização.
- Reusabilidade. Métricas de reusabilidade indicam a probabilidade de um artefato ser reusável. Está relacionada à estimativa de reusabilidade de um componente. São potencialmente úteis em duas áreas chave de reuso, projeto de reuso e reengenharia para reuso.
- Métricas de bibliotecas de reuso. São usadas para gerenciar e rastrear o uso de um repositório de reuso. Uma biblioteca de reuso é um repositório para armazenar artefatos reusáveis, juntamente com uma interface para permitir pesquisas no repositório. Os componentes são certificados por um processo, que assegura que os mesmos possuem os atributos desejados. Os componentes são classificados de modo que possam ser efetivamente pesquisados.

Para Pressman (1997), a biblioteca de reuso é um elemento de um repositório CASE mais amplo e fornece facilidades para armazenar uma extensa variedade de artefatos reusáveis (especificações, projetos, código, casos de teste, guias do usuário). A biblioteca inclui um banco de dados e as ferramentas necessárias para acessar o banco de dados e recuperar componentes.

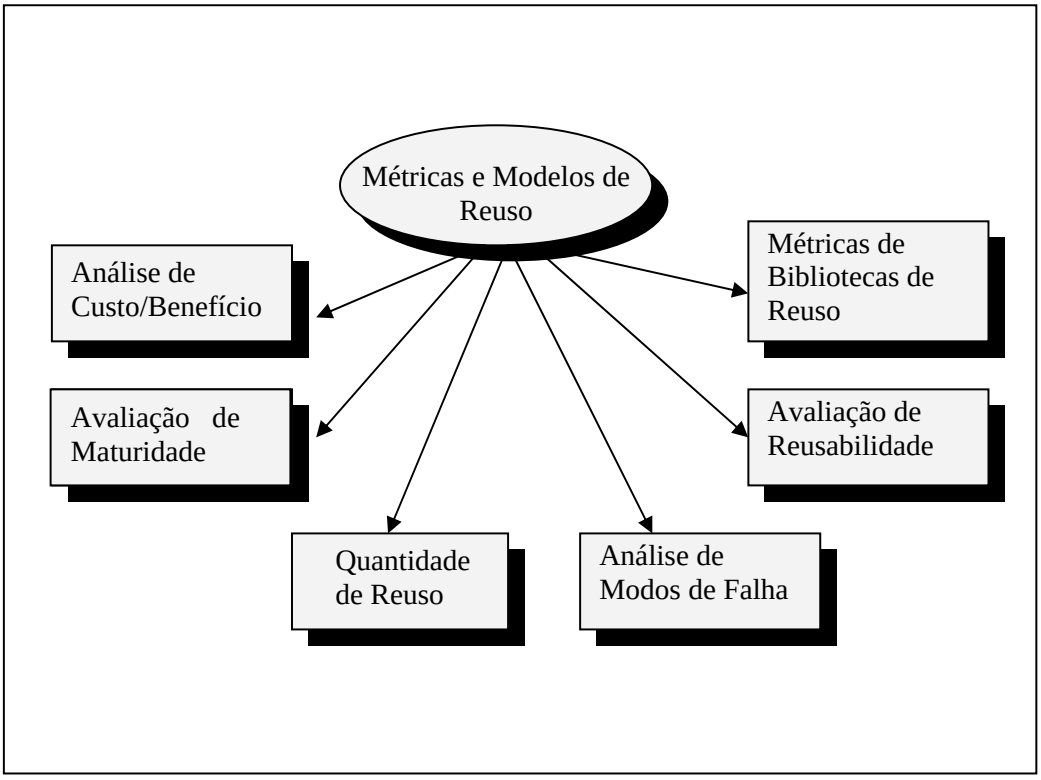


Figura 38. Categorização de métricas e modelos de reuso (FRAKES, 1996)

Para Frakes (1996) as organizações enfrentam a necessidade dessas métricas e modelos, de acordo com a ordem apresentada.

Dentre os vários termos encontrados para designar algo que pode ser reusado, encontram-se: artefatos reusáveis, *assets* reusáveis e componentes reusáveis. O contexto de reuso também tem variado ao longo dos anos e, na literatura da área poderão ser encontradas diversas definições com distintas abordagens e diversas abrangências sobre reuso, desde código simplesmente até estruturas mais complexas.

Os tipos de reuso também têm merecido a atenção da comunidade de engenharia de software. É imprescindível que as organizações determinem os tipos de reuso que pretendem empregar uma vez que deve haver um planejamento sobre quais modelos utilizar para modelá-los, que

métodos serão utilizados (desenvolvidos) visando a sua colocação e conseqüente recuperação do repositório de artefatos e assim por diante.

O reuso de software pode se aplicado a qualquer produto do ciclo de vida, não somente a fragmentos de código fonte. Isto significa que os desenvolvedores podem buscar o reuso de documentos de requisitos, especificações de sistema, estruturas de projeto e outros artefatos de desenvolvimento (FRAKES, 1996).

Conforme pode ser visto na Tabela 14, foram identificados dez aspectos potencialmente reusáveis de projetos de software. Em adição a esses produtos do ciclo de vida, processos de desenvolvimento de software (tal como o modelo em cascata e o CMM), casos de teste, resultados de teste, dados de inspeções e conhecimento dentre tantos outros, são também potencialmente reusáveis.

Para Frakes (1996), definições claras de tipos de reuso, são pré-requisitos necessários para medição. A Tabela 16 fornece uma classificação de definições de reuso. Os termos sinônimos estão entre parêntesis.

Tabela 16. Tipos de reuso de software (FRAKES, 1996)

Escopo de desenvolvimento	Modificação	Abordagem	Escopo de domínio	Gerenciamento	Entidade reusada
Interno (privado)	Caixa branca	Generativo	Vertical	Sistemático (planejado)	Código
Externo (público)	Caixa preta (literal)	Composicional	Horizontal	“ad hoc”	Nível abstrato
	Adaptativo	*In-the-small*			Nível de instância
		In-the-large			Reuso de adaptação
		Indireto			Genérico
		Direto			Código-fonte
		Carried over			
		leveraged			

O escopo de desenvolvimento refere-se a se os componentes reusáveis são provenientes de uma fonte externa ou interna a um projeto. Modificação refere-se a quanto um artefato

reusável é modificado. Abordagem refere-se a diferentes métodos técnicos para implementar reuso. Escopo de domínio refere-se a se o reuso ocorre dentro de uma família de sistemas ou entre famílias de sistemas. Gerenciamento refere-se ao grau em que o reuso é feito sistematicamente. Entidade reusada refere-se ao tipo de objeto reusado (FRAKES, 1996).

Rine (1997) define partes/componentes reusáveis de software como módulos de código, documentação, dados de teste, arquitetura/projeto/código, documentação e sistemas/subsistemas completos que podem ser integrados com partes/componentes de software comprados ou construídos em um produto de software novo ou existente.

O reuso em uma organização pode ser definido pela seleção de pares de termos a partir da Tabela 16. Ou seja, uma organização pode escolher usar tanto reuso de código externo como interno, possibilitando somente a modificação caixa preta, como parte de uma abordagem composicional. Pode escolher concentrar-se no reuso dentro de seu domínio (vertical) e buscar uma abordagem de gerenciamento sistemática (FRAKES, 1996).

Para Frakes (1996), a partir do momento que as organizações adotam o reuso sistemático de software, provavelmente a primeira questão que surge relaciona-se a custo e benefício. As organizações necessitam justificar o custo e o tempo, envolvidos no reuso sistemático, estimando os potenciais custos e resultados finais. Os modelos de análise de custo e benefício incluem modelos econômicos de custo e benefício e análise de resultados de qualidade e de produtividade.

As Tabelas 17(a), 17(b) e 17(c), ilustram dez aspectos de projetos de software potencialmente reusáveis. Em adição a esses produtos, processos (tais como o modelo em cascata para desenvolvimento de software e o modelo CMM) e conhecimento também são potencialmente reusáveis.

Tabela 17(a). Definições de tipos de reuso (FRAKES, 1996)

Tipo de Reuso	Descrição
Reuso no nível abstrato	É o uso de abstrações de alto nível dentro de uma estrutura de herança orientada a objeto como a fundamentação para novas idéias ou esquemas de classificação adicionais (MacGregor e Sykes, 1992).
Ad-hoc	Refere-se à seleção de componentes os quais não são designados para reuso de bibliotecas genéricas; reuso é conduzido de maneira informal (Prieto-Diaz, 1993).
Adaptativo	É uma estratégia de reuso que usa grandes estruturas de software como constantes e restringe a variabilidade a localizações isoladas de baixo nível. Um exemplo é a mudança de argumentos para módulos parametrizados (Barnes e Bollinger, 1991).
Caixa-branca	É o reuso de componentes de software sem qualquer modificação (Prieto-Diaz, 1993).Ver reuso literal.
Reuso *carry-over*	Quando uma versão de um componente de software é para ser usada como uma versão subseqüente do mesmo sistema (Ogush, 1992).
Composicional	É uma estratégia de reuso que usa pequenas partes como constantes; funcionalidade variante une essas partes. Programar em uma linguagem de alto nível é um exemplo (Barnes e Bollinger, 1991). Reuso composicional é o uso de componentes existentes como blocos de construção para novos sistemas. A *shell* do Unix é um exemplo (Prieto-Diaz, 1993).
Reuso de adaptação	É o uso de herança orientada a objeto para apoiar desenvolvimento incremental. Uma nova aplicação pode herdar informação de uma classe existente, suprimindo certos métodos e adicionando novos comportamentos (MacGregor e Sykes, 1992).

Tabela 17(b). Definições de tipos de reuso (FRAKES, 1996)

Tipo de Reuso	Descrição
Direto	É o reuso sem passar por uma entidade intermediária, (Bieman e Karunanithi, 1993).
Externo	O nível de reuso externo é o número de itens de mais baixo nível de um repositório externo em um item de nível mais alto, dividido pelo número total de itens de mais baixo nível no item de nível mais alto (Frakes, 1990). Ver reuso público.
Generativo	É o reuso no nível de especificação com geradores de aplicação ou de código. Reuso generativo oferece o 'potencial de resultado final mais alto', (Prieto-Diaz, 1993).
Genérico	È o reuso de pacotes genéricos, tal como *templates* para pacotes ou subprogramas (Bieman e Karunanithi, 1993).
Escopo horizontal	É o reuso de partes genéricas em diferentes aplicações. Bibliotecas de sub-rotinas são exemplos (Prieto-Diaz, 1993).
In-the-large	É o reuso de grandes pacotes auto-contidos tais como *spreadsheets* e sistemas operacionais (Favaro, 1991).
In-the-small	É o reuso de componentes os quais são dependentes do ambiente da aplicação para funcionalidade completa. Favaro (1991) afirma que reuso orientado a componente é reuso *in-the-small*.
Indireto	É reuso por meio de uma entidade intermediária. O nível de indireção é o número de entidades intermediárias entre o item de reuso e o item que está sendo reusado (Bieman e Karunanithi, 1993).

Os termos das Tabelas 17(a), 17(b) e 17(c), mostram várias questões relacionadas a reuso. Alguns termos se sobrepõem quanto ao significado. Ambos os termos, público e externo, descrevem a parte de um produto que foi construído externamente; privado e interno descrevem a parte de um produto que não foi construído externamente, mas foi desenvolvido e reusado dentro de um único produto. Os termos, literal e caixa preta descrevem reuso sem modificação; os termos *leveraged* e caixa branca descrevem reuso com modificação. As referências incluídas na descrição dos tipos de reuso, são fornecidas como informações adicionais não indicando necessariamente o primeiro uso do termo (FRAKES, 1996).

Tabela 17(c). Definições de tipos de reuso (FRAKES, 1996)

Tipo de Reuso	Descrição
Reuso ao nível de instância	É a forma mais comum de reuso em um ambiente orientado a objeto. É definido simplesmente como a criação de uma instância de uma classe existente (MacGregor e Sykes, 1992).
Interno	O nível de reuso interno é o número de itens de mais baixo nível de um repositório não externo os quais são usados mais de uma vez, divididos pelo número total de itens de mais baixo nível de um repositório não externo (Frakes, 1990). Ver reuso privado.
Leveraged	Definido por Bieman e Karunanithi (1993) como reuso com modificações.
Privado	Fenton (1991) define reuso privado como ‘a quantidade de módulos dentro de um produto que são reusados dentro do mesmo produto’. Ver reuso interno.
Público	Definido por Fenton (1991), como ‘a proporção de um produto o qual foi construído externamente’. Ver reuso externo.
Reuso de código-fonte	É a modificação de baixo nível de uma classe orientada a objeto existente, para mudar suas características de desempenho.
Sistemático (modo planejado)	É uma prática sistemática e formal de reuso tal como encontrada em fábricas de software (Prieto-Diaz, 1993).
Literal	É o reuso de um item sem modificações (Bieman e Karunanithi, 1993).
Escopo vertical	É o reuso dentro da mesma aplicação ou domínio. Um exemplo é análise de domínio ou modelagem de domínio (Prieto-Diaz, 1993).
Caixa branca	É o reuso de componentes por modificação ou adaptação (Prieto-Diaz, 1993). Ver reuso *Leveraged*.

O principal objetivo em se adotar desenvolvimento baseado em componente deve-se ao fato de que essa abordagem tem um alto foco na redução de complexidade funcional e de custo (HERZUM, 2000).

5.3 REUSO BASEADO EM GERADOR

Para Sommerville (1996), uma alternativa à abordagem de reuso orientada a componentes é a abordagem baseada em gerador. Nessa abordagem para reuso, conhecimento reusável é apreendido em um sistema gerador de programa o qual pode ser programado em uma linguagem orientada a domínio. A descrição da aplicação específica incluiria, de um modo abstrato, quais componentes reusáveis serão utilizados, como estão agrupados e suas parametrizações. Usando essas informações, um sistema de software operacional pode ser gerado. A Figura 39 ilustra essa abordagem para reuso.

De acordo com Butler (1997), no reuso por geração, os componentes ou sistema são gerados automaticamente a partir de uma descrição da solução desejada. A descrição pode usar uma linguagem *script*, tal como linguagem de quarta geração para bancos de dados relacionais ou *metaphor* visual, como em construtores de interfaces gráficas para o usuário.

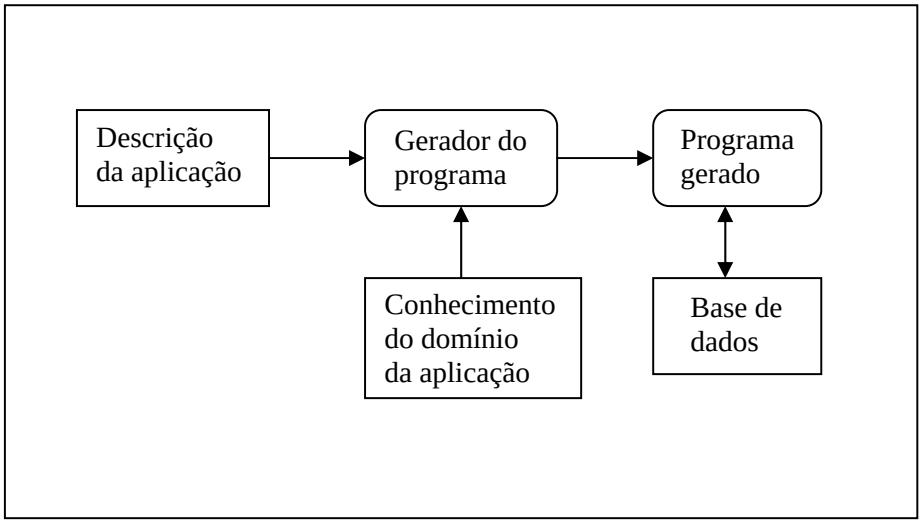


Figura 39. Reuso de conhecimento de domínio da geração de aplicação (SOMMERVILLE, 1996)

Os geradores de programa, mais amplamente utilizados, são compiladores de linguagem de alto nível, onde os componentes reusáveis são fragmentos de código objeto correspondente a construções de linguagem de alto nível. Os elementos reusados são abstrações correspondentes a declarações de linguagem de programação. Quando uma notação específica de domínio é usada para descrever a aplicação, abstrações de domínio mais amplas podem ser reusadas. Para Sommerville (1996), exemplos dessa abordagem mais abstrata são:

- Geradores de aplicação para processamento de dados de negócio (transação). A entrada para esses pode ser uma linguagem de quarta geração ou pode ser completamente interativa, onde o usuário define telas e ações de processamento. A saída é um programa em uma linguagem tal como COBOL[31] ou SQL[32].
- Geradores de analisadores para processamento de linguagem. A entrada do gerador é uma gramática descrevendo a linguagem a ser analisada e a saída é um analisador de linguagem.
- Gerador de código em ferramenta CASE. A entrada desses geradores é um projeto de software e a saída é um programa implementando o sistema projetado.

Geradores de aplicação para sistemas de negócio (transação) são úteis porque muitas aplicações de processamento de dados envolvem a extração de informação da base de dados, a execução de alguma informação relativamente simples, processamento e execução de relatórios a partir dessa informação. O gerador inclui uma linguagem de controle que possibilita que esses componentes sejam combinados em programas. Uma aplicação completa pode então ser gerada.

Reuso baseado em gerador é eficiente quanto ao custo, mas depende da identificação estereotípica de abstrações do domínio. Isto tem sido possível para processamento de dados de negócios.

Um caso especial de reuso de software é reuso de sistema de aplicação onde todo o sistema é reusado por meio de sua implementação entre uma série de diferentes computadores e

[31] Do inglês: ***Co**mmon **B**usiness **O**riented **L**anguage*.

[32] Do inglês: *Structured Query Language*.

sistemas operacionais. O problema aqui não é descobrir componentes para reuso, mas desenvolver o sistema de modo que este seja portável em diferentes plataformas.

Em 1995, Rine (1997), realizou um estudo com o intuito de investigar o relacionamento entre reuso de software, produtividade e qualidade. Os resultados indicaram que:

- As organizações com alta capacidade de reuso de software têm uma abordagem de linha de produto, arquitetura com interfaces padronizadas e formatos de dados, arquitetura de camada de software entre a linha de produto, projeto para abordagem de manufatura, engenharia de domínio, processo de reuso, gerenciamento com conhecimento relacionado a reuso, defesa de reuso de software no gerenciamento de mais alto nível, ferramentas e métodos de reuso do estado da arte, precedência no reuso de alto nível de artefatos tal como, reuso de requisitos e projeto versus apenas reuso de código e rastrear os requisitos do usuário final para componentes (sistemas, subsistemas e/ou módulos de software) que o apóiam.
- As práticas associadas com os principais vaticinadores da capacidade de reuso de software condizem com as práticas bem sucedidas usadas na manufatura de hardware. Há uma crença substanciada na indústria de hardware de que linhas de produto, arquitetura, projeto (*design*) para manufaturabilidade e engenharia de domínio aumentam a produtividade e a qualidade.
- As práticas associadas com os principais vaticinadores de capacidade de reuso de software combinam sólidas práticas de negócio.
- Muitas organizações utilizam estudos de casos de negócios e análise econômica para determinar se uma dada prática é viável de ser implementada no que concerne a custo.
- Na maioria dos casos, as organizações com um processo de reuso de software podem apresentar melhor qualidade (uma vigorosa capacidade de reuso de software) que aquelas sem processo. Também parece razoável que organizações que têm refinado seu processo de reuso de software ao longo do tempo para grandes execuções, podem apresentar melhor qualidade que aquelas organizações as quais apenas recentemente definiram um processo.
- Na maioria dos casos, a eficiência e a eficácia dos processos de software podem ser melhoradas com ferramentas. O aumento da produtividade pode ser obtido

quando o método de desenvolvimento de software, processos e ferramentas são integrados. Parece razoável, que organizações que têm tecnologia de reuso (métodos e ferramentas) podem apresentar melhor qualidade que organizações que não possuem suporte automatizado ou que o suporte é pouco sofisticado.

- Uma organização que tem uma ferramenta para acompanhar os requisitos do usuário final quanto a componentes que os atenda, pode alcançar reuso ao nível de requisitos reusando sistemas ou subsistemas completos. Há, pelo menos, uma ordem de diferença de magnitude entre ter capacidade para reusar sistemas completos e simplesmente ter capacidade para reusar um módulo de software em particular.
- Há um número de barreiras não técnicas para reuso de software. Parece razoável que a defesa de reuso pela gerência de alto nível, a qual entende questões relacionadas a reuso de software, pode superar essas barreiras.
- As organizações mais efetivas (eficientes e eficazes) têm o foco contínuo na melhoria do processo e do produto. Melhoria na qualidade ou desempenho de uma parte preferencial, normalmente resulta em melhorias em todos os produtos na linha de produto que usam tal parte. Conseqüentemente, a abordagem de linha de produto multiplica os efeitos de melhorar partes preferenciais em particular. Entender esse conceito tem possibilitado a algumas organizações, dominar seus domínios (conquistar um amplo mercado compartilhado).
- Uma vez que uma organização tem uma massa crítica de partes preferenciais altamente confiáveis (comprovada ao longo de anos de uso operacional), novas organizações têm grandes dificuldades de entrar no mercado e competir com sucesso.
- Uma vez dominado o mercado, a organização busca novas oportunidades de negócio para impulsionar sua massa crítica de partes preferenciais. Decisões para entrar em um novo mercado podem se tornar arriscadas pela comparação entre os requisitos do usuário final de um novo domínio com os requisitos do domínio atual (análise de domínio).
- Os três contra vaticinadores de capacidade de reuso de software, mais significativos são: a criação de biblioteca, recuperação ou abordagem de refugo para reuso de software; a efetividade da biblioteca de software ou repositório e; a certificação de componentes para algum nível de qualidade.

- A abordagem de biblioteca e qualidade de componentes reusáveis não são vaticinadores de capacidade de reuso de software.
- Aproximadamente 70% (setenta por cento) dos respondentes à pesquisa, tentavam implementar uma abordagem de biblioteca para reuso de software. A grande maioria desses profissionais havia encontrado os níveis de sucesso esperado. Uma razão para essa falta de sucesso é que a abordagem de biblioteca não soluciona questões de permutabilidade ou cuida das necessidades do cliente antes de adicionar componentes no repositório. Em contraste, 100% (cem por cento) das organizações com alta capacidade de reuso, seguem a abordagem de manufatura.
- Com uma abordagem de biblioteca, os desenvolvedores de software buscam componentes reusáveis após o projeto do sistema. É provável que quando os desenvolvedores sejam capazes de localizar um componente útil, este não será integrado em seu projeto sem grande modificação. Não é incomum que na busca de componentes de baixo nível (tal como estruturas de dados) desenvolvedores de software localizem múltiplos componentes reusáveis no repositório. Cada vez que isso acontece, o desenvolvedor de software é forçado a avaliar cada componente para selecionar o que melhor atende suas necessidades.
- Uma vez que a abordagem de biblioteca carece de um forte foco no cliente, muitos componentes são adicionados no repositório com a esperança de que serão úteis posteriormente. Muitos repositórios contêm componentes que nunca serão reusados ou contêm múltiplos componentes que executam a mesma função. O problema de múltiplos componentes acontece porque pouca análise é feita para determinar qual parte é a melhor. Uma análise das partes preferenciais, tal como usado em manufatura, garante que componentes atendem os requisitos do usuário (qualidade e desempenho) e ajuda a expurgar o repositório de todos os componentes duplicados.
- Embora existam problemas, não significa que as bibliotecas ou repositórios de software não são importantes. Repositórios, assim como processos de gerenciamento de configuração devem ser vistos como capacidades suplementares. As contribuições mais significativas para reuso de software são tecnologia para linha de produto, arquitetura e projeto (*design*) para o processo de manufatura.

Rine (1997) enfatiza que o reuso de software é vaticinador de produtividade e qualidade. Há um conjunto de fatores de sucesso comuns entre organizações e têm relação de previsibilidade para reuso de software. As organizações com mais alta capacidade de reuso possuem:

- Abordagem de linha de produto.
- Arquitetura com interfaces padrão e formatos de dados.
- Arquitetura de software comum entre a linha de produto.
- Projeto para abordagem de manufatura.
- Engenharia de domínio.
- Processo de reuso.
- Gerenciamento com entendimento de questões sobre reuso.
- Defesa de software no gerenciamento de alto nível.
- Precedência de reuso de alto nível de artefatos de software tal como requisitos e projeto versus apenas reuso de código.
- Acompanhamento dos requisitos do usuário final para os componentes (sistemas, subsistemas e (ou) módulos de software) que os apóiam.

Pressman (1997), afirma que, um rápido estudo dos possíveis benefícios de se reusar software, revela que existe muito mais a ser ganho que uma simples economia de custo, durante o desenvolvimento de um produto de software. Por exemplo, o reuso de um componente de software o qual é tido como confiável introduz menos riscos que re-projetar e re-codificar o mesmo componente para cada nova aplicação. Questões sobre eficiência também podem ser mais efetivamente tratadas se o foco da atenção estiver na otimização de um conjunto de componentes reusáveis ao invés de ter que continuamente re-otimizar novas versões de módulos existentes.

A seguir, são apresentados alguns fatores que têm contribuído grandemente com a atividade de reuso, os quais se configuram como um dos resultados das gestões da comunidade de engenharia de software em função da constante evolução no processo de desenvolvimento de software; das mudanças havidas no mercado de software, das exigências desse mercado e da competitividade existente no mercado de desenvolvimento de software.

5.4 FRAMEWORKS, PADRÕES DE PROJETO E REUSO

A evolução na indústria de software dos sistemas tradicionais para os modernos sistemas distribuídos pode ser vista como amplamente constante. Com isso, novas tecnologias e abordagens têm sido introduzidas contínua e progressivamente. Do ponto de vista de uma única organização, essa evolução é freqüentemente percebida como um intervalo (uma brecha). Esse intervalo deve-se ao fato de que a maioria das organizações tem ciclos de desenvolvimento tipicamente constantes de dois a quatro anos ou mais sobre um conjunto fixo de tecnologias. Com o tempo, as forças do mercado exigem uma organização para atualizar essas tecnologias e arquiteturas, a indústria tem evoluído muito e tão rápido que as organizações se defrontam com uma nova fase de aprendizado e um salto real em complexidade e custo. É essencial que as organizações criem meios para lidar com a complexidade e os custos crescentes (HERZUM, 2000).

Segundo Herzum (2000), a capacidade para desenvolver software de qualidade com todas as características funcionais e extra-funcionais exigidas, requer a combinação de vários fatores. Não se trata apenas de um problema de tecnologia ou metodologia, mas de uma complexa mistura de fatores que devem ser misturados de forma consistente, de modo que todos esses fatores sejam tratados como um todo unificado. Os fatores mais importantes que influenciam o desenvolvimento de software em termos de uma avaliação de custo e benefício, conforme ilustrado na Figura 40, são os seguintes:

- Ferramentas integradas e ambientes de desenvolvimento. São exigidos para a construção de software. Em termos de construção são um aspecto crítico (de domínio) de tarefas de construções complexas.
- Pessoas e organização. Construção de software prescinde uma prática de construção pessoal intensa e o fator humano é considerado o fator mais importante na indústria.
- Práticas de engenharia de software. Incluem as metodologias, processo de desenvolvimento e boas práticas exigidas para se ter uma organização que usa as ferramentas integradas e os ambientes de desenvolvimento para produzir software eficientemente.

- Arquitetura e reuso. Refere-se ao conjunto de princípios arquiteturais, padrões (*patterns*), artefatos reusáveis e *frameworks* para apoiarem a construção de software.

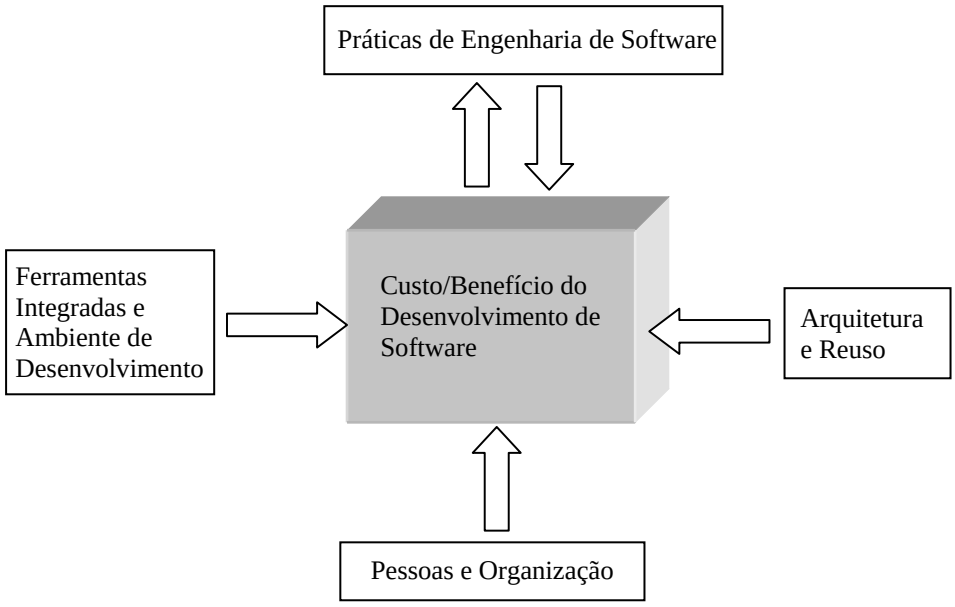


Figura 40. Fatores que influenciam o desenvolvimento de software em termos de custo/benefício (HERZUM, 2000)

Dada a tendência das grandes organizações de oferecerem amplos serviços a todos os seus departamentos, seções e divisões, as soluções de software padrão estão cada vez mais em demanda. Contudo, à medida que a padronização aumenta os usuários cada vez mais demandam adaptações em estações de trabalho individuais que atendam suas necessidades específicas. *Frameworks* e padrões de projeto são uma resposta para isso e para uma padronização e flexibilidade de requisitos similares (FACH, 2001).

O interesse por padrões foi despertado pelo trabalho de um arquiteto, Christopher Alexander, cujos padrões englobavam conhecimento do projeto e construção de comunidades e edifícios. O uso da palavra “padrão” (*pattern*) assumiu mais significado que a definição usual do dicionário. Os padrões de Alexander consistiam da descrição de um padrão recorrente de

elementos arquiteturais e da regra para como e quando criar tal padrão. Padrões produtivos compartilham muitas vantagens dos padrões não gerativos. Eles fornecem uma linguagem aos projetistas que torna fácil planejar, falar sobre e documentar projetos. Têm a vantagem adicional de ser mais fáceis de usar por desenvolvedores não experientes, de acordo com Beck (1994).

Um *framework* pode ser descrito como uma arquitetura de software semi-definida que consiste de um conjunto de unidades individuais e de interconexões entre elas, de forma a criar uma infra-estrutura pré-fabricada para desenvolvimento de aplicações de um ou mais domínios específicos (GIMENES, 2000). Da mesma forma, os padrões permitem que a experiência de desenvolvedores de software, seja documentada e reutilizada, registrando-se soluções de projeto, para um determinado problema ou contexto particular (GAMMA, 2000).

Segundo Fach (2001), o conceito chave para o uso de *frameworks* é reuso de projeto. Em contraste à abordagem do passado que aplicava o termo reuso para funções individuais de software, o objetivo de *frameworks* é reusar unidades completas específicas do domínio, por exemplo, documentação de clientes, contas ou contas de segurança. Em outras palavras, tenta-se preservar o trabalho de desenvolvimento existente, tal como análise e projeto de classe de domínio, criando um arcabouço reduzido representando, por exemplo, a implementação de uma conta e seus componentes de interface. Desse modo, os programadores da aplicação necessitam somente adequar o arcabouço ao domínio específico de uma aplicação em particular. Para tornar clara a diferença: em abordagens tradicionais para reuso de software, cria-se um programa de aplicação usando bibliotecas de classe para acesso à base de dados, funções matemáticas e a interface com o usuário. Em contraste, reuso de projeto significa adaptar um envoltório completamente desenvolvido – para subclasse, por exemplo. Assim, em *frameworks*, a lógica do programa já está presente.

Quando do desenvolvimento de *frameworks*, primeiramente o foco inicial é um domínio de negócios relativamente pequeno e claramente estruturado. Em uma equipe constituída de usuários potenciais (clientes) e desenvolvedores, avalia-se as atividades típicas de trabalho tal como: objetos da tarefa, desempenho da tarefa e trabalho cooperativo. Com base nos modelos de tarefa resultantes da análise de tarefa, são projetados os *frameworks* de domínio (classes

para manter regras, por exemplo). Potenciais usuários são envolvidos não apenas durante a análise de tarefas, mas durante todo o processo de desenvolvimento (revisões e prototipagem). Esse modo de desenvolver software freqüentemente é referenciado como abordagem do usuário participativo, ou simplesmente como participação do usuário (FACH, 2001).

A partir de uma perspectiva técnica, *framework* é um conjunto de classes cooperativas (C++ ou Java) que é usado como uma unidade auto-contida. Contudo, o número de tais classes pode se tornar muito grande e a cooperação entre elas intricada, de modo que documentar a funcionalidade de um *framework* ou explicá-lo para novos membros de equipe, freqüentemente é difícil. Padrões de projeto são metáforas que descrevem o comportamento e estrutura de um único *framework* ou a ação recíproca entre dois ou mais *frameworks* e ilustram conceitualmente como funcionam. Por outro lado, padrões de projeto oferecem excelentes oportunidades para participação do usuário no processo evolucionário de desenvolvimento de *frameworks*, onde são um instrumental para melhorar os pré-requisitos em um sistema altamente reusável (FACH, 2001).

Nos métodos de desenvolvimento tradicional de software, a função dos guias de estilo tem se tornado progressivamente importante, assim como as aplicações são cada vez maiores. A razão é óbvia: o número de programadores de aplicações envolvidos nos projetos está aumentando; novos componentes estão sendo adicionados constantemente e há o perigo de se perder o olhar consistente e não ser possível vê-lo crescer continuamente. Guias de estilo para grandes aplicações *Windows* ou aplicações baseadas em multimídia normalmente contêm mais de 500 (quinhentos) páginas. Experiências têm mostrado que engenheiros de software raramente usam tais documentos volumosos (FACH, 2001).

Para Fach (2001), guias de estilo para desenvolvimento de *frameworks* devem considerar duas questões:

- Critério para reuso. É importante para definir precisamente as situações nas quais os desenvolvedores podem reusar uma ferramenta em particular. Por exemplo, um catálogo deve listar todas as ferramentas disponíveis para reuso junto com os pré-requisitos para usá-las. Por outro lado, há o perigo de que as ferramentas possam ser usadas fora do contexto pretendido, uma vez que foram

implementadas anteriormente e estão prontas para reuso. Experiências têm mostrado que problemas surgem nesses casos, dado que adaptar ferramentas para uso de médio e curto prazo neutraliza os benefícios de reuso e desestabiliza os negócios ou mesmo camadas de configurações da aplicação.

- Modelos de uso. Um modelo de uso é um conjunto de todas as metáforas que indicam como usar uma aplicação. O modelo de uso deve ser padronizado no início do processo de desenvolvimento, do contrário, desenvolvedores podem de modo não intencional, usar modelos diferentes, tal como constroem simultaneamente ferramentas para vários domínios de negócio. Na pior das hipóteses, isso pode requerer a modificação de *frameworks* do domínio da aplicação, atrasando ou mesmo alterando o processo de maturação (sedimentação) do *framework*.

Quando da construção de *frameworks* é necessário alterar nomes dos itens do menu, botões de ação, atalhos e assim por diante, dado que um novo domínio de aplicação pode solicitar uma terminologia diferente. Contudo, é possível restringir os guias de estilo de *frameworks* por critério de reuso, modelos de uso e convenções de nomes. Em um determinado contexto, por exemplo, o guia de estilo pode estar disponível em um único arquivo.

Valério (1997) conceitua *frameworks* como sendo macro-componentes desenvolvidos como sistemas de objetos, tais como classes, que cooperam modelando uma funcionalidade ou uma característica relevante de um domínio. Arquiteturas de software representam o modelo de alto nível das aplicações em um domínio e são compostas pelos *frameworks* que modelam as funcionalidades expostas pelas aplicações. Os componentes reusáveis são agrupados em *frameworks*, a fim de implementar a funcionalidade modelada pelo *framework* e permitem adaptar e adequar as aplicações de acordo com as necessidades do cliente.

Outro conceito que, conforme Valério (1997) está rapidamente mostrando sua difusão na indústria de software é padrão (*pattern*). Um padrão é ‘o meio repetido ou regular no qual alguma coisa acontece ou é feita’ ou ‘um arranjo (organização) de entidades ou formato no qual a mesma entidade ou formato é repetido de um modo regular’ ou ainda, ‘um modelo ou diagrama que pode ser usado como um guia quando se está fazendo alguma coisa ou lidando

com algum problema'. Um padrão de projeto, seguindo a abordagem de Alexander, é uma solução geral e demonstrada para um problema específico em um contexto definido com limites e restrições.

Para Valério (1997), essas técnicas são a expressão de uma reação que retrata o interesse em reuso de software a partir do código e de pequenos componentes para artefatos abstratos e de alto nível que incorporam uma parte relevante de conhecimento e projeto no ciclo de vida de software. Essa busca contínua quanto às práticas de reuso para uma melhor compensação entre esforço de desenvolvimento e retorno sobre o investimento influencia o processo de software básico.

Esse processo de software é baseado na produção de *frameworks* e, em geral, libera para o cliente, aplicações compostas pelo agrupamento de artefatos reusáveis de acordo com a arquitetura de domínio. Sua institucionalização fomenta uma mudança na organização acerca do modelo de 'fábrica de software'. *Frameworks*, arquiteturas e padrões são um meio privilegiado para valorizar a experiência obtida no ciclo de vida do software, formalizando questões e soluções de projeto (VALÉRIO, 1997).

A conjunção de reuso de software, *frameworks* e padrões, tem muitos efeitos na visão de Valério (1997), a saber:

- Desenvolver aplicações de software por meio da composição de *frameworks*, seguindo uma arquitetura de referência e agrupando os componentes reusáveis nos *frameworks*, aumenta a eficiência das práticas de reuso reduzindo, ao mesmo tempo, o esforço de desenvolvimento.
- *Frameworks* e arquitetura intensificam a modularidade e a estrutura das aplicações, encapsulando os detalhes de implementação dentro de componentes permutáveis criando o princípio de localidade; além disso, o conceito de 'interface' contribui para a divisão em módulos.
- Aplicar padrões de projeto e usar *frameworks* e arquiteturas comprovadas, melhora a robustez e a qualidade da aplicação; além disso, é possível também que pessoas com pouca experiência em um campo específico produzam uma boa

aplicação seguindo padrões de projeto disponíveis publicamente e utilizem *frameworks* e arquitetura "*off-the-shelves*".

- *Frameworks* e arquiteturas promovem interoperabilidade e comunicação entre diferentes produtos.

Embora não seja uma relação completa, esses são alguns dos benefícios que *frameworks*, arquiteturas e padrões podem explorar quando aplicados no processo de software para apoiar reuso de software. *Frameworks* e padrões podem fornecem relevantes benefícios, quando usados em um processo de software e integrados a um amplo e estruturado programa de reuso (VALÉRIO, 1997).

Os esforços para alcançar reuso de software, deixaram, na concepção de Jan (2000), duas importantes lições de aceitação geral:

- Reuso oportunista, algumas vezes referenciado como 'código de salvação', não é efetivo na prática; um programa de reuso bem sucedido deve ser uma tentativa pró-ativa e bem planejada dentro de uma organização.
- Reuso *bottom-up*, isto é, a composição arbitrária de componentes para construir sistemas, não funciona na prática; programas de reuso bem sucedidos são requeridos para empregar uma abordagem *top-down*, isto é, são desenvolvidos componentes que se ajustam na estrutura de alto nível definida por uma arquitetura de software. Contudo, particularmente na presença de componentes que necessitam ser incorporados à uma linha de produto, o processo pode consistir também de algumas decisões de projeto *bottom-up*. Por exemplo, a decisão de utilizar um componente construído por terceiros, tal como um protocolo de comunicação, terá efeitos no projeto da arquitetura de linha de produto.

Na visão de Herzum (2000), qualquer abordagem séria para construção de software deve ser centrada em reuso. Isso significa que o processo de desenvolvimento, boas práticas, arquitetura e a fábrica de software participam (todos) diretamente do suporte aos altos níveis de reuso. A distinção entre usar e reusar um artefato de software pode parecer arbitrária. Na abordagem de componentes de negócios, o simples fato de se estabelecer uma arquitetura e

processo de desenvolvimento apropriado encoraja equipes ao uso e reuso de componentes, modelos, padrões (*patterns*) e outros artefatos de software produzidos por outras equipes. Nesse sentido, reuso é uma preocupação constante e presente diariamente na vida de uma organização, em termos de efetividade de custo, ainda que não haja um programa explícito de reuso.

Um programa de reuso em um avançado negócio de desenvolvimento é o conjunto de princípios e ferramentas que estão especifica e explicitamente relacionados com suporte a reuso. No desenvolvimento normal de software efetivo, do ponto de vista de custo, reuso é um aspecto complementar importante, em um programa de reuso, é o principal objetivo.

Com essa distinção em mente, é melhor ter como alvo um programa de reuso após ter obtido pelo menos o marco principal de funcionamento de uma fábrica. Por exemplo, em geral, não faz sentido propor para reuso o desenvolvimento de um componente para um projeto, exceto se o projeto que se deseja reusar compartilha muitas das mesmas características de funcionamento da fábrica, nem faz sentido estabelecer uma equipe explícita de reuso antes de a organização ter adotado uma metodologia comum.

Reuso não é apenas uma questão de tecnologia, arquitetura e processos de desenvolvimento, mas também, e mais importante, é uma questão organizacional. Dois aspectos organizacionais devem ser tratados: Como evoluir o desenvolvimento da organização para atividades de reuso e como montar uma organização que irá conduzir reuso, por exemplo, projetar e administrar o repositório de bens reusáveis. A organização também requererá novas funções as quais incluem o gerenciamento de programa de reuso, e possivelmente, segurança de reuso – talvez entendida como uma ou mais pessoas responsáveis por cuidar de artefatos de software existentes e alçá-los ao nível de qualidade requerido pelo reuso. Outras funções são relacionadas ao repositório de artefatos, por exemplo, gerenciamento do repositório de artefatos (responsável para a organização funcionar com sucesso, pelo uso do repositório e por sua evolução de longo prazo); administração de repositório de artefatos (responsável diário pela operação do repositório) e; desenvolvimento do repositório de artefatos (fornecer as instalações e evolução técnica do repositório de software em si e também de seu possível

desenvolvimento). Essas funções podem não requer uma pessoa em tempo integral e várias funções podem estar ao encargo de uma mesma pessoa (HERZUM, 2000).

A tendência para produtividade na indústria de software, está forçando importantes mudanças no modo como o desenvolvimento e manutenção de software têm sido realizados. Mas isso requer uma abordagem holística para o software e para o processo de software. Na realidade, visão holística da organização que incorpora cultura, pessoas e procedimentos, é requerida a fim de melhorar a qualidade do processo.

De acordo com Butler (1997), as tendências técnicas que mais contribuem para melhorar a produtividade são: a melhoria da qualidade de produtos e de processos a fim de reduzir o custo de manutenção; o reuso de código e outros componentes de software, tais como projetos e requisitos, a fim de reduzir custos de produção e melhorar a qualidade de componentes individuais e a aceitação ampliada de notações de modelagem e ferramentas, para fornecer um modelo de software – na realidade, diferentes modelos, oferecendo diferentes perspectivas – como um auxílio essencial para o entendimento do software, reduzindo custos de desenvolvimento, manutenção e solução.

As vantagens competitivas que uma organização alcança com o reuso são: tempo de entrega melhorado, conhecimento aprimorado do domínio de aplicações e, produtividade melhorada. A introdução bem sucedida de reuso requer que o processo de desenvolvimento na organização seja bem definido, isto é, nível três escala do CMM, de acordo com Butler (1997).

A preocupação com o cronograma – tempo – prazos, custos, com a não descoberta de defeitos antes que software seja liberado ao cliente e, com as dificuldades em medir, monitorar e acompanhar o progresso enquanto o software está sendo desenvolvido, suscitou uma grande preocupação com o software e com a maneira como o mesmo é desenvolvido, o que leva à adoção de práticas de engenharia de software (PRESSMAN, 1997). Além disso, a demanda por produtos e serviços de qualidade tem suscitado o estudo e a proposição de métodos,

técnicas e ferramentas cujo objetivo principal é auxiliar a construção de produtos de qualidades ao mesmo tempo, melhorando o seu processo de construção.

É com essa preocupação, e considerando um contexto mais específico, que é o ambiente de fábrica de software, que uma idéia de contribuir para com a área de engenharia de software no tocante a melhoria da qualidade de produto e de processo, amadureceu, tomou forma e se converteu em uma proposta concreta de melhoria.

A fundamentação teórica apresentada, não apenas legitima como **justifica** a iniciativa de desenvolver uma abordagem para melhoria da qualidade, dentro do contexto especificado. Um mercado carente e exigente, que demanda cada vez mais produtos de alta qualidade e a preocupação crescente das organizações em se manterem competitivas dentro desse mercado, por meio da construção e comercialização de produtos de alta qualidade e de baixo custo, aliados ao constante empenho da comunidade da área em busca de soluções para atender a esses dois segmentos, são razões suficientes para motivar e balizar o trabalho ora proposto.

6. ABORDAGEM PROPOSTA

Atualmente, organizações que buscam vantagens competitivas têm investido pesadamente na automação de seus processos de negócio. Grande confiança é colocada nos produtos de software, a ponto de o software assumir um papel crítico e estratégico nos negócios da organização. Com esse nível de importância e a confiança depositada nos produtos de software, tornou-se uma necessidade melhorar a qualidade dos produtos de software. Também é necessário melhorar a eficiência e a produtividade do processo de desenvolvimento e de manutenção dos produtos. Assim, pesquisadores e profissionais têm aumentado a atenção dada para entender e melhorar a qualidade dos produtos de software. Alguns estudos têm se concentrado nas técnicas e abordagens, para assegurar a qualidade de produtos de software; enquanto outros têm focado o processo de desenvolvimento de software, como defini-lo, avaliá-lo e melhorá-lo (WONG, 2004).

A grande preocupação das empresas, entidades de classe e do governo com aspectos de qualidade e o constante empenho da comunidade acadêmica da área em pesquisar tal tema, são os principais fatores a impulsionar e legitimar o trabalho em questão.

Nos capítulos anteriores, foram abordados alguns dos temas que têm influência direta na qualidade de software e, portanto, contribuição determinante na abordagem ora proposta.

O estudo realizado apontou para duas atividades que podem ser consideradas como de grande influência na qualidade do produto e do processo, além da prática sistemática de reuso. Tais atividades - inspeções e teste - se levadas a efeito sistematicamente, fornecem um indicativo para que uma empresa consiga determinados níveis de maturidade, tomando como base, por exemplo, o modelo CMMI, e, por conseguinte, se realizadas em conjunto, podem, melhorar a qualidade do produto e do processo por meio do qual o produto é desenvolvido, possibilitando assim, que os níveis de qualidade almejados sejam atingidos. O uso combinado das atividades de teste e inspeção, aliado à prática sistemática de reuso no contexto de fábrica de software vem ao encontro do que propõe a abordagem de fábrica proposta pela Microsoft, que enfatiza

a necessidade de maximizar o reuso de software visando à melhoria da qualidade, bem como, para possibilitar o aprendizado por meio de projetos já realizados e, consequentemente, a transferência de conhecimento entre projetos.

6.1 DESCRIÇÃO DA ABORDAGEM PROPOSTA

Como explicitado anteriormente, a abordagem proposta compreende o uso combinado e sistemático das atividades de teste e inspeção e do reuso, para a melhoria da qualidade de software, no contexto de fábrica de software.

A Figura 41 ilustra a proposta de metodologia apresentada neste trabalho. Ao longo do processo de desenvolvimento, são realizadas – de forma combinada e sistemática - as atividades de inspeção e teste, visando à remoção de defeitos e à melhoria da qualidade do processo, dos artefatos intermediários - gerados em cada fase do processo – e, consequentemente, do produto final. Tal como o ocorre com as atividades de teste e inspeção, o reuso sistemático é realizado durante todo o processo de desenvolvimento.

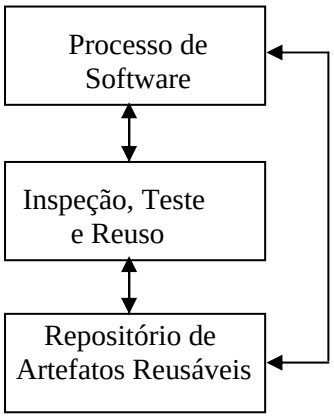


Figura 41. Modelo da proposta de abordagem para controle de qualidade

Conforme pode ser observado na Figura 41, existe um repositório de artefatos para apoiar a metodologia proposta. Trata-se de um repositório de artefatos reusáveis, gerados em cada fase do processo de engenharia de software, incluindo, documentos de especificação, documentos de análise, dados de inspeções, dados de teste, código etc. Este repositório contém os artefatos apropriados, a serem reusados em cada fase do processo de desenvolvimento, de acordo com as suas especificidades. O repositório de artefatos é, portanto, um repositório de produtos software e de quaisquer outros artefatos derivados durante as fases de desenvolvimento e que possam ser reutilizados. A idéia é que este repositório seja de fácil acesso a todos que dele necessitem. Além disso, neste repositório também estarão disponíveis artefatos já existentes, derivados de projetos anteriores que podem ser reusados no desenvolvimento do projeto atual.

De acordo com o que mostra a Figura 41, há uma seta – a menor - de dupla direção entre o repositório e o processo onde ocorrem as atividades de inspeção e teste e a prática de reuso, a qual indica que durante a realização dessas atividades e prática, o repositório de artefatos é alimentado com os artefatos criados durante o ciclo de vida, bem como, que o repositório provê a essas atividades artefatos para serem reusados. O teste, comprovadamente uma das atividades mais efetivas para a melhoria da qualidade do produto e, além disso, dados resultantes da atividade de teste proporcionam boa indicação da confiabilidade do software como um todo. A inspeção, por outro lado, é uma das atividades mais importantes para a melhoria da qualidade de software uma vez que atua como um fator de melhoria tanto no produto como no processo por meio do qual é desenvolvido. A atividade de inspeção possibilita aos envolvidos no processo de inspeção, analisar o produto e qualquer documentação de análise, desenvolvimento e manutenção relacionada ao produto em inspeção, inclusive os artefatos gerados pela atividade de teste.

A seta maior, também de dupla direção, mostra a interação entre o processo e o repositório de artefatos reusáveis, indicando que durante o processo de desenvolvimento é possível utilizar artefatos derivados de projetos anteriores para serem integrados no projeto atual. A dupla direção das setas (seta menor) tem como propósito mostrar que há uma constante interação entre o processo, as atividades para melhoria da qualidade, a prática de reuso e o repositório, contudo, para se ter acesso ao repositório, não é necessário passar pela inspeção e teste e/ou reuso (seta maior). Na realidade, é possível acessar o repositório a qualquer momento durante

o processo de desenvolvimento, tanto para agregar novos artefatos ao repositório como para recuperar algum artefato já existente no repositório para ser utilizado no momento que se julgar necessário e adequado. A idéia é que seja possível agregar os artefatos intermediários gerados e devidamente testados e inspecionados no momento a qualquer momento durante o processo de desenvolvimento, ou seja, não é necessário chegar ao término do processo de desenvolvimento para poder acessar o repositório, para agregar ou recuperar artefatos.

Na metodologia proposta, o uso combinado das atividades de inspeção e teste, bem como a prática de reuso devem ser realizadas em paralelo e sistematicamente, funcionando como atividades guarda-chuva, cobrindo todo processo de desenvolvimento. As setas de dupla direção indicam que há interação entre o processo, as atividades de inspeção e teste, e o repositório de artefatos reusáveis. Essa interação mostra também, que durante a realização das atividades para assegurar qualidade os repositórios são alimentados e há um *feedback* que possibilita a comunicação, a troca de experiências entre os envolvidos no processo por intermédio das atividades, do repositório e das etapas do ciclo de vida. Essa interação e *feedback*, possibilitam a detecção e remoção de defeitos, a análise, a classificação e a catalogação de artefatos de software. Além disso, ao mesmo tempo em que se alimenta o repositório com artefatos intermediários gerados nas diversas fases do processo, também é possível utilizar os artefatos já existentes, no processo atual de desenvolvimento.

Cabe ressaltar, que a atualização do repositório de artefatos reusáveis, deve ser realizada e avaliada durante e ao término de cada fase do processo de desenvolvimento, visando assegurar a pertinência e a qualidade do seu conteúdo, dado que os objetivos do repositório de artefatos são: reduzir custos e aumentar a produtividade de desenvolvimento (HERZUM, 2000).

Na abordagem proposta é imprescindível que seja incorporada a prática sistemática de reuso às atividades de inspeção e teste para melhoria da qualidade de software, não cabendo, de forma alguma dentro dessa abordagem, a prática oportunista – a prática de conveniência - de reuso, inspeção e teste.

Essa proposta não está vinculada a nenhum modelo para construção de software. A idéia é que o processo possa ser instanciado por qualquer dos paradigmas de engenharia de software existentes, dado que o processo independe do tipo de aplicação a ser desenvolvido bem como da abordagem utilizada. A metodologia proposta deve ser capaz de alcançar a melhoria da qualidade almejada, independentemente de se utilizar um modelo estruturado, orientado a objetos ou baseado em componentes, para o desenvolvimento de uma determinada aplicação. À organização responsável pelo desenvolvimento caberá a decisão quanto à utilização de um paradigma de engenharia de software que seja o mais adequado possível às suas necessidades. Desse modo, a metodologia proposta deve atender com igual eficiência o desenvolvimento de projetos instanciados desde o modelo de vida clássico até os mais atuais como RUP - *Rational Unified Process* (KRUCHTEN, 2003) e Desenvolvimento Baseado em Componentes (CHEESMAN e DANIELS, 2001), dentre outros.

Cada uma das atividades para assegurar qualidade por si só, possui vantagens que justificam e requerem o seu uso. Contudo, o uso conjunto, integrado e em paralelo de ambas, potencializa a vantagem de cada uma delas dado que são complementares e cobrem todo o ciclo de vida, aumentando, cada uma, com procedimentos, ferramentas e planejamentos distintos, a confiança do usuário no produto a ele liberado. Tanto os testes como as inspeções funcionam como aprimoramento do produto, entretanto, as inspeções funcionam também na melhoria do processo (WEI, 1996).

6.2 COMO UTILIZAR A METODOLOGIA PROPOSTA

Como já visto anteriormente, as empresas que adotam a abordagem de fábrica para desenvolvimento de seus produtos e serviços, devem realizar mudanças culturais e estruturais. As mudanças culturais e estruturais referem-se ao compromisso assumido com o desenvolvimento de produtos e serviços, dentro dos padrões de qualidade desejados, desenvolvendo de linha de produto e adotando a prática de reuso sistemático para obter economia de escala e de escopo, adaptando e integrando componentes padrões para produzir

produtos similares, porém distintos, padronizando e automatizando seus processos de produção, apoiadas por ferramentas extensíveis que possam ser configuradas para automatizar tarefas repetitivas.

As empresas que adotam a abordagem de fábrica de software para o desenvolvimento de seus produtos e serviços, devem seguir alguns passos sobre como utilizar a metodologia proposta neste trabalho, de modo a assegurar o uso bem sucedido e, consequentemente, obter os resultados, passos esses descritos a seguir.

Deve haver um planejamento, antes da execução da atividade de inspeção. Deve-se primeiramente, definir o tipo de inspeção a ser realizada, podendo ser a inspeção por pares, conforme incentivada pelo próprio CMMI. Após a definição do tipo de inspeção que será realizado, deve-se:

- Definir a equipe de profissionais envolvidos nesta atividade, a qual deve analisar as representações dos artefatos e indicar os possíveis erros.
- Preparar uma possível lista de prováveis erros para orientar o processo de inspeção, a qual deve ser estabelecida no início do processo de inspeção e atualizada à medida que mais experiência vai sendo adquirida no processo de inspeção.
- Caso haja necessidade, investir em treinamento de equipes para realizar a atividade de inspeção.
- Definir que artefatos deverão ser inspecionados primeiramente. Deve-se estabelecer uma prioridade para a realização de inspeção caso haja vários artefatos a serem inspecionados. Contudo, o ideal é que cada artefato seja inspecionado logo após serem gerados.
- Garantir que os artefatos a serem inspecionados estejam completos e bem especificados.
- Apresentar à equipe de inspeção os artefatos a serem inspecionados, para que a equipe possa estudar o material a ser inspecionado.
- Registrar e manter atualizada toda a documentação resultante da atividade de inspeção.

- Identificar e indicar o estado dos artefatos inspecionados, indicando se estão livres de erros, se precisam ser modificados, se necessitam ser re-inspecionados, se devem ser descartados etc.
- Estabelecer um mecanismo de controle para a atividade de inspeção, por projeto e por artefato.
- Realizar um relatório da atividade de inspeção, o qual deve ser divulgado a todos os envolvidos no processo de inspeção.
- Registrar todas as informações relacionadas ao processo de inspeção, visando a formação os Estabelecer uma linha básica.

A essa lista, é possível adicionar outras atividades que podem ser levadas a efeito visando à realização do processo de inspeção, baseado na experiência da empresa.

Assim como no caso da atividade de inspeção, a atividade de teste deve ser planejada antes que seja iniciada a sua execução. Uma empresa que adota a abordagem de fábrica de software para o desenvolvimento de seus produtos e serviços, deve seguir as seguintes etapas do processo de teste:

- Planejamento da atividade de teste, onde são definidos o escopo, os objetivos e a abordagem de teste a ser utilizada, resultando um documento de plano de teste. O teste deve ser planejado de modo que todos os requisitos sejam individualmente testados.
- Projeto de teste, envolvendo o projeto dos casos de teste resultando no início da das especificações dos casos de teste.
- Preparação e execução do teste, envolvendo a preparação do ambiente de teste, a execução dos casos de teste e a descoberta de erros, tendo como resultado, relatórios de erros.
- Avaliação e melhoria do teste, onde os resultados do teste são avaliados e é gerado um relatório sucinto de teste.

Uma vez concluído o planejamento da atividade de teste, é fundamental:

- Definir a equipe de profissionais envolvidos nesta atividade, a qual deve analisar as representações dos artefatos e indicar os possíveis erros.
- Estabelecer um cronograma completo de teste e alocar recursos para esse cronograma.
- Estabelecer técnicas, critérios e métodos para elaborar casos de teste.
- Todos os produtos de software a serem testados devem ser especificados.
- Conhecer o software, sua função, suas entradas, como essas entradas podem ser combinadas e o ambiente no qual o software opera.
- Definir a estratégia de teste a ser utilizada para a realização da atividade de teste.
- Definir a seqüência de atividades de teste.
- Preparar uma possível lista de prováveis erros para orientar o processo de teste.
- Caso haja necessidade, investir em treinamento de equipes para realizar a atividade de teste.
- Caso seja necessário, definir a prioridade dos artefatos a serem testados. Contudo, o ideal é que cada artefato seja testado logo após serem gerados.
- Garantir que os artefatos a serem testados estejam completos e bem especificados.
- Apresentar à equipe de teste os artefatos a serem testados.
- Registrar e manter atualizada toda a documentação resultante da atividade de teste.
- Identificar e indicar o estado dos artefatos testados. Por testar, já testados etc.
- Estabelecer um mecanismo de controle para a atividade de teste, por projeto e por artefato.
- Realizar um relatório da atividade de teste.
- Manter sempre a documentação atualizada dos testes realizados.
- Registrar sistematicamente os resultados dos de modo que seja possível inspecionar o processo de teste para averiguar se o mesmo foi realizado corretamente e, ao mesmo tempo, formar uma linha básica de dados.
- Especificar a utilização de ferramentas de hardware e software requeridas.
- Prever as restrições que possam vir a afetar o processo de teste (escassez de recursos humanos, por exemplo).

É importante salientar que essa linha mestre é apenas uma sugestão de passos a serem seguidos para a realização das atividades de inspeção e teste. Cada empresa pode, de acordo com suas especificidades, necessidades e experiências anteriores, estabelecer sua própria diretriz para a aplicação da metodologia proposta.

A abordagem de metodologia proposta está alinhada à abordagem de fábrica de software, fazendo uso de suporte computacional para enfatizar o apoio á fabrica de software por meio da utilização do X-ARM, um modelo para representação de *assets* que possibilita a representação de informações sobre os diversos aspectos de funcionalidades de um repositório de componentes reusáveis e da X-Packager, uma ferramenta baseada - no modelo X-ARM - que possibilita a descrição e empacotamento de *assets* em pacotes únicos, os quais são disponibilizados no repositório, devidamente classificados e certificados.

O modelo X-ARM, o UML *Components* e a ferramenta X-Packager são descritos a seguir.

6.3 AVALIAÇÃO DA ABORDAGEM PROPOSTA

Foi realizado um estudo de caso para avaliar a metodologia proposta, à qual foi aplicado o UML *Components*, uma abordagem para construção de software baseada em componentes, proposta por Cheesman e Daniels (2001). Para apoiar o processo de avaliação da metodologia proposta, foi utilizada a ferramenta X-*Packager* (SANTOS, 2006) - baseada no modelo X-ARM - a qual tem como finalidade auxiliar na descrição e empacotamento de *assets*[33], que compõem o repositório de artefatos reusáveis.

Contudo, antes de descrever os procedimentos utilizados para a realização do estudo de casos, se faz necessária uma descrição do UML *Components*, do modelo X-ARM, bem como da

[33] O termo *assets* é o utilizado por Santos (2006), de forma equivalente a artefatos neste trabalho de tese.

ferramenta X-*Packager*, os quais são elementos chave para a realização do estudo de caso e, consequentemente, para a avaliação da abordagem proposta.

6.3.1 Uml components

O processo de desenvolvimento baseado em componentes difere das abordagens anteriores por separar especificação de componentes de implementação e por dividir a especificação de componentes em interfaces (CHEESMAN e DANIELS, 2001). Dividir especificação de componentes em uma ou mais interfaces significa que as pendências inter-componentes podem ser restringidas a interfaces individuais, ao invés de abranger a especificação do componente como um todo.

O UML *Components* (CHEESMAN e DANIELS, 2001) é uma abordagem para desenvolvimento baseado em componentes, que compreende vários *workflows*, conforme mostra a Figura 42.

A escolha dessa abordagem se justifica pelo fato de que a mesma foi utilizada para avaliar e apresentar um exemplo da utilização do modelo X-ARM e da ferramenta X-Packager (SANTOS, 2006), por meio da descrição e do empacotamento de um conjunto de assests produzidos utilizando o processo de desenvolvimento UML *Components* (CHEESMAN e DANIELS, 2001). Desse modo, para que fosse possível avaliar os resultados da aplicação da metodologia proposta, optou-se por utilizar a mesma abordagem para que fosse possível avaliar os resultados bem como para estar em conformidade com o exemplo de uso utilizado.

Embora o modelo compreenda os seis *workflows* representados na Figura 42, nessa abordagem a ênfase está colocada nos três estágios do *workflow* de especificação, os quais são descritos em detalhe em Cheesman e Daniels (2001), além de uma descrição sucinta dos

workflows provisioning e *assembly*. Contudo, não é feita nenhuma descrição relacionada aos *workflows* teste e *deployment*.

O modelo apenas menciona o *workflow* de teste, o que pode significar uma fragilidade, já que por se tratar de desenvolvimento baseado em componentes, grande ênfase deve ser colocada nos testes de unidade, no teste de integração e no teste de sistema. Cada componente deve devidamente testado e inspecionado, assim como deve ser feita a inspeção e o teste das interfaces durante a integração dos componentes e, por fim, com o devido planejamento deve ser realizada a atividade de teste para avaliar a aplicação como um sistema. Essa sistemática é condição para o uso da metodologia ora proposta.

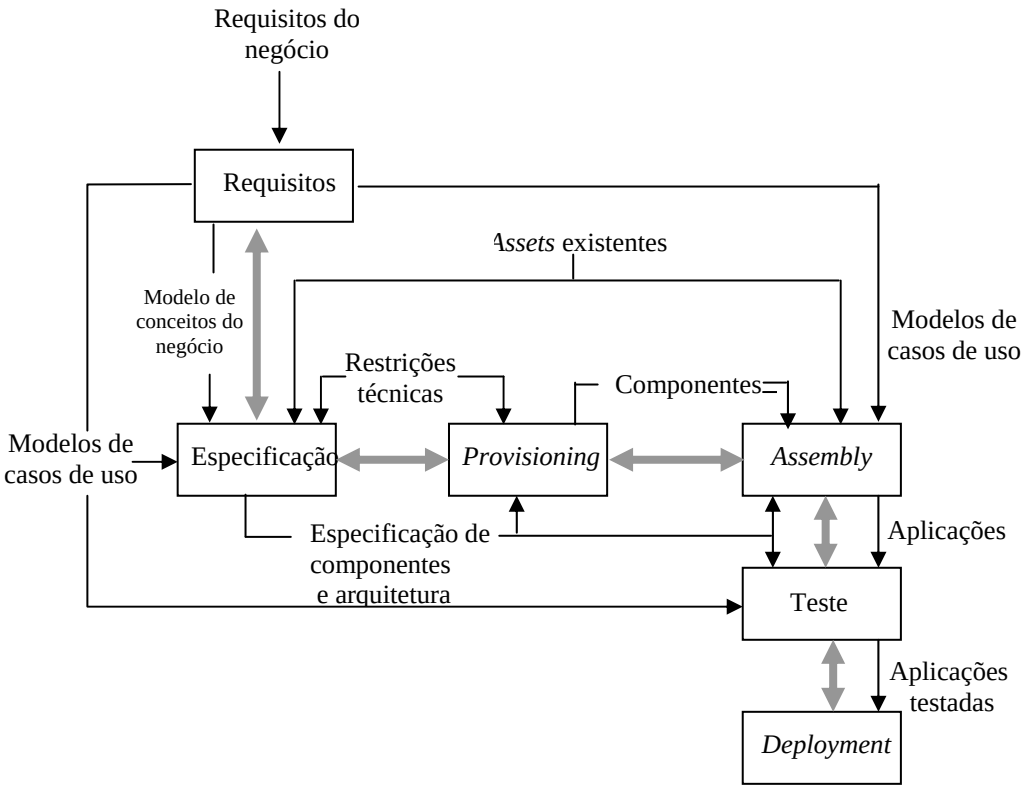


Figura 42. *Workflows* do processo de desenvolvimento (CHEESMAN e DANILES, 2001)

O *Workflow* de requisitos deve liberar para o *workflow* de especificação, um modelo de conceitos do negócio e um conjunto de casos de uso. O modelo de conceitos do negócio lista

os conceitos importantes no domínio do problema e mostra as relações entre eles. Os casos de uso clarificam a fronteira do software, identificando os atores que interagem com o sistema e descrevem essas interações.

O *workflow* de especificação gera uma arquitetura inicial de componentes. No *workflow* de especificação, os componentes podem ser especificados a partir do modelo de casos de uso, do modelo de conceitos do negócio ou de *assets* já existentes, levando em conta as restrições técnicas do projeto.

O propósito do *workflow provisioning* é gerar implementações de componentes, tanto implementando diretamente a especificação ou encontrando um componente existente que preencha a especificação desejada. Embora a abordagem proposta (CHEESMAN e DANIELS, 2001) postule que as especificações de componentes são independentes tanto quanto possível de tecnologias, afirma também que algumas técnicas de especificação não são diretamente apoiadas por todas as tecnologias que podem ser utilizadas, de modo que certa quantidade de interpretação é requerida.

Quando se fala em tecnologias que podem ser utilizadas, está se referindo principalmente ao ambiente de execução do componente. Também inclui linguagem de programação, mas com importância secundária. Neste contexto, ambiente de componente é um ambiente distribuído, junto com um conjunto de regras com as quais os componentes devem estar em conformidade, e um conjunto de serviços de infra-estrutura (suporte a transação, segurança, concorrência) dos quais os componentes da aplicação podem depender. Posteriormente, esse ambiente é restringido para ambientes de componentes que fornecem esses serviços de infra-estrutura declarativamente, usando uma abordagem de *framework* – algumas vezes, denominada programação baseada em atributos.

Integração (*Asembly*), segundo Cheesman e Daniels (2001, p. 164), é um processo de juntar componentes e *assets* existentes em um sistema de trabalho e descrever uma interface com o usuário, baseado nos casos de uso para formar uma aplicação. Integração compartilha muitas características com práticas de gerenciamento de configuração padrão. Cada componente ou *asset* existente integrado, pode ser visto como um item de configuração separado, sob

controle de versão. A arquitetura do componente representa a definição de configuração do sistema – quais são os itens de configuração requerida e quais são suas dependências. Integração também é responsável pela definição da interface com o usuário e a lógica do diálogo com o usuário, a qual é estabelecida de acordo com os casos de uso no modelo de casos de uso. Após a aplicação é passada para o *workflow* de teste, para teste de aceitação e teste de sistema.

O UML *Components* (CHEESMAN e DANIELS, 2001) é uma abordagem para desenvolvimento baseado em componentes, que, segundo Gimenes *et al* (2005, p. 28) "está baseada em uma clara separação entre a modelagem do domínio e a modelagem da especificação. Essa separação é essencial para o desenvolvimento de modelos e o claro entendimento de seus propósitos e significados".

Conforme explicitado em (GIMENES *et al*, 2005, p. 28), "a modelagem do domínio [...] é realizada para entender o contexto de uma situação ou negócio e pode ser desenvolvida independentemente do fato de se estar pretendendo utilizar um método DBC".

Os modelos resultantes da modelagem de domínio conforme mostra a Figura 43, são: modelo de casos de uso, modelo conceitual e modelo comportamental.

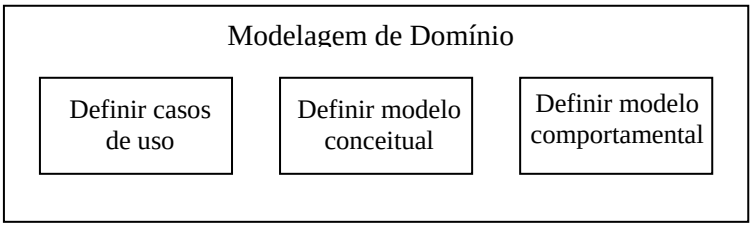


Figura 43. Modelos resultantes da modelagem de domínio (GIMENES *et al*, 2005)

Segundo Cheesman e Daniels (2001), a modelagem de especificação compreende três estágios os quais são mostrados na Figura 44.

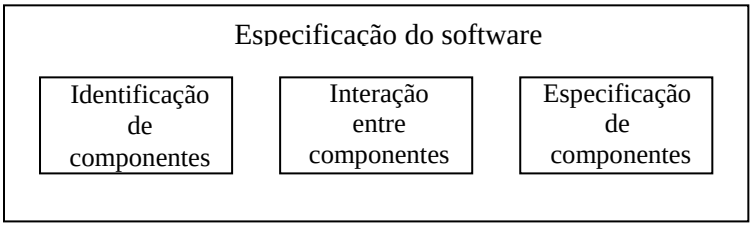


Figura 44. Estágios da Modelagem de especificação (GIMENES *et al*, 2005)

A modelagem de Especificação, de acordo com o estabelecido em (GIMENES *et al*, 2005, p. 33), “envolve a concepção de um sistema de software para resolver um problema”.

A identificação de componentes é baseada no modelo de conceitos do negócio e no modelo de casos de uso. Os modelos resultantes desse estágio são: o modelo de tipos do negócio, modelo de especificação e arquitetura inicial de componentes, modelo de interfaces do negócio, modelo de interfaces e operações do sistema.

A Figura 45 ilustra esses estágios juntamente com os respectivos artefatos, segundo o apresentado em Cheesman e Daniels (2001).

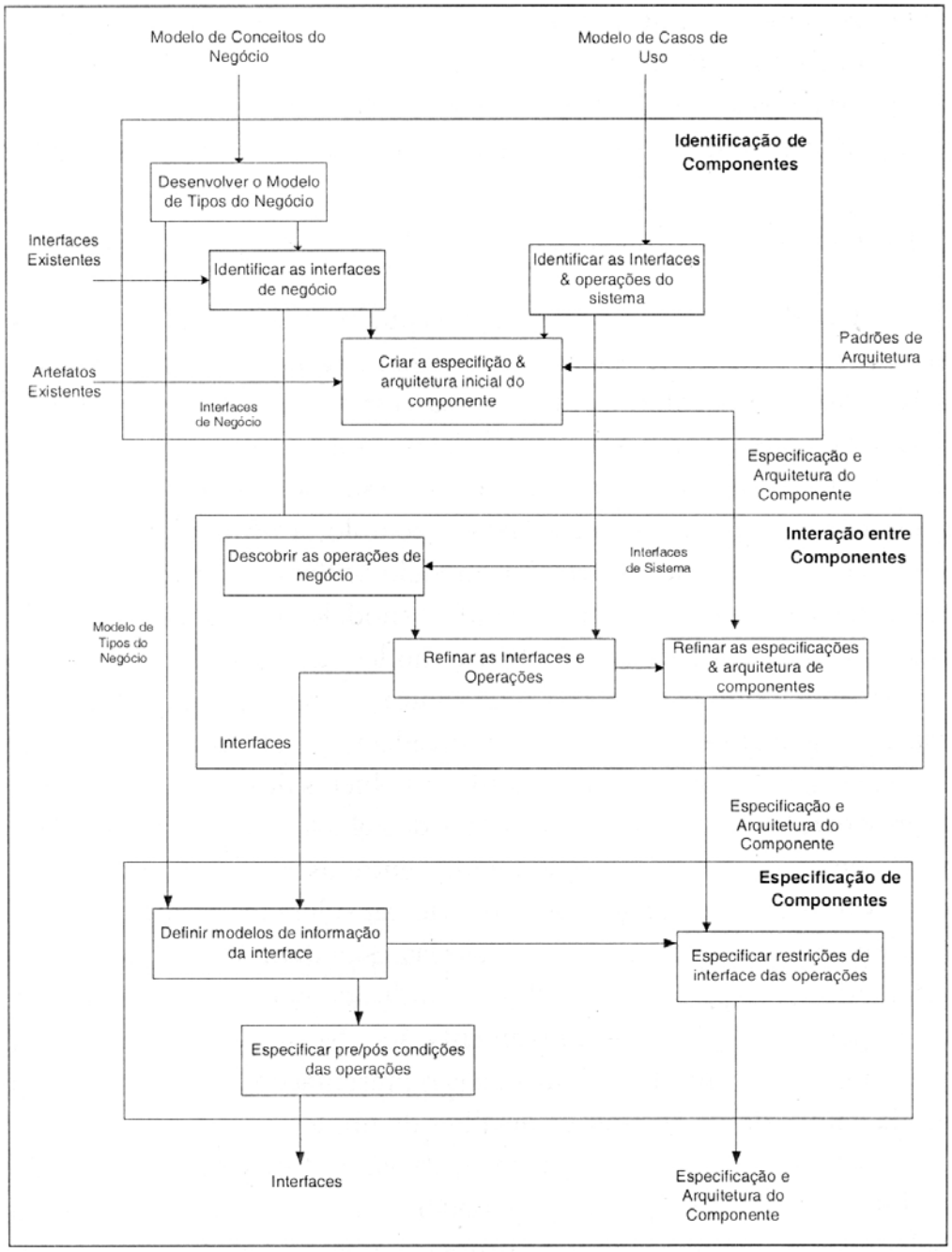


Figura 45. *Workflow* da modelagem de especificação[34]

[34] Fonte: (GIMENES *et al*, 2005, p.35)

No estágio de identificação de componentes são descobertos quais componentes são necessários para atender as funcionalidades do software. Neste estágio:

- Especificações iniciais de componentes são definidas e organizadas em uma arquitetura inicial de componentes.
- São identificadas as interfaces e os componentes do sistema – camada de serviços do sistema.
- São identificadas as interfaces e os componentes tipo do negócio – camada de serviço do negócio.
- É produzido um conjunto inicial de especificação de interfaces do sistema.
- É produzida uma especificação inicial de interfaces do negócio.
- Regras do negócio, assim como restrições, são capturadas no modelo de tipos do negócio.
- As interações de componentes são investigadas.
- É desenvolvido o modelo de tipos do negócio, que contém um conjunto de informações de tipos e regras do negócio, usado posteriormente, para desenvolver os modelos de informação de interfaces.

Os componentes identificados no estágio anterior fornecem um conjunto inicial de interfaces e componentes para este estágio. No estágio de interação, é necessário decidir como os esses componentes trabalharão juntos para cumprir as funcionalidades do sistema. Neste estágio são:

- Desenvolvidos os modelos de interação para cada operação de interface do sistema.
- Descobertas as operações de interfaces do negócio e suas assinaturas.
- Refinadas as responsabilidades.
- Definidas quaisquer restrições n arquiteturas de componentes.
- Fatoradas as interfaces e operações – dividir suas responsabilidades entre duas ou mais interfaces.

No estágio de especificação de componentes, a preocupação é com o contrato de uso, gerado partir da especificação de interfaces e com o contrato de realização derivado da especificação de componentes. Neste estágio deve-se:

- Derivar o modelo de informação de interface a partir do modelo de tipos do negócio.
- Especificar interfaces e operações. Uma interface é especificada por um conjunto de especificação de operações que operam em um modelo de informação de interface. Cada operação é especificada usando um par de pré e pós-condição. A pós-condição define os efeitos da operação, enquanto a pré-condição define as condições sob as quais as pós-condições são garantidas.
- Construir um modelo de informação para cada interface.
- Definir as pré e pós-condições.
- Especificar invariantes.
- Especificar interfaces do sistema.
- Especificar componentes, incluindo as especificações de interfaces fornecidas e requeridas.

As atividades resultantes de cada *wokflow* são apresentadas posteriormente, quando da descrição do estudo de caso realizado.

O modelo X-ARM e a ferramenta X-*Packager*, utilizada para apoiar a aplicação do modelo UML *Components* à metodologia proposta, são descritos a seguir.

6.3.2 O modelo x-arm

O que motivou a proposição do modelo X-ARM, "acrônico para o XML-*based Asset Representation Model*" (SANTOS, 2006, p.4), foi a escassez de abordagens para representação de metadados de *assets*. A proposição do modelo X-ARM, se apóia no fato de que poucos modelos de representação foram definidos com o propósito de descrever diferentes tipos de unidades de software, incluindo diversos artefatos utilizados no desenvolvimento baseado em componentes e também no fato de que as propostas existentes apresentam deficiências. Para embasar sua proposta, Santos (2006) descreve vários modelos, grande parte dos quais, baseados em XML, cujos conteúdos estão resumidamente representados na Tabela 19.

Tabela 19. Comparação dos modelos de representação de artefatos de software (SANTOS, 2006)

	RAS	OSD	CDML	Redolfi *et al*	X-ARM
Nomeação	Sim	Sim	Sim	Sim	Sim
Versionamento	Sim	Sim	Não	Sim	Sim
Requisistos não funcionais	Não	Sim	Sim	Sim	Sim
Interfaces providas	Sim (sem reuso)	Não	Sim (sem reuso)	Sim (sem reuso)	Sim (com reuso)
Interfaces requeridas	Não	Não	Sim (sem reuso)	Sim (sem reuso)	Sim (com reuso)
Eventos	Não	Não	Sim	Não	Sim
Propriedades	Não	Não	Não	Não	Sim
Modelos de negócios	Não	Não	Sim (rudimentar)	Não	Sim
Certificação do *assest*	Não	Não	Não	Sim (teste)	Sim
Certificação do produtor	Não	Não	Não	Não	Sim
Classificação	Sim (rudimentar)	Não	Não	Não	Sim
Informações sobre o processo de desenvolvimento	Sim (UML *Components*)	Não	Não	Não	Sim (Qualquer processo)
Composição de componentes	Não	Não	Não	Não	Sim
Controle de visibilidade	Sim (rudimentar)	Não	Não	Não	Sim

O objetivo do modelo X-ARM é representar os tipos de *assets* cuja descrição se faz necessária em um repositório de componentes (SANTOS, 2006, p. 5). Estão incluídas as descrições de componentes, de interfaces, de artefatos gerados ao longo do processo de desenvolvimento além dos arquivos de apoio ao funcionamento do repositório, como os próprios esquemas do

X-ARM. O X-ARM, tem também o objetivo de apoiar o desenvolvimento de componentes construídos a partir de quaisquer processos.

O modelo X-ARM foi desenvolvido com a finalidade de possibilitar a representação informações sobre os diversos aspectos das funcionalidades de um repositório.

> A representação de metadados pode ser utilizada em diversos aspectos da funcionalidade de um repositório de componentes, incluindo controle de visibilidade, suporte a processos de desenvolvimento, classificação, modelos de negócios, certificação, composição por outros componentes, requisitos não funcionais e representação de componentes de acordo com diversos modelos de componentes (SANTOS, 2006, p.17).

Conforme ilustrado na Tabela 20, no X-ARM os *assets* estão divididos em quatro categorias, as quais são especificadas por seus respectivos *profiles* (*Artifact Profile, Interaction Profile, Component Profile*, e *Model Profile*). Essas categorias estão subdivididas em 18 tipos de *assets*, que representam artefatos gerados por processos de desenvolvimento ou modelos que descrevem conceitos associados ao DBC (SANTOS, 2006, p. 18).

No contexto do X-ARM, o termo *asset* representa qualquer artefato de software gerado por um processo de desenvolvimento, tais como casos de uso, código fonte, código executável, planos de teste etc (SANTOS, 2006, p. 17).

Tabela 20. Categorias e tipos de *assets* do X-ARM[35]

X-ARM	
CATEGORIA	***ASSETS***
Artifact Profile	*X-ARM XML Schema* *Licence* *Producer Certification* *Resource*
Interaction Profile	*Independent Interface* *Dependent Interface* *Independent Exception* *Dependent Exception* *Independent Event* *Dependent Event*
Component Profile	*Independent Component Especification* *Dependent Component Especification* *Component Implementation*
Model Profile	*Development Process* *Certification Model* *Business Model* *Component Model* *Application Areas*

Conforme pode ser Observado na Tabela 20, os componentes podem ser especificados utilizando-se o *X-ARM Component Profile*, e podem ser representados em três níveis de abstração: a especificação de um componente independente de modelos de componentes; a especificação de um componente dependente de seu modelo de componente e; a implementação do componente dependente do modelo de componente.

Interfaces são “coleções de operações que são utilizadas para especificar os serviços providos e requeridos pelos componentes [...]. Exceções são recursos utilizados para descrever erros que podem acontecer durante a execução de uma operação” (SANTOS, 2006, p. 19). Interfaces, eventos e exceções, são especificados utilizando-se o *X-ARM Interaction Profile* e, consideradas suas distintas as particularidades, são representados em dois níveis de abstração, representando diferentes tipos de *assets*: suas especificações são dependentes do modelo de componentes adotado ou suas especificações são independentes do modelo de componentes adotado.

[35] Adaptado de Santos (2006, p. 18)

De acordo com Santos (2006), são considerados artefatos, quaisquer outros produtos de software, exceto componentes, interfaces, eventos e exceções, gerados durante um processo de desenvolvimento, como: diagramas de atividades e de classe, casos de uso, e planos de testes. “Em alguns casos, artefatos podem ser reutilizados integralmente entre versões de um mesmo componente. No entanto, em outros casos [...], podem ser reutilizados com adaptações e servir de base para a construção de outros componentes de software" (SANTOS, 2006, p. 21).

Também pode ser visto por meio da Tabela 20, que os artefatos são especificados utilizando-se o *X-ARM Artifact profile* e, “além de representar produtos de software gerados por um processo de desenvolvimento, este *profile* também representa outros três tipos de *assets*” (SANTOS, 2006, p. 21).

No X-ARM as diversas informações descritas nos *assets*, devem estar em conformidade com um determinado padrão, processo ou modelo, ou seja, um componente deve adotar um modelo de componente, ser construído usando um processo de desenvolvimento, ser classificado de acordo com um padrão e apoiar um conjunto de modelos de negócio. O *X-ARM Model Profile* possibilita a especificação de padrões, modelos ou processos utilizados, fornecendo cinco tipos de *assets*, conforme mostra a Tabela 20.

“Os tipos de *assets* gerados a partir do *X-ARM Model Profile* representam os conjuntos de valores possíveis de serem utilizados nos demais *profiles*, assegurando um mecanismo uniforme de descrever e indicar os padrões, modelos ou processos associados” (SANTOS, 2006, p. 22).

Uma das “funcionalidades presentes no X-ARM são: a identificação, classificação, informações sobre o uso dos *assets* e o próprio empacotamento dos artefatos que compõem os *assets*“ (SANTOS, 2006, p. 141).

> O X-ARM constitui uma importante contribuição ao DBC, uma vez que supera algumas limitações significativas de outros modelos. Exemplos de limitações

> superadas são: representação de diversos modelos de negócio, controle de visibilidade, representação de certificação de *assets*, reuso de interfaces e eventos, suporte a diferentes processos de desenvolvimento, representação de componentes de diversos modelos de componentes existentes atualmente [...] e a representação de componentes de possíveis modelos futuros (SANTOS, 2006, p. 141).

O X-ARM é um modelo de representação de *assets*, útil para a indexação, busca e recuperação – de forma padronizada – dos *assets* existentes em um repositório. “No contexto do X-ARM, todo *asset* deve ser descrito em um arquivo de manifesto [...] chamado *manifest.xml*. Os arquivos que compõem o *asset* devem estar dentro do arquivo de pacote e devem ser referenciados pelo manifesto”(SANTOS, 2006, p. 23).

> Os arquivos de manifesto habilitam: (a) verificação de consistência dos arquivos empacotados antes do armazenamento de um *asset* em um repositório, garantindo que os *assets* referenciados já estão registrados no repositório; (b) o *download* de *assets* requeridos em tempo de execução ou desenvolvimento; e (c) detectar dependências entre *assets*, tornando possível assegurar que um *asset* somente pode ser excluído de um repositório quando nenhum outro *asset* depende dele (SANTOS, 2006, p. 23).

A identificação, independência e transparência de localização dos *assets* mantidos em um repositório são garantidas por meio de um esquema de nomeação hierárquico, adotado pelo X-ARM, sem referenciar a localização física. A hierarquia de nomes é ilustrada pela Figura 46.

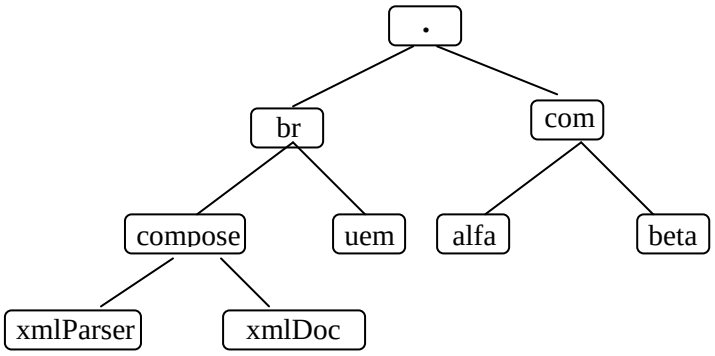


Figura 46. Hierarquia de nomes no X-ARM (SANTOS, 2006, p. 23)

“Em um repositório compatível com o X-ARM, os *assets* armazenados são logicamente agrupados em zonas e domínios [...], zonas representam os produtores de *assets*, enquanto domínios são usados basicamente para agrupar famílias de produtos” (SANTOS, 2006, p. 23).

Uma zona pode possuir domínio, os quais são sempre subordinados a alguma zona. No caso da Figura 46, o ponto (.) representa a zona raiz. O *asset* cujo nome é *xmlParser* é gerado por desenvolvedores pertencentes à zona ou domínio *br.compose* e *br.uem* é uma subzona ou subdomínio da zona ou domínio *br*.

O controle de versões é feito de modo que a versão do *asset* é definida por uma seqüência de números inteiros separados por pontos, como, por exemplo, “1.1.0”, sendo que as versões posteriores obrigatoriamente devem ser maiores que as versões anteriores. O identificador único do *asset* é gerado unindo-se o seu nome à sua versão, separados por um hífen “-“ (EX: br.compose.xmlParser-1.1.0).

Para que seja possível atingir os objetivos inerentes à metodologia proposta neste trabalho, é fundamental que haja uma maneira de sistematizar a identificação, a classificação, o armazenamento, a recuperação e a manutenção de artefatos que possam atender os requisitos de aplicações em desenvolvimento. Desse modo, fica evidente a necessidade de que a metodologia proposta seja apoiada por uma ferramenta que possibilite a representação dos artefatos cuja descrição se faz necessária no repositório de artefatos reusáveis.

O uso do modelo X-ARM como ferramenta de apoio, atende perfeitamente as premissas da abordagem proposta, dado que a mesma foi concebida com o propósito de ser aplicada independentemente do modelo utilizado para criar os artefatos e o X-ARM provê as representações pertinentes a cada artefato gerado ao longo do processo de desenvolvimento, independente do modelo utilizado para instanciar o processo de desenvolvimento.

O modelo X-ARM possibilita que os artefatos gerados quando da adoção da metodologia proposta, sejam armazenados e mantidos de acordo com uma representação, a qual é útil e necessária tanto para a inserção como para a recuperação de artefatos no repositório.

A representação de artefatos propiciada pelo modelo X-ARM, pode ser usada em diversos aspectos da funcionalidade do repositório, tais como: classificação, controle de visibilidade, modelos de negócios, certificação, requisitos não funcionais, dentre outros. Desse modo, todas as informações necessárias ao uso de um determinado artefato estão disponíveis no repositório.

Para apoiar o uso do X-ARM, foi desenvolvida a ferramenta X-Packager, com o propósito de produzir descrições de *assets* na forma de arquivos de manifesto e agrupá-los, juntamente com os demais arquivos associados aos *assets* em pacotes únicos, a qual é descrita a seguir.

6.3.3 A ferramenta x-packager

A ferramenta X-Packager foi criada com o propósito de facilitar a descrição e o empacotamento de *assets* em conformidade com o modelo X-ARM, uma vez que a grande quantidade de características descritas pelo X-ARM, impossibilita a descrição e empacotamento dos *assets* de maneira *ad-hoc* (SANTOS, 2006, p. 141). A partir de um conjunto de informações a X-Packager gera a descrição e o pacote do *asset*, juntamente com os arquivos que o compõem.

A ferramenta X-Packager foi desenvolvida como um *plug-in* para o Eclipse possibilitando que *assets* gerados em projetos utilizando este ambiente, possam ser empacotados de maneira integrada. Contudo, a X-Packager pode ser utilizada para empacotar *assets* cujos artefatos foram desenvolvidos utilizando outros ambientes e ferramentas.

O *plug-in* foi desenvolvido como uma ferramenta do tipo *Wizard* que auxilia o usuário, ao requerer gradativamente, por meio de formulários, as informações necessárias ao empacotamento de um determinado *asset* (SANTOS, 2006, p. 94). O *Wizard* apresenta um conjunto de formulários que varia de acordo com o tipo de *asset* a ser empacotado.

Para Santos (2006), o fato de a ferramenta ter sido desenvolvida como um *plug-in* para o ambiente Eclipse traz a vantagem da funcionalidade de empacotamento de *assets* estar integrada ao próprio ambiente de desenvolvimento dos artefatos que compõem os *assets*. Essa vantagem traz ainda um ponto positivo, que é o fato de o ambiente Eclipse permitir a própria incorporação de *plug-ins*, permitindo que *plug-ins* com distintas funcionalidades possam ser utilizados na geração de diferentes tipos de artefatos.

A ferramenta X-Packager também pode ser utilizada como uma aplicação isolada, desse modo, os artefatos podem ser desenvolvidos em diferentes ambientes e, quando do empacotamento de um *asset*, os artefatos que o compõem devem ser importados em um projeto do Eclipse, para que a ferramenta possa ser utilizada (SANTOS, 2006).

Para a utilização da ferramenta X-Packager, o usuário deve utilizar a opção exportar, oferecida pelo ambiente, no momento de gerar o pacote de *asset*. Ao utilizar a opção exportar, a ferramenta disponibiliza uma tela que possibilita a escolha do destino da exportação, que inclui a opção "*X-ARM Package File*".

Após o usuário ter escolhido a opção de exportar arquivos como um pacote X-ARM, o *plug-in* é iniciado, e conforme o usuário preenche as informações solicitadas em cada formulário, outros formulários são apresentados na seqüência, até o momento em que o usuário escolhe o nome e o local onde salvar o manifesto ou o pacote do *asset*. "Entretanto, a descrição de *assts* pode ser uma tarefa relativamente longa, considerada a grande quantidade de características representadas pelo X-ARM e considerado que o usuário pode não possuir algumas informações quando estiver empacotando um *asset*" (SANTOS, 2006, p. 96).

Os *assets* podem ser empacotados em qualquer momento do desenvolvimento de um projeto, basta que os arquivos que compõem um determinado *asset* estejam concluídos. Desse modo, mesmo antes do término do projeto, é possível empacotar os artefatos produzidos associados a *um asset*, caso estes já tenham sido finalizados.

> A fim de evitar que informações já inseridas sejam perdidas, o X-Packager permite que o empacotamento do *asset* seja interrompido, fazendo com que o pacote do *asset*, ou mesmo apenas o manifesto, seja gerado antes mesmo que todas as informações tenham sido fornecidas. Quando as informações restantes forem conseguidas, o usuário pode acessar novamente a ferramenta e pedir para carregar um pacote X-ARM ou um manifesto que foi gerado anteriormente de maneira incompleta. No caso de um pacote, o X-Packager carrega um arquivo .zip, e no caso de um manifesto, a ferramenta carrega arquivos .xml ou . rmd. Desta forma, o *plug-in* apresentará as informações já preenchidas, permitindo a modificação das mesmas e a inserção das informações que restavam (SANTOS, 2006, p. 96).

Após ter gerado um pacote ou manifesto incompleto, o usuário pode seguir desenvolvendo seus artefatos e ao mesmo tempo, ir adicionando arquivos em seu projeto Eclipse. Quando o empacotamento de um *asset* é reiniciado, a ferramenta X-Packager destaca os arquivos adicionados ao projeto Eclipse, dando ao usuário a opção de adicionar ou não novos arquivos ao pacote do *asset*.

Além de permitir o carregamento de informações de manifestos incompletos, a ferramenta X-Packager possibilita também que manifestos de outros *assets* possam ser carregados para auxiliar a descrição de um *asset*. “Este recurso é útil para evitar que informações que se repetem em vários *assets* tenham que ser introduzidas novamente para cada *asset*” (SANTOS, 2006, p. 97). Desse modo, quando da descrição de um *asset*, podem ser carregadas todas as informações do manifesto de um *asset* já descrito, sendo necessário alterar somente os dados que são diferentes no *asset* que está sendo empacotado.

No que tange aos aspectos de qualidade descritos neste trabalho, a confiabilidade é um dos fatores que influenciam a qualidade, presente em todos os modelos de qualidade de produto. Para Santos (2006), é importante que os componentes utilizados em uma aplicação sejam confiáveis, o que pode ser conseguido por meio da certificação de componentes. A ferramenta X-Packager traz essa informação, ou seja, se o componente foi certificado e por qual modelo de certificação.

6.4 ESTUDO DE CASO

Como dito anteriormente, foi realizado um estudo de caso para avaliar a abordagem de melhoria proposta. O estudo de caso consistiu da aplicação do modelo UML *Components* utilizando o mesmo exemplo utilizado em (CHEESMAN e DANIELS, 2001). Assim sendo, o exemplo utilizado se refere aos artefatos gerados na modelagem de um sistema de reservas de hotel. A primeira parte desta atividade está relacionada ao estudo do UML *Components* visando à compreensão, resultando num conjunto de atividades, cujos conteúdos estão representados na Tabelas 21, 22, 23, 24 e 25.

Tabela 21. Atividades e artefatos do *workflow* de requisitos

ANÁLISE DE REQUISISTOS	
ATIVIDADES	ARTEFATOS
1. Processo do Negócio	
Compreender o processo do negócio	
Descrever o processo de negócio	**Modelo de Processo do Negócio**
2. Modelo de Conceito do Negócio	
Especificar requisitos	**Modelo de Conceitos do Negócio**
3. *Envisioning System*	
Definir a fronteira do software (quais funções são de responsabilidade do software)	
4. Casos de Uso	
Alocar responsabilidades	Modelo de Processo do Negócio com Responsabilidades
Identificar como atores se relacionam com o modelo de conceito de negócio	Modelo de Conceito do Negócio com Relacionamento dos Atores
Identificar casos de uso	**Modelo de Casos de Uso**
Descrever casos de uso	Modelo de Casos de Uso com Adição de Extensões e Variações

São três os artefatos gerados nessa fase: o modelo de conceitos de negócio, o modelo de processo do negócio e modelo de casos de uso, a partir dos quais, são gerados posteriormente, o modelo de tipos do negócio e a especificação e arquitetura inicial de componente.

Na Tabela 22 estão descritas as atividades e os artefatos gerados na primeira fase do *wokflow* de especificação. Os artefatos gerados estão em destaque, a saber: modelo de tipos do negócio, o modelo de interfaces do negócio, modelo de interfaces do sistema e operações e o modelo contendo a especificação e arquitetura inicial de componentes.

Tabela 22. Atividades e artefatos do *workflow* de especificação

IDENTIFICAÇÃO DE COMPONENTES	
ATIVIDADES	ARTEFATOS
1. Identificar Interfaces	
Refinar o modelo de conceito de negócio (visão humana) em um modelo de tipos do negócio (visão do sistema)	
Desenvolver um conjunto de interfaces do negócio	
Caracterizar a camada de diálogo com o usuário com correspondência ao software de diálogo com o usuário	(Modelo de) Entradas da Interface e Correspondência para as Camadas da Arquitetura da Aplicação
2. Identificar Interfaces do Sistema e Operações	
Definir um tipo de diálogo e uma interface do sistema para cada caso de uso . Mapear casos de uso para as interfaces do sistema	Modelo – mapa de Casos de Uso para Interfaces do Sistema (use case/ tipos de diálogo/ passos do use case)
Definir as interfaces e operações iniciais do sistema	**Modelo de Interfaces do Sistema e Operações**
3. Identificar Interfaces do Negócio	**Modelo de interfaces do negócio**
Criar o Modelo de Tipos do Negócio	
Converter o modelo de conceito do negócio em um modelo de tipos do negócio	(Modelo) – Derivando o Modelo de Tipos do Negócio do Modelo de Conceito do Negócio
Refinar o Modelo de Tipos do Negócio	**Modelo de Tipos do Negócio**
Copiar o modelo de conceito do negócio e adicionar e remover elementos até que seu escopo esteja correto	(Modelo) – Escopando o Modelo de Tipos do Negócio
Definir Regras do Negócio	
Identificar quais associações podem ser derivadas de outras	
Escrever algumas restrições e introduzir novos atributos	Modelo de tipos do Negócio Atualizado
Identificar Tipos *CORE* do Negócio	
Criar Interfaces do Negócio e Atribuir Responsabilidades	Modelo de Responsabilidade de Interface do Modelo de Tipos do negócio
Alocar Responsabilidades para Associações . Determinar direção de referência . Estabelecer associações entre tipos gerenciados por diferentes interfaces (associação inter-interface) . Construir um modelo determinando a direção de referência	(Modelo) Atribuição de Direção de Referência
4. Criar uma Especificação Inicial de Interface	(Modelo) – Pacote de Detalhe de Estrutura
5. Interfaces Existentes e Sistemas	
Verificar interfaces existentes que podem ser utilizadas	
Adicionar ao conjunto de interfaces do sistema quaisquer interfaces adicionais que são parte do ambiente	
6. Arquitetura de Especificação de Componentes	
Criar um conjunto inicial de especificação de componentes	
Criar uma especificação de componente separada para cada especificação de interface identificada	
Especificação de Componentes do Sistema	Modelo de Especificação de Componentes do Sistema
Especificação de Componentes do Negócio	Modelo de Especificação de Componentes do Negócio
Arquitetura Inicial	**Especificação e Arquitetura Inicial de Componentes**

Tabela 23. Atividades e artefatos do *workflow* de especificação

INTERAÇÃO DE COMPONENTES	
ATIVIDADES	ARTEFATOS
Decidir como os componentes trabalharão juntos para cumprir as funcionalidades do sistema	Modelo de interações
Definir as várias interações necessárias no sistema	
Refinar definições de interfaces existentes	
Identificar como as interfaces serão usadas	
Descobrir operações do negócio	Diagrama de arquitetura inicial de componente
Descobrir novas interfaces e operações	
Descobrir interações do negócio	Diagrama de interação de componentes
Manter integridade referencial	
Especificar arquitetura de componente (objeto)	Diagrama para restrições na arquitetura do componente
Controlar referências inter componentes (alocar responsabilidades)	Modelo de interação para integridade referencial
	Modelo de interfaces e operações do negócio
Assinalar padrões comuns de uso	
Identificar operações gerais para substituir as específicas	
Entender dependências entre interfaces	
Reexaminar e refinar a arquitetura de componentes	**Modelo de especificação e arquitetura de componentes**
Especificar interações	
Modelar interações	Modelo de interações
Investigar e agrupar alternativas de interfaces de componentes	
Definir a estrutura do sistema	
Descobrir as dependências entre componentes	
Descobrir operações do negócio	**Modelo de operações do negócio**
Refinar as interfaces e operações do sistema	
Refinar a especificação de componentes	
Refinar a arquitetura de componentes	
Definir restrições de implementação	
Fatorar interfaces e operações	
Refinar responsabilidades	
Desenvolver modelos de interação p/ cada operação do sistema	Modelo de Interação
Definir quaisquer restrições	
Controlar referência inter-componentes	
Definir restrições na arquitetura de componentes	Diagrama de integridade referencial

Na Tabela 24, estão descritas as atividades realizadas e os artefatos gerados no último estágio do *workflow* de especificação, a saber: diagramas de especificação de interface, modelo de

especificação de pré e pós-condições, modelo de informação de interface e modelo de restrições de interfaces de componentes.

Tabela 24. Atividades e artefatos do *workflow* de especificação

ESPECIFICAÇÃO DE COMPONENTES	
ATIVIDADES	ARTEFATOS
Especificar contratos de uso e realização	
Especificar interfaces do sistema	**Diagramas de especificação de interface**
Especificar operações	Modelo de interfaces das operações
Desenvolver modelos de informação de interfaces	Modelos de informação de interfaces
Desenvolver modelo de responsabilidades de interface	Diagrama de responsabilidades de interfaces
Especificar em detalhe as operações	
Especificar em detalhes as restrições	
Captar regras do negócio como restrições	
Escrever regras precisas para cada operação	
Definir interfaces individuais	
Especificar restrições específicas a uma especificação de componente independente de cada interface	
Definir modelos de informação de interfaces	Modelo de informação de interfaces
Especificar pré e pós-condições para operações	**Modelo de especificação de pré e pós-condições**
Especificar restrições de interfaces de componentes	
Especificar componentes	Modelo de especificação de componente
Especificar arquitetura de componentes	Modelo de especificação de arquitetura
Especificar invariantes	Diagrama de especificação de interfaces
Construir diagramas *Snapshots* antes e depois da mudança	Diagrama de instâncias *snapshots*
Refatorar interfaces do sistema	**Modelo de informação de interface**
Reafirmar regras do negócio	
Localizar regras do negócio	
Identificar interfaces fornecidas e requeridas	
Dividir o diagrama inicial de especificação de arquitetura em partes específicas para cada especificação de componente	Modelo de especificação para cada componente
Definir restrições de componentes	**Modelo de restrições de interfaces de componentes**
Contextualizar interações	Diagrama de contextualização de interação
Especificar interações de cada componente	
Definir restrições inter interfaces fornecidas e requeridas	
. Parâmetros de operação. Mecanismos para manipulação de erros e exceções . Suporte para interfaces e herança de interfaces operações . Propriedades das interfaces . Criação de objeto . *Raising Events*	Modelo de *Outgoings* Interfaces

Tabela 25. Atividades e artefatos do *workflow provisioning*

PROVISIONING	
ATIVIDADES	ARTEFATOS
Definir o ambiente de execução do componente	
Definir a linguagem de implementação	
Definir um conjunto de serviços de infra-estrutura	
Definir conjunto de regras para operar no ambiente	
Identificar áreas onde as especificações devem acomodar restrições particulares para possibilitar um mapeamento claro p/ a tecnologia de implementação	
Especificação de componentes que realizam componentes	Modelo: Especificações realizam implementações
Definir relacionamento entre um componente e uma interface como um relacionamento << realiza>>	
Definir relacionamento entre um componente e uma interface como um relacionamento << oferece>>	
Caracterizar a camada de diálogo com o usuário correspondendo ao software de diálogo com o usuário	
Representar a função do software com um ou mais tipos de diálogo	
Implementar componentes	**Componentes**

É no *workflow provisioning* que os componentes são implementados, portanto, neste *workflow* foi gerado apenas um tipo de artefato: componentes.

A segunda atividade a ser realizada no estudo de casos deveria ser a geração de *assets*, de acordo com a modelagem do sistema de reservas de hotel, descrito em Cheesman e Daniels (2001), contudo, dada a exigüidade de tempo, a quantidade de *assets* a serem gerados e ao fato de que os *assets* resultantes da aplicação do UML Components ao sistema de reserva de hotel descrito em (CHEESMAN e DANIELS, 2001) já terem sido gerados durante o trabalho de validação da ferramenta X-Packager (SANTOS, 2006), optou-se pelo reuso desses *assets* ao invés de investir tempo em re-trabalho. Assim sendo, procurou-se entender melhor o funcionamento da ferramenta X-Packager com o intento de compreender como era feita a geração de *assets* e, como esses *assets* eram descritos e empacotados.

O processo de avaliação da ferramenta X-Packager (SANTOS, 2006) resultou na descrição e empacotamento dos *assets* gerados ao longo do processo de desenvolvimento, os quais foram agrupados seguindo-se as fases do processo UML *Components*, exceto os *assets* representados com o *X-ARM Model Profile*, os quais foram descritos anteriormente à geração dos demais *assets*, dado que normalmente, a descrição do *asset* com X-ARM precisa referenciar outros *assets*, os quais, por sua vez, precisam estar registrados e empacotados no repositório.

Conforme descrito em Santos (2006), foram gerados 66 (sessenta e seis) *assets*, de acordo com o que mostra a Tabela 26.

Tabela 26. Tipos e quantidade de *assets* gerados (SANTOS, 2006)

Workflow	**Qtde[36] de *assets***	**Tipo**
Requisitos	3	*Resource*
Especificação - identificação	4	*Resource*
Especificação - interação	13	2 *Resource* 4 *Idenpendent interface* 7 *Independent component specification*
Especificação - especificação	12	*Resource*
Provisioning	34	8 *Dependent exception* 8 *Dependent interface* 9 *Development component specification* 9 *Component implementation*
TOTAL	66	

Embora no trabalho realizado no estudo de caso foram gerados alguns artefatos com o propósito de exemplificar de que maneira a metodologia proposta pode ser apoiada por um repositório, mais especificamente, pelo repositório definido na ferramenta X-Packager.

Na metodologia proposta, as atividades de inspeção e teste perpassam todas as fases de qualquer modelo para construção de software utilizado para instanciar o processo de desenvolvimento, neste caso específico, essas atividades perpassam todos os *workflows* do

[36] Quantidade.

UML *Components*. Assim sendo, deparou-se com a seguinte questão: o que devem ser gerados como decorrência das atividades de teste e inspeção ao longo do processo de desenvolvimento: *assets* ou artefatos?

Como essas atividades são aplicadas a todos os artefatos intermediários gerados em cada fase do modelo utilizado, o entendimento foi de que os arquivos contendo: planejamento das atividades de teste e inspeção; os projetos de casos de teste, os arquivos contendo os resultados da aplicação de teste e inspeção em cada um dos artefatos gerados, os arquivos contendo as informações sobre validação da atividade de teste, os arquivos contendo os resultados das inspeções das atividades teste, dentre outros, deveriam ser empacotados como arquivos associados aos *assets* referentes aos artefatos apresentados em (CHEESMAN e DANIELS, 2001). Os arquivos relacionados às atividades de teste e inspeção foram gerados como artefatos que compõem os *assets* aos quais estão relacionados. Assim sendo, esses arquivos passam a compor, juntamente com o conjunto de arquivos associados aos *assets* e os próprios *assets*, um pacote único.

Na realização do estudo de caso, não foi visto como seriam de fato realizadas as atividades de teste e inspeção, dado que os artefatos gerados não foram realmente testados, não sendo, portanto, possível, por exemplo, avaliar os resultados esperados de teste com os resultados obtidos e tampouco, avaliar por meio do uso combinado dessas atividades, um determinado componente, ao longo do processo, conforme postula a metodologia proposta. Porém, foi possível verificar a viabilidade da metodologia proposta ao nível de reuso, dado que os artefatos gerados foram empacotados em um único bloco, juntamente com os arquivos associados ao *asset* do qual fazem parte.

O ideal para o estudo de caso realizado, seria testar e inspecionar, de fato, todos os *assets* e artefatos gerados. Contudo, diante da exigüidade de tempo, foram criados alguns arquivos referentes às atividades de inspeção e teste de alguns para que fosse possível exemplificar a realização dessas atividades ao longo do processo, bem como validar a metodologia proposta. Tais arquivos foram empacotados como artefatos, juntamente com os demais arquivos que compõem esses *assets*, conforme pode ser visto nas Figuras 47 e 48, as quais ilustram apenas dois exemplos de artefatos gerados, com o propósito de evitar repetições.

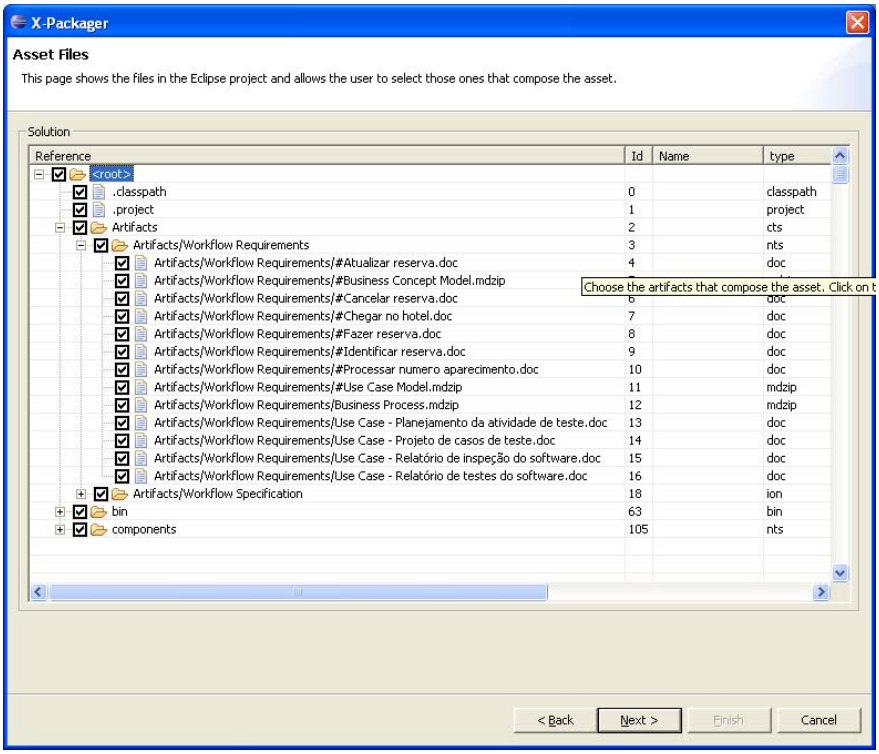


Figura 47. Artefatos gerados no *workflow* de requisitos

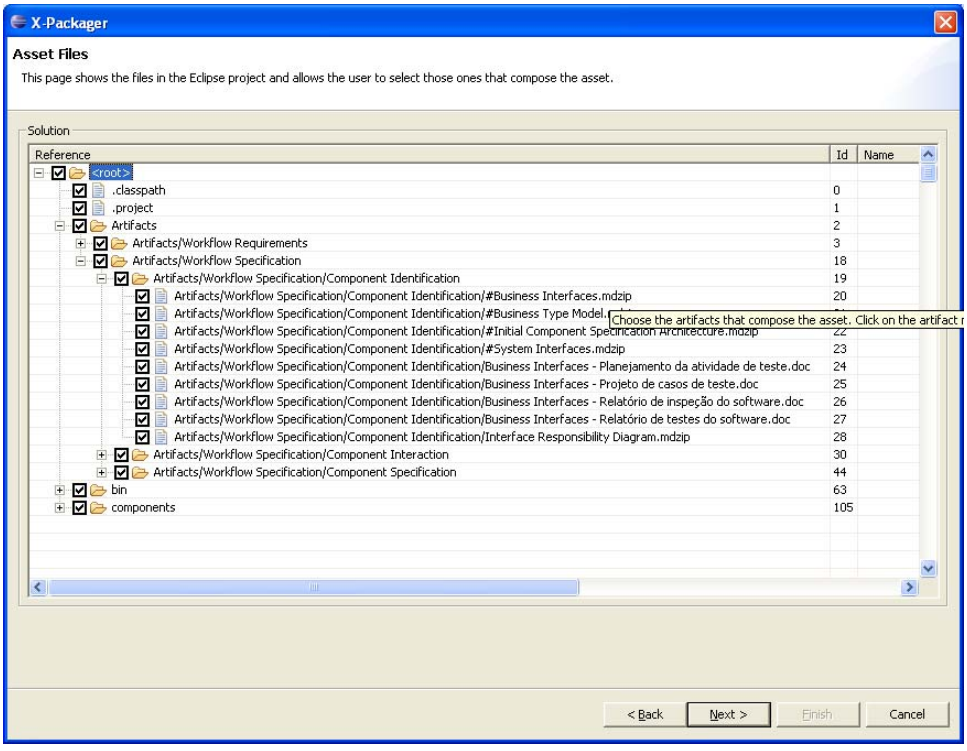


Figura 48. Artefatos gerados no *workflow* de especificação

O estudo de caso mostrou que a metodologia proposta é viável, contudo, deve-se considerar que a mesma deve ser apoiada por uma ferramenta tal como experienciado, uma vez que é praticamente impossível realizar as atividades de inspeção e teste e a prática de reuso sistematicamente sem o suporte de uma ferramenta que possibilite a descrição e o empacotamento de artefatos para formar e manter um repositório de artefatos reusáveis.

Como visto anteriormente, o modelo X-ARM, possibilita também a representação de metadados que descrevem os componentes de software, esses metadados, devem ser representados sem ambigüidades com o objetivo de descrever o que é o componente, que interfaces requer e fornece, os eventos que dispara e recebe, quais componentes o compõem, a quais certificações foi submetido, certificado por quais entidades, quem pode recuperá-lo, qual o seu custo e as formas de pagamentos. É possível sim, gerar os artefatos relativos às atividades de inspeção e teste, todavia, sem o apoio de uma ferramenta fica inviabilizado o uso da representação de metadados de *assets*/artefatos nos diversos aspectos das funcionalidades de um repositório de artefatos reusáveis.

6.5 CONSIDERAÇÕES FINAIS

O fato do modelo X-ARM prover um repositório que possibilita o armazenamento não apenas de componentes, mas também dos metadados que os descrevem, aliado ao fato de que uma vez disponíveis no repositório, tais metadados podem ser utilizados na indexação e classificação de componentes, tornando possível a descoberta de componentes que atendam as necessidades de quem deles necessitem, constitui uma característica importante para as empresas que utilizam a abordagem de fábrica para ao desenvolvimento de seus produtos de software, mais ainda para as empresas cuja abordagem adotada está baseada no modelo de fábrica proposto pela Microsoft, dado que tal modelo contempla o uso sistemático de metadados, para derivar automação e de *assets*, para reduzir custos e prazos de entrega no mercado, melhorando a qualidade do produto durante o seu desenvolvimento.

Desse modo, o suporte computacional utilizado para avaliar a metodologia proposta está inserido no contexto de fábrica de software, indo ao encontro, principalmente da abordagem de fábrica da

Microsoft, possibilitando o apoio necessário e adequado para atender aos objetivos propostos em tal abordagem que visa primordialmente à qualidade por meio de reuso sistemático, linha de produto, desenvolvimento dirigido à modelo dentre outros.

Ainda que com conotações distintas, o uso de *assets* está presente no modelo X-ARM bem como no modelo de fábrica proposto pela Microsoft. Do mesmo modo, a utilização de metadados, cujo objetivo é, segundo Santos (2006), fornecer uma descrição do componente, dos componentes que o compõem, das interfaces que o mesmo requer e provê, dos eventos que dispara, das certificações às quais foi submetido e por quais entidades certificadoras, bem como, informando quem pode recuperá-lo, além do preço e das formas de pagamento possíveis.

As dificuldades decorrentes da realização deste trabalho estão relacionadas, à pesquisa intensa e detalhada que compõe o referencial teórico que embasa este trabalho. Posteriormente, a dificuldade envolveu o entendimento e conseqüente uso da ferramenta X-Packager. Em relação ao uso da ferramenta, problemas com as versões do Eclipse e do Java dificultaram e atrasaram sobremaneira a execução do estudo de caso e, finalmente, a falta de tempo testar e inspecionar, com dados reais, todos os artefatos e *assets* gerados e aplica-los à ferramenta para validar a metodologia proposta. Outra dificuldade relacionada ao uso da ferramenta diz respeito à usabilidade da ferramenta. O uso da X-Packager é difícil devido à grande quantidade de formulários a serem preenchidos e devido à exigência de conhecimento do modelo X-ARM.

Como contribuições deste trabalho, além do aprofundamento dos conteúdos relacionados aos modelos de qualidade de produto e de processo, aos conceitos de fabrica de software, podem ser destacadas: a apresentação da metodologia proposta, a qual requer o uso combinado e sistemático ao longo de todo o processo de desenvolvimento, das atividades de teste e inspeção e de reuso, visando à melhoria de produto e de processo.

A realização e o registro dos resultados do estudo de caso realizado, mostrando que a metodologia proposta pode ser apoiada por um repositório de artefatos reusáveis, bem como, por um suporte computacional que possibilita a consecução dos objetivos propostos tanto pelas empresas que adotam o conceito de fábrica de software como por empresas que desejam

adotar a metodologia proposta, constituem em importantes contribuições para a área de engenharia de software.

7. TRABALHOS FUTUROS

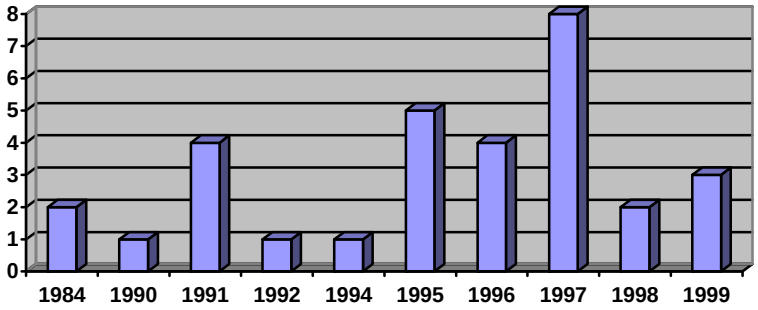
Como trabalhos futuros, a metodologia pode ser estendida para contemplar reuso de conhecimento, nesse sentido, ter-se-ia que estender também o modelo X-ARM e a ferramenta X-Packager, para que fosse possível a descrição e o empacotamento de conhecimento gerado ao longo do processo de desenvolvimento.

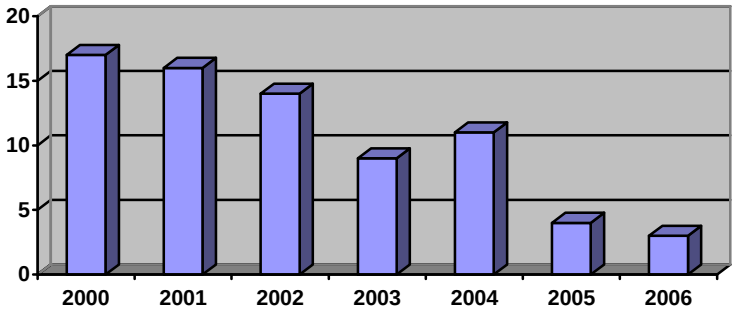
Para contemplar o reuso de conhecimento, um repositório específico pra este fim deve ser projetado. Além disso, um estudo aprofundado sobre como tratar, classificar, armazenar, classificar e disponibilizar esse conhecimento deve ser realizado.

Poder-se-ia aplicar a metodologia proposta a outros modelos de desenvolvimento e confrontar os resultados obtidos, assim como, acompanhar os resultados do desenvolvimento de um determinado produto de software com e sem a aplicação da metodologia proposta com o intuito de avaliar os resultados em ambas as situações, ou seja, com e sem a aplicação da metodologia, comparando os resultados numa tentativa de verificar o quão diferentes são os resultados obtidos em ambas as situações de uso.

8. GRÁFICOS DAS REFERÊNCIAS BIBLIOGRÁFICAS UTILIZADAS

Os gráficos abaixo refletem a bibliografia usada, de acordo com o ano de publicação. Procurou-se utilizar referências atuais para embasar a proposta de tese em questão, contudo, alguns artigos são considerados clássicos em relação a alguns dos temas abordados e, por isso, foram utilizados para fundamentar este trabalho.





9. REFERÊNCIAS BIBLIOGRÁFICAS

AAEN, I.; BOTCHER, P.; MATHIASSEN, L. *Software Factories*. Proceedings of the twentieth Information Systems Research Seminar in Scandinavia, Oslo, 1997.

ALBUQUERQUE, S. *O Impacto do Processo de Reutilização de Componentes nas Organizaçõe*s *que Produzem Sistemas Aplicativos. Guideline* BRISA, Março, 2001.

ALSPAUGH, T. A. *Software Quality. Software Specification and Quality Engineering*. UCIrvine-University of California, Irvine, Spring, 2005.

BARROS, A. de J. P. de; LEHFELD, N. A. de S. Projeto de Pesquisa: Propostas Metodológicas. Petrópolis: Editora Vozes, 1990.

BASS, L. *et al*. *Software Architecture in Practice*. [S.l], Addison Wesley: Longman, 1998.

BASILI, V. R.; CALDIERA, G. *Improve Software Quality by Reusing Knowledge an Experience*. MITSloan Management Review, Vol. 37, Nº 1, 1995.

BASILI, V. R., *et al*. *New Year's Resolutions for Software Quality*. IEEE Software. January/February, vol. 21, nº 1, 2004, p. 12-13.

BECK, K.; JOHNSON, R. E. *Patterns Generate Architectures*. In Proceedings of the ECOOP'94. Berlin: Springer Verlag, 1994.

BIFFI, S. *Using Inspection Data for Defect Estimation*. IEEE software, November/ December, 2000.

BOOCH, G., *et al*. *The Unified Modeling Language Users Guide*. [S.1]: Addison-Wesley Publishing Company, 1999.

BOSCH, J. *Design et Use of Software Architectures Adopting and Evolving a Product-line Approach*. Addison Wesley, 2000.

BRASIL. Ministério da Ciência e Tecnologia. *Qualidade no setor de software brasileiro - 1999.* Brasília: Ministério da Ciência e Tecnologia, consulta realizada em junho de 2005. Disponível em: <http://www.mct.gov.br/Temas/info/dsi/palestra/palestras.htm>

BUTLER, G. *Quality and Reuse in Industrial Software Engineering*. Proceedings of Asia-Pacific Software Engineering Conference and International Computer Science Conference, December, 3-5, Hong Kong, IEEE Computer Society Press, Los Alamitos, CA, 1997. p. 3-12.

CABRAL, D. M.; LIMA, R. B.V. *Agile Creation of A Software Factory*. The 2004 International Conference on Software Engineering Research an Practice, SERP'04, Las Vegas, Nevada, USA, June 21-25, 2004.

CHERNAK, Y. *Validating and Improving Test-Case Effectiveness*. Valley Forge Consulting, IEEE Software, January/February, 2001.

CHEESMAN, J; DANIELS, J. UML Components: A Simple Process for Specifying Component-Based Software. Addison-Wesley. 2001.

CHRISSIS, M. B.; KONRAD, M.; SHRUM, S. *CMMI®: Guidelines for Process Integration and Product Improvement*. Addison Wesley, February, 2003.

CURTIS, B. *The Global Pursuit of Process Maturity*. IEEE Software. July/August, vol. 17, nº 4, 2000, p. 76-78.

DELAMARO, M. E.; MALDONADO, J. C.; MATHUR, A. P. *Interface Mutation: An Approach for Integration Testing*. IEEE Transactions on Software Engineering, vol. 27, nº 3, March, 2001.

DEVANBU, P. T.; PERRY, D. E.; POULIN, J. S. *Guest Editors' Introduction: Next Generation Software Reuse*. IEEE Transactions on Software Engineering, vol. 26, nº 5, May, 2000.

EADIE, S. *A GSI'S Perspective of Software Factories*. Microsoft Architect Journal. Journal 9. October, 2006.

EICKELMANN, N. *Measuring Maturity Goes Beyond Process*. IEEE Software, vol. 21, nº 4, July/August, 2004, p. 12-13.

EMAM, El K. *Validating the ISO/IEC 15504 – Measure of Software Requirements Analysis Process Capability*. IEEE Transaction on Software Engineering, vol. 26, nº 6, June, 2000.

FACH, P. W. *Design Reuse Through Frameworks and Patterns*. IEEE Software. September/October, 2001.

FAZENDA, I. *Metodologia da Pesquisa Educacional*. 6ª edição. São Paulo: Cortez Editora, 2000.

FERNESTRÖM, C.; K-H., NÄRFELT; OLSSON, L. *Software Factory Principles, Architecture and Experiments*. IEEE Software, March, 1992, p. 36-44.

FERRARI, S. *Integração da Ferramenta de Teste POKE-TOOL em Ambientes de Engenharia de Software*. Dissertação (Mestrado) – Universidade Estadual de Campinas, Campinas, Junho, 1998.

FOWLER, G. S.; KORN, D. G.; VO, KIEM-PHONG. *Principles for Writing Reusable Librari*es. Proceedings of the 1995 Symposium on Software Reusability, April 29-30, Seatle, Washington, USA, 1995, p. 150-159.

FRAKES, W.; TECH, V.; TERRY, C. *Software Reuse: Metrics and Models*. ACM Computing Surveys, vol. 28, nº 2, June, 1996.

FULLER, Tom. *A Foundation for the Pillars of Software Factories*. Microsoft Architect Journal. Jounal 9. October, 2006.

GAMMA, E. *et al*. *Padrões de Projeto – Soluções Reutilizáveis de Software Orientado a Objetos*. Porto Alegre: Bookman, 2000.

GENUCHTEN, M. V. *et al*. *Using Group Support Systems for Software Inspections*. IEEE Software, May/June, 2001.

GHEZZI, C.; JAZAYERI, M.; MANDRIOLI, D. *Fundamentals of Software Engineering*. Prentice –Hall, 1991.

GILL, N. S. *Factors Affecting Effective Software Quality Management Revisited*. ACM SIGSOFT Software Engineering Notes, vol. 30, nº 2, March, 2005.

GIMENES, I. M. S.; TRAVASSOS, G. H. *O Enfoque de Linha de Produto para Desenvolvimento de Software*. Trabalho Apresentado na Jornada de Atualização em Informática, SBC, 34 p. Florianópolis, SC, 2002.

GOUVEIA, A. J. *Notas a Respeito das Diferentes Propostas Metodológicas Apresentadas*. Maio, 1984.

GREENFIEL, J; SHORT, K. *Software Factories Assembling Applications with Patterns, Models, Frameworks and Tools*. Conference on Object Oriented Programming Systems Language and Applications – OOPSLA'03. Anaheim, California, USA. October, 2003, p. 26-30.

GREENFIEL, J. *A Case for Software Factories*. Microsoft Architect Journal. Journal 3. July, 2004.

HENNINGER, S. *An Evolutionary Approach to Constructing Effective Software Reuse Repositories*. ACM Transactions on Software Engineering and Methodology, vol. 6, nº 2, April, 1997, p. 111-140.

HERZUM, P.; SIMS, O. *Business Component Factory – A Comprehensive Overview of Component-Based Development for the Enterprise*. OMG Press, Wiley Computer Publishing, 2000.

HILBURN, T. B.; TOWHIDNEJAD, M. *Software Quality: A Curriculum Postcript*? Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education. Austin, Texas, USA, 2000, p. 167-171.

HUMPHREY, W. S. *The Team Software Process^{SM} (TSP^{SM}) (CMU/SEI-2000-TR-023).* Software Engineering Institute, Carnegie Mellon, November, 2000.

ISO/IEC 12207. *Information Technology – Software Life Cycle Processes*. ISO/IEC 12207:2002/FDAM 2:2003 (E). ISO/IEC JTC1/SC7/WG7 nº 0759, 2003.

.
ISO/IEC CD 15504-1.2. ISO/IEC TC JTC1/SC 7 N 470. *Information Technology – Process Assessment – Part 1: Concepts and Vocabulary*. 2003.

ISO/IEC FDIS 15504-3: 2003 (E). ISO/IEC JTC 1/SC 7 WG 10 N 472. *Information Technology – Process Assessment Part 3: Guidance on Performing an Assessment*. 2003.

ISO/IEC, ISO 9126. *Quality Characteristics and Guidelines for Their Use*. Dezembro, 1991.

ISO/IEC, ISO 9126-1. *Software Engineering – Product quality – Part 1: Quality model*. First edition, June, 2001.

JONES, T. C. *Reusability in Programming – A Survey of the State of the Art*. IEEE Transactions on Software Engineering. Vol. SE-10, nº 5, September, 1984.

JONES, C. *Produtividade no Desenvolvimento de Software*. São Paulo: MAKRON Books do Brasil Editora Ltda, 1991.

JORGENSEN, M. *Software Quality Measurement*. Advances in Engineering Software, Vol. 30, [S.L.], 1999, p. 907-912.

KHADDAJ, S.; HORGAN, G. *The Evaluation of Software Quality Factors in Very Large Information Systems*. Electronic Journal of Information Systems Evaluation. vol. 7, nº1, 2004, p. 43-48.

KELLY, D; SHEPARD, T. Special issue: Application of Statistics in Software Engineering. Journal of Systems and Software. vol. 73, nº 2, October, 2004, p. 361-368.

KITCHENHAM, B. A. *Modeling Software Measurement Data*. IEEE Transactions on Software Enginnering, vol. 27, nº. 9, September, 2001.

KOMI-SIRVIÖ, S; MÄNTYNIEMI, A.; SEPPÄNEN, V. *Toward a Practical Solution for Capturing Knowledge for Software Projects*. IEEE Software. vol. 19, nº 3, May/June, 2002, p. 60-62.

KONTIO, J. *OTSO: A Systematic Process Reusable Software Component Selection*. CS-TR-3478, UMIACS-TR-95-63. Institute for Advanced Studies and Department of Computer Science, University of Maryland, December, 1995.

KRUCHTEN, P., Introdução ao RUP – *Rational Unified Process*. Editora Moderna, 2003.

KULPA, M. K.; KENT, A. J. *Interpreting the CMMI: A Process Improvement Approach*. Auerbach Publications, [S.L.], 2003.

LAITENBERGER, O.; EMAM, K. El; HARBICH, T. G. *An Internally Replicated Quasi-Experimental Comparision of Checklist and Perspective-Based Reading of Code Documents*. IEEE Transaction on Software Engineering, vol. 27, nº 5, May, 2001.

LAVAZZA, L., *Providing Automated Support for the GQM Measurement Process*. IEEE Software, vol. 17, nº 3, May/June, 2001, p. 56-62.

LAVAZZA, L.; BARRESI, G. *Automated Support for process: Aware Definition and Execution of Measurement Plans*. Proceedings of the 27th International Conference of Software Engineering, St. Louis, Mo, USA, 2005, p. 234-243.

LAZILHA, F. R. *Uma Proposta de Arquitetura de Linha de Produto para Sistemas de Gerenciamento de Workflow*, Dissertação (Mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2002.

LINDVALL, M.; RUS, I. *Process Diversity in Software Development.* IEEE Software, vol. 17, nº 4, August, 2000, p. 14-71.

MALDONADO, J. C. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*, Tese de doutorado, DCA/FEE/UNICAMP, Julho, 1991.

MARQUES, H. M.; RAMOS, R. T.; SILVA, I. G. I. *Adaptação de um Processo* de *Desenvolvimento para Fábricas de Software Distribuídas*. 7º Workshop Ibero-Americano de Engenharia de Requisitos e Ambientes de Software, IDEAS'2004, Arequipa, Peru, 3-7 de Maio, 2004.

MENDONÇA, M. G.; BASILI, V. R. *Validation of an Approach for Improving Existing Measurement Frameworks*. IEEE Transactions on Software Engineering, vol. 26, nº 6, June, 2000.

MENS, T.; DEMEYER, S. *Future Trends in Software Evolution Metrics*. Proceedings of the 4th International Workshop on principles of Software Evolution, Viena, Áustria, September, 2001, p. 83-86.

MICHAEL, C. C.; McGRAW, G.; SCHATZ, M. A. *Generating Software Test Data by Evolution*. IEEE Transactions on Software Engineering, vol. 27, nº. 12, December, 2001.

MORISIO, M.; TULLY, C.; EZRAM, M. *Diversity in Reuse Process*. IEEE Software, July/August, vol. 17, nº 4, 2000, p. 56-62.

MUTAFELIJA, B.; STROMBERG, H. *Systematic Process Improvement Using ISO 9901-2000 and CMMI (cm)*. Artech House INC. Computing Library, [S.L.], 2003.

NTAFOS, S. C. *An Comparision of Random, Partition and Proportional Partition Testing*. IEEE Transactions on Software Engineering, vol. 27, nº 10, October, 2001.

PÁDUA, E. M. M de. Metodologia da Pesquisa e Abordagem Teórico-Prática. Papirus Editora, 10ª Edição Revisada e Atualizada, Coleção Magistério: Formação e Trabalho Pedagógico, Campinas, 2004.

PAILO, L. F., *et al*. *Avaliação da Adequação de Conjunto de Teste Funcional Utilizando Critérios Estruturais com o Apoio de Ferramentas de Teste Comerciais*. Workshop do projeto de Validação e Teste de Sistemas de Operação, WPVTSO97, Águas de Lindóia, Janeiro, 1997.

PAULK, M. C., et al. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison - Wesley Publishing Company, 1995.

PAULK, M. C., *et al*. *A Preview of the Software CMM - Version 2*. 1996. http://citeseer.ist.psu.edu/correct/12058.

PAULK, M. C. *Toward Quantitative Process Management with Exploratory Data Analysis*. International Conference on Software Quality, Cambridge, May, 1999.

PRESSMAN, R. S. *Engenharia de Software*. São Paulo: MAKRON Books do Brasil Editora Ltda, 1995.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1997.

RAMASUBRAMANIAN, S; JAGADEESAN, G. *Knowledge Management at Infosys*. IEEE Software. vol. 19, nº 3, May/June 2002, p. 53-55.

RAMESH, B. *Process Knowledge Management With Traceability*. IEEE Software. vol. 19, nº 3, May/June 2002, p. 50-52.

REITER, D. J. *A Little Art of Knowledge Is a Dangerous Thing*. IEEE Software. vol. 19, nº 3, May/June 2002, p. 14-15.

RINE, D. C. *Success Factors for Software Reuse That Are Aplicable Across Domains And Business*. Proceedings of the 1997 ACM Symposium on Applied Computer, February 28, March 1, San José, CA, USA, 1997.

RIFKIN, S. *What Makes Measuring Software So Hard*? IEEE Software, vol. 18, nº 3, May/June, 2001, p. 41-45.

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

RUS, I.; LINDVALL, M. *Knowledge Management in Software Engineering*. IEEE Software. vol. 19, nº 3, May/June 2002, p. 26-38.

SCHMIETENDORF, A.; DUMKE, R.; TELEKOM, E. F. *Metrics Based Asset Assessment*. ACM SIGSOFT – Software Engineering Notes, vol. 5, nº 4, July, 2000.

SEAMAN, C. B., *et al*. *User Interface Evaluation and Empirically-Based Evolution of a Prototype Experience Management Tool*. IEEE Transactions on Software Engineering, vol. 29, nº 9, September, 2003.

SEI: CMMI Product Development Team. *CMMI for Systems Engineering/Software Engineering/Integrated Product and Process Development*. Version 1.1 Staged Representation (CMU/SEI-2002-TR-004, ESC-TR-2002-004). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, January 2002a. <http://www.sei.cmu.edu/publications/documents/02.reports/02tr004.html>.

SEI: CMMI Product Development Team. *CMMI for Systems Engineering/Software Engineering/Integrated Product and Process Development*. Version 1.1 Continuous Representation (CMU/SEI-2002-TR-003, ESC-TR-2002-003). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, January 2002b. <http://www.sei.cmu.edu/publications/documents/02.reports/02tr003.html>.

SEI: CMMI Product Development Team. *CMMI for Systems Engineering/Software Engineering/Integrated Product and Process Development/Supplier Sourcing*. Version 1.1 Staged Representation (CMU/SEI-2002-TR-012, ESC-TR-2002-012). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, March 2002c. <http://www.sei.cmu.edu/publications/documents/02.reports/02tr012.html>.

SEI: CMMI Product Development Team. *CMMI for Systems Engineering/Software Engineering/Integrated Product and Process Development/Supplier Sourcing*. Version 1.1 Continuous Representation (CMU/SEI-2002-TR-011, ESC-TR-2002-011). Pittsburgh, PA:

Software Engineering Institute, Carnegie Mellon University, March 2002d. <http://www.sei.cmu.edu/publications/documents/02.reports/02tr011.html>.

SEI: Software Engineering Institute. *CMMI Product Development Team. CMMI for Sy*stems *Engineering/Software Engineering*. Version 1.1 Staged Representation (CMU/SEI-2002-TR-002, ESC-TR-2002-002). Pittsburgh, Carnegie Mellon University, January 2002f. <http://www.sei.cmu.edu/publications/documents/02.reports/02tr002.html>.

SIY, H. P. *et al*. *Making the Software Factory Work: Lessons from a Decade of Experience*. Seventh International Software Metrics Symposium. London, England, April 04-06, 2001.

SOMMERVILLE, I. *Software Engineering*. Fifth edition, Addison Wesley, 1996.

TIAN, J. *Quality-Evaluation Models and Measurement*. IEEE Software, vol. 21, nº 3, May/June, 2004, p. 84-91.

VALÉRIO, A. *Special Issue on the Effects of Frameworks and Patterns on Software Reuse*. ACM SIGAPP COMPUTING REVIEW, vol. 5, nº 2, September, 1997.

VERGÍLIO, S. R.; MALDONADO, J. C.; JINO, M. *Resultados de um Experimento de Aplicação de Diferentes Técnicas de Geração de Dados de Teste*. Workshop do projeto de Validação e Teste de Sistemas de Operação, WPVTSO97, Águas de Lindóia, Janeiro, 1997.

VOTH, D. *Packaging Reusable Software Assets*. IEEE Software, vol. 21, nº 3, May/June, 2004, p. 107-110.

VRIES, M. de. *Measuring Success with Software*. Microsoft Journal. Journal 9, October 2006.

WANG, J. A. *Towards Component-Based Software Engineering*. Proceedings of the Seventh Annual Consortium on Computing in Small Colleges Midwestern Conference, Valparaiso, Indian, Consortium for Computing in Small Colleges, USA, 2000.

WANGENHEIM, C. A. G. von. *Reutilização Baseada em Casos de Experiência na Área de Mensuração em Engenharia de Software*. Tese – Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia De Produção, Florianópolis, BR-SC, 2000.

WEI, C.; HU, P. J.; CHEN, H. *Design and Evaluation of a Knowledge Management System*. IEEE Software. vol. 19, nº 3, May/June, 2002, p. 56-59.

WEINBERG, G. M. *Software com Qualidade – Volume 3 – Ação Congruente*. MAKRON BOOKS do Brasil, 1996.

WERNER, C. M. L.; BRAGA, R. M. M. *Desenvolvimento Baseado em Componentes*. Anais do XIV Simpósio Brasileiro de Engenharia de Software. João Pessoa, 2000.

WONG, B. *Second Workshop on Software Quality*. *Proceedings of the 26th International Conference on Software Engineering (ICSE'04*). January/February, 2004, p. 780-782.

ZUBROW, D. *The Measurement and Analysis Process Area in CMMI*. Software Engineering Institute, 2002. <http://www.sei.cmu.edu/cmmi/publications/meas-anal-cmmi.html>.