INSTITUTO DE PESQUISAS TECNOLÓGICAS DO ESTADO DE SÃO PAULO

SUZANA MAYUMI ASSATO

Proposição de uma Estratégia de Inspeção de *Software* para Sistemas Acadêmicos Orientados a Objetos

São Paulo
2008

SUZANA MAYUMI ASSATO

Proposição de uma Estratégia de Inspeção de *Software* para Sistemas Acadêmicos Orientados a Objetos

Dissertação apresentada ao Instituto de Pesquisas Tecnológicas do Estado de São Paulo - IPT, para obtenção do título de Mestre em Engenharia da Computação.

Área de concentração: Engenharia de *Software*

Orientador: Prof. Dr. Marco Dimas Gubitoso

São Paulo
2008

Ficha Catalográfica
Elaborada pelo Departamento de Acervo e Informação Tecnológica – DAIT
do Instituto de Pesquisas Tecnológicas do Estado de São Paulo - IPT

A844p **Assato, Suzana Mayumi**

Proposição de uma estratégia de inspeção de software para sistemas acadêmicos orientados a objetos. / Suzana Mayumi Assato. São Paulo, 2008.

121p.

Dissertação (Mestrado em Engenharia de Computação) - Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Área de concentração: Engenharia de Software.

Orientador: Prof. Dr. Marco Dimas Gubitoso

1. Inspeção de software 2. Teste de software 3. Software orientado a objetos 4. Engenharia de software 5. Tese I. Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Coordenadoria de Ensino Tecnológico II. Título

08-209 CDU 004.415.53’045(043)

DEDICATÓRIA

Dedico este trabalho aos meus pais, aos meus familiares e a todos que tenham interesse em aprofundar seus conhecimentos sobre inspeção do processo de desenvolvimento de sistemas.

AGRADECIMENTOS

Agradeço em primeiro lugar a Deus, por me proporcionar a oportunidade de me dedicar a um mestrado que exige esforço e disciplina.

Em seguida, agradeço ao meu orientador Prof. Dr. Marco Dimas Gubitoso, que tem me apoiado e ajudado muito para que o trabalho se tornasse cada vez melhor estruturado e apresentado.

Agradeço aos participantes da banca examinadora, Prof. Dr. Reginaldo Arakaki e Prof. Dr. Flavio Soares Correa da Silva, que contribuíram com críticas construtivas para o aperfeiçoamento do trabalho.

Ao coordenador Prof. Dr. Mário Yoshikazu Miyake e a todos os demais professores do Ipt, agradeço pelos conhecimentos transmitidos.

Aos meus pais, aos familiares e aos queridos amigos, agradeço por me ajudarem e sempre incentivarem a me desenvolver e atingir os objetivos.

RESUMO

Muitas vezes a identificação e a correção dos erros são deixadas para fases posteriores do desenvolvimento de sistemas e, dessa forma, maiores serão os esforços e os custos envolvidos na sua remoção, pois poderão migrar e se acumular de uma fase para outra.

A tarefa de medir o quão completamente um produto foi testado ou de avaliar o grau de profundidade em que os casos de teste foram aplicados ao produto é complexa. Podem-se identificar erros, mas não se pode garantir que um artefato não os possua.

Inspeção de *software* é o processo de analisar um artefato de *software* a fim de identificar defeitos. Embora ela seja importante e necessária, não existem informações completas e precisas sobre sua aplicação em códigos e modelos orientados a objeto.

Este trabalho objetiva que haja maior qualidade e rigor no desenvolvimento de sistemas ao se aplicar a inspeção e propõe uma estratégia para o acompanhamento e documentação da inspeção de *software*. Essa estratégia foi parcialmente implementada em um ambiente *Wiki*.

Palavras-chave: inspeção de *software*; teste de *software*; projeto de *software*; orientação a objetos; sistemas acadêmicos.

ABSTRACT

Often errors identification and correction are left to later phases of systems development and consequently, the efforts and costs related to their removal will be greater because errors can migrate and accumulate from phase to phase.

The task of measuring how completely a product has been tested or the task of evaluating to which level test cases were applied to the product is complex. It is possible to identify errors, but not to assure that an artifact does not contain them.

Software inspection is the process of analyzing a software artifact to identify defects. Although it is important and necessary, there is not complete and precise information about applying it on object-oriented code and models.

This work intends to improve quality and rigor in systems development by applying inspection and proposes a strategy for software inspection documentation and maintenance. This strategy was partially implemented within a Wiki environment.

Key-words: software inspection; software testing; software project; object-oriented; academic systems.

Lista de ilustrações

Figura 1 - A hierarquia dos atributos de qualidade de *software* 18
Figura 2 – O Modelo V 57
Figura 3 – O Modelo U 58
Figura 4 – Modelo Entidade-Relacionamento proposto 92
Figura 5 – Tela principal proposta do *Twiki* 111
Figura 6 – Exemplo de criação de um requisito 113
Figura 7 – Exemplo de criação de uma classe 114

Lista de tabelas

Tabela 1 - Proposta de equipes de inspeção e de teste 78
Tabela 2 - Processo de inspeção de *software* 80
Tabela 3 - *Checklists* da fase de Análise 83
Tabela 4 - *Checklists* da fase de Projeto 84
Tabela 5 - *Checklists* da fase de Código 87
Tabela 6 - Processo de teste de *software* 89

Lista de abreviaturas

ANSI	American National Standards Institute
BCEL	Byte Code Engineering Library
E/S	Entrada e Saída
EOF	End of File
FDL	Free Documentation License
GNU	GNU is Not Unix
GPL	General Public License
HTML	HyperText Markup Language
IEEE	Institute of Electrical and Electronics Engineers
MER	Modelo Entidade-Relacionamento
OOA	Análise Orientada a Objetos
PDF	Portable Document Format
POO	Projeto Orientado a Objetos
RTF	Rich Text Format
XHTML	eXtensible Hypertext Markup Language

SUMÁRIO

Resumo
Abstract
Lista de ilustrações
Lista de tabelas
Lista de abreviaturas
Capítulo 1 11
1 Introdução 11
1.1 Motivação 11
1.2 Objetivo 13
1.3 Resultados obtidos e contribuições 13
1.4 Metodologia 14
1.5 Organização do trabalho 14
Capítulo 2 16
2 Revisão Bibliográfica 16
2.1 Introdução 16
2.2 Análise estática 19
2.2.1 Inspeção 20
2.2.2 Equipe de inspeção 27
2.2.3 Processo de inspeção 30
2.2.4 *Checklists* como auxílio à inspeção de *software* 34
2.2.5 Reuniões de inspeção 35
2.2.6 Melhorias de inspeção levantadas na bibliografia 36
2.2.7 Vantagens e desvantagens da inspeção 37
2.3 Teste dinâmico 38
2.3.1 Teste de caixa preta *(Black-Box testing)* e Teste de caixa branca *(White-Box testing* ou *Glass-box testing)* 40
2.3.2 Teste de regressão *(Regression testing)* 43
2.3.3 Teste de unidade 46
2.3.4 Teste de integração 46
2.3.5 Teste de sistema 47
2.4 Comparação entre análise estática e teste dinâmico 49
2.5 Experimentos de inspeção x teste 51
2.6 Formas do processo de teste – Verificação e Validação 53
2.7 Modelo U e Modelo V 56
2.8 Análise Orientada a objetos (OOA) 58
2.9 Projeto orientado a objetos (POO) 59
2.10 Conclusão 60
Capítulo 3 62
3 Ambientes colaborativos 62
3.1 Introdução ao *Wiki* 62
3.1.1 MediaWiki 65
3.1.2 *Twiki* 65
3.2 Outras ferramentas 75
Capítulo 4 77
4 Estratégia de inspeção de *software* 77
Capítulo 5 90
5 Implementação da Proposta 90

5.1 MER (Modelo Entidade-Relacionamento) 90
5.2 Ambiente *Twiki* proposto 93
Capítulo 6 108
6 Estudo de Caso 108
Capítulo 7 115
7 Conclusões 115
Referências Bibliográficas 117

Capítulo 1

1 Introdução

1.1 Motivação

Software é desenvolvido por seres humanos que naturalmente cometem erros. Erros são desvios ou não conformidade com a especificação ou com os resultados desejados (FAGAN, 1976).

Uma vez que erros sempre vão existir, o mais importante é poder identificá-los e providenciar medidas corretivas o mais breve possível, especialmente para os mais críticos. Eles podem estar presentes em diferentes fases do processo de desenvolvimento de *software*, não apenas nos códigos, mas também em requisitos do produto, especificações funcionais, manual de usuário e em outros documentos.

Normalmente a identificação e a correção dos erros são postergadas para fases posteriores do desenvolvimento de sistemas (LAITENBERGER et al., 2002). Quanto mais tarde for a detecção de erros, maiores serão os esforços e os custos envolvidos na sua identificação e remoção, pois eles poderão ser migrados para as fases subseqüentes do desenvolvimento de sistemas.

Estima-se que 30 a 50% do tempo de desenvolvimento seja dedicado aos testes, incluindo a detecção e remoção de erros. Os recursos dispendidos em fases de testes anteriores são recompensados futuramente. O tempo investido no planejamento de testes possibilita que o tempo do ciclo total de desenvolvimento seja menor. Se os testes são realizados sem planejamento, entra-se num ciclo vicioso interminável de detecção e correção de erros (RAMACHANDRAN,1996; KIT, 1995).

Um objetivo do teste de *software* é a garantia de um desenvolvimento de sistemas com qualidade. Conforme Harrold (2000), mais de 50% do custo do desenvolvimento de *software* é dispendido com testes, sendo que em aplicações críticas, o custo é ainda maior.

Não é simples medir o quão completamente um produto foi testado, nem é fácil avaliar o grau de profundidade em que os casos de teste foram aplicados ao produto. Podem-se identificar e apontar erros, mas não é possível afirmar que um artefato não os possua. Quanto maiores os volumes de dados e mais complexos

forem os sistemas, mais difíceis serão os testes. Em se tratando de sistemas multimídia/gráficos, a dificuldade dos testes existe devido à multiplicidade de entradas possíveis.

O teste consome tempo e geralmente é realizado de forma superficial. Ainda que a atividade de teste de *software* seja importante para a sua qualidade, normalmente não recebe a atenção necessária. Ele normalmente não é realizado de modo sistemático ou por meio de uma metodologia rigorosa, mas ao contrário, depende da intuição e experiência do Testador. A criação dos casos de teste e o teste propriamente dito são realizados sem nenhum critério. Tudo isso acaba comprometendo a qualidade do sistema. A atividade de teste normalmente não recebe recursos, prioridade e tempo dedicado adequados até que o desenvolvimento inicial esteja completo (PARVEEN, TILLEY E GONZALEZ, 2007; RYSER, BERNER E GLINZ ,1998).

O uso de técnicas de teste sistemáticas não é uma prática comum nas indústrias (HARROLD, 2000). Mesmo as técnicas de teste menos poderosas não têm sido aplicadas pelos profissionais e praticantes. Ainda que o reteste do *software* possa ser automatizado, emprega-se o modo manual.

O objetivo principal da engenharia de *software* é a entrega de produto de alta qualidade, com número reduzido de defeitos e apoiado por um processo de desenvolvimento de *software* maduro (WINKLER, RIEDL e BIFFL, 2005).

A inspeção é considerada uma das técnicas mais efetivas de garantia de qualidade na área de engenharia de *software* pois pode-se detectar e corrigir entre 50 a 90% dos erros presentes em um artefato de *software* (LAITENBERGER e ATKINSON, 1999). Porém, mesmo após 30 anos a partir de sua criação, a inspeção ainda não é totalmente compreendida e executada, muitas empresas ainda não se utilizam da sua aplicação em seus projetos (FAGAN, 2002).

Inspeção de *software*, que é o processo de realizar a leitura ou análise do artefato de *software* com o objetivo de identificar defeitos, foi criada no período em que o processo de *software* procedural era dominante. Apesar de sua eficácia e importância, não se têm informações suficientes e adequadas sobre a aplicação da inspeção em códigos orientados a objeto (DUNSMORE, ROPER e WOOD, 2003).

Os modelos orientados a objeto não têm sido considerados adequadamente por métodos de inspeção. Isso representa um problema devido ao fato do processo de desenvolvimento e paradigma orientado a objetos terem substituído o processo de desenvolvimento estruturado tradicional e atualmente, é o processo mais utilizado (LAITENBERGER e ATKINSON, 1999).

Segundo Kit (1995), não existem ferramentas para detecção de erros nas fases iniciais do desenvolvimento. Portanto, ter uma boa estratégia de inspeção é fundamental.

1.2 Objetivo

O objetivo deste trabalho é propor uma estratégia de inspeção de *software* que proporcione maior rigor no desenvolvimento de sistemas acadêmicos orientados a objetos, minimizando o tempo e os custos dispendidos nessa atividade.

1.3 Resultados obtidos e contribuições

Resultados obtidos

A proposta foi formalizada e implementada parcialmente em um ambiente de documentação cooperativa (*Twiki*). Uma aplicação foi documentada em parte como prova de conceito desse ambiente.

Contribuições

Citam-se como contribuições deste trabalho a proposta da estratégia de inspeção de *software*, a ampla pesquisa bibliográfica realizada para verificação do estado da arte do assunto em questão e a implementação parcial da proposta no ambiente *Twiki*.

O resultado obtido pode ser facilmente adaptado para empresas que desenvolvem projetos de *software*.

1.4 Metodologia

A metodologia deste trabalho consistiu inicialmente em um amplo levantamento bibliográfico sobre inspeção e teste de *software* em artigos científicos, teses, livros e pesquisas na *Internet*. A pesquisa foi realizada em uma diversidade de autores, fontes e em diferentes períodos cronológicos.

Foi proposta uma estratégia de inspeção de *software* embasada na pesquisa científica por ser o assunto de grande importância para o desenvolvimento de *software* e por ter sido constatada carência e necessidade de informação nesse tema.

É importante haver implementação da estratégia para se avaliar a sua aplicação. Dessa forma, ela foi implementada parcialmente por meio de um ambiente colaborativo que permite documentar e inspecionar todas as informações pertinentes e relevantes ao projeto de desenvolvimento de *software*.

Foi utilizado como exemplo de aplicação um projeto real desenvolvido por alunos do curso de graduação de Ciência da Computação.

1.5 Organização do trabalho

O Capítulo 2, “Revisão Bibliográfica”, apresenta os principais trabalhos relacionados ao tema que dão sustentação ao trabalho sendo proposto.

O Capítulo 3, “Ambientes colaborativos”, apresenta o *Wiki,* o MediaWiki e o *Twiki* como ambientes amigáveis que dispõem de poderosos recursos para permitir interação entre equipes de projeto, armazenamento de informações, controle de alterações, entre outros.

O Capítulo 4, “Estratégia de inspeção de *software*”, apresenta a estratégia de inspeção de *software* proposta propriamente dita.

O Capítulo 5, “Implementação da proposta”, apresenta de que forma a estratégia pode ser implementada para que possa ser aplicada diretamente em um projeto.

O Capítulo 6, “Estudo de caso”, apresenta a aplicação da proposta e avaliação de um caso real.

E finalmente o Capítulo 7, “Conclusões”, apresenta as conclusões e considerações finais deste trabalho.

Capítulo 2

2 Revisão Bibliográfica

2.1 Introdução

O Capítulo 2 traz um resumo dos principais trabalhos sobre o tema proposto sendo desenvolvido. Ressalta-se a importância de se ter uma estratégia e técnicas de inspeção e de testes para a melhoria da qualidade do produto e do processo de desenvolvimento.

Inicialmente, são apresentadas algumas definições usadas no decorrer do texto:

Artefato de *software* é qualquer produto de trabalho resultante das diferentes fases do desenvolvimento de sistemas. Os artefatos correspondentes às diferentes fases do desenvolvimento possuem características comuns tais como: devem ser claros e refletir completamente as necessidades do sistema especificadas nos requisitos, não devem conter informação irrelevante que comprometa o seu esclarecimento, não devem conter restrições desnecessárias de outros domínios que impactem a solução, não devem ser inconsistentes ou ambíguos para que o sistema possa ser implementado corretamente. Os revisores do artefato devem ter um conhecimento geral do domínio para avaliar se o sistema pode e é viável ser desenvolvido (BASILI et al., 1999).

Defeito é a propriedade que ocorre quando o artefato não corresponde aos requisitos de qualidade definidos.

Citam-se abaixo alguns tipos de defeitos envolvidos com o processo de desenvolvimento de *software* (CHAAR, HALLIDAY e BHANDARI, 1993):

- atribuição: valores atribuídos incorretamente ou não atribuídos;
- verificação: erros associados à ausência ou à incorreta verificação dos parâmetros/dados nas declarações condicionais;
- algoritmo: erros que envolvem alteração de estrutura de dados ou reimplementação de algoritmos;

- interface: problemas de comunicação entre módulos ou componentes;

- função: podem ser erros de interface do produto, do usuário final, interface com arquitetura de *hardware*;

- construção/pacote: problemas com sistemas de bibliotecas ou com o gerenciamento de controle de versão;

- documentação: erros nos manuais do usuário, nos manuais de instalação, nos comentários dos códigos.

O real significado do termo “defeito” não está relacionado necessariamente ao fator de qualidade “correção”, como normalmente é associado. Um documento pode estar correto (livre de erros); por outro lado, do ponto de vista de “manutenção”, pode ser categorizado como defeito se não tiver como característica boas práticas de projeto apropriadas para uma boa manutenibilidade que facilite modificações. Como exemplo de boas práticas de projeto citam-se alta coesão e baixo acoplamento (LAITENBERGER e ATKINSON, 1999).

Todos os esforços devem ser voltados para a identificação de defeitos antes que o produto seja disponibilizado em produção (FAGAN, 2002).

Qualidade de *software* é a combinação dos conceitos baixo número de defeitos e alta satisfação do usuário. Uma das causas de baixa qualidade de *software* é a correção de defeitos de maneira inadequada (HEISER, 1997).

Qualidade de software

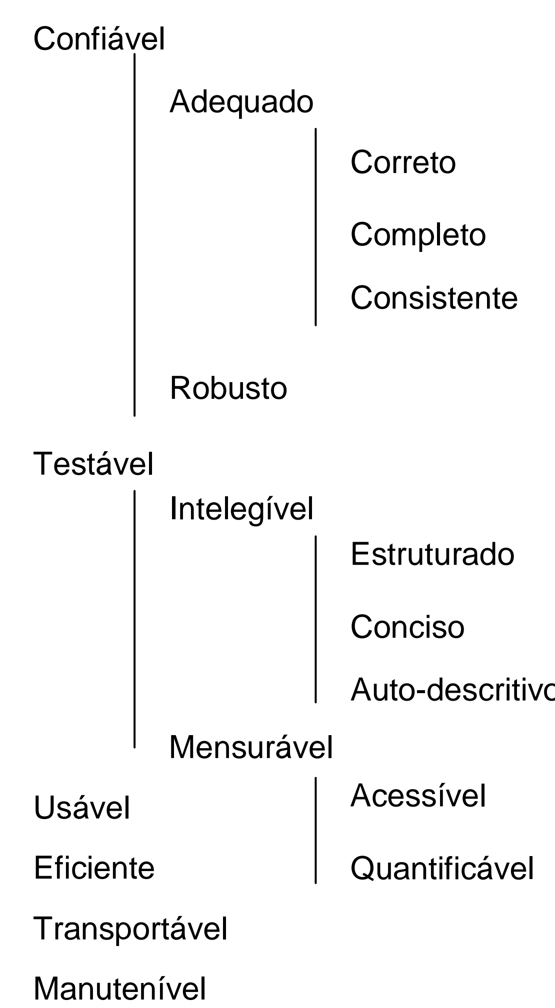


Figura 1 - A hierarquia dos atributos de qualidade de *software*
Fonte: ADRION; BRANSTAD; CHERNIAVSKY, 1982

Conforme exposto na figura acima, os principais atributos de qualidade de *software* são os que se situam à esquerda da figura, ou seja, Confiabilidade, Testabilidade, Usabilidade, Eficiência, Transportabilidade e Manutenibilidade. Os atributos que se encontram mais abaixo e à direita correspondem às características dos atributos de qualidade. Por exemplo, o atributo de qualidade Testabilidade possui as características de Inteligibilidade e Mensurabilidade. Inteligibilidade por sua vez requer que o produto em cada fase possua as características de ser Estruturado, Conciso e Auto-descritivo.

Muitas vezes, um determinado fator de qualidade pode entrar em conflito com outros. Assim, para cada tipo de projeto se priorizam alguns fatores de qualidade de *software* mais adequados.

Enquanto é difícil afirmar ou mensurar que um produto de *software* tem boa qualidade, é fácil identificar quando há má qualidade devido à presença visível de erros ou ao mau funcionamento do sistema (ADRION, BRANSTAD, CHERNIAVSKY, 1982).

A atividade de teste pode ser dividida em 2 categorias: análise estática e teste dinâmico.

2.2 Análise estática

Na análise estática, pode-se fazer a leitura do código-fonte do programa, analisar a sua estrutura, verificar se o programa está de acordo com os requisitos especificados, analisar qualquer documento técnico como projeto, planos de teste, documentação de usuário, verificar se as propriedades apresentadas são corretas e verdadeiras, sem a necessidade da execução do programa, não importando o trajeto e o resultado de execução (OSTERWEIL, 1996).

De acordo com Kelly e Shepard (2000), o termo geral inspeção de *software* compreende a revisão estática do produto de *software* para detecção e registro de defeitos.

Os modelos de revisão de *software*, como por exemplo o *walkthrough* (roteiro) e a inspeção, são classificados de acordo com o local em que as atividades de detecção de defeitos ocorrem (CARVER, 2003).

No *walkthrough*, a detecção de defeitos ocorre totalmente durante a reunião em equipe. O autor do documento participa da revisão do documento junto com a equipe. Um ponto fraco destacado é que os resultados dessa revisão dependem muito da qualidade da apresentação do autor do documento.

Em uma inspeção formal, a detecção de defeitos ocorre durante a reunião em equipe e também durante a preparação individual dos Inspetores.

O *walkthrough* e a inspeção são similares, ambos necessitam de uma equipe normalmente conduzida por um Moderador e se caracterizam pela leitura de produtos de trabalho como por exemplo, requisitos, especificações ou código (ADRION, BRANSTAD, CHERNIAVSKY, 1982). Porém, o *walkthrough* é conduzido apenas por membros da equipe de desenvolvimento, que analisam parte específica

do produto (SWEBOK, 2001). Por outro lado, a inspeção de *software* inclui também além da equipe de desenvolvimento, outros perfis de profissionais como Leitor, Inspetor e Relator.

Inspeções caracterizam-se por uma preparação individual antes da reunião. O líder da equipe de inspeção orienta os participantes, de acordo com seu papel de revisor, sob qual ponto de vista devem ler o material. Busca-se que seja encontrada a maioria dos erros durante essa preparação, embora na reunião de inspeção também serão encontrados erros adicionais.

Por outro lado, *walkthrough*s são menos formais que inspeções porque não há uma preparação antes da reunião, o apresentador é o criador do documento inspecionado e os outros participantes não têm uma carga de trabalho pesada. O objetivo principal não é encontrar erros ou defeitos, mas sim, familiarizar-se com o produto, promover a discussão e a comunicação com as pessoas principais na reunião (KIT, 1995).

Em comparação com o *walkthrough*, a inspeção normalmente tem menores variações e maior repetitibilidade nos resultados (FAGAN, 1976). Além disso, a inspeção se mostrou ser melhor no potencial de detecção de defeitos (LAITENBERGER et al., 2002).

Embora inspeções e *walkthrough*s sejam técnicas de análise humana, são técnicas formais, estruturadas, rigorosas e definidas com cautela (OSTERWEIL, 1996).

2.2.1 Inspeção

A inspeção é um modo formal e estruturado de verificação, uma forma de avaliar e garantir a qualidade do produto inspecionado, a identificação e correção de defeitos do produto e do processo de desenvolvimento, redução do tempo e do custo envolvidos com o projeto.

A inspeção pode ser aplicada sobre qualquer artefato como requisitos, especificações, projeto, código, teste pois não exige artefatos executáveis. É uma forma de verificação eficiente pois é uma técnica de teste que pode ser utilizada já

nas fases iniciais do processo de desenvolvimento de *software*. Quanto mais cedo se detectarem falhas, menores serão os esforços e custos de correção.

A inspeção de *software* contribui para a detecção e remoção de defeitos do sistema, assegurando que os artefatos de *software* correspondam às necessidades dos usuários (BASILI et al., 1999). Assim, tem-se uma maior qualidade do produto entregue, a princípio para a atividade dos Testadores e conseqüentemente, ao usuário final (CHAAR, HALLIDAY e BHANDARI, 1993). Trata-se uma análise visual de um produto de trabalho, é caracterizada por uma estrutura rígida, pelo uso de *checklists* (listas de verificação) e critérios de avaliação pré-determinados (HEISER, 1997).

A literatura cita que Michael Fagan é o criador e fundador da inspeção no final dos anos 70, embora outros pesquisadores também tenham participado do desenvolvimento da inspeção, conforme citado por Laitenberger et al. (2002).

Segundo Fagan (1976), o profissional que inspeciona não deve ser o próprio criador do artefato. Muitas vezes, é mais fácil analisar e identificar erros em um artefato sob o ponto de vista externo. O criador do artefato pode não conseguir visualizar os próprios erros, ao passo que alguém que esteja desenvolvendo outro artefato pode visualizar.

O processo de inspeção envolve uma análise e leitura passo-a-passo de um artefato. Durante essa análise, deve-se confrontar o produto com erros históricos comuns, verificar conformidade com a especificação e com padrões definidos (ADRION, BRANSTAD, CHERNIAVSKY, 1982).

Segundo Gilb (2000), o processo de inspeção envolve a detecção e a prevenção de defeitos:

- a detecção de defeitos é um processo reativo que ocorre devido à existência de defeitos de modo que ações corretivas sejam providenciadas em seqüência;
- a prevenção de defeitos é um processo pró-ativo pois atua na avaliação dos defeitos detectados anteriormente, constitui uma forma de aprendizagem dos defeitos ocorridos para que não ocorram em condições similares.

Os defeitos considerados principais (*major defects*) são aqueles mais críticos e com conseqüências mais desastrosas. Devem-se concentrar esforços na detecção

e correção dos defeitos principais e não desperdiçar esforços com defeitos secundários (*minor defects*) (GILB, 2000). Para identificar os defeitos principais, podem-se utilizar *checklists* que verificam se o documento analisado está em conformidade com os padrões definidos e se as ocorrências caracterizam-se como defeitos. Há uma relutância em se classificar os defeitos como principais para que o responsável pelo defeito não seja avaliado desfavoravelmente pelo seu superior hierárquico (WELLER, 1993).

As falhas têm diferentes graus de dificuldades de detecção, ou seja, algumas são mais difíceis de serem localizadas do que outras. Além disso, há tipos de erros difíceis de serem detectados independentemente do método utilizado e há tipos de erros cuja detecção varia de método para método (MYERS, 1978; SCHNEIDER, MARTIN E TSAI, 1992).

Defeitos não muito fáceis de se identificar podem ter conseqüências mínimas, enquanto defeitos fáceis de se identificar podem ter grandes conseqüências, ou seja, a facilidade na localização de defeitos não tem uma relação direta com a gravidade da sua conseqüência (KELLY e SHEPARD, 2000). É importante diferenciar a dificuldade de localização e da correção dos defeitos da sua gravidade e conseqüência.

A inspeção deve ser utilizada freqüentemente de modo planejado e sistemático (ACKERMAN, BUCHWALD e LEWSKI, 1989). Dependendo da complexidade do sistema, várias inspeções podem ser necessárias e cada inspeção pode ser aplicada sobre um subconjunto do sistema, como por exemplo, parte dos requisitos, dos casos de uso, dos diagramas de classe, entre outros (BASILI et al., 1999).

Inspeção de requisitos

O objetivo da fase de requisitos é garantir que as necessidades dos usuários sejam bem entendidas antes que sejam traduzidos para a fase do projeto. O desafio dos requisitos é distinguir as reais necessidades dos usuários dos seus desejos e vontades. Requisitos incompletos ou incorretos resultam em projeto inadequado e conseqüente implementação incorreta (ADRION, BRANSTAD, CHERNIAVSKY, 1982). Se os requisitos são mal-entendidos ou sujeitos constantemente a mudanças,

o processo de inspeção terá pouco efeito sobre o sucesso ou falha do projeto (WELLER, 1993). Deve-se analisar os requisitos com relação à sua completude, consistência e precisão (ADRION, BRANSTAD, CHERNIAVSKY, 1982).

Segundo a definição do IEEE (Institute of Electrical and Electronics Engineers) / Ansi (American National Standards Institute), “requisito” é a condição que um sistema deve atender conforme definido em documentos de especificação (ou contrato) para que o usuário atinja um objetivo (KIT, 1995).

É importante se ter em mente que os requisitos não são estáticos, eles mudam. Por isso, deve-se ter o controle sobre as mudanças através da verificação dos requisitos e gerenciamento de configuração das diferentes versões.

O documento de requisitos do usuário deve descrever detalhadamente as funcionalidades do sistema proposto e as características de desempenho desejadas. Trata-se de um acordo formal entre o usuário e o desenvolvedor sobre o que precisa ser desenvolvido, devendo ser completo, objetivo, não ambíguo e claro (SCHNEIDER, MARTIN e TSAI, 1992).

A qualidade de um documento de requisitos depende de quem escreverá o documento e o quanto está familiarizado com padrões. A qualidade dos requisitos é um indicador do nível de maturidade da organização. Precisa-se haver um consenso de como criar um documento de requisitos, caso contrário, o desenvolvimento e os testes serão difíceis (KIT, 1995).

De acordo com Kit (1995), na inspeção é importante que se detectem omissões, se ter uma visão crítica de questionar e identificar o que deveria estar presente e explícito na especificação e não está. A inspeção deve não somente analisar e verificar se o artefato atende às especificações, mas também apontar itens necessários que não tenham sido levados em consideração. Além de identificar omissões, formalizar requisitos contribui também para identificar imprecisão, ambigüidade ou contradições na especificação do sistema, o que compromete a qualidade do sistema e pode resultar em erros futuros.

Detectar omissões não é uma tarefa fácil. Para assegurar completude dos requisitos, pode-se definir cenários esperados de uso do sistema, dados de teste e resultados almejados. Definir casos de teste com os resultados esperados contribui para verificar a testabilidade dos requisitos. Requisitos vagos ou não testados

resultam em produtos sobre os quais não se pode afirmar que são os produtos exigidos (ADRION, BRANSTAD, CHERNIAVSKY, 1982).

Conforme Ryser, Berner e Glinz (1998), os modos de avaliar requisitos são: revisão de especificação de requisitos, protótipo e simulação.

- revisão da especificação: realizado por um grupo de pessoas para analisar documentos com o objetivo de encontrar falhas e anormalidades;
- protótipo: utilizado para validação de requisitos, do atendimento de expectativas de usuários, para avaliação da conformidade do sistema em relação à interface do usuário, interação e comportamento funcional. Não é indicado para avaliar se os requisitos estão completos e consistentes;

  O protótipo contribui para que o usuário tenha uma idéia do ambiente visual, da navegação e das funcionalidades do sistema. Trata-se de uma ferramenta importante pois pode-se avaliar o sistema antes dele estar totalmente desenvolvido e finalizado.

  Conforme Carrington (1997), o protótipo é uma forma de teste de *software* em que obtém-se o imediato retorno do usuário. As entradas são selecionadas para a execução e os resultados obtidos são comparados com os esperados.
- simulação: é utilizado para validação do comportamento do sistema de acordo com situações definidas.

Os tipos de falhas existentes em documentos de requisitos de usuário podem ser classificados em (SCHNEIDER, MARTIN e TSAI, 1992; BERLING e THELIN, 2003):

- informação omitida
    - funcionalidade: qualquer funcionalidade importante e necessária ao sistema que tenha sido omitida da especificação de requisitos;
    - interface: quando não tenha sido definida a interface do sistema com objetos externos ao escopo do sistema;

    - desempenho: quando se omite a informação detalhada do desempenho desejado do sistema ou a informação é insuficiente, quando os níveis de aceitação do desempenho não são totalmente detalhados;
    - ambiente: informações como tipos de *hardware*, *software*, base de dados e outras relacionadas ao ambiente necessário à execução do sistema não totalmente definidas.
- informação errônea
    - informação ambígua: informação imprecisa ou confusa, causando desentendimento;
    - informação inconsistente: quando as informações se contradizem, ambas não podem ser simultaneamente verdadeiras;
    - fato incorreto: informação errada contida na especificação de requisitos;
    - seção errada: informação essencial deslocada da seção ou do local apropriados.
- informação extra: informação desnecessária que não é utilizada deve ser descartada (BASILI et al., 1999).

A fase de análise e inspeção de requisitos é uma das mais importantes do ciclo de desenvolvimento de *software* pois é a partir da fase de requisitos que as fases subseqüentes irão se basear e serão desenvolvidas.

Segundo Andrews (2006), a inspeção de requisitos proporciona um maior custo-benefício e maiores ganhos ao se detectarem falhas antecipadamente nas fases iniciais do que permitir que sejam detectadas em fases posteriores do desenvolvimento, como por exemplo, na fase do código. Nesse caso, haveria o retrabalho da correção dos requisitos e do projeto, além da correção do código. O problema das falhas serem detectadas em fases posteriores é de serem propagadas e migradas para as fases subseqüentes, resultando em maiores custos de detecção e remoção. Estima-se mais do que 50% dos erros sejam introduzidos na fase dos requisitos (KIT, 1995).

Investir recursos e tempo no retrabalho e na correção de defeitos impede o investimento de esforços em novas funcionalidades, resulta em atraso na entrega do produto ou na entrega do produto com funcionalidades reduzidas. Deve-se minimizar o retrabalho e concentrar esforços em funcionalidades adicionais já que os clientes procuram e compram produtos por novas funcionalidades.

Retrabalho é um fator importante a ser considerado e avaliado no esforço total de desenvolvimento de *software*. Estima-se que de 30 a 80% do esforço total de desenvolvimento seja dispendido no retrabalho resultante da identificação e correção de defeitos (FAGAN, 2002).

Inspeção de projeto

A inspeção de projeto antecede a inspeção de código. O projeto é concebido na fase inicial do processo de desenvolvimento de *software*.

A inspeção de projeto deve ser realizada para identificar casos de omissões, lógica imprecisa, incompatibilidades da interface do módulo, inconsistências da estrutura dos dados, erros de E/S (Entrada e Saída) e interface de usuário inadequada. Deve-se verificar se o projeto detalhado é completo e consistente com os requisitos e com o projeto preliminar (ADRION, BRANSTAD, CHERNIAVSKY, 1982).

Um erro de projeto pode ocorrer por exemplo, se os diagramas do projeto não forem consistentes entre si ou se eles não refletirem adequadamente os requisitos (BASILI, 1999).

A priorização do desenvolvimento do código em detrimento de uma definição e desenvolvimento completo do projeto pode resultar em falhas (BERLING e THELIN, 2003).

Inspeção de código

A inspeção de código, conforme citado por Schach (1996) e Laitenberger (1998), envolve três passos: identificação, coleta e correção de defeitos.

Antes de qualquer coisa, precisam-se detectar os defeitos existentes no código. A coleta de defeitos é a organização e análise dos defeitos. Uma vez identificados os defeitos, organizados e analisados, devem-se eliminá-los para se ter a conformidade com as especificações e se ter a garantia da qualidade do código. A correção de defeitos deve ser realizada pelo autor do módulo inspecionado.

Segundo Kelly e Shepard (2001), a inspeção de código proposta por Fagan (30 anos atrás) ainda é a forma mais praticada de inspeção e normalmente é mais discutida do que a inspeção de requisitos e de projeto. O código é mais fácil de ser inspecionado em relação, por exemplo, à inspeção de requisitos, os quais geralmente são especificados inadequadamente, dificultando a inspeção (FAGAN, 2002).

Pelo fato da inspeção poder detectar erros o mais breve possível, os custos de correção de erros são reduzidos em cerca de 39% para a inspeção de código e cerca de 44% para a inspeção de projeto (LAITENBERGER e ATKINSON, 1999).

As revisões do projeto e a inspeção de código podem reduzir os custos e o cronograma em 30% (HEISER, 1997).

2.2.2 Equipe de inspeção

Conforme Fagan (1976), um número ideal para formar a equipe de inspeção é de quatro pessoas. A equipe é composta de:

- moderador: profissional principal, é o mediador, deve ser cuidadoso, ser um programador ou técnico competente, possibilita interação entre a equipe, deve ter capacidade de gerenciar equipe, de liderar a inspeção, de agendar e conduzir reuniões, de apresentar resultados da inspeção, de realizar acompanhamentos (*follow-up*);
- projetista: programador responsável por criar o projeto do programa;
- codificador/implementador: programador responsável por traduzir projeto em código;
- testador: programador responsável por criar e executar casos de teste e testar o produto de *software*.

Crossman (1977) adicionalmente inclui também como membro da equipe de inspeção:

- representante do Grupo de Manutenção: responsável por manter cada sistema. Um membro da equipe de inspeção deve ser desse grupo.

Gilb (2000) inclui também outros membros da equipe de inspeção:

- autor: criador ou mantenedor do documento inspecionado. Pode ajudar a identificar defeitos, mas não pode ser o moderador;

  O papel do Autor do documento é importante para que os Inspetores não dispendam muito tempo e esforço em sua análise e entendimento. Além disso, esse perfil promove a comunicação junto aos Inspetores para a adequada correção dos erros (ACKERMAN, BUCHWALD e LEWSKI, 1989).

- leitor: lê e descreve o material durante a inspeção a todos os membros da equipe;

  O papel de Leitor normalmente é o menos entendido e o que mais é desconsiderado no processo de inspeção de uma organização. Esse perfil é importante porque enfatiza a atenção de todos os Inspetores nos detalhes do documento inspecionado. Para isso, o Leitor ressalta e sumariza cada subseção do material (ACKERMAN, BUCHWALD e LEWSKI, 1989).

- relator: categoriza e registra defeitos ou outros assuntos pertinentes durante a reunião de inspeção. Essa atividade pode ser feita pelo moderador em caso de grupos de inspeção pequenos;
- inspetor: procura por defeitos potenciais. Todos os membros da equipe devem ser Inspetores, além de outras responsabilidades adicionais.

Para Fagan (1976) e Crossman (1977), as funções adicionais dos membros da equipe de inspeção descritas acima por Gilb (2000) estão implícitas distribuídas entre os membros da equipe.

Segundo Fagan (1976), ter um moderador com treinamento adequado para a condução do processo e ter o apoio da gerência também são fundamentais para o sucesso da inspeção.

Fagan (1976) enfatiza que o profissional que inspeciona não deve ser o criador do próprio artefato. Embora não categorize explicitamente o autor do documento como membro da equipe de inspeção, no processo de inspeção o autor do documento é citado como membro da equipe. Já Gilb (2000) inclui explicitamente o autor do documento inspecionado como membro da equipe de inspeção.

A autora deste trabalho propõe que o autor do documento inspecionado possa participar como membro da equipe de inspeção, mas que não exerça a função de Inspetor do documento. Ao contrário, que o Autor possa contribuir com detalhamento, esclarecimento do documento e também com correções ou alterações necessárias.

A partir do levantamento sobre a equipe de inspeção, segue abaixo uma classificação geral:

- moderador;
- autor;
- leitor;
- relator;
- inspetor;
- projetista;
- codificador/implementador;
- testador;
- representante do Grupo de Manutenção.

A eficácia, eficiência e o sucesso da equipe de inspeção dependem de quão familiarizada a equipe esteja com o produto inspecionado, da capacidade individual de cada membro da equipe e também de sua capacidade de trabalhar em equipe (WELLER, 1993; FAGAN, 2002).

2.2.3 Processo de inspeção

Processo de inspeção de Fagan – Visão geral do método

Michael Fagan, o criador da inspeção, era engenheiro de controle de qualidade e propôs um processo de inspeção na IBM nos anos 70.

Conforme Doolan (1992), o método proposto pode ser aplicado sobre qualquer artefato e prevê as seguintes etapas:

- a inspeção se inicia pela escolha do Moderador, que pode ser feita por meio de um grupo de garantia da qualidade. O Moderador recebe cópia do documento a ser inspecionado, que precisa satisfazer critérios pré-definidos. Compõe também um grupo de inspeção de até 5 pessoas, incluindo o autor do documento inspecionado;
- o Moderador faz um planejamento para determinar quantos ciclos serão necessários, quem deve participar de cada ciclo da inspeção (GILB e GRAHAM, 1993);
- uma vez definida a equipe de inspeção, é agendado o *"Kick-off meeting"* (reunião inicial) com essa equipe de inspeção em torno de 20 a 30 minutos para que todos tenham conhecimento do objetivo da inspeção, acesso ao documento inspecionado por meio da distribuição de cópias do material e o que se espera de cada membro. A cada membro da equipe são atribuídas a responsabilidade e tarefa de inspeção sobre determinado aspecto do documento;

  É a fase da primeira reunião da equipe de inspeção em que se tem a explicação sobre a concepção do sistema, interface com sistemas existentes e os requisitos do sistema (FAGAN, 1976). Os Inspetores precisam estar familiarizados com o ambiente e com o método de desenvolvimento (TERVONEN, 1996).

  Nessa fase, o objetivo é que todos os Inspetores tenham o correto entendimento do documento a ser inspecionado para que possam inspecioná-lo facilmente. Para tanto, o autor do documento deve explanar por meia ou uma hora sobre o documento em questão (ACKERMAN, BUCHWALD e LEWSKI, 1989).

- após o *"kick-off meeting"*, cada membro da equipe de inspeção definida deve analisar o material de inspeção, compará-lo com padrões estabelecidos, com os requisitos definidos, avaliar se o material está completo e correto, identificar os defeitos existentes no artefato, detectar omissões de acordo com padrões e *checklists* definidos;

  Trata-se de uma preparação individual, análise e entendimento do material de inspeção e dos documentos envolvidos. É nessa fase em que a maior parte dos erros é identificada (FAGAN, 1976).

  Na fase de preparação, os Inspetores devem registrar o tempo dispendido nessa atividade para posterior avaliação de produtividade junto ao Moderador (ACKERMAN, BUCHWALD e LEWSKI, 1989).

- após a inspeção de um artefato ou documento, pode-se ter os seguintes resultados (CROSSMAN, 1977):
  - reescrita: refazer ou reescrever o documento inspecionado. Essa decisão deve ser feita pelo Gerente de Projeto;
  - revisão: o Gerente de projeto deve decidir se apenas uma revisão é suficiente e não seja necessário reescrever o documento;
  - aceite: aceitar o documento do modo como foi concebido, sem necessidade de outras inspeções ou ter apenas revisões sem reinspeções.
- a reunião de registro dos defeitos (*defect-logging meeting*) de até 2 horas é agendada com a participação da equipe toda para discussões dos defeitos encontrados, seu registro e documentação;
- na fase de classificação dos erros, deve-se identificar, organizar, classificar e agrupar os tipos de erros, criando-se uma base histórica de conhecimento dos erros;

  Deve-se mensurar e classificar os tipos de erros, a freqüência de ocorrência de cada um, tempo maior dispendido na identificação e correção dos diferentes tipos de erros. Tendo-se um histórico e registro da análise dos resultados da

inspeção, objetiva-se uma melhoria do processo de desenvolvimento (FAGAN, 1976).

- a reunião de análise e identificação da causa (*root causes*) dos defeitos encontrados deve ser realizada para prevenção de erros futuros, podendo sugerir alterações no processo desenvolvimento como melhoria contínua. Esse tipo de reunião não é uma parte explícita do processo original proposto por Fagan (DOOLAN, 1992). A análise da causa dos defeitos é uma análise qualitativa de cada defeito encontrado, um a um, realizada por uma equipe com conhecimentos adequados. Essa análise proporciona retorno aos desenvolvedores em qualquer etapa do processo de desenvolvimento (CHAAR, HALLIDAY e BHANDARI, 1993);

  O objetivo do retorno dos resultados da inspeção deve ser o benefício e a melhoria dos processos, do produto e sob nenhuma hipótese avaliar e controlar a produtividade do profissional (FAGAN, 1976).

- segue-se o procedimento de correção dos erros na fase em que ocorrerem, evitando que os erros sejam migrados para fases posteriores pois nesse caso, o esforço e custos de correção serão maiores;
- o Moderador deve se assegurar que ações corretivas tenham sido realizadas para cada um dos defeitos levantados; deve monitorar, controlar e acompanhar cada erro identificado. Ele também deve coletar estatísticas para controle de qualidade do processo de inspeção;
- como parte do processo de inspeção, cita-se a melhoria contínua do processo, ou seja, monitoramento, análise e ações corretivas e de aperfeiçoamento do processo (FAGAN, 2002).

Mashayekhi et al. (1993) apresenta uma proposta de inspeção dividida em 4 fases:

Inicialização da inspeção:

- identificar o Moderador, Revisores e Relatores;

- obter consenso entre os revisores;
- convidar novos participantes para a inspeção;
- identificar critérios de inspeção.

Preparação para inspeção:

- reunião de introdução;
- fornecer materiais de inspeção;
- revisões individuais;
- levantar falhas e ambigüidades.

Condução da inspeção:

- discussão dos itens a partir da lista de falhas;
- obter consenso sobre as falhas;
- levantar falhas através de discussões;
- registrar ações;
- determinar estado da inspeção.

Pós-inspeção:

- revisar estado da inspeção;
- atualizar relatórios.

Os dados da inspeção contribuem para a tomada de decisões pela equipe de desenvolvimento em tempo real sobre qual o melhor processo a seguir (WELLER, 1993).

Um modo de auxiliar o processo de inspeção na identificação de defeitos são as técnicas de leitura, como por exemplo, leitura baseada em perspectiva, que é uma técnica aplicada com sucesso nas empresas. Nesse tipo de leitura, a inspeção é avaliada sob o ponto de vista do usuário (*stakeholder)* do artefato. Assim, há interesse particular no aspecto de qualidade do artefato e será avaliado o fator que for mais relevante para aquele usuário, e não todo o documento.

2.2.4 *Checklists* como auxílio à inspeção de *software*

Segundo Brykczynski (1999), *checklists* são um meio formal de verificação, uma forma de auxiliar a inspeção de *software* na detecção de defeitos e podem ser aplicados sobre produtos de trabalho como requisitos, projeto, código e planos de teste. Têm o objetivo de garantir consistência da informação, que deve ser correta e completa. Os *checklists* ajudam a instruir os Inspetores sobre o que procurar, como identificar defeitos e direcionam o trabalho do Inspetor (TERVONEN, 1996). Além disso, possuem adicionalmente outros objetivos além da detecção de defeitos tais como: reuso de bibliotecas, melhoria da manutenibilidade, decisões relacionadas a custo (BRYKCZYNSKI,1999).

As áreas de desenvolvimento e teste geralmente têm diferentes *checklists*. *Checklists* de teste preocupam-se com a confiabilidade e usabilidade do produto. Por outro lado, *checklists* de desenvolvimento preocupam-se com manutenibilidade e padrões de codificação (KIT, 1995). É importante ter *checklists* genéricos e também os criados especificamente para um dado projeto. Podem-se criar *checklists* próprios para tudo que se deseja revisar.

Para tirar o máximo progresso da verificação, é necessário que se tenha uma pessoa responsável pela atualização e manutenção dos *checklists*.

São citadas a seguir características dos *checklists (*BRYKCZYNSKI, 1999):

- devem ser atualizados e retratar a realidade;
- não devem conter mais do que uma página, não devem ser muito extensos;

- podem ser representados na forma de uma questão, como verificação de algum aspecto do artefato inspecionado;
- não devem ser muito genéricos, devem ser claros, objetivos e diretos.

2.2.5 Reuniões de inspeção

Conforme Votta Jr. (1993), as reuniões de inspeção não têm apenas vantagens, ao contrário, custam muito. Porém, permitem que haja um processo definido e sistemático. Segundo Bradac, Perry e Votta (1993 apud VOTTA JR., 1993), 20% do intervalo de tempo do levantamento de requisitos e do desenvolvimento do projeto é dispendido na espera e organização dos revisores se reunirem. O ideal é minimizar o número de participantes da reunião de inspeção.

Programadores normalmente evitam que haja conflitos com suas próprias opiniões e crenças, não valorizando as reuniões de inspeção. Porém, elas são importantes para neutralizar esse efeito e permitir maior interação entre os envolvidos.

Algumas razões para haver as reuniões de inspeção: sinergia, educação, cronograma e prazo.

- sinergia: inspeção por meio de reuniões tem maior probabilidade de identificação de defeitos não encontrados na inspeção individual, pois há maior discussão e troca de idéias em um ambiente de trabalho em equipe;
- educação: oportunidade de profissionais mais experientes trabalharem em conjunto com os menos experientes de modo que esses possam aprender e se desenvolver com os mais experientes. Trata-se de uma forma de aprendizagem, troca de experiências e de conhecimento do processo de desenvolvimento, processo de revisão e de inspeção;
- cronograma e prazo: tem-se uma maior definição e planejamento de prazos e de cronograma para serem seguidos.

2.2.6 Melhorias de inspeção levantadas na bibliografia

Kelly e Shepard (2000) propuseram uma nova técnica de inspeção (técnica orientada à tarefa) que tem o objetivo de fornecer estrutura e orientação ao Inspetor individual para melhoria da eficácia e melhor aproveitamento do tempo individual. É um processo de inspeção com maior enfoque em contribuição individual do que em contribuição de equipe.

A inspeção individual é um processo sistemático, específico e distinto:

- sistemático: cada Inspetor possui claras instruções sobre como proceder a inspeção;
- específico: cada Inspetor deve focar em um aspecto específico da inspeção do artefato e em determinadas classes de defeitos;
- distinto: cada Inspetor deve avaliar um determinado aspecto do artefato diferente de outro Inspetor.

Segundo Knight and Myers (1993 apud KELLY e SHEPARD, 2000), a inspeção é auxiliada por *checklists*. Cada fase ou inspeção parcial enfatiza uma única propriedade específica do produto. Deve ser selecionado o profissional de inspeção que possua os perfis necessários e adequados para cada fase da inspeção. As tarefas devem ser distribuídas de acordo com a experiência anterior desses profissionais.

A nova técnica proposta ajuda na análise e no melhor entendimento do código de *software* e de outros produtos de trabalho. Experiências demonstram que profissionais de inspeção experientes utilizando essa nova técnica de inspeção orientada à tarefa identificam maior probabilidade de defeitos em se tratando de códigos que exijam maior aprofundamento da compreensão do código.

Com essa nova técnica proposta, a inspeção é considerada mais como processo de avaliação do que um processo de correção, diferente da técnica de Fagan.

2.2.7 Vantagens e desvantagens da inspeção

As vantagens da inspeção são a capacidade de análise na detecção de erros nas fases iniciais do desenvolvimento de sistemas. Por outro lado, as desvantagens são a não garantia de que a análise humana é livre de erros e assim, não se garante 100% de detecção de erros (DUNSMORE, ROPER e WOOD, 2003). Sabendo-se que erros sempre vão ocorrer, assegura-se a identificação dos defeitos, mas não se pode afirmar que não existirão mais erros.

Segundo Myers (1978), *walkthroughs*/inspeções tem um bom custo-benefício pois ao se detectarem erros em fases iniciais do desenvolvimento, tem-se menores custos envolvidos e maior probabilidade de correção. Além disso, *walkthroughs*/inspeções permitem que os profissionais tenham maior interação e enfatizam mais a lógica do programa do que a entrada e saída de dados.

A inspeção permite melhoria da produtividade, melhor controle e gerenciamento de projeto, garantia da confiabilidade, qualidade e melhoria do produto e do processo de desenvolvimento, redução dos custos e do tempo de entrega do produto, melhor qualidade da manutenção e maior envolvimento das pessoas no processo geral de desenvolvimento (FAGAN, 1976; CROSSMAN, 1977; WELLER, 1993).

Citam-se alguns benefícios da inspeção (FAGAN, 2002):

- redução de defeitos reportados pelo usuário;
- maior satisfação do cliente;
- maior cumprimento dos prazos;
- conscientização e aprendizagem dos autores dos artefatos em evitar a disseminação dos defeitos a partir de práticas de aplicação da inspeção nos produtos de trabalho.

Quanto mais efetiva for a aplicação da inspeção na identificação dos defeitos, maior a probabilidade de se cumprirem o prazo e os custos planejados (FAGAN, 2002).

Antes a inspeção não era muito valorizada pelos programadores pois era vista como algo desnecessário. Atualmente, a inspeção é mais valorizada e há maior aceitação do processo devido aos benefícios resultantes (FAGAN, 1976).

Pesquisadores afirmam que inspeção é eficaz. Por outro lado, o mercado da indústria afirma que a inspeção consome e demanda tempo, é custosa, ineficaz e difícil. Não há tempo disponível para a inspeção, há dificuldades com o agendamento e execução das reuniões de inspeção, ainda que haja aspectos positivos como a melhoria da qualidade e retorno de investimento (JOHNSON, 1998 apud KELLY e SHEPARD, 2000; KELLY e SHEPARD, 2001). Além disso, cita-se a resistência de algumas pessoas na mudança de rotina de trabalho como um fator impeditivo para a disseminação e o efetivo uso da inspeção (FAGAN, 2002).

2.3 Teste dinâmico

Conforme Osterweil (1996), teste dinâmico é a execução do programa e a análise dos resultados. O teste de *software*, que exige código executável, compreende a execução dos casos de teste e comparação dos resultados obtidos contra o documento de especificação e os resultados esperados. Nesse tipo de teste, a especificação do comportamento do programa é avaliada pela análise das saídas (ou resultados) resultantes da execução dos parâmetros de entrada (ou casos de teste). Analisa-se o quão correto e completo são os requisitos e documentos de projeto. O objetivo do teste é analisar desvios em relação ao resultado esperado, corrigir e prevenir erros antes que o produto seja entregue ao cliente (WINKLER, RIEDL e BIFFL, 2005). Cita-se também como objetivo de negócio do teste de *software* o ato de analisar os riscos de negócio relacionados ao *software* e reportá-los à gerência (HEISER, 1997).

O teste dinâmico analisa e compara comportamentos não funcionais tais como confiabilidade, precisão, desempenho e usabilidade com os requisitos especificados.

As vantagens do teste dinâmico são a execução dos testes e a imediata análise do comportamento do programa no ambiente de execução. Como desvantagem, cita-se não se poder afirmar que não ocorrerão falhas caso se

reexecutem testes que já foram executados anteriormente sem falhas. Segundo Dijkstra (1972 apud CARRINGTON, 1997), o teste identifica a presença de erros, mas não garante a ausência total de erros.

O teste pode ser dividido em fases de planejamento, execução e análise. A fase de planejamento identifica casos de teste, que são dados de entrada, condições de execução e o resultado esperado da execução. Essa fase de planejamento também identifica o que deve ser testado e quando. A fase de análise compara o resultado obtido com o esperado. As fases de análise e execução podem ser realizadas em paralelo ou em seqüência.

A atividade de teste (especialmente o teste completo) é uma atividade muito difícil, que demanda tempo e nunca se garante o corretismo. Por isso, tem sido dada ênfase na garantia da qualidade durante o processo de desenvolvimento geral e não no final do processo ou no produto de trabalho finalizado (ADRION, BRANSTAD, CHERNIAVSKY, 1982).

Deve-se evitar testar os próprios programas desenvolvidos, pois a detecção dos erros pode indicar uma falha pessoal, o que acaba não sendo uma motivação para encontrar novas falhas. Ao contrário, deve-se testar programas desenvolvidos por outros, mesmo porque muitas vezes é mais fácil perceber erros a partir de um ponto de vista externo (CARRINGTON, 1997).

Os Testadores devem trabalhar em conjunto com desenvolvedores pois o objetivo final é o mesmo, ou seja, a entrega de um produto de *software* para o cliente com qualidade (WHITTAKER, 2000).

Myers (1978) realizou um experimento envolvendo três tipos de teste: teste funcional (caixa preta), teste estrutural (caixa branca) e método de *walkthrough*/inspeção. Nessa experiência, concluiu-se que houve considerável variabilidade em número e tipo de erros encontrados comparando-se o desempenho individual, independentemente dos anos de experiência dos profissionais, pois de acordo com a experiência, um documentador encontrou muito mais erros do que um especialista em teste de programa. Em virtude dessa variabilidade, é necessária a habilidade de seleção pessoal de perfis de profissionais para o papel de teste de programa, deve-se avaliar a competência do profissional em relação ao perfil adequado para a função de teste.

Questiona-se quando se deve parar a atividade de testes. Seguem algumas considerações possíveis (HEISER, 1997):

- o cronograma previsto e o capital disponível se encerram;
- não se pode elaborar ou planejar novos casos de teste;
- o retorno de investimento diminui;
- testes sucessivos não identificam novos defeitos;
- o nível de cobertura de testes desejado é alcançado;
- todos os defeitos são removidos;
- os testes se tornam intermináveis.

O teste de *software* deve ser avaliado também sob o ponto de vista econômico. Pode-se pensar que uma maior quantidade de testes encontrará um maior número de defeitos. Porém, é importante avaliar não somente o fato de encontrar todos os defeitos, mas também se o custo de encontrar defeitos remanescentes pode ser justificado. O que deve ser analisado é a probabilidade do teste encontrar defeitos adicionais, o custo de encontrá-los, a probabilidade dos usuários encontrarem esses defeitos e o impacto causado por eles (CHAAR, HALLIDAY e BHANDARI, 1993).

A combinação das diferentes técnicas de teste detalhadas a seguir deve ser aplicada para se obter uma maior qualidade do *software* testado. Deve ser analisado o problema em particular para a adequada aplicação das técnicas de teste (ADRION, BRANSTAD, CHERNIAVSKY, 1982).

Os tipos de testes descritos a seguir exigem código executável.

2.3.1 Teste de caixa preta *(Black-Box testing)* e Teste de caixa branca *(White-Box testing* ou *Glass-box testing)*

Os casos de teste são eventos, ações ou entradas para a execução dos testes, procedimentos a serem seguidos pelos Testadores, detalhamento dos resultados esperados, informação relevante aos Testadores de modo padronizado. É importante que os casos de teste estejam bem definidos e padronizados, por

exemplo, para facilitar a integração e a produtividade de um novo membro da equipe.

O desafio da criação dos casos de teste é ter um número suficiente de casos de teste para maximizar a detecção de falhas e ao mesmo tempo minimizar o desperdício de se ter vários casos de teste que identifiquem o mesmo tipo de falha.

As técnicas de teste de caixa preta e teste de caixa branca foram citados por Schach (1996), Laitenberger (1998), Andrews (2006), Pan (1999), Denger e Kolb (2006), Kit (1995), McGregor (2001), Parveen, Tilley e Gonzalez (2007), Carrington (1997), Ryser, Berner e Glinz (1998), Li (1990) e Chaar, Halliday e Bhandari (1993).

O teste de caixa preta também foi citado por Gittens et al. (2002), Dalal, Horgan e Kettenring (1993) e Osterweil (1996).

No teste de caixa preta (teste funcional ou teste de especificação, dirigido a dados), a informação utilizada para a criação dos casos de teste é a especificação e os requisitos funcionais do sistema.

Não é preciso conhecer detalhes de implementação (tratados como uma caixa preta), nem o comportamento ou a estrutura interna do programa e não há necessidade de conhecimento do código interno e da lógica interna do sistema. O Testador precisa conhecer apenas as entradas, saídas e a especificação. A ênfase ocorre na execução das funções, na avaliação do produto em relação ao usuário final.

O teste de caixa preta não objetiva testar todos os elementos da estrutura de um programa, mas, ao contrário, garantir que os requisitos exigidos do programa sejam atendidos.

Analisando as saídas a partir das entradas, podem-se identificar as funcionalidades do sistema. As saídas podem ser validadas de acordo com a especificação, a partir dos parâmetros de entrada.

Passos para o teste de caixa preta (HEISER, 1997):

- identificar as funções esperadas de execução do sistema;
- desenvolver dados de teste para verificar se as funções são executadas;

- determinar a resposta correta para os dados de teste.

Vantagens do teste de caixa preta (HEISER, 1997):

- os casos de teste continuam sendo úteis mesmo que a implementação sofra alterações;
- os casos de teste podem ser desenvolvidos paralelamente à implementação;
- os casos de teste podem ser utilizados para o teste de aceite do cliente.

Desvantagens do teste de caixa preta:

- vários casos de teste podem simular a mesma parte do código.

No teste de caixa branca (teste estrutural ou teste de código, dirigido à lógica), a informação utilizada para a criação dos casos de teste é o código, análise da estrutura do código-fonte, detalhes da implementação, lógica interna do sistema e linguagem. O Testador precisa conhecer detalhes da estrutura interna do programa. É necessário que se tenham habilidades em programação pois todos os caminhos percorridos pelo programa devem ser identificados, deve-se exercitar o código o máximo possível e deve-se avaliar os resultados esperados contra os resultados obtidos. Deve-se assegurar que a maior parte das declarações, comandos ou caminhos do código seja executada pelo menos uma vez.

O teste de caixa branca também não é indicado para testar todo o sistema e não identifica funcionalidades omitidas.

Passos para o teste de caixa branca (HEISER, 1997):

- determinar a estratégia e a abrangência de teste;
- construir casos de teste para implementar a estratégia.

Vantagens do teste de caixa branca (HEISER, 1997):

- a redundância dos casos de teste é minimizada.

Desvantagens do teste de caixa branca:

- os casos de teste precisam mudar caso a implementação se altere;
- o desenvolvimento dos casos de teste precisa aguardar o término da implementação.

Deve-se combinar o teste de caixa branca e o teste de caixa preta para uma melhor otimização do teste (HEISER, 1997).

Além da criação dos casos de teste, o processo de testes envolve também a criação dos planos de teste, da descrição dos resultados de teste, dos relatórios de erros encontrados. Os testes podem ser realizados em diferentes níveis como testes de unidade, de sistema, de integração, de regressão, entre outros (PARVEEN, TILLEY e GONZALEZ, 2007).

2.3.2 Teste de regressão *(Regression testing)*

Testes de regressão (atividade de reteste) foram citados por Gittens et al. (2002), Li e Wahl (1999), Dalal, Horgan e Kettenring (1993), McGregor (2001), Harrold (2000), Whittaker (2000), Carrington (1997), Adrion, Branstad e Cherniavsky (1982).

Teste de regressão é um tipo de teste aplicado sempre que ocorrerem alterações nas especificações do sistema ou no código. Pode ocorrer por mudanças na especificação (manutenção adaptativa) ou por correção de erros (manutenção corretiva). Esse tipo de teste pode ser aplicado nas fases de desenvolvimento ou de manutenção. Não se trata de um novo tipo de teste, mas trata-se da reexecução de uma parte ou de todos os testes definidos para uma atividade de teste específica; pode ser executado sobre cada atividade como teste de unidade, teste de sistema,

entre outros (KIT, 1995). Não se podem garantir idênticas condições para cada reexecução dos testes.

Mais de 50% dos custos do ciclo de desenvolvimento de *software* estão associados com a atividade de manutenção (ADRION, BRANSTAD e CHERNIAVSKY, 1982).

Normalmente, somente as partes do sistema envolvidas com modificações devem ser retestadas. Porém, mudanças em um certo nível do ciclo do sistema requerem que as atividades dos níveis inferiores também sejam reverificadas. Por exemplo, alterações no nível do projeto requerem que o projeto seja reverificado e também que sejam retestados o teste de unidade e o teste do sistema.

Mesmo após alterações no sistema, deve-se garantir que ele ainda atenda às especificações, que novos erros não sejam introduzidos nas partes não alteradas. Deve-se isolar a causa do erro para não introduzir novas falhas em partes que anteriormente funcionavam corretamente.

Pode-se reusar e selecionar os casos de teste ou novos casos de teste serão necessários para as partes alteradas do sistema. Deve-se avaliar o número de retestes necessários, selecionar e priorizar os casos de teste adequados para minimizar os custos, sem que a cobertura (abrangência) do teste seja prejudicada, identificar os casos de teste desnecessários que não estejam mais sendo utilizados após alterações no sistema (RYSER, BERNER e GLINZ, 1998).

Segundo Whittaker (2000), um reparo ou acerto pode consertar apenas o problema em questão, falhar na tentativa de correção do próprio problema, consertar o problema mas deixar que outro item que funcionava corretamente não funcione mais, falhar na tentativa de correção e inclusive comprometer outro item que funcionava bem anteriormente.

Existem dois tipos de estratégias de teste de regressão:

- estratégia de reteste total (*retest-all*): quando todos os casos de teste são reusados para o reteste, o que resulta no desperdício de tempo e de recursos para que todos os casos de teste sejam testados desnecessariamente;
- estratégia seletiva (*selective*): quando se seleciona casos de teste para o teste de regressão. A unidade de teste será reexecutada somente se as entidades de

programa (funções, variáveis) que ela abrange for alterada. Objetiva-se redução de tempo e de custo de reteste por selecionar somente os casos de teste que correspondem à parte modificada do código.

Ao selecionar os casos de teste, deve-se analisar o alinhamento do caso de teste com a parte modificada ou com o módulo do sistema que chama (executa) outra função que tenha sido modificada. Outra possibilidade é analisar métodos que utilizam atributos de dados modificados ou utilizados por código modificado. O desafio na seleção dos casos de teste é identificar e fazer a correspondência dos casos de teste com os módulos alterados em estudo.

Deve-se avaliar o custo-benefício das estratégias reteste total e seletiva, verificar se os custos e esforços necessários para selecionar os casos de teste na estratégia seletiva não superam o custo extra de se utilizar todos os casos de teste para o reteste.

Segundo Harrold (2000), a atividade de teste de regressão tem um custo elevado e é um dos mais altos das atividades do ciclo de vida de um sistema, estimado em um terço (1/3) do custo total. O custo do teste de regressão é alto por haver trabalho manual de seleção de casos de teste e análise dos resultados do teste. Devem-se dispender esforços na automatização e na redução dos custos de testes (OSTERWEIL, 1996).

Para o reteste de *software*, devem-se incluir:

- casos de teste que tenham causado falha no sistema anteriormente a fim de garantir que a falha tenha sido corrigida;
- casos de teste que abranjam a parte modificada do código do sistema para validar a alteração propriamente dita;
- casos de teste que abranjam módulos não modificados, mas, que chamam (executam) módulos modificados ou que utilizam atributos de dados que foram modificados.

É muito importante que a documentação dos testes (contendo os diversos casos de teste) seja bem organizada e bem feita pois no teste de regressão,

reutilizam-se os casos de teste. Quanto maior a qualidade dessa documentação, menores serão os custos e esforços envolvidos nesse tipo de teste (ADRION, BRANSTAD, CHERNIAVSKY, 1982).

2.3.3 Teste de unidade

O teste de unidade é o teste e validação de módulos individuais do sistema, processo iterativo que verifica a consistência lógica de código de baixo nível e pode ser executado a partir da finalização do código. Trata-se da fase de teste inicial executada pelo criador do código durante a fase de desenvolvimento pois é ele quem conhece detalhes internos do módulo.

Pode-se utilizar o teste funcional (a partir da especificação) ou teste estrutural (a partir do código) para garantir que o módulo não faça o que realmente não deva fazer (LI, 1990; LI e WAHL, 1999; DALAL, HORGAN e KETTENRING, 1993; MCGREGOR, 2001; RAMACHANDRAN, 1996; HOUSE e NEWMAN,1989; KIT, 1995).

O teste de unidade é o teste da menor unidade testável do produto de *software* que pode ser compilada independentemente, possibilitando o paralelismo simultâneo de múltiplos componentes ou módulos; identifica a conformidade com a especificação funcional, valida as *interfaces* (parâmetros) de entrada e saída que trocam informações com o programa, a ordem e o número correto de argumentos passados, definições e uso de estrutura de dados, tipos adequados de argumentos (BERTOLINO e MARCHETTI, 2005). O objetivo do teste de unidade é identificar discrepâncias entre a especificação da interface do módulo e o seu comportamento real, isolar partes do programa que possam ser validadas como unidades corretas, facilitando a identificação e correção de erros pois é menos complexo testar partes do que testar o sistema como um todo.

Exemplos de unidade: programa, função, procedimento, sub-rotina ou classe.

2.3.4 Teste de integração

Conforme citado por Dalal, Horgan e Kettenring (1993), McGregor (2001) e Kit (1995), é o teste em que módulos individuais são dispostos juntos, teste que avalia a

interação entre as unidades testadas, identifica falhas não detectadas pelo teste de unidade, verifica requisitos funcionais, requisitos de desempenho e de confiabilidade.

O teste de integração agrega componentes executáveis, verifica a interação entre componentes conforme especificação definida. As unidades podem estar corretas e validadas, porém, pode haver problemas de integração entre os componentes ou erros nas *interfaces* entre os componentes (BERTOLINO e MARCHETTI, 2005). Deve-se também analisar e verificar inconsistências entre as unidades de *software* e o *hardware*.

Os testes de integração são executados após os testes de unidade e antes dos testes de sistema.

Conforme Li e Wahl (1999), cerca de 40% dos erros de *software* se referem aos problemas de integração de módulos identificados durante o teste de integração.

2.3.5 Teste de sistema

Teste de sistema é o teste completo para verificar se todos os elementos do sistema estão integrados adequadamente, é o teste de todas as combinações do produto visto como unidade, se o sistema inteiro funciona conforme os requisitos especificados, de acordo com o ponto de vista do cliente, sem que se tenha conhecimento de detalhes da implementação e de estruturas internas do sistema. O teste de sistema é executado após a conclusão dos testes de unidade e de integração e é necessário apenas conhecer as funções do sistema e a interface entre os componentes do sistema (DALAL, HORGAN e KETTENRING, 1993; LI, 1990; LI e WAHL, 1999; GITTENS et al., 2002; McGREGOR, 2001; HOUSE e NEWMAN, 1989).

Os objetivos do teste de sistema são identificar falhas não detectadas em testes de unidade e de integração, avaliar a credibilidade do produto como fator de qualidade, dispor de informações suficientes para a decisão de liberar a entrega do produto ao cliente (BERTOLINO e MARCHETTI, 2005).

O teste de sistema deve ser executado por um grupo de teste independente, que não faça parte do grupo de desenvolvimento, antes que o produto seja entregue ao cliente (KIT, 1995).

A partir da criação do projeto, podem-se realizar testes de integração e de sistema.

Citam-se alguns tipos de teste de sistema (KIT, 1995):

- teste de volume: verificar se o programa tem a capacidade de manipular satisfatoriamente volumes desejados de dados, pedidos, entre outros;
- teste de carga/stress: identificar condições de pico em que o sistema falha no tempo de resposta de processamento exigido;
- teste de segurança: analisar se os requisitos de segurança do programa estão sendo atendidos;
- teste de usabilidade (fatores humanos): identificar operações que criem dificuldades para os usuários;
- teste de desempenho: verificar se o programa atende aos requisitos de desempenho definidos;
- teste de uso dos recursos: avaliar se o programa extrapola os limites de uso dos recursos como por exemplo, memória e espaço do disco;
- teste de configuração: verificar se o comportamento do programa é adequado quando os recursos de *hardware* e *software* estejam configurados de acordo com os requisitos exigidos;
- teste de instalação: identificar condições em que procedimentos de instalação podem resultar em erros;
- teste de recuperação: analisar se o programa se comporta de acordo com os requisitos de recuperação após ocorrerem falhas;
- teste de disponibilidade: verificar se o programa atende aos requisitos de disponibilidade definidos.

2.4 Comparação entre análise estática e teste dinâmico

Na análise estática, a representação do comportamento de execução do programa se baseia na estrutura e semântica do programa. Por outro lado, no teste dinâmico, a representação do comportamento real do programa se baseia nas suas linhas de execução (OSTERWEIL et al., 1996).

O modelo de execução na análise estática não é desenvolvido com grandes exigências de procedimentos ou de grandes perfis necessários para execução (não é de alto padrão). A desvantagem da análise estática é a imprecisão do modelo de execução, o que pode resultar em incorreta análise. Cita-se também como desvantagem a existência de um conjunto limitado de tipos diferentes de falhas.

Em contrapartida, o modelo de execução no teste dinâmico é preciso e correto.

Levando-se em consideração as características de cada modelo de execução, deve-se combinar os 2 tipos de modelos, a precisão do modelo dinâmico com a generalidade da análise estática, junto com a verificação formal.

A vantagem da análise estática é a análise, por exemplo, do código, independente das condições do caminho de sua execução. Sua desvantagem é não poder medir o desempenho dessa execução e o fato da qualidade dos resultados depender da habilidade dos Revisores e dos Inspetores (HEISER, 1997).

Por outro lado, o teste dinâmico tem a capacidade de avaliar o desempenho da execução do *software.* Porém, sua desvantagem é a dificuldade na construção de casos de teste que abrangerão a cobertura desejada (HEISER, 1997).

Inspeção e teste são duas formas de detecção e correção de defeitos; entretanto, a inspeção demanda menos tempo e esforço em relação ao teste (ACKERMAN, BUCHWALD e LEWSKI, 1989). O objetivo da inspeção e do teste é aumentar a qualidade do produto e reduzir os custos envolvidos com as falhas existentes nas diferentes fases do ciclo de vida do sistema (CHAAR, HALLIDAY e BHANDARI, 1993).

Comparando-se inspeção e teste, a inspeção possibilita a detecção de falhas e a imediata localização do defeito no artefato. Por outro lado, o teste permite identificar a existência de defeitos por meio de falhas ocorridas durante a execução

do código. Porém, no teste, a localização do defeito no código e da causa da falha já é um processo mais demorado e demanda mais esforço (LAITENBERGER, 1998).

Embora ambas sejam técnicas de detecção de falhas, existem diferenças entre o teste e a inspeção. Enquanto o Testador identifica problemas ou comportamento inadequado e precisa isolá-los para identificar as causas e solucioná-los, o revisor (na inspeção) identifica as falhas no documento e não há necessidade do seu isolamento pois a causa (raiz) do problema já é identificada previamente (ANDERSSON et al., 2003).

No teste, as falhas são corrigidas uma a uma, enquanto que na inspeção, todos os defeitos identificados e discutidos em reunião podem ser corrigidos de uma única vez.

Sugere-se realizar a inspeção antes do teste por duas razões (RUSSELL, 1991 apud WELLER, 1993):

- a inspeção se baseia numa equipe motivada na detecção de erros. Já o teste possui a ilusão de que o produto funciona adequadamente;
- quando o teste é extenso e o produtor do código consome muito tempo na criação do produto, ele pode resistir a otimizações que demandam muito retrabalho.

Desvantagens de se executar a inspeção após o teste (WELLER, 1993):

- a execução do teste de unidade reduz a motivação da equipe de inspeção na localização de defeitos. Se já foram executados os testes e se conclui que o produto funciona, por que inspecionar? Trata-se de uma falsa confiança de que o produto funciona e não uma garantia;
- a inspeção antes do teste inspeciona pequenas partes do produto enquanto todos os segmentos de código devem ser testados juntos no teste de unidade. Se a inspeção é feita após o teste, pode-se questionar que não se pode inspecionar todo o código simultaneamente;
- a inspeção de código pode identificar defeitos não detectados na inspeção do projeto. Causaria grande retrabalho se a inspeção fosse feita após o teste de

unidade, desperdiçando tempo e recursos por detectar e corrigir erros tardiamente.

A inspeção não substitui o teste, mas reduz o esforço dispendido com o teste já que a inspeção é de 2 a 10 vezes mais eficiente que o teste na remoção de defeitos. Além disso, a inspeção detecta e remove falhas com menores custos e reduz o tempo de desenvolvimento (ACKERMAN, BUCHWALD e LEWSKI, 1989).

A inspeção deve ser utilizada em conjunto com o teste de *software* para se obterem os benefícios das duas técnicas de forma maximizada, uma complementando a outra. A inspeção e o teste são conjuntamente necessários para a garantia da qualidade do produto (DUNSMORE, ROPER e WOOD, 2003; BERLING e THELIN, 2003).

2.5 Experimentos de inspeção x teste

Segundo experimentos de Laitenberger (1998), Andrews et al. (2006), Denger e Kolb (2006) e Laitenberger et al. (2002), a inspeção foi superior ao teste na detecção de defeitos.

Experimentos de Laitenberger (1998) realizados com estudantes de graduação de Ciência da Computação demonstraram que, em relação à efetividade de detecção de defeitos, a inspeção de código foi superior em relação ao teste estrutural. Tanto na análise de resultado do experimento individual, quanto na análise de resultados da equipe, a eficiência da inspeção foi superior ao teste estrutural.

Tem-se pouca ocorrência de que o teste estrutural possa detectar erros de diferentes tipos ou não identificados pela inspeção e vice-versa. Considerando a idéia de que a inspeção é mais eficiente do que o teste na detecção de defeitos e que ambos não enfocam a detecção de defeitos de diferentes classes, conclui-se com o experimento que a inspeção e o teste estrutural não se complementam bem quando aplicados em seqüência, ou seja, defeitos não detectados na inspeção não foram detectados ao se aplicar o teste estrutural.

Conforme Laitenberger et al. (2002), a inspeção de código foi mais efetiva que o teste funcional e o teste estrutural na detecção de defeitos, proporcionou maior qualidade de código e identificou diferentes tipos de defeitos em relação ao teste. O projeto em estudo na área de inspeção de *software* foi o projeto Impact que buscou analisar as causas do sucesso e lições aprendidas para serem consultadas como referência por outras organizações.

Já Denger e Kolb (2006) compararam inspeção de código com teste funcional, enfatizando o potencial das técnicas na fase do código. A inspeção e o teste funcional mostraram ser eficientes e eficazes na detecção e remoção de defeitos e puderam detectar tipos diferentes de defeitos, se aplicados sobre artefatos idênticos.

Nesse estudo, compararam-se as técnicas de detecção de defeitos “inspeção de código” e “teste funcional” com ênfase no desenvolvimento de linha de produto (ver detalhes abaixo). O objetivo principal do estudo foi avaliar o potencial e a eficácia das técnicas de detecção de defeitos sobre componentes de *software* reusáveis, incluindo características comuns e variantes (específico de produto).

Linha de produto de *software* é baseada no reuso planejado e sistemático de esforços de desenvolvimento de produtos funcionais similares. Os benefícios ganhos com o reuso incluem não somente a redução de tempo e dos custos de desenvolvimento e manutenção, mas também agilidade na entrega de novas características e produtos no mercado.

A linha de produto facilita o reuso de projetos, de implementações, de testes e de outros artefatos de desenvolvimento em se tratando de produtos similares de um mesmo domínio.

A inspeção foi mais eficaz do que o teste em cerca de 66,39% e houve menor esforço para a detecção de defeitos em 36,84%.

Tanto a inspeção quanto o teste se mostraram ineficazes na detecção de defeitos variantes específicos.

A importância de se ter alta qualidade de componentes de *software* reusáveis se deve ao fato de se reusar não somente a funcionalidade do componente, mas também as falhas contidas no mesmo componente.

No artigo de Andrews et al. (2006), comparou-se a inspeção de projeto com o teste funcional. A inspeção foi significativamente mais eficiente e eficaz do que o teste. A análise não levou em consideração os custos do retrabalho e nesse caso, a eficiência da inspeção sobre o teste seria ainda maior. Isso porque o custo de correção de defeito identificado durante a inspeção de projeto é muito menor do que a correção do defeito identificado no teste funcional, já que nesse caso há o retrabalho das fases do projeto e do código.

2.6 Formas do processo de teste – Verificação e Validação

As definições de teste não se referem apenas a teste de código, mas abrangem também testes de documentos e de outros modos não executáveis de um produto.

As atividades de verificação e validação têm o propósito de medir e assegurar a qualidade do produto ou do sistema (ao longo de seu ciclo de vida) tais como: funcionalidade, confiabilidade e interface amigável ao usuário. Elas também objetivam que somente as funções esperadas do sistema sejam adequadamente executadas (WALLACE e FUJII, 1989). A verificação é a atividade de garantir que o sistema como um todo esteja de acordo com os requisitos especificados. Já a validação é a atividade de garantir que o sistema se comporte de acordo com as necessidades do usuário (BERLING e THELIN, 2003).

Conforme definido pelo IEEE/Ansi, **verificação** é o processo de avaliar um sistema ou componente para definir se os produtos gerados em uma dada fase de desenvolvimento estão de acordo com as condições impostas no início daquela fase. Devem-se avaliar a consistência, a completude e o corretismo do sistema em cada fase e entre cada fase do ciclo de vida de desenvolvimento (ADRION, BRANSTAD, CHERNIAVSKY, 1982). Trata-se também do processo de avaliar, revisar, inspecionar os produtos de trabalho tais como as especificações de requisitos, as especificações de projeto e código.

Cada atividade de verificação é uma fase do ciclo de vida de teste. O objetivo dessa atividade é detectar o maior número de erros possível (KIT,1995). Se o objetivo é obter menores custos e maior qualidade de produto, a atividade de

verificação não deve se restringir a uma fase, mas sim, a todas as fases do ciclo de desenvolvimento.

O sucesso da atividade de verificação ao longo do ciclo de desenvolvimento depende da existência de produtos de trabalho de cada fase do ciclo bem definidos e claros (ADRION, BRANSTAD, CHERNIAVSKY, 1982).

Há quem denomine verificação como sendo análise ou testes humanos porque normalmente envolve análise de documentos em papel.

Na atividade de verificação, devem-se analisar as áreas específicas do produto que serão verificadas e as áreas que não serão verificadas. Nesse caso, deve-se avaliar o risco das áreas que não serão verificadas.

O objetivo principal de se investir na atividade de verificação o mais breve possível é poder identificar erros potenciais custosos de um modo pró-ativo antes que os custos de correção aumentem geometricamente (ADRION, BRANSTAD, CHERNIAVSKY, 1982).

A verificação deve ser feita tanto quanto possível pois tem-se demonstrado ser uma das melhores formas em custo-benefício para a melhoria da qualidade em curto e longo prazo. Além disso, é uma forma de melhorar a comunicação e a maturidade do ambiente de desenvolvimento.

Conforme definido pelo IEEE/Ansi, **validação** é o processo de avaliar um sistema ou componente durante ou no final do processo de desenvolvimento para determinar se ele atende aos requisitos especificados conforme previsto, avaliar a conformidade do projeto e da implementação de um produto de *software* contra critérios pré-definidos. Objetiva-se a identificação dos erros antecipadamente para reduzir os custos envolvidos com a sua remoção (CHAAR, HALLIDAY e BHANDARI, 1993). Validação é um processo de teste baseado em computador, envolvendo a execução do *software* real (KIT, 1995)**.**

Alguns fatores que podem influenciar nas necessidades de validação: tamanho e orçamento do projeto, criticidade, custo ou conseqüências dos problemas encontrados.

Sistemas críticos que podem resultar em severas conseqüências como por exemplo, controle de aeronaves ou transações financeiras bancárias, justificam

investimento e esforço nas atividades de validação (ADRION, BRANSTAD, CHERNIAVSKY, 1982).

Citam-se vários benefícios das atividades de verificação e validação (WALLACE e FUJII, 1989):

- avaliam o produto contra os requisitos do sistema, tornando-o mais estável;
- contribuem para um melhor planejamento do desenvolvimento do sistema e melhor cumprimento dos prazos;
- identificam previamente erros de alto risco, induzindo a equipe a ter mais a compreensão da solução do que uma correção rápida somente para satisfazer a pressão do cumprimento dos prazos de desenvolvimento;
- proporcionam acompanhamento gerencial da qualidade e do progresso do esforço de desenvolvimento e dessa forma, possibilitam uma melhor tomada de decisões;
- permitem que o usuário tenha uma visão incremental do sistema e possa solicitar ajustes parciais o mais breve possível.

Por outro lado, citam-se também algumas de suas desvantagens (WALLACE e FUJII, 1989):

- aumentam de 10 a 30% o custo de desenvolvimento. Porém, os custos podem ser melhor recuperados quanto mais cedo as atividades de verificação e validação se iniciarem;
- exigem interações adicionais entre a equipe do projeto;
- podem reduzir a produtividade do desenvolvimento.

Teste é a combinação das atividades de verificação e validação, que se complementam. Ambas são incumbidas de encontrar diferentes tipos de problemas no produto, devem ser feitos no tempo certo e sobre os produtos de trabalho adequados.

A verificação geralmente é mais efetiva que o teste de validação pois pode encontrar erros que seria impossível encontrar durante a validação. O mais importante é que a verificação permite encontrar erros o mais breve possível (KIT,1995).

As principais subatividades da verificação e da validação são a inspeção e o teste. A inspeção busca a identificação de defeitos no início do ciclo de desenvolvimento de *software*, antes da implementação do produto. Já o teste identifica os defeitos no final do ciclo, após a implementação (BERLING e THELIN, 2003).

2.7 Modelo U e Modelo V

Embora tenha uma base comum, o modelo de ciclo de vida de *software* varia para cada tipo de organização ou depende das circunstâncias.

O modelo V enfatiza a atividade de teste não como uma simples fase, mas como uma parte integrante do ciclo de desenvolvimento de *software* (PFLEEGER, 2001 apud CARVALHO**,** 2004). O objetivo é a realização do teste de cada fase do ciclo de vida do *software* para que se possa detectar os defeitos antecipadamente. Para cada fase do ciclo, diferentes atividades de teste são necessárias. Assim, o plano de teste pode ser elaborado à medida em que os testes de cada fase são realizados e os casos de teste podem ser gerados durante as diferentes fases do desenvolvimento (CARVALHO**,** 2004).

O modelo V do processo de desenvolvimento de *software* é uma extensão do modelo *Waterfall* (cascata) de desenvolvimento. Para cada fase do desenvolvimento, há uma atividade de verificação ou validação associada (SOMMERVILLE, 1996 apud ALMEIDA, 2001). Esse modelo possibilita um retorno de identificação de defeitos para cada fase do processo de desenvolvimento (GILB e GRAHAM, 1993).

O modelo U representa a integração entre o ciclo de desenvolvimento e o ciclo de teste. Para cada fase do desenvolvimento, há uma fase de teste correspondente (KIT, 1995). Cada produto da fase de desenvolvimento entregue é testado (verificado ou validado) pela equipe de testes.

O modelo U é similar ao modelo V, porém inclui a simulação do produto após a fase de especificação de projeto e após a fase de código.

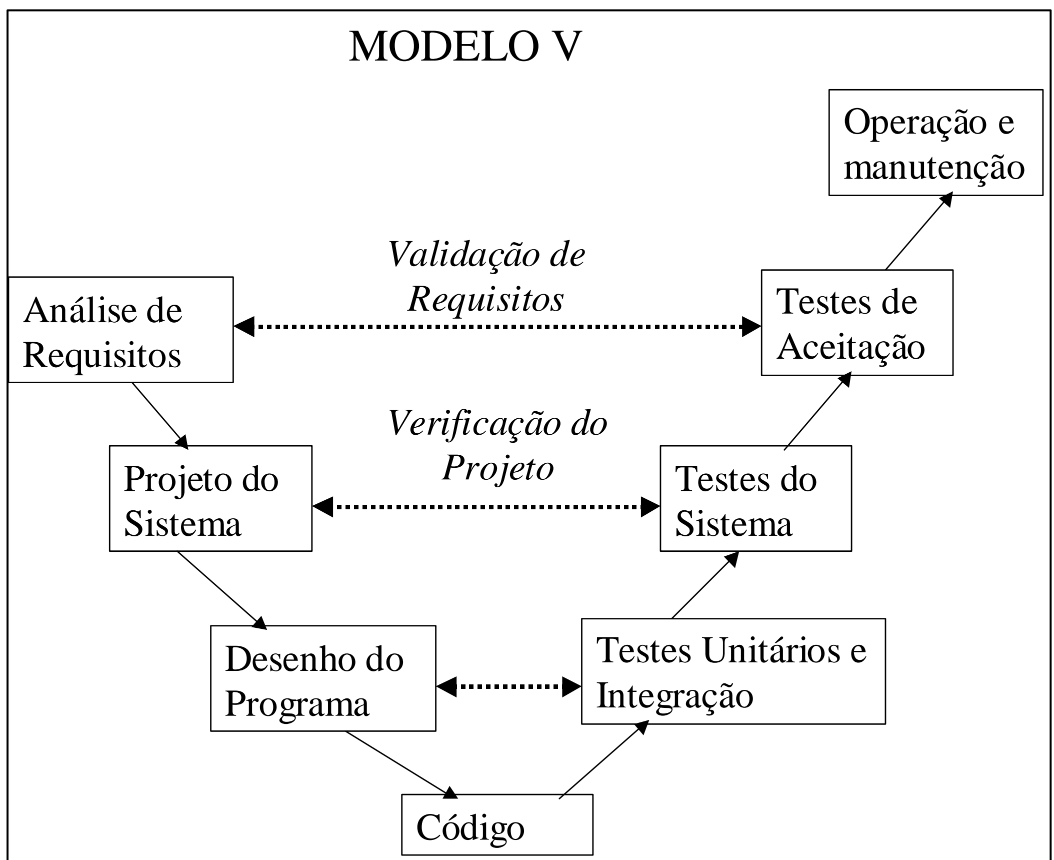


Figura 2 – O Modelo V

Fonte: PFLEEGER, 2004 apud CARVALHO, 2004

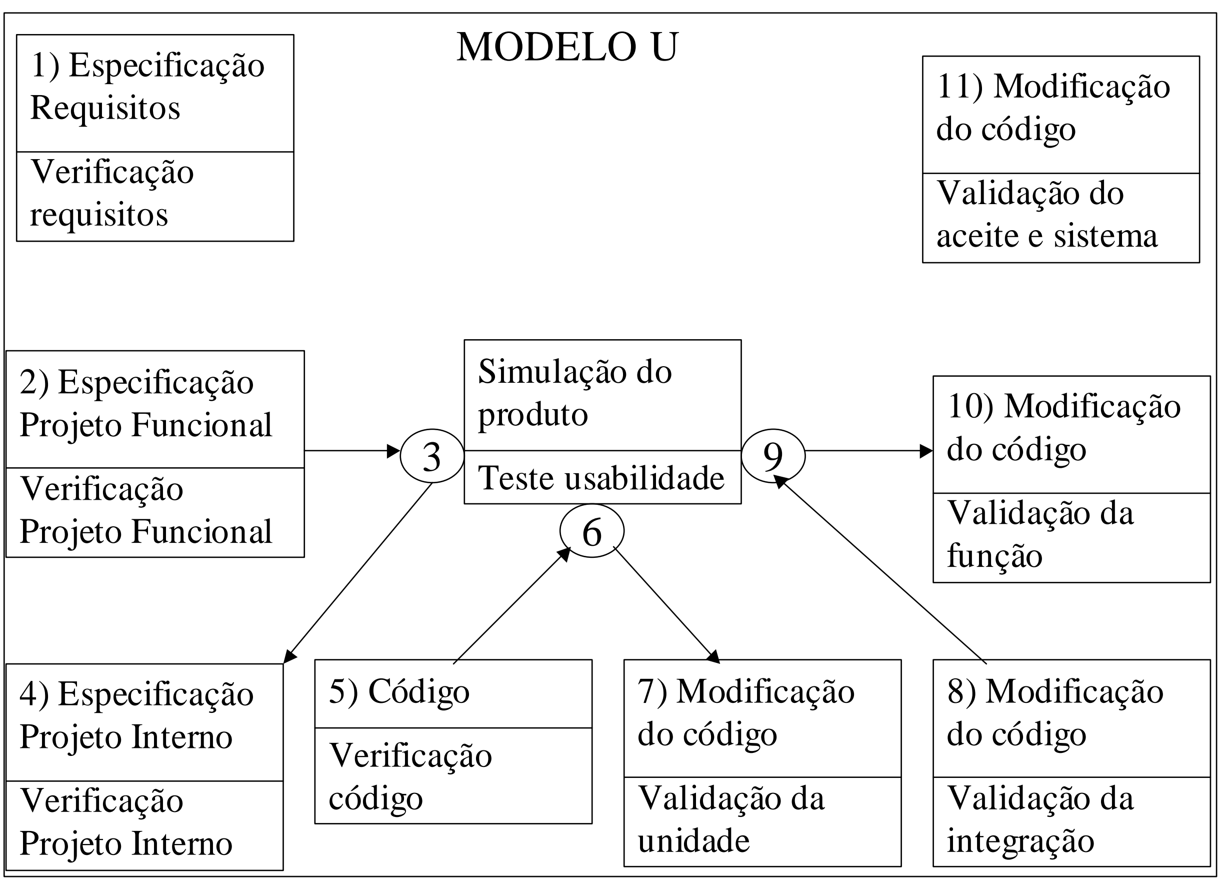


Figura 3 – O Modelo U

Fonte: KIT, 1995

2.8 Análise Orientada a objetos (OOA)

Como objetivo da OOA de definir as classes relevantes ao problema em estudo, devem-se realizar as seguintes atividades (PRESSMAN, 2002):

1) definição dos requisitos do usuário pelo cliente ao engenheiro de *software*;
2) identificação dos cenários ou casos de uso;
3) identificação e definição das classes e objetos de acordo com a definição dos requisitos;
4) identificação dos atributos e das operações para cada objeto do sistema;
5) especificação da estrutura e hierarquia de classes;
6) representação das relações (conexões) entre objetos;
7) modelagem do comportamento do objeto;

8) revisão do modelo de análise (OOA) de acordo com os casos de uso (cenários) definidos.

2.9 Projeto orientado a objetos (POO)

Passos para realizar um projeto orientado a objetos (PRESSMAN, 2002):

1) descrever cada subsistema e alocá-lo a processadores e tarefas;
2) escolher uma estratégia de projeto para implementar gestão de dados, suporte de interface e gestão de tarefas;
3) desenvolver um projeto para a interface do usuário;
4) projetar um mecanismo de controle adequado para o sistema;
5) desenvolver o projeto de objetos detalhando as operações e os atributos de classe;
6) desenvolver projeto de mensagens incluindo as relações entre os objetos;
7) criar um modelo de mensagens;
8) considerar as condições de limite e as providências a serem tomadas;
9) revisar o modelo de projeto.

Questões de Projeto

Cinco critérios para se julgar a capacidade que um método de projeto consiga modularidade, relacionando com projeto orientado a objeto (MEYER, 1998 apud PRESSMAN, 2002):

- capacidade de decomposição: facilidade com que um problema complexo possa ser subdividido em problemas menores com soluções mais simples;
- capacidade de composição: capacidade em que módulos possam ser reusados para criar outros sistemas;
- compreensibilidade: facilidade com que um componente possa ser entendido;

- continuidade: capacidade de fazer pequenas mudanças num programa e que resultem em alterações em apenas um ou em poucos módulos;
- proteção: redução da propagação de efeitos colaterais caso ocorra erro num módulo.

Características para se categorizarem objetos e classes (PRESSMAN, 2002)**:**

- tangibilidade: a classe representa algo tangível (exemplo: teclado ou sensor) ou informação abstrata (exemplo: uma saída prevista);
- inclusividade: a classe é atômica (não inclui outras classes) ou é agregada (inclui pelo menos um objeto aninhado);
- seqüencialidade: a classe é concorrente (tem sua própria linha de controle) ou seqüencial (é controlada por recursos externos);
- persistência: a classe é transiente (criada e removida durante a operação do programa), temporária (criada durante a operação do programa e removida quando o programa termina) ou permanente (é armazenada em um banco de dados);
- integridade: a classe é corruptível (não protege contra o acesso externo dos recursos) ou protegida (possui mecanismo de proteção dos recursos por controle de acesso).

2.10 Conclusão

O capítulo 2 buscou levantar e reunir trabalhos e soluções propostas para o mesmo tema ou assunto sendo desenvolvido na dissertação.

Devido à solução apresentada pelos trabalhos não ser muito detalhada, pretende-se com este trabalho desenvolver uma estratégia de inspeção com maior detalhamento.

As técnicas de inspeção e de teste apresentadas acima podem ser usadas concomitantemente, pois o uso de uma técnica não impede nem anula a utilização de outra técnica.

Capítulo 3

3 Ambientes colaborativos

3.1 Introdução ao *Wiki*

Os *Wikis* são mídias hipertextuais, com estrutura de navegação não-linear. Cada página geralmente contém um grande número de ligações para outras páginas. As ligações são criadas usando-se uma sintaxe específica, o chamado "padrão *link*".

A origem da palavra *Wiki* é havaiana, que significa “rápido”.

O modelo *Wiki* é uma rede de páginas da *Internet* contendo diversas informações, que podem ser modificadas e ampliadas por qualquer pessoa através de navegadores comuns, tais como o *Internet* Explorer, Mozilla Firefox, Netscape, Opera ou outro programa capaz de ler páginas em HTML (HyperText Markup Language) e imagens.

Uma das características da tecnologia *Wiki* é a facilidade com que as páginas são criadas e alteradas. Os acompanhamentos dos debates, dos projetos, dos artigos ocorrem em um único espaço ou documento, em tempo real, impossibilitando a perda de dados. A maioria dos *Wikis* é aberta a todo o público ou pelo menos a todos que têm acesso ao seu servidor, pois o registro de usuários não é obrigatório.

Dentro de um universo *Wiki*, não existem dois artigos com títulos repetidos, pois faz parte de sua filosofia utilizar-se da tecnologia de armazenamento para ajudar a eliminar ambigüidades. Além disso, o *Wiki* tem a sensibilidade de distingüir maiúsculas de minúsculas como letras distintas para o armazenamento.

Seguem abaixo algumas características de um *Wiki*:

- meio de comunicação útil de modo assíncrono via *web*, no âmbito da *intranet* ou de páginas *Internet* públicas;
- facilidade de edição: *Wikis* são editados usando uma linguagem de texto própria, que é em geral mais fácil do que o HTML ou o XHTML (eXtensible Hypertext

Markup Language), possibilitando o uso também por quem não detém grande conhecimento técnico. Digita-se o texto desejado livremente, usando marcações bem simples, como será visto adiante;

- colaboração: diferentemente de páginas normalmente acessadas da *Internet*, o *Wiki* permite a edição de suas páginas por usuários que navegam por elas. Assim, podem-se corrigir erros, complementar idéias, adicionar novas informações. Em função da coletividade, o conteúdo de um artigo se mantém atualizado;
- revisão e reversão: freqüentemente, autores que trabalham colaborativamente nas páginas *Wiki* podem revisar mudanças realizadas por outros autores e podem discordar delas. Assim, podem-se reverter as mudanças para versões anteriores se necessário, ou seja, existe o controle de versão automático;
- sistemas de navegação: uma navegação hierárquica tradicional não é adequada para *Wikis* porque geralmente múltiplos autores criam e excluem páginas. O *Wiki* disponibiliza ponteiros internos, pesquisas e *tags* (etiquetas) para facilitar a navegação;
- permissões: administradores podem atribuir diferentes níveis de permissão para visitantes visualizarem, editarem, criarem ou excluírem páginas e informações. As permissões previnem o uso inadequado da aplicação;

  Alguns *Wikis* permitem que o banco de dados *on-line* compartilhado seja configurado para o modo apenas de leitura, enquanto outros adotam uma política em que apenas usuários registrados possam editar. Normalmente, prejuízos causados por mal-intencionados podem ser revertidos facilmente.

- páginas de discussão: autores usualmente utilizam páginas de discussão para obter consenso sobre o conteúdo das páginas. A maioria dos *Wikis* possui espaço de discussão associado a cada página *Wiki*;
- um ponto negativo que pode ser destacado é que no *Wiki*, por ser um ambiente que permite colaboração por diferentes usuários, as informações que são registradas nem sempre são feitas por especialistas do assunto, podendo ocorrer vandalismo e substituição do conteúdo de um artigo. Mas ao mesmo tempo, as correções de vandalismo também podem ocorrer de forma rápida. De

um modo geral, espera-se que as informações sejam acrescentadas por quem detém conhecimento do assunto abordado.

O modelo *Wiki* tem sido usado não somente no âmbito editorial, mas também em várias empresas e instituições de ensino, por assegurar agilidade e uniformidade em projetos que demandam participações de várias pessoas situadas em localidades diferentes.

Cita-se o Wikipédia como exemplo de um sistema que se baseia nesse modelo. Segundo a sua própria definição, é uma enciclopédia multilíngüe *on-line*, livre e colaborativa, ou seja, escrita internacionalmente por várias pessoas comuns de diversas regiões do mundo, todas elas voluntárias. Por ser livre, entende-se que qualquer artigo dessa obra pode ser transcrito, modificado e ampliado, desde que preservados os direitos de cópia e de modificações, visto que o conteúdo da Wikipédia está sob a licença GNU/FDL (Free Documentation License) ou GFDL (Licença GNU de Documentação Livre).

O Projeto GNU foi um projeto iniciado por Richard Stallman em 1984, com o objetivo de criar um sistema operacional totalmente livre, que qualquer pessoa teria direito de usar, modificar e redistribuir o programa e seu código fonte, desde que garantido para todos os mesmos direitos. Esse sistema operacional GNU deveria ser compatível com o sistema operacional UNIX, porém não deveria utilizar-se do código fonte do UNIX. Stallman escolheu o nome GNU porque esse nome, além do significado original do mamífero Gnu, é um acrônimo recursivo de: GNU is Not Unix (GNU não é Unix).

GNU/FDL é uma licença para documentos e textos livres publicada pela Free *Software* Foundation. É inspirada na GNU GPL (General Public License), da mesma entidade, que é uma licença livre para *software.* A GNU FDL permite que textos, apresentações e conteúdo de páginas na *web* sejam distribuídos e reaproveitados, mantendo, porém, alguns direitos autorais e sem permitir que essa informação seja usada de maneira indevida. A licença não permite, por exemplo, que o texto seja apossado por outra pessoa, ou que sejam impostas sobre ele restrições que impeçam que seja distribuído da mesma maneira em que foi adquirido.

A enciclopédia, sem fins lucrativos, está disponível em 257 idiomas. Estão incluídas imagens, páginas de usuários, páginas de discussão, categorias, predefinições, páginas de gestão dos projetos. Desde seu início, a Wikipédia tem aumentado sua popularidade e seu sucesso tem feito surgir outros projetos similares.

3.1.1 MediaWiki

O MediaWiki, que pode ser encontrado no endereço eletrônico <http://www.mediawiki.org/wiki/MediaWiki>, também é um *Wiki* e é indicado para usuários corporativos e público em geral. A linguagem de programação utilizada é o PHP e o Ocaml e o banco de dados é o MySQL. Há prevenção de *spam*, permissões, controle de acesso e uso de HTML. A licença de *software* é a GPL.

Citam-se alguns de seus recursos:

- conflitos de edição: permite que usuários simultâneos editem e salvem as alterações, tentando aglutinar as modificações automaticamente se possível, ou solicita a sua junção de forma manual;

- assinatura automática, formatação de textos, suporte multilíngüe, páginas de discussão, depósito de arquivos, pesquisas de texto;

- notificação de mensagem: recebimento de mensagens de notificação, por exemplo, ao ser alterada uma página de discussão de usuários.

3.1.2 *Twiki*

Para este trabalho, foi escolhida uma implementação especial de ambientes *Wiki* chamada *Twiki*, que está descrita a seguir.

O *Twiki* é mais leve, simples de se aprender, de se utilizar e exige menos recursos computacionais. O seu desenvolvimento é de código aberto e conta com a contribuição de colaboradores de muitos países. Normalmente ele é usado em ambientes comerciais, em *intranets* corporativas. Dentre as empresas que utilizam o *Twiki*, podem-se citar a Disney, British Telecom, SAP e Motorola.

Pode-se trabalhar com interesses comuns e pode-se colaborar utilizando um navegador *web.* O *Twiki* é similar a uma página *web* normal, com a diferença de que encoraja a contribuição e edição de páginas, questionamentos, respostas, comentários e atualizações.

O ambiente é bastante interativo, dinâmico, estruturado e é composto de um conjunto de *webs* que podem ser criadas da maneira desejada. Cada *web* representa um ambiente de colaboração. Dentro de cada *web* podem ser criados diversos tópicos que podem ser acessados dinamicamente por ponteiros.

Por ser um *Wiki,* o *Twiki* permite as seguintes operações:

- por meio do navegador, adicionar ou alterar qualquer informação nos tópicos;
- publicar, compartilhar e trocar informações e idéias com outros de um modo transparente via *web*;
- fazer ligações entre vários tópicos;
- criar listas numeradas, tabelas, formulários, modelos de documentos;
- depositar arquivos;
- formatar textos como por exemplo, para itálico e negrito;
- realizar alterações de um modo experimental, não definitivo, usando uma *web* especial (Sandbox);
- organizar tópicos, por exemplo, classificar páginas por assunto ou data;
- visualizar o histórico das alterações dos tópicos, quem alterou, o que e quando; visualizar as versões anteriores dos tópicos e diferenças entre duas versões;
- desfazer alterações;
- receber mensagens de notificação por correio eletrônico sempre que ocorrerem alterações nas páginas *Twiki*, conforme configurado no sistema.

***Plugin* (componente modular):**

O *Twiki* permite extensões chamadas de *plugins*, criadas pelo usuário. Algumas dessas extensões já estão disponíveis na distribuição padrão desse sistema.

Segue abaixo uma lista de *plugins* instalados e habilitados no *Twiki*:

- SpreadSheetPlugin: adiciona cálculos de planilhas como por exemplo, somatória, a tabelas *Twiki* e textos;
- CommentPlugin: permite aos usuários rapidamente postar comentários a uma página sem ter que editá-la e salvá-la;
- EditTablePlugin: permite a edição de campos de tabelas *Twiki*;
- InterwikiPlugin : permite criar ponteiros para páginas externas usando apelidos (*aliases*) definidos em um tópico especial;
- PreferencesPlugin: permite a edição de preferências usando campos predefinidos em um formulário;
- SlideShowPlugin: cria apresentações *web* baseadas nos tópicos;
- SmiliesPlugin: disponibiliza ícones;
- TablePlugin: controla atributos de tabelas e ordena colunas;
- TwistyPlugin: biblioteca JavaScript para manipular conteúdo dinamicamente;
- WysiwygPlugin: editor visual para as páginas.

Sandbox *Web:*

A *web* Sandbox é utilizada para testes, onde podem-se realizar alterações em caráter experimental. Qualquer pessoa pode criar e alterar os tópicos e seu conteúdo não afeta o resto do sistema.

Para cada *web* ou para cada novo tópico criado, têm-se as seguintes opções em seu rodapé:

- edit: edita um tópico ou *web* existentes;

- raw view: exibe o código-fonte sem permitir a edição do tópico;
- attach: adiciona arquivos ao tópico corrente;
- backlinks: pesquisa por tópicos que fazem referência ao tópico corrente;
- printable: exibe a página apropriada para impressão;
- history: importante para o controle de revisão, exibe o histórico completo das alterações do tópico, como por exemplo, quem alterou, o que e quando;
- r3>r2>r1: exibe as versões anteriores do tópico ou a diferença entre duas versões;
- “more topic actions”: permite acesso a um conjunto de funções, como por exemplo, renomear, mover, remover, definir um novo pai para o tópico, comparar revisões, entre outros.

Formatação:

Existem diferentes modos de se formatarem textos:

- negrito: deve-se colocar o texto entre asteriscos. Ex.: *sua frase* ;
- lista em tópicos: usa-se 3 espaços, 1 asterisco e 1 espaço. Ex.:   * seu texto em tópicos;

  A profundidade da lista é indicada pelo número de espaços, em múltiplos de 3.

- cabeçalho ou título: 3 hífens, de 1 a 6 sinais de mais e 1 espaço. Ex.: ---++ Seu cabeçalho ;
- itálico: coloca-se o texto entre sublinhado. Ex.: _suas palavras_ ;
- desabilitar formatação: use <nop> para desabilitar formatação não desejada. Ex.: <nop> _palavra_ (não irá formatar em itálico) ;
- parágrafos: apenas separe com uma linha em branco.

WikiWord:

WikiWord são nomes de tópicos com uma formatação especial.

Cada tópico criado pode ser acessado automaticamente por meio de um ponteiro, desde que o nome do tópico obedeça a alguns critérios tais como: ser iniciado por letra maiúscula, seguido de números ou uma ou mais letras minúsculas e seguido novamente por uma letra maiúscula, sem espaço. Exemplo: SistemaOrientadoObjeto.

A vantagem de se terem WikiWords é não ser necessário se conhecer o caminho completo onde o tópico está armazenado para acessá-lo (apenas se digita seu nome) e a não necessidade de se escreverem comandos HTML.

Referência a outras páginas ou a outros tópicos:

Conforme descrito anteriormente, cada Wikiword automaticamente se torna um ponteiro. Entretanto, muitas vezes se deseja adicionar uma descrição ao ponteiro ao tópico ou à página *web* e não simplesmente exibi-lo. Para tanto, utilizam-se colchetes para separar o tópico ou a página da descrição a ser exibida, conforme exemplo abaixo:

```
[[SistemaOrientadoObjeto][Voltar à Página Principal]]
```

Variáveis e comandos especiais:

No *Twiki*, todas as variáveis são delimitadas por símbolos '%', o mesmo acontece em comandos especiais.

Exemplos: `%HOMETOPIC%` (tópico inicial de uma *web*), `%MAINWEB%` (nome da *web* principal), `%DISPLAYTIME%` (exibe a hora atual).

Para os comandos, os parâmetros ou cláusulas são indicados entre '{' e '}' da seguinte forma:

```
%comando{param1=valor1 param2=valor2 ....}%
```

A seguir mostramos os principais recursos que foram utilizados na implementação da proposta.

Tabela:

De um modo fácil, podem-se construir tabelas usando barras verticais:

```
| Gato   | Feline  |
| Urso   | Ursine  |
| Raposa | Vulpine |
```

o que é exibido como:

Gato	Feline
Urso	Ursine
Raposa	Vulpine

Edittable:

O *Plugin* Edittable permite criar tabelas que podem ser editadas e salvas para consultas futuras.

```
%EDITTABLE{header="| *Equipe do Projeto* | *Preencha
adequadamente* |" format="| text, 20,  | text, 50,  |"
changerows="on"}%
|*Equipe do Projeto*|*Preencha adequadamente*|
| Gerente do Projeto |  |
| Analista de Requisitos |  |
| Administrador de Banco de Dados |  |
| Programador |  |
| Analista de Testes |  |
```

O Edittable precedido do símbolo % exibe um botão “Edit table”. Toda vez que se almeje incluir ou alterar os dados de cada tabela que compõe a documentação do projeto, deve-se executá-lo, documentar a informação desejada e em seguida, executar o botão “Save table”.

Múltiplas tabelas contidas em um tópico podem ser editadas, porém apenas uma por vez.

“Header” especifica o cabeçalho da tabela.

O comando "format" junto ao comando “Edittable” permite definir no momento da edição, o formato e o tamanho de cada coluna (campo) da tabela. Cada coluna da tabela pode ter diferentes formatos tais como: texto, área de texto, seleção, botão de rádio, *checkbox* (caixa de múltipla escolha), rótulo, número de linha, data.

A cláusula “changerows” indica se linhas podem ser adicionadas e removidas (=”on”) ou apenas adicionadas (= “add”).

Formulário:

Passos a serem seguidos para se construir uma aplicação baseada em formulários *web*:

- defina um formulário modelo padrão;
- habilite o formulário criado para a *web*;
- adicione o formulário para um novo tópico;
- construa uma pesquisa formatada para listar tópicos que compartilhem um mesmo formulário.

Definindo um formulário

Um modelo de formulário especifica os seus campos e nada mais é do que uma página que contém uma tabela *Twiki*. Cada linha da tabela especifica um campo do formulário.

Devem-se seguir os seguintes passos:

- crie um novo tópico como sendo o formulário;
- nesse novo tópico, crie uma tabela *Twiki* e na primeira linha, cada coluna representando um elemento de entrada de cada campo:
    - name: Nome do campo;

- type: Tipo do campo, podendo ser: *checkbox*, botão de rádio, rótulo, seleção, texto, área de texto ou data;
- size: tamanho do campo;
- values: valores (conteúdo) do campo;
- tooltip message: mensagem exibida quando o cursor é posicionado sobre o campo no modo de edição;
- attributes: especifica atributos para o campo, podendo ser simultâneo se separado por vírgulas. Ex.: "H" indica que o campo não deve ser exibido no modo de visualização, embora ele armazene informação; "M" indica que o campo é obrigatório, não pode ser nulo.

- para cada campo, preencha uma nova linha;
- salve o tópico.

Segue abaixo exemplo de um formulário:

```
| *Name* | *Type* | *Size* | *Values* | *Tooltip message* |*Attributes*|
| Tipo   | select | 32     | Artefato,\
                             ErroSoluc,\
                             Reuniao,\
                             Arquivo  | Tipo da entidade  | M |
| Nome   | text   | 16     |          |                   | M |
| Pai    | text   | 32     | WebHome  |                   |   |
```

Nesse formulário existem três entradas: um botão de seleção, com nome "Tipo" e com opções "Artefato", "ErroSoluc", "Reuniao" e "Arquivo". Os outros dois são entradas de texto: "Nome" com 16 caracteres e "Pai" com 32.

%Search:

O comando %Search permite realizar pesquisas interessantes entre os tópicos criados. Podem-se formatar os resultados de pesquisa de uma maneira flexível e customizada.

```
%SEARCH{"^.*?\| *Nome: *\| *[^ \|]+ *\|.*"
                              web="%WEB%"
                              topic="%TOPIC%"
                              scope="text"
                              type="regex"
                              noheader="on"
                              nototal="on"
                              nosearch="on"
                              format="$pattern(.*Nome: *\| *([^
\|]+) *\|.*)"
                           }%
```

No exemplo acima, destacam-se as seguintes declarações:

- *web*: *web* a ser pesquisada (ex.: %*WEB*% significa *Web* corrente);
- topic: tópico pesquisado (ex.: %TOPIC% significa tópico corrente);
- scope: o escopo pode ser “topic” (título do tópico), “text” (corpo do tópico) ou “all” (título e corpo);
- type: tipo pode ser “keyword” (palavra-chave), “literal” ou “regex” (expressão regular);
- noheader = ”on”: suprime o cabeçalho de pesquisa padrão “Topics: Changed: By:”;
- nototal = ”on”: não exibe o número de tópicos encontrados;
- nosearch = ”on”: suprime o termo de pesquisa;
- format: permite a formatação dos resultados do modo escolhido pelo usuário. Por meio desse comando, podem ser usadas funções indicadas pelo símbolo '$'.

Em particular nesse exemplo, são procuradas tabelas que contenham a cadeia “Nome” em uma das colunas no tópico corrente. O resultado é o valor correspondente digitado para a coluna (campo) “Nome”. A função '$pattern' permite escolher um pedaço da cadeia encontrada, por meio de expressões regulares. O trecho selecionado é delimitado por '(' e ')' na expressão.

%Metasearch:

Cada tópico contém informações embutidas sobre sua criação, estado e hierarquia, que são chamadas de meta-dados. Além disso, o usuário pode incluir seu próprio conjunto de meta-dados adicionando um formulário aos tópicos de interesse, conforme será visto mais adiante.

O comando %Metasearch% permite a busca em meta-dados.

```
%METASEARCH{
    type="field"
    name="Tipo" value="Requisito"
    default="Nenhum Requisito definido ainda"
    title="*Requisitos definidos:* <br>"
    format="[[$topic][%$nopSEARCH{
              \"$topic\"
              scope=\"topic\"
              topic=\"$topic\"
              nonoise=\"on\"
              format=\"$topic\"
          }$nop%]]<br>"
}%
```

As seguintes expressões:

- type: pesquisa por tipo igual a "topicmoved" (tópicos que foram movidos), igual a "parent" (tópicos que têm um pai específico), igual a "field" (tópicos que têm um particular valor de campo de formulário; usa-se os parâmetros "name" e "value" para especificar por quais campos pesquisar);
- title: texto predefinido a ser exibido nos resultados da pesquisa;
- default: texto padrão exibido se nenhum item da pesquisa for encontrado.

Existem muitos outros comandos, documentados nas páginas de ajuda de cada instância do *Twiki*. Os exemplos aqui mostrados foram escolhidos por serem mais relevantes ao nosso trabalho. Maiores detalhes sobre o *Twiki* poderão ser encontrados por meio do endereço eletrônico <http://*Twiki*.org/>.

3.2 Outras ferramentas

Citam-se abaixo algumas ferramentas de colaboração, de identificação de erros e de documentação existentes:

O **FindBugs**, escrito na linguagem Java, é um programa fácil de se utilizar, que utiliza análise estática para identificar erros nos códigos em Java de qualquer versão, mais especificamente em seus *bytecodes*, que são arquivos de classe compilados. Disponível no endereço eletrônico <http://findbugs.sourceforge.net>, ele pode ser executado com qualquer máquina virtual compatível com o JDK 1.4 da Sun e possui uma arquitetura de *plugins*. Por meio da análise estática, esses arquivos são inspecionados (e não executados) e dessa forma, não há necessidade do código-fonte do programa. O FindBugs utiliza o BCEL (Byte Code Engineering Library) para analisar arquivos de classe Java. Essa biblioteca também é utilizada com sucesso em vários outros projetos como compiladores, otimizadores, geradores de código e ferramentas de análise. A sua licença de *software* é a GPL e sua técnica busca um equilíbrio entre precisão, eficiência e usabilidade.

Embora a ferramenta possa detectar falsas mensagens de alerta que não sejam necessariamente erros, isso representa um índice inferior a 50%.

O **FogBugz**, que pode ser encontrado no endereço eletrônico <http://www.fogcreek.com/FogBUGZ/>, é um sistema de gerenciamento de projetos que promove a comunicação entre as equipes de *software*, coordena e prioriza as tarefas atribuídas à equipe de desenvolvimento. É uma ferramenta paga baseada em *web* que facilita o acesso pela equipe envolvida. Novas solicitações, mensagens de clientes, erros, discussões relativas ao projeto são imediatamente acessados e rastreados.

Além disso, o FogBugz inclui um *Wiki* para centralizar e gerenciar toda a documentação do sistema. Podem-se armazenar a especificação funcional e técnica, documentos do projeto ou qualquer documento a ser acessado e alterado colaborativamente por membros de uma equipe. A ferramenta é fácil de ser usada, permite a edição de documentos, formatação de textos, uso de ponteiros, imagens,

arquivos anexos, tabelas, verificador ortográfico, entre outros.

O endereço <http://www.stack.nl/~dimitri/doxygen/> disponibiliza maiores informações sobre o **Doxygen**, que é uma ferramenta que analisa as declarações de código para gerar documentação compreensiva dos sistemas criados nas linguagens C++, C, Java, Python, IDL, Fortran, VHDL, PHP e C#. Embora desenvolvido sob os sistemas operacionais Linux e Macintosh, é configurado para ser portável também para Unix e Windows. A licença de *software* é a GPL.

A partir de um conjunto de fontes de arquivos, pode-se gerar a documentação *on-line* do sistema em formato HTML, RTF - Rich Text Format (Ms-Word), PostScript, PDF (Portable Document Format). Sendo gerada diretamente a partir de seu código-fonte, a documentação se torna mais consistente e real. Pode-se conhecer melhor a relação entre os elementos por meio da geração automática dos diagramas de colaboração, de classe e de herança.

Citam-se algumas características da ferramenta: compatibilidade com Javadoc, geração automática de referências cruzadas, destaque da sintaxe do código-fonte e organização dos elementos em grupos.

Nenhuma dessas ferramentas citadas apresenta uma estratégia para a inspeção de *software*. O FindBugs é específico para a linguagem Java e se concentra no acompanhamento do código. Já o Doxygen trata do código fonte e de sua documentação. Nos dois casos não há acompanhamento das outras fases de um projeto. Dos três, o FogBugz é o mais completo e possui recursos similares aos pretendidos neste trabalho, porém, trata-se de um sistema comercial, não disponível publicamente.

Capítulo 4

4 Estratégia de inspeção de *software*

A proposta deste trabalho compõe-se de duas etapas:

1) apresentar a estratégia de inspeção para ser aplicada em projetos de *software* acadêmicos orientados a objeto. Para fins de teste, foi escolhido um projeto do curso de graduação em Ciência da Computação pelo Instituto de Matemática e Estatística (IME);

2) haja vista que há menos suporte para a inspeção e que é uma atividade mais eficiente do que o teste de *software*, decidiu-se por facilitar o processo de inspeção propondo-se formatos eletrônicos de documentos por meio da ferramenta *Wiki*.

Já que as reuniões de inspeção são custosas porém imprescindíveis, podem-se minimizá-las com o uso de colaboração *on-line* e de recursos virtuais atuais como troca de mensagens eletrônicas que são registradas e podem compor atas de reunião.

Como o processo de inspeção gera documentos em papel, deve-se ter um modo de registrar eletronicamente os seus resultados e outros documentos da inspeção para atividade colaborativa de armazenamento, consulta e alteração. Podem-se utilizar ferramentas como o *Wiki* para registro e manipulação desses documentos.

Como estratégia de definição de equipe de inspeção, a partir da fundamentação em Fagan (1976), Crossman (1977) e Gilb (2000), este trabalho descreve para cada uma das fases principais do ciclo de vida do sistema, a equipe de inspeção necessária. A definição das equipes neste trabalho buscou reunir e complementar as idéias de cada um, pois assim resulta em equipes mais completas, com diferentes perfis necessários a uma boa inspeção.

A tabela abaixo apresenta as equipes de inspeção e de teste distribuídas pelas principais fases do processo de desenvolvimento de *software*:

<table>
<tr><td>Equipe de inspeção
Fase de requisitos

Moderador
Autor
Leitor
Relator
Inspetor</td><td></td></tr>
<tr><td>Equipe de inspeção
Fase de projeto

Moderador
Autor = Projetista
Leitor
Relator
Inspetor</td><td></td></tr>
<tr><td>Equipe de inspeção
Fase de código

Moderador
Autor = Codificador/Implementador
Leitor
Relator
Inspetor
Projetista
Testador
Representante do Grupo de Manutenção</td><td>Equipe de Teste
Fase de código

Moderador
Relator
Projetista
Codificador/Implementador
Testador
Representante do grupo de manutenção

obs.: os perfis Autor, Leitor e Inspetor têm maior relevância na análise estática de inspeção, não sendo considerados no teste dinâmico</td></tr>
</table>

Tabela 1 - Proposta de equipes de inspeção e de teste

Deve-se definir uma equipe de inspeção para cada artefato inspecionado dentro do ciclo de vida do sistema. Assim, por exemplo, o perfil definido a um Moderador não é necessariamente um perfil fixo para todo o ciclo de vida do sistema, mas está atribuído circunstancialmente à função de Moderador de uma determinada reunião de inspeção acerca de um artefato inspecionado de uma fase do sistema.

A equipe de inspeção definida acima identifica diferentes papéis ou responsabilidades de cada integrante. Vale ressaltar que um profissional pode

desempenhar um ou mais papéis durante um processo de inspeção e um papel também pode ser desempenhado por um ou mais profissionais.

O Moderador deve aparecer em todas as fases pois ele é o mediador que irá gerenciar e conduzir toda a equipe e o processo de inspeção. Para cada fase, o Moderador pode ser o mesmo profissional ou um outro integrante da equipe. O Moderador pode ser alguém do projeto, externo ao projeto ou pode ser até mesmo alguém externo contratado para esse fim.

Os perfis Moderador, Autor, Leitor, Relator e Inspetor são os perfis em comum que aparecem em todas as fases da inspeção. Todos esses profissionais podem ser participantes do projeto ou podem ser externos ao projeto.

Entende-se como Autor o criador do documento inspecionado que deve compor a equipe para melhor esclarecimento, análise e também contribuir para a identificação dos defeitos. Na fase de requisitos, o Autor é o profissional que levanta e elucida os requisitos do sistema. Na fase de projeto, o Autor é o profissional que cria o projeto. Na fase de código, o Autor é o Codificador.

O Leitor é o papel desempenhado de leitura dos documentos analisados nas reuniões de inspeção para que sejam discutidas nesse momento todas as questões relevantes à inspeção.

O Relator é o profissional que registra todas as informações relevantes das reuniões de inspeção como por exemplo, registro dos defeitos ou qualquer outra informação importante que precisem ser registradas.

O papel Inspetor é atribuído a todos os profissionais incumbidos da responsabilidade de analisar e inspecionar documentos procurando por defeitos propriamente ditos ou defeitos potenciais que poderão causar problemas futuros.

O perfil Projetista aparece a partir da fase de projeto e desse ponto o perfil começa a atuar. Na fase do projeto, o Autor é o próprio Projetista que criou o projeto do sistema.

Os perfis Codificador, Testador e Representante do Grupo de Manutenção aparecem somente na fase de código, pois é nessa fase que esses perfis são necessários. Na fase de código, o Autor é o próprio Codificador que desenvolveu o programa. A participação do Testador e do Grupo de manutenção é importante na

inspeção para que se familiarizem com cada levantamento, análise e registro dos defeitos encontrados nos artefatos e assim, quando forem desempenhar suas atividades, interajam mais com o processo de desenvolvimento e tenham uma atitude mais pró-ativa.

A tabela abaixo apresenta os procedimentos do processo de inspeção:

Processo de inspeção
1) seleção da equipe de inspeção; 2) reunião inicial sobre o sistema, o objetivo da inspeção e a identificação de critérios de inspeção; 3) leitura e análise (não execução) do documento de inspeção, por cada membro, individualmente e registro das ambigüidades e dos defeitos encontrados; 4) condução de reuniões sobre os defeitos encontrados e suas causas, análise da criticidade e gravidade de cada defeito. Nessas reuniões deve-se realizar: • análise da causa dos defeitos; • obtenção de consenso sobre os problemas encontrados; • definição de prioridades da correção dos defeitos de acordo com a sua criticidade. 5) correção dos erros e eliminação das ambigüidades pelo criador do artefato; 6) organização, classificação e registros dos erros, pelos responsáveis, por tipo, gravidade, freqüência, tempo dispendido na identificação, na correção e na solução de cada erro; 7) condução de reuniões sobre a classificação e a correção dos erros e dos defeitos.

Tabela 2 - Processo de inspeção de *software*

O processo de inspeção descrito acima está fundamentado nas idéias de Doolan (1992), Fagan (1976) e Mashayekhi et al. (1993). É importante que haja um conjunto de passos a serem seguidos para orientar a inspeção de um modo organizado, e dessa forma, evitar que se esqueça de algum passo necessário.

O princípio de tudo é definir o Moderador e a equipe de inspeção do projeto, o que pode ser feito por meio da implementação no *Twiki*, visto com detalhamento no capítulo anterior. Nesse ambiente, pode-se registrar a equipe de inspeção de um modo global para cada fase do projeto. Adicionalmente, pode-se também definir uma equipe de inspeção para cada artefato inspecionado. Devem-se definir as responsabilidades de cada membro da equipe. É importante ressaltar que o papel de Inspetor não deve ser atribuído aos próprios criadores do artefato inspecionado.

As reuniões sobre os objetivos e os critérios da inspeção são importantes pois todos precisam ter conhecimento de suas responsabilidades nessa tarefa e ter maior explicação sobre o sistema para facilitar a familiarização com o ambiente e a eficiência da inspeção. Além disso, o objetivo e expectativa de todos devem estar bem definidos e bem alinhados.

O acesso e a distribuição do documento de inspeção são facilitados com a sua disponibilização no ambiente do *Twiki*, onde cada membro da equipe de inspeção poderá entrar na página, visualizar os documentos e interagir colaborativamente, incluindo e alterando dados pertinentes à inspeção do projeto. Para a leitura e análise do documento de inspeção, verificação de não-conformidades ou problemas, pode-se visualizar no próprio *Twiki* ou se desejado, pode-se imprimi-lo para análise posterior.

Os *checklists* ajudam na identificação de defeitos durante a análise do documento. Eles servem como guia, orientação e lembrete para a inspeção e para verificar informações ambíguas, omitidas, incorretas ou imprecisas. O próximo capítulo, que apresentará a implementação do *Twiki*, mostrará que durante a criação dos vários artefatos, existirá também o processo de sua inspeção orientado pelos *checklists.*

O registro dos defeitos encontrados também pode ser feito diretamente no ambiente *Twiki*, onde pode-se registrar cada problema encontrado em cada artefato, sua causa e sua respectiva solução.

Precisa-se discutir a análise de cada membro da equipe sobre o artefato inspecionado. As reuniões sobre os defeitos encontrados são necessárias e importantes pois contribuem para um melhor entendimento das causa dos defeitos, planejamento de correção e de prioridades de acordo com a sua criticidade.

A correção dos erros deve ser feita preferencialmente pelo criador do artefato pois assim, ganha-se agilidade e eficiência nessa atividade.

Uma vez registrados os defeitos no ambiente de colaboração *Twiki*, é de responsabilidade do Moderador monitorar e garantir a correção de cada problema identificado em tempo hábil, respeitando a sua criticidade e prioridade.

Deve-se registrar cada defeito e cada solução para que se tenha uma base histórica de pesquisa de erros e suas respectivas soluções, incluindo a emissão de

relatórios. A idéia é prever os erros que possam ocorrer e já dispor de soluções ou planos de contingência para que a aplicação seja o mais estável possível. Além disso, estatisticamente é interessante o registro do tempo consumido na identificação e correção dos problemas classificados por gravidade para um melhor desempenho contínuo do processo.

São necessárias reuniões para discussão e alinhamento do conhecimento de todos sobre a classificação dos erros e as respectivas medidas corretivas.

É importante também considerar a melhoria contínua do processo de inspeção, o que inclui monitoramento, análise e ações corretivas e de aperfeiçoamento do processo. Esse processo não é estático, precisa ser monitorado para se identificar deficiências e novas necessidades.

A seguir, são apresentadas tabelas contendo *checklists* relacionados à inspeção das Fases de Análise, de Projeto e de Código:

Checklists
Fase de Análise

Requisitos (KIT, 1995)

- analise com cuidado termos persuasivos como: certamente, portanto, evidentemente, obviamente;
- atente para palavras como: algum, às vezes, freqüentemente, geralmente, maior parte das vezes;
- quando se referir a uma lista de itens inacabada, assegure-se que o significado da formação da lista tenha sido bem entendido. Atente para termos como: etc., assim por diante;
- atente para listas sem exemplos ou exemplos insuficientes para o seu completo entendimento;
- atente para verbos vagos como: processado, rejeitado, eliminado;
- cuidado com construções passivas, pois elas não determinam quem é o autor que está realizando o trabalho;
- cuidado com o uso de pronomes que são claros para o autor, mas não para o leitor.

Devem-se analisar e verificar as seguintes características dos requisitos:
- completo: verifique se todos os itens imprescindíveis para a solução do problema foram abrangidos;
- correto: analise se cada item especificado está correto;
- preciso, não-ambíguo e claro: identifique se cada item está legível, sem dupla interpretação ou ambigüidades;

- consistente: analise se não há idéias conflitantes;
- relevante: verifique se todas as informações são pertinentes ao problema;
- testável: é possível se testar cada item dos requisitos e verificar a conformidade com a especificação?
- flexível: pode-se alterar itens da especificação sem que ocorra grande impacto nos outros itens?

Tabela 3 - *Checklists* da fase de Análise

A estratégia de inspeção deste trabalho foi baseada no modelo V que sugere a inspeção propriamente dita das principais fases do desenvolvimento de sistemas. O modelo U, conforme citado neste trabalho, não foi adotado por sugerir a simulação do produto após determinadas fases, o que não faz parte do escopo deste trabalho.

Explicitar requisitos não é uma tarefa simples porque pode haver dupla interpretação, desentendimento, ambigüidades. Trata-se da fase inicial do processo de desenvolvimento a partir da qual todas as outras fases subseqüentes serão desenvolvidas. Assim, é extremamente importante que os requisitos sejam muito bem e corretamente definidos e por isso foi apresentado um conjunto de orientações para analisar e verificar cada requisito, conforme fundamentado por Kit (1995).

Checklists
Fase de Projeto

Classes

- integridade das classes: Analise se as classes definidas são corruptíveis (não protegem contra o acesso externo dos recursos) ou protegidas (possui mecanismo de proteção dos recursos por controle de acesso) (PRESSMAN, 2002).

Para a correta identificação das classes, verifique se a classe definida faz parte de um dos itens abaixo (PRESSMAN, 1995):

- entidades externas (exemplo: outros sistemas, dispositivos, pessoas que produzem ou consomem informação usada por um sistema);
- coisas (exemplo: relatórios);
- ocorrências ou eventos (exemplo: instalação);
- papéis (funções) (exemplo: gerente, engenheiro);
- unidades organizacionais;
- lugares;
- estruturas (exemplo: sensores);
- outros.

Herança

Para cada classe (DUNSMORE, ROPER e WOOD, 2003):

- as heranças são apropriadas e adequadamente implementadas de acordo com o projeto?

Reuso
O analista de domínio da aplicação deve analisar classes e objetos potenciais para o reuso (PRESSMAN, 1995):

- identifique os objetos candidatos a reuso;
- indique as razões pelas quais o objeto foi identificado para reuso;
- defina adaptações ao objeto que possam também ser reusadas;
- estime o percentual (%) de aplicações do domínio que poderia fazer reuso do objeto;
- estime o percentual (%) que uma aplicação poderia ser construída utilizando objetos reusáveis.

Requisitos de Projeto
Deve-se avaliar o modelo para certificar de que os requisitos desejados estejam sendo satisfeitos com relação a (KIT, 1995) :

- requisitos de desempenho (ex.: tempo de resposta do sistema);
- facilidade do uso do sistema (navegação, interatividade, uso dos menus);
- simplicidade (facilidade de entendimento e dedução do sistema);
- modularidade (habilidade do sistema reunir informações vistas como uma unidade para diminuir a complexidade, possibilitar um melhor entendimento, facilitar o reuso e diminuir o impacto das alterações);
- manutenibilidade (facilidade do sistema ser alterado para implementar mudanças, adaptações ou correções necessárias);
- portabilidade (facilidade em que o sistema pode ser portado para uma plataforma diferente).

(LAITENBERGER e ATKINSON, 1999)

- analise o grau de Coesão do módulo, ou seja, a robustez funcional relativa de um módulo, indicação qualitativa do grau em que um módulo focaliza apenas uma coisa (o ideal é alta coesão);
- analise o grau de Acoplamento do módulo, ou seja, a medida da interdependência relativa entre módulos (o ideal é baixo acoplamento).

Tabela 4 - *Checklists* da fase de Projeto

A identificação das classes, conforme fundamentação de Pressman (1995), é imprescindível para sistemas orientados a objeto. É necessário se assegurar que as classes foram identificadas corretamente, se realmente fazem parte da classificação de classes. Além disso, deve-se verificar a integridade das classes e a proteção

contra o acesso externo. As classes precisam ser confiáveis, robustas e não vulneráveis. Dessa forma, foram definidas dicas para identificação das classes e verificação de suas integridades.

As heranças devem ser definidas baseadas na análise de toda a aplicação.

Uma das características e princípios de sistemas orientados a objeto é o reuso. É preciso analisar criteriosamente os objetos candidatos ao reuso, os motivos dessa escolha, as aplicações que poderiam reutilizá-los, conforme fundamentado por Pressman (1995). A análise do reuso no processo de inspeção foi definida neste trabalho devido à sua importância de se poupar tempo e esforços e evitar a criação desnecessária de novos objetos que tenham as mesmas propriedades de outros objetos existentes.

Definir requisitos funcionais é essencial pois é a base de todo o sistema. Adicionalmente, em se tratando de projeto, outros itens que precisam ser levados em consideração são o desempenho, a facilidade de uso, a simplicidade, a manutenibilidade e a portabilidade do sistema. Todos esses aspectos não são menos importantes que os requisitos funcionais do sistema e portanto, precisam ser considerados e estão definidos como *checklists* neste trabalho, fundamentado em Kit (1995). Por exemplo, se um sistema atende a todos os requisitos funcionais, mas tem problemas de desempenho, não é intuitivo, apresenta dificuldades em sua navegação, não está estruturado de uma forma que adapte modificações ou novas necessidades, indubitavelmente, não terá uma longa vida útil ou poderá ser abandonado.

Outro aspecto considerado neste trabalho é o grau de acoplamento e coesão dos módulos, fundamentado em Laitenberger e Atkinson (1999). São aspectos importantes que devem ser avaliados para facilitar a manutenção do sistema pois as equipes de desenvolvimento/manutenção não são eternas, equipes diferentes do desenvolvimento podem realizar a manutenção. Quanto maior a coesão e menor o acoplamento dos módulos, melhor será a sua manutenibilidade.

Checklists
Fase de código

Variáveis (KIT, 1995)
Erros de referência de dados/uso de variáveis
- existe alguma variável referenciada não expressamente definida ou não inicializada?
- foram atribuídos valores às variáveis compatíveis com seus respectivos tamanhos e tipos?
- existem variáveis com nomes similares? Isso não é necessariamente um erro, mas deve-se analisar se não ocorre uma confusão entre o uso das variáveis com nomes similares;
- verifique se todas as variáveis definidas são utilizadas e referenciadas.

Conversão
- existe tratamento para conversão entre tipos de variáveis com tipo de dados ou tamanhos inconsistentes?

Erros relacionados a cálculo ou expressões
- verifique se os cálculos são efetuados usando variáveis com tipos de dados inconsistentes;
- há cálculos que utilizam variáveis com mesmo tipo de dados mas diferentes tamanhos? Verifique se não poderão ocorrer problemas com o limite máximo do tamanho da variável;
- verifique se a variável que vai receber o resultado de um cálculo não é de tamanho menor do que o tamanho do resultado da expressão do lado direito?
- é possível se ter uma divisão por zero?
- analise expressões contendo mais do que um operador e verifique se a ordem da execução dos operadores está sendo feita de maneira correta.

Erros de controle de laço (iteração)
- verifique se foi definido um laço cuja condição de execução nunca ocorre e se dessa forma, isso representa um bloco de código desnecessário;
- há algum laço cuja condição de saída nunca ocorre e quais as conseqüências?

Parâmetros e argumentos
- o número, a ordem, o tipo, o tamanho dos parâmetros recebidos estão em conformidade com o que é esperado pelo módulo chamado?

Arquivo
- todos arquivos foram abertos antes do uso?
- a condição de fim de arquivo (EOF - End of File) e de erro de E/S foram previstos e tratados?

Construtor de classe (DUNSMORE, ROPER e WOOD, 2003)

- todas as variáveis de instância (objetos) foram inicializadas com valores significativos e adequados?
- se no construtor em referência é necessária uma chamada ao construtor da superclasse, isso está definido adequadamente?

Para cada classe (DUNSMORE, ROPER e WOOD, 2003):

Sobrescrita de método

- todos os métodos herdados que têm comportamentos diferentes foram sobrescritos adequadamente?

Para cada método (DUNSMORE, ROPER e WOOD, 2003):

- todos os parâmetros são utilizados dentro do método?
- o parâmetro de retorno contém um tipo de dado correto e adequado à definição?

Outras verificações (KIT, 1995)

- verifique se o programa foi compilado com sucesso, mas gerou mensagens de alerta que devem ser analisadas;
- avalie o número e qualidade dos comentários definidos para as principais partes do código.

Tabela 5 - *Checklists* da fase de Código

O código do programa também merece a atenção necessária sobre diferentes aspectos, conforme fundamentação de Kit (1995). O intuito de se atentar para as partes do código é minimizar e evitar os erros futuros que possam ocorrer, não somente os erros críticos que causam a interrupção imediata do programa, como também erros apresentados diretamente ao usuário final. Dessa forma, este trabalho faz considerações importantes sobre diversas questões de código tais como:

- variáveis: por existirem inúmeras variáveis, deve-se ter o cuidado e certificar de que todas têm conteúdo inicializado devidamente, não ter valores imprevisíveis, prever o seu conteúdo conforme o seu tipo e tamanho, criar variáveis com nomes mnemônicos para facilitar a manutenção e a clareza do código, atentar para as conversões de variáveis. Existe uma relação de tipos de variáveis que podem ser convertidas para que não haja estouro de campo ou perda de informações, prejudicando a integridade dos dados. Conforme ocorram alterações no código, variáveis podem ser criadas e posteriormente, não mais

utilizadas; assim, deve-se verificar se todas variáveis definidas são referenciadas;

- cálculos e expressões: deve-se atentar e prever se algum cálculo pode gerar alguma divisão por zero que possa causar a interrupção do programa, analisar se uma determinada expressão contendo uma grande quantidade de operadores e operandos respeita a prioridade das operações de maneira correta, pois caso contrário, resultará em erros nos cálculos e falta de confiabilidade das informações apresentadas;
- laços ou iteração: qualquer programa tem pelo menos um laço que executa um conjunto de instruções em um número determinado de vezes ou repetições. Muitas vezes, realizam-se tantas alterações de código, resultando em blocos desnecessários ou laços que nunca serão executados. Outra hipótese é ter laços infinitos que nunca têm condições de saída. Assim, deve-se fazer uma análise minuciosa de cada laço definido para evitar laços infinitos ou desnecessários;
- parâmetros: é importante verificar se os parâmetros são utilizados realmente ou se foram apenas definidos mas nunca referenciados, o que resulta em códigos menos eficientes e mais complexos para manutenção. Deve-se verificar também a conformidade dos parâmetros recebidos e de retorno com a sua definição de tipo, tamanho e ordem;
- arquivos: embora pareça óbvio, todos arquivos definidos devem ser abertos antes do uso. Dependendo da quantidade de arquivos existentes, pode ocorrer deles serem utilizados sem que se tenha aberto previamente. Além disso, é muito importante que se verifiquem as condições de fim de arquivo para que não ocorram interrupções de execução por esse motivo;
- método: deve-se analisar o comportamento dos métodos herdados, se há necessidade de sobrescrita e nesse caso, se todas as sobrescritas de método foram feitas adequadamente.

Não é o intuito deste trabalho aprofundar o estudo sobre o processo de teste de *software*. Porém, é importante citar aspectos do processo de teste de maneira análoga ao processo de inspeção.

A tabela abaixo apresenta os procedimentos do processo de teste:

Processo de Teste
1) seleção da equipe de teste; 2) avaliação e análise da especificação e dos requisitos funcionais do sistema (em caso de teste de caixa preta); 3) avaliação e análise do código-fonte (em caso de teste de caixa branca); 4) elaboração dos casos de teste de modo a maximizar a detecção de falhas e minimizar o número de casos de teste; 5) execução dos testes por quem não desenvolveu os programas; 6) análise dos resultados da execução dos casos de teste sob o ponto de vista da precisão, do desempenho e da usabilidade; 7) registro dos resultados da execução para cada caso de teste; 8) registro de cada defeito por meio de relatório de erros; 9) plano de correção para cada defeito; 10) correção dos defeitos pelo criador do programa; 11) organização e classificação dos erros por tipo, freqüência, tempo dispendido na identificação e na correção; 12) o Moderador deve se assegurar que os erros foram corrigidos, acompanhar e monitorar cada erro identificado.

Tabela 6 - Processo de teste de *software*

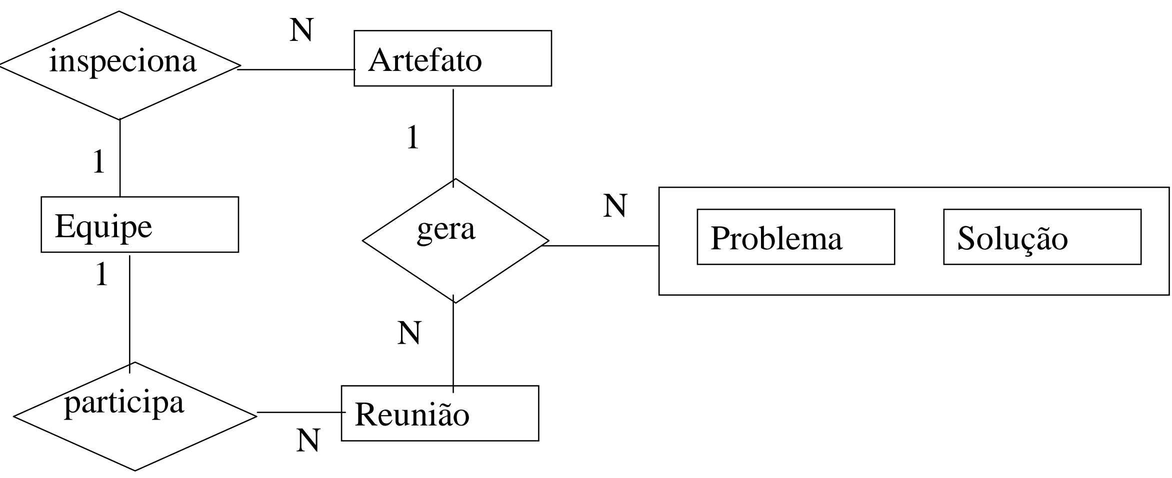
Capítulo 5

5 Implementação da Proposta

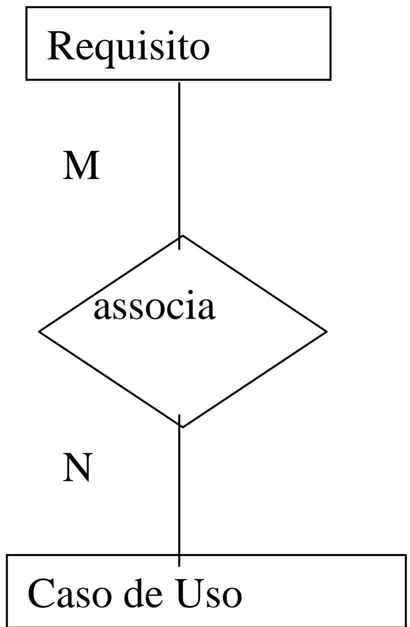
5.1 MER (Modelo Entidade-Relacionamento)

Foi proposto o Modelo Entidade-Relacionamento das Fases de Inspeção, Análise, Projeto e Código. Foram representados os principais artefatos utilizados neste trabalho e suas respectivas implementações no *Twiki*. As diferentes Fases Análise, Projeto e Código podem ser interligadas, entretanto, isso foi deixado para uma implementação futura. Componentes e pacotes têm a sua devida importância na Orientação a Objetos, porém também não fizeram parte do escopo deste trabalho.

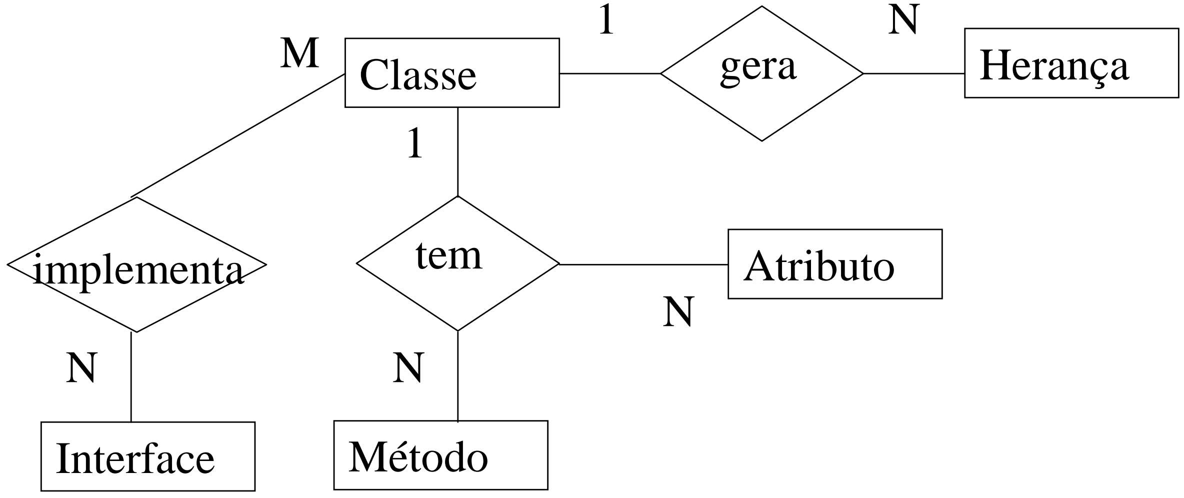
INSPEÇÃO



FASE DA ANÁLISE



FASE DO PROJETO



FASE DO CÓDIGO

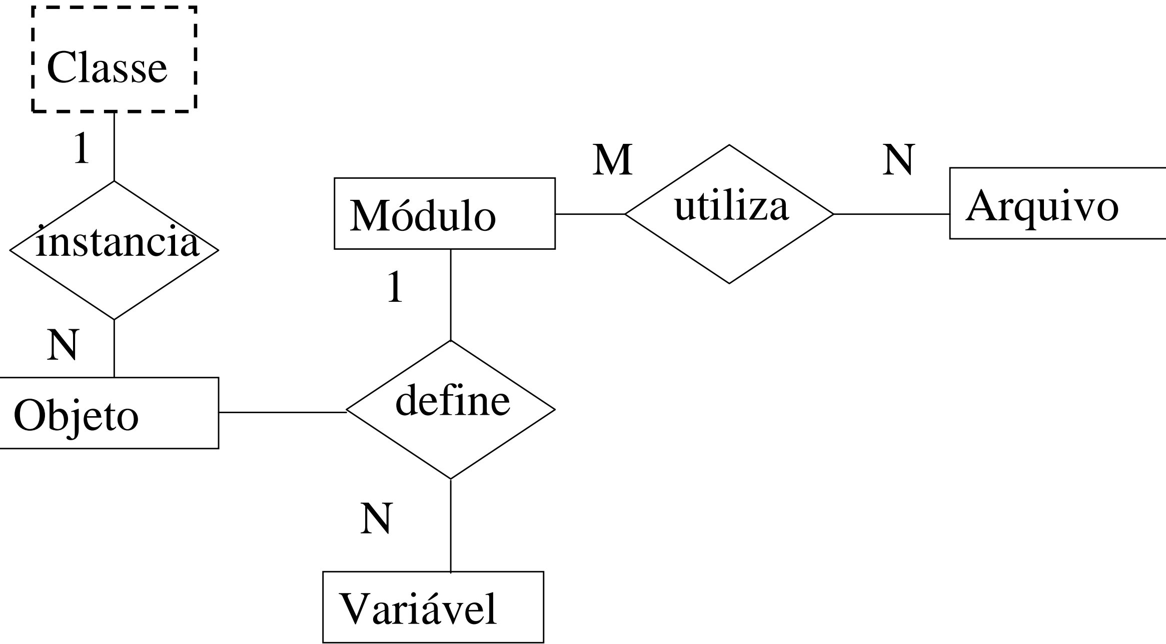


Figura 4 – Modelo Entidade-Relacionamento proposto

5.2 Ambiente *Twiki* proposto

Neste trabalho, optou-se por utilizar o *Twiki*. O objetivo é que o usuário preencha formulários e as informações sejam salvas, possibilitando um acompanhamento do desenvolvimento de todo o projeto.

Como parte de ilustração da estratégia de trabalho, foi criado um ambiente de implementação parcial para se realizar a documentação do projeto orientado a objetos, desde a definição dos requisitos, dos casos de uso, criação das classes, das heranças, até a criação de módulos, variáveis, arquivos. Além disso, pode-se também definir a equipe do projeto e a equipe de inspeção. Paralelamente à documentação do projeto, existem diversos *checklists* a serem aplicados nas diferentes fases do desenvolvimento. Dessa forma, a ênfase do trabalho não foi a implementação do *Twiki*, mas sim a apresentação da estratégia de inspeção.

Optou-se por se ter controle de acesso de modo que somente usuários registrados possam visualizar e alterar informações no ambiente *Twiki*, ou seja, usuários visitantes não terão acesso.

Em relação aos perfis e permissões de acesso de usuários, embora sejam fáceis de serem implementados, foram deixados para uma versão posterior, uma vez que se trata apenas de uma prova de conceito.

O ambiente *Twiki* proposto pode ser acessado por meio do endereço eletrônico <http://mairinque.ime.usp.br/cgi-bin/*Twiki*/view/Aventura/WebHome>.

Como ponto de partida, foi criado o tópico (página) acessível pelo ponteiro SistemaOrientadoObjeto. Observa-se que todas as ações são realizadas executando os diferentes ponteiros existentes na página ou acionando os botões presentes nos tópicos.

Pode-se definir a equipe de projeto, por exemplo, definir perfis como Gerente do Projeto, Patrocinador do Projeto, Arquiteto do Sistema, Analista de Requisitos, Administrador de Dados, Administrador de Banco de Dados, Programador, Analista de Testes. Além disso, pode-se incluir, alterar ou excluir outros perfis da equipe do projeto. Para tanto, foi definido o seguinte comando:

```
%INCLUDE{EquipeProjeto}%
```

O tópico EquipeProjeto por sua vez executa o seguinte comando:

```
%EDITTABLE{header="| *Equipe do Projeto* | *Preencha adequadamente* |"
format="| text, 20,  | text, 50,  |" changerows="on"}%
|*Equipe do Projeto*|*Preencha adequadamente*|
| Gerente do Projeto |  |
| Patrocinador do Projeto |  |
| Arquiteto do Sistema |  |
| Analista de Requisitos |  |
| Administrador de Dados |  |
| Administrador de Banco de Dados |  |
| Programador |  |
| Analista de Testes |  |
```

Ao executar o ponteiro PropostaInspecao, a estratégia de inspeção é apresentada como uma forma de guia para que cada passo seja executado de um modo organizado.

A tela foi dividida em três principais Fases: Fase da Análise, do Projeto e do Código.

Como no *Twiki* os tópicos devem ter nomes únicos e como também são necessárias várias formas de busca, optou-se por utilizar um formulário base referenciado por outros tópicos. Nesse formulário, foram definidos os diferentes tipos de artefatos existentes, como por exemplo, Reunião, Requisito, Caso de Uso, entre outros. Dessa forma é possível associar artefatos de mesmo nome a tópicos diferentes (métodos, por exemplo) e também categorizar cada tópico de acordo com o tipo de artefato.

Detalhes são exibidos no tópico InspecaoForm:

```
| *Name* | *Type* | *Size* | *Values* | *Tooltip message* |
*Attributes* |
| Tipo   | select | 32     | Artefato,\
                             ErroSoluc,\
                             Reuniao,\
                             Equipe,\
                             EquipeAnalise,\
                             EquipeProjeto,\
                             EquipeCodigo,\
                             Requisito,\
                             CasoUso,\
                             Classe,\
                             Interface,\
                             Metodo,\
                             Atributo,\
```

```
                                   Objeto,\
                                   Modulo,\
                                   Auxiliar,\
                                   Variavel,\
                                   Arquivo,\
                                   Projeto,\
                                   Codigo     | Tipo da entidade   | M |
| Nome    | text    | 16       |            |                    | M |
| Topico  | text    | 32       |            |                    |   |
| Pai     | text    | 32       | WebHome    |                    |   |
| Inspec  | select  | 3        | nao,sim    | Foi inspecionado?  |   |
```

Na fase de Análise, os ponteiros:

CriaRequisitos	permite criar requisitos e também o registro da inspeção de cada requisito criado.
CriaCasoUso	permite criar casos de uso, definir atores primários e secundários, fluxos de eventos básico e alternativo, pré e pós-condições.

Pode-se também definir a equipe de inspeção da fase da Análise.

O ponteiro CriaRequisitos, ao ser acessado, exibe um botão com o mesmo nome e executa os comandos abaixo:

```
<form name="InspAnalReq"
action="%SCRIPTURL{"save"}%/%WEB%/RequisitoAUTOINC00">
       <input type="hidden" name="templatetopic"
value="InspecaoAnaliseRequisitos" />

       <input type="hidden" name="formtemplate"
value="InspecaoForm" />
       <input type="hidden" name="Nome"
value="RequisitoAUTOINC00" />
       <input type="hidden" name="Pai" value="CriaRequisitos"
/>
       <input type="hidden" name="Tipo" value="Requisito" />
       <input type="hidden" name="Topico"
value="RequisitoAUTOINC00" />

       <input type="hidden" name="topicparent"
value="CriaRequisitos"/>
       <input type="submit" class="twikiSubmit"  value="Cria
Requisitos" />
</form>
```

Conforme verificado acima, o formulário base que está sendo utilizado é o “InspecaoForm”.

Como trata-se de criação de requisitos, o campo do formulário “Tipo” foi atribuído para “Requisito”

O comando:

```
      <input type="submit" class="twikiSubmit"  value="Cria
Requisitos" />
```

apresenta um botão com o rótulo "Cria Requisitos".

O botão CriaRequisitos, ao ser pressionado, cria um novo tópico com o formato padrão RequisitoAUTOINC00, ou seja, criará requisitos no formato Requisito00, Requisito01, Requisito02 e assim por diante, definido no seguinte comando:

```
action="%SCRIPTURL{"save"}%/%WEB%/RequisitoAUTOINC00"
```

No ambiente proposto *Twiki*, para cada artefato criado por meio do botão "Cria", são gerados tópicos na *Web* definida, ou seja, cada artefato criado é um novo tópico.

Em seguida, executa o tópico padrão "InspecaoAnaliseRequisitos", por meio do comando abaixo:

```
      <input type="hidden" name="templatetopic"
value="InspecaoAnaliseRequisitos" />
```

Esse tópico padrão por sua vez executa os comandos resumidos abaixo, para definir os Requisitos e inspecioná-los:

```
%EDITTABLE{header="| *Requisito* | *Preencha adequadamente* |"
format="| label, 20,  | text, 50,  |" changerows="off"}%
|*Requisito*|*Preencha adequadamente*|
| Nome do Requisito: |
| Descrição do Requisito: |

%EDITTABLE{header="| *Checklist de Análise dos Requisitos* |
*Preencha adequadamente* |" format="| label, 20,  | text, 50,
|" changerows="off"}%
|*Checklist de Análise dos Requisitos*|*Preencha
adequadamente*|
| Data da Inspeção: |
| Nome do(s) inspetor(es): |
```

```
%METASEARCH{
    type="field"
    name="Tipo" value="Requisito"
    default="Nenhum Requisito definido ainda"
    title="*Requisitos definidos:* <br>"
    format="[[$topic][%$nopSEARCH{
                \"$topic\"
                scope=\"topic\"
                topic=\"$topic\"
                nonoise=\"on\"
                format=\"$topic\"
            }$nop%]]<br>"
}%
```

O comando Metasearch acima permite pesquisar todos os tópicos criados cujo tipo definido seja “Requisito”.

Conforme explanado no capítulo anterior, o comando "format" utilizado dentro de uma pesquisa tem a função de formatar a apresentação da saída da pesquisa. No exemplo acima, será apresentado o ponteiro para o tópico pesquisado, ou seja, para todos os Requisitos definidos.

Para definir a Equipe de inspeção da Fase da Análise, foi necessário o seguinte comando:

```
%INCLUDE{EquipeInspecaoAnalise}%
```

O tópico “EquipeInspecaoAnalise” por sua vez executa o seguinte comando:

```
%EDITTABLE{header="| *Equipe de Inspeção de Análise* |
*Preencha adequadamente* |" format="| label, 20,  | text, 50,
|" changerows="off"}%
| *Equipe de Inspeção de Análise* | *Preencha adequadamente* |
|Moderador|
|Autor|
|Leitor |
|Relator|
|Inspetor|
```

Na fase de Projeto, os ponteiros:

CriaInterface	possibilita criar todas as *Interface*s do projeto, seus atributos e definições dos métodos.
CriaClasse	possibilita criar todas as Classes e também a inspeção de cada uma.
InspecaoProjeto	possibilita realizar a inspeção de importantes aspectos do projeto tais como: desempenho, usabilidade, simplicidade, modularidade, manutenibilidade e portabilidade.

No momento em que se definem as Classes (ou posteriormente), podem-se também definir todos os seus atributos com seus respectivos formatos.

Por meio do próprio ponteiro CriaClasse, podem-se definir todas as Heranças das Classes bases definidas em CriaClasse. Nesse momento também pode-se realizar a inspeção de cada Herança.

É também na Fase de Projeto que pode-se definir a equipe de inspeção dessa fase.

No tópico CriaClasse abaixo, podem-se ressaltar algumas observações:

```
%EDITTABLE{header="| *Classe* | *Preencha adequadamente* |"
format="| label, 20,  | text, 50,  |" changerows="off"}%
|*Classe*|*Preencha adequadamente*|
| Nome: |  |

<!--

   * Local ClassName = %SEARCH{"^.*?\| *Nome: *\| *[^ \|]+
*\|.*"
                           web="%WEB%"
                           topic="%TOPIC%"
                           scope="text"
                           type="regex"
                           noheader="on"
                           nototal="on"
                           nosearch="on"
                           format="$pattern(.*Nome: *\| *([^
\|]+) *\|.*)"
                        }%

-->
```

Conforme exposto anteriormente, o comando Edittable permite que se edite uma tabela e que seja preenchido o nome da Classe que se deseja criar.

Foi definida uma variável ClassName para que, por meio do comando Search, possa se obter o nome da Classe criada. É interessante ressaltar que a pesquisa pelo nome da Classe é feita por meio de expressão regular.

```
<form name="InspAnalCls" action="%SCRIPTURL{"save"}%/%WEB%/%ClassName%">
       <input type="hidden" name="templatetopic"
value="InspecaoAnaliseClasse" />
       *Classe base:*
           <select name="topicparent">
                %METASEARCH{
                     type="field"
                     name="Tipo" value="Classe"
                     default="<option $marker>Nenhuma classe definida
ainda</option>"
                    format="%$nopSEARCH{
                            \"$topic\"
                            scope=\"topic\"
                            topic=\"$topic\"
                            nonoise=\"on\"
                            format=\"<option
$marker>$formfield(Nome)</option>\"
                        }$nop%"
}%

           </select>
```

Conforme exposto no quadro acima, para cada nova Classe criada, um tópico associado é criado e o tópico padrão que sempre é executado a partir dessa criação é o "InspecaoAnaliseClasse".

O campo “topicparent” é o que permite que se defina o tópico-pai para o novo tópico que está sendo criado. Para tanto, foi utilizado o comando “Select” junto com o comando “Metasearch” para que se pesquise apenas os artefatos do tipo Classe que foram criados. Somente esses por sua vez podem ser selecionados como tópico-pai da nova Classe criada.

No *Twiki*, para cada novo tópico criado, pode-se definir a sua hierarquia, indicar a qual tópicos estará subordinado.

A seguir, algumas considerações sobre o tópico “InspecaoAnaliseClasse”:

```
*Nome da Classe:  %TOPIC%*
Classes ancestrais: %META{"parent" nowebhome="off"}%
Classe Mãe: %META{"parent" dontrecurse="on"}%
Classes Filhas: %METASEARCH{
                topic="%TOPIC%"
                type="parent"
                dontrecurse="on"
                format="%$nopSEARCH{
```

```
                            \"META:FIELD.*?value=\"Classe\"\"

                            format=\"[[$formfield(Nome)][$formfield(Topico)]]
                            \"

                            type=\"regex\"

                            scope=\"text\"

                            topic=\"$topic\"

                            nototal=\"on\"

                            nosearch=\"on\"

                            dontrecurse=\"on\"
                    }$nop%"
               }%
```

Para cada nova Classe criada, é interessante observar a hierarquia das Classes, como por exemplo, qual a Classe mãe (Classe base), a hierarquia das Classes ancestrais e as Classes filhas (ou subclasses). Essas pesquisas são efetuadas por meio dos comandos “%Meta” e “%Metasearch”, conforme descrito acima.

O comando abaixo

```
%METASEARCH{
                  topic="%TOPIC%"
                  type="parent"
```

pesquisa todos os tópicos que tenham como tópico-pai o tópico atual (corrente).

É interessante observar que o comando ”META:FIELD.*?value=\"Classe\" pesquisa, dentre todos os tópicos-filhos do tópico atual, os tópicos que tenham um formulário associado cujo campo tenha valor igual a “Classe”.

Ou seja, por meio do uso de formulários, pôdem-se categorizar os Artefatos criados como sendo por exemplo, Classe, Método, Objeto, entre outros, e dessa forma, facilitar a sua pesquisa pelo tipo do Artefato. Trata-se de uma pesquisa aninhada pois é uma pesquisa dentro de outra subpesquisa.

```
<form name="InspMet"
action="%SCRIPTURL{"save"}%/%WEB%/%TOPIC%Metodo%MethodName%">
       <input type="hidden" name="templatetopic" value="InspecaoMetodo" />
       <input type="hidden" name="formtemplate" value="InspecaoForm" />
       <input type="hidden" name="Tipo" value="Metodo" />
       <input type="hidden" name="Nome" value="%MethodName%" />
       <input type="hidden" name="Topico" value="%TOPIC%Metodo%MethodName%"
/>
       <input type="hidden" name="Pai" value="%TOPIC%" />
       <input type="hidden" name="topicparent" value="%TOPIC%"/>
       <input type="submit" class="twikiSubmit"  value="Cria Método" />
</form>

*Métodos*: %METASEARCH{
                              topic="%TOPIC%"
                              type="parent"
                              dontrecurse="on"
                              format="%$nopSEARCH{

\"META:FIELD.*?value=\"Metodo\"\"

format=\"[[$topic][%META{"formfield" name="Nome"}%]]\"
                                                    type=\"regex\"
                                                    scope=\"text\"
                                                    topic=\"$topic\"
                                                    nototal=\"on\"
                                                    nosearch=\"on\"
                                                    dontrecurse=\"on\"
                                       }$nop%"
                           }%
```

Os métodos são criados a partir da criação das Classes, ou seja, após a definição das Classes, definem-se os métodos que cada Classe possui.

No ambiente *Twiki*, não é necessário criar métodos no mesmo momento da criação das Classes. Caso se deseje acessar uma Classe já criada, executa-se o próprio ponteiro CriaClasse, informa-se o nome da Classe criada anteriormente e será apresentada a Classe com a última informação salva. Dessa forma, o ponteiro CriaClasse deve ser usado não somente para a criação de novas Classes, mas também acessar Classes existentes.

O procedimento descrito acima é o mesmo utilizado para a criação e acesso de todos os demais artefatos no *Twiki*.

Após a criação dos Métodos de cada Classe, podem-se inspecioná-los. Observa-se que o formato padrão para criação dos tópicos do tipo método é "%TOPIC%Metodo%MethodName%", ou seja, o Nome da Classe + o termo “Metodo” + o Nome do Método informado pelo usuário.

```
        <input type="hidden" name="Nome" value="%MethodName%" />
        <input type="hidden" name="Topico"
value="%TOPIC%Metodo%MethodName%" />
```

Deve-se ressaltar a diferença entre o campo "Nome" e o campo "Topico" do formulário associado. O campo "Nome" que contém a variável %MethodName%, armazena o nome do método que é informado pelo usuário no momento de sua criação. Já o campo "Topico" terá como conteúdo o Nome da Classe + "Metodo" + o Nome real do Método informado. Essa distinção é importante porque o nome do Tópico é mais complexo pois identifica a Classe relacionada e o tipo do artefato. Quando se cria um novo Método, informa-se apenas o seu nome. Esse recurso é utilizado para a criação dos demais artefatos (tópicos) que têm como regra esse padrão de nomenclatura.

Na fase de Código, os ponteiros:

CriaArquivo	permite definir os arquivos gerados e também a sua inspeção.
CriaModulo	possibilita que se definam os módulos do sistema e para cada um deles, que se realize a inspeção verificando diversas questões como por exemplo, a coesão e o acoplamento, as condições de laço, entre outras.
CriaObjeto	permite criar os objetos, inspecioná-los acerca questões como reuso e inicialização.
InspecaoCodigo	possibilita que se atente para questões relativas ao código nem sempre consideradas como a qualidade dos comentários e mensagens de alerta.

É também na Fase do Código que pode-se definir a equipe de inspeção dessa fase.

Para cada novo Módulo criado, é executado o tópico padrão InspecaoModulo. Realiza-se a inspeção do Módulo por meio dos *Checklists* de Projeto e de Código, conforme abaixo:

```
%EDITTABLE{header="| *Módulo* | *Preencha adequadamente* |"
format="| label, 20,  | text, 50,  |" changerows="off"}%
|*Módulo*|*Preencha adequadamente*|
| Nome do Módulo |
| Descrição do Módulo |

%EDITTABLE{header="| *Checklist de Módulo - Projeto* |
*Preencha adequadamente* |" format="| label, 20,  | text, 50,
|" changerows="off"}%

%EDITTABLE{header="| *Checklist de Módulo - Código* | *Preencha
adequadamente* |" format="| label, 20,  | text, 50,  |"
changerows="off"}%
```

As variáveis são criadas a partir da criação dos Módulos, ou seja, após a definição dos Módulos, definem-se as variáveis que cada Módulo referencia.

```
%EDITTABLE{header="| *Variável* | *Preencha adequadamente* |"
format="| label, 20,  | text, 50,  |" changerows="off"}%
|*Variável*|*Preencha adequadamente*|
| Nome da Variável: |  |


<form name="InspCodVar"
action="%SCRIPTURL{"save"}%/%WEB%/%TOPIC%Variavel%VariableName%
">
       <input type="hidden" name="templatetopic"
value="InspecaoCodigoVariavel" />
       <input type="hidden" name="formtemplate"
value="InspecaoForm" />
       <input type="hidden" name="Tipo" value="Variavel" />
       <input type="hidden" name="Nome" value="%VariableName%"
/>
       <input type="hidden" name="Topico"
value="%TOPIC%Variavel%VariableName%" />
       <input type="hidden" name="Pai" value="%TOPIC%" />
       <input type="hidden" name="topicparent"
value="%TOPIC%"/>
       <input type="submit" class="twikiSubmit"  value="Cria
Variável" />
</form>
```

No ambiente *Twiki*, não é necessário criar variáveis no mesmo momento da criação dos Módulos. Caso se deseje acessar um Módulo já criado, executa-se o próprio ponteiro CriaModulo, informa-se o nome do Módulo criado anteriormente e será apresentado o Módulo com a última informação salva. O ponteiro CriaModulo

deve ser usado não somente para a criação de novos Módulos, mas também acessar Módulos existentes.

Após a definição das diversas variáveis utilizadas em cada Módulo, podem-se inspecioná-las considerando diversas questões tais como: inicialização, compatibilidade com formato e tamanho, nomenclatura, cálculos, conversão.

Observa-se que o formato padrão para criação dos tópicos do tipo variável é "%TOPIC%Variavel%VariableName%", ou seja, o Nome do Módulo + o termo "Variavel" + o Nome da Variável informada pelo usuário.

Todo Objeto criado é derivado de alguma Classe, ou seja, é uma instância de uma determinada Classe.

O tópico CriaObjeto está detalhado abaixo:

```
<form name="CriacaoObjeto"
action="%SCRIPTURL{"save"}%/%WEB%/CriaNovoObjeto">
       <input type="hidden" name="templatetopic" value="InspecaoObjeto" />
       <input type="hidden" name="formtemplate" value="InspecaoForm"/>
       <input type="hidden" name="Tipo" value="Auxiliar"/>
       *Classe base:*
           <select name="Pai">
                %METASEARCH{
                     type="field"
                     name="Tipo" value="Classe"
                     default="<option $marker>Nenhuma classe definida
ainda</option>"
                    format="%$nopSEARCH{
                            \"$topic\"
                            scope=\"topic\"
                            topic=\"$topic\"
                            nonoise=\"on\"
                            format=\"<option
$marker>$formfield(Nome)</option>\"
                        }$nop%"
}%
           </select>
       NOME: <input type="text" name="Nome" size=16/>
       <input type="submit" class="twikiSubmit"  value="Cria Objeto" />
</form>
```

Conforme exposto acima, o comando select permite selecionar uma Classe existente como Classe base para o novo Objeto a ser criado. Após se selecionar a Classe base, deve-se informar o nome do novo Objeto. Adotou-se como convenção utilizar o nome da Classe como parte do nome do Objeto. Dessa forma, para a criação de Objetos, foi necessário utilizar um tópico auxilar CriaNovoObjeto. Com a utilização de formulário, definiu-se como o campo "Pai" a Classe base selecionada.

No tópico CriaNovoObjeto detalhado abaixo:

```
---+ Criação de objetos do tipo %META{"formfield" name="Pai"}%
(Pesquisa Objetos dessa Classe)
Objetos: %METASEARCH{
                 topic="%META{"formfield" name="Pai"}%"
                 type="parent"
                 dontrecurse="on"
                 format="%$nopSEARCH{
                         \"META:FIELD.*?value=\"Objeto\"\"
                         format=\"[[$formfield(Topico)][$formfield(Nome)]]\"
                         type=\"regex\"
                         scope=\"text\"
                         topic=\"$topic\"
                         nototal=\"on\"
                         nosearch=\"on\"
                         dontrecurse=\"on\"
                      }$nop%"
               }%
```

Ressalta-se a diferença entre os comandos “%Meta” e “%Metasearch”.

O comando:

%META{"formfield" name="Pai"}%

não realiza uma busca. Apenas verifica e exibe para o tópico corrente, o conteúdo do campo "Pai" do formulário.

Já o comando:

```
%METASEARCH{
                 topic="%META{"formfield" name="Pai"}%"

                         \"META:FIELD.*?value=\"Objeto\"\"
```

efetivamente realiza a busca entre todos os tópicos segundo o critério de pesquisa pelo tópico-pai igual ao campo "Pai" do formulário corrente e, dentre todos os tópicos encontrados, busca todos os tópicos que sejam do tipo "Objeto" (subpesquisa ou pesquisa aninhada).

Ainda no tópico CriaNovoObjeto:

```
<!--
   * Local ObjectName = %META{"formfield" name="Nome"}%
   * Local ClassName = %META{"formfield" name="Pai"}%
-->

<form name="CriacaoDeMaisUmObjeto"
action="%SCRIPTURL{"save"}%/%WEB%/CriaNovoObjeto">
       <input type="hidden" name="templatetopic"
value="InspecaoObjeto" />
       <input type="hidden" name="formtemplate"
value="InspecaoForm"/>
       <input type="hidden" name="Tipo" value="Auxiliar"/>
       <input type="hidden" name="Pai" value="%ClassName%"}%>
        NOME: <input type="text" name="Nome" size=16/>
       <input type="submit" class="twikiSubmit"
value="Seleciona" />
</form>
```

Podem-se criar vários Objetos da mesma Classe previamente selecionada ou podem-se criar Objetos de uma nova Classe.

O botão Seleciona permite definir Objetos adicionais a serem criados de uma Classe anteriormente selecionada por meio do tópico CriaObjeto.

```
<p>
Para criar %META{"formfield" name="Nome"}%:
<form name="InspObjeto"
action="%SCRIPTURL{"save"}%/%WEB%/%ClassName%Objeto%ObjectName%">
       <input type="hidden" name="templatetopic" value="InspecaoObjeto" />
       <input type="hidden" name="formtemplate" value="InspecaoForm" />
       <input type="hidden" name="Nome" value="%ObjectName%" />
       <input type="hidden" name="Pai" value="%ClassName%" />
       <input type="hidden" name="Tipo" value="Objeto" />
       <input type="hidden" name="Topico"
value="%ClassName%Objeto%ObjectName%" />
       <input type="hidden" name="topicparent" value="%ClassName%" />
       <input type="submit" class="twikiSubmit"  value="Cria Objeto" />
</form>
```

Continuando no mesmo tópico, o botão CriaObjeto é o que definitivamente irá criar os novos Objetos da mesma Classe selecionada.

O ponteiro CriaDocumentoInspecao localizado na tela principal apresenta uma lista de todos Documentos de inspeção já criados e um botão “Cria Documento Inspeção” que irá criar novos documentos. Esse documento poderá ser criado para cada artefato inspecionado e possibilita também o registro das reuniões de inspeção, dos problemas encontrados, de sua correção e solução. Podem-se também definir o Moderador, Inspetor, Autor, Leitor e Relator que compõem a

Equipe de inspeção de cada artefato registrado por meio desse ponteiro CriaDocumentoInspecao.

Deve-se ressaltar que os tópicos EquipeInspecaoAnalise, EquipeInspecaoProjeto e EquipeInspecaoCodigo permitem definir respectivamente, a equipe de inspeção da fase de Análise, Projeto e Código de um modo geral para o ciclo de vida do sistema. Para detalhar a equipe de inspeção para cada artefato inspecionado, deve-se utilizar o ponteiro CriaDocumentoInspecao.

Cada artefato de cada fase poderá ser criado executando cada ponteiro exibido. Em seguida, é apresentada uma tela listando todos os artefatos similares criados e também um botão que cria o artefato propriamente dito. A cada execução de cada botão que cria o artefato, será salva a informação no próprio ambiente interagido. Assim, tudo que é digitado como informação pertinente ao projeto será armazenado e poderá ser recuperado em qualquer outro momento futuro.

Para cada artefato criado, existe um processo de inspeção associado que não precisará ser preenchido no mesmo momento de sua criação. É possível posteriormente retornar e inspecionar cada artefato definido.

Como uma análise de cobertura, o tópico EstatísticasCobertura apresenta os tipos de Entidades que foram e as que ainda não foram inspecionadas, o que é importante para acompanhar o progresso de cada artefato criado.

Capítulo 6

6 Estudo de Caso

O estudo de caso analisado é um projeto utilizado em uma disciplina de graduação do Curso de Bacharelado em Ciência da Computação do IME-USP (disciplina MAC0242 – Laboratório de Programação II). Trata-se de um jogo de aventura do tipo texto, escrito em Java. O objetivo deste estudo foi fazer uma prova de conceito e exercitar as características implementadas no *Twiki,* além de verificar a consistência e viabilidade da proposta de inspeção sugerida.

Citam-se alguns requisitos do sistema:

- portabilidade;
- facilidade de geração de estórias;
- idioma português;
- gramática simplificada dos comandos de ação do jogador;
- possibilidade de se salvar a qualquer momento e recuperar o jogo no ponto em que se interrompeu a execução (serialização);
- possibilidade de se associar pontuação ao jogo.

A análise foi feita somente após a conclusão do projeto. O jogo foi desenvolvido ao longo do segundo semestre de 2007, em 4 fases e de modo incremental. Para o estudo, foi utilizada uma implementação padrão.

Na primeira fase foi definida a estrutura interna do programa e os vários tipos de objetos que fazem parte de sua implementação.

A segunda fase conteve o arcabouço básico do jogo. O mecanismo geral da aventura é relativamente simples, com uma inicialização, o disparo do jogo e um laço de iteração.

A terceira fase implementou o interpretador de comandos do usuário, segundo uma gramática própria do jogo. Ela também é responsável pela serialização e interface básica com o usuário.

A quarta e última fase tratou da interface gráfica e dos ajustes finais.

Como se pode notar, cada uma das fases faz uso do que foi desenvolvido nas fases anteriores. Assim, o preenchimento incremental das informações no *Twiki* documenta naturalmente o andamento do processo. Essa disponibilização de informações permite que problemas possam ser identificados e discutidos em tempo curto e hábil, pois uma vez publicadas, todos os membros do projeto têm acesso.

Todo o processo de inspeção foi simulado criteriosamente, observando-se todos os passos, verificando contingências e corrigindo eventuais imperfeições.

Foram selecionadas abaixo algumas telas do *Twiki* para uma melhor visualização do ambiente.

A seguinte tela, acessível pelo ponteiro <http://mairinque.ime.usp.br/cgi-bin/*Twiki*/view/Aventura/SistemaOrientadoObjeto>, é a principal e por meio da qual todos os outros tópicos derivam:

You are here: TWiki > Aventura Web > SistemaOrientadoObjeto

Nesta página, será possível realizar a documentação de todo o Sistema Orientado a Objetos. Além disso, será possível também realizar a inspeção do Sistema.

Dados do Projeto	Preencha adequadamente
Data de início do projeto	20/06/2007
Data prevista de término do projeto	30/11/2007
Descrição do Projeto	Jogo de aventura do tipo texto.

Equipe do Projeto	Preencha adequadamente
Gerente do Projeto	Marco Dimas Gubitoso
Patrocinador do Projeto	Universidade de São Paulo
Arquiteto do Sistema	Marco Dimas Gubitoso
Analista de Requisitos	Suzana Assato
Administrador de Dados	
Administrador de Banco de Dados	
Programador	alunos
Analista de Testes	monitor

Atenção: Execute o botão "Concluído" somente quando a criação da Equipe estiver finalizada!

Concluído

Voltar à Página Principal

-- SuzanaIpt - 07 Dec 2007

A seguir apresentamos uma proposta de Inspeção e meios para se criar documentos de Inspeção:

Guia para a Inspeção

Análise

Pode-se definir a equipe de Inspeção da fase da Análise.

Podem-se definir os requisitos e os casos de uso, bem como a inspeção desses artefatos.
FaseAnalise

Projeto

Pode-se definir a equipe de Inspeção da fase do Projeto.

Podem-se definir as interfaces, classes, as heranças, os métodos, bem como a inspeção desses artefatos.
FaseProjeto

Código

Pode-se definir a equipe de Inspeção da fase do Código.

Podem-se definir os arquivos, módulos, objetos, variáveis, bem como a inspeção desses artefatos.
FaseCodigo

Inspeção de Artefatos
CriaDocumentoInspecao

Análise de Cobertura da Inspeção
EstatisticasCobertura
-- SuzanaIpt - 05 Dec 2007
E dit | W YSIWYG | A ttach | P rintable | R aw View | Backlinks: We b , A l l Webs | H istory: r6 < r5 < r4 < r3 < r2 | M ore topic actions

Figura 5 – Tela principal proposta do *Twiki*

A próxima tela, acessível pelo ponteiro <http://mairinque.ime.usp.br/cgi-bin/*Twiki*/view/Aventura/Requisito00>, mostra um exemplo de criação de um requisito:

You are here: TWiki > Aventura Web > CriaRequisitos > Requisito00

Requisito	Preencha adequadamente
Nome do Requisito:	Facilidade de geração de estórias
Descrição do Requisito:	Deve ser fácil construir e alterar estórias, o usuário não precisa ter experiência avançada em programação

Checklist de Análise dos Requisitos	Preencha adequadamente
Data da Inspeção:	
Nome do(s) inspetor(es):	
Analise com cuidado termos persuasivos como: certamente, portanto, evidentemente, obviamente	
Atente para palavras como: algum, às vezes, freqüentemente, geralmente, maior parte das vezes	
Quando se refere a uma lista de itens inacabada, assegure-se que o significado da formação da lista tenha sido bem entendido. Atente para termos como: etc., assim por diante	
Atente para listas sem exemplos ou exemplos insuficientes para o seu completo entendimento	
Atente para verbos vagos como: processado, rejeitado, eliminado	
Cuidado com construções passivas pois elas não determinam quem é o ator que está realizando o trabalho	
Cuidado com o uso de pronomes que são claros para o autor, mas, não para o leitor	
Completo: verifique se todos os itens imprescindíveis para a solução do problema foram abrangidos	
Correto: analise se cada item especificado está correto	
Preciso, não-ambíguo e claro: identifique se cada item está legível, sem dupla interpretação ou ambiguidades	
Consistente: analise se não há idéias conflitantes	
Relevante: verifique se todas as informações são pertinentes ao problema	
Testável: é possível se testar cada item dos requisitos e verificar a	

Flexível: pode-se alterar itens da especificação sem que ocorra grande impacto nos outros itens?

Edit

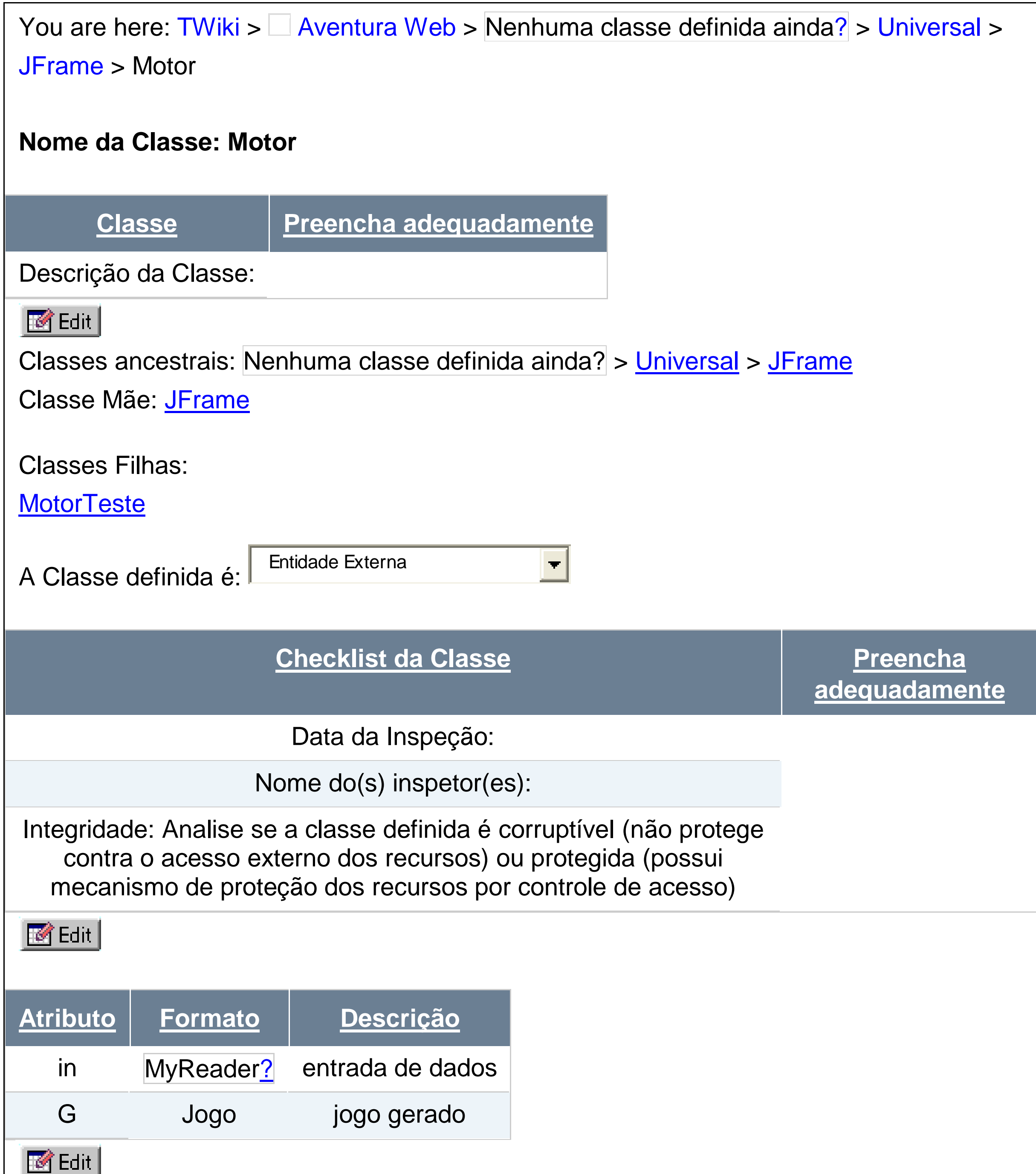
Criar novos Requisitos

Voltar para a Fase da Análise

Voltar à Página Principal

Figura 6 – Exemplo de criação de um requisito

A seguinte tela, acessível pelo ponteiro <http://mairinque.ime.usp.br/cgi-bin/*Twiki*/view/Aventura/Motor>, representa a criação da Classe Motor:



You are here: TWiki > Aventura Web > Nenhuma classe definida ainda? > Universal > JFrame > Motor

Nome da Classe: Motor

Classe	Preencha adequadamente
Descrição da Classe:	

Edit

Classes ancestrais: Nenhuma classe definida ainda? > Universal > JFrame

Classe Mãe: JFrame

Classes Filhas:

MotorTeste

A Classe definida é: Entidade Externa

Checklist da Classe	Preencha adequadamente
Data da Inspeção:	
Nome do(s) inspetor(es):	
Integridade: Analise se a classe definida é corruptível (não protege contra o acesso externo dos recursos) ou protegida (possui mecanismo de proteção dos recursos por controle de acesso)	

Edit

Atributo	Formato	Descrição
in	MyReader?	entrada de dados
G	Jogo	jogo gerado

Edit

Método	Preencha adequadamente
Nome do Método:	Status

Edit

Cria Método

Métodos: Status Titulo

Interfaces utilizadas

Edit

Criar novas Classes

Voltar para a Fase de Projeto

Voltar à Página Principal

Figura 7 – Exemplo de criação de uma classe

Capítulo 7

7 Conclusões

Foi apresentada uma estratégia de inspeção com propostas de perfis de equipe e de procedimentos, baseada em uma revisão detalhada da literatura bibliográfica. Como esse conceito é complementado com a atividade de teste, ela também foi apresentada como parte da pesquisa. A atividade de inspeção não exclui a execução das diferentes técnicas de testes; ao contrário, ambas são importantes para a garantia da qualidade do processo e do produto de desenvolvimento.

Embora essa estratégia não tenha sido testada em escala maior, nem foram coletados dados quantitativos, foi mostrado que é de fundamental importância se ter uma estratégia de inspeção para conduzir o processo de forma organizada, levando em consideração também os principais artefatos orientados por seus *checklists*.

Utilizou-se do modelo gráfico de entidades e relacionamentos (MER) para um maior esclarecimento e visualização da proposta. Nesse modelo, foram representados os principais artefatos existentes resultantes das diferentes fases do desenvolvimento e seus relacionamentos entre si. A partir dessa análise, construiu-se a arquitetura implementada no *Twiki*.

A ênfase foi Projetos Orientados a Objeto pois são os mais freqüentemente utilizados e para se delimitar o escopo dos projetos em referência, embora a idéia proposta possa ser utilizada e adaptada também para diferentes projetos. Os ambientes de desenvolvimento acadêmicos são os mais deficitários em ferramentas de acompanhamento, pois a própria natureza dos projetos é dinâmica, exigindo ambientes ágeis.

O ambiente *Twiki* proposto, apesar de preliminar, se mostra útil para acompanhamento de projetos e de comunicação entre os elementos das equipes. A utilização pelo usuário final é bastante simplificada pois a sua estrutura reflete os passos propostos e permite documentação detalhada. Este tipo de solução é eficiente e viável, com custo baixo e de fácil implementação.

O *Twiki*, apesar de ser bastante prático, ainda possui uma documentação muitas vezes confusa e insuficiente, o que acabou consumindo um tempo adicional

para a definição das soluções adotadas. No entanto, essa dificuldade acabou não sendo transferida para o usuário final, pois após muitas tentativas e muito empenho, foi possível realizar a implementação com sucesso, encapsulando os detalhes técnicos.

A análise de cobertura ainda é insuficiente, mas extensões são facilmente incorporadas pois pôde-se verificar a potencialidade do *Twiki*, se bem explorado. O intuito da aplicação do estudo de caso não foi de se aprofundar, mas apenas de demonstrar a viabilidade do ambiente. A ênfase foi dada para a implementação da proposta nesse ambiente e demonstração de sua estruturação e de suas capacidades.

Como trabalhos futuros, podem-se citar:

- explorar mais o ambiente *Twiki*, que oferece recursos poderosos de implementação;
- completar o arcabouço com o desenvolvimento de pacotes e detalhamento dos componentes;
- gerar relatórios mais completos;
- aperfeiçoar o mecanismo de autenticação;
- aprofundar a análise de cobertura;
- aplicar a proposta em diferentes projetos como estudo de caso e comparar analiticamente os resultados.

Pode-se concluir que a proposta de inspeção, acompanhada do ambiente *Twiki* implementado, é viável e apresenta potencial para uma utilização em escala maior.

Referências Bibliográficas

ACKERMAN, A.F.; BUCHWALD, L.S.; LEWSKI, F.H. **Software Inspections: An Effective Verification Process.** Institute for Zero-Defect Software, Qualitech e AT&T Bell Laboratories. IEEE Software, May, 1989.

ADRION, W.R.; BRANSTAD, M.A; CHERNIAVSKY, J.C. **Validation, Verification, and Testing of Computer Software.** ACM. Computing Surveys, Vol. 14, No. 2, June, 1982.

ALMEIDA, E.S. **Método de teste baseado na especificação de requisitos do software para Projetos orientados a objetos.** 93f. Dissertação (Mestrado em Engenharia de Software) – IPT – Instituto de Pesquisas Tecnológicas do Estado de São Paulo, 2001.

ANDERSSON, C. et al. **An Experimental Evaluation of Inspection and Testing for Detection of Design Faults.** Proceedings of the 2003 International Symposium on Empirical Software Engineering. IEEE, 2003.

ANDREWS, A. et al. **What do we know about Defect Detection Methods?** IEEE Software, 2006.

BASILI, V.R. et al. **Detecting Defects in Object Oriented Designs: Using Reading Techniques to Increase Software Quality.** ACM, 1999.

BERLING, T.; THELIN, T. **An Industrial Case Study of the Verification and Validation Activities.** Proceedings of the Ninth International Software Metrics Symposium (METRICS'03). IEEE, 2003.

BERTOLINO, A.; MARCHETTI, E.. **A Brief Essay on Software Testing.** Chapter of Software Engineering Volume 1: Development process Third Edition, IEEE Computer Society/Wiley Interscience,pp. 393-411, 2005. Disponível em: <http://dienst.isti.cnr.it/Dienst/Repository/2.0/Body/ercim.cnr.isti/2004-TR-36/pdf >. Acesso em: 11 abr. 2007.

BRYKCZYNSKI, B. **A Survey of Software Inspection Checklists.** ACM SIGSOFT. Software Engineering Notes vol. 24, No. 1, January 1999.

CARRINGTON, D. **Teaching Software Testing.** School of Information Technology. The University of Queensland. ACM, 1997.

CARVALHO, E.L. de. **Roteiro de avaliação de ferramenta de automação de teste.** 152f . Dissertação (Mestrado em Engenharia de Software) – IPT – Instituto de Pesquisas Tecnológicas do Estado de São Paulo, 2004.

CARVER, J.C. **The impact of background and experience on software inspections.** 2003. 331p. Dissertation (Doctor of Philosophy) - University of Maryland, College Park, 2003.

CHAAR, J.K.; HALLIDAY, M.J; BHANDARI, I.S. **In-Process Evaluation for Software Inspection and Test.** IEEE Transactions on Software Engineering, Vol. 19, No. 11, November, 1993.

CROSSMAN, T. D. **Some Experiences in the Use of Inspection Teams.** The Standard Bank of South Africa Limited. 15th Annual ACM Computer Personnel Research Conference, Aug. 1977.

DALAL, S.R; HORGAN, J.R.; KETTENRING, J.R. **Reliable Software and Communication: Software Quality, Reliability, and Safety.** Information Sciences and Technologies Research Laboratory, IEEE, 1993.

DENGER, C.; KOLB, R. **Testing and Inspecting Reusable Product Line Components: First Empirical Results.** Fraunhofer Institute for Experimental Software Engineering (IESE), ACM, 2006.

DOOLAN, E. P. **Experience with Fagan's Inspection Method.** Software – Practice and Experience, Vol. 22(2), 173-182, February 1992.

DUNSMORE, A.; ROPER, M.; WOOD, M. **Practical Code Inspection Techniques for Object-Oriented Systems: An Experimental Comparison.** IEEE Computer Society, IEEE Software, 2003.

FAGAN, M. **Design and code inspections to reduce errors in program development.** IBM Systems Journal 15, No. 3: 182-211 (Reprinted in IBM Systems Journal 38, No. 2: 259-287), 1976. Disponível em <http://www.research.ibm.com/journal/sj/153/ibmsj1503C.pdf>. Acesso em: 12.mar.2007.

FAGAN, M. **Reviews and inspections.** Sd&m Conference 2001, Software Pioneers, Springer 2002. Disponível em: <http://www.mfagan.com/software_pioneers.pdf>. Acesso em: 04.mai.2007.

GILB, T. **Software Inspection - Planning to Get the Most Out of Inspection**. Software Quality Professional, Vol. 2, No. 2, March 2000, pp. 7-19. Disponível em: <http://qualitypress.asq.org/pub/sqp/past/vol2_issue2/sqpv2i2gilb.pdf>. Acesso em: 12. mar.2007.

GILB, T. ; GRAHAM, D. **Software Inspection:** Addison-Wesley Publishing Company,1993.

GITTENS, M. et al. **An Empirical Evaluation of System and Regression Testing.** University of Western Ontario e IBM, 2002.

HARROLD, M.J. **Testing: A Roadmap.** Georgia Institute of Technology, ACM, 2000.

HEISER, J.E. **An Overview of Software Testing.** IEEE, 1997.

HOUSE, D.E..; NEWMAN, W.F. **Testing Large Software Products.** ACM SIGSOFT SOFTWARE ENGINEERING NOTES vol. 14 No. 2, April 1989.

KELLY, D.; SHEPARD, T. **Task-Directed Software Inspection Technique: An Experiment and Case Study.** Royal Military College of Canada. IBM CASCON, Toronto, Nov. 2000.

KELLY, D.; SHEPARD, T. **How to do Inspections When There is No Time.** Dept. of Electrical and Computer Engineering, Canada, IEEE, 2001.

KIT, E. **Software Testing in the Real World – improving the process**: Editora Addison-Wesley – acm press, 1995.

LAITENBERGER, O. **Studying the Effects of Code Inspection and Structural Testing on Software Quality.** Proc. 9th Int'l Symp. Software Reliability Eng., IEEE CS Press, 1998.

LAITENBERGER, O.; ATKINSON, C. **Generalizing Perspective-based Inspection to handle Object-Oriented Development Artifacts.** Fraunhofer Institute for Experimental Software Engineering, Germany, ACM, 1999.

LAITENBERGER, O. et al. **Software Inspections, Reviews & *Walkthrough*s.** ACM, 2002.

LI, E.Y. **Software Testing In A System Development Process: A Life Cycle Perspective.** Journal of Systems Management, 41 (8), August 1990, 23-31. Disponível em: <http://dogget.cob.calpoly.edu/Faculty/Management/eli/pdf/jsm-90.pdf>. Acesso em: 11 abr. 2007.

LI, Y.; WAHL, N.J. **An Overview of Regression Testing.** ACM SIGSOFT. Software Engineering Notes, vol. 24, No. 1, January 1999.

MASHAYEKHI, V. et al. **Distributed, Collaborative Software Inspection.** University of Minnesota at Minneapolis. IEEE, September 1993.

McGREGOR, J.D. **Testing a Software Product Line.** CMU/SEI-2001-TR-022. Carnegie Mellon Software Engineering Institute, December 2001.

MYERS, G.J. **An Controlled Experiment in Program Testing and Code *Walkthrough*s/Inspections.** IBM Systems Research Institute. Communications of ACM, volume 21, number 9, September 1978.

OSTERWEIL, L. et al. **Strategic Directions in Software Quality.** ACM Computing Surveys, Vol. 28, No. 4, December 1996.

PAN, J. **Software Testing.** Carnegie Mellon University, 1999. Disponível em: <http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/>. Acesso em: 22 nov. 2006.

PARVEEN, T.; TILLEY, S.; GONZALEZ,G. **A Case Study in Test Management.** North Carolina, USA, ACM March, 2007.

PRESSMAN, R.S. **Engenharia de Software:** Editora Mc Graw Hill, 1995.

PRESSMAN, R.S. **Engenharia de Software 5ª. edição:** Editora Mc Graw Hill, 2002.

RAMACHANDRAN, M. **Requirements-Driven Software Test: A Process-Oriented Approach.** ACM SIGSOFT. Software Engineering Notes, vol. 21, No. 4, July 1996.

RYSER, J.; BERNER, S.; GLINZ, M. **On the State of the Art in Requirements-based Validation and Test of Software.** Department of Computer Science of the University of Zurich, Switzerland, Nov. 1998. Disponível em: <http://historical.ncstrl.org/litesite-data/unizh_ifi/ifi-98.12.pdf >. Acesso em: 11 abr. 2007.

SCHACH, S.R. **Testing: Principles and Practice.** ACM Computing Surveys, Vol. 28, No. 1, March 1996.

SCHNEIDER, G.M.; MARTIN, J.; TSAI, W.T. **An Experimental Study of Fault Detection In User Requirements Documents.** ACM Transactions on Sofware Engineering and Methodology, Vol. 1, No. 2, April 1992, 188-204.

SWEBOK. **Guide to the Software Engineering Body of Knowledge - Trial Version 1.00**. A Project of the Software Engineering Coordinating Committee. May 2001.

TERVONEN, I. **Support for Quality-Based Design and Inspection.** University of Oulu. IEEE Software, January, 1996.

VOTTA JR., L.G.. **Does Every Inspection Need a Meeting?** AT&T Bell Laboratories, 1993.

WALLACE, D.R.; FUJII, R.U. **Software Verification and Validation: An Overview.** IEEE Software, 1989.

WELLER, E.F. **Lessons from Three Years of Inspection Data.** IEEE, September, 1993.

WHITTAKER, J.A. **What Is Software Testing? And Why Is It So Hard?** Florida Institute of Technology. IEEE Software. January/February 2000.

WINKLER, D.; RIEDL, B.; BIFFL, S. **Improvement of Design Specifications with Inspection and Testing.** Proceedings of the 2005 31st EUROMICRO Conference on Software Engineering and Advanced Applications. IEEE, 2005.